The Open Run-Time Environment (OpenRTE): A Transparent Multi-cluster Environment for High-Performance Computing

R.H. Castain[1], T.S. Woodall[1], D.J. Daniel[1]
J.M. Squyres[2], B. Barrett[2], and G.E. Fagg[3]

[1] Los Alamos National Lab
[2] Indiana University
[3] University of Tennessee, Knoxville

Abstract. The Open Run-Time Environment *(OpenRTE)*—a spin-off from the Open MPI project—was developed to support distributed high-performance computing applications operating in a heterogeneous environment. The system transparently provides support for interprocess communication, resource discovery and allocation, and process launch across a variety of platforms. In addition, users can launch their applications remotely from their desktop, disconnect from them, and reconnect at a later time to monitor progress. This paper will describe the capabilities of the OpenRTE system, describe its architecture, and discuss future directions for the project.

1 Introduction

The growing complexity and demand for large-scale, fine-grained simulations to support the needs of the scientific community is driving the development of petascale computing environments. Achieving such a high level of performance will likely require the convergence of three industry trends: the development of increasingly faster individual processors; integration of significant numbers of processors into large-scale clusters; and the aggregation of multiple clusters and computing systems for use by individual applications.

Developing a software environment capable of supporting high-performance computing applications in the resulting distributed system poses a significant challenge. The resulting run-time environment (RTE) must be capable of supporting heterogeneous operations, efficiently scale from one to large numbers of processors, and provide effective strategies for dealing with fault scenarios that are expected of petascale computing systems [7]. Above all, the run-time must be easy to use, providing users with a transparent interface to the petascale environment in a manner that avoids the need to customize applications when moving between specific computing resources.

The Open Run-Time Environment (OpenRTE) has been designed to meet these needs. Originated as part of the Open MPI project [3]—an ongoing collaboration to create a new open-source implementation of the Message Passing

B. Di Martino et al. (Eds.): EuroPVM/MPI 2005, LNCS 3666, pp. 225–232, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Interface (MPI) standard for parallel programming on large-scale distributed systems [1,8]—the OpenRTE project has recently spun-off into its own effort, though the two projects remain closely coordinated. This paper describes the design objectives that under-pin the OpenRTE and its architecture.

Terminology. The concepts discussed in the remainder of this paper rely on the prior definition of two terms. A *cell* is defined as a collection of computing resources *(nodes)* with a common point-of-contact for obtaining access, and/or a common method for spawning processes on them. A typical cluster, for example, would be considered a single cell, as would a collection of networked computers that allowed a user to execute applications on them via remote procedure calls. Cells are assumed to be persistent—i.e., processors in the cell are maintained in an operational state as much as possible for the use of applications.

In contrast, a *local computer* is defined as a computer that is not part of a cell used to execute the application, although application processes can execute on the local computer if the user so desires. Local computers are not assumed to be persistent, but are subject to unanticipated disconnects. Typically, a local computer consists of a user's notebook or desktop computer.

2 Related Work

A wide range of approaches to the problem of large scale distributed computing environments have been studied, each primarily emphasizing a particular key aspect of the overall problem. LAM/MPI, for example, placed its emphasis on ease of portability and performance [9], while LA-MPI and HARNESS FT-MPI focused on data and system fault tolerance (respectively) [4,5]. Similarly, the Globus program highlighted authentication and authorization to allow operations across administrative zones [6].

The OpenRTE project has drawn from these projects, as well as other similar efforts, to meet objectives designed to broaden the petascale computing user community.

3 Design Objectives

The OpenRTE project embraces four major design objectives: ease of use, resilient operations, scalability, and extensibility.

Ease of Use. Acceptance of a RTE by the general scientific community (i.e., beyond that of computer science) is primarily driven by the system's perceived ease of use and dependability. While both of these quantities are subjective in nature, there are several key features that significantly influence users' perceptions.

One predominant factor in user acceptance is transparency of the RTE—i.e., the ability to write applications that take advantage of a system's capabilities without requiring direct use of system-dependent code. An ideal system should support both the ability to execute an application on a variety of compute resources, and allow an application to scale to increasingly larger sizes by drawing

resources from multiple computational systems, without modification. This level of transparency represents a significant challenge to any RTE in both its ability to interface to the resource managers of multiple cells, and the efficient routing of shared data between processes that may no longer be collocated within a highly-interconnected cell (e.g., a cluster operating on a high-speed network fabric).

