Report PCRC-001, July 1993

Common Runtime Support for High-Performance Parallel Languages
Parallel Compiler Runtime Consortium

Geoffrey C. Fox, Syracuse University, Chair

Sanjay Ranka, Syracuse University, Deputy Chair

Jim Browne, University of Texas

Marina C. Chen, Yale University

Alok Choudhary, Syracuse University

Thomas Cheatham, Harvard

Jan Cuny, University of Massachussettes

Rudolf Eigenmann, University of Illinois at Urbana-Champaign

Amr Fahmy, Harvard University

Ian Foster, Argonne National Labs

Dennis Gannon, Indiana University

Tom Haupt, Syracuse University

Mike Karr, SEI

Carl Kesselman, California Institute of Technology

Chuck Koelbel, Rice University

Wei Li, University of Rochester

Monica Lam, Stanford University

Thomas LeBlanc, University of Rochester

Allen D. Malony, University of Oregon

Jim Openshaw, ARPA/CSTO

David Padua, University of Illinois

Constantine Polychronopoulos, University of Illinois

Joel Saltz, University of Maryland

Michael Scott, University of Rochester

Alan Sussman, University of Maryland

Gil Weigand, ARPA/ISTO

Kathy Yelick, University of California at Berkeley

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. Thus integrated support of different languages appears an essential pragmatic feature of high performance computing environment.

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 the 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 represents the discussions of a working

group and provides a detailed analysis of the issues, implications and organization required for a common runtime support.

This is the first major public report produced by the consortium and will be circulated broadly for comments. We expect other groups will join the consortium. We have an electronic repository of working material which is described in the next section.

Documents and Organization

This document (PCRC-001) is available via anonymous ftp at minerva.npac.syr.edu under the pcrc directory. Electronic mail can be addressed to pcrc@npac.syr.edu. Please mail to pcrc-request@npac.syr.edu for adding your name to the pcrc list. Geoffrey Fox is currently the Chair and Sanjay Ranka is the Deputy Chair of the Consortium. The subgroups are coordinated by the following individuals:

1. Runtime Support for Data Parallelism - Sanjay Ranka (ranka@top.cis.syr.edu)

2. Runtime Support for Task Parallelism - Michael Scott (scott@cs.rochester.edu)

3. Performance and Debugging Infrastructure - Allen Malony (malony@cs.uoregon.edu)

For any questions/comments please contact Sanjay Ranka at 315-443-4457 or ranka@top.cis.syr.edu.

Common Runtime Support for Data Parallelism

Parallel Compiler Runtime Consortium: IP1 Subgroup

Marina Chen, James Cowie, Alok Choudhary, Amr Fahmy, Geoffrey Fox,
Dennis Gannon, Chris Goldthrope, Tom Haupt, Chuck Koelbel, Wei Li,
Sanjay Ranka, Joel Saltz, Alan Sussman

1.1 Research Summary

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.

At a recent multiagency workshop on HPCC and grand challenge applications at Pittsburgh, the application scientists were asked for the software requirements for solving their applications. There was a consensus that no one particular language was sufficient for the parallelization of their applications. 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. The codes involved a mixture of data parallelism as well as task parallelism. Many of the applications required parallelization of irregular and unstructured problems which could not be easily or efficiently represented with the current features provided by High Performance Fortran. Thus, for these applications, the most natural programming language for every user/application may be any of HPF, HPC++, an extended form of ADA, etc.

There is ongoing work on developing libraries with specialized capabilities for parallel machines. Some examples include SCALAPACK [11] for linear algebra, P++ and G++ [12], PTREE [4, 5] for distributed data structures, etc. An application scientist would like to take advantage of these specialized libraries to solve their applications and develop new libraries

which could be useful for others.

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 would 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.

1.2 Research Issues and Work Statement

Our work would be focused on developing a portable and scalable Common Integrated Multiplatform Runtime Support system. A high level description of the important tasks is given

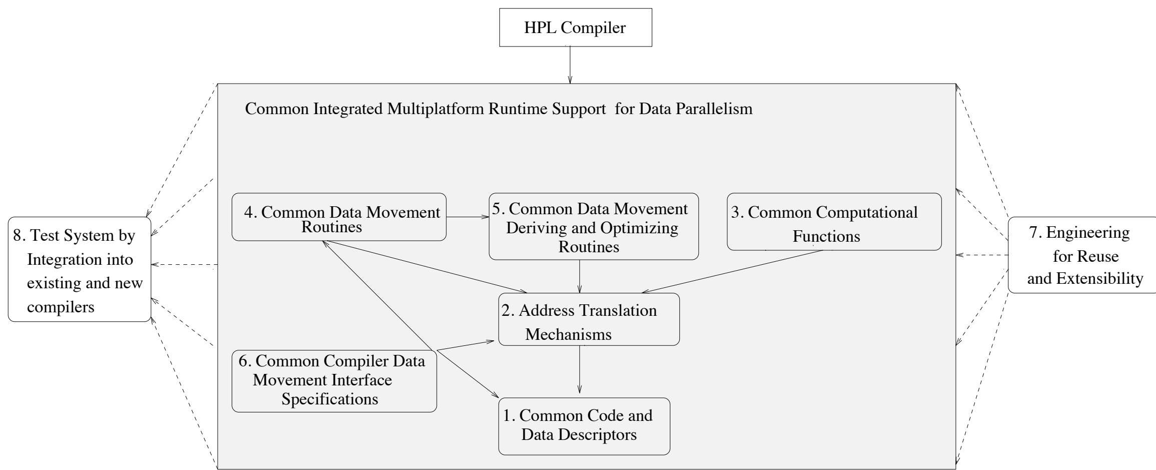

Figure 1.1: Overview of the Common Runtime System

in Figure 1.1. In the following subsections, we describe the important issues and the corresponding work required for each of the tasks.

1.2.1 Common Code and Data Descriptors

Because of the scale and complexity of grand challenge applications that will be ported to high performance machines, HPC programs written in Fortran 90, C++, Ada, and other relevant languages will need to work together. This implies that an HPF program may need to call routines and libraries written in HPC++ and vice versa. The challenge for the compiler implementer is to make certain that array and structure data types have a run-time representation that is consistent between languages and, if possible, across machines. In the case of HPF this is a non-trivial problem. Array alignment and distribution information passed from the programmer to the compiler by HPF annotations must be encoded into the runtime structure of the array so that a software module written in another language can understand the data placement. Consequently, data access protocols as well as access functions should be shared.

