A Parallel Plug-in Programming Paradigm[⋆]

Ronald Baumann[1,2], Christian Engelmann[1,2], and Al Geist[2]

[1] Department of Computer Science
The University of Reading, Reading, RG6 6AH, UK
[2] Computer Science and Mathematics Division,
Oak Ridge National Laboratory, Oak Ridge, TN 37831-6164, USA
{r.baumann,c.engelmann}@reading.ac.uk
{baumannr,engelmannc,gst}@ornl.gov
http://www.csm.ornl.gov

Abstract. Software component architectures allow assembly of applications from individual software modules based on clearly defined programming interfaces, thus improving the reuse of existing solutions and simplifying application development. Furthermore, the plug-in programming paradigm additionally enables runtime reconfigurability, making it possible to adapt to changing application needs, such as different application phases, and system properties, like resource availability, by loading/unloading appropriate software modules. Similar to parallel programs, parallel plug-ins are an abstraction for a set of cooperating individual plug-ins within a parallel application utilizing a software component architecture. Parallel programming paradigms apply to parallel plug-ins in the same way they apply to parallel programs. The research presented in this paper targets the clear definition of parallel plug-ins and the development of a parallel plug-in programming paradigm.

1 Introduction

Today, parallel and distributed scientific computing is a tool that enables researchers world-wide to solve large-scale problems in many different research areas, such as climate, nanotechnology, quantum chemistry, nuclear fusion, and astrophysics. Scientific high-end computing (HEC) utilizing tens-to-hundreds of thousands of processors enables new scientific breakthroughs in these areas using computational simulations (simulated experiments) of real-world problems. HEC exploits multi-processor parallelism of scientific algorithms on a large scale using common parallel programming paradigms, such as single program multiple data (SPMD) and multiple program multiple data (MPMD). The scientific computation is performed by a set of cooperating individual processes or tasks communicating via message passing and/or remote method invocation as part of a parallel scientific application.

[⋆] This research is sponsored by the Mathematical, Information, and Computational Sciences Division; Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

Software component architectures, such as Harness [1, 2] and the Common Component Architecture (CCA) [3, 4], allow assembly of scientific applications from individual software modules based on clearly defined programming interfaces, thus improving the reuse of existing solutions and simplifying application development. Furthermore, the plug-in programming paradigm additionally enables runtime reconfigurability making it possible to adapt to changing application needs, such as different application phases, and system properties, like resource availability, by loading/unloading appropriate software modules.

Component architectures and plug-in programming are well understood technologies for non-parallel system architectures. HEC applications are inherently parallel requiring adaptation of these technologies to parallel and distributed system architectures. Past research focused on the following two approaches:

– In Harness, the component framework itself runs in a distributed virtual machine (DVM) fashion, where all processes cooperate in a virtual machine environment. This approach enables assembly of SPMD and MPMD applications from components, but requires the DVM to setup all components based on locality. It is not very scalable, but applications are easy to implement due to the support for global component setup. Fault tolerance is addressed through plug-in checkpoint/restart mechanisms using the DVM as a highly available backbone for management and storage.
– In CCA, the component framework itself runs in a parallel (SPMD) fashion, where each process individually is able to manage its own set of components. This approach also enables assembly of SPMD and MPMD applications from components, but requires each component framework instance to setup its components based on locality. It is very scalable, but applications are difficult to implement due to the required local component setup. Fault tolerance is addressed trough application checkpoint/restart mechanisms.

The notion of *parallel plug-ins* evolved from research within the Harness project. Similar to parallel programs, parallel plug-ins are an abstraction for a set of cooperating individual plug-ins communicating via message passing and/or remote method invocation as part of a parallel scientific application. Parallel programming paradigms (SPMD/MPMD) apply to plug-ins in the same way they apply to programs. Parallel plug-ins within Harness are effectively realized by utilizing the distributed virtual machine. Within CCA, parallel plug-ins are effectively realized by implementing parallel programs consisting of plug-ins. Up until now, both approaches avoided a clear definition of parallel plug-ins as it was not well understood.

