DARPA's HPCS Program: History, Models, Tools, Languages

Jack Dongarra, University of Tennessee and Oak Ridge National Lab
Robert Graybill, USC Information Sciences Institute
William Harrod, DARPA
Robert Lucas, USC Information Sciences Institute
Ewing Lusk, Argonne National Laboratory
Piotr Luszczek, University of Tennessee
Janice McMahon, USC Information Sciences Institute
Allan Snavely, University of California – San Diego
Jeffery Vetter, Oak Ridge National Laboratory
Katherine Yelick, Lawrence Berkeley National Laboratory
Sadaf Alam, Oak Ridge National Laboratory
Roy Campbell, Army Research Laboratory
Laura Carrington, University of California – San Diego
Tzu-Yi Chen, Pomona College
Omid Khalili, University of California – San Diego
Jeremy Meredith, Oak Ridge National Laboratory
Mustafa Tikir, University of California – San Diego

Abstract

The historical context surrounding the birth of the DARPA High Productivity Computing Systems (HPCS) program is important for understanding why federal government agencies launched this new, long-term high performance computing program and renewed their commitment to leadership computing in support of national security, large science, and space requirements at the start of the 21st century. In this chapter we provide an overview of the context for this work as well as various procedures being undertaken for evaluating the effectiveness of this activity including such topics as modeling the proposed performance of the new machines, evaluating the proposed architectures, understanding the languages used to program these machines as well as understanding programmer productivity issues in order to better prepare for the introduction of these machines in the 2011-2015 timeframe.

1. HISTORICAL BACKGROUND

The historical context surrounding the birth of the High Productivity Computing Systems (HPCS) program is important for understanding why federal government agencies launched this new, long-term high performance computing program and renewed their commitment to leadership computing in support of national security, large science, and space requirements at the start of the 21st century.

The lead agency for this important endeavor, not surprisingly, was DARPA, the Defense Advance Research Projects Agency. DARPA's original mission was to prevent technological surprises like the launch of Sputnik, which in 1957 signaled that the Soviets had beaten the U.S. into space. DARPA’s mission is still to prevent technological surprises, but over the years it has expanded to include creating technological surprises for America's adversaries. DARPA conducts its mission by sponsoring revolutionary, high-payoff research that bridges the gap between fundamental discoveries and their military use. DARPA is the federal government's designated "technological engine" for transformation, supplying advanced capabilities, based on revolutionary technological options.

Back in the 1980s, a number of agencies made major investments in developing and using supercomputers. The High Performance Computing and Communications Initiative (HPCCI), conceived in that decade, built on these agency activities and in the 1990s evolved into a broad, loosely coupled program of computer science research. Key investments under the HPCCI and other programs   have enabled major advances in computing technology and helped maintain U.S. leadership in the world computer market in recent decades.

In the late 1990s and early 2000s, U.S. government and industry leaders realized that the trends in high performance computing were creating technology gaps. If left unchecked, these trends would threaten continued U.S superiority for important national security applications and could also erode the nation's industrial competitiveness.   The most alarming trend was the rapid growth of less-innovative, commodity-based clustered computing systems ("clusters"), often at the expense of the leading-edge, capability class of supercomputers with key characteristics supportive of an important set of applications. As a result of this strong market trend, the entire ecosystem needed to maintain leadership in high-end, capability-class supercomputers was in peril: the few companies producing high-end supercomputers had less money to invest in innovative hardware research and development, and firms that created high-performance versions of software applications, environments, and tools for high-end supercomputers had a more difficult time making a business case for this specialized activity. The seemingly inexorable advance of commodity microprocessor speeds in obedience to Moore's Law propelled the growth of clusters with hundreds, then thousands of processors (although this same increasing parallelism also gave rise to the programming challenge that continues to plague the high performance computing industry today).

A CHRONOLOGY

The goal of this section is to provide the first comprehensive chronology of events related to the HPCS program. The chronology is based on reports, documents and summaries that have been accumulated over time by more people than I can mention here. Special credit is due to Charles Holland, Richard Games and John Grosh for their contributions, especially in the early years leading up to the HPCS program. In the chronology, events that were part of the HPCS program, or sponsored by the program, are highlighted in italics.

1992: DARPA funding support focuses on companies developing massively parallel processing (MPP) systems based on commodity microprocessors (e.g., Thinking Machines' Connection Machine CM5, Intel's Paragon system).

1995: The Department of Energy establishes the Accelerated Strategic Computing Initiative (ASCI) to ensure the safety and reliability of the nation’s nuclear weapons stockpile through the use of computer simulation rather than nuclear testing. ASCI adopts commodity HPC strategy.

25 February 1996: Silicon Graphics acquires Cray Research, which becomes a subsidiary of SGI.

17 May 1996: The University Corporation for Atmospheric Research (UCAR), a federally funded agency in Boulder, Colorado, awards a $35 million contract for a supercomputer purchase to a subsidiary of NEC of Japan. The U.S.-based subsidiary of NEC outbids two other finalists for the contract—Fujitsu U.S. and Cray Research of Eagan, Minnesota—to supply a supercomputer to UCAR’s National Center for Atmospheric Research (NCAR) for modeling weather patterns.

29 July 29 1996: Cray (now an SGI subsidiary) petitions the International Trade Administration (ITA), a division of the U.S. Commerce Department, claiming that it had been the victim of “dumping.” The ITA upholds the dumping charge and the NCAR purchase of the NEC supercomputer is cancelled.

19 June 1997: Sandia National Laboratories' "ASCI Red" massively parallel processing system uses 9,216 Intel Pentium Pro microprocessors to achieve 1.1 trillion floating point operations per second on the Linpack benchmark test, making it the top supercomputer in the world and the first to break the teraflop/s barrier.

26 September 1997: The International Trade Commission (ITC) determines that Cray Research has suffered “material injury” and imposes punitive tariffs of between 173% and 454% on all supercomputers imported from Japan, a barrier so high it effectively bars them from the U.S. market.

22 September 1999: SGI announces that it will be receiving significant financial aid from several U.S. government agencies, including the National Security Agency (NSA), to support the development of the company’s Cray SV2 vector supercomputer system.

15 November 1999: Jacques S. Gansler tasks the Defense Science Board (DSB) to address DoD supercomputing needs, especially in the field of cryptanalysis.

2 March 2000: Tera Computer Company acquires the Cray vector supercomputer business unit and the Cray brand name from SGI. Tera renames itself as Cray Inc.

11 October 2000: The DSB Task Force on DoD Supercomputing Needs publishes its report. The Task Force concludes current commodity-based HPCs are not meeting the computing requirements of the cryptanalysis mission. The Task Force recommends that the government:

(1) Continue to support the development of the Cray SV2 in the short term.
(2) In the midterm, develop an integrated system that combines commodity microprocessors with a new, high-bandwidth memory system.
(3) Invest in research on critical technologies for the long term.

Fall 2000: *DARPA Information Technology Office (ITO) sponsors high performance computing technology workshops led by Candy Culhane and Robert Graybill (ITO)*

23 March 2001: Dave Oliver and Linton Wells, both from the DoD, request a survey and analysis of national security high performance computing requirements to respond to concerns raised by U.S. Representative Martin Sabo (D-Minn.) that eliminating the tariffs on Japanese vector supercomputers would be a bad idea.

April 2001: Survey of DoD HPC requirements concludes that cryptanalysis computing requirements are not being met by commodity-based high performance computers, although some DoD applications are being run reasonably well on commodity systems because of a significant investment by the DoD HPC Modernization Program to make their software compatible with the new breed of clusters. But the survey also reveals significant productivity issues with commodity clusters in almost all cases. The issues range from reduced scientific output due to complicated programming environments, to inordinately long run times for challenge applications.

26 April 2001: Results of the DoD HPC requirements survey are reviewed with Congressman Sabo. Dave Oliver, Delores Etter, John Landon, Charlie Holland, and George Cotter attend from the DoD. The DoD commits to increasing its R&D funding to provide more diversity and increase the usefulness of high performance computers for their applications.

3 May 2001: Commerce Department lifts tariffs on vector supercomputers from Japan

11 June 2001: *Release of the DoD Research and Development Agenda for High Productivity Computing Systems White Paper, prepared for Dave Oliver and Charlie Holland. The white paper team was led by John Grosh (Office of the Deputy Under Secretary of Defense for Science and Technology) and included Robert Graybill (DARPA)), Dr. Bill Carlson (Institute for Defense Analysis Center for Computing Sciences), and Candace Culhane. The review team consisted of Dr. Frank Mello (DoD High Performance Computing Modernization Office), Dr. Richard Games (The MITRE Corporation), Dr. Roman Kaluzniacki, Mr. Mark Norton (Office of the Assistant Secretary of Defense, Command, Control, Communications, and Intelligence), and Dr. Gary Hughes.*

June 2001: *DARPA ITO sponsors an IDA Information Science and Technology (ISAT) summer study, "The Last Classical Computer," chaired by Dr. William J. Dally from Stanford University.*

July 2001*: DARPA approves High Productivity Computing Systems Program based to large degree on the HPCS white paper and ISAT studies. Robert Graybill is the DARPA program manger. The major goal is to provide economically viable high productivity computing systems by the end of 2010. These innovative systems will address the inherent difficulties associated with the development and use of current high-end systems and applications, especially programmability, performance, portability and robustness.*

To achieve this aggressive goal, three program phases are envisioned: (1) concept study; (2) research and development; and (3) design and development of a petascale prototype system. The program schedule is defined as follows:

- *I. June 2002 – June 2003: Five vendors to develop concept studies for an HPC system to appear in 2010.*
- *II. July 2003 – June 2006: Expected down selection to 2 – 3 vendors (number depends on funding level) to develop detailed system designs for the 2010 system and to perform risk reduction demonstrations.*
- *III. July 2006 – December 2010: Down selection to 1 – 2 vendors (number depends on funding level) to develop research prototypes and pilot systems.*

January 2002: *DARPA's HPCS Phase I Broad Area Announcement (BAA) is released to industry.*

February 2002: Congress directs the DoD to conduct a study and deliver by 1 July 2002 a development and acquisition plan, including budgetary requirements for a comprehensive, long-range Integrated High-End Computing (IHEC) program. NSA is designated as the lead agency. DARPA, the DoD HPC Modernization Program, NIMA, NRO, DOE/NNSA, and NASA are named as contributing organizations.