The most challenging problem to be solved is how to build the data access descriptors and functions in a "platform neutral" manner. Some vendors can be expected to have complete and near optimal solutions to this problem at the time of machine introduction. However, experience dictates that most will only have partial solutions. A platform-neutral solution is one that supplies only access policy mechanisms and not specific bit-by-bit data layout rules and operations. Consequently, it will be possible to define an implementation for each machine that best exploits the hardware of that machines but still provides one coherent view

of the data to the different language implementations.

1.2.2 Address Translation Mechanisms

Given the descriptors for distributed arrays, the runtime system must still have standardized methods of using them to produce address translations. In essence, this defines a semantics for the syntax of the descriptors. An important subproject will therefore be to develop a standard interface for address translation that will be closely tied to the distributed array descriptors. This interface will include:

- Formulas and procedures for finding the processor storing a particular global index (i.e. an array reference made relative to the full array).
- Formulas and procedures for translating global indexes into local indices (i.e. an offset relative to the section of an array stored on a particular processor).
- Formulas and procedures for translating a local index to a global index.
- Procedures for querying attributes of a global or local index, such as whether that array element is replicated on other processors.

This will form a common basis for compilers to represent array references, thus allowing higher-level operations to be defined. It is important to note that compilers may optimize the formulas in the actual implementation, and thus may not appear to use them directly. However, these address translation mechanisms will ensure that there is agreement on data layout and addressing conventions, thus avoiding integration problems that may exist today.

Additional runtime support is required for irregular data distributions. For distributed memory machines, we will work on table based address translation schemes, such as the paged translation tables used in the Parti routines for irregular problems citedas93c. Also, we will investigate techniques for handling address translation for hybrid regular/irregular distributions. For example, we will look at the issues that arise in translating addresses for a multi-dimensional array that is irregularly distributed in one dimension and block distributed in another dimension.

For machines that provide hardware support for distributed shared memory address translation, we will look at the usefulness of techniques for addressing irregularly distributed data that have already been developed for distributed memory machines [19]. However, such techniques should improve the performance of the hardware address translation mechanisms, by improving locality (i.e. making more data accesses local, rather than requiring data located

elsewhere). For example, on a KSR machine, address translation optimization should be performed with the KSR subpage and page structure in mind, and data migration should take advantage of the KSR hardware support.

1.2.3 Common Computational Functions

Several basic computational operations are very valuable in parallel algorithm design for massively parallel machines. HPF adds several classes of parallel operations to those in Fortran 90, either as intrinsics or as standard library functions. We have developed a library of routines for a subset of HPF intrinsic functions for different data distributions [2].

We would develop a library of common computational functions that uses the common code and data descriptors described above.

1.2.4 Common Data Movement Routines

Efficient parallelization of data parallel constructs on distributed memory machines requires the use of collective communication to access non-local data. This communication could be structured (like shift, broadcast, all-to-all communication) or unstructured.

We have developed routines which remap data between regularly and irregularly distributed arrays [15]; converting arrays from one data layout to another [8]; carry out structured collective communication patterns such as shift, transpose, spread, scans, etc. [6]; scheduling and carrying out unstructured collective communication [17, 18]; carry out the communication needed to fill in overlap regions or ghost cells [1, 6]; carry out the communication associated with moving regular sections within a particular distributed array or between two conforming or non-conforming distributed arrays [1]; carry out the dynamic data partitioning and communication patterns associated with many tree structured problems [4, 5].

We will develop a common library of collective structured and unstructured data movement routines. Experimentation and experience gained from development of these routines would be useful in the development of the Common Compiler Data Movement Interface (to be described later).

1.2.5 Common Data Movement deriving and optimizing routines

We will develop a common set of routines that optimize the data movement required by various data distribution schemes at runtime. These routines will help provide efficient implementations of the data movement primitives for different communication patterns required by the routines in the common data movement library (see the previous subsection). Such routines would use runtime techniques to optimize communication for regular, irregular, block

structured, tree structured and other types of problems. Since the common runtime interface is targeted at multiple machines, the runtime techniques may include both machine independent and dependent optimizations, and provide efficient implementations for both message passing and distributed shared memory machines. The routines for regular and block structured problems would be based on existing work at Syracuse and Yale for the Fortran90D compiler [6, 13, 7] and Maryland for the multiblock Parti library [1, 20]. The routines for problems with completely irregular data distributions would be derived from the Parti routines [3, 9]. Routines for tree structured communication distributions would be drawn from work carried out at Yale [4, 5].

We will also use the results of ongoing research to develop runtime support that implements optimizations that target communication patterns characterized by runtime dependencies patterns. In runtime dependent communication patterns, communication cannot occur concurrently as dependency conditions constrain the order in which communication can be carried out. Development of efficient communication optimizations for runtime dependent communication patterns is an active area of current research. A joint project [14] by Yale and IBM tackling the problem of deriving such runtime dependent parallelization and communication optimizations by new compile-time analysis techniques and a compiler-generated runtime "scheduler generator". Optimizations targeted at tree structured problems have been carried out at Yale [4, 5]. Also, a joint Maryland, Rutgers, and IBM project has investigated scheduling of runtime dependent communication patterns in the context of sparse factorization methods [21].

We will develop a common interface for the partitioning (and repartitioning) of irregularly distributed data structures. The partitioners will use a (distributed) representation of information that is commonly used to generate irregular data structure partitions. In general, data structures for irregularly distributed data are defined in the form of a graph. The information contained in such graphs that can be used for partitioning can be limited to one or more of the following: 1) the connectivity of the graph representing dynamic dependence information, 2) the weights of the nodes of the graph representing the computational load, and 3) a static graph representing the physical geometry of the computation. Our interface will also be defined in a way that makes possible encoding of runtime data dependency information. An interface will be developed both for the partitioning algorithms (to define both the input and output data for a partitioner), and for the support routines for moving data associated with partitioned graphs. Also, the data structures required for a common interface will be defined. This definition will draw on work associated with the generic **GeoCoL** graph (which allows the use of **G**eometry, **C**onnectivity and **L**oad information) for the Parti routines [15], the IDG (iteration dependency graph) representation for the scheduler generator [14], and on the

results of ongoing work by the Yale, Maryland, Syracuse and Rice groups on data structures which encode runtime dependency information [4, 5, 19].

