Scalable Interaction with Parallel Applications

Filippo Gioachin, Chee Wai Lee and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
gioachin@uiuc.edu, cheelee@uiuc.edu, kale@cs.uiuc.edu

***Abstract*—A range of tools, from parallel debuggers to performance analysis/visualization to simulation visualizers, can benefit from interaction with a running parallel application. To be effective, this live interaction needs to be highly scalable. Such scalability for parallel applications is available in the form of the Charm++/AMPI runtime system. Charm++ is an object-based, message-driven model of parallel programming that has enabled high levels of scalability for complex applications like NAMD, a production bio-molecular simulation code frequently used on Teragrid systems. External tools may communicate via a socket connection with an executing parallel Charm++ application through our Converse Client-Server (CCS) interface and framework. We demonstrate various tools that exploit the scalable properties of Charm++'s adaptive runtime to intersperse interaction activity and communication with the running application's work. We show how this allows tools to interact effectively and scalably with parallel applications running on thousands of processors.**

I. Introduction

There are several situations where an application developer, or even an end-user, can benefit from interacting with a live parallel application. For example, debugging an application that exhibits the faulty behavior only on a large number of processors, or visualizing application performance or application behavior during its operation. Often such capabilities, when they exist, suffer from scalability limits. There are multiple reasons for such limits. These include: inability to share the same network used by the application to perform communication for data collection, limits on number of connections clients can maintain, and performance bottlenecks in the communication infrastructure.

We present our experience with scalable interaction in the context of the Charm++/AMPI runtime system, and its client-server interface called CCS. By exploiting the fact that our runtime system is message-driven and allows effortless interspersing of "in-band" and "out-of-band" communication (i.e. the application's communication and the communication needed for the interactive functionality can be interspersed), we overcome some of the traditional limitations. CCS intrinsic capability to communicate with the parallel application, with data collections from all (or any subset) of processors, as well as the ability to inject a message onto any specific processor, makes it feasible to interact with large number of processors.

We will enumerate a few use-cases of this scalable interaction framework, and demonstrate the scalability of the interactive functionalities with metrics including response time and impact on application performance. The use-cases include: a parallel debugger, a performance analysis tool, and live application-level visualization/analysis applications.

II. Converse Client-Server Model

Converse Client-Server (CCS)[3] is a communication protocol that allows parallel applications to receive requests from remote clients. This protocol is part of Charm++'s underlying system specifications and is therefore available to any Charm++ application. Note that "application" does not mean only the user written code, but also the Charm++ runtime system and its modules that run as part of the application itself. In this scenario, if a system module decides to use CCS, the user code does not require any change, unless it want to explicitely take advantage of the feature. We shall see some examples of this in later sections.

CCS obeys normal Charm++ semantics. Upon a request made by a CCS client, a message is generated inside the application. Computation by the application is triggered by the delivery of this message. As such, CCS requests are serviced asynchronously with respect to the rest of the application which can proceed unaffected. When an application, or Charm++ module, desires to use the CCS protocol, it must register one or more handlers, each with an associated tag. This ensures that requests sent by clients can be correctly matched and delivered to the intended handler. Registration is performed by calling a function into the CCS framework. Moreover, at startup, a flag must be passed to the application to ensure that the runtime system opens a socket and listens for incoming connections. The connection parameters are printed to standard output by the Charm++ RTS. Remote clients can send requests to the parallel application using this information. After receiving a CCS request message, the application can perform any kind of operation, including complicated parallel broadcast and reductions. Finally, a reply can be returned to the client via the CCS protocol.

More recently, we created a new extension to CCS in the context of high-level scripting languages. This al-

lows Python scripts to be dynamically uploaded into a running parallel application. These scripts are executed by the parallel application into Python interpreters created on demand, and can interact with the application in various ways, as described in [4]. This extension allows greater flexibility to determine the functionality needed by the application past compile time.