8 March 2002: NEC Corporation announces the delivery of its vector parallel computing system based on the NEC SX-6 architecture to the Japanese Earth Simulator Center. The system sustains 35.6 Tflop/s on the Linpack benchmark, making it the fastest computer in the world—approximately 5 times faster than the previous #1, the DOE "ASCI White" computer at the Lawrence Livermore National Laboratory. Jack Dongarra, who helps compile the Top500 computer list, compares the event's shock impact with the Sputnik launch, and dubs it "Computenik."

May-June 2002: The NSA-led Integrated High-End Computing (IHEC) study commences with a number of focused workshops.

June 2002*: Phase I of the DARPA HPCS program begins with one-year study contracts awarded to Cray, HP, IBM, SGI, and Sun. NSA provides additional funds for Phase I awards. The goal of the program is to develop a* ***new revolutionary generation*** *of* ***economically viable*** *high productivity computing systems for national security and industrial user communities by 2010, in order to* ***ensure U.S. leadership, dominance, and control*** *in this critical technology*

The vendors' conceptualizing efforts include a high degree of university participation (23), resulting in a wealth of novel concepts. In addition, a number of innovative technologies from DARPA's active embedded programs are considered by the vendors: Data Intensive Systems (DIS), Polymorphous Computing Architectures (PCA), and Power Aware Computing and Communications (PACC).

21 October 2002: Sandia National Laboratories and Cray Inc. announce that they have finalized a multiyear contract, valued at approximately $90 million, under which Cray will collaborate with Sandia to develop and deliver a new supercomputer called Red Storm. The machine will use over 16,000 AMD Opteron microprocessors and have a peak processing rate of 100 trillion floating point operations per second.

21 November 2002: Users of the Japanese Earth Simulator capture three out of five Gordon Bell prizes awarded at the Supercomputing 2002 conference. In one case, scientists run a 26.58 Tflop/s simulation of a complex climate system. This corresponds to 66% of the peak processing rate. Competing commodity systems in the U.S. deliver 10% or less of peak rates, illustrating one of the productivity issues that the DARPA HPCS program is proposing to address.

Dec 2002: The FY03 federal budget includes language proposing the development of an interagency R&D roadmap for high-end computing core technologies, along with a federal high-end computing capacity and accessibility improvement plan. In response to this guidance, the White House Office of Science and Technology Policy (OSTP), in coordination with the National Science and Technology Council, commissions the creation of the interagency High-End Computing Revitalization Task Force (HECRTF). The interagency HECRTF is charged with developing a five-year plan to guide future federal investments in high-end computing.

June 2003: Computing Research Association leads a workshop, chaired by Dr. Daniel A. Reed, on "The Road for the Revitalization of High-End Computing," as part of the High-End Computing Revitalization Task Force's effort to solicit public comment on the planning process.

July 2003*: DARPA HPCS Phase I down-select is completed and Phase II three-year research and development Other Transactions Authority (OTA) contracts are awarded to Cray, IBM, and Sun.*

July 2003*: A multi-agency initiative (DARPA, DOE Office of Science, NNSA, NSA, NSF, and NASA) funds a three-year HPCS productivity team effort led by Dr. Jeremy Kepner from MIT-Lincoln Laboratory. The productivity team is comprised of universities, laboratories, Federally Funded Research and Development Centers (FFRDCs) and HPCS Phase II vendors. Bi-annual public productivity conferences are held on a regular basis throughout the three-year Phase II program.*

10 May 2004: High-End Computing Revitalization Task Force (HPCRTF) Report is released by the Office of the Science and Technology policy (OSTP)

May 2004*: DARPA sponsors the High Productivity Language System (HPLS) workshop, which is organized by Dr. Hans P. Zima from JPL to form the basis for the HPCS experimental language development activity. Experimental languages discussed include Chapel (Cray), X10 (IBM), and Fortress (Sun).*

Nov 2004*: First formal public announcement is made at the Supercomputing Conference (SC2004) of the new HPC Challenge benchmarks, based on the work done under the HPCS Productivity Team efforts led by the University of Tennessee.*

2004: The National Research Council (NRC) releases a report, "The Future of Supercomputing," sponsored by the DOE Office of Science.

August 2005: Completion of the report from the Joint UK Defense Scientific Advisory Council and U.S. Defense Science Board Study on Critical Technologies. High Performance Computing is identified as a critical technology and the report makes key recommendations to maintain U.S. /UK HPC superiority.

***Sept 2005**: The Army High Performance Computing Research Center (AHPCRC) and DARPA sponsor the first Parallel Global Address Space (PGAS) programming models conference in Minneapolis. Based on the interest level in the first conference, the plan is to turn this event into an annual conference.*

***Nov 2005**: First HPC Challenge performance and productivity awards are made at SC2005.*

***Dec 2005**: Dr. William Harrod becomes the HPCS program manager after Robert Graybill's six year DARPA term expires.*

***Nov 2006**: DARPA HPCS Phase II down-select is completed. Phase III multi-year prototype development Other Transaction Authority (OTA) cost-sharing contracts are awarded to Cray and IBM, with petascale prototype demonstrations planned for the end of 2010. This is a mulit-agency effort involving DARPA (lead agency), NSA, DOE Office of Science, and NNSA (the HPCS mission partners) each contributing to Phase III funding.*

As this chronology suggests, this period represented a tumultuous transition period for supercomputing, resulting in no shortage of reports, recommendations, and ideas on the roles of public and private sector in maintaining U.S. superiority from the national security and economic perspectives. There was also growing public awareness that theoretical ("peak") performance could no longer be a sufficient measure of computing leadership. During this period of public/private partnerships, the future of supercomputing has been altered by new wave of innovations and real sense that the real value of the computing is in achieving end users business objectives, agency mission and scientific discovery.

1.1 HPCS Motivation

As already noted, high performance computing was at a critical juncture in the United States in the early 2000s, and the HPCS program was created by DARPA in partnership with other key government agencies to address HPC technology and application challenges for the next decade.

A number of DoD studies[1, 2] stressed that there is a national security *requirement* for high performance computing systems, and that, consistent with this requirement, DoD historically had provided partial funding support to assist companies with R&D for HPC systems. Without this government R&D participation, high-end computing might one day

[1] "Task Force on DOD Supercomputing Needs," Defense Science Board Study, October 11, 2000.
[2] "Survey and Analysis of the National Security High Performance Computing Architectural Requirements," Presentation by Dr. Richard Games, MITRE, April 26, 2001.
[3] "DARPA HPCS Application Analysis and Productivity Assessment," MITRE, October 6, 2002.

be available only through manufacturers of commodity clusters based on technologies developed primarily for mass-market consumer and business needs.

While driving U.S. superiority in high-end computing technology, the HPCS program will also contribute significantly to leadership in these and other critical DoD and industrial applications areas: operational weather and ocean forecasting; planning for the potential dispersion of airborne contaminants; cryptanalysis; weapons (warheads and penetrators); survivability/stealth design; intelligence/surveillance/reconnaissance systems; virtual manufacturing/failure analysis of large aircraft, ships, and structures; and emerging biotechnology applications. The HPCS program will create new systems and software tools that will lead to increased productivity of the applications used to solve these critical problems.

The critical mission areas are described below. Some descriptions were derived from a report submitted to Congress by the Office of the Secretary of Defense ("High Performance Computing for the National Security Community") Others came from a report created by MITRE[3]. The list is not exhaustive. HPCS systems are likely to be used for other missions—both military and commercial—if the systems provide a balanced architecture and are easy to use.

Operational Weather and Ocean Forecasting – Provides worldwide 24-hour weather guidance to the military, CIA, and Presidential Support Unit for current operations, weapons of mass destruction contingency planning, etc.

Signals Intelligence – The transformation, cryptanalysis, and intelligence analysis of foreign communications on the intentions and actions of foreign governments, militaries, espionage, sabotage, assassinations, or international terrorism. There are both research and development and operational aspects of this activity.

Intelligence, surveillance, and reconnaissance – Processing the outputs of various types of sensors to produce battlespace situation awareness or other actionable intelligence. Includes target cueing, aided target recognition, and other special exploitation products. These operational applications have to meet throughput and latency requirements as part of a larger system.

Dispersion of airborne contaminants – Predicts the dispersion of hazardous aerosols and gasses in the atmosphere. Supports military operation planning and execution, intelligence gathering, counter terrorism, and treaty monitoring.

Weapons design – Uses computer models to augment physical experimentation to reduce costs and explore new concepts that would be difficult to test. Computational mechanics are used to understand complex projectile-target interactions to develop advanced survivability and lethality technologies. Computational fluid dynamics is used for modeling flight dynamics of missiles and projectiles.

Survivability and stealth – Includes performing research into reducing the radar signatures of airplanes such as the JSF and F22, and providing technical support for acquisition activity. Uses computational electromagnetics for radar cross-section/signature prediction.

Engineering design of large aircraft, ship and structures – Applies computational structural mechanics used to do forensic analysis after terrorist bomb attacks and predictive analysis for the design of safer military and embassy structures. Augments aircraft wind tunnel experiments to reduce costs.

Biotechnology – Uses information technology to create, organize, analyze, store, retrieve and share genomic, proteomic, chemical and clinical data in the life sciences. This area is not strictly considered a national security mission area, but it is of use to the military. More important, it is a growing field in private industry. If HPCS meets biotechnology users' needs, it may enhance the commercial viability of computer systems developed under the HPCS program.

1.2 HPCS Vision

The HPCS vision of developing economically viable high productivity computing systems, as originally defined in the HPCS white paper, has been maintained throughout the life of the program. The vision of economically viable—yet revolutionary—petascale high productivity computing systems led to significant industry and university partnerships early in the program and a heavy industry focus later in the program. To achieve the HPCS vision, DARPA created a three phase program. A broad spectrum of innovative technologies and concepts were developed during Phase I. These were then evaluated and integrated into a balanced, innovative preliminary system design solution during Phase II. Now, in Phase III, the systems are under development, with prototype petascale demonstrations planned for late 2010.

The end product of the HPCS program will be systems with the ability to efficiently run a broad spectrum of applications and programming models in support of the national security and industrial user communities. HPCS is focusing on revolutionary, productivity-enhancing improvements in the following areas:

- **Performance:** Computational capability of critical national security applications improved by 10X to 40X over the 2002 capability.
- **Programmability:** Reduce time to develop, operate, and maintain HPCS application solutions to one-tenth of 2002's cost.
- **Portability:** Make available research and operational HPCS application software that is independent of specific hardware and software architectures.
- **Robustness (Reliability):** Continue operating in the presence of localized hardware failure, contain the impact of software defects, and minimize the likelihood of operator error.