The research presented in this paper targets the clear definition of parallel plug-ins and the development of a parallel plug-in programming paradigm that combines both approaches in order to further improve reuse of existing solutions and to simplify application development. Harness is being used as a proof-of-concept research vehicle to prototype parallel plug-in management and experimental parallel plug-ins.

In the following sections, we first briefly discuss past and ongoing related research and development efforts. Secondly, we present a clear definition of parallel plug-ins, their programming models (types) and their programming requirements (communication, coordination, and fault tolerance). We continue with a description of our prototype implementation for two distinct scientific application scenarios. This paper concludes with a short summary of the presented research and a discussion future work.

2 Related Work

The research in Harness [1, 2] is a collaborative effort among Oak Ridge National Laboratory (ORNL), University of Tennessee, Knoxville, and Emory University focusing on the design and development of technologies for flexible, adaptable, reconfigurable, lightweight environments for heterogeneous parallel and distributed scientific metacomputing.

As part of the Harness project, a variety of experiments and system prototypes were developed to explore lightweight pluggable runtime environments, assembly of scientific applications from software modules, highly available DVMs, fault-tolerant message passing, fine-grain security mechanisms, and heterogeneous reconfigurable communication frameworks.

Currently, there are three different Harness system prototypes, each concentrating on different research issues. The teams at ORNL [5–8] and at the University of Tennessee [9–12] provide different C variants, while the team at Emory University [13–16] maintains a Java-based alternative.

Conceptually, the Harness software architecture consists of two major parts: a runtime environment (RTE) and a set of plug-in software modules. The multi-threaded RTE manages the set of dynamically loadable plug-ins. While the RTE provides only basic functions, plug-ins may provide a wide variety of services needed in fault-tolerant parallel and distributed scientific computing, such as messaging, scientific algorithms, and resource management. Multiple RTE instances can be aggregated into a DVM.

Our research in parallel plug-ins focuses on the C-based lightweight Harness RTE from ORNL [7] using its dynamic, heterogeneous, reconfigurable communication framework (RMIX) [6] plug-in for fault-tolerant message passing and remote method invocation.

RMIX allows software components to communicate using various remote method invocation (RMI) and remote procedure call (RPC) protocols, such as ONC RPC, by facilitating dynamically loadable provider plug-ins to supply different protocol stacks. While the RMIX base library contains functions that are common to all protocol stacks, like networking and thread management, RMIX provider plug-ins contain protocol stack specific functions for connection management, message formats, and data encoding. Since it is up to the provider plug-ins to reuse RMIX base library functions, implementations may range from lightweight to heavyweight. Moreover, client- and server-side object stubs are very lightweight and protocol independent as they only perform an adaptation to the RMIX system. In addition to standard synchronous RMI/RPC mechanisms, RMIX also offers advanced RMI/RPC invocation semantics, such as asynchronous and one-way. RMIX is not a high-performance message passing system. Its RMI/RPC mechanisms are designed for loosely-coupled systems.

The Harness-RMIX plug-in contains the RMIX base library as well as client- and server-side object stubs of the Harness RTE. Stubs for Harness plug-ins are implemented as separate plug-ins. Since the Harness RTE supports plug-in dependencies, a plug-in requiring RMIX automatically loads its stub plug-in(s), which subsequently loads the RMIX plug-in.

The already mentioned Common Component Architecture (CCA) [3, 4] is a component-based approach targeted at the needs of large-scale, complex, high-end, scientific simulations. CCA relies on a standardized component framework model for scientific applications based on an interface description language for component interfaces, a port model for unified component interaction, core component framework services, a framework configuration API and a framework repository API. Several CCA framework implementations exist that serve different areas of interest for scientific applications. Furthermore, a set of CCA components exist as well as a number of CCA-based scientific applications.

It is our hope that the work presented in this paper will be eventually incorporated in some form into production-type component architectures for scientific HEC, such as the Common Component Architecture, to further improve reuse of existing solutions and to simplify application development.

Other related past and ongoing research and development efforts include lightweight plug-in design patterns [17], a various number of pluggable component frameworks (e.g. the Open CORBA Component Model Platform [18, 19]), as well as recent accomplishments in RTEs for parallel and distributed system architectures, such as Open RTE [20].