Tree structures are useful in supporting many applications such as N-body simulations. Runtime support for tree operations can be divided into two interacting subsystems, *link traversal* and *per-node computation.* A tree traversal begins as a single-link traversal to the logical root, following which control alternate between single-link traversal and per-node evaluation until the traverse is complete. A third subsystem, the *mailbox module*, supports the sending of raw or typed data between any two tree nodes, using some application-supplied global addressing methods. These primitive operations can be used for developing more complex application-specific protocols (e.g. PUT and GET [5]) which may involve multiple synchronization and communication phases. A fourth subsystem, distribution module, provides key services to the other three, resolving logical link traversals and global mailbox addresses to either local pointers or remote processor/pointer pairs. Data distribution, adaptive load balancing and optimization of physical communication are the responsibility of the distribution module.

More concretely, the work to be done consists of:

- Runtime support for communication optimizations targeted at block structured problems and regular problems in which the number of processors and array dimensions are not bound until runtime—carry this out for one distributed memory architecture (e.g. Paragon, CM-5, SP-1) and one distributed shared memory architecture (e.g. KSR-1, T3D)

- Optimizations for concurrent irregular communication patterns, including implementation of software caching methods that recognize when multiple copies of the same off-processor data item are being requested. Carry this out for one distributed memory and one distributed shared memory architecture.

- Optimizations for communication characterized by runtime determined dependency patterns. Draw information from ongoing research projects in this area, evaluate and compare different approaches to scheduling runtime dependent information. Develop a small set of primitives capable of covering different communication graph characteristics (e.g. tree, grid shaped directed acyclic graph). Implement for one distributed memory and one distributed shared memory architecture.

- Standardization of common interface for runtime support employed in partitioning irregular data structures used in concurrent irregular problems. Implementation using standard interface in distributed memory and distributed shared memory architecture.

- Standardization of common interface for runtime support employed in partitioning data structures described by runtime data dependencies. Implementation using standard interface in distributed memory and distributed shared memory architecture.

1.2.6 Common Compiler Data Movement Interface Specifications

Application programs must be portable across all HPCC platforms. Consequently it is important that users have a message passing interface for distributed memory systems that hides architectural details. A number of standard user level communication libraries exist. These include the MPI standard, Express, PVM and the BLACS. All are well designed for human use: they have special error and type checking and their features are limited for portability reasons.

Compilers, on the other hand, are tuned to exploit the characteristics of individual machines. Compilers do global optimizations that exploit knowledge of hardware details and they generate code that is tied to specific architectures and low level data structure descriptors that humans never see. Consequently, the compiler can generate code for a wide spectrum of data movement operators that optimize performance at the expense of object code portability.

This subproject will organize an on-going effort to define and document communication library primitives for use at the compiler level. We will consider a spectrum of data movement functions. At the lowest level will be specific classes of primitives that can be shown to work well on a subclass of machines, but perhaps not all. For example, support is needed for small, "active message" style communication which can exploit very low latency networks. Similarly, there are the special functions that move blocks of data, like pages in a virtual shared memory environment, that must co-operate with the operating systems. A working group will be established to define these functions to make sure they are well documented and they can be used in the same environment without interfering with each other.

A broader topic that requires more work is the definition of the common data movement functions that operate on the internal data descriptors of HPF, HPC++ and ADA arrays and other data structures. These functions are key to the performance of all HPCC languages and they include not only data movement but communication based reduction and parallel prefix operations. Because they are intimately connected to the common data descriptor definition project, the working groups that define them will probably overlap. These functions also form the lowest layer in the Basic Linear Algebra Communication Subroutines (BLACS) defined by Oak Ridge. Consequently they are essential to the HPC Library community and they must be included in the specification and design effort.

1.2.7 Engineering the Runtime System for Reuse and Extensibility

To achieve the objective of a large degree of sharing and reuse of the common runtime system modules, the modules must be well-engineered to allow ease of use, modification, specialization, and extension. The kind of extension we consider includes the support for new distributed data structures, new language features, new runtime system mechanisms and algorithms, and new back-ends.

The first requirement is a modular design of the runtime system, separating the system into well-defined subsystems with clean and well-documented interfaces. The internal data structures and algorithms of a module also need to be well-documented to allow modification. The data structures themselves need to be specified in an interface language for ease of modification and extension. Modularity needs to be maintained at all levels, from top level system design to the internal structure of a particular algorithm.

The second requirement is a collection of mechanisms to deal with the complexity of modification, specialization, and extension. New modules will be added with special attention to reusing existing modules; modification or specialization of existing code should preserve both internal and external modularity. A new module can be added without the researcher learning the details of the entire system. Replacing a functionally equivalent module should not disturb the rest of the system. It seems that object-oriented methodology should be employed here to take advantage of the highest degree of sharing and reuse and the least amount of duplicated effort. Version control and facilities for browsing the system should also be there to aid the users.

The engineering aspect of the common runtime system effort will be an on-going process while we perform the integration and reorganization based on an existing repertoire of runtime modules. Development of new modules should follow the design principles and system organization.

We can consider various different implementation languages for the well-engineered, ease-to-use system. Since most existing system are written in C or C++, it seems reasonable to choose C++ as the implementation language.

1.2.8 Retargeting Current Compilers and Developing New Compiler based on the Runtime Support

We would integrate the above runtime support into our existing compiler work at Indiana, Maryland, Rice, Syracuse and Yale. Harvard would utilize the above runtime support for the development of their new compiler.

Tasks	Indiana	Maryland	Rice	Syracuse	Harvard	Yale	Rochester
1	L	$\surd$	$\surd$	$\surd$		$\surd$	
2	$\surd$	$\surd$	L	$\surd$		$\surd$	
3				L			
4	$\surd$	$\surd$	$\surd$	L		$\surd$	
5		L		$\surd$		$\surd$	
6	L	$\surd$	$\surd$	$\surd$		$\surd$	
7	$\surd$	$\surd$	$\surd$	$\surd$		L	
8	E	E	E	L	N	E	E

Table 1.1: Distribution of work: The Description of each of these tasks can be found in Section refriws-section

$\surd$-Member; L - Leader
E- Existing Compilers; N - New Compilers

1.3 Organization

Table 1.1 gives a description of the involvement of different groups in different tasks. Each task is related to the corresponding subsection in Section refriws-section. The leader is expected to coordinate the activities within the group.

1.4 Timelines

We envision a three year effort for the development of this common runtime support. The timelines described in this section are relative to the time of inception of the effort.