Several desirable system features also factor into users' perceptions of a RTE's ease of use. These include the ability to:

- Remotely launch an application directly from the user's desktop or notebook computer—i.e., without requiring that the user login to the remote computing resource and launch the application locally on that system. Incorporated into this feature is the ability to disconnect from an application while it continues to execute, and then reconnect to the running application at a later time to monitor progress, potentially adjust parameters "on-the-fly", etc.
- Forward input/output to/from remote processes starting at the initiation of the process, as opposed to only after the process joins the MPI system (i.e., calls MPI_INIT).
- Provide support for non-MPI processes, including the ability to execute system-level commands on multiple computing resources in parallel.
- Easily interface applications to monitoring and debugging tools. Besides directly incorporating support for the more common tools (e.g., TotalView), the RTE should provide interfaces that support integration of arbitrary instrumentation (e.g., those custom developed by a user).

Finally, the RTE should operate quickly (in terms of startup and shutdown) with respect to the number of processes in an application, and should not require multiple user commands to execute. Ideally, the run-time will sense its environment and take whatever action is required to execute the user's application.

Resilient. Second only to transparency in user acceptance is dependability. The RTE must be viewed as solid in two key respects. First, the run-time should not fail, even when confronted with incorrect input or application errors. In such cases, the run-time should provide an informational error message and, where appropriate, cleanly terminate the offending application.

Secondly, the RTE should be capable of continuing execution of an application in the face of node and/or network failures. Current estimates are that petascale computing environments will suffer failure of a node every few hours or days [7]. Since application running times are of the same order of magnitude, an acceptable RTE for petascale systems must be capable of detecting such failures and initiating appropriate recovery or shutdown procedures. User-definable or selectable error management strategies will therefore become a necessity for RTE's in the near future.

Scalable. A RTE for distributed petascale computing systems must be capable of supporting applications spanning the range from one to many thousands of processes, operating across one to many cells. As noted earlier, this should be accomplished in a transparent fashion—i.e., the RTE should automatically

scale when adding processes. This will require that users either provide binary-compatible images for each machine architecture in the system, pre-position files and libraries as necessary—or that the run-time be capable of providing such services itself.

Extensible. The design objectives presented thus far have all dealt with the RTE from the user's perspective. However, there are also significant requirements in relation to both developers and the larger computer science community. Specifically, the RTE should be designed to both support the addition of further features and provide a platform for research into alternative approaches for key subsystems.

This latter element is of critical importance but often overlooked. For example, the possible response of the run-time to non-normal termination of a process depends somewhat on both the capabilities of the overall computing environment, the capabilities of the RTE itself, and the nature of the application. The responses can vary greatly, ranging from ignoring the failure altogether to immediate termination of the application or restarting the affected process in another location. Determining the appropriate response for a given situation and application is a significant topic of research and, to some extent, personal preference.

Supporting this objective requires that the RTE allow users and developers to *overload* subsystems—i.e., overlay an existing subsystem with one of their own design, while maintaining the specified interface, in a manner similar to that found in object-oriented programming languages.

4 Architecture

The OpenRTE is comprised of several major subsystems that collectively form an OpenRTE *universe*, as illustrated in Figure 1. A universe represents a single instance of the OpenRTE system, and can support any number of simultaneous applications. Universes can be *persistent* – i.e., can continue to exist on their own after all applications have completed executing – or can be instantiated for a single application lifetime. In either case, a universe belongs to a specific user, and access to its data is restricted to that user unless designated otherwise.

Implementation of the OpenRTE is based upon the Modular Component Architecture (MCA) [3] developed under the Open MPI project. Use of component architectures in high-performance computing environments is a relatively recent phenomenon [2,9,10], but allows the overlay of functional building blocks to dynamically define system behavior at the time of execution. Within this architecture, each of the major subsystems is defined as an MCA *framework* with a well-defined interface. In turn, each framework contains one or more *components*, each representing a different implementation of that particular framework.

Thus, the behavior of any OpenRTE subsystem can be altered by simply defining another component and requesting that it be selected for use, thereby enabling studies of the impact of alternative strategies for subsystems without the burden of writing code to implement the remainder of the system.