Achieving the HPCS vision will require an optimum balance between revolutionary system requirements incorporating high risk technology, and features and functionality needed for a commercial viable computing system. The HPCS strategy has been to encourage the vendors to not simply develop evolutionary systems, but to make bold step *productivity* improvements, with the government helping to reduce the risks through R&D cost sharing. Productivity, by its very nature, is difficult to assess because it depends upon the specifics of the end user mission, applications, team composition, and end use or workflow as shown in Figure 1.1. A proper assessment requires a mixture of qualitative and quantitative (preferred) analysis to develop a coherent and convincing argument. The productivity goals of the HPCS Phase III system can be loosely grouped into **execution time and development time** goals. The goals of the program have been refined over the three phases as they have gone through this very challenging balancing process. The following refined goals have emerged from that process.

Productivity (development time) goals.

- Improve development productivity by 10X over 2002 development productivity for specified government workflows (Workflows 1, 2, 4 and 5).
- Improve execution productivity to 2 petaflops sustained performance (scalable to greater than 4 petaflops) for Workflow 3.

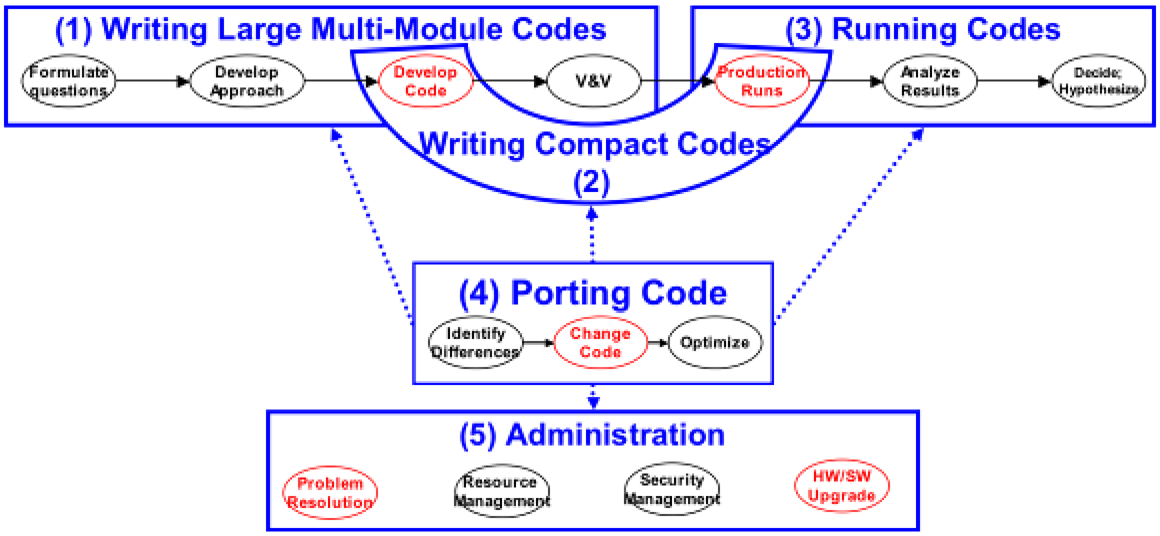


Figure 1.1. Level 1 Functional Workflows. Workflows comprise several steps; many overlapping. Items in red represent areas with highest HPC-specific interest.

No single productivity number applies to workflows for the 2002 starting point, or to 2010 workflows. Productivity will vary based on the specific machine, user, and application. DoD and other government agencies will determine 2002 baseline productivity metrics for their government applications and mission requirements, and will then evaluate the petascale prototype system to demonstrate the 10X improvement.

The HPCS program must address overarching issues impeding the development and utilization of high-end computational systems:

- Balanced system performance
- Improved software tools and methodologies
- Robustness strategy
- Performance measurement and prediction
- System tailorability (ability to scale up and out)

The following Table 1.1 lists the current and HPCS-targeted capabilities (execution time) for HPC systems. Note that "current" is defined as 2007, rather than the program's 2002 starting point.

Benchmark	**Description**	**Current**	**HPCS**
Global High-Performance LINPACK (G-HPL) (PF/s)	Sustained execution speed @ local nodes	~0.2	2+
STREAM (PB/s)	Data Streaming mode – data processing rate	~0.1	6.5
Global Random Access (GUPS/s)	GUPS – Random Access across entire memory system	35	64K
Bisection B/W (PB/s)	Min bandwidth connecting equal halves of the system	~.001 - .01	3.2

Table 1.1. Performance (Execution times) derived from HPC Challenge Benchmarks

These future HPC systems will also have to operate as major subsystems of the HPCS mission partners' computing centers and meet their growing input/output and data storage requirements. The goals listed below represents the mission partners' requirements.

- 1 trillion files in a single file system
- 10,000 metadata operations per second
- Streaming I/O at 30 GB/sec full duplex
- Support for 30,000 nodes

These objectives cannot be met simply by tracking Moore's Law and leveraging evolutionary commercial developments, but will require revolutionary technologies and close partnerships between vendors and candidate procurement agencies. A fall back to evolutionary HPC systems with a focus on performance at the expense of productivity by vendor product organizations is not an acceptable alternative.

1.3 Program Overview

The HPCS acquisition strategy shown in Figure 2.1 is designed to enable and encourage revolutionary innovation by the selected contractor(s) in close coordination with government HPC end users.

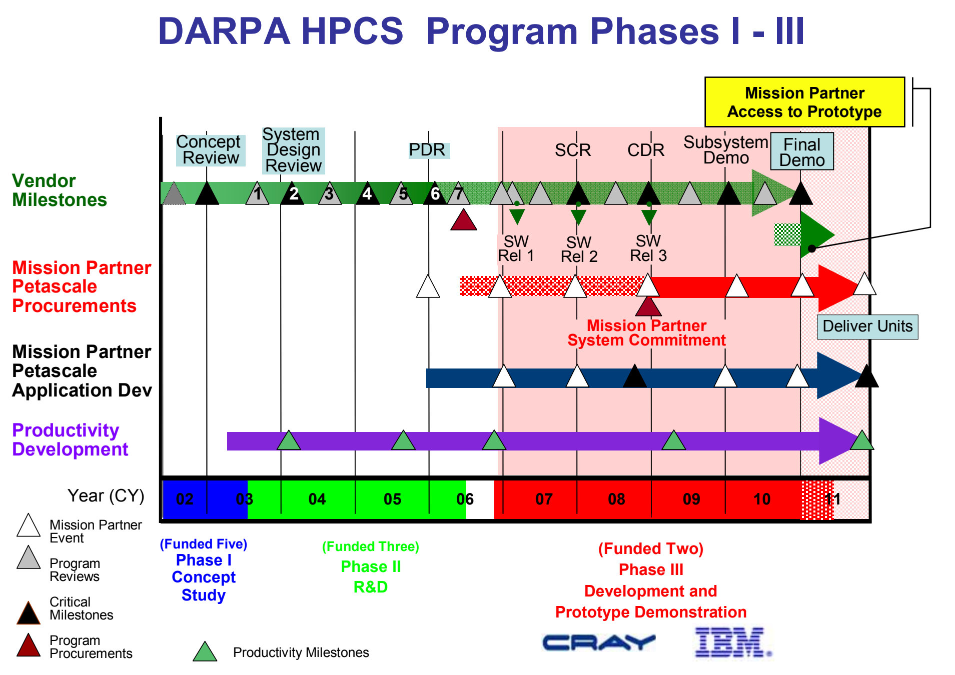

DARPA HPCS Program Phases I - III
Mission Partner Access to Prototype
Concept Review
System Design Review
PDR
SCR
CDR
Subsystem Demo
Final Demo
Vendor Milestones
1
2
3
4
5
6
7
SW Rel 1
SW Rel 2
SW Rel 3
Mission Partner Petascale Procurements
Mission Partner System Commitment
Deliver Units
Mission Partner Petascale Application Dev
Productivity Development
Year (CY)
02
03
04
05
06
07
08
09
10
11
Mission Partner Event
Program Reviews
Critical Milestones
Program Procurements
Productivity Milestones
(Funded Five) Phase I Concept Study
(Funded Three) Phase II R&D
(Funded Two) Phase III Development and Prototype Demonstration
CRAY
IBM

Figure 1.2, HPCS Program and Acquisition Strategy

As Figure 1.2 illustrates, the DARPA-led HPCS program (denoted by the arrow labeled "Vendor Milestones") is divided into three phases. Phase I, an industry concept study, was completed in 2003. DARPA awarded 12-month contracts to industry teams led by Cray, HP, IBM, SGI, and Sun. The study provided critical technology assessments, revolutionary HPCS concept solutions, and new productivity metrics in order to develop a new class of high-end computers by the end of this decade. The outputs from Phase I convinced government decision-makers of the merits of continuing the program.

Phase II was a three-year effort that began with continuation awards to the Cray, IBM and Sun teams. These teams performed focused research, development, and risk reduction engineering activities. The technical challenges and promising solutions identified during the Phase I concept study were explored, developed, and simulated or prototyped. The work culminated in the contractors' preliminary HPCS designs. These designs, along with the vendors' risk reduction demonstrations, analyses, lifecycle cost projections, and their Phase III proposals were used by the decision-makers to determine whether it was technically feasible and fiscally prudent to continue to develop and procure HPCS systems.

Phase III, now under way, is a design, development and prototype demonstration effort that will last four and a half years. The Phase III vendors, Cray and IBM, will complete the detailed design and development of a productive, petascale system and will prove out the system by demonstrating a petascale prototype. They will demonstrate the HPCS performance and productivity goals for an agreed-upon set of applications by the end of 2010. The Phase III mission partners will have access to these systems for the first six months in 2011. Figure 1.2, in addition, outlines the relationship between the HPCS program and agency procurement programs aimed at addressing the petascale computing challenges of the DOE Office of Science, National Nuclear Security Agency, National Security Agency, and the National Science Foundation.

1.4 Cray "Cascade" and IBM "PERCS" Overview

Cray and IBM have provided summaries of their proposed Phase III systems with special emphasis on their revolutionary hardware and software architectures resulting in significant improvement in overall user productivity. Detailed descriptions of the Phase III vendors' innovative architectures are not described here, due to the proprietary nature of the designs at this time. The authors encourage readers to inquire directly with Cray and IBM for fuller descriptions of their novel system architectures.

The Cray Cascade System

Cray's Cascade system is based on two observations regarding high performance computing and productivity. The first is that no one processing technology is best for all applications. Application requirements and coding paradigms vary widely, and different forms of parallelism can be best exploited by different processor architectures. Effort spent trying to fit code onto a particular processing architecture is one the largest drains on productivity. The second observation is that programming productivity starts with appropriate support at the architectural level. Machine characteristics such as the compute/bandwidth balance, overhead of synchronization, availability of threads and latency tolerance of processors have a significant impact on software's ability to efficiently exploit a variety of code constructs and allow programmers to express problems in the most natural way.