Because of the central nature to much of the run time system of the common code and data descriptors, Task 1 must be completed in the first year of the project. Initial candidate proposals for the common model will be collected in the first month. A working group of the compiler implementors will need to meet (by video conference) weekly until a draft plan is adopted. Initial implementations on each of the base platforms will be designed and implemented by the end of year 1. In the following years, the working group will meet as needed to discuss extensions to structures other than arrays and to provide additional functionality required by other aspects of the runtime system.

Task 2, 3, 4, and 5 would be based on prototype runtime support developed for the

Tasks	I	II	III
1. Common Code and Data Descriptors	1	0.5	0.5
2. Address Translation Mechanisms	2	2	2
3. Common Computational Functions	3	2	2
4. Common Data Movement Routines	2	3	2
5. Common Data Movement deriving and optimizing routines	4	4	4
6. Common Compiler Data Movement Interface Specification	2	1	1
7. Engineering the Runtime System for Reuse and Extensibility	1	1	1
8. Retargeting Current Compilers and Developing New Compilers	1	3	3
Total	16	16.5	15.5

Table 1.2: Relative units of work required

compiler projects at Indiana, Maryland, Rice, Syracuse and Yale. These runtime support prototypes exist in several disjoint libraries; these libraries would be expanded, merged, and designed so that they share the same data access descriptors.

Task 6 will organize an on-going effort to define and document communication library primitives for use at the compiler level. A working group will be established to define these functions to make sure they are well documented and they can be used in the same environment without interfering with each other.

The engineering aspect (Task 7) of the common runtime system effort will be an on-going process while we perform the integration and reorganization based on an existing repertoire of runtime modules. Development of new modules should follow the design principles and system organization.

All of the above would be incorporated at different stages in the compiler efforts at Harvard, Indiana, Maryland, Rice, Syracuse and Yale. Figure 1.4 gives the relationship between different tasks.

1.5 Amount of Effort

Table 1.2 gives an assessment of the relative amount of work required for the different tasks during the different phases of the project.

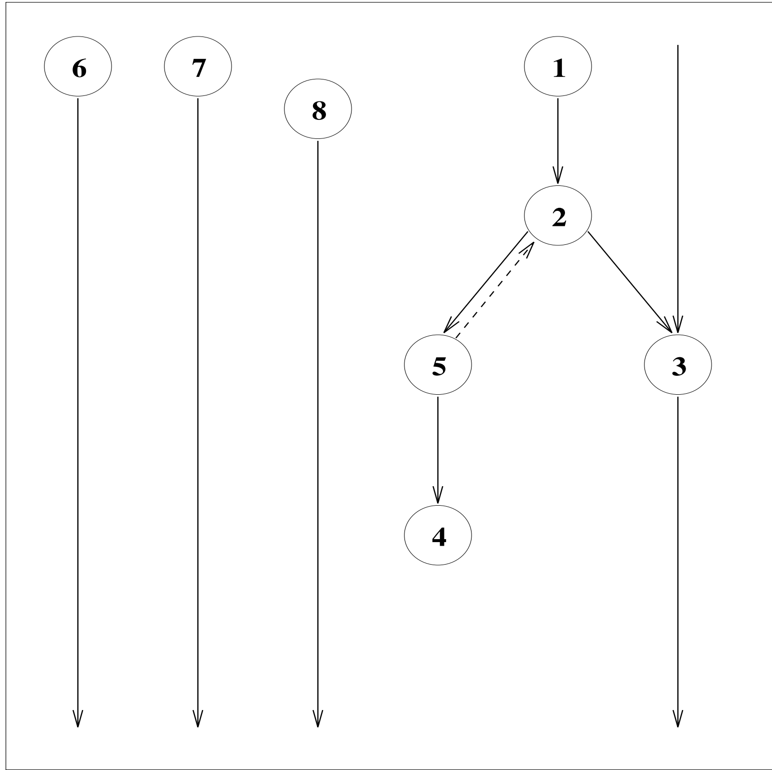

Figure 1.2: Time dependencies between different tasks

Solid Arrow refers to a direct dependency
Dashed Arrow refers to feedback

Bibliography

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block structured applications. Technical Report CS-TR-3080 and UMIACS-TR-93-45, University of Maryland, Department of Computer Science and UMIACS, April 1993. To appear in Supercomputing '93.

[2] Ishfaq Ahmad, Alok Choudhary, Geoffrey Fox, Kanchana Parasuram, Ravi Ponnusamy, Sanjay Ranka, and Rajeev Thakur. Implementation and Scalability of Fortran 90D Intrinsic Functions on Distributed Memory Machines. Technical Report SCCS-256, Northeast Parallel Architectures Center at Syracuse University, March 1992.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] M. Chen and J. Cowie. Prototyping Fortran-90 Compilers for Massively Parallel Machines. *ACM Symposium on Programming Language Design and Implementation (PLDI)*, June 1992, San Fransisco, California.

[8] M. Chen and J. Wu. Optimizing Fortran-90 Programs for Data Motion on Massively Parallel Systems. Technical Report TR-882, Yale University, 1991.

[9] 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.

[10] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. Submitted to *Journal of Parallel and Distributed Computing*, June 1993.

[11] J. Dongarra, R. van de Geijn, and D. Walker. A Look at Scalable Dense Linear Algebra Libraries. *Proceedings of Scalable High Performance Computing Conference (SHPCC-92)*, Williamsburg, Virginia, April 1992.

[12] Max Lemke and Daniel Quinlan. P++, A Parallel C++ Array Class Library for Architecture-Independent Development of Structured Grid Applications. *ACM SIGPLAN Notices*, 28(1):pp. 21–23, September 1992.

[13] Jingke Li and Marina Chen. Compiling communication-Efficient Programs for Massively Parallel Machines. *IEEE Transaction on Parallel and Distributed Systems*, pp. 361-376, July 1991.

[14] Lu and Marina Chen. Parallelizing loops with indirect array references or pointers. *the 4th Workshop on Programming Languages and Compilers for Parallel Computing*, Santa Clara, California, August, 1991. In *Programming Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing series, MIT Press.

[15] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. Technical Report CS-TR-3055 and UMIACS-TR-93-32, University of Maryland, Department of Computer Science and UMIACS, April 1993. To appear in Supercomputing '93.

[16] S. Ranka and J.C. Wang. Static and runtime scheduling of unstructured communication. *International Journal of Computing Systems in Engineering*, 1993. To appear.

