SANDIA REPORT

SAND2007-2047
Unlimited Release
Printed May 2007

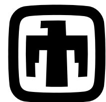
Workshop on Programming Languages for High Performance Computing (HPCWPL) Final Report

Richard C. Murphy

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

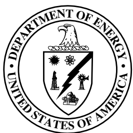
Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online



SAND2007-2047
Unlimited Release
Printed May 2007

Workshop on Programming Languages for High Performance Computing (HPCWPL) Final Report

Richard C. Murphy
HPCWPL Technical Chair
Scalable Computer Architecture, 1422
Sandia National Laboratories
P.O. Box 5800, MS-1319
Albuquerque, NM 87185-1319

Abstract

This report summarizes the deliberations and conclusions of the Workshop on Programming Languages for High Performance Computing (HPCWPL) held at the Sandia CSRI facility in Albuquerque, NM on December 12-13, 2006.

Acknowledgment

This report would be impossible without the efforts of a large number of people. Each of the participants presenters provided invaluable expert input into our effort to understand the state of applications, architecture, and programming models. We are very thankful for their active participation.

The workshop's steering and program committees were invaluable in shaping its content and direction: Ron Brightwell, Bill Carlson, David Chase, Bill Gropp, Bruce Hendrickson, Mike Heroux, Fred Johnson, Peter Kogge, Bob Lucas, Bob Numrich, Marc Snir, Thomas Sterling, Jeffrey Vetter, Christoph von Praun, Zhaofang Wen, and Hans Zima.

We are thankful to the chairs for providing direction for each of their sessions: Peter Kogge served as the workshop chair, Thomas Sterling chaired the architecture session, Jeffrey Vetter chaired the applications session, Marc Snir chaired the HPCS Languages Panel, which served as the basis for many of the recommendations given in this report, Bill Gropp chaired the programming models session, and Bob Lucas chaired final conclusions session.

Each of the deputy chairs from Sandia also provided invaluable input into the final report: Keith Underwood, Mike Heroux, Arun Rodrigues, and Ron Brightwell.

Richard C. Murphy, Technical Chair
Neil D. Pundit, Sandia Host

Executive Summary

This report describes the critical need for additional research in programming languages for high performance computing to enable new applications and the architectures to support those applications. It is the result of a synthesis of the contents of the Workshop on Programming Languages for High Performance Computing (HPCWPL) held at Sandia National Laboratories' Computer Science Research Institute on December 12-13, 2006. This report makes the following observations:

1. **Today's High Performance Computing (HPC) applications consist of large, slowly evolving software frameworks.** These frameworks are typically decades old and have already survived large transitions in computer architecture (e.g., Vector to MPP).

2. **Emerging HPC applications are very different from their traditional scientific computing counterparts.** As a result, the "lessons" from scientific computing do not address the problems of these applications.

3. **Programmers within the DOE are typically experts.** Consequently, performance is more important than rapid prototyping.

4. **HPC workloads share little in common with the most common commercial workloads.** Thus, very little reuse can be made of commercial programming models. For example, Transactional Memory is touted as the correct mechanism for programming multicore architectures, but it is inappropriate for HPC workloads.

5. **The memory wall dominates computer architecture and is increasingly problematic.** Even in the commodity space, the transition to multicore architectures will tremendously increase the demands on memory systems. Memory latency dominates the performance of individual processor cores, while memory bandwidth dominates the number of cores that can be put on a die. Both are problems.

6. **While supercomputer scaling was previously dominated by interconnect performance future machine scaling will be dominated by memory.** Because current programming models focus on interconnection networks rather than memory, this could significantly limit the ability to scale applications.

7. **Both conventional and emerging architectures require additional concurrency to be easily specified.** It is virtually certain that MPI will not be able to provide all of the required concurrency.

8. **Emerging heterogeneous architectures (such as Cell) pose a significant workload partitioning problem.** The job of performing this partitioning is currently the programmer's, and there is little hope for automation on the horizon.

9. **Lack of proper vendor support for Co-Array Fortran and UPC is noted regretfully.**

As a result, the following recommendations are proffered:

I. **We should pursue a minimalist approach to HPC languages.** There are both historical precedents (in the form of MPI) and practical requirements (in the form of a multi-billion dollar investment in existing code) that dictate this path. These languages must be compatible with existing tools and legacy code.

II. **This path should address application requirements first and machine architecture requirements second (but should address both).** New programming models (and architectures) have already been adopted to solve problems that cannot be addressed using the conventional MPI/MPP model. There are common features among radically different problems that cannot be solved today that should be the focus of new programming language extensions. These features include requirements for fine-grained parallelism, synchronization, and data access.

III. **An HPC language should introduce abstractions, using a minimalist approach, which should match HPC's unique requirements.** MPI succeeded by introducing a very small number of key routines that mapped well onto applications and architectures. In the commercial space, Transactional Memory introduces small extensions to introduce nondeterminism into programs (and find parallelism). However, HPC applications typically require *deterministic* parallelism. We encourage vendors to be more cooperative in embracing changes.

IV. **Languages need to support large numbers of efficient, light-weight, locality aware threads.** This serves to increase concurrency, address the core problem of the memory wall, and begin to address the cache coherency problem.

V. **We must identify problems that cannot be solved today, and develop programming languages (and architectures) that solve those problems.** This supports recommendations I-IV by enabling the creation of a significant user-base to support the early development of these languages, and focusing language development on satisfying a specific need.

We are currently at a unique point in the evolution of HPC. Application and underlying technology requirements dictate programming models to support new ways

of solving humanity's problems. Today, architectures and applications are evolving faster than the programming models needed to support them. These recommendations are intended as a first cut strategy for addressing those needs and enabling the solution to new, larger-scale problems. We urge a greater cooperation from vendors in supporting these changes.

Table 1. HPCWPL Participants

Name	Affiliation	Name	Affiliation
Berry, Jon	SNL	Lucas, Bob	ISI
Brightwell, Ron	SNL	Maccabe, Barney	UNM
Chamberlain, Brad	Cray	Mehrotra, Piyush	NASA
Chase, David	Sun	Murphy, Richard	SNL
Chtchelkanova, Almadena	NSF	Numrich, Bob	U. of Minnesota
DeBenedictis, Erik	SNL	Olukotun, Kunle	Stanford
DeRose, Luiz	Cray	Perumalla, Kalyan	ORNL
Doerfler, Doug	SNL	Poole, Stephen	ORNL
Gao, Guang	U. of Delaware	Pundit, Neil	SNL
Gokhale, Maya	LANL	Rodrigues, Arun	SNL
Gropp, Bill	ANL	Snir, Marc	UIUC
Hall, Mary	USC/ISI	Sterling, Thomas	LSU
Hecht-Nielsen, Robert	UCSD	Underwood, Keith	SNL
Hemmert, Scott	SNL	Vetter, Jeffrey	ORNL
Hendrickson, Bruce	SNL	von Praun, Christoph	IBM
Heroux, Mike	SNL	Watson, Greg	LANL
Hofstee, Peter	IBM	Wen, Zhaofang	SNL
Iancu, Costin	LBL	Zima, Hans	CalTech/JPL
Kogge, Peter	U. of Notre Dame		

