Debugging Large Scale Applications in a Virtualized Environment

Filippo Gioachin, Gengbin Zheng and Laxmikant V. Kalé
gioachin@ieee.org, gzheng@illinois.edu, kale@illinois.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract. With the advent of petascale machines with hundreds of thousands of processors, debugging parallel applications is becoming an increasing challenge. Aside from the complicated debugging techniques required to debug applications at such scale, it is often difficult to gain access to these machines for a sufficient period of time, if at all. Some existing parallel debuggers are capable of handling these machines, but they still require the whole machine to be allocated. In this paper, we present an innovative approach to address debugging on such extreme scales. By leveraging the concept of object-based processor virtualization, our technique enables debugging of even a million processor execution under a simulated environment using only a relatively small cluster. We describe the obstacles we overcame to achieve this goal within two message passing programming models: CHARM++ and MPI. We demonstrate the results using real world applications such as Molecular Dynamics and Cosmological simulation programs.

1 Introduction

Debugging a parallel application requires numerous iterative steps. Initially, the application is tested on simple benchmarks on a few processors. At this point, many errors due to the communication exchanges between the processes in the parallel scenarios can be captured. Later, during production runs, the application will be deployed with larger input datasets, and on much bigger configurations. Oftentimes, the application will not behave as expected, and terminate abnormally. When this happens, the programmer is left to hunt the problem at the scale where it manifests, with possibly thousands of processors involved. If lucky, he may be able to recreate the problem on a smaller scale and debug it on a local cluster, but this is not always possible.

One example of a bug that may not be reproduced on a smaller scale is when the bug is located in an algorithm, and this algorithm depends on how the input data is partitioned among the processors. Reducing the problem size might be a solution to scale down the problem, but the inherent physics of the problem may not allow that. Another example is when the physics simulation output is incorrect. In this case, the problem can derive from rare conditions that only big datasets expose. Again, the problem size may not be reduced since otherwise the bug disappears. In all these examples, the only alternative left to the programmer is to use the whole machine, and debug with the full problem size on possibly thousands of processors.

Interactive sessions on large parallel machines are usually restricted to small allocations. For large allocations, batch scheduling is often required. To debug the application, the programmer will have to launch the job through the scheduler and be in front of the terminal when the job starts. Unless a specific allocation slot is pre-requested, this can happen at unpredictable, inconvenient times. Furthermore, the nature of debugging is such that it may require multiple executions of the code to track the bug, and to try different potential solutions. This exacerbates the problem and leads to highly inefficient debugging experience.

Moreover, debugging sessions on large number of processors are likely to consume a lot of allocation time on supercomputers, and significantly waste precious computation time. During an interactive debugging session, the programmer usually lets the program execute for some time and then pauses it to inspect its data structures, then iteratively advances it step-by-step, while monitoring some data of interest. Therefore, processors are idle most of the times waiting for the user to make decision on what to do next, which is a very inefficient use of supercomputers.

The innovative approach we describe in this paper is to enable programmers to perform the interactive debugging of his application at full scale on a simulated target machine using much smaller clusters. We do this by making each processor in the application a *virtual processor*, and mapping multiple virtual processors to a single physical processor. This reduces the processor count needed for debugging. This mapping is transparent to the application, and only the underlying runtime system needs to be aware of the virtualization layer. A parallel debugger connected to the running application presents to the programmer the application running on thousands of processors, while hiding the fact that maybe only a few dozen were actually used.

Our idea transcends the programming model used for the virtualization and how the debugging infrastructure is implemented. However, to prove the feasibility of this approach, we implemented it within the CHARM++ runtime system [1, 2], using the BigSim emulation environment, and the CHARMDEBUG debugger. Thus applications written in CHARM++ will be the main target for our debugging examples. MPI applications are supported via a virtualized MPI implementation called AMPI [3].

In the remainder of this paper, we start by describing the infrastructure of the debugger, CHARMDEBUG, in Section 2 and BigSim Emulator in Section 3. We present the object-based virtualization approach we adopted to integrate the two systems into a virtualized debugger in Section 4. Section 5 further describes how we applied this method in the context of debugging MPI applications. Sections 6 and 7 analyze our system in terms of overhead and functionality with some examples. Related work is described in Section 8 followed by some comments on future work in the concluding section.

2 CharmDebug

CHARMDEBUG [4] is a graphical debugger designed for CHARM++ applications. It consists of two parts: a GUI with which a programmer interacts, and a plugin inside the CHARM++ runtime itself. The GUI is the main instrument that a programmer will see when debugging his application. It is written in Java, and is therefore portable to all operating systems. A typical debugging session is shown in Figure 1. The user will start the CHARMDEBUG GUI on his own workstation. He can then choose to start a new application to debug, or attach to a running application manually, using the appropriate commands available in the GUI. By default, every CHARM++ application contains a CHARMDEBUG plugin inside. This plugin is responsible to collect information from the running application, and to communicate with the CHARMDEBUG GUI. With this plugin integrated in the application itself, no external tool is necessary on every compute node. Thanks to the tight coupling between these two components of CHARMDEBUG, the user can visualize several kinds of information regarding his application. Such information includes, but is not limited to, the CHARM++ objects present on any processor and the state of any such objects, the messages queued in the system, and the memory distribution on any processor.

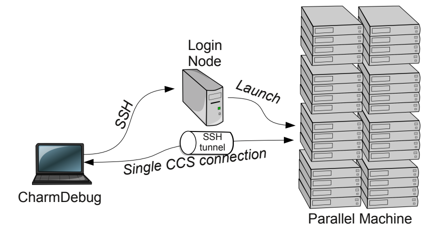


Fig. 1. Diagram of CHARMDEBUG's system.