The Cascade system is designed to be highly configurable and extensible. It provides a high-bandwidth, globally addressable memory, and supports multiple types of compute blades that can be tailored for different application requirements. The interconnect can be dialed from low to very high local and global bandwidth, and scales to well over 100,000 compute nodes with low network diameter. Compute blades in Cascade use commodity microprocessors augmented with both communication and computational accelerators. The communication accelerators extend the address space, address translation and communication concurrency of the commodity processors, and provide global synchronization and efficient message-passing support. The computational accelerators, based on massively multithreaded and vector technology, can adapt their mode of operation to the characteristics of the code, and provide significant speedup with little programmer intervention. Typical parallel architectures present numerous barriers to achieving high performance, largely related to memory access, and communication and synchronization between threads. The Cascade architecture removes many of these barriers, allowing programmers to write codes in a straightforward, intuitive manner and still achieve high performance. The key attributes of the architecture are motivated by a desire to both increase performance and improve programmability:

- Cascade provides a large, globally addressable memory and extremely high global bandwidth. This enables very low-overhead access to shared data, which allows programmers to express algorithms naturally, rather than laboring to reduce and lump together inter-processor communication.

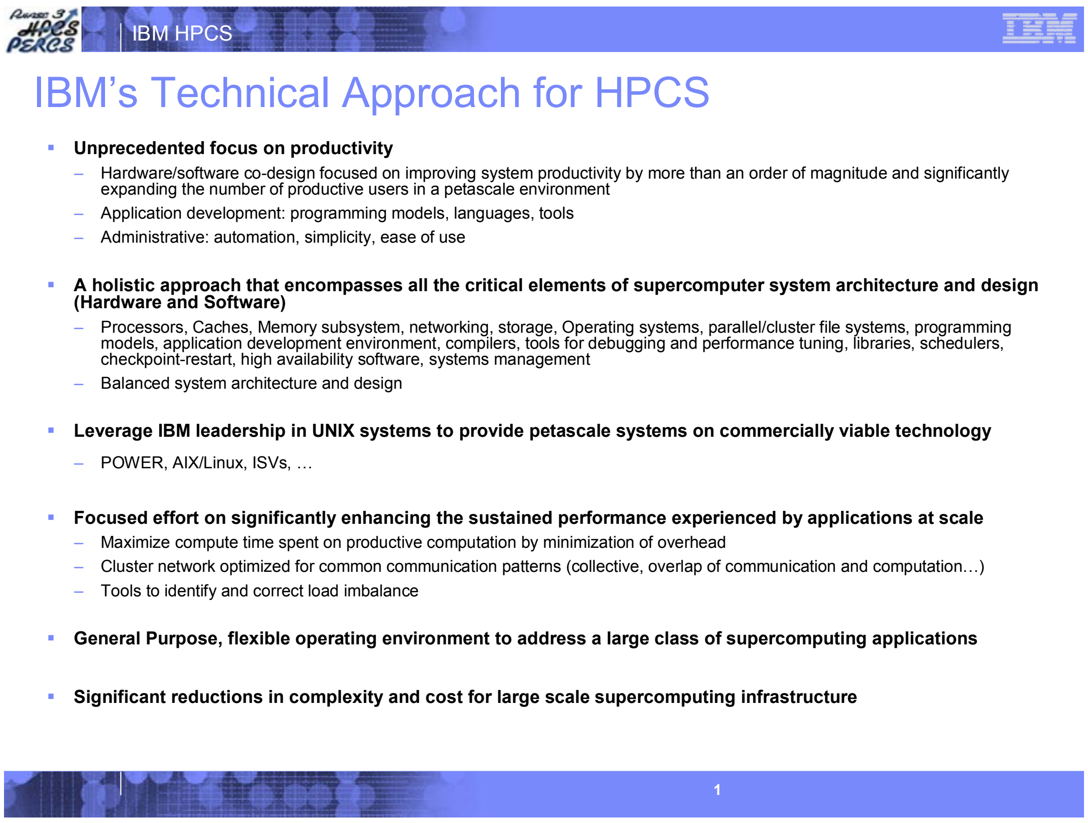
To better serve a variety of application requirements and to support more natural parallel programming idioms, the system provides a set of heterogeneous processing capabilities, each optimized for executing a different style of computation. Serial and latency sensitive computations are executed on commodity microprocessors, data-parallel computations with regular control flows are accelerated by vector processing, and parallel computations with irregular control flows are accelerated via massive multithreading.

The system supports low overhead, heavily-pipelined communication and synchronization, allowing fine-grained parallelization and enhancing scalability.

The processor and system architecture support a programming environment that greatly simplifies the parallel programming task via higher productivity languages and programming models, and innovative tools that ease debugging and performance tuning at large scales. The Cascade programming environment supports MPI, OpenMP, pthreads, SHMEM and Global Arrays, as well as the global address space languages Unified Parallel C and Co-Array Fortran. For the most productive programming experience, Cascade supports global-view programming models, which provide a parallel programming experience more similar to uniprocessor programming. The new Chapel language provides data and control abstractions that simplify parallel programming and create a clean separation between high-level algorithms and low-level details such as data decomposition and layout. This enables a programmer to first focus on expressing the parallelism inherent in the algorithm being implemented, and later redefine the critical data structures to exploit locality and processor affinity.

The IBM PERC System

The Figures 1.3 and 1.4 highlight the key features of IBM's PERC system that is under development through the HPCS program.



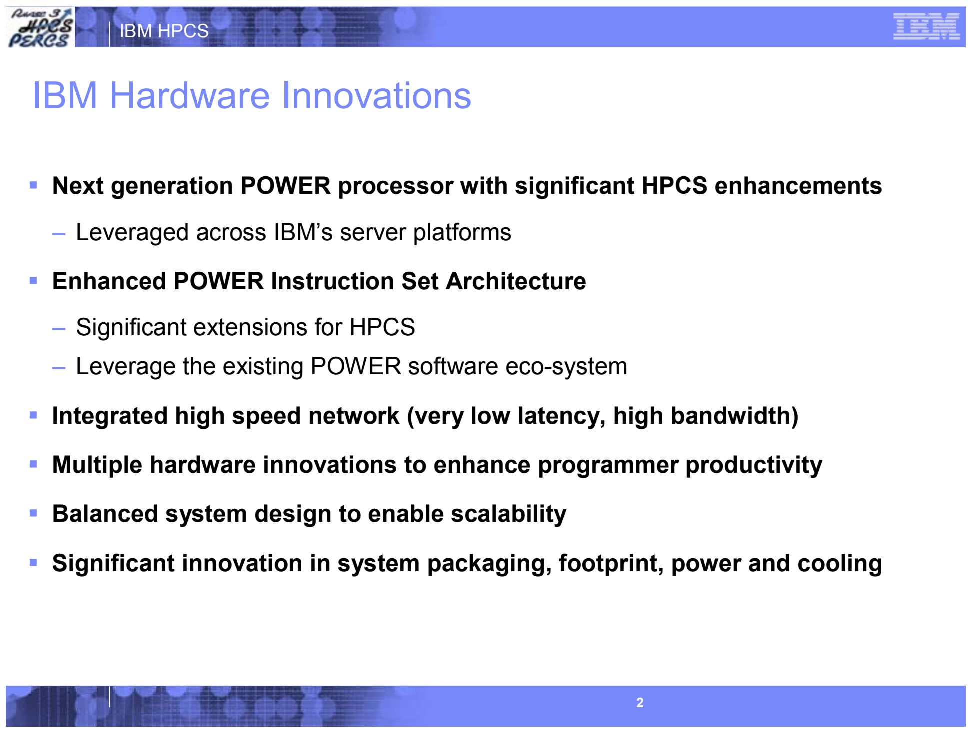
IBM HPCS

IBM's Technical Approach for HPCS

- **Unprecedented focus on productivity**
  - Hardware/software co-design focused on improving system productivity by more than an order of magnitude and significantly expanding the number of productive users in a petascale environment
  - Application development: programming models, languages, tools
  - Administrative: automation, simplicity, ease of use
- **A holistic approach that encompasses all the critical elements of supercomputer system architecture and design (Hardware and Software)**
  - Processors, Caches, Memory subsystem, networking, storage, Operating systems, parallel/cluster file systems, programming models, application development environment, compilers, tools for debugging and performance tuning, libraries, schedulers, checkpoint-restart, high availability software, systems management
  - Balanced system architecture and design
- **Leverage IBM leadership in UNIX systems to provide petascale systems on commercially viable technology**
  - POWER, AIX/Linux, ISVs, …
- **Focused effort on significantly enhancing the sustained performance experienced by applications at scale**
  - Maximize compute time spent on productive computation by minimization of overhead
  - Cluster network optimized for common communication patterns (collective, overlap of communication and computation…)
  - Tools to identify and correct load imbalance
- **General Purpose, flexible operating environment to address a large class of supercomputing applications**
- **Significant reductions in complexity and cost for large scale supercomputing infrastructure**

1

Figure 1.3 IBM HPCS overview



IBM HPCS

IBM Hardware Innovations

- **Next generation POWER processor with significant HPCS enhancements**
  - Leveraged across IBM's server platforms
- **Enhanced POWER Instruction Set Architecture**
  - Significant extensions for HPCS
  - Leverage the existing POWER software eco-system
- **Integrated high speed network (very low latency, high bandwidth)**
- **Multiple hardware innovations to enhance programmer productivity**
- **Balanced system design to enable scalability**
- **Significant innovation in system packaging, footprint, power and cooling**

2

Figure 1.4 IBM HPCS overview

As stated earlier, productivity, by its very nature, is difficult to assess because it depends upon the specifics of the end user mission, application, team composition, and end use or workflow. The challenge is to develop a productivity assessment strategy based on a mixture of qualitative and quantitative (preferred) analysis based metrics that will not only be used to evaluate the HPCS vendors but also adopted by the larger HPC community.

Figure 1.2 also shows a multi-year HPCS productivity initiative that was started in Phase II, funded by DARPA, DOE Office of Science, NNSA, NSA, and NSF, and led by Dr. Jeremy Kepner from MIT-Lincoln Laboratory. The Productivity Team was comprised of universities, laboratories, FFRDCs and HPCS Phase II vendors. Bi-annual public productivity conferences were held on a regular basis throughout the three year Phase II program. The HPCS Phase II Productivity Team projects can be loosely grouped as execution time and development time research elements. **The next sections will delineate an expanded set of productive research elements and findings resulting from the HPCS Phase II Productivity Team projects.** The representative research elements presented are performance benchmarking, system architecture modeling, productivity workflows/metrics and new languages.