3 Parallel Plug-ins

The intent of our research presented in this paper is to merge plug-in programming technologies with pluggable component frameworks for parallel architectures using common parallel programming models and appropriate design patterns in order to provide better reuse of existing solutions as well as easier scientific application development.

A *parallel plug-in* can be defined as a set of individual plug-ins cooperating in a parallel architecture to perform a common task, such as solving a computation or providing a service. Participating individual plug-ins may be located on the same or on distributed computational resources, and communicate with each other for task control and data transfer purposes. Similar to a parallel program, a parallel plug-in is a parallel programming abstraction, where the same parallel programming models, such as SPMD and MPMD, apply.

While the execution environment of a parallel program is a parallel or distributed operating system, parallel plug-ins reside within a component framework for parallel architectures, *i.e.,* within a parallel program. They provide a componentized approach for building parallel applications by offering parallel building blocks with clearly defined interfaces.

In the past, parallel plug-ins have been effectively realized by by utilizing a pluggable DVM environment or by implementing parallel programs consisting of individual plug-ins. Both approaches dealt with the necessary coordination of individual cooperating plug-ins by using either distributed control [8] or localized control [3], while avoiding a clear definition of parallel plug-ins. In both cases, it is up to the plug-in component programmer to take care of task control and data transfer without access to appropriate design patterns.

3.1 Parallel Plug-in Types

A parallel plug-in consists of one or more individual plug-ins that add features and capabilities to the runtime environment of a component framework as part of a parallel application. The following parallel plug-in types (see Figure 1) can be defined based on number, location, and purpose of involved individual plug-ins:

- The singleton (or service) plug-in is a special case of parallel plug-in as it involves only one individual plug-in at one node within the context of a parallel application.
- The SPMD (or replicated) plug-in follows the known SPMD parallel programming model and involves more than one node within the context of a parallel application. The same plug-in code is replicated to different nodes and applied to different data.
- The MPMD (or distributed) plug-in follows the known MPMD parallel programming model and involves more than one node within the context of a parallel application. Different plug-in codes are distributed to different nodes and applied to different data.

3.2 Parallel Plug-in Communication

Communication is essential for parallel plug-ins. Similar to parallel programs, parallel plug-ins communicate using message passing and/or RMI/RPC in order to coordinate individual tasks and to transfer necessary data. In order to identify individual collaborating plug-ins within a parallel plug-in, naming and message/invocation routing is needed.

Message passing systems, such as PVM [21] and MPI [22], typically only provide naming and routing mechanisms for individual processes of a parallel application and not for individual plug-ins. Message tags may be used to identify individual plug-ins within an individual process. However, a separate plug-in naming service is needed in order to support dynamic adaptation, *i.e.*, dynamic loading and unloading of parallel plug-ins.

RMI/RPC systems typically provide a naming and routing mechanism for exported objects or program parts. Individual plug-ins that are part of a parallel plug-in may be exported as objects or program parts using a RMI/RPC system for naming and routing of invocations. Furthermore, local services of individual component framework processes may be exported as objects or program parts as well to enable remote access to basic component framework services, such as loading and unloading of plug-ins.