[17] S. Ranka, J.C. Wang and G. Fox. Static and runtime algorithms for all-to-many personalized communications on permutation networks. *Proceedings of the 1992 International*

Conference on Parallel and Distributed Systems, pp. 211–218, HsinChu, Taiwan, December 1992.

[18] 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.

[19] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):pp. 603–612, May 1991.

[20] 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.

[21] S. Venugopal, V.K. Naik, and J. Saltz. Performance of distributed sparse Cholesky factorization with pre-scheduling. *Proceedings Supercomputing '92*, pp. 52–61. IEEE Computer Society Press, November 1992.

Common Runtime Support for Task Parallelism

Parallel Compiler Runtime Consortium: IP2 Subgroup

Rudolf Eigenmann, Ian Foster, Carl Kesselman, Monica Lam,
Tom LeBlanc, Wei Li, David Padua, Constantine Polychronopoulos,
Michael Scott, and Kathy Yelick

2.1 Motivation

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 [CCL93, Pra92, SSO93].

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++ [ChK92], Jade [RSL93], and Fortran M [FoC92], as well as parallelizing compilers that generate multithreaded or task parallel code [EHJ91, Poo89, Li92, AmL93, AnL93, PaE93]. 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.

2.2 Technical Framework

There are currently a number of efforts to develop task-parallel runtime systems for a variety of high-level programming languages, such as CC++ [ChK92], Jade [RSL93], Natasha [CrL92], and Fortran M [FoC92]. In addition, several groups are developing parallelizing compilers that recognize implicit task parallelism in sequential programs [AmL93, AnL93, Poo89, EHJ91,

Li92, PaE93]. 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 [Tan81] provides a conceptual framework for layered protocols, and Arizona's x-kernel project [PHO90] provides an excellent example of the identification and exploitation of commonality in different protocol stacks.

2.2.1 Control Hierarchy

The control hierarchy contains the following layers, presented from the bottom to the top:

- C1: A fixed set of kernel-supported processes (typically one per processor), and basic interprocessor communication mechanisms. Depending on hardware architecture, the latter may include load and store instructions, interprocessor interrupts, message passing primitives, or active message handlers [vCG92]. Facilities at this level are provided, for example, by Split-C [DGK93].

- C2: Lightweight local threads, with mechanisms for thread creation, blocking and deletion. Threads need not be preemptable. They may or may not have priorities. Facilities at this level are provided, for example, by Nexus (a common runtime system for CC++ and Fortran M, under development at Argonne and CalTech), and by the low-level parts of Mercury [Kon92].

- C3: Global threads, with mechanisms for scalable synchronization [MeS91] and for moving a thread's state between processors. Migration mechanisms may be implemented on top of an object system like SAM [ScL93], or on a cache-coherent architecture like that of DASH or the KSR-1. Facilities at this level are provided by the high-level parts of Mercury.

- C4: Dynamic load balancing, with thread migration policies. Interactions with the data hierarchy are likely to be required in order to maximize the co-location of threads and the data they use [MaL92].

- C5: Loop scheduling mechanisms, such as those employed by compiler projects at Illinois [EHJ91, PoK87], and Rochester [MaL93].

2.2.2 Data Hierarchy

The data hierarchy contains the following layers, presented from the bottom to the top:

- D1: Hardware-specific distributed address space layer, with interfaces to all pertinent hardware features, including interprocessor communication (significant overlap with layer C1), cache and TLB control, VM fault reflection, fences for weakly consistent memory, etc.

- D2: Static, shared address space layer, as provided by the Split-C and Nexus systems, and the CRAY T3D hardware. Provides load (get) and store (put) primitives, as well as atomic operations, overlapped prefetch/poststore, etc. Compilers may generate code for this layer, and thereby gain portability between machines like the T3D, CM5, and Paragon, as well as simulation testbeds such as Mint [Vee93] or Tango [DGH91].

- D3: Dynamically replicated and cached data layer. May be designed around logical objects, as in SAM [ScL93], or around physical blocks, as in machines with hardware cache coherence or virtual memory-based software coherence [BFS89, CoF89, LaE91, NiL91]. Objects or blocks can be relocated, replicated, and kept consistent using a variety of protocols. Dynamic memory allocation and garbage collection are also at this level.

- D4: Distributed data structure layer such as the Multipol library, which is being developed to support irregular applications [ChY93, WeY93]. Distributed data structures include queues, trees, hash tables, etc. that are spread across a machine using replication or partitioning.

Languages such as Jade, CC++, and Fortran M would make use of the top two levels of the data hierarchy, with object-based coherence and distributed data structures. Compilers such as Parafrase-2 [Poo89], Pnuma [Li92], Polaris [PaE93], and SUIF [AmL93, AnL93] would build on top of the lower, static layers of the data hierarchy. These static layers provide an abstraction that is similar to a shared address space multiprocessor. Recent work suggests [DGK93, vCG92] that this abstraction can be implemented on distributed-memory machines with performance rivaling that of explicit message passing. By employing a global "address" space interface instead of a message-passing interface, we gain the advantage of easier portability to NUMA machines and even cache-coherent machines. This approach stands in contrast to that of the IP2 section of the current report, which is primarily based upon a message-passing interface.

NB: a top-level client based solely on layers D3 and D4 is unlikely to make the best possible use of cache-coherent hardware, and may not make the best possible use of NUMA hardware (in which the cost of fine-grain remote references is lower than on distributed-memory machines). Such a client may wish to use the upper layers for *part* of its work, however; our open architecture allows it to see through those layers when appropriate.

2.3 Suggested Research Plan

The design of appropriate runtime mechanisms and policies for large-scale parallel systems—especially task-parallel systems—is an active topic of research. The field has not yet reached the point where standardization on a single set of runtime interfaces is either feasible or appropriate. Nonetheless, there is much obvious potential commonality among the many systems under development, and serious efforts to reduce duplication of effort and promote portability and interoperability appear to be very much worthwhile. We therefore recommend a three-phase research and development plan designed to produce a common runtime system. In the

first phase, existing software systems are integrated into our hierarchical system architecture and made to work together. In the second phase, the resulting runtime system software is used in demonstration/evaluation projects and disseminated within the community. The resulting experience and feedback is used to guide a redesign of the software, hopefully with greater commonality. In the third phase, the runtime system is reengineered and packaged for broad distribution.