The communication between the CHARMDEBUG GUI and the CHARMDEBUG plugin happens through a high-level communication protocol called Converse Client-Server, or CCS [4]. This protocol has become a standard for CHARM++, and is built into all CHARM++ applications. It can be used both by the user directly into his own application, for example to enable live streaming of images to remote clients, or internally by the system, as in this case by CHARMDEBUG to collect status information. The CCS server, which is the parallel application in the case of CHARMDEBUG, opens a single socket connection and listens to it for incoming connections. Later, the CHARMDEBUG CCS client initiates the communication by connecting to this socket and sending a request. This request is translated by the server (i.e. the application), into a CHARM++ level message which is then delivered to a pre-registered routine. This routine can perform any operation it deems necessary, including operations involving parallel computations. This leverages the message-driven scheduler running on each processor in CHARM++: in addition to dealing with application messages, the scheduler also naturally handles messages meant for debugging handlers. Finally, the server can return an answer to the waiting client, if appropriate. Note that since only one single connection is needed between the debugger and the application under examination, we avoid the scalability bottleneck of having the debugger connect directly to each process of the parallel application. This allows CHARMDEBUG to scale to as large a configuration as CHARM++ does.

In CHARM++, every parallel application is integrated with debugging support in the form of a CHARMDEBUG plugin. When a program starts, this plugin registers inspection functions that the CHARMDEBUG GUI will send requests to. This initialization happens by default during CHARM++'s startup without the user intervention. Therefore, any program is predisposed for analysis with CHARMDEBUG. Although lacking direct connection to each processor, the user can request the debugger to open a GDB [5] session for any particular processor. This gives the user flexibility to descend to a lower level and perform operations that are currently not directly supported by CHARMDEBUG.

3 Bigsim Emulator

Although CHARMDEBUG as described in the previous section is implemented to be scalable and efficient for debugging very large scale applications, in practice, its usefulness is greatly impaired by the constraint of large amount of resource needed for debugging. This motivated the work in this paper to exploit a virtualized environment called BigSim to reduce the need for the whole machine.

BigSim [6, 7] is a simulation framework that provides fast and accurate performance evaluation of current and future large parallel systems using much smaller machines, while supporting different levels of fidelity. It targets petascale systems composed of hundreds of thousands of multi-core nodes. BigSim consists of two components. The first component is a parallel *emulator* that provides a virtualized execution environment for parallel applications. This emulator generates a set of event logs during execution. The second component is a post-mortem trace-driven parallel *simulator* that predicts parallel performance using the event logs as input, and supports multiple resolutions for prediction of sequential and network performance. For example, the simulator can (optionally) predict communication performance accurately by simulating packets of each message flowing through the switches in detail, using a parallel discrete event simulation technique. Since the simulator only considers the trace logs and does not re-execute the application at the code level, it is not suitable for debugging purpose. However, the BigSim Emulator, which supports emulation of a very large application using only a fraction of the target machine, is useful for debugging. In the remainder of this section, we shall focus our attention on the emulator component.

Since multiple target processors are emulated on one physical processor, the memory usage on a given physical processor may increase dramatically. It may thus become impossible to fit the whole application into the physical memory available. Interestingly, our studies show that many real world scientific and engineering applications, such as molecular dynamics simulation, do not require a large amount of memory. For example, in one experiment, we were able to emulate NAMD [8] running on a 262,144-core Blue Waters machine [9] using just 512 nodes of the Ranger cluster, a Sun Constellation Linux Cluster at the Texas Advanced Computing Center (TACC).

For applications with large memory footprint, the physical amount of memory available per processor indeed poses a constraint. However, even in this scenario, we can still emulate these applications by using an efficient out-of-core technique [10, 11] optimized for the BigSim Emulator. Clearly, out-of-core execution, even with optimization, incurs a much higher overhead than the pure in-memory execution, mainly due to the constraint imposed by disk I/O bandwidth. For example, we observed a slowdown of about 18 times in terms of the total execution time of a jacobi application in [10].

For interactive debugging, the degraded performance due to the out-of-core execution may impact the user experience with slow responsiveness especially when the user requests involve all the virtual processors on disk. Increasing the number of emulating processors, and hence memory, helps reducing the need for extensive disk I/O in the out-of-core execution. Even though inefficient, this is a viable debugging solution when there is no other workaround.

4 Debugging CHARM++ Applications on BigSim

In order to combine the BigSim emulation system with CHARMDEBUG debugging framework, several new problems had to be solved. Most arose from the fact that CHARMDEBUG needs to deal with the virtualized CHARM++ and other virtualized layers in the emulation environment.

Normally, CHARM++ is implemented directly on top of CONVERSE, which is responsible for low-level machine-dependent capabilities such as messaging, user-level threads, in addition to message-driven scheduling. This is shown on the left branch of Figure 2. When CHARM++ is re-targeted to the BigSim Emulator, there are multiple target CHARM++ virtual processors running on one physical processor, as explained in the previous section. Therefore, all layers underneath CHARM++ must be virtualized. This new software stack is shown in the same Figure 2, on the right branch. Specifically, the virtualized CONVERSE layer becomes BigSim CONVERSE, which is the CONVERSE system implemented using the BigSim Emulator as communication infrastructure. This is equivalent to treating the BigSim Emulator as a communication sub-system.

4.1 Communicating with virtual processors

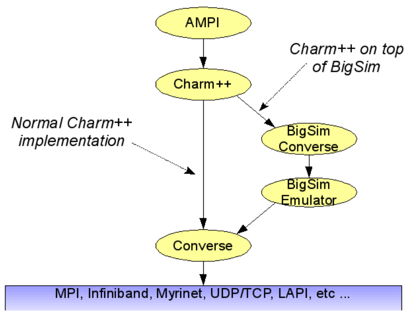

AMPI
Charm++ on top of BigSim
Charm++
Normal Charm++ implementation
BigSim Converse
BigSim Emulator
Converse
MPI, Infiniband, Myrinet, UDP/TCP, LAPI, etc ...