In summary, the HPCS program represents a very unique partnership between DARPA, industry and the government end users (mission partners). Since this partnership represents a very different model from the past, what “it is not” is just as important as “what it intends” to be. The things the program is not are as follows:

- A One-off system. The HPCS system must be a viable commercial product.
- Meeting only one set of requirements. Aside from the varied requirements of the mission partners, the system must support a spectrum of a applications and configurations
- Available only at petascale. The system must scale from a single cabinet to very large configurations.
- Using only new languages. The HPCS system must also support existing programming languages, including C, C++ and Fortran and with MPI and existing PGAS languages.

2. Productivity Systems Modeling

The HPCS vision centers around the notion of computing systems with revolutionary hardware-software architectures that will enable substantially higher productivity than projected continuations of today's evolutionary system designs. Certainly a crucial component of productivity is the actual ("sustained") performance the revolutionary computing systems will be able to achieve on applications and workloads. Because these architectures will incorporate novel technologies to an unusually large extent, predicting application performance will be considerably more difficult than is the case for next-generation systems based on evolutionary architectures. Hence, the availability of sophisticated performance modeling techniques will be critically important for designers of HPCS computing systems and for the success of the HPCS program as a whole.

Performance models allow users to predict the running time of an application based on the attributes of the application, its input, and the target machine. Performance models can be used by system architects to help design supercomputers that will perform well on assorted applications. Performance models can also inform users which machines are likely to run their application fastest, and to alert programmers to performance bottlenecks that they can then attempt to remove.

The ***convolution problem*** in performance modeling asks how to predict the performance of an application on different machines, based on two things: (1) machine profiles consisting of rates at which a computer can perform various types of operations as measured by simple benchmarks; and (2) an application signature consisting of counts of various types of operations performed by the application. The underlying notion is that the performance of an application can be represented by some combination of simple benchmarks measuring the ability of the target machine to perform different kinds of operations on the application’s behalf.

There are three different methods for doing performance convolutions, each based on matrix operations, within the San Diego Supercomputing Center's Performance Modeling and Characterization (PmaC) framework for performance prediction. Each method is appropriate for answering a different set of questions related to correlating application performance to simple benchmark results. Each requires a different level of human insight and intervention. And each, in its own way, can be used to predict the performance of applications on machines where the real runtime is unknown.

The first method uses Least Squares fitting to determine how good a statistical fit can be made between observed runtimes of applications on different machines, using a set of machine profiles (measured by using simple benchmarks); the resulting application signatures can then be used within the framework for performance prediction. This method has the virtue of being completely automated.

The second method uses Linear Programming to fit the same input data as the first method (or similar input data). In addition, however, it can also mark input machine profiles and/or application runtimes as suspect. This corresponds to the real world situation in which one wants to make sense out of benchmarking data from diverse sources, including some that may be flawed. While potentially generating more accurate application signatures that are better for performance prediction, this method requires some user interaction. When the solver flags data as suspect, the user must either discard the data, correct it, or insist that it is accurate.

Instead of inference from observed application runtimes, the third method relies on instrumented application tracing to gather application signatures directly. While arguably the most general-purpose and accurate of the three methods, it is also the most labor intensive. The tracing step is expensive compared to measuring the un-instrumented application runtimes, as used by the Least Squares and Linear Programming methods to generate application signatures. Moreover, unlike the first two methods, forming the performance model requires substantial expert interaction to "train" the convolution framework, though subsequently performance predictions can be done automatically.

Finally we demonstrate how a judicious mix of these methods may be appropriate for large performance modeling efforts. The first two allow for broad workload and HPC asset characterizations, such as understanding what system attributes discriminate performance across a set of applications, while the last may be more appropriate for very accurate prediction and for guiding tuning efforts. To evaluate these methods, we tested them on a variety of HPC systems and on applications drawn from the Department of Defense's Technical Insertion 2006 (TI-06) application workload [5].

2.1 Problem Definition and Unified Framework

As an example of a simple pedagogical convolution, consider Equation 1. Equation 1 predicts the runtime of application a on machine m by combining three of application a 's operation counts (the number of floating-point, memory and communication operations) with the corresponding rates at which machine m can perform those operations.

$$\mathrm{Runtime}_{a,m} \approx \frac{\mathrm{FloatOps}_a}{\mathrm{FloatOpRate}_m} + \frac{\mathrm{MemoryOps}_a}{\mathrm{MemoryOpRate}_m} + \frac{\mathrm{CommOps}_a}{\mathrm{TransferRate}_m} \tag{1}$$

In practice, FloatOpRate, MemoryOpRate and TransferRate could be determined by running a set of simple synthetic benchmarks such as HPL, STREAM, and EFF_BW, respectively, from the HPC Challenge benchmarks [10]. Likewise, FloatOps, MemoryOps, and CommunicationOps could be measured for the application using hardware and software profiling tools such as the PMaC MetaSim Tracer [3] for floating and memory operations, and MPIDTrace [1] for communication events.

We could generalize Equation 1 by writing it as in Equation 2, where OpCount represents a vector containing the three operation counts for application a, Rate is a vector containing the corresponding three operation rates for machine m, and $\oplus$ represents a generic operator. This generic operator could, for example, take into account a machine's ability to overlap the execution of two different types of operations.

$$\begin{aligned}\text{Time}_{a,m} \approx P_{a,m} &= \frac{\text{OpCount}(1)}{\text{Rate}(1)} \oplus \frac{\text{OpCount}(2)}{\text{Rate}(2)} \\ &\oplus \frac{\text{OpCount}(3)}{\text{Rate}(3)}\end{aligned} \tag{2}$$

If the number of operation counts used in Equation 2 is expanded to include other performance factors, such as the bandwidth of strided accesses to L1 cache, the bandwidth of random-stride accesses to L1 cache, the bandwidth of strided accesses to main memory, network bandwidth, etc., we could represent application a's operation counts by making OpCount a vector of length c, where c is the total number of different operation types represented for application a. We could further expand OpCount to represent more than one application by making OpCount a matrix of dimension $c \times n$, where n is the total number of applications characterized. Similar expansion could be done for Rate, making it a $k \times c$ matrix, where k is the total number of machines, each of which is characterized by c operation rates. This would make P a $k \times n$ matrix in which P_{ij} is the predicted runtime of application i on machine j. That is, the generalized Equation 2 represents the calculation of predicted runtimes of n different applications on k different machines and can be expressed as $P = \text{Rate} \otimes \text{OpCount}$. Since each column in OpCount can also be viewed as the application signature of a specific application, we refer to OpCount as the application signature matrix, A. Similarly, each row of Rate is the machine profile for a specific machine, and so we refer to Rate as the machine profile matrix, M. Now the convolution problem can be written as $P = M \otimes A$. In expanded form, this looks like:

$$\begin{bmatrix} p_{1,1} & \cdots & p_{1,n} \\ p_{2,1} & \cdots & p_{2,n} \\ p_{3,1} & \cdots & p_{3,n} \\ \vdots & \ddots & \vdots \\ p_{k,1} & \cdots & p_{k,n} \end{bmatrix} = \begin{bmatrix} m_{1,1} & \cdots & m_{1,c} \\ m_{2,1} & \cdots & m_{2,c} \\ m_{3,1} & \cdots & m_{3,c} \\ \vdots & \ddots & \vdots \\ m_{k,1} & \cdots & m_{k,c} \end{bmatrix} \otimes \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{c,1} & \cdots & a_{c,n} \end{bmatrix} \tag{3}$$

Given Equation 3 as the general convolution problem, the relevant questions are how to determine the entries of M and A, and what to use for the $\otimes$ operator to generate accurate performance predictions in P?

Populating M is fairly straightforward, at least if the machine or a smaller prototype for the machine exists. Traditionally, this has been done by running simple benchmarks. It should be noted that determining c, the smallest number of benchmarks needed to accurately represent the capabilities of the machine, is generally considered an open research problem [3]. In this work, to populate M, we used the *netbench* and *membench* synthetic benchmarks from the TI-06 benchmark suite [22]. These benchmarks can be considered a superset of the HPC Challenge Benchmarks. For example, Figure 2 plots the results of running membench on an IBM system. We could then populate M with several memory bandwidths corresponding to L1 cache bandwidth for strided loads, L1-L2 (an intermediate bandwidth), L2 bandwidth, etc., to represent the machine's capabilities to service strided load requests from memory; similarly, rates for random access loads, stores of different access patterns, floating-point and communication operations can be included in M. One could think of the upper curve in Figure 2 as giving the results of running the STREAM benchmark from the HPC Challenge Benchmarks at different sizes ranging from small to large, and the lower curve as the serial version of the RandomAccess benchmark run in the same way. (Some implementation details differ between the TI-06 synthetics and HPC Challenge, but the overall concepts and the rates they measure are the same.)

Populating A is less straightforward. While traditionally users have consulted performance counters to obtain operation counts, this may not reveal important operation subcategories such as the access pattern or locality of memory operations. For example, we can see from Figure 2 that not all memory operations are equal; rates at which machines can complete memory operations may differ by orders-of-magnitude, depending on where in the memory hierarchy they fall. An alternative to performance counters is application tracing via code instrumentation. Tracing can, for example, discover memory addresses and locality, but is notoriously expensive [7]. The methods we propose in this section find the entries of A using three different methods, each with different tradeoffs in accuracy versus effort.

2.2 Methods to Solve the Convolution Problem

In this work, we investigate three methods for calculating A and determining the $\otimes$ operator. We classify the first two methods as *empirical* and the third one as *ab initio*. Empirical methods assume M and some values of P are known, and then derive the matrix A. Matrix A can then be used to generate more values of P. Ab initio methods, on the other hand, assume both M and A are known and then calculate P from first principles. In addition to this classification, the first two methods may be considered *top down* in that they attempt to resolve a large set of performance data for consistency, while the last may be considered *bottom up* as it attempts to determine general rules for performance modeling from a small set of thoroughly characterized loops and machine characteristics.