III. Large-Scale Parallel Debugging

Debugging applications is known to be a difficult task for programmers. Errors in parallel applications are even harder to track: to the list of problems found in sequential programs, others appear due to the distributed nature of parallel applications. Major parallel debuggers, like TotalView [13] or DDT [1], maintain individual connections between the client and each node in the parallel application. This hinders usability when using thousands of processors due to the large response time for even simple operations. Other tools like STAT [10], based on MRNet, has shown excellent scalability to very large machines. Unfortunately, STAT is not a full debugger, and it can only focus the programmer toward a set of processors to look at. Moreover, the range of errors detected is limited.

CharmDebug [6] is a graphical tool written in Java. It targets applications written in Charm++, allowing programmers to remotely debug them. In a typical scenario, the programmer starts CharmDebug on her own workstation. Then, through CharmDebug, she can select and start an application on a remote parallel machine where she has previously compiled it. Inside Charm++ there is a built-in CharmDebug plugin. The combination of the graphical tool and this plugin allows the programmer to visualize information pertinent to her code. Such information includes, but is not limited to, the messages queued in the system, the objects present on a processor, and the state of any object. Moreover, the programmer can set breakpoints on any entry method.

By using CCS as underlying communication layer, CharmDebug can connect as easily to a ten thousand processor parallel application as to one running on just a few processors. The sequential client only manages meta-data of the application, and does not need to maintain a debugging connection to each single process allocated. The application notifies the client CharmDebug when breakpoints and other events occur. One such event is also the crash of one of the processor. If the application were to crash and disappear, debugging would be impossible. Thus, we rely on the Charm++ RTS to freeze the faulty processors upon error, and make it available to CharmDebug for inspection.

Another feature available thanks to the Python scripting interface is introspection [4]. The programmer can upload a script to check for out-of-band values in the application data structures. When such anomalies are discovered, Charm++ will stop the application and give the user the possibility to analyse further. For example, he could attach a sequential debugger to the faulty processor.

To prove the capabilities of CharmDebug at scale, we ran tests using Kraken, a TeraGrid Cray XT4 machine at NICS (University of Tennessee/ORNL). We used a simple Hello World program, and froze it at the beginning of execution. Since we used the MPI build of Charm++ (the only one currently available on the system), we had to rely on the system launcher to start the parallel application. Once the application was started, we attached to it, submitted a few test queries, and timed them. Each query gathered the total amount of memory allocated by all processors, therefore involving all the processors in the operation. We ran queries from tens of processors up to 4,096 processors, and in all cases the response time perceived by the client has been between 12 and 20 milliseconds. The attach process itself took only tens of millisconds for the considered range of processors, in comparison to other tools where either direct connections or indirect communication networks have to be established. Note that this time includes the processing time inside the java client itself, the communication between the client and the parallel application, and the parallel operation to gather the memory information.

In the future, we are looking at enhancing the features provided by CharmDebug to help the programmer focus on the faulty processors, and isolate them from the whole. For example, by integrating the Charm++ checkpoint-restart mechanisms [14] we can store and retrieve checkpoints using the full power of the parallel machine, and even extract single processors to re-execute on a local workstation.

IV. Scalable Runtime Performance Analysis and Visualization

Projections is the built-in parallel performance instrumentation and analysis framework [7] of the Charm++ runtime. As a Charm++ application executes, relevant performance information on each processor is tracked and recorded into a local memory buffer. This information can be in the form of a highly detailed trace event log or a more compact profile that captures a summary of performance metrics for each pre-determined time-interval.

The traditional parallel performance analysis process for Charm++ applications typically involves three steps. Performance data is first written out to disk at the termination of the parallel application. The files are

then transferred from the supercomputing facility to a local workstation. Finally, an analyst uses the visualization component of PROJECTIONS to read the files in order to find performance problems or bottlenecks.

