Common Runtime Support for High-Performance Parallel Languages

Parallel Compiler Runtime Consortium *

Geoffrey C. Fox, Sanjay Ranka, Michael Scott, Allen D. Malony,
Jim Browne, Marina C. Chen, Alok Choudhary, Thomas Cheatham,
Jan Cuny, Rudolf Eigenmann, Amr Fahmy, Ian Foster,
Dennis Gannon, Tom Haupt, Mike Karr, Carl Kesselman,
Chuck Koelbel, Wei Li, Monica Lam, Thomas LeBlanc,
Jim Openshaw, David Padua, Constantine Polychronopoulos, Joel Saltz,
Alan Sussman, Gil Weigand, Kathy Yelick

1 Introduction

Parallel Computers have recently become powerful enough to outperform conventional vector based supercomputers. Several parallel languages are currently under development for exploiting the data and/or task parallelism available in the applications. In this report, we propose the development of a basic public domain infrastructure to provide runtime support for high level parallel languages. This would support several projects developing different compilers for a given language such as C++, ADA, or Fortran but also give a unified support for compilers of different languages. There are two particularly important motivations for this common runtime support system.

Firstly, it will accelerate the development of new compiler projects investigating particular modules or concepts by providing a public domain infrastructure which can be built on and not replicated.

Secondly there is currently no universally "best" language; each excels in different aspects of the performance, expressivity, reliability, user familiarity and other metrics. This fact is corroborated by the findings of the recent multiagency workshop on HPCC and grand challenge applications at Pittsburgh. A typical example of software development involved using C++ as a high level language to achieve modularity, Fortran as a high performance assembly language for coding the computationally intensive fragments, and using AVS for visualization and some coarse grain software integration. Thus integrated support of different languages appears an essential pragmatic feature of high performance computing environments.

The above issues were discussed by several researchers which led to a workshop at Syracuse University on common runtime support for compilers and formation of the Parallel Compiler Runtime Support Consortium. Three central and relatively orthogonal topics were identified for common runtime support:

1. Common Runtime Support for Data parallelism
2. Common Runtime Support for Task parallelism
3. Performance and Debugging Infrastructure for Compiler Runtime Systems

Data parallelism and Task parallelism are two important kinds of exploitable parallelism available in most applications. The need for debuggers and performance estimation is of utmost importance for any software environment.

The parallel runtime compiler consortium was originally put together on the initiative of Gil Weigand. The current members of the consortium represent many of the major compiler groups supported by ARPA. The purpose of this report is to present important issues in providing a common framework for runtime support of compilers. The report is organized into three general parts, corresponding to the above three topics. Each part corresponding to the above three topics. Each part represents the discussions of a working group and provides a detailed analysis of the issues, implications and organization required for a common runtime support. The working groups were

*The workshop at Syracuse was sponsored by DARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

lead by Sanjay Ranka, Michael Scott and Alan Malony respectively.

Details and background of this report can be accessed via anonymous ftp from minerva.npac.syr.edu

2 Data Parallelism

Recently there have been major efforts in developing programming language and compiler support for parallel machines. For example, High Performance Fortran has been standardized. A similar effort is currently in progress for HPC++. We use the term High Performance Language (HPL), to refer to HPF, HPC++, an extended (data parallel) form of ADA, or some other relevant language.

A system that would allow different components, perhaps written in various HPLs, to operate with each other and execute in an integrated fashion is sorely needed for the following reasons: (1) different pieces of an application program in one HPL may be best handled by different runtime components (e.g. program segments with regular data access patterns versus irregular access patterns); (2) different components may be best written in one or more HPLs due to the nature of the components and the particular types of language support (e.g. Ada/HPF combination); (3) building components that are reusable across different applications, perhaps written in different HPLs; (4) sharing of infrastructure (data structures, intermediate forms, etc.) across systems.

We believe that there is a great deal of commonality in the support for parallelism in these languages, since parallelism is inherent in the problem and not in the problem's representation in a particular HPL. We should develop a unified framework for integrating and accommodating different program transformation and runtime components for supporting data parallelism. The runtime components developed will be available in the public domain. This will allow groups to build and test compiler subsystems and will accelerate research and development in this area.

The following is a summary of important research issues and innovations that would result from designing such a unified framework:

- *Portable and Scalable Multi-platform Runtime Support*

  Runtime support must efficiently support the address translations and data movements that occur when one embeds a globally indexed program onto a multiple processor architecture. Compilers and runtime support for HPLs can be built in a way that assumes the availability of multiple independent processors and an interface to a message passing system (such as PVM, Express, proprietary vendor message passing systems, MPI, etc.). Alternately compilers and runtime support can assume the existence of hardware supported address translation and data migration mechanisms, such as those found on Kendall Square KSR-1 machines. The issue there will be purely figuring out how data should be migrated.