2.3.1 Integrate existing runtime systems

Most existing runtime system components have been designed for a specific programming paradigm and/or machine architecture. While several layers of our control and data hierarchies (e.g. C2) are likely to be more-or-less universally applicable, many of the high-leverage opportunities to integrate existing components are likely to occur among closely-related programming paradigms and architectural classes. As a practical matter, we recommend that researchers

1. identify sets of client compilers and target machines with similar characteristics,
2. implement complete interoperability among members of the same set, so that every client compiler runs on every machine, with common instances of the layers in the middle, and
3. identify individual modules of more widespread utility, for use with dissimilar clients or machines.

This effort would emphasize the use of code from existing runtime systems, many of which have been used extensively and can guide the integration process. Likely sets of clients and machines include:

- Programming paradigms based on logical objects (e.g. Jade, CC++, and Fortran M) on distributed memory machines. Languages such as these are ideally suited to exploit the object-based facilities of the D3 and D4 data layers, which in turn admit an efficient implementation on distributed memory machines.
- Programming paradigms based on parallel loops (e.g. High-Performance Fortran and its dialects, PCF Fortran (ANSI X3h5), and Cedar Fortran [Hoe91, EHJ92]) on machines with a global physical address space. Such machines admit a wide range of parallelizations (more general than owner computes), with hardware remote reference facilities allowing a potentially finer granularity of sharing than is feasible on distributed-memory machines.

2.3.2 Experiment, export, and refine

Once the prototype common runtime libraries have been developed, experiments should be conducted to evaluate their utility and efficiency and to identify their limitations. With this goal in mind, we recommend that researchers

- develop, in collaboration with the team developing performance evaluation tools, instrumentation techniques for the runtime libraries that permit automatic and manual collection of performance data,
- evaluate performance achieved using the common runtime for different compilers and on different real and simulated architectures, and compare with that achieved using native runtime systems where available, and
- attempt to integrate various programming systems, for example task-parallel programming languages and loop-based compilers, so as to identify areas of commonality across programming paradigms and architectures.

It is also important to document the runtime system software and disseminate it within the community, so as to obtain feedback from other users.

Based on the experimentation and feedback, we anticipate researchers designing a refined runtime system that further integrates existing components where commonality has been identified, provides additional functionality where this has been found necessary, and addresses performance issues identified in experimentation.

An important goal of experimentation during this phase of the research should be to study portability issues between distributed-address-space and global-address-space machines. This report envisions common runtime support for cache-coherent machines like the KSR-1, NUMA machines like the T3D, and distributed-memory multicomputers like the CM-5 and the Paragon. The extent to which this commonality can be achieved without compromising performance requires careful study.

On a distributed-memory multicomputer, a compiler must generate messages (or D2-level copy operations implemented via messages) to effect statically-determined data-access patterns, and must generate run-time checks to identify patterns that cannot be recognized at compile time. On a cache-coherent multiprocessor, the messages and checks are unnecessary; the hardware moves things automatically. A NUMA machine falls in-between. Its global address space eliminates the need for messages, though processors must self-invalidate lines in their caches at appropriate times. (Optionally, they can create copies of multi-cache-line data in local memory, to amortize block-movement costs and reduce the cache-miss penalty.)

Likewise, data with statically undetermined access patterns can be accessed remotely (with caching off), though at a loss in performance.

Distributed-memory multicomputers can emulate NUMA machines if provided with a runtime system like Split-C that uses messages to perform the equivalent of loads and stores. Similarly, both distributed-memory multicomputers and NUMA machines can emulate cache-coherent machines if provided with VM-driven software coherence systems. We envision investigating the effectiveness of emulating global addresses on distributed-memory machines by targeting compilers intended for global-address-space machines at the D2 (Split-C) level of the data hierarchy. The compilers would generate puts and gets instead of loads and stores for any simple or compound data not known to be local. The effectiveness of this approach can be evaluated by comparing HPF compilers for the CM-5 and Paragon with a global-address-space HPF compiler. Similarly, given sufficient resources, we would envision investigating the effectiveness of software cache coherence by running code intended for machines like the KSR-1 or DASH on top of a virtual shared memory system.

2.3.3 Reengineer and distribute

The goal of the final phase of the project should be to reengineer software produced in Phase I so as to satisfy the revised design produced in Phase II. This software should be well documented and broadly disseminated. In addition, both the various compilers targeted to the original common runtime and the interoperability experiments conducted in Phase II should be retargeted to this common runtime.

Over time, we believe it is likely that previously-unrecognized areas of commonality across programming paradigms and architectures will emerge. This is particularly likely in areas or runtime functionality that are currently the subject of research. Examples include application-specific scheduling and coherence policies, e.g. to exploit known patterns of synchronization or relaxed consistency requirements.

It is too early to tell exactly how much diversity of runtime functionality will eventually be required. The development of common runtime infrastructure must be an iterative process. As refinements are made, the increasing uniformity of runtime support will facilitate interoperability among application modules written in different programming languages. This interoperability would be possible first among languages with similar programming paradigms. With appropriate interface mechanisms [MSL91], it should be possible among languages with dissimilar paradigms as well.

Bibliography

[AmL93] 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.

[AnL93] 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.

[BFS89] 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.

[ChY93] S. Chakrabarti and K. Yelick. Implementing an Irregular Application on a Distributed Memory Multiprocessor. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 20-22 May 1993.

[ChK92] K. M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. California Instititue of Technology, 1992.

[CoF89] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32-44, Litchfield Park, AZ, 3-6 December 1989. In *ACM SIGOPS Operating Systems Review 23*:5.

[CrL92] 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.

[CCL93] 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.

[DGH91] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor Simulation and Tracing Using Tango. *Proceedings of the 1991 International Conference on Parallel Processing*, II, Software:99-107, August 1991.

[DGK93] A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Supercomputing '93*, Portland, Oregon, to appear, November 1993.

[EHJ91] 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.

[EHJ92] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. The Cedar Fortran Project. Technical Report 1262, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1992.

[FoC92] I. T. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. Preprint MCS-P327-0992, Argonne National Laboratory, June 1992.

[Hoe91] J. Hoeflinger. Cedar Fortran Programmer's Handbook. Technical Report 1157, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1991.

[Kon92] L. Kontothanassis. The Mercury User's Manual. Technical Report DRAFT, Computer Science Department, University of Rochester, September 1992.

[LaE91] R. P. LaRowe, Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319-363, November 1991.

[Li92] 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.