2.2.1 Empirical Method

Although traditionally we assume P is unknown and its entries are to be predicted by the model, in practice some entries of P are always measured directly by timing application runs on real machines. This may be done simply to validate a prediction, although the validation may be done some time in the future, as is the case when predicting runtimes on proposed machines. A key observation is that running an application on an existing machine to find an entry of P is generally significantly easier than tracing an application to calculate the P entry through a model. This suggests that we treat some entries of P as known for certain existing systems and M as also known via ordinary benchmarking effort, rather than treating P as an unknown and A as knowable only via extraordinary tracing effort. This, combined with assumptions about the structure of the convolution operator $\otimes$, allows us to solve for A. Once A is known for the applications of interest, it can be convolved with a new M' to predict performance on these other machines, where

M' is just M with rows for the new machines, for which simple synthetic benchmarks may be known or estimated but for which full application running times are unknown.

We refer to the methods that treat A as the unknown as *empirical* in the sense that one can deduce the entries of a column of A by observing application runtimes on a series of machines that differ by known quantities in their ability to perform operations in each category. As an example, intuitively, if an application has very different runtimes on two systems that are identical except for their network latency, we may deduce that the application's sensitivity to network latency comes from the fact that it sends numerous small messages.

We formalize this intuition and demonstrate two different techniques for empirical convolution. In both methods we assume that there is a set of machines on which we have gathered not only the synthetic benchmarks used to fill in the matrix M, but also actual runtimes for n applications of interest in P. We further assume that the operator $\otimes$ is the matrix multiplication operator. We now describe two different approaches for using P and M to solve for entries of the application matrix, A, within a reasonable range of error.

The first empirical approach uses Least Squares to find the entries of A, and is particularly appropriate when the running times are known on more machines than we have benchmark data for. The second empirical approach uses Linear Programming to find the entries of A and can be useful in the under-constrained case where we have real runtimes on fewer machines than we have benchmark data for. In addition to finding entries in the application matrix A, both methods can also be used to address questions such as:

- What is the best fit that can be achieved with a given assumption about the convolution? (For example, we may assume the convolution operator is a simple dot-product and operation counts are machine-independent for each application.)

- Can one automatically detect outliers, as a way to gain insight into the validity of benchmark and runtime data from various sources?

- Can one calculate application weights for a subset of the systems and use those weights to accurately predict runtimes on other systems?

- What properties of systems are most important for distinguishing their performance?

2.2.2 Solving for A using Least Squares

Consider solving the matrix equality $P = MA$ for A. We can solve for each column of A individually (i.e., $P_i = MA_i$), given the (plausible) assumption that the operation counts of one application do not depend on those of another application. If we further decide to

compute application operation counts that minimize the 2-norm of the residual $P_i - MA_i$, then the problem becomes the much-studied Least-Squares problem. Furthermore, because only non-negative application operation counts have meaning, we can solve $P_i = MA_i$ as a nonnegative Least Squares problem. These problems can be solved by a technique described in [9] and implemented as `lsqnonneg` in Matlab.
In practice, before applying the nonnegative Least Squares solver, we normalize both the rows and columns of the equation with respect to the largest entry in each column. Rescaling the columns of M so that the largest entry in each column is 1 allows us to weight different operations similarly, despite the fact that the cost of different types of operations can vary by orders of magnitude (e.g., network latency versus time to access the L1 cache). Rescaling the rows of M and P_i so that the entries of P are all 1 allows us to normalize for different runtimes.

The Least Squares approach has the advantage of being completely automatic, as there are no parameters to change or constraints that may need discarding. Thus, it also partially answers the question: if all the benchmark and runtime data are correct, how well can we explain the running times within the convolution framework? However, if some of the data is suspect, the Least Squares method will attempt to find a compensating fit rather than identifying the suspect data.

2.2.3 Solving for A using Linear Programing

Unlike the Least Squares Method that seeks the minimum quadratic error directly, the Linear Programming Method is more subtle — and it can also be more revealing. There are various ways to rephrase Equation 3 as a Linear Programming problem; in our implementation, for every i and j ($1 \le i, j \le n$), Equation 3 is relaxed to yield the following two inequalities:

$$p_{ij} \cdot (1-\beta) \ge m_i \oplus a_j$$
$$p_{ij} \cdot (1-\beta) \le m_i \oplus a_j$$

where $0 < \beta < 1$ is an arbitrary constant and each element a_j is a non-negative variable.

Therefore, each pair of inequalities defines a stripe within the solution space, and the actual solution must lie within the intersection of all n stripes. Should any stripe fall completely outside the realm of the others, no solution that includes that machine-application pair exists. Given a simplifying assumption that similar architectures have similar frequencies of operations per type for a given application, it is expected that the intersection of the stripes will not be null, since the application execution time p_{ij} will likely be a direct result of synthetic capability m_i (when neglecting more complex, possibly non-deterministic execution properties such as overlapping operations, pre-fetching, and speculative execution). A null solution, therefore, suggests that an error may lie in one of the execution times or one of the synthetic capability measurements.

To determine the "optimal" solution for a_j, the intersection of all stripes must result in a bounded space. In such a case, the intersection vertices are each tested via an objective function to determine the best solution. For this implementation, a minimum is sought for

$$f(a_j) = \sum_{i=1}^{k} (m_i \oplus a_j) \tag{4}$$

in order to force the estimate for the application times to be inherently faster than the actual application times. The error for each may then be associated with operation types such as I/O reads and writes that are not represented in the set of basic operations.

In applying the Linear Programming method, the value for β was increased until all stripe widths were sufficiently large, in order to achieve convergence. Estimates for the application times were then calculated for each machine (1) to determine the overall extent of the estimation error and (2) to identify any systems with outlying error values when clustering error percentages using a nearest-neighbor technique. Any system identified in (2) was removed from consideration, since an error in its application or synthetic benchmarks was suspected. This methodology was applied iteratively until the minimum value for β that achieved convergence and the overall estimation error were both considered to be small.

2.2.4 Ab Initio Method

Methods that assume P is unknown are referred to here as *ab initio*, in the sense that the performance of an application running on a the system is to be determined from its first principles. The assumption, then, is that both M and A are known but the generic $\otimes$ operator and P are not known.

To separate concerns we split the problem of calculating P into two steps. The first step is to predict the execution time of the parallel processors between communication events. Following the format of Equation 3, memory and floating-point operations are gathered by tracing and are further fed through a simulator to propagate the corresponding entries of a matrix A'. Each column of A' then holds floating-point operations and memory operations (but not communication operations), broken down into different types, access patterns, locality, etc., for a particular application. P' is obtained by multiplying A' with M, and thus a row of P' represents the application's predicted time spent doing work on-processor during execution on the machines of M. In the second step, the Dimemas [6] simulator processes the MPI trace data and P' in order to model the full parallel execution time. The output from Dimemas is then the final calculated execution time for the application(s) on the target machine(s) (P).

In the remainder of this section, we describe how the trace data is used to determine the entries of A' and how A' is used to calculate the entries of P'. Since the time spent doing floating-point operations tends to be small compared to the time spent doing memory operations in large scale parallel applications, we focus on describing how we determine the entries of A' related to memory performance.

Our approach is to instrument and trace the applications using the PMaC MetaSim Tracer, and then to use the PMaC MetaSim Convolver to process the traces in order to find the entries in A'. Details of the PMaC MetaSim Tracer and the processing of the trace by the PMaC MetaSim Convolver can be found in [19].

Before preceding it is important to note that the ab initio methods relax two constraints of the Least Squares and Linear Programming methods. First, an application's trace, particularly its memory trace, is fed to a cache simulator for the machine(s) to be predicted. This means a column of A' can be different on different machines., This represents a notable sophistication over the empirical methods: it is no longer assumed that operation category counts are the same on all machines (for example, machines with larger caches will get more operations that hit in cache). Second, rather than assuming a simple combining operator such as a dot product, the convolver can be trained to find a better operator that predicts performance with much smaller prediction errors. This operator may, for example, allow overlapping of floating-point operations with memory operations – again a notable advance in sophistication that is more realistic for modern machines.

Since tracing is notoriously expensive, we employ cross-platform tracing in which the tracing is done only once on a single system, but the cache structure of many systems is simulated during tracing. Figure 1 shows some of the information that MetaSim Tracer collects for every basic block (a basic block is a straight run of instructions between branches) of an application. Fields that are assumed under the cross-platform tracing assumption to be the same across all machines are collected or computed from direct observation; but fields in the second category are calculated by feeding the dynamic address stream on-the-fly to a set of cache simulators, unique to each machine.

The MetaSim Convolver predicts the memory performance of each basic block by mapping it to some linear combination of synthetic benchmark memory performance results (entries of M) using the basic block fields, such as simulated cache hit rates and stride access pattern, as shown in Figure 2.1. This convolver mapping is implemented as a set of conditions to be applied to each basic block to determine which bandwidth region and curve of Figure 2.1 to use for its estimated performance. A sample of one of these conditions for the L2 cache region on the ARL P690 system is shown on Figure 2.1. The main advance on this method described in this work is to refine these conditions to improve prediction accuracy as described next. The rules in Figure 2.1 further exemplify the conditions that were developed for each region and interpolation area between regions of the membench curves.

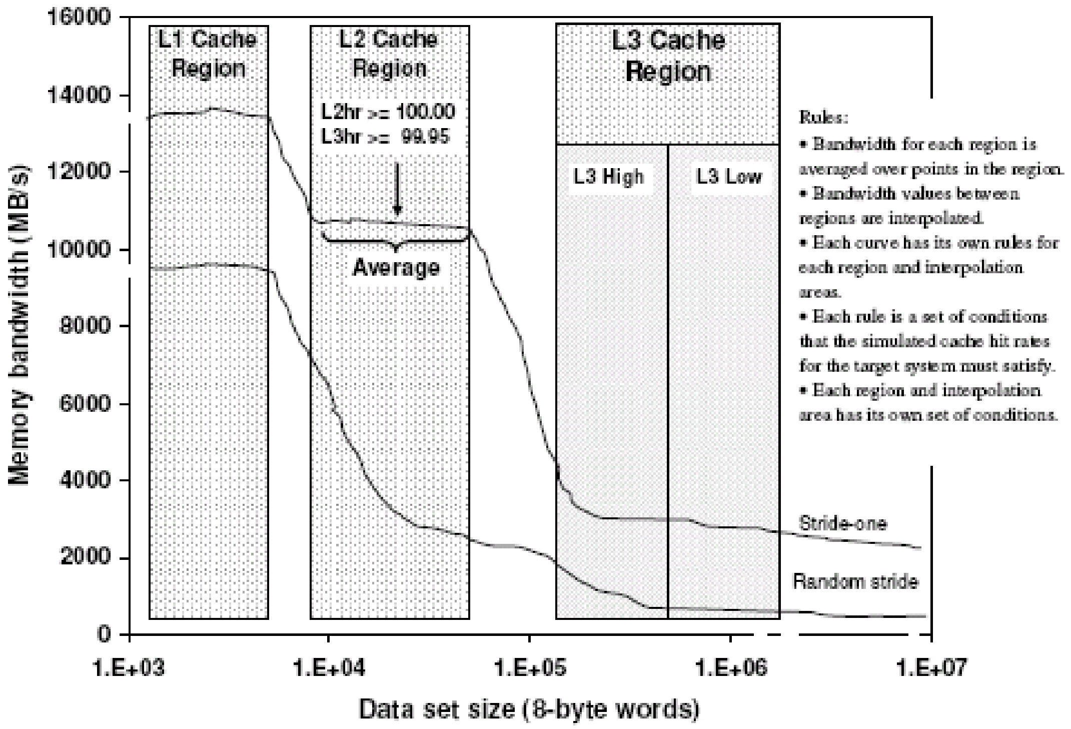


Figure 2.1. The strided (upper) and random (lower) memory bandwidth test from membench taken from an IBM P690 system at Army research Laboratory (ARL), along with example rules for mapping a basic-block to its expected memory performance.