Fig. 2. BigSim Charm++ Software Stack.

One problem we had to overcome was the integration of the CCS framework into BigSim. CCS connects CHARMDEBUG and a running application in the context of CHARM++ processors. However, in the BigSim Emulation environment, CCS is unaware of the emulated target processors because it is implemented directly on CONVERSE. Therefore, it needs to be adapted to the emulation system so that the CHARMDEBUG client can connect to the emulated virtual processors. To achieve this, we created a middle layer for CCS (virtualized CCS) so that messages can reach the destination virtual processor. The target of a CCS message becomes now the rank in the virtual processor space. Figure 3 depicts the new control flow.

When a CCS request message is sent from CHARMDEBUG to a virtual processor, the message first reaches the CCS host (1). From here, it is routed to the real processor where the destination virtual processor resides (2). The processor level scheduler in CONVERSE will pick up the request message, but not execute the message immediately. Instead, it enqueues the message to the corresponding virtual processor, and activates it (3). The scheduler on the virtual processor will serve the CCS request by invoking the function associated with the request message (4), and return a response message. Notice that the response does not need intervention from CONVERSE since the virtual processor has direct access to the data structures stored in the common address space. Multicast and broadcast requests are treated in the virtualized environment. While this can add some overhead to the execution of a CCS request, it greatly simplifies the system, and the code reuse between the emulated and non-emulated mode.

Some CCS request messages are not bound to any specific virtual processor. For example, CHARMDEBUG may send CCS requests to physical processors to query processor-wide information such as those related to the system architecture or the memory system. However, since all virtual processors on the same physical processor have access to the processor information including the whole memory, any of these virtual processors can, in fact, serve the CCS requests. Therefore, our approach is to have CHARMDEBUG client always send such CCS requests to a virtual processor on a physical processor. This approach greatly simplifies the design and implementation of the CCS protocol, since we eliminate the need of having to specify if the request needs to be treated at the physical processor level, or at the virtual processor level.

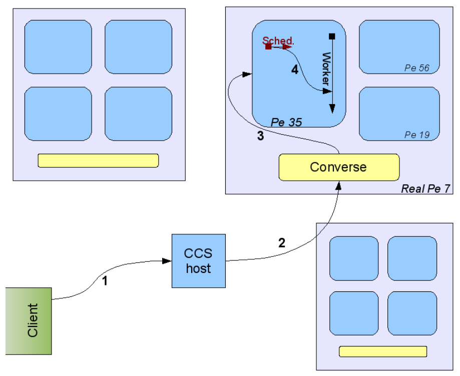


Fig. 3. Diagram of CCS scheme under BigSim Emulation.

4.2 Suspending virtual processors

Another challenge was to figure out how to suspend the execution of a single virtual processor. Notice that while a processor is suspended, we still want to deliver messages to it. For example, requests from the debugger should be honored regardless of the processor's state. At the same time, we do not want other virtual processors emulated inside the same physical processor to be affected. In the non-virtualized environment, the technique we use to suspend a processor is to enter a special scheduler when the processor needs to be suspended. In this mode, regular messages are placed into a queue, and buffered in FIFO order until the processor can deliver them. This scheduler is also in charge of driving the network, and receiving incoming messages. In this way, commands from the debugger can still be executed. In the virtualized environment, the scheduler that drives the network and forwards messages to the virtual processes is a separate entity from the scheduler inside each virtual processor. In this case, it is not possible to have each virtual processor driving the network, which will be too chaotic.

We modified our scheme to move the buffering of messages inside each individual virtual processor. When a worker processor needs to suspend due to an explicit debugger "freeze" command or due to a breakpoint, it calls its own scheduler recursively. Since this scheduler is stateless, such a recursive scheme is feasible. This new scheduler then starts the buffering of messages. When the processor is released by the debugger, and is allowed to continue its normal execution, we terminate the internal scheduler, and return control to the outer one. Buffered messages are guaranteed to be executed in the same order as they were received while we exit from the internal scheduler. Meanwhile, the main CONVERSE scheduler remains the only one that drives the network and receives messages. Moreover, the CONVERSE scheduler is always active, and never enters a buffering mode.

With the techniques described, we can now debug applications in the virtualized environment as if they were running on a real machine. We shall see an example of using CHARMDEBUG on a real application in section 7. In the future work section, we will outline other topics we plan to address.

5 Debugging MPI Applications on BigSim

Debugging a large scale MPI application on a smaller machine requires running multiple MPI "processes" on one processor. This can be done using existing MPI implementations, if allowed by the operating system. However, this is often infeasible for various reasons. First, operating systems often impose hard limits on the total number of processes allowed by a user on one processor, making it challenging to debug a very large scale application. Secondly, processes are heavy-weight in terms of creation and context switching. Finally, there are very few MPI implementations that support out-of-core execution which is needed for running applications with large memory footprints.

To overcome these challenges, we adopted the same idea of processor virtualization used in CHARM++: each MPI processor is now a virtual processor implemented as a *light-weight* CONVERSE user-level thread. This leads to Adaptive MPI, or AMPI [3], an implementation of the MPI standard on top of CHARM++. As illustrated in Figure 4, each physical processor can host a number of MPI virtual processors (or AMPI threads). These AMPI threads communicate via the underlying CHARM++ and CONVERSE layers. This implementation also takes advantage of CHARM++'s out-of-core execution capability. Since AMPI is a multithreaded implementation of the MPI standard, global variables in MPI applications may be an issue. AMPI provides a few solutions to automatically handle global variables [12] to ensure that an MPI application compiled against AMPI libraries runs correctly.