Table 2. HPCWPL Topics

Topic	**Speaker**
Architecture	Chair: Thomas Sterling
	Deputy: Keith Underwood
Session Keynote: Past Predictions, the Present, and Future Trends	Peter Kogge
Multithreaded	Kunle Olukotun
Reconfigurable	Maya Gokhale
Cell	Peter Hofstee
PIM	Arun Rodrigues
Summary: What is the future of architecture?	Thomas Sterling
Keynote Address: Neurocomputing	Robert Hecht-Nielsen
Applications	Chair: Jeffrey Vetter
	Deputy: Mike Heroux
Session Keynote: Productivity Metrics	Bob Lucas
Graph-Based Informatics	Bruce Hendrickson
Petra Object Model	Mike Heroux
Parallel Discrete Event	Kalyan Perumalla
Summary: How are Applications Evolving?	Jeffrey Vetter
HPCS Language Panel	Chair: Marc Snir
	Deputy: Arun Rodrigues
Chapel	Brad Chamberlain
Fortress	David Chase
X-10	Christoph von Praun
Summary: Ideas for an Ideal Language	Marc Snir
Programming Models Keynote: MPI: The Last Large Scale Success	Bill Gropp
Programming Models	Chair: Bill Gropp
	Deputy: Ron Brightwell
UPC	Costin Iancu
Programming Model Building Blocks	Mary Hall
Co-Array Fortran	Bob Numrich
Locality-Aware High-Productivity Languages	Hans Zima
ParalleX	Thomas Sterling
Programming Environments/Debugging	Greg Watson
Programming and Compiling for TiNyThreads	Guang Gao
Summary: What is the future of programming models?	Bill Gropp
Panel: What is a realistic vision of the future?	Chair: Bob Lucas
	Deputy: Richard Murphy
Panelists: Peter Kogge, Bill Gropp, Jeffrey Vetter, Thomas Sterling	

Contents

Executive Summary **5**

Participants **8**

Topics **9**

List of Abbreviations **14**

1 Introduction **17**

2 Architecture **21**

Memory 21

State 23

Future Architectures 24

Multicore and Multithreaded 24

Heterogeneous 26

Accelerators and Reconfigurable Architectures 26

Near Memory 26

The Shared Memory Controversy 28

The Transactional Memory Controversy 28

Conclusions and Impact on Programming Models 29

3 Applications **31**

Architecture and Application Lifetime 31

Emerging Applications 32

Graph-Based Informatics . . . 32
Petra Object Model . . . 34
Parallel Discrete Event Simulation . . . 35
Conclusions . . . 35

4 Programming Models **37**
MPI . . . 37
Successful Languages . . . 38
Co-Array Fortran (CAF) . . . 38
UPC . . . 38
Key Lessons from PGAS Languages . . . 38
Emerging Models . . . 39
ParalleX . . . 39
Locality-Aware High-Productivity Languages . . . 40
TiNyThreads . . . 40
Tools . . . 41
Programming Environments/Debugging . . . 41
Building Blocks, Flexible Compilers, and Tunable Components . . . 41
Productivity . . . 41
The HPCS Languages . . . 42
Conclusions . . . 43

5 Conclusions **45**

List of Figures

2.1 The Impact of Latency and Bandwidth on Traditional Scientific Codes (Cube3, an iterative solver) and Emerging Combinatorial Codes (DFS, a Depth First Search from Graph Theory). Figure courtesy of Richard Murphy, Sandia National Laboratories. 22

2.2 The growth of CPU state over time. The increase in relative memory latency has led to an exponential growth in processor state. Figure courtesy of Peter Kogge, University of Notre Dame. 24

2.3 Transistor density compared to packaging density. Figure courtesy of Peter Kogge, University of Notre Dame. 25

2.4 The Instruction Mix and Percentage of Computation Time Required for Execution. Figure courtesy of Arun Rodrigues, Sandia National Laboratories 27

3.1 The effects of shrinking machine lifetime and architectural diversification. In comparison, the Sandia's CTH application appeared on the Cray-1 in the late 1970's, and is still a core physics code used by the DOE and DOD. Figure courtesy of Jeffrey Vetter, Oak Ridge National Laboratory (see J.S. Vetter, B.R. de Supinski, J. May, L. Kissel, and S. Vaidya, "Evaluating High Performance Computers," *Concurrency and Computation: Practice and Experience*, 17(10):1239-70, 2005.) 32

3.2 An Attributed Relational Graph Example. Figure courtesy of Tamara Kolda, Sandia National Laboratories. 33

3.3 The typical flow of object construction in ePetra. Figure courtesy of Mike Heroux, Sandia National Laboratories. 34

List of Tables

1 HPCWPL Participants 8

2 HPCWPL Topics 9

List of Abbreviations

ASC The Department of Energy's Advanced Simulation and Computing Program

ASIC Application Specific Integrated Circuit

CAF Co-Array Fortran

CPU Central Processing Unit

CSRI Sandia's Computer Science Research Institute

CTH A 3-d shock physics application developed at Sandia.

DARPA Defense Advanced Research Projects Agency

DFS Depth-First Search of a graph

DIMM Dual In-Line Memory Module

DOD Department of Defense

DOE Department of Energy

DRAM Dynamic Random Access Memory

HPC High Performance Computing

HPCS The DARPA High Productivity Computing Systems Program

HPF High Performance Fortran

ILP Instruction Level Parallelism

IPC Instructions Per Cycle

MPI the Message Passing Interface

MPP Massively Parallel Processor

MTA Multithreaded Architecture

NRE Non-Recurring Engineering

PDES Parallel Discrete Event Simulation

PGAS Partitioned Global Address Space

PIM Processing-In-Memory

PNM Processing-Near-Memory

SMP Simultaneous Multiprocessor

SPMD Single Program Multiple Data

UPC Unified Parallel C

Chapter 1

Introduction

Programming Language Research is vitally important to high performance computing given current trends in architecture and applications. The MPI/MPP programming model has proven highly successful given its tight coupling between architecture and programming environment. Research in programming languages has the potential to enable novel architectures, and vice versa. The Workshop on Programming Languages for High Performance Computing explored these issues, driven by an emerging set of applications that solve critical large-scale problems and are ill-suited to the existing MPI/MPP model. This exploration was accomplished in three primary sessions: emerging architectures, applications, and programming models.

The architecture session explored trends in silicon devices, power, and emerging architectures. The general trend does not favor the MPP architecture. The commodity processor on-node speed-up seen since the advent of Moore's Law will come in the form of an increase in the number of cores (requiring more parallelism) rather than an increase in clock rate and memory bandwidth. Furthermore, given a large projected increase in the number of cores, and a limited projected increase in the number of pins available off chip, the memory bandwidth is unlikely to scale linearly with the number of cores. Chapter 2 demonstrates that this will have a significant negative impact on application performance. Furthermore, emerging heterogeneous architectures (e.g., Cell, accelerators, etc.) make the partitioning of workloads particularly challenging.

The architecture community identified three critical needs for future programming languages (which were echoed identically by the application community):

1. **Concurrency:** The trend toward an increased number of (typically simpler) cores requires additional concurrency. Although studies of SMPs imply that this concurrency can be achieved via MPI, multicore machines differ from SMPs in memory organization. Unlike a multicore architecture, an SMP allows a linear scaling of memory bandwidth as the number of nodes increases. Additionally, the available memory footprint of a multicore processor may be significantly smaller than a traditional SMP (due to limitations on the number of DIMMs that can be connected to a chip). As a result, parallelism below the level of MPI may be more desirable. In the case of emerging architectures, additional

concurrency in the form of threads is often used as the primary mechanism for tolerating memory latency. These threads most likely cannot be provided by MPI.

2. **Synchronization:** As concurrency increases, additional synchronization is required. There are emerging models for specifying synchronization, particularly *Transactional Memory*, that address the needs of commercial workloads. However, these models do not necessarily map well to high performance computing applications. Similarly, some specialized architectures (such as the Cray MTA and XMT) provide synchronization mechanisms but do so through a platform-specific programming model.