  We expect that all HPL compilers will make use of at least some optimizations for reducing communication costs such as message blocking, collective communication, message coalescing, aggregation and latency hiding. Prototype runtime support has been developed to carry out these optimizations in the contexts of structured, adaptive, block structured and tree structured problems [?, ?, ?, ?, ?, ?, ?]. We will develop an integrated runtime support system that carries out address translation and communication optimizations, this runtime support will be built on top of a message passing interface.

  We will also develop versions of common runtime support to take advantage of hardware supported distributed shared memory mechanisms. HPL data structure decompositions and processor mappings will make it necessary to carry out rather complex mappings between logical program addresses and locations in the machine's distributed memory. Given these complex mappings, we do not expect hardware supported distributed shared memory alone to be able to efficiently handle data migration and address translation. Instead, we will develop runtime support capable of leveraging the capabilities of hardware supported distributed shared memory.

- *Methodology for Integrated Multilanguage Support*

  We would design and develop common code and data descriptors, and libraries and routines which operate on them for supporting data parallelism in HPLs. This would allow different programming languages to share data structures that are distributed across the memory hierarchy of scalable parallel systems and operate upon them.

  We would design a common compiler data movement interface specification that will provide a set of communication standards that compilers can link into the runtime system for applications. Unlike the user level message passing interface standard, the compiler interface can be more extensive in its capabilities, ranging from very low level primitives that exploit special hardware properties to very high level primitives directly coupled to the common array and data structure formats. The interface standard will make it possible to write compilers that achieve a much greater efficiency on a wider variety of machines than we can with current user level message passing mechanisms. In addition, a common runtime interface will allow a compiler to be easily adapted to a new machine, and still allow customization in the library implementation to improve performance.

- *Methodology for structuring code and data representations to support extensibility*

  We will develop a methodology for the engineering aspect of the described runtime support to allow ease of use, modification, specialization, and extension. The kind of extension we consider includes support for new distributed data structures, new language features, new runtime system mechanisms and algorithms, and new message passing or distributed shared memory interfaces.

3 Task Parallelism

We define *task* parallelism as parallelism not dictated by the distribution of data structures. It includes the execution of different functions in parallel, as well as the parallelization of loops via mechanisms other than (or in addition to) the "owner computes" rule of HPF, pC++, etc. Task parallelism is common in many existing systems. It is particularly useful for irregular applications. Recent research also suggests that there are important classes of applications that require both task and data parallelism in order to obtain good performance [?, ?, ?].

The requirements of a runtime system for task-level parallelism are different from those for data parallelism. First, there is a need for dynamic creation of tasks or processes. Dynamic load balance is necessary since these tasks generally have very different execution times. Second, the interactions between different tasks can be very complex and need the support of sophisticated synchronization primitives. Finally, to take advantage of locality of reference, it is important to dynamically cache and replicate data. The runtime system must provide support for processes to locate data in the distributed address space and to manage the local memory.

We recommend that research efforts in task-parallel runtime systems be combined to build common runtime infrastructure. The infrastructure would be built in layers, and all layers would be accessible to top-level clients. The infrastructure should run on a variety of high performance parallel machines, including cache-coherent multiprocessors like DASH or the KSR-1, NUMA machines like the Cray T3D, and distributed-memory multicomputers like the Intel Paragon or the Thinking Machines CM-5. It should support high level parallel languages such as CC++ [?], Jade [?], and Fortran M [?], as well as parallelizing compilers that generate multithreaded or task parallel code [?, ?, ?, ?, ?, ?]. Prototypes of many of the layers we envision already exist, so the implementation effort should be manageable.

A common runtime infrastructure for task parallelism would have the following benefits:

- Provide a machine-independent layer for portability across machines. This will leverage the lower level system construction currently being done by individual groups.

- Enable shared efforts, both within the group of developers and for external groups that currently lack the resources to build portable runtime systems.

- Encourage better software design through the definition of interfaces between pieces of software.

- Provide validation of results by facilitating comparisons between different approaches on a common software architecture.

- Allow for inter-operability between different runtime systems. With an open layered architecture, compiler writers would be able to access whichever level provides appropriate functionality.

- Enable the comparative study of multiple programming paradigms and multiple machine architectures. Because top-level clients will run on a common substrate, which in turn runs on many machines, "apples and apples" comparisons between languages and compilers will be considerably easier, as will comparisons between machines.

- Provide a framework for identifying commonality in runtime systems built for ostensibly different environments (e.g. on different hardware, or for different languages). Beyond the common facilities described in this report, it is likely that additional opportunities for standardization will be found as research progresses, e.g. in the area of scheduling policies.