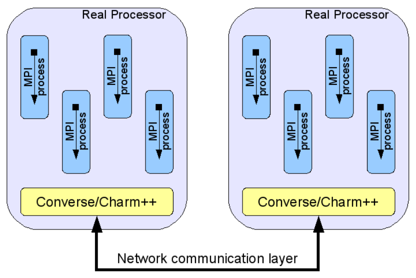
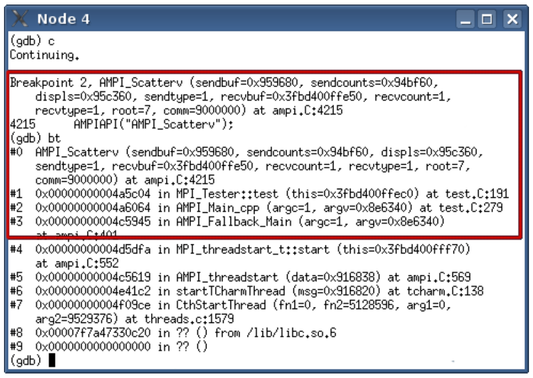


Fig. 4. AMPI virtualization using CHARM++.

Debugging MPI applications can now use any arbitary number of physical processors. For example, when debugging Rocstar [13], a rocket simulation program in MPI developed by the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois, a developer was faced with an error in mesh motion that only appeared when a particular problem was partitioned for 480 processors. Therefore, he needed to run the application on a large cluster at a supercomputer center to find and fix the bug. However, the turn-around time for a 480 processor batch job was fairly long since the batch queue was quite busy at that time, which made the debugging process painfully slow. Using AMPI, the developer was able to debug the program interactively, using 480 virtual processors distributed over 32 physical processors of a *local* cluster, where he could easily make as many runs as he wanted to resolve the bug.



```
(gdb) c
Continuing.

Breakpoint 2, AMPI_Scatterv (sendbuf=0x959680, sendcounts=0x94bf60,
    displs=0x95c360, sendtype=1, recvbuf=0x3fbd400ffe50, recvcount=1,
    recvtype=1, root=7, comm=9000000) at ampi.C:4215
4215        AMPIAPI("AMPI_Scatterv");
(gdb) bt
#0  AMPI_Scatterv (sendbuf=0x959680, sendcounts=0x94bf60, displs=0x95c360,
    sendtype=1, recvbuf=0x3fbd400ffe50, recvcount=1, recvtype=1, root=7,
    comm=9000000) at ampi.C:4215
#1  0x00000000004a5c04 in MPI_Tester::test (this=0x3fbd400ffec0) at test.C:191
#2  0x00000000004a6064 in AMPI_Main_cpp (argc=1, argv=0x8e6340) at test.C:279
#3  0x00000000004c5945 in AMPI_Fallback_Main (argc=1, argv=0x8e6340)
    at ampi.C:401
#4  0x00000000004d5dfa in MPI_threadstart_t::start (this=0x3fbd400fff70)
    at ampi.C:552
#5  0x00000000004c5619 in AMPI_threadstart (data=0x916838) at ampi.C:569
#6  0x00000000004e41c2 in startTCharmThread (msg=0x916820) at tcharm.C:138
#7  0x00000000004f09ce in CthStartThread (fn1=0, fn2=5128596, arg1=0,
    arg2=9529376) at threads.c:1579
#8  0x00007f7a47330c20 in ?? () from /lib/libc.so.6
#9  0x0000000000000000 in ?? ()
(gdb) █
```

Fig. 5. Screenshot of GDB attached to a specific MPI rank, and displaying its stack trace.

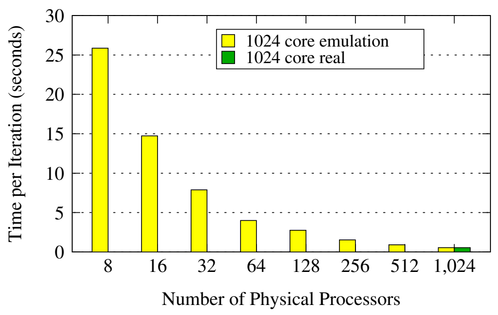
Since AMPI is implemented on top of CHARM++, the basic techniques for debugging as decribed in Section 4 work on AMPI programs automatically. In addition, if the user desires to perform more in-depth analysis on a specific MPI rank, he can choose to start a GDB sequential debugger attached to that rank. This GDB process is shown in Figure 5 for a simple test program. In this example, the user has set a breakpoint on MPI_Scatterv function, and when the breakpoint was hit, he printed the stack trace.

6 Debugging Overhead in the Virtualized Environment

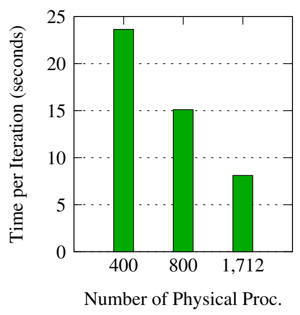
In this section, we study the debugging overhead using a synthetic Jacobi benchmark and a real application NAMD, running on the modified BigSim emulator with CHARMDEBUG support.

Our test environment is Blue Print, a Blue Waters interim system at National Center for Supercomputing Applications (NCSA). It is a 2,000-core IBM Power 5+ system. There are 107 compute nodes actually available for running a job, and each node has 16 cores (i.e. 1712 cores total).