There are scalability problems with the traditional process. We frequently conduct scaling studies for our key CHARM++ applications like NAMD [11] and OPENATOM [2] on a wide variety of very large machines. For large simulations executed on very high processor counts, the total volume of performance data can grow extremely large. This directly impacts the time needed to transfer the files to a workstation for analysis. It also affects the time taken to load an appropriate portion of the data, particularly when the information needs to sum metrics across all processors. As a part of the solution to these scalability problems, we have investigated techniques to reduce the volume of data generated at the end of an application while preserving the necessary performance information [9].

In the case of the traditional performance analysis process, we observe that performance data stays unused in its buffers until written out to disk at the end of the application. The CCS interface to an external software agent presents an opportunity for scalability improvements. We exploit the CHARM++ runtime's ability to adaptively schedule work while overlapping computation with communication. Captured performance data could be processed on each processor and gathered to a single root processor through a global reduction. Depending on the size of the gathered data, the frequency of data gathering and the nature of the application at large scales, the impact on the application's performance profile could be minimal. An external client can then acquire the gathered performance data through CCS communication with that root processor.

This approach yields two immediate benefits. The first is we can now choose to re-use buffer space cleared by the data gathering process. The second benefit is the ability to start sending performance data early to an external client while the CHARM++ application is still running. The external client could, in turn, provide immediate visualization support and/or write the incoming data into files for subsequent analysis.

We have implemented a very basic initial scheme that provides an external client with a CCS hook directly into the PROJECTIONS performance instrumentation framework. Commensurate with the goals of live analysis, the implementation supports only profile summary performance data, which is more compact and easier to manipulate than full event traces. The framework, from processor 0, requests performance data from each processor every 1 second. A simple text-based external client makes requests to the framework on processor 0. Each time a request arrives, the framework sends to the client whatever complete data is gathered.

Num. Cores	exec time (no CCS)	exec time (with CCS)
4095	21.44 s	21.46 s
8191	37.84 s	37.71 s

TABLE I: **Impact of performance data gathering (for CCS interaction) on program time. A global reduction of 8 kB messages from each processor occurs every second.**

We conducted our initial tests on up to 8,191 Kraken XT5 processors at NICS. Our preliminary results using a simple CHARM++ program are promising. Even on 4,095 and 8,191 processors, the program was able to tolerate continuous live streaming of performance data with no significant overhead over a normal profile-generating execution of the same program (Table I).

Future work will involve a more complete overhead study by varying the volume of data contributed by each processor. We will use more complex applications like NAMD and OPENATOM at far larger scales. We will explore more flexible schemes for communicating performance data to a root processor, for example, one which seeks windows of opportunity when no application-based work is being performed on a processor. We will also aim to demonstrate a more full-featured client which can make use of this feature. In the longer term, we can use CCS as a mechanism to conduct post-mortem analysis of in-memory performance data just prior to normal application termination. This exploits the availability of both the large memory pool of a supercomputer as well as the computational power after an application is done with its work. Post-mortem analysis could involve the analyst interactively guiding the analysis framework in its data reduction work. This could improve the quality of the performance data retained while maintaining the scalability benefits.

V. Visualizing Realtime Application-generated Images

Scientific simulations typically produce large output files that need to be analyzed to make breakthough discoveries. As the outputs' size grows faster than the memory and pocessing capability of a single machine, the need for a parallel visualization and analysis tool increases. Moreover, in many situations, scientific applications can benefit from human steering while running. In order to steer the application, the scientist must to be able to inspect the progress of the application, and make decisions accordingly.

LiveViz is a visualization tool that allows CHARM++ applications to easily compose and deliver images to clients requesting them. An application can decide to run in one of two modes. In pull mode, a client request triggers the generation of an image. A pre-registered callback function is used by LiveViz to collect

sub-images from a collection of CHARM++ objects in the application. In push mode, the application generates images when needed, and posts them to LiveViz. A client can collect posted images at any later time. The LiveViz module provides all the functionality to interact with clients. Internally, LiveViz uses the CHARM++ broadcast and reduction framework to combine portions of an image from the scattered object into a single location. While combining the final image, different composition mechanism are selectable.