3. **Locality Representation:** Memory hierarchies have become increasingly deep and complex. Additionally, programmers (particularly those implementing emerging applications) have asserted the need for global address space languages to facilitate ease of programming. Partitioned Global Address Space (PGAS) languages attempt to facilitate this style of programming, with varying support for "remote memory requests" provided by the hardware. Proposed architectures and accelerators that utilized radically different "local"/"remote" memory access mechanisms make a unified mechanism to represent the locality of data objects particularly challenging for programming language developers.

Application developers specified a similar set of challenges to those presented by the architects. Part of this was driven by their use of novel architectures to solve problems not well suited to commodity architectures. These applications are latency dominated (because of the architectural problem of the memory wall), are difficult to partition, access very sparse data structures, and typically exhibit fine-grained parallelism.

In the case of the graph-based informatics application, 4 MTA-2 processors provide the equivalent 32k BlueGene/L nodes of performance. This indicates a significant challenge in solving large-scale problems of this type.

The adoption of emerging programming models is also a challenge, and it is often tied to the success of new architectures. From the Sandia perspective, three key trends were identified:

i. Most existing DOE applications consist of very large, slowly evolving software frameworks consisting of multiple programming languages. As a result, interoperability with existing programming models is required.

ii. The adoption of new programming models is application driven. Specifically, problems that decompose well into MPI/MPP applications will most likely continue to be written using that model. Problems that cannot be easily solved using that model, however, have the potential to rapidly adopt new programming models. Using the MTA to solve graph-based informatics applications is a good example of this phenomena.

iii. Application developers at Sandia are experts who are primarily focused on performance and scalability. Because of the expense of machine time at the high end, the model tends not to be focused on rapid prototyping or increasing the productivity of relatively inexperienced programmers (which is critically important to other development groups).

As a result, the wide-scale adoption of a new programming model at Sandia would tend to require two competing properties: first, interoperability with old code; and second, enabling new problems to be solved. The measure of success for the adoption of these new models is performance (e.g., an improvement in runtime, an improvement in scalability, or the ability to solve previously unsolvable problems). Consequently, this report recommends a minimalist approach to programming languages research that, like MPI, operates well with other programming models and is capable of being integrated into existing code bases. Additionally, this report recommends that programming languages features be tied **both** to application demands and to emerging architectural features. In particular, features addressing the memory wall problem are critical: providing additional latency toleration, object locality descriptions, and increasing concurrency are key examples.

Historically, language efforts that are tightly focused and address specific problems have been the most successful approach[1]. For example, the C programming language is a necessary precursor to developing the Unix operating system, and became generally useful after being developed for that specific task. This report identifies two emerging problem classes of large-scale importance that fit this description: graph-based informatics and parallel discrete event simulation. Despite being significantly different, they share similar programming model requirements. The identification of additional problems that are difficult or impossible to solve today is strongly recommended.

The remainder of this report is organized as follows: Chapter 2 describes the current state of and future trends in computer architecture. Chapter 3 addresses emerging applications and their relationship to the MPI programming model. Chapter 4 discusses successful and emerging programming models. Finally, Chapter 5 presents the conclusions.

[1]In fact, one is hard pressed to identify a single successful programming language effort that did not begin life meeting the needs of a specific problem or class of problems.

Chapter 2

Architecture

This chapter discusses the current state of computer architecture, and is meant to serve as the foundation for further discussions of applications and programming languages. Fundamentally, the success of a parallel architecture should be thought of in terms of Little's Law, which comes from economics:

$$Throughput = \frac{Concurrency}{Latency} \tag{2.1}$$

Architecturally, the trend in both concurrency and latency is unfavorable. Instruction Level Parallelism (ILP) is the dominant form of concurrency in conventional processors. Current systems provide very little ILP, and the problem of memory latency only makes that worse. MPI has provided significant concurrency, but trends towards multicore architectures make dramatically increasing the amount of concurrency provided by MPI challenging. Finally, that leaves thread level concurrency, which would allow architectures to provide more latency toleration, but has yet to be widely adopted.

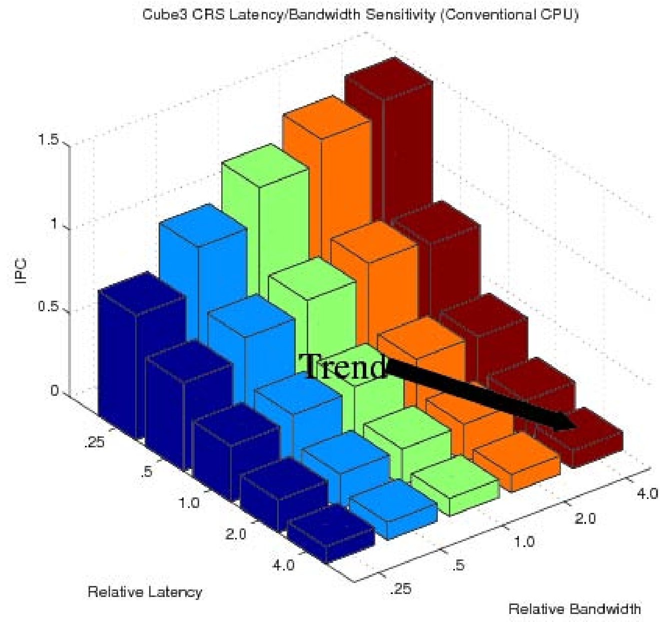
The remainder of this chapter is organized as follows: First, the memory system is examined, as well as the tremendous increase in internal processor state. A subset of interesting architectures is analyzed. We then examine the shared memory and transactional memory controversies. Finally, this chapter concludes with a discussion of the impact on programming models.

Memory

Memory has been the biggest problem in computer architecture for over six decades, and continues to plague high performance computing. It is typically the biggest expense in a supercomputer, and determines the performance of the applications run on that computer. While bandwidth is the typical metric used to evaluate performance, it is memory latency that dominates.

Figure 2.1 depicts the relative impact of memory latency and bandwidth on an Opteron-like processor running at 2 GHz on both traditional HPC applications (in

Cube3



DFS

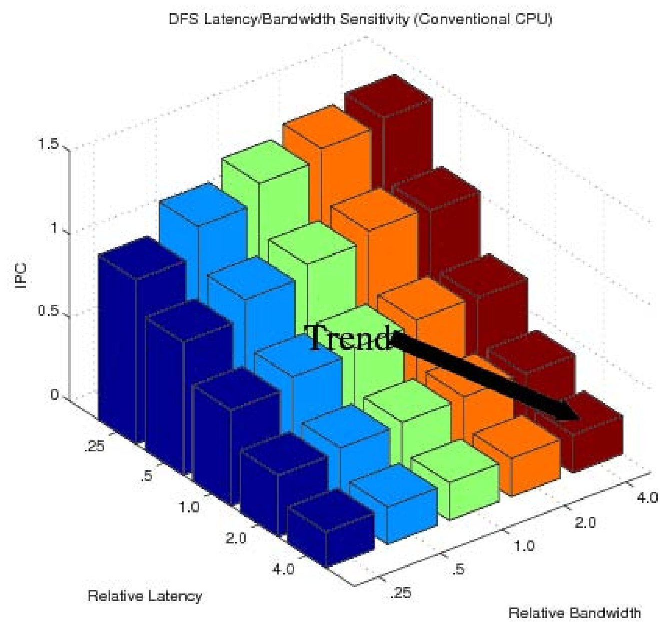


Figure 2.1. The Impact of Latency and Bandwidth on Traditional Scientific Codes (Cube3, an iterative solver) and Emerging Combinatorial Codes (DFS, a Depth First Search from Graph Theory). Figure courtesy of Richard Murphy, Sandia National Laboratories.

this case, a linear solver), and codes of emerging importance (in this case, a depth first search on a large, sparse graph). The results demonstrate that halving the available bandwidth leads to only a 5-7% degradation in performance. By contrast, reducing the memory latency by half leads to a 50-100% *increase* in performance. When combined with the problem of the memory wall (which dictates that memory latency is always increasing relative to processor cycle time), the simple fact is that memory latency is the dominant performance factor for HPC. This is particularly true given that the primary research focus throughout the 1990's was interconnection networks, while memory has been largely unchanged over the past 5 decades.