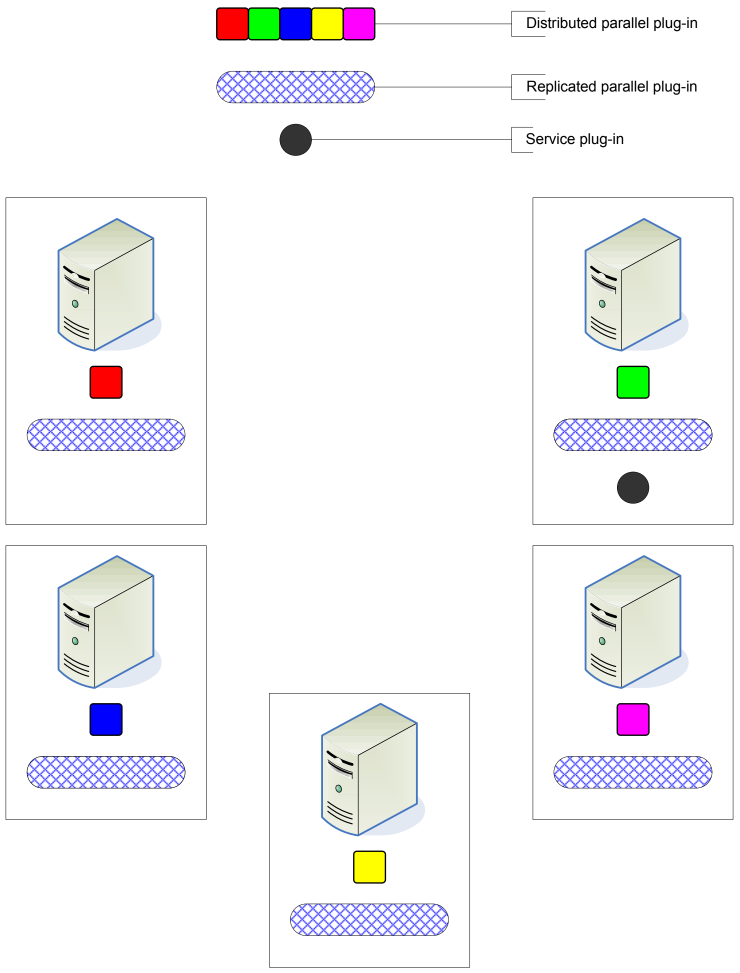

Fig. 1. Parallel Plug-in Types

3.3 Parallel Plug-in Coordination

Coordination of individual plug-ins that are part of a parallel plug-in is needed to perform loading, unloading, task and data distribution, and fault tolerance mechanisms. This coordination is accomplished via separate coordination (service) plug-ins and via the parallel plug-in itself using the communication subsystem with its plug-in naming scheme to address individual plug-ins.

In order to load a parallel plug-in, the used component framework itself or a separate service plug-in contacts individual component framework processes and loads the appropriate individual plug-in based on the parallel plug-in type. Unloading a parallel plug-in is implemented in the same way.

Task and data distribution can be performed by the parallel plug-in itself if it supports self configuration. However, in order to simplify parallel plug-in and application development, a separate service plug-in may be used to execute task and data distribution based on the parallel plug-ins programming model, *i.e.*, by automatically partitioning data and/or assigning tasks.

Similarly, fault tolerance mechanisms, such as checkpoint/restart, may be performed by the parallel plug-in itself if supported, but may also be coordinated by a separate service plug-in based on a design pattern that matches the fault behavior and fault tolerance requirements of the parallel plug-in.

3.4 Parallel Plug-in Fault Tolerance

Fault tolerance is typically realized in three steps: detection, notification, and reconfiguration. In parallel computing, detection and notification are typically performed by the communication system using timeouts to detect faults and the naming scheme to identify faulty communication endpoints.

A parallel plug-in may be reconfigured using the same techniques as for parallel programs, with the exception that reconfiguration takes place within the parallel plug-in programming scope. Parallel plug-in state may be regularly stored on stable storage in the same way a parallel program checkpoint is stored. Upon failure, the entire parallel plug-in or parts of it (individual plug-ins) may be restarted on different nodes. Loss of state due to failures may be ignored if the application is able to continue without extensive reconfiguration [23].

As already mentioned earlier, fault tolerance mechanisms for parallel plug-ins may be encapsulated into a separate service plug-in based on a design pattern that matches the fault behavior and fault tolerance requirements of the parallel plug-in. Furthermore, such a service plug-in may directly interface to existing fault tolerance technologies, such as checkpoint/restart layers.

4 Prototype Implementation

A proof-of-concept prototype has been implemented as part of a Master's thesis [24] using the Harness RTE as a research vehicle.

The Harness RTE offers a lightweight backbone to load and unload individual plug-ins into a multi-threaded process residing on a single node. Multiple Harness RTEs located on the same or different nodes are used as a lightweight backbone to load and unload parallel plug-ins utilizing a parallel parallel plug-in manager for coordination and the Harness-RMIX plug-in for communication.

In the following, we describe the developed parallel plug-in manager and our efforts in implementing a parallel plug-in design pattern for two distinct scientific application scenarios.