We first tested a Jacobi3D program written in CHARM++ on 1024 virtual processors on a varying number of physical processors with CHARMDEBUG enabled, and measured the execution time per step. Figure 6(a) shows the results of the execution time with varying number of physical processors, from 8 to 1024. The last bar in the figure is the actual execution time of the same code on the 1024 processors with normal CHARM++. We can see that by using exactly same number of processors, Jacobi under BigSim runs as fast as the actual execution in normal CHARM++, showing almost no overhead of the virtualization in BigSim CHARM++. When we use fewer physical processors to run the same Jacobi emulation on 1024 virtual processors, the total execution time increases as expected. However, the increase in the execution time is a little less than the time proportional to the loss of processors. For example, when using 1024 physical processors, the execution time is 0.25s, while it takes only 23.96s when using only 8 physical processors. That is about 92 times slower (using 128 times fewer processors). This is largely due to the fact that most communication becomes in-node communication when using fewer number of nodes.



(a) 1024 emulated processors. The last bar is the actual runtime on 1024 processors.



(b) 1M (1,048,576) emulated processors.

Fig. 6. Jacobi3D execution time with varying number of physical processors.

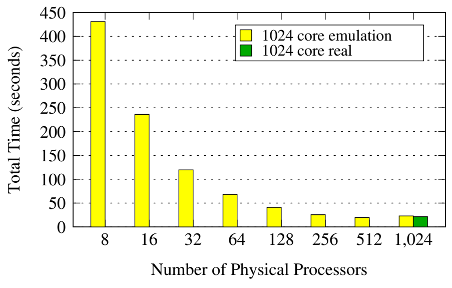
As a stress test, we ran the same Jacobi3D program on one million (1,048,576) emulated processors, while trying to use as fewer number of physical processors as possible. Figure 6(b) shows the execution time when running on 400, 800, and 1712 physical processors. These experiments show that it is feasible to debug an application in a virtualized environment for very large number of target processors using a much smaller machine.

To test how much time typical operations take from the debugger point of view, we used a similar Jacobi3D program, this time written in MPI. Table 1 reports timings for starting the MPI application, loading the list of messages queued on a given processor, and perform a step operation (deliver a single message) on all virtual processors. The latter two operations perform in an almost identical amount of time in all scenarios, including the case when the application is run in the non-virtualized environment.

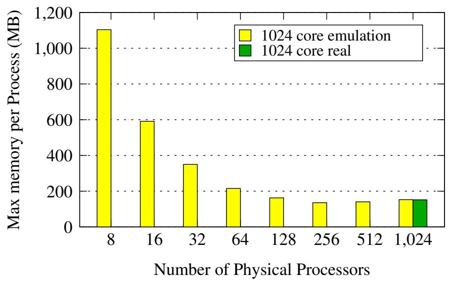
We also studied the BigSim overhead on a real application. NAMD [14, 8] is a scalable parallel application for Molecular Dynamics simulations written using the CHARM++ programming model. It is used for the simulation of biomolecules to understand their structure. In these experiments, we ran NAMD on a 1024 emulated processors with Apolipoprotein-A1 (ApoA1) benchmark for 100 timesteps. We measured the total execution time of each run (including startup and I/O) using a varying number of physical processors, from 8 to 1024. This is illustrated in Figure 7(a). Same as for Jacobi, we ran NAMD also in non-emulated mode using 1024 physical processors. The total execution time is shown in the last bar of the figure. We can see that NAMD running on the BigSim Emulator is only marginally slower (by 6%) compared to the normal execution on 1024 physical processors, showing little overhead of the emulator. On 512 processors, however, NAMD running in the emulation mode is even slightly faster than the actual run on 1024 processors. This is due to savings in the NAMD initial computation phases: faster global synchronization on fewer nodes.

	8	16	32	64	128	256	512	1024	original
Startup (seconds)	11.60	11.63	13.34	13.12	15.86	14.41	16.45	17.71	17.85
Load a message queue (ms)	398	399	399	400	400	399	399	379	379
Single step, all pe (ms)	131	99	213	66	41	118	67	118	114

Table 1. Time taken by the CHARMDEBUG debugger to perform typical operations, using MPI Jacobi3D application with 1024 emulated processors on varying number of physical processors.



(a) Execution Time



(b) Memory Usage

Fig. 7. NAMD on emulated 1024 processors using varying number of physical processors. The last bar is the actual run on 1024 processors.

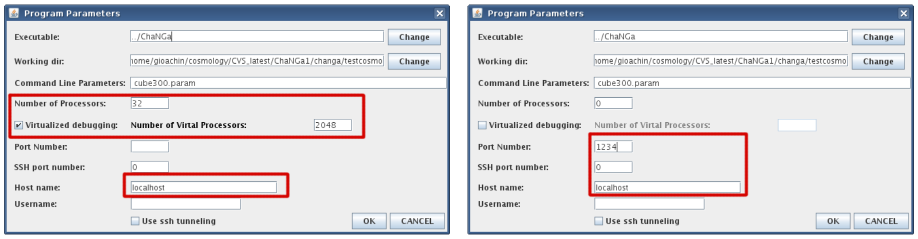
Overall, this demonstrates that in terms of the time cost, debugging in a virtualized environment using much smaller number of processors is possible. Although it takes a little longer time (19 times slower from 1024 to 8 processors) to run the application, debugging on a much smaller machine under a realistic scenario is not only easily accessible and convenient, but also simpler for setting up debugging sessions.

We further studied the memory overhead under the virtualized environment. Using the same NAMD benchmark on 1024 virtual processors, we gathered memory usage information for each processor. Figure 7(b) shows the peak memory usages across all physical processors. Again, the last bar is with the non-emulated CHARM++. Note that in emulation mode, the total memory usage is the sum of the application's memory usage across all emulated processors, plus the memory used by the emulator itself. It can be seen that there is no difference in memory usage between the emulation mode and non-emulation mode when using 1024 physical processors. When the number of processors decreases to 512, or even 256, the memory usage remains about the same. This is because NAMD has some constant memory consumption that dominates the memory usage (for example, read-only global data such as molecule database, which is replicated on each node), and the emulator itself tends to use less memory when the number of processors decreases. However, when the number of physical processors keeps reducing, each physical processor hosts a much larger number of emulated virtual processors whose memory usage starts to dominate, therefore the total memory usage increases significantly. Nevertheless, when the number of physical processors is down to 8, the peak memory usage reaches about 1GB, which is still very feasible on machines nowadays. Note that this is an increase of only about 7 fold compared to the 1024 processor case, due to the sharing of the global read-only data at the process level.