State

To cope with the memory wall, all of modern processor architecture work is dominated by mechanisms for avoiding or tolerating latency. As examples, caches attempt to avoid memory latency by bringing frequently used items physically "closer" to processing resources; out of order execution provides a small measure of latency toleration by using instruction level parallelism to continue to make progress while high latency operations are being performed (though we continue to see significantly less than one instruction retired in every clock cycle, see Figure 2.1); and finally, the emerging use of threads in architectures to allow the programmer to specify additional concurrency to be used during high latency memory operations is a clear indication of the latency problem. The consequence of these efforts, however, is a tremendous growth in the state that the CPU must manage.

Figure 2.2 shows the growth in CPU state over time. This increase in state complexity is primarily due to architectural mechanisms for coping with long memory latencies. Those mechanisms are typically *ineffective*. As an example, the effectiveness of out of order execution is measured by the number of instructions per cycle (IPC) the processor retires. Figure 2.1 shows IPCs of less than 0.5 for two key applications, which is typical for HPC workloads. Considering that such a processor could have an IPC as high as 4, such results demonstrate extreme inefficiency. In fact, a processor with an IPC of 1.0 clocked at half the clock rate would execute both programs faster at significant power savings.

The cost of this additional state is simple: chip area (and, consequently manufacturing yield), design complexity (which increases NRE and verification costs), and power (which tends to outstrip the cost of the hardware for a supercomputer deployment).

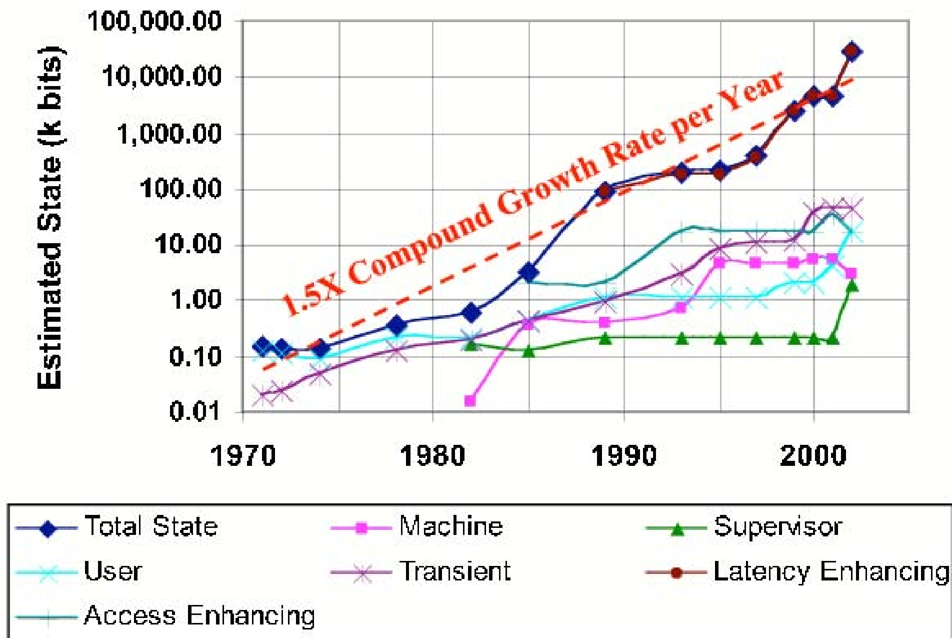


Figure 2.2. The growth of CPU state over time. The increase in relative memory latency has led to an exponential growth in processor state. Figure courtesy of Peter Kogge, University of Notre Dame.

Future Architectures

The key problems in computer architecture (as enumerated above) are fundamentally unchanged. There is a disconnect between the problem of memory and the interconnection-network focus of high performance systems architecture research that has dominated since the early 1990's. This section examines four areas of specific architectural innovation: multicore and multithreaded, heterogeneous, accelerators and reconfigurable, and near memory.

Multicore and Multithreaded

Though there is an undeniable trend towards multicore architectures, from the programmer's standpoint these machines appear to be either an "SMP on a chip" or a multithreaded machine. In either case, the programming model is fundamentally threaded. Currently there exists very little capability to program a machine with dramatically more threads of execution than existing supercomputers. Furthermore, for traditional scientific applications, significantly increasing the number of processing

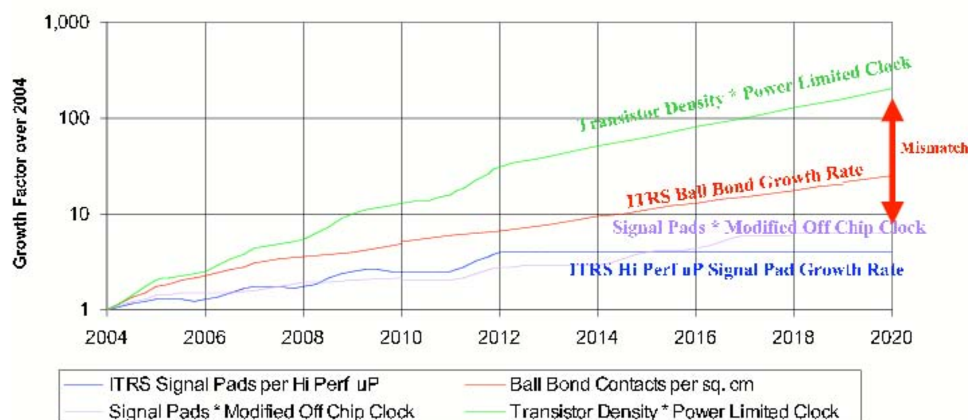


Figure 2.3. Transistor density compared to packaging density. Figure courtesy of Peter Kogge, University of Notre Dame.

nodes is a challenge. For example, scaling the number of compute nodes in an MPI program by an order of magnitude can cause difficulty in mesh generation and create load imbalance.

Architecturally, the performance of multicore processors is limited by the following characteristics:

- Pins off the chip (to memory) become the strangle point;
- the memory wall problem is intensified since there are fewer available memory banks per core than unicore architectures; and,
- conventional architectures currently lack the efficient inter-processor coordination needed when the number of threads in the system is increased.

The trend towards an increased number of simpler cores exacerbates all of these problems. Furthermore, it is currently very difficult for the programmer to extract thread-level concurrency. The successful mechanisms that exist to do so exist in the proprietary compilers for specialized architectures (such as the Cray XMT) rather than in compilers or programming languages that support a wide range of architectures of this class.

Figure 2.3 depicts the mismatch between the growth in logic density and the availability of off-chip communication pins. This mismatch means that while fabrication processes will support significantly more cores on a chip the channels available to off-chip memory will become increasingly limited.

Heterogeneous

Heterogeneous architectures offer the programmer multiple models of computation. As an example, IBM's Cell processor supports a gather-compute-scatter memory model and both a vector and scalar compute model. The memory model looks quite familiar to the HPC community as it is similar to the overlays used in vector machines. Despite over three decades of research the overlay programming problem has never been automated. Additionally choosing the right model of computation in a heterogeneous architecture is challenging. In short, these architectures have no clear path to an efficient or portable programming model.