In order to determine and validate the best set of conditions for minimizing prediction error, a small experimental $P^{measured} = MA'$ problem was defined. We chose several computational loops as a "training set" to develop and validate the convolving conditions. We chose 40 computational loops from two parallel applications, HYCOM and AVUS, from the TI-06 application suite; then execution times for these loops on 5 systems were measured to propagate the entries of $P^{measured}$. The loops were chosen judiciously, based on their coverage of the trace data space: their constituent basic blocks contain a range of hit rates, different randomness, etc. We then performed a human-guided iteration, trying different sets of conditions the MetaSim Convolver could implement to determine each loop's A' and thence entries in $P^{predicted}$. Thus, we were looking for rules to map basic blocks to expected performance; the rules were constrained to make sense in terms of first principles properties of the target processors. For example, a loop's memory work cannot obtain higher than measured L1 memory performance, or lower than measured main memory performance. More subtly, the L1 hit rate value required to assign a basic block to get L1 cache performance could not be less than the hit rate value that would assign L3 cache performance.

We sought then to determine conditions in such a way that the generated elements of A' would produce the most accurate calculated $P^{predicted}$. In other words, we looked for conditions to minimize:

$$\text{Total error} = \sum_{i,j}^{m,n} | (P_{i,j}^{measured} - P_{i,j}^{predicted}) / P_{i,j}^{measured} | \tag{5}$$

where m is the number of machines (5) and n is the number of loops (40).

Finally, having developed the conditions for the training set that minimized total error, we used the same conditions to convolve and to predict the full AVUS and HYCOM applications, as well as a larger set of applications on a larger set of machines.

2.2.5 Experimental Results

To evaluate the usefulness of the *empirical* and *ab initio* methods, we tested both on several strategic applications run at several processor counts and inputs on the systems listed in Table 2.1. We chose several applications from the Technical Insertion 2006 (TI-06) application workload that covered an interesting space of computational properties, such as ratio of computation to communication time, ratio of floating-point operations to memory operations, and memory footprint size. None of these codes on the inputs given are I/O intensive; thus we do not model I/O in the remainder.

The applications used are AVUS, a code used to determine the fluid flow and turbulence of projectiles and air vehicles; CTH, which models complex multidimensional, multiple-material scenarios involving large deformations or strong shock physics; HYCOM, which models all of the world's oceans as one global body of water; OVERFLOW, which is used for computation of both laminar and turbulent fluid flows over geometrically complex, non-stationary boundaries; and WRF, which is a next-generation mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs.

The applications were run on two different inputs each (DoD designations "Standard" and "Large") and on 3 different processor counts between 32 and 512 for each input on all 20 of the systems listed. More information on these applications can be found at [3] and [5]. It should be clear that populating the $P^{measured}$ matrix for the Least Squares and Linear Programming methods required in this case measuring about 600 application runtimes (20 systems, 5 applications, 2 inputs each, 3 cpu counts each). These are full applications that run, on average, about one or two hours each, depending on input and cpu count. The $P^{measured}$ values were therefore collected by a team of people from the Department of Defense High Performance Computing Modernization Program (HPCMP) centers. The authors populated the M matrix by running the membench and netbench benchmarks on these same systems.

In testing the empirical methods, we tried several variants of the M matrix, in part to explore the relationship between the number of columns in M (and rows in A) and the resulting accuracy. This complements investigations in [3], where the authors studied the smallest number of benchmarks required to accurately represent the capabilities of machines. For example, we tried 10 - and 18 -column variants of the M matrix, both based on the same machine benchmark data. The one with 10 columns had 8 measures pertaining to the memory subsystem from membench (i.e., drawn from plots similar to Figure 2.1), and 2 being the off-node bandwidth and the latency of the interconnect from netbench, as described above. The set with 18 columns had 14 measures pertaining to the memory subsystem (i.e., taking more points from the membench curve in Figure 2.1), 2 for off-node bandwidth and latency, and 2 for on-node bandwidth and latency. From the synthetic measures pertaining to the memory subsystem, half were chosen from strided membench results and the other half were chosen from random access membench results.

We now discuss the data in Table 2.1, which summarizes the results of using the empirical methods to understand our data set.

2.2.6 Fitting the data using Least Squares

Error Summary	**Average Absolute Error**	
Systems	LS	LP
ASC_SGI_Altix	4%	8%
SDSC_IBM_IA64	12%	—
ARL_IBM_Opteron	12%	8%
ARL_IBM_P3	4%	4%
MHPCC_IBM_P3	6%	6%
NAVO_IBM_P3	9%	6%
NAVO_IBM_p655 (Big)	5%	6%
NAVO_IBM_p655 (Sml)	5%	5%
ARSC_IBM_p655	4%	2%
MHPCC_IBM_p690	8%	7%
NAVO_IBM_p690	7%	9%
ARL_IBM_p690	10%	6%
ERDC_HP_SC40	6%	8%
ASC_HP_SC45	5%	4%
ERDC_HP_SC45	5%	6%
ARSC_Cray_X1	8%	5%
ERDC_Cray_X1	51%	3%
AHPCRC_Cray_X1E	14%	—
ARL_LNX_Xeon (3.06)	6%	8%
ARL_LNX_Xeon (3.6)	16%	8%
Overall Error	**%**	**%**

Table 2.1. Average absolute error for all applications tested on 20 DoD systems. Format of column 1 is acronym of Department of Defense computer center-computer manufacturer-processor type.

Because the Least Squares method computes A to minimize overall error, it can be used to answer the question, How well can we explain measured entries of P as a function of measured entries of M, assuming a standard dot-product operator and machine-independent application signatures.? The Least Squares column (denoted as *LS*) in Table 2.1 answers this question for our set of data. When averaged over all the applications and inputs and processor counts, we found that the performance and the performance differences of the applications on these machines could be represented within about 10% or 15%. As noted previously, we tested this method with both a 10-and an 18 column M matrix. In looking at the errors averaged for each case separately, we found that the results were only slightly better with the latter.

Overall the LS results demonstrate that one can characterize the observed performance differences of many applications on many different machines by using a small set of benchmark measurements (the M matrix) whose entries pertain to each machine's memory and interconnect subsystems. The ERDC Cray X1 is the only machine that Least Squares seems unable to fit well. This is discussed in the next section.

2.2.7 Detecting Outliers By Using Linear Programming

The Linear Programming method also tries to find the A that best fits the entries in M and P under the same assumptions as with the Least Squares method, but Linear Programming has the advantage of being able to identify entries in M and P that seem suspect.

When we first computed the errors given in Table 2.1 using the initial measured entries of M, large errors for the ASC SGI Altix and the ARL IBM Opteron led us to question the benchmark data on those machines. After rerunning the benchmarks on those machines and recalculating the errors using the Least Squares and Linear Programming methods, we ended up with the (much improved) results in Table 2.1. The empirical methods were able to identify suspicious benchmark data. Upon further investigation, we found that the membench benchmark had originally been run on those two machines with a poor choice of compiler flags, resulting in unrealistically low bandwidths.

We note that in generating the results in Table 2.1, the Linear Programming (denoted as *LS*) method still flagged the SDSC IBM IA64 and the AHPCRC Cray X1E as being nonconforming (thus these machines are omitted from the LP column), suggesting that there are inconsistencies in either the benchmark data or the runtime data on those machines. Unfortunately, recollecting benchmark data on these machines did not improve the results, leading us to surmise that the problem lies in the application runtimes rather than in the benchmark. Looking specifically at the AHPCRC Cray X1, we observe that it was flagged when the ERDC Cray X1 was not. It is possible that the codes run on the AHPCRC Cray X1 were not properly vectorized. We have not tested our hypotheses yet,- because rerunning the applications is a time- and resource-intensive process usually done by the teams at each center once per year (unlike rerunning the benchmarks, which is much easier), and has not been completed at this time.

Both these sets of results could be further interpreted as saying that, if one is allowed to throw out results that seem due to errors in the input data (either benchmarks or application runtimes), one may attribute as much as 95% of the performance differences of these applications on these machines to their benchmarked capabilities on less than 20 simple tests. Results like these have implications for how much benchmarking effort is really required to distinguish the performance capabilities of machines.

Both these methods, taken together with the earlier results in [3], at least partially answer the question, “what properties of systems are most important to distinguish performance”? In [3] it was shown that three properties (peak floating-point issue rate, bandwidth of strided main-memory accesses, and bandwidth of random main-memory accesses) can account for as much of 80% of observed performance on the TI-05 applications and machines (a set with substantial similarities to and overlap with TI-06). The results in this work can be seen as saying that another 10% (for 90% or more total) is gained by adding more resolution to the memory hierarchy and by including communication.

2.3 Performance prediction

We now describe how the empirical and ab initio convolution methods can be used to predict overall application performance. In what follows we refer to both a $P^{measured}$, which consists of measured runtimes that are used to generate either A (for the empirical methods) or to improve the convolver conditions (for the ab initio method), and to a $P^{predicted}$, which consists of runtimes that are subsequently predicted.

2.3.1 Empirical methods

To predict the performance of an application on a machine by using empirical convolution methods, we first determine the application signature by using runtimes in $P^{measured}$ that were collected on other machines. We then multiply the application signature with the characteristics of the new machine in order to arrive at a predicted runtime, $P^{predicted}$. If we have the actual measured runtime on the new machine, we can use it to evaluate the accuracy of the predicted time.