In summary, we have demonstrated that debugging under virtualized environment incurs reasonably low overhead, considering the overhead proportional to the loss of processors. This makes it feasible to debug applications running on a large machine using only a portion of it.

7 Case Study

To demonstrate the capabilities of our technique, we used a few examples of complex applications, and debugged them in the virtualized environment. It is not the purpose



Executable: ../ChaNGa
Working dir: ıome/gioachin/cosmology/CVS_latest/ChaNGa1/changa/testcosmo
Command Line Parameters: cube300.param
Number of Processors: 32
[✓] Virtualized debugging: Number of Virtal Processors: 2048
Port Number:
SSH port number: 0
Host name: localhost
Username:
☐ Use ssh tunneling

(a) Launching scenario

Executable: ../ChaNGa
Working dir: ıome/gioachin/cosmology/CVS_latest/ChaNGa1/changa/testcosmo
Command Line Parameters: cube300.param
Number of Processors: 0
☐ Virtualized debugging: Number of Virtal Processors:
Port Number: 1234
SSH port number: 0
Host name: localhost
Username:
☐ Use ssh tunneling

(b) Attach scenario

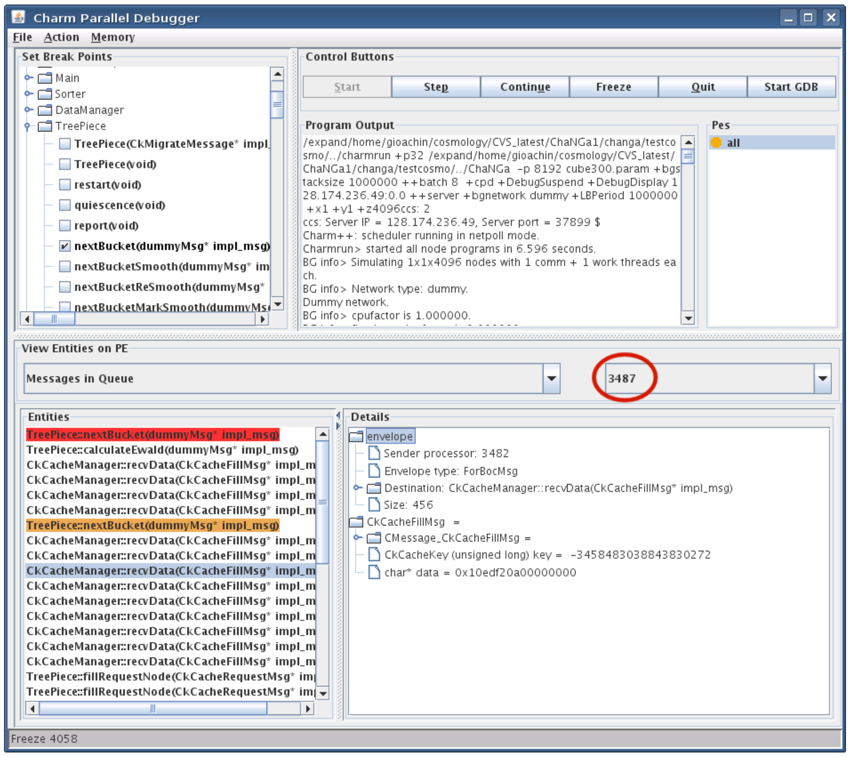
Fig. 8. Screenshots of CharmDebug parameter window.

of this section to describe actual bugs that were found with this technique, but rather illustrate how the user has available all the tools that he has in a normal scenario. With those tools, the user can search for the bug as he seems fit. Some applications have been described in section 6 while considering the overhead our technique imposes to the application under debugging. In this section, we use another real world application as an example.

ChaNGa [15] is a production code for the simulation of cosmological evolution, currently in its second release. It is capable of computing standard gravitational and hydrodynamic forces using Barnes-Hut and SPH approaches respectively. This application is natively written in Charm++, and it uses most of the language abstractions provided by the runtime system. While most of the computation is performed by Charm++ *array elements*, which are not bound to the number of processors involved in the simulation, the application also uses Charm++ *groups* and *nodegroups* for performance reasons. These groups have the characteristic of having one entity per processor, thus modifying the application behavior when scaling to larger number of processors. The complexity of this application is one reason why we chose it over other examples.

After the user has built the Charm++ runtime system with support for BigSim emulation, and compiled the ChaNGa program over the virtualized Charm++, he can start CharmDebug's GUI. Figure 8(a) shows the dialogue box for the application parameters. In here, the user will indicate the location of his executable, the arguments, and the number of processors he wants to run on. The only difference with a standard non-virtualized execution is the presence of a checkbox to enable the virtualization. In general, the user will input the number of desired processors in the "Number of Processors" textfield and confirm. In this case, "Number of Processors" refers to the number of physical processors CharmDebug will allocate on the machine. The number of processors the user wants to debug on has to be specified in the field named "Number of Virtual Processors". These fields are highlighted in the Figure. At this point the user can confirm the parameters, and start the execution of the program from CharmDebug's main view.