Accelerators and Reconfigurable Architectures

Accelerators and reconfigurable architectures offer significant opportunity to accelerate portions of a computation, but often ignore Amdahl's law. Amdahl's law dictates that if a fraction of a program (P) can be accelerated by a speedup (S) then the overall speedup is given by:

$$\frac{1}{(1-P)+\frac{P}{S}} \tag{2.2}$$

Consequently, given an *infinite* speedup on 50% of the program, the overall gain dictated by equation 2.2 is only $2\times$.

Accelerators suffer from an even more subtle Amdahl problem: typically they are second-class citizens in any system architecture. If the accelerator requires transfers from main memory to the accelerator's memory between steps of the computation, then the denominator in equation 2.2 increases. Thus if an additional transfer overhead (expressed as a fraction of execution time) of T is required to access the accelerator, the speedup is dictated by:

$$\frac{1}{(1-P)+T+\frac{P}{S}} \tag{2.3}$$

Given the same problem as above with an additional 5% transfer time to access the accelerator the speedup is 1.82. That is, the performance improvement suffers a 9% penalty for the transfer.

Near Memory

One solution to the problem of memory latency is to move the processing resources physically closer to the memory. There are two clear paths to doing so: Processing-

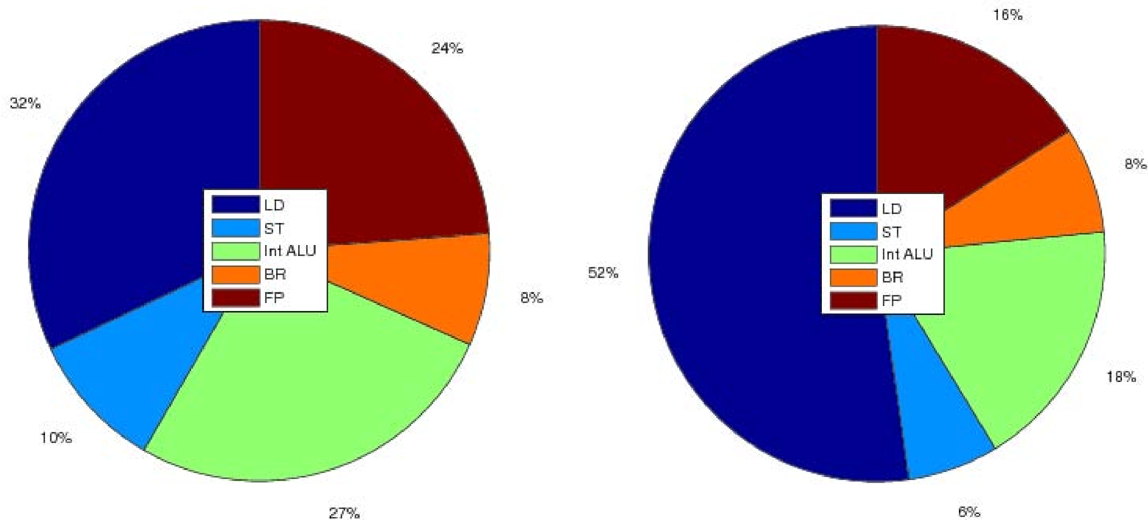


Figure 2.4. The Instruction Mix and Percentage of Computation Time Required for Execution. Figure courtesy of Arun Rodrigues, Sandia National Laboratories

In-Memory (PIM) combines dense DRAM and logic on the same die (or two dies stacked in three dimensions); and Processing-Near-Memory (PNM) more tightly couples processing resources to the memory without merging fabrication processes.

Figure 2.4 shows the instruction mix and percentage of the program execution time for each class of instructions for a mix of real-world supercomputer applications at Sandia. Loads dominate both: they are 32% of all instructions and consume 52% of the program's execution time. Opportunities for PIM and PNM have increased and become more economical:

- PIM fabrication is now offered commercially by most ASIC foundries by adding DRAM to conventional logic fabrication processes;
- conventional processors are moving from bus-based memory interfaces to more serial interfaces requiring high-speed signaling (and consequently fabricated separately from the DRAM), which permits adding PNM resources close to DRAM at very low cost (since the logic ASIC is already required); and,
- the fabrication of "stacked" systems with multiple modules is becoming increasingly common in the commodity space (e.g., cell phones often stack processors and memory to improve packaging).

Additionally the dense local memory available to a PIM architecture can alleviate the bandwidth constrictions conventional multicore architectures experience. It has been demonstrated that a PIM-based multicore architecture can outperform a conventional multicore machine when executing scientific and emerging HPC codes by $2\times$-$40\times$ or more given the same chip area and a PIM clocked at half the rate of the conventional machine because the PIM has to perform fewer accesses to external DRAM.

The Shared Memory Controversy

The strongest point of contention for future architectures was the debate over shared memory. The argument can be summarized as follows:

- **Pro:** shared memory eases programming and enables solving problems that are not easily solved on distributed memory machines. Further, the preeminence of distributed memory architectures has affected the choice of problems to solve.
- **Con:** distributed memory programming forces the programmer to partition the data appropriately and improves performance. As a result, distributed memory architectures scale better.

Both sides agree that conventional *cache coherent shared memory* architectures are not sufficiently scalable due to the overhead associated with coherency. Furthermore, several emerging applications of critical importance are very difficult to program in the MPI model (see Chapter 3). The Partitioned Global Address Space (PGAS) model of computation appears promising, but lacks sufficient architecture and programming language support at this time.

The Transactional Memory Controversy

Given that multicore architectures are inevitable, some form of parallel programming is required in the main stream market. The dominant choice of technology today is Transactional Memory which permits the programmer to bundle atomic operations together easily and allows the compiler and runtime system to schedule those operations independently while still maintaining the dataflow requirement of the application. There are three significant problems with using transactional memory for high performance computing:

- Transactional Memory is an extension of database semantics, which are not appropriate for traditional scientific computing applications (though some emerging applications may benefit from this model). HPC is focused on concurrent

deterministic programming whereas transactions are inherently nondeterministic.

- Transactional Memory suffers from an Amdahl's law problem in that only independent atomic sections can be scheduled concurrently. The HPC community requires significantly more concurrency at all levels (thread level, MPI level, etc.) in order to build peta-scale systems and beyond.

- Transactional Memory is typically implemented inside the cache coherency protocol of an SMP, and coherency is generally considered too expensive and difficult to scale for scientific computing. The only other alternative today is software which is slower still. It is possible that a lighter-weight version could be developed in the future.

Consequently, there is a need to identify programming methods to increase concurrency that match the requirements of HPC applications. More significantly, there is a disconnect between the programming models pursued by industry for multicore architectures (which target commercial applications) and those useful to HPC. This is a particularly daunting problem because architectural support for programming models outside of the main stream is limited at best.

Conclusions and Impact on Programming Models

Programming models require improvement in three main areas relating to architecture:

1. **Concurrency:** Architectures require a greater specification of parallelism to increase throughput and provide for tolerating long memory latencies. This is particularly true for multicore architectures. Additionally, applications require additional concurrency to scale to larger problem sizes. MPI remains the primary, manually identified mechanism for specifying concurrency, but may be insufficient in and of itself in future machine generations.

2. **Synchronization:** Both architectures and applications require better mechanisms for synchronizing between concurrently running portions of the application. Commercial workloads often consist of numerous independent tasks, which is not true for high-end HPC applications.

3. **Locality Representation:** There currently exists no convenient way of representing the relationship between algorithmic objects and data constructs. In the MPI model, this is done explicitly by the programmer.

Each of these areas represents a fundamental architectural challenge for programmers. Multicore architectures rely on increasing numbers of simple cores, requiring more parallelism from the application. This increased parallelism requires synchronization for coordination. And, finally, data partitioning will become increasingly difficult for MPP machines, particularly if multiple levels of parallelism (MPI and threads, for example) are required to exploit future architectures.