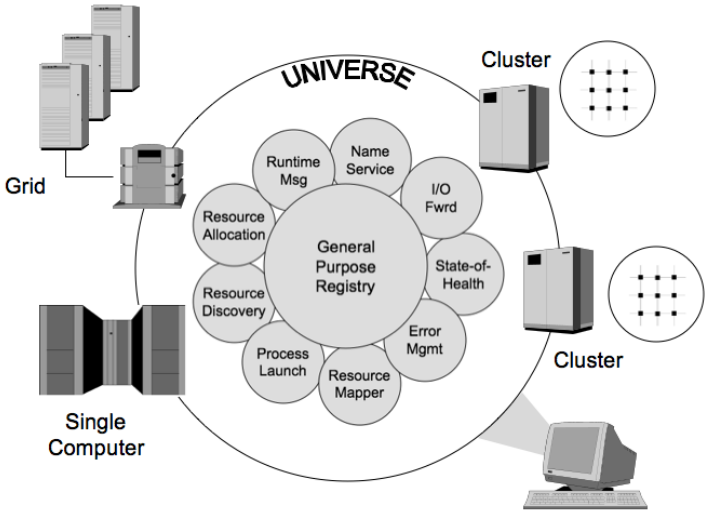

Fig. 1. The OpenRTE architecture

Researchers wishing to study error response strategies, for example, can overlay the standard error manager with their own implementation while taking full advantage of the system-level support from the process launch, state-of-health monitor, and other OpenRTE subsystems.

This design also allows users to customize the behavior of the run-time at the time of application execution. By defining appropriate parameters, the user can direct the OpenRTE system to select specific subsystem components, thus effectively defining system behavior for that session. Alternatively, the user can allow the system to dynamically sense its environment and select the best components for that situation.

OpenRTE's subsystems can be grouped into four primary elements.

General Purpose Registry. At the core of the OpenRTE system is a general purpose registry (GPR) that supports the sharing of data (expressed as key-value pairs) across an OpenRTE universe. Information within the GPR is organized into named *segments*, each typically dedicated to a specific function, that are further subdivided into *containers*, each identified by a set of character string tokens. Collectively, the container tokens, segment names, and data keys provide a searchable index for retrieving data. Users have full access to the system-level information stored on the GPR, and can define their own segments/containers to support their applications.

The GPR also provides a publish/subscribe mechanism for event-driven applications. Users can specify both the data to be returned upon an event, the process(es) and function(s) within the process(es) to receive the specified data, and combinations of actions (e.g., modification, addition, or deletion of data entries) that define the event trigger. Notification messages containing the specified data are sent to the recipients as asynchronous communications via the OpenRTE messaging layer (described below).

Resource Management. Four independent, but mutually supportive, subsystems collectively operate to manage the use of resources by applications within the OpenRTE system. Together, these subsystems provide services for resource discovery, allocation, mapping, and process launch.

True to its name, the *Resource Discovery Subsystem* (RDS) is responsible for identifying the computational resources available to the OpenRTE system and making that information available to other subsystems. The RDS currently contains two components: one for reading hostfiles in several formats covering the common MPI implementations. Hostfiles are typically generated by a specific user and contain information on machines that might be available to that user; and another that obtains its information from a system-level resource file containing an XML-based description of the cells known to the OpenRTE system. Information from each component in the RDS is placed on the GPR for easy retrieval by other subsystems within that universe.

The *Resource Allocation Subsystem*(RAS) examines the environment and the command line to determine what, if any, resources have already been allocated to the specified application. If resources have not been previously allocated, the RAS will attempt to obtain an allocation from an appropriate cell based on information from the RDS. Once an allocation has been determined, the RAS constructs two segments on the GPR – a node segment containing information on the nodes allocated to the application, and a job segment that holds information on each process within the application (e.g., nodename where the application is executing, communication sockets, etc.).

Once resources have been allocated to an application, the application's processes must be mapped onto them. In environments where the cell's resource manager performs this operation, this operation does not require any action by the OpenRTE system. However, in environments that do not provide this service, OpenRTE's *Resource Mapping* (RMAP) subsystem fills this need.

Finally, the *Process Launch Subsystem* (PLS) utilizes the information provided by the prior subsystems to initiate execution of the application's processes. The PLS starts by spawning a *head node process* (HNP) on the target cell's frontend machine. This process first determines if an HNP for this user already exists on the target cell and if this application is allowed to connect to it – if so, then that connection is established. If an existing HNP is not available, then the new HNP identifies the launch environment supported by that cell and instantiates the core universe services for processes that will operate within it. The application processes are then launched accordingly.