There are currently a number of efforts to develop task-parallel runtime systems for a variety of high-level programming languages, such as CC++ [?], Jade [?], Natasha [?], and Fortran M [?]. In addition, several groups are developing parallelizing compilers that recognize implicit task parallelism in sequential programs [?, ?, ?, ?, ?, ?]. These efforts have resulted in runtime software for a large set of machines, but because the systems were developed independently, each typically runs on only one or two machines. We recommend the initiation of a project to develop a common runtime system to facilitate the implementation of high-level programming languages and compilers that exploit task-level parallelism.

To manage the complexity of such a system, we recommend development of a runtime system architecture consisting of well-defined layers of abstraction. Each layer will be exposed to the user—some compilers may be built only on lower layers whereas others may use a mixture of all layers. In addition, multiple instances of a single layer may exist to permit efficient implementations on different architectures, or to provide a different set of abstractions to higher layers. For example, locality may be achieved by a shared object system, a virtual shared memory layer, or hardware shared memory.

We envision a runtime system with sufficient flexibility to span a wide range of machine architectures, including cache-coherent multiprocessors like DASH or the KSR-1, NUMA machines like the Cray T3D, and distributed-memory multicomputers like the Intel Paragon and the Thinking Machines CM-5.

In describing our system architecture, we separate functions into control and data hierarchies. The control hierarchy provides threads, scheduling, synchronization, and load balancing facilities, while the data hierarchy contains names (addresses), data objects, and object relocation facilities. In practice, of course, control and data management facilities are seldom independent; a single software module is likely to provide a combination of both. Interactions between them include reduction operations, aligning data and control (i.e. scheduling for locality), associating synchronization objects with data objects (to facilitate relaxed consistency) and waiting for prefetch/poststore operations to complete.

We expect there to be substantial commonality in both the control and data hierarchies across the spectrum of architectures and programming paradigms. At the same time, alternative module implementations, and even alternative interfaces, will be needed in certain layers in order to accommodate major architectural differences, or to provide the performance and functionality required by dissimilar programming paradigms. Protocol hierarchies for communication networks provide an instructive analogy. The ISO hierarchy [?] provides a conceptual framework for layered protocols, and Arizona's x-kernel project [?] provides an excellent example of the identification and exploitation of commonality in different protocol stacks.

4 Performance and Debugging Infrastructure

The rapidly evolving state of system, run-time, and application software demands performance evaluation and debugging technology that is portable across diverse implementation platforms, and that can be readily extended to include the results of emerging research. Creating a common performance evaluation and debugging infrastructure that meets these requirements for current application and run-time software implies a research effort with two specific foci:

1. integration of application and run-time software with both extant and proposed performance and debugging analysis systems through the specification and development of software interfaces that isolate the implementation of specific instrumentation and analysis techniques behind software "firewalls," ensuring that instrumented software can be ported to systems with different instrumentation implementations; and

2. application of performance evaluation and debugging techniques during run-time software execution through new, dynamic performance and debugging instrumentation, query, and presentation techniques, enabling the development of adaptive application and run-time software.

No single performance analysis or debugging tool provides all the functionality needed to debug and optimize all software, nor should it; experience has shown that a collection of simpler tools is preferable to a single, complex tool. However, software developers should be able to easily integrate, combine, and analyze data from multiple instrumentation and data analysis tools. At present, this is not possible. The goal of the software integration focus is to provide run-time system software developers a set of standard, high-level interfaces to performance and debugging tools. Without these standard interfaces, individual run-time system projects would likely design and develop performance and debugging software specific to their problem area, rather than deal with the nuances of each tool's use. Not only would these systems be incompatible, they would be unable to exploit cross-domain information (e.g., run-time library and compiler information) in a uniform way. A common platform can be achieved only through the standardization of software interfaces that isolate the implementation of specific performance/debugging instrumentation and analysis techniques behind software boundaries. These interfaces provide an integration veneer which ensures that application and run-time software can be ported to systems with differing performance and debugging implementations. For tool developers, the standard interfaces will provide broad access to performance and debugging software that is compliant with the interface definitions.

Although standard software interfaces support a portable, reusable performance evaluation and debugging infrastructure, the requirements posed by emerging software systems challenge existing performance and debugging technology. Run-time systems for high-level languages (e.g., for HPF and HPC++); environments for creating and accessing parallel, distributed data structures; and software for adaptive application execution and run-time decision analysis will all require new performance and debugging techniques, particularly for dynamic instrumentation, run-time queries, dynamic guidance, and execution state access. The present opportunity to develop new performance and debugging techniques in concert with run-time software is unique. Exploiting this opportunity will maximize the likelihood that the resulting software will be well-targeted, quickly applied, and reused by future run-time system development efforts.

The Performance Evaluation and DebuggInG softwaRE infrastructurE (PEDIGREE)[1] research project will create a portable, extensible performance evaluation and debugging infrastructure, based on the research foci above, that is broadly applicable to both run-time libraries and application software. In particular, the PEDIGREE infrastructure will include the following key components:

[1] The PEDIGREE acronym is intended to imply a common basis for performance and debugging support that will be applicable to all run-time system software.