4.1 Parallel Plug-In Manager

The parallel plug-in manager (PPM) is itself a service plug-in and provides parallel plug-in management services for both parallel plug-in programming models, SPMD and MPMD. These services include: loading, unloading, task and data distribution, and fault tolerance support.

The PPM loads a parallel plug-in by starting and contacting a set of Harness RTEs via remote method invocation to load a specific individual Harness plug-in. The set of available nodes is given by the user. The number of involved Harness RTEs must be equivalent or more than the number of individual plug-ins needed. A round robin schedule may be used in a more sophisticated solution to allow oversubscribtion of nodes.

Loading a parallel plug-in introduces the problem of partial success, *i.e.*, not all individual plug-ins were loaded due to unavailability of resources (plug-in, Harness RTE, or node). If there are more nodes available than needed, the PPM retries to load an individual plug-in at a different location.

The parallel plug-in loading fails if not all required individual plug-ins were loaded. This decision entirely depends on the parallel plug-in to load and is guided by the user by configuring the PPM appropriately.

Task and data distribution depending on the parallel plug-in programming model may be performed as part of the parallel plug-in loading procedure. Furthermore, fault tolerance mechanisms are supported via the PPM by offering restart of failed parallel plug-in parts.

4.2 Monte Carlo Integration

The first proof-of-concept parallel plug-in has been implemented using the SPMD programming model performing a Monte Carlo integration algorithm in a bag-of-tasks fashion. The developed parallel plug-in consists of a Monte Carlo integration algorithm, where each individual plug-in performs an equal share of the overall computation. The PPM loads the parallel plug-in on all available nodes, while accepting partial loading success.

Fault tolerance has been implemented using a separate service plug-in to reload failed plug-ins upon notification. The Monte Carlo integration share of a failed plug-in is repeated entirely. The degree of fault tolerance is $n-1$, *i.e.*, $n-1$ out of n Harness RTEs may fail.

4.3 Image Processing

The second proof-of-concept parallel plug-in has been implemented using the MPMD programming model performing a sequence of image processing algorithms in a pipeline fashion. The developed parallel plug-in consists of a set of individual plug-ins, each performing a different computation and forwarding its result to the next plug-in. The PPM loads the parallel plug-in on the necessary number of nodes. It does not accept partial loading success.

Fault tolerance has been implemented using a separate service plug-in to reload failed plug-ins and to reconfigure the pipeline upon notification. Each plug-in stores its results temporarily until completion has been acknowledged by the next plug-in in the pipeline. The image processing algorithm of a failed plug-in is repeated for all unacknowledged results. The degree of fault tolerance is 1, *i.e.*, 1 Harness RTE may fail. The degree may be increased significantly using stable storage for intermediate results.

5 Conclusion

With this paper, we presented results of our recent research in a parallel plug-in programming paradigm for software component architectures in parallel and distributed scientific high-end computing. We defined the parallel plug-in abstraction, associated programming models, and resulting programming requirements. We demonstrated similarities and differences between parallel plug-ins and programs with regards to their programming models and execution environments. We described a Harness-based proof-of-concept prototype of a parallel plug-in manager and of parallel plug-ins for two distinct scientific application scenarios. Further implementation details have been published in a Master's thesis [24].

Our research indicates that the parallel plug-in programming paradigm presented in this paper is an appropriate design template for software component architectures in parallel and distributed scientific high-end computing.

It is our hope that the work presented in this paper will be eventually incorporated in some form into production-type component architectures, such as the Common Component Architecture, to further improve reuse of existing solutions and to simplify application development.