Error Management. Error management within the OpenRTE is performed at several levels. Wherever possible, the condition of each process in an application is continuously monitored by the *State-of-Health Monitor* (SOH)[1]. The SOH subsystem utilizes its components to field instrumentation tailored to the local environment. Thus, application processes within a BProc environment are monitored via the standard BProc notification service. Similarly, the SOH might

[1] Some environments do not support monitoring. Likewise, applications that do not initialize within the OpenRTE system can only be monitored on a limited basis.

monitor application processes executing on standalone workstations for abnormal termination by detecting when a socket connection unexpectedly closes.

Once an error has been detected, the *Error Manager* (EMGR) subsystem is called to determine the proper response. The EMGR can be called in two ways: locally, when an error is detected within a given process; or globally, when the SOH detects that a process has abnormally terminated. In both cases, the EMGR is responsible for defining the system's response. Although the default system action is to terminate the application, future EMGR components will implement more sophisticated error recovery strategies.

Support Services. In addition to the registry, resource management, and error management functions, the OpenRTE system must provide a set of basic services that support both the application and the other major subsystems. The *name services* (NS) subsystem is responsible for assigning each application, and each process within each application, a unique identifier. The identifier, or *process name*, is used by the system to route inter-process communications, and is provided to the application for use in MPI function calls.

Similarly, the *Run-time Messaging Layer* (RML) provides reliable administrative communication services across the OpenRTE universe. The RML does not typically carry data between processes – this function is left to the MPI messaging layer itself as its high-bandwidth and low-latency requirements are somewhat different than those associated with the RML. In contrast, the RML primarily transports data on process state-of-health, inter-process contact information, and serves as the conduit for GPR communications.

Finally, the *I/O Forwarding* (IOF) subsystem is responsible for transporting standard input, output, and error communications between the remote processes and the user (or, if the user so chooses, a designated end-point such as a file). Connections are established prior to executing the application to ensure the transport of *all* I/O from the beginning of execution, without requiring that the application's process first execute a call to MPI_INIT, thus providing support for non-MPI applications. IOF data is usually carried over the RML's channels.

5 Summary

The OpenRTE is a new open-source software platform specifically designed for the emerging petascale computing environment. The system is designed to allow for easy extension and transparent scalability, and incorporates resiliency features to address the fault issues that are expected to arise in the context of petascale computing. A beta version of the OpenRTE system currently accompanies the latest Open MPI release and is being evaluated and tested at a number of sites.

As an open-source initiative, future development of the OpenRTE will largely depend upon the interests of those that choose to participate in the project. Several extensions are currently underway, with releases planned for later in the year. These include several additions to the system's resource management and fault recovery capabilities, as well as interfacing of the OpenRTE to the Eclipse

integrated development environment to allow developers to compile, run, and monitor parallel programming applications from within the Eclipse system.

Interested parties are encouraged to visit the project web site at http://www.open-rte.org for access to the code, as well as information on participation and how to contribute to the effort.

Acknowledgments

This work was supported by a grant from the Lilly Endowment, National Science Foundation grants 0116050, EIA-0202048, EIA-9972889, and ANI-0330620, and Department of Energy Contract DE-FG02-02ER25536. Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. This paper was reviewed and approved as LA-UR-05-2718. Project support was provided through ASCI/PSE and the Los Alamos Computer Science Institute, and the Center for Information Technology Research (CITR) of the University of Tennessee.

References

1. A. Geist et all. MPI-2: Extending the Message-Passing Interface. In *Euro-Par '96 Parallel Processing*, pages 128–135. Springer Verlag, 1996.
2. D. E. Bernholdt et. all. A component architecture for high-performance scientific computing. *to appear in Intl. J. High-Performance Computing Applications*.
3. E. Gabriel et all. Open MPI: Goals, concept, and design of a next generation mpi implementation. In *11th European PVM/MPI Users' Group Meeting*, 2004.
4. R.T. Aulwes et all. Architecture of LA-MPI, a network-fault-tolerant mpi. In *18th Intl Parallel and Distributed Processing Symposiun*, 2004.
5. G. Fagg and J. Dongarra. HARNESS Fault Tolerant MPI Design, Usage and Performance Issues. *Future Generation Computer Systems*, 18(8):1127–1142, 2002.
6. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.
7. E.P. Kronstadt. Petascale computing. In *19th IEEE Intl Parallel and Distributed Processing Symposium*, Denver, CO, USA, April 2005.
8. Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.
9. J.M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *10th European PVM/MPI Users' Group Meeting*, 2003.
10. V. Sunderam and D. Kurzyniec. Lightweight self-organizing frameworks for metacomputing. In *11th International Symposium on High Performance Distributed Computing*, Edinburgh, UK, July 2002.