If the machine to be used for debugging requires jobs to be submitted through a batch scheduler (or if the user desires to start the application himself), only the fields regarding executable location and CCS host/port connection need to be specified. These are highlighted in Figure 8(b). When the attach command is issued from the main view, the CharmDebug plugin will automatically detect the number of processors in the simulation, and if the execution is happening in the virtualized environment.



```
Main
Sorter
DataManager
TreePiece
    TreePiece(CkMigrateMessage* impl_
    TreePiece(void)
    restart(void)
    quiescence(void)
    report(void)
    ✓ nextBucket(dummyMsg* impl_msg)
    nextBucketSmooth(dummyMsg* im
    nextBucketReSmooth(dummyMsg*
    nextBucketMarkSmooth(dummyMs
```

```
/expand/home/gioachin/cosmology/CVS_latest/ChaNGa1/changa/testco
smo/../charmrun +p32 /expand/home/gioachin/cosmology/CVS_latest/
ChaNGa1/changa/testcosmo/../ChaNGa  -p 8192 cube300.param +bgs
tacksize 1000000 ++batch 8  +cpd +DebugSuspend +DebugDisplay 1
28.174.236.49:0.0 ++server +bgnetwork dummy +LBPeriod 1000000
 +x1 +y1 +z4096ccs: 2
ccs: Server IP = 128.174.236.49, Server port = 37899 $
Charm++: scheduler running in netpoll mode.
Charmrun> started all node programs in 6.596 seconds.
BG info> Simulating 1x1x4096 nodes with 1 comm + 1 work threads ea
ch.
BG info> Network type: dummy.
Dummy network.
BG info> cpufactor is 1.000000.
```

```
TreePiece::nextBucket(dummyMsg* impl_msg)
TreePiece::calculateEwald(dummyMsg* impl_msg)
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
TreePiece::nextBucket(dummyMsg* impl_msg)
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
CkCacheManager::recvData(CkCacheFillMsg* impl_m
TreePiece::fillRequestNode(CkCacheRequestMsg* im
TreePiece::fillRequestNode(CkCacheRequestMsg* im
```

```
envelope
    Sender processor: 3482
    Envelope type: ForBocMsg
    Destination: CkCacheManager::recvData(CkCacheFillMsg* impl_msg)
    Size: 456
CkCacheFillMsg  =
    CMessage_CkCacheFillMsg =
    CkCacheKey (unsigned long) key =  -3458483038843830272
    char* data = 0x10edf20a00000000
```

Fig. 9. Screenshot of ChaNGa debugged on 4,096 virtual processors using 32 real processors.

Once the program has been started, and CHARMDEBUG has connected to it, the user can perform his desired debugging steps, oblivious of the fact that the system in using fewer resources internally. Figure 9 shows the CHANGA application loaded onto four thousand virtual processors. Underneath, we allocated only 32 processors from four local dual quad-core machines. In the bottom left part of the view, we can see all the messages that are enqueued in the selected processor (processor 3,487 in the Figure). Some messages have a breakpoint set (7^{th} message, in orange), and one has actually hit the breakpoint (1^{st} message, in red). In the same message list, we can see that some messages have as destination "TreePiece" (a CHARM++ array element), while others have as destination "CkCacheManager", one of the groups mentioned earlier. One such message is further expanded in the bottom right portion of the view (10^{th} message).

When joining multiple processes inside the same address space, the behavior of the system might be altered. First of all, one virtual processor could corrupt the memory belonging to another one. To solve this problem, the techniques described in [16] can be used. Another problem regards the kind of bugs that can be detected, in particular race conditions. By reducing the amount of physical processors available, a race condition might not appear anymore. A solution is to use record-replay techniques to force the execution of a particular message ordering. This is already available in the virtualized environment, as described in [17]. The other possibility is to force the delivery of messages in the virtualized environment in a different order each time.

8 Related Work

In the realm of parallel debugging, there are several tools that a programmer can use to understand why his program is misbehaving and correct the problem. Widely used commercial products are TotalView [18] from TotalView Technologies, and DDT [19] from Allinea. At least one of these tools is generally available in the majority of parallel supercomputers. Within the Open Source community, a tool worth mentioning is Eclipse [20]. Several Eclipse plugins have been developed to address parallel computing, in particular the Parallel Tools Platform (PTP) [21]. All these debuggers target applications written both in C/C++ and Fortran languages, and using MPI and/or OpenMP [22] as programming models. None of them supports the CHARM++ programming model natively. They all could manage CHARM++ programs if CHARM++ were built with MPI as its underlying communication layer. In this case, though, users would be exposed to the CHARM++ implementation, rather than their own program. Most importantly, while all the tools mentioned can scale to large number of processors, they all require the whole set of processors to be allocated. If the users desires to perform his debugging using one hundred thousand processors, then a big machine has to be used and occupied for long periods of time for the debugging to happen.

Virtualization for High Performance Computing has been claimed to be important [23]. Nevertheless, no tool known to the authors does, at present, provide a debugging environment tailored to thousands of processors or more, while utilizing only the few processors that a local cluster can provide. A few techniques have been developed in contexts other than High Performance Computing leveraging the concept of virtualization. These target the debugging of embedded systems [24], distribute systems [25], or entire operating systems using time-travel techniques [26, 27]. All of them target virtual machines (such as Xen [28] or IBM Hypervisor [29]) where the entire operating system is virtualized. Using virtual machines may pose problems for a normal user as the installation and configuration of such virtual environments require administration privileges, and most supercomputers do not provide them by default. Our technique, instead, resides entirely in the user space, and does not suffer from this limitation.