- standard software interfaces for performance and debugging tools;

- dynamically activated performance instrumentation, application-initiated performance queries, performance-directed decision procedures, and data presentation techniques that allow software developers to guide computations; and

- run-time debugging infrastructure that utilizes techniques for dynamic breakpointing to uniformly support run-time breakpoint management, state and event-based query, and dynamic visualization.

Standard interfaces will allow both instrumented run-time systems and applications to be moved to different parallel systems without porting a particular performance or debugging implementation. In addition, standard interfaces will encourage the development of "meta-tools" that combine data from multiple performance and debugging systems. The primary focus of existing tools is user-level performance analysis and debugging; the new infrastructure will enable run-time systems to access performance and debugging data during their execution and to use this data as input to dynamic decision procedures.

We believe that by delivering these three PEDIGREE components, current and future runtime system and application software developments will more likely utilize common performance evaluation and debugging tools rather than develop specialized software, leading to a sorely needed integration and uniformity of technology in the two areas.

References

[1] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):pp. 159–178, June 1991.

[2] Sandeep Bhatt, Marina Chen, James Cowie, Cheng-Yee Lin, and Pangfeng Liu. Object-Oriented Support for Adaptive Methods on Parallel Machines. *OONSKI'93 Object Oriented Numerics Conference*, Sunriver, Oregon, April 25-27, 1993.

[3] S. Bhatt, M. Chen, C.Y. Lin, and P. Liu. Abstractions for Parallel N-body Simulations. *Proceedings of Scalable High Performance Computing Conference (SHPCC '92)*, Williamsburg, Virginia, April 1992.

[4] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka, and Jhy-Chun Wang. Message Passing Environment Requirements for the Fortran 90D Compiler. Technical Report SCCS-FEB-93, Northeast Parallel Architectures Center at Syracuse University, February 1993.

[5] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems - data copy reuse and runtime partitioning. J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Runtime Environments for Distributed Memory Machines*, pp. 185–220. Elsevier, 1992.

[6] S. Ranka, J.C. Wang, and M. Kumar. Personalized Communication Avoiding Node Contention on Distributed Memory Systems. *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993. To appear.

[7] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley. PARTI primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3(1-4):pp. 73–86, 1992. Papers presented at *the Symposium on High-Performance Computing for Flight Vehicles*, December 1992.

[8] L. A. Crowl, M. Crovella, T. J. LeBlanc, and M. L. Scott. Beyond Data Parallelism: The Advantages of Multiple Parallelizations in Combinatorial Search. TR 451, Computer Science Department, University of Rochester, April 1993. Earlier version published as TR 92-80-06, Oregon State University Department of Computer Science, December 1992.

[9] T. Pratt. Kernel-Control Parallel Versus Data Parallel: A Technical Comparison. In *Proceedings of the Second Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, pages 5-8, Boulder, CO, 30 September – 2 October 1992. In *ACM SIGPLAN Notices 28*:1 (January 1993).

[10] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Programming Task and Data Parallelism on a Multicomputer. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 20-22 May 1993.

[11] K. M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. California Instititue of Technology, 1992.

[12] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level Machine-Independent Language for Parallel Programming. *Computer*, 26(6):28-38, June 1993.

[13] I. T. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. Preprint MCS-P327-0992, Argonne National Laboratory, June 1992.

[14] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua. Restructuring Fortran Programs for Cedar. *Proceedings of the 1991 International Conference on Parallel Processing*, I, Architecture:57-66, August 1991.

[15] C. D. Polychronopoulos and others. Parafrase-2: A Multilingual Compiler for Optimizing, Partitioning, and Scheduling Ordinary Programs. In *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[16] W. Li and K. Pingali. Access Normalization: Loop Restructuring for NUMA Compilers. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285-295, Boston, MA, 12-15 October 1992.

[17] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, 21-25 June 1993.

[18] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, 21-25 June 1993.

[19] D. Padua and R. Eigenmann. Polaris: A New Generation Parallelizing Compiler for MPPs. Technical Report 1306, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1993.

[20] L. A. Crowl and T. J. LeBlanc. Control Abstraction in Parallel Programming Languages. In *Proceedings of the International Conference on Computer Languages*, pages 44-53, Oakland, CA, April 1992.

[21] A. S. Tanenbaum. Network Protocols. *ACM Computing Surveys*, 13(4):453-489, December 1981.

[22] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao. The *x*-Kernel: A Platform for Accessing Internet Resources. *Computer*, 23(5):23-33, May 1990.

[23] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19-31, Litchfield Park, AZ,

3-6 December 1989. In *ACM SIGOPS Operating Systems Review 23*:5.