References

1. Geist, G.A., Kohl, J.A., Scott, S.L., Papadopoulos, P.M.: HARNESS: Adaptable virtual machine environment for heterogeneous clusters. Parallel Processing Letters **9**(2) (1999) 253–273
2. Beck, M., Dongarra, J.J., Fagg, G.E., Geist, G.A., Gray, P., Kohl, J.A., Migliardi, M., Moore, K., Moore, T., Papadopoulous, P., Scott, S.L., Sunderam, V.: HARNESS: A next generation distributed virtual machine. Future Generation Computer Systems **15**(5–6) (1999) 571–582
3. Common Component Architecture Forum: Home Page. Available at http://www.cca-forum.org (2006)
4. SciDAC Center for Component Technology for Terascale Simulation Software (CCTTSS): High-Performance Scientific Component Research: Accomplishments and Future Directions. Available at http://www.cca-forum.org/db/news/documentation/whitepaper05.pdf (2005)
5. Oak Ridge National Laboratory, Oak Ridge, TN, USA: Harness project. Available at http://www.csm. ornl.gov/harness (2006)
6. Engelmann, C., Geist, G.A.: RMIX: A dynamic, heterogeneous, reconfigurable communication framework. In: Lecture Notes in Computer Science: Proceedings of International Conference on Computational Science (ICCS) 2006, Part II. Volume 3992., Reading, UK (May 28-31, 2006) 573–580
7. Engelmann, C., Geist, G.A.: A lightweight kernel for the Harness metacomputing framework. In: Proceedings of 14^{th} Heterogeneous Computing Workshop (HCW) 2005, Denver, CO, USA (April 4, 2005)

8. Engelmann, C., Scott, S.L., Geist, G.A.: Distributed peer-to-peer control in Harness. In: Lecture Notes in Computer Science: Proceedings of International Conference on Computational Science (ICCS) 2002. Volume 2330., Amsterdam, The Netherlands (April 21-24, 2002) 720–727
9. University of Tennessee, Knoxville, TN, USA: Harness project. Available at http://icl.cs.utk.edu/harness (2006)
10. University of Tennessee, Knoxville, TN, USA: FT-MPI project. Available at http://icl.cs.utk.edu/ftmpi (2006)
11. Fagg, G.E., Bukovsky, A., Vadhiyar, S., Dongarra, J.J.: Fault-tolerant MPI for the Harness metacomputing system. In: Lecture Notes in Computer Science: Proceedings of International Conference on Computational Science (ICCS) 2001. Volume 2073. (2001) 355–366
12. Fagg, G.E., Bukovsky, A., Dongarra, J.J.: Harness and fault tolerant MPI. Parallel Computing **27**(11) (2001) 1479–1495
13. Emory University, Atlanta, GA, USA: Harness project. Available at http://www.mathcs.emory.edu/harness (2006)
14. Kurzyniec, D., Wrzosek, T., Sunderam, V., Slominski, A.: RMIX: A multiprotocol RMI framework for Java. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2003, Nice, France (April 22-26, 2003) 140–145
15. Kurzyniec, D., Sunderam, V.S., Migliardi, M.: PVM emulation in the Harness metacomputing framework - Design and performance evaluation. In: Proceedings of International Symposium on Cluster Computing and the Grid (CCGRID) 2002, Berlin, Germany (May 21-24, 2002) 282–283
16. Sunderam, V., Kurzyniec, D.: Lightweight self-organizing frameworks for metacomputing. In: Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC) 2002, Edinburgh, Scotland (July 24-26, 2002) 113–124
17. Mayer, J., Melzer, I., Schweiggert, F.: Lightweight plug-in-based application development. In: Lecture Notes In Computer Science: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODe'02). Volume 2591., Erfurt, Germany (October 7-10, 2002) 87–102
18. Object Management Group, Inc: CORBA Component Model. Available at http://www.omg.org/technology/documents/formal/components.htm/ (2006)
19. ObjectWeb Consortium: OpenCCM - The Open CORBA Component Model Platform. Available at http://openccm.objectweb.org/ (2006)
20. OpenRTE Team: OpenRTE project. Available at http://www.open-rte.org (2006)
21. Geist, G.A., Beguelin, A., Dongarra, J.J., Jiang, W., Manchek, R., Sunderam, V.S.: PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, USA (1994)
22. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. MIT Press, Cambridge, MA, USA (1996)
23. Engelmann, C., Geist, G.A.: Super-scalable algorithms for computing on 100,000 processors. In: Lecture Notes in Computer Science: Proceedings of International Conference on Computational Science (ICCS) 2005. Volume 3514., Atlanta, GA, USA (May 22-25, 2005) 313–320
24. Baumann, R.: Design and development of prototype components for the Harness high-performance computing workbench. Master's thesis, Department of Computer Science, University of Reading, UK (March 6, 2006)