[MaL92] E. P. Markatos and T. J. LeBlanc. Load Balancing Versus Locality Management in Shared-Memory Multiprocessors. In *Proceedings of the 1992 International Conference on Parallel Processing*, St. Charles, IL, August 1992. Also published as TR 399, Computer Science Department, University of Rochester, September 1991.

[MaL93] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, to appear (accepting pending revision), 1993. Earlier version presented at *Supercomputing '92*, and published as TR 410, Computer Science Department, University of Rochester, March 1992.

[MSL91] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-Class User-Level Threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110-121, Pacific Grove, CA, 14-16 October 1991. In *ACM SIGOPS Operating Systems Review 25*:5.

[MeS91] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21-65, February 1991.

[NiL91] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52-60, August 1991.

[PaE93] 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.

[PHO90] 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.

[Poo89] 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.

[PoK87] C. D. Polychronopoulos and D. J. Kuck. Guided Self-scheduling: A Scheduling Scheme for Supercomputers. *IEEE Transactions on Computers*, C-36(4), April 1987.

[Pra92] 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).

[RSL93] 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.

[ScL93] D. J. Scales and M. S. Lam. *A Shared Memory Architecture for Distributed Memory.* Computer Science Department, Stanford University, June 1993.

[SSO93] 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.

[Tan81] A. S. Tanenbaum. Network Protocols. *ACM Computing Surveys*, 13(4):453-489, December 1981.

[Vee93] J. E. Veenstra. Mint Tutorial and User Manual. TR 452, Computer Science Department, University of Rochester, May 1993.

[vCG92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.

[WeY93] C. Wen and K. Yelick. Parallel Timing Simulation on a Distributed Memory Multiprocessor. Technical Report UCB/Computer Science Department-93-723, Computer Science Division, EECS, University of California at Berkeley, 1993. Revised version submitted for publication.

Performance and Debugging Infrastructure for Compiler Runtime Systems

Parallel Compiler Runtime Consortium: IP3 Working Group

Allen Malony, Dan Reed, Rudolf Eigenmann, Alok Choudhary, Jan Cuny

3.1 Research Summary

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:

- 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

[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.

- 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.

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. In the research plan below, we briefly describe each of these three PEDIGREE infrastructure components.

3.2 Research Plan

3.2.1 Standard Software Interfaces

The primary motivations for developing standard performance and debugging software interfaces are portability and interoperability, both for run-time systems and for performance and debugging tools. 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; any performance or debugging implementation that satisfies the interface constraints can be used. In addition, standard interfaces will encourage the development of "meta-tools" that combine data from multiple performance and debugging systems (e.g., one might combine performance data from task scheduling and data distribution to study the interactions of task scheduling and compiler run-time data distribution).

To provide both interoperability and portability via standard tool interfaces, the PEDIGREE project will necessarily build on the foundation of existing performance instrumentation and data analysis efforts, notably the Pablo [3], HPC++ [2], Fortran-D [7], and PTOPP projects [5], to develop a library of external and internal performance and debugging interfaces. The external interface library will provide the "glue" that connects the run-time software systems and PEDIGREE's implementation of performance and debugging support. Similarly, the internal interface library will provide standard access methods for interaction among performance analysis and debugging tools.

3.2.2 Dynamic Performance Infrastructure and Analysis

Adaptability is a key aspect of an increasingly large fraction of parallel applications (e.g., adaptive meshing and particle methods) [1] as well as an integral component of run-time systems (e.g., dynamic task scheduling, multi-version code execution, and scatter/gather operations for remote memory access on machines with physically distributed memory) [4].

Although dynamic decision procedures, the heart of any adaptive system, require real-time access to dynamic performance data by either the application code or by a controlling user, current performance analysis systems provide little support for on-line performance queries.

Developing support for real-time performance queries will make possible the efficient implementation of a broad range of adaptive application and run-time decision procedures. For example, with efficient access to dynamic data, compiler run-time systems could adaptively select the most efficient of several, compiler-generated code variants. Similarly, with immersive data presentation techniques and real-time performance data, users could interactively steer the assignment of tasks to processors, the mapping of data to distributed memories, or the placement of files on storage devices.

The four keys to adaptive control are dynamic instrumentation, data reduction, queries, and data presentation. To make dynamic queries efficient and practical, it must be possible to dynamically enable and disable instrumentation points in response to program or user requests. Moreover, the time needed to enable instrumentation points must be small and their cost when active must be low. If the hysteresis is too great or the overhead is too high, decision procedures will be driven by data that is neither timely or accurate. Hence, developing low-overhead query and instrumentation mechanisms is of critical importance.

Real-time data reduction is the complement to low-overhead instrumentation. It is not feasible to analyze captured data only at the time a query is issued — the data volume is too great and the computation time too high. Because an adaptive decision procedure will repeatedly issue similar queries (e.g., to guide dynamic data distribution), the instrumentation system must both adapt to query patterns by precomputing the answers to expected queries and identify anomalous performance behavior (e.g., via dynamic statistical clustering).

Choosing appropriate instrumentation and data reduction depends strongly on the character of the expected dynamic queries. We must work closely with the run-time system developers to design the query interfaces needed to support adaptive decisions including dynamic task scheduling [6], multi-version code execution, and dynamic data partitioning.

Finally, real-time, adaptive control requires not only access to dynamic performance data but also effective decision procedures. Techniques for controlling dynamic parallel systems and their hardware and software resources remain an active research area. Data immersion (e.g., via virtual reality techniques) would permit closed-loop, human control of dynamic resource resource allocation for testing and tuning new resource control polices and for aggregate system control (e.g., as an operator).

3.2.3 Dynamic Debugging Infrastructure

Debugging itself is an adaptive process. Because it is neither possible to examine all aspects of a computation in detail nor to know *a priori* which aspects to view, users abstract program behavior in a variety of ways and change those abstractions as the focus of their attention shifts. Dynamic debugging, then, has the same basic requirements as dynamic performance analysis: dynamic instrumentation, data reduction, queries, and data presentation. In addition, it requires reproducibility and access to logically meaningful global states. Existing replay techniques address the issues of reproducibility, thus we focus here on global state access.

We will develop the infrastructure for parallel breakpoint debugging (PBD) software that utilizes run-time software semantics to support coherent, local and global breakpointing in a replay environment. For breakpoint debugging, it is not sufficient to halt processes; they must be halted in globally consistent states that are meaningful to the programmer. This is difficult to do with current debuggers targeted at the machine level; PBD software, however, is targeted at the run-time system. Using run-time semantics, coherent states could be defined between operations in data parallel code, statements in SPMD programs, or at phase transitions, for example. Our infrastructure will provide common mechanisms for instrumenting run-time software for breakpoint management. In addition, it will provide a "parallel breakpoint executive" that supports an external user interface for eliciting information about program state, using queries appropriate to the run-time software abstractions. We will work with run-time system developers in designing these query mechanisms.