Architecturally, the challenges remain the same: the technology drives power, reliability, and chip pin counts; application workloads are changing in nature (see Chapter 3); and these technology and application drivers make programmability increasingly difficult.

Chapter 3

Applications

The focus of the applications session at HPCWPL was on those applications that are currently not well served by the MPP computing model. Over the course of discussion, it became clear that while software tends to be the largest and longest living investment (the ASC program spends approximately 10% on hardware that lasts three to five years, while applications live for decades), the choice of applications that are developed is partially driven by how "natural" it is to map a given algorithm on to a machine. For example, the three dimensional simulations of physics on a computer required by the national labs map very well onto MPP machines programmed with the MPI programming model. They tend to be partitionable in space, map very well to the mesh or toroidal networks that are popular on these platforms, and decompose well into large messages. Many emerging applications are significantly more irregular, sparse, and difficult to partition. Because they are so difficult to express in existing programming models and architectures, these applications are best served by new ones, which should be the focus of new programming languages research. By contrast, most of the current application base is well served by existing programming models (e.g., MPI) which map very efficiently onto the architectures that run them (e.g., MPPs).

The remainder of this chapter is organized as follows: First, the relative lifetimes of architectures and application code bases is examined. Next, there is a discussion of the applications presented at the workshop focusing on the difficulties these applications pose to existing programming models and architectures. Finally, the conclusions are examined.

Architecture and Application Lifetime

Figure 3.1 depicts the lifetime of various HPC architectures over time. Architectures have diversified and now exhibit shorter life spans. On the other hand, software tends to exhibit a very long life span (decades), and often outlives the architecture for which it was written (e.g., codes that began life on vector machines have been ported to MPPs).

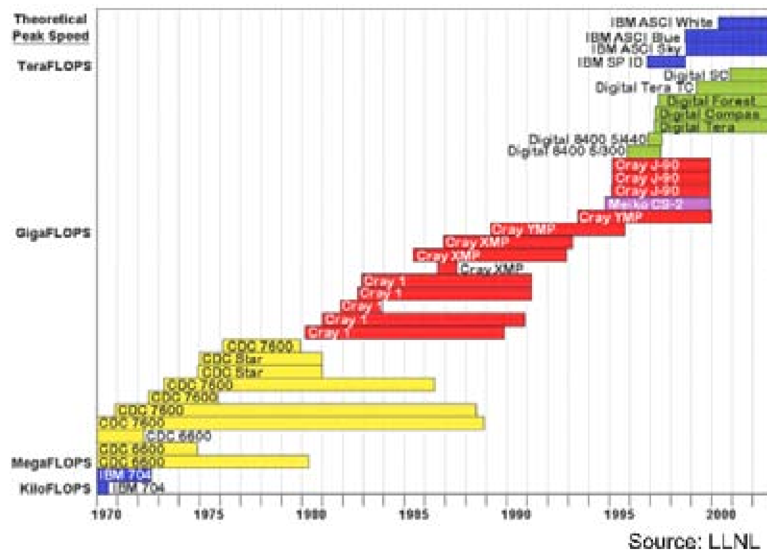


Figure 3.1. The effects of shrinking machine lifetime and architectural diversification. In comparison, the Sandia's CTH application appeared on the Cray-1 in the late 1970's, and is still a core physics code used by the DOE and DOD. Figure courtesy of Jeffrey Vetter, Oak Ridge National Laboratory (see J.S. Vetter, B.R. de Supinski, J. May, L. Kissel, and S. Vaidya, "Evaluating High Performance Computers," *Concurrency and Computation: Practice and Experience*, 17(10):1239-70, 2005.)

In each generation, HPC tends to abandon the applications that are not well suited to that generation's architecture.

Emerging Applications

The workshop examined three applications detailed in this section: graph-based informatics, the Petra object model for data movement in Trilinos, and parallel discrete event simulation. This section details each application's unique properties.

Graph-Based Informatics

The analysis of large graphs is a potential HPC application with little in common with traditional scientific applications. Problems can be huge, exhibit no exploitable global structure, and may be impossible to partition due to the lack of locality. **The body of knowledge acquired from scientific computing is of limited utility.**

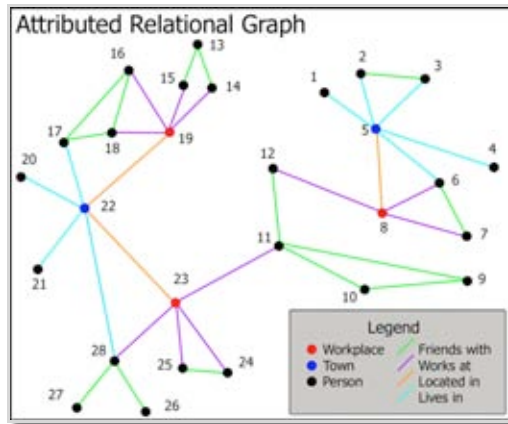


Figure 3.2. An Attributed Relational Graph Example. Figure courtesy of Tamara Kolda, Sandia National Laboratories.

Critically, these problems typically cannot be efficiently executed at scale on MPP machines and benefit from alternative programming models. Because of the fine-grained access requirements, MPI implementations tend to be very inefficient.

Figure 3.2 shows an example of such a graph with vertices that represent people and locations, and edges that represent relationships. When traversing this graph, parallelism is often very fine-grained, and the runtime is dominated by memory latency. There are four desirable architectural features:

1. Low Latency/High Bandwidth Memory Accesses
2. Latency Toleration
3. Light weight, fine-grained synchronization mechanisms
4. Global address space

It has been demonstrated that this problem benefits significantly from unconventional multithreaded architectures. Four processors of the Cray MTA-2 have been demonstrated to equal approximately 32 thousand processors of BlueGene/L.

This has been listed as one of the "13 Dwarfs" that represent key problems for multicore architecture[1].

[1]See *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report Number UCB/EECS-2006-183

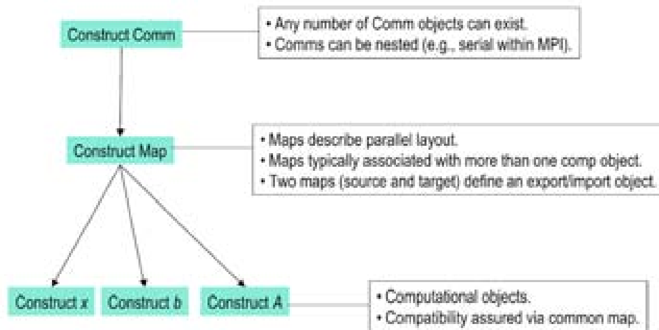


Figure 3.3. The typical flow of object construction in ePetra. Figure courtesy of Mike Heroux, Sandia National Laboratories.

Petra Object Model

Many scientific and engineering applications (particularly those at Sandia) spend $30-80\%$ of their runtime in solvers, which makes solver libraries a particularly good point of optimization. Early vector architectures were designed to perform sparse solves efficiently. Commodity processors have significant room for improvement in executing them. Solvers are also subject to significantly degraded performance with increased memory latency (see Section 2).

While these problems have relatively efficient MPI implementations and represent the computational core of many HPC applications for MPPs, there is significant room for improvement.

Figure 3.3 provides an example of the construction of objects in the ePetra portion of Trilinos, which represents basic linear algebra objects. The key abstraction is the construction of a `Comm` object that abstracts the communication mechanism from the programmer. This serves to abstract the behavior and attributes of the parallel machine from the user. Currently, serial, MPI, and shared memory `Comm` objects are supported, although new architectures can be added easily. `Comm` objects can also be arranged hierarchically. This abstraction allows for significant portability and extensibility of the library.

ePetra also supports `Map` objects that allow for the redistribution of the data underlying a given class. This permits the library to choose more efficient loop executions, and allows for data redistribution.