Generally we found this method to be about 90% accurate if the machine being predicted was architecturally similar to machines in $P^{measured}$. For example, RISC-based architectures can be well predicted using an A derived from runtimes and benchmark results of other RISC machines. As an example, Table 2.2 shows the error in $P^{predicted}$, using both the Least Squares and Linear Programming methods, with sets of 10 and 18 machine characteristics, to predict the performance of AVUS and HYCOM on the ERDC SC45. (In this case we report signed error, as we are not averaging, and so no cancellation in the error is possible.) The measured runtimes on the ERDC SC45 were not included in $P^{measured}$, but were used to calculate the error in $P^{predicted}$. We found the predictions using both methods to be generally within 10% of the actual runtimes.

Error	10 bin		18 bin	
Application	LS	LP	LS	LP
avus 32	3%	8%	-5%	10%
avus 64	1%	7%	-6%	6%
avus 128	3%	10%	-3%	8%
cth 32	-9%	-8%	3%	-1%
cth 64	-8%	-3%	1%	4%
cth 96	-13%	-1%	-1%	2%

Table 2.2. Signed error in predicting the performance of AVUS and CTH on different processor counts on the ERDC SC45, using application signatures generated through empirical methods.

In contrast, Table 2.3 gives an example where $P^{measured}$ does not contain a machine architecturally similar to the machine being predicted. Shown are the results of using the Least Squares and Linear Programming methods to predict performance on the ASC SGI Altix. Unlike almost all of the other machines in Table 2.1, the Altix is neither a RISC-based machine nor a vector machine. Rather, it uses a VLIW and has other unique architectural features too extensive to describe here. So, once the ASC SGI Altix is removed from $P^{measured}$, there is no architecturally similar machine other than the SDSC IA64 (already flagged as suspect). In this case we see that the predicted runtime is only rarely within even 20% of the actual runtime. This seems to suggest that inclusion of an architecturally similar machine in $P^{measured}$ is crucial for determining a useful A for subsequent prediction.

Error	10 bin		18 bin	
Application	LS	LP	LS	LP
avus 32	27%	42%	-117%	-117%
avus 64	18%	-27%	-135%	-100%
avus 128	16%	32%	-129%	-80%
cth 32	-108%	-64%	-56%	8%
cth 64	-98%	30%	-35%	-35%
cth 96	-171%	-170%	-47%	-44%

Table 2.3. Signed error in predicting the performance of AVUS and CTH on different processor counts on the ASC Altix, using application signatures generated through empirical methods.

Tables 2.2 and 2.3 demonstrate that empirical methods are particularly useful for prediction if the new machine has an architecture that is similar to those used in generating the application signature. However, if the new machine has a unique architecture, accurate prediction through these methods may be problematic.

2.3.2 Ab initio methods

Table 2.4 gives the results of applying the convolver, trained against a 40-computational-loop training set, to predict the performance of all of the applications on all of the machines listed in Table 2.4. So far, 9 RISC-based machines have been trained in the convolver, and work is ongoing to add additional systems and architecture classes. Once the convolver is trained for a system, any application trace can be convolved to produce a prediction. It should be clear from the table that this involved 270 predictions (5 applications, 2 inputs, 3 cpu counts, 9 machines) at an average of 92% accuracy. Since measured application runtimes can vary by about 5% or 10%, accuracy greater than that shown in Table 2.4 may not be possible unless performance models take into account contention on shared resources and other sources of variability.

Systems	average error
ARL_IBM_Opteron	11%
NAVO_IBM_P3	7%
NAVO_IBM_p655 (Big)	6%
ARSC_IBM_p655	4%
MHPCC_IBM_p690	8%
NAVO_IBM_p690	18%
ASC_HP_SC45	7%
ERDC_HP_SC45	9%
ARL_LNX_Xeon (3.6)	5%
Overall Error	**%**

Table 2.4. Absolute error averaged over all applications for the computational loop study.

Although the ab initio method "knows" nothing about the performance of any application on any real machine, it is more accurate than the empirical methods. Thus it seems the power and pure predictive nature of the ab initio approach may sometimes justify its added expense and complexity. Of further note is that the ab initio approach actually constructs an overall application performance model from many small models of each basic block and communications event. This means the models can be used to understand where most of the time is spent and where tuning efforts should be directed. The empirical methods do not provide such detailed guidance.

2.4 Related Work

Several benchmarking suites have been proposed to represent the general performance of HPC applications. Besides those mentioned previously, the best known are perhaps the NAS Parallel [2] and the SPEC [20] benchmarking suites, the latter of which is often used to evaluate micro-architecture features of HPC systems. A contribution of this work is to provide a framework for evaluating the quality of a spanning set for any benchmark suite (i.e., its ability to attribute application performance to some combination of its results).

Gustafson and Todi [8] performed seminal work relating “mini-application” performance to that of full applications. They coined the term “convolution” to describe this general approach; but they did not extend their ideas to large-scale systems and applications, as this work does. McCalpin [12] showed improved correlation between simple benchmarks and application performance, but did not extend the results to parallel applications as does this work.

Marin and Mellor-Crummey [11] show a clever scheme for combining and weighting the attributes of applications by the results of simple probes, similar to what is implemented here, but their application studies were focused primarily on “mini application” benchmarks, and were not extended to parallel applications and systems.

Methods for performance evaluations can be broken down into two areas [21]: structural models and functional/analytical models. A fairly comprehensive breakdown of the literature in these two areas is provided in the Related Work section of Carrington, et al. [3], and we direct the reader’s attention there for a more thorough treatment.

Saavedra [15, 16, 17] proposed applications modeling as a collection of independent Abstract FORTRAN Machine tasks. Each abstract task was measured on the target machine and then a linear model was used to predict execution time. In order to include the effects of memory system, they measured miss penalties and miss rates to include in the total overhead. These simple models worked well on the simpler processors and shallower memory hierarchies of the mid to late 1990s.

For parallel system predictions, Mendes [13, 14] proposed a cross platform approach. Traces were used to record the explicit communications among nodes and to build a directed graph based on the trace. Sub-graph isomorphism was then used to study trace stability and to transform the trace for different machine specifications. This approach has merit and needs to be integrated into a full system for applications tracing and modeling of deep memory hierarchies in order to be practically useful today.

2.5 Conclusions

We presented a general framework for the convolution problem in performance prediction and introduced three different methods that can be described in terms of this framework. Each method requires different amounts of initial data and user input, and each reveals different information.

We described two empirical methods which assume some runtimes are known, and used them to determine application characteristics in different ways. The Least Squares method determines how well the existing data can be explained given particular assumptions. The Linear programming method can additionally correctly identify systems with erroneous benchmarking data (assuming the architectures of the target systems are roughly similar). Both can generate plausible fits relating the differences in observed application performance to simple performance characteristics of a broad range of machines. Quantitatively it appears they can attribute around 90% of performance differences to 10 or so simple machine metrics. Both empirical methods can also do fairly accurate performance prediction on machines whose architectures are similar to some of the machines used to determine application characteristics.

We then addressed the situation where real runtimes of the applications are not available, but where there is an expert who understands the target systems. We described how to use an ab initio approach in order to predict the performance of a range of applications on RISC machines with good accuracy, using just timings on a set of representative loops on representative applications (in addition to machine benchmarks) and a detailed report of the operations of the basic blocks that make up the loops used to train the convolver. This last method is capable of generating accurate predictions across a wide set of machines and applications.

It appears empirical approaches are useful for determining how cohesive a large quantity of application and benchmarking performance data from various sources is, and that, further, reasonable effort may attribute 90% or so of observed performance differences on applications to a few simple machine metrics. The more fully predictive ab initio approach is more suitable for very accurate forecasting of application performance on machines for which little full application runtime data is available.

3. Productivity Evaluation on Emerging Architectures

The future systems from DARPA’s High Productivity Computing Systems program will present a new level of architectural and programming diversity beyond existing multicore microprocessor designs. In order to prepare for the challenges of measuring productivity on these new devices, we created a project to study the performance and productivity of computing devices that will reflect this diversity: the STI Cell processor, Graphical Processing Units (GPU), and Cray’s MTA multithreaded system.

We believe that these systems represent an architectural diversity more similar to the HPCS systems than do existing commodity platforms – which have generally been the focus of evaluations of productivity to date. Homogenous multi-core systems require coarse-grain, multi-threaded implementations, while GPUs and Cell systems require users to manage parallelism and orchestrate data movement explicitly. Taken together, these attributes form the greatest challenge for these architectures in terms of developer productivity, code portability, and performance. In this project, we have characterized the relative performance improvements and required programming effort for two diverse workloads across these architectures: contemporary multi-core processors, Cell, GPU, and MTA-2. Our initial experiences on these alternative architectures (e.g., STI CELL, NVIDIA graphical processors, Cray MTA) lead us to believe that these evaluations may have to be very intricate and require a substantial investment of time to port and optimize benchmarks. These architectures will span the range of the parameters for development and execution time, and force us to understand the sensitivities of our current measurement methodologies. For example, consider the complexity of writing code for today’s CELL system or graphics processors. Initial HPCS systems may be equally challenging.

Contemporary multi-paradigm, multi-threaded, and multi-core computing devices can provide several orders-of-magnitude performance improvement over traditional single-core microprocessors. These devices include mainstream homogenous multi-core processors from Intel and AMD, the STI Cell Broadband Engine (BE) 44, Graphical

Processing Units (GPUs) [30], and the Cray XMT [23] (Eldorado[39]) systems. These architectures have been developed for a variety of purposes, including gaming, multimedia, and signal processing.

For productivity, our initial evaluations used source lines of code (SLOC) for the serial version against the target implementations using a tool called sloccount[2]; for performance, we measure algorithm time-to-solution. We were unable to use existing tools to measure many of the other metrics used in the HPCS productivity effort because they were incompatible with the programming environment or architecture we were evaluating. We share the concerns of the entire HPCS community that the SLOC metric does not fully capture the level of effort involved in porting and optimizing an algorithm on a new system; however, it does provide a quantitative metric to compare and contrast different implementations in a high-level language – C – across the diverse platforms in our study.

3.1 Architecture Overviews

3.1.1 Homogeneous Multi-core Processors

Our target commodity multi-core platforms are the dual-core and quad-core platforms from Intel. *Clovertown* is a quad-core Xeon 5300 series processor, which consists of two dual-core 64-bit Xeon processor dies, packaged together in a multi-chip module (MCM). Although a Level 2 cache is shared within each Xeon dual-core die, there is no cache shared between both dies of an MCM. Like the Intel Xeon 5100 series dual-core processor known as *Woodcrest*, the Clovertown processor uses Intel's next-generation Core 2 microarchitecture [50]. The clock frequencies of our target platforms are 2.4 GHz for the Clovertown processor and 2.66 GHz for the Woodcrest processor.