To facilitate the application of different debugging strategies, the PBD software will utilize both static (via compiler) and dynamic instrumentation, existing tools for event-based debugging analysis [8], and techniques for parallel program visualization [9]. The user interacts with the PBD software to define global states and state queries of interest. These states and queries form the instrumentation requirements for assertion monitoring and event generation. (With respect to instrumentation, dynamic performance and debugging infrastructure share similar needs. We intend to leverage common instrastructure development where possible and appropriate.) The PBD software will extend event-based behavioral debugging techniques to operate in a higher-level debugging environment where run-time semantics form the basis for event definition and behavioral abstraction. In addition, access to run-time semantics wil make it possible to use standard animation systems in supporting dynamic, high-level visualizations of parallel, distributed data and control structures.

The common PBD infrastructure will be used to build breakpoint debugger tools for the run-time software development efforts. In particular, we intend to demonstrate the efficacy of our approach by implementing initially a PBD for HPC++. HPC++ utilizes a SPMD

programming style with barrier synchronizations after collection operations and thus it is a good candidate for breakpointing. Our present work developing performance tools for HPC++ and HPF will facilitate PBD implementation efforts and provide a direct source of user feedback. Because of the SPMD programming and run-time similarities between the HPC++ and HPF languages, we will consider migrating our PBD software to an HPF environment.

3.3 Work Statement

The PEDIGREE project will deliver performance evaluation and debugging software infrastructure to meet the needs of current and future generations of run-time software. The proposed infrastructure has two emphases, integration and dynamic analysis, though the primary focus is the latter.

We will define standard interfaces to existing performance and debugging infrastructure, including measurement (e.g., profiling and tracing), data analysis, and visualization software. The interfaces will hide the details of specific tool implementation idiosyncrasies, ensuring the portability of instrumented run-time software.[2] The interface definitions will also provide a standard foundation for extending interface capabilities as new tools are developed or existing tools are extended. The interface development will take two forms, external and internal. The external interface library will provide the "glue" that connects the run-time software systems and PEDIGREE's implementation of performance and debugging support. Similarly, the internal interface library will provide standard access methods for interaction among performance analysis and debugging tools; members of the internal interface library are not intended for use by run-time system or application software developers, but will permit integration of current tools.

A standard software interface, though invaluable, is not sufficient. Run-time systems are constantly evolving, creating new performance and debugging requirements. With the explosion of interest in heterogeneous environments and in adaptive solution techniques for irregular and adaptive problems, run-time software is increasingly dependent on dynamic adaptability to achieve high performance. Furthermore, the size and diversity of the execution state space for adaptive software systems requires higher level interfaces (i.e., interfaces with larger semantic content) for state reduction and correctness debugging. We will develop a software infrastructure that supports dynamic instrumentation, queries and data presentation (i.e., activation of instrumentation points, assertion monitoring, and queries by an executing

[2] Though support for run-time systems is our primary focus, the resulting interface will be more broadly applicable.

application or run-time system). This will make possible the efficient implementation of a broad range of adaptive decision procedures (under both application and human control) and as well as execution-time correctness debugging.

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 will implement real-time performance queries for a set of common run-time issues, including dynamic task scheduling, multi-version code execution, and dynamic data partitioning, together with new data presentation techniques that will allow software developers to interact with and guide the decision procedures. For debugging, we will implement an execution breakpointing mechanism that uniformly supports high-level, run-time state access, assertion handling, and execution time debugging facilities.

The majority of the PEDIGREE project resources will be devoted to support for dynamic adaptability; it will require significant, new development effort. The creation of standard software interfaces to existing performance and debugging tools is an extension to current work (notably the Pablo, HPC++, Fortran-D and PTOPP projects) and principally involves identifying and implementing the functionality required by run-time software developers.

3.4 Organization and Level of Effort

The PEDIGREE project represents a three year research effort, organized according to the three activities described above: 1) standard interfaces, 2) performance evaluation, and 3) debugging. The first activity, to be completed during the first two years, will require coordination (by video conference) between group members, principally Illinois and Oregon, for interface definition and implementation. The other two activites will last the entire project period. Although they will be led separately (the performance evaluation work will be led by Reed, Illinois; Malony/Cuny, Oregon, will lead the debugging activity), interaction between group members will continue to be important; Eigenmann (Illinois) and Choudhary (Syracuse) will be active in various aspects across these two main project activities.

The level of effort required for the PEDIGREE project per year is approximately three full-time equivalents (one FTE represents the total support for the IP1 group members) and six research assistants.

Bibliography

[1] H. Berryman, J. Saltz, and J. Scroggs, Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines, *Concurrency Practice and Experience*, 3(3):159–178, June, 1991.

[2] J. K. Lee and D. Gannon, *Object Oriented Parallel Programming: Experiments and Results* Proceedings of Supercomputing 91 (Albuquerque, Nov.), IEEE Computer Society and ACM SIGARCH, 1991, pp. 273–282.

[3] D. Reed, R. Olson, R. Aydt, T. Madhyasta, T. Birkett, D. Jensen, B Nazief, B. Totty, *Scalable Performance Environments for Parallel Systems*, Proceedings of the 6th Distributed Memory Computing Conference, pp. 562–569, 1991.

[4] 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):73–86, 1992.

[5] R. Eigenmann, *Practical Tools for Optimizing Parallel Programs*, 1993 SCS Multiconference, Arlington, VA.

[6] J. Saltz, R. Mirchandaney, and K. Crowley, Runtime Parallelization and Scheduling of Loops, *IEEE Transactions on Computers*, 40(5):603–612, May, 1991.

[7] G. Fox et al., Fortran D Language Specification, Technical Report 90-141, Rice University, Department of Computer Science, December 1990.

[8] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, and L. Snyder, *The Ariadne Debugger: Scalable Application of Event-Based Abstraction*, Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 85–95, May, 1993.

[9] A. Hough and J. Cuny, *Perspective Views: A technique for Enhancing Visualizations of Parallel Programs*, Proceedings 1990 International Conference on Parallel Processing, pp. II:124–132, Aug. 1990.