The organization of modern high performance computing libraries (like Trilinos) serves to enable the adoption of novel architectures, particularly when those libraries

represent such a significant portion of the runtime. Because of their size, however, they may represent a challenge to the adoption of entirely new programming languages if those languages do not support the existing code base. This tends to support the conclusion that incremental extensions to existing programming languages may best support today's significant investment in software.

Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) is emerging as a key mechanism for simulating organizations (command and control, processes, etc.) and social dynamics (operations planning, foreign policy, etc.), in addition to its more traditional uses in network, traffic, and sensor simulation. Problem sizes continue to increase, and it represents an excellent example of a non-traditional HPC application that requires capability computing. For example, the Red Storm machine recently enabled the simulation of a terrorism problem at significantly enhanced resolution. Results from these simulations tend to impact emergency planning, resource consumption estimates, and policy formulation. Consequently, high-resolution results are required quickly.

PDES shares many of the same properties of the graph-based applications discussed in Section 3. Very little floating point is executed, the communication pattern is irregular, and fine-grained synchronization is required.

Conclusions

These emerging irregular applications represent a shift away from traditional HPC applications that are well suited to the MPI programming model and MPP architecture. They require fine-grained data access and synchronization that MPI simply does not support. More significantly, MPP architectures are ill suited to support that type of synchronization and data access. Additionally, SMP-based architectures may support the fine-grained data access, but they do not support the required synchronization. Finally, only non-commodity architectures, such as the Cray MTA, support the fine-grained parallelism exhibited by many classes of emerging applications (e.g., graph based informatics and PDES).

Chapter 4

Programming Models

Enabling hard problems to be solved is the key challenge for new programming models, and was the central theme explored in the programming models session of the workshop. Additionally, it is the defining characteristic of successful programming languages endeavors. Examples include MPI, which enabled the large-scale simulation of physics, C, which enabled the creation portable operating systems.

The remainder of this chapter is organized as follows: First, MPI and other successful language projects are addressed. Then, emerging language proposals and language related tools are examined. Given this background, the concept of productivity and the HPCS languages are discussed. And, finally the conclusions are presented.

MPI

MPI has been by far the most successful programming model endeavor over the last two decades. Its success can be attributed to six critical properties. MPI is:

- **Portability:** Aside from running on a large range of machines, MPI enables performance enhancing features without requiring hardware support for them. For example, nonblocking message passing permits "zero-copy" data transfers, but does not require them.
- **High Performance:** MPI's distributed memory model forces the programmer to perform a data decomposition. In high-end physics codes, this has resulted in higher performance than other models.
- **Simplicity and Symmetry:** Although MPI is feature rich, the majority of MPI programs use a relatively small number of primitives that map well to basic MPP architectural features.
- **Modularity:** MPI is both designed to support libraries and implemented as a library, which helps to enable modern software architectures built from components and frameworks.

- **Composability:** MPI's structure allows it to be part of a diverse set of programming environments. It is essentially "language neutral".

- **Completeness:** MPI avoids simplifications that limit the programming model. The user does not have to change programming models to add features.

As a highly successful parallel programming model, MPI enabled the solution to extremely difficult problems (particularly physics problems at the DOE). There is an emerging class of applications that require alternative programming models to solve large scale problems (see Chapter 3).

Successful Languages

The workshop examined Co-Array Fortran and UPC as examples of successful work in programming languages.

Co-Array Fortran (CAF)

Co-Arrays first appeared in 1991, and will be part of the Fortran 2008 standard, which demonstrates the fundamentally long path required for the adoption of programming language constructs. CAF deliberately restricts the compiler to optimizing local memory latency accesses, and requires the programmer to focus on remote data management (very much like MPI). As a Partitioned Global Address Space (PGAS) language, CAF fundamentally provides the programmer with a single address space, but avoids the requirement of cache coherency which limits the scalability of SMP-based architectures. This model is currently not well supported in hardware due to the coherency requirements of modern processors.

UPC

The UPC model provides a PGAS memory model that enables Single Program Multiple Data (SPMD) parallelism, forces the programmer to consider the implications of local vs. global data layout, and provides the simplicity of a global address space.

Key Lessons from PGAS Languages

There are three points related to PGAS languages that were consistently brought up at the workshop:

- Synchronization is very difficult to specify, often coarse grained, typically left to the programmer, and not generally supported in hardware.

- Data decomposition has to be done manually.

- Work distribution has to be done manually.

There is significant debate as to whether or not these properties are "good" things. One side contends that because of their manual nature, the programmer is forced to optimize the application and choose high-performance trade-offs; the other is concerned that this impacts programmer productivity.

Emerging Models

This section explores three programming language and execution model proposals that address emerging hardware and programmer productivity requirements: ParalleX, Locality-Aware High-Productivity Languages, and TiNyThreads.

ParalleX

The basis for the ParalleX execution model is the fundamental observation that existing technology trends require changes in computer architecture. Those changes can only be enabled by changes in the execution model. Because ParalleX is an execution model, it does not specify policies for implementation, technology, structure, or mechanism (and is thus not a programming model, architecture, or virtual machine). ParalleX provides:

- **Split Phase Transactions:** that provide independent actions on exchanged values.

- **Localities:** representing contiguous physical domains that provide "local" and "remote" latency information to the programmer.

- **A Global Name Space:** with no coherence between localities (very much like PGAS languages). Communication between localities occurs via *Parcels*, with primitives for synchronization and threads.

- **Parcels:** are a form of message-driven computation that specify a function to be performed on a named data element. They provide the ability to move work between objects in different localities.

- **Multi-Grain Multithreading:** to specify the relationship between operations and provide latency tolerance.
- **Percolation:** as a latency hiding and scheduling technique that moves data and tasks to local temporary storage for execution. This is particularly important for accelerators, functional elements, and other precious resources.
- **Fine-Grain Event Driven Synchronization:** in a number of forms, including message-driven remote thread instantiation, lightweight objects, and in-memory synchronization.

Locality-Aware High-Productivity Languages

The High Performance Fortran (HPF) project provides critical lessons for Locality-Aware High Productivity Languages. HPF-1 worked very well for small problems, but presented difficulties in expressivity and performance for large-scale applications. Many of these difficulties arose from the challenges in providing a mature compiler infrastructure, inconsistency in the implementation, and the complexity of the relationship between the compiler and the target platform. HPF-2 addressed some of these problems, but only after significant work.

HPF was the first attempt at a high-productivity language for HPC, and provided the key concept of locality awareness. However, the acceptance of new languages depends on many criteria, including:

- functionality and the performance of target code;
- a mature compiler and runtime infrastructure;
- user familiarity;
- easy integration of legacy codes;
- an integrated development environment;
- flexibility to deal with new platforms;
- and both research and major vendor support.

TiNyThreads

Given a new generation of multicore architectures that significantly reduce the computation, managing on-chip parallelism and bandwidth, to tolerate off-chip bandwidth

and latency limitations is a significant challenge. The Cyclops architecture provides a unique design point to consider in the space of multithreaded machines. It provides an efficient hardware multi-threading model with explicit memory regions and in-memory atomic operations. This type of hardware enables research into static dataflow architectures. Revisiting dataflow architectures and runtime systems is an increasingly prevalent research technique given the characteristics of emerging machines.

Tools

This section examines programming environments and compiler building blocks.