9 Conclusions and Future Work

In this paper, we presented an innovative technique to address the issue of debugging applications on very large number of processors without consuming large amount of resources. In order to do this, we extended and integrated CHARMDEBUG, a debugger for CHARM++ applications, with BigSim, an emulator for large machines. By combining these two systems, and solving the resultant challenges in scaling and integration, we were able to provide the user a seamless debugging approach that uses much fewer processors than those requested by the user. This is accomplished by internally allocating multiple *virtual* processors inside each physical processor. We demonstrated the feasibility of this approach by studying the virtualization overhead with real world applications. We showed examples of the debugger used on many processors, displaying information about objects, breakpoint, and the content of each virtual processor. Furthermore, we also extended this technique to applications written in MPI, one of the most popular parallel programming model.

With the co-existence of multiple virtual processors inside the single address space of a physical processor, some memory operations have been disabled. For examples, searching for memory leaks. This and other operations require the debugger to disambiguate which virtual processor allocated the memory. One approach would be to use the same memory tagging mechanism described in [16] and cluster memory allocation by virtual processor.

Another future work regards MPI. As we described in section 5, currently CHARM-DEBUG focuses primarily on applications written in CHARM++. While it can debug MPI applications using the AMPI implementation of the MPI standard, we realize that for a programmer debugging his application there may be unnecessary overhead. For the future, we are considering possible extensions to provide a more natural debugging also for MPI programs.

Acknowledgments

This work was supported in part by the NSF Grant OCI-0725070 for Blue Waters, and by the Institute for Advanced Computing Applications and Technologies (IACAT).

References

1. L. V. Kale and G. Zheng, “Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects,” in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.
2. L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, “Programming Petascale Applications with Charm++ and AMPI,” in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. Chapman & Hall / CRC Press, 2008, pp. 421–441.
3. C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, “Performance Evaluation of Adaptive MPI,” in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.
4. F. Gioachin, C. W. Lee, and L. V. Kalé, “Scalable Interaction with Parallel Applications,” in *Proceedings of TeraGrid’09*, Arlington, VA, USA, June 2009.
5. Free Software Foundation, “GDB: The GNU Project Debugger,” http://www.gnu.org/software/gdb/.
6. G. Zheng, A. K. Singla, J. M. Unger, and L. V. Kalé, “A parallel-object programming model for petaflops machines and blue gene/cyclops,” in *NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS)*, Fort Lauderdale, FL, April 2002.
7. G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé, “Simulation-based performance prediction for large parallel machines,” in *International Journal of Parallel Programming*, vol. 33, no. 2-3, 2005, pp. 183–207.
8. J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, “NAMD: Biomolecular simulation on thousands of processors,” in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, MD, September 2002, pp. 1–18.
9. National Center for Supercomputing Applications, “Blue Waters project,” http://www.ncsa.illinois.edu/BlueWaters/.
10. C. Mei, “A preliminary investigation of emulating applications that use petabytes of memory on petascale machines,” Master’s thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2007, http://charm.cs.uiuc.edu/papers/ChaoMeiMSThesis07.shtml.

11. M. Potnuru, “Automatic out-of-core exceution support for charm++,” Master’s thesis, University of Illinois at Urbana-Champaign, 2003.
12. S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kale, and P. M. Ricker, “Automatic MPI to AMPI Program Transformation using Photran,” in *3rd Workshop on Productivity and Performance (PROPER 2010)*, no. 10-14, Ischia/Naples/Italy, August 2010.
13. X. Jiao, G. Zheng, P. A. Alexander, M. T. Campbell, O. S. Lawlor, J. Norris, A. Haselbacher, and M. T. Heath, “A system integration framework for coupled multiphysics simulations,” *Engineering with Computers*, vol. 22, no. 3, pp. 293–309, 2006.
14. A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, “Overcoming scaling challenges in biomolecular simulations across multiple platforms,” in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.
15. P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, “Massively parallel cosmological simulations with ChaNGa,” in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.
16. F. Gioachin and L. V. Kalé, “Memory Tagging in Charm++,” in *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, Seattle, Washington, USA, July 2008.
17. F. Gioachin, G. Zheng, and L. V. Kalé, “Robust Record-Replay with Processor Extraction,” in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD - VIII)*, Trento, Italy, July 2010.
18. TotalView Technologies, “TotalView® debugger,” http://www.totalviewtech.com/TotalView.
19. Allinea, “The Distributed Debugging Tool (DDT),” http://www.allinea.com/index.php?page=48.
20. T. E. Foundation, “Eclipse - an open development platform,” http://www.eclipse.org/.
21. G. R. Watson and C. E. Rasmussen, “A strategy for addressing the needs of advanced scientific computing using eclipse as a parallel tools platform,” Los Alamos National Laboratory, Tech. Rep. LA-UR-05-9114, December 2005.
22. L. Dagum and R. Menon, “OpenMP: An Industry-Standard API for Shared-Memory Programming,” *IEEE Computational Science & Engineering*, vol. 5, no. 1, January-March 1998.
23. M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, “Virtualization for high-performance computing,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, pp. 8–11, 2006.
24. Y. Pan, N. Abe, K. Tanaka, and H. Taki, “The virtual debugging system for developing embedded software using virtual machinery,” *Embedded and Ubiquitous Computing*, vol. 3207, pp. 139–147, 2004.
25. D. Gupta, K. V. Vishwanath, and A. Vahdat, “Diecast: Testing distributed systems with an accurate scale model,” in *In Proceedings of the 5th USENIX Symposium on Networked System Design and Implementation (NSDI08*. USENIX Association, 2008.
26. P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor, “Virtual machine time travel using continuous data protection and checkpointing,” *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 1, pp. 127–134, 2008.
27. S. T. King, G. W. Dunlap, and P. M. Chen, “Debugging operating systems with time-traveling virtual machines,” in *ATEC ’05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 1–1.
28. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.
29. “The research hypervisor: A multi-platform, multi-purpose research hypervisor,” IBM Research, http://www.research.ibm.com/hypervisor/.