LiveViz has been effectively used in Salsa [12], a parallel analysis tool for particle-based datasets. An analyst can load a large dataset onto a parallel machine, visualize it, and apply either predefined filters or ad-hoc Python scripts to the data under analysis. LiveViz is also under integration into ChaNGa [5] as a way to follow the progress of a simulation. In its design, LiveViz had some drawbacks that might cause reduced performance under certain circumnstances: such as when a client is either at a great distance from the parallel machine (long latency), or on a slower network (small bandwidth), the framerate may be too small for productive analysis. While small bandwidth can be tolerated by using compressing techniques, long latencies are harder to tolerate in LiveViz.

More recently, a new scheme based on CCS and using Parallel Impostors, described in Lawlor's thesis [8], has been developed. In this scheme, the client does not reload a complete image at every frame, but maintains 2-dimensional images in a local cache. These images are moved accordingly to the observer's camera movement. When the relative position between an image and the camera exceeds a predetermined threshold, a new image is generated in the server and delivered to the client. This mechanism allows the client to run at a higher framerate than what is currently possible with LiveViz, while still maintaining accuracy in the visualization. Large delays in the network or network congestion can be easily tolerated by the client using the cached images. For a comparison with other parallel visualization tools, we refer to Lawlor's thesis.

LiveViz and Parallel Impostors are lively project currently funded. Along with our collaborators, and we are looking at integrating the new scheme into Salsa and ChaNGa. We expect the applications to benefit from the higher framerate enabled by the improved latency and bandwidth tolerance. We are also testing the system on many thousands of processors to validate its performance.

VI. Conclusion and Future Work

We presented several case studies of scalable interaction between a client, be it a visualization tool, debugger, or performance analysis tool, with a remote parallel application acting as a server. The common underlying framework providing scalability and interactiveness is the CHARM++ parallel runtime system and CCS, its client-server communication protocol. In each case, we showed how we addressed and effectively dealt with the issue of scalability. The metric considered is dependent on the application requirements: response time for parallel debugging, overhead/response time for performance analysis, and framerate for remote visualization.

Acknowledgements: This work have been made possible in part by grants NSF OCI-0725070, NASA NNX08AD19G, and NSF ITR-0205611. We would like to thank TeraGrid for the compute time granted through allocation TG-ASC050039N. The authors are grateful to Prof. Tom Quinn (Univ. of Washington) and Prof. Orion Lawlor (Univ. of Alaska), who are collaborators with them on the scientific visualization applications.

References

[1] Allinea. The Distributed Debugging Tool (DDT). http://www.allinea.com/index.php?page=48.

[2] Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, and Glenn J. Martyna. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[3] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CONVERSE programming language manual*, 2006.

[4] Filippo Gioachin and Laxmikant V. Kalé. Dynamic High-Level Scripting in Parallel Applications. In *To appear in Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, May 2009.

[5] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[6] Rashmi Jyothi, Orion Sky Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[7] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.

[8] Orion Sky Lawlor. *Impostors for Parallel Interactive Computer Graphics*. PhD thesis, University of Illinois at Urbana-Champaign, December 2004.

[9] Chee Wai Lee, Celso Mendes, and Laxmikant V. Kalé. Towards Scalable Performance Analysis and Visualization through Data Reduction. In *13th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Miami, Florida, USA, April 2008.

[10] 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. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[11] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[12] Thomas Quinn, Laxmikant Kale, Filippo Gioachin, Orion Lawlor, Graeme Lufkin, and Gregory Stinson. Salsa: a parallel, interactive, particle-based analysis tool. Poster at Supercomputing 2004.

[13] TotalView Technologies. TotalView® debugger. http://www.totalviewtech.com/TotalView.

[14] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.