Programming Environments/Debugging

One of the important gaps in existing parallel research tends to be programming environments and debugging, which offer productivity enhancements for legacy and emerging codes. Environments such as Eclipse allow for the simplification of complex tool chains and presentation of a more logical workflow. Additionally, these tools can offer higher-level correctness and consistency checking than the compiler.

Building Blocks, Flexible Compilers, and Tunable Components

The emergence of heterogeneous architectures (such as accelerators, the Cell processor, etc.) pose a significant challenge to both programmers and compiler writers (see Chapter 2). Future compiler infrastructure needs to be built from components that are modular and easy to understand. This would enable bringing new platforms up quickly and facilitate compiler research. No systematic, principled approach exists today.

There are existing examples of compiler tuning beating library frameworks, but the existing lack of modularity limits the adoption of these techniques.

Productivity

Fundamentally, two views of productivity were explored at the workshop: first, that productivity enables programs to be formulated more quickly, or computing resources

to be used more easily, often by less skilled programmers; and second, that productivity enables expert programmers to achieve higher performance on a given system quickly. The core of the debate is the role of computer scientists on a high-performance computing team. Because Sandia hosted the workshop and "high-performance" tends to mean very high-end supercomputers, the focus was intentionally high-end. Because programming teams at Sandia typically consist of highly knowledgeable application domain experts and computer scientists, and because of the expense of the machines involved, the Sandia contingent tended to prefer the second definition of productivity over the first. This is a critical distinction between the DOE community of supercomputer users and others.

The HPCS Languages

Each of the HPCS language vendors was asked to address the following questions:

What is the time-frame for generating large-scale applications (10's to 100's of thousands of lines of code) on large machines with performance comparable to MPI?

Each of the vendors felt that this would depend entirely on each language's adoption rate. Funding had not been allocated in DARPA HPCS Phase III proposals to undertake this effort, especially because of the shift post-2007 to a consortium effort. It is also felt that the HPCS languages could significantly reduce the line-count of codes.

What features of your language are most suited to current and forthcoming scientific computing applications on highly parallel systems? What about emerging HPC applications (such as those discussed in Chapter 3)?

Each language claimed:

- Control of locality and distributed data structures
- Fine-grained threading and synchronization
- Enhanced data structures (e.g., Chapel's sparse/associative/opaque domains)

If a standardization effort was started to create a single HPCS language, what critical features of your language would be required? Do you think such an effort makes sense?

- A Partitioned Global Address Space (PGAS) model
- Shift from SPMD programming and execution models
- Automatic data distribution

- Control over locality for data and computation
- Type safety
- Generators to manage parallelism
- Structured parallelism

Each of the vendors thought that this effort was sensible.

Conclusions

Programming languages research needs to address problems not addressed by existing programming models, particularly those that result from enabling new architectural features. As the most successful model thus far, MPI provides a unique library-based implementation that can be combined with numerous existing architectures and runtime environments, and can augment legacy codes. If MPI codes are to become "legacy", an emerging language has to maintain the same capability: specifically, drawing in and enhancing existing code bases. It will simultaneously have to address the challenges of emerging applications that require significantly different programming models.

Chapter 5

Conclusions

This report has examined the state of HPC programming languages in the context of applications and architectures. The maturity of the MPP/MPI architecture and programming model combination, and its inability to address the problem of critical emerging applications means that there is significant opportunity to develop new programming languages that address key application problems. This report strongly recommends focusing language effort in that area, and identifies two important classes of problems that can benefit from the same set of language features. It is further recommended that additional difficult or impossible to solve problems be identified for further study.

Focusing on emerging problems allows the successful HPC language effort to develop a significant user base that can be expanded over time.

To foster the adoption of a new programming language outside its core user base, an evolutionary approach is recommended. In this model, the “new” programming language is a minimal set of useful extensions to an existing language, rather than something entirely new. Aside from increased programmer familiarity, this approach stresses inter-operation with existing programming environments and frameworks. It also recognizes the existence of a multi-billion dollar investment in applications that are unlikely to be rewritten from scratch. Finally, it leverages a large and rich set of compiler infrastructure and optimizations developed over the past four decades. MPI is an excellent example of this design philosophy.

Architecturally, there are two critical trends that impact the design of future programming languages:

- First, memory is an increasing problem (both latency, which has been given very little attention over the past 20 years, and bandwidth); and
- second, both conventional and emerging architectures require significantly more concurrency to enable supercomputers to continue scaling.

The conventional trend only emphasizes both problems: the number of cores is increasing (requiring additional concurrency), without providing the same increase in total chip bandwidth (exacerbating the memory wall).

A new programming language should address both problems by providing increased concurrency, and helping the programmer to manage increasingly deep memory hierarchies (to more efficiently utilize the available bandwidth). The same requirements exist for emerging architectures: multithreaded and latency tolerant architectures require a larger number of threads to achieve high throughput, and suffer from the same memory problem as any other machine. Heterogeneous architectures (such as the Cell) have all the same properties with the addition that the computation must be partitioned between very different functional units to provide better performance.

Finally, because the DOE model tends to be the development of large-scale software frameworks that last significantly longer than the underlying architectures. Given this long view and a set of very experienced application developers, emerging programming languages are required to facilitate the large-scale solution of problems that cannot be solved today, or an increase in scalability for existing codes that have likely made at least one transition in the underlying architecture.

DISTRIBUTION:

1 Bill Camp, Intel, PO Box 5800, MS-1318, Albuquerque, NM 87185-1318

1 Almadena Chtchelkanova, NSF, 4201 Wilson Boulevard, Arlington, VA 22230

1 Bill Harrod, DARPA/IPTO, 3701 Fairfax Drive, Arlington, VA 22203-1714

1 Fred Johnson, DOE Office of Science, SC-21, Germantown Building, 1000 Independence Ave SW, Washington, DC 20585

1 Jose Munoz, NSF, 4201 Wilson Boulevard, Arlington, VA 22230

1	MS 1322	John Aidun, 1435
1	MS 9159	Heidi Ammerlahn, 8962
1	MS 1319	Jim Ang, 1420
1	MS 1319	Bob Benner, 1422
1	MS 0104	Thomas Bickel, 1200
1	MS 0376	Ted Blacker, 1421
1	MS 1320	Scott Collis, 1414
1	MS 1319	Doug Doerfler, 1422
1	MS 1322	Sudip Dosanjh, 1400
1	MS 9152	Jerry Friesen, 8963
1	MS 0139	Art Hale, 1900
1	MS 9159	Mike Hardwick, 8964
1	MS 9151	Howard Hirano, 8960
1	MS 0316	Scott Hutchinson, 1437
1	MS 9158	Curtis Janssen, 8961
1	MS 0801	Rob Leland, 4300
1	MS 1318	Scott Mitchell, 1411
1	MS 9151	Len Napolitano, 8900

1	MS 0321	Jennifer Nelson, 1430
1	MS 0807	John Noe, 4328
1	MS 1319	Carl E. Oliver, 1000
1	MS 1319	Neil Pundit, 1423
1	MS 0384	Art Ratzel, 1500
1	MS 1316	Danny Rintoul, 1409/1412
1	MS 0822	David Rogers, 1424
1	MS 1318	Suzanne Rountree, 1415
1	MS 1318	Andrew Salinger, 1416
1	MS 0806	Len Stans, 4336
1	MS 0370	Jim Strickland, 1433
1	MS 0378	Randy Summers, 1431
1	MS 1322	Jim Tomkins, 1420
1	MS 0370	Tim Trucano, 1411
1	MS 0831	Mike Vahle, 5500
1	MS 1318	David Womble, 1410
1	MS 0823	John Zepper, 4320
2	MS 9018	Central Technical Files, 8944
2	MS 0899	Technical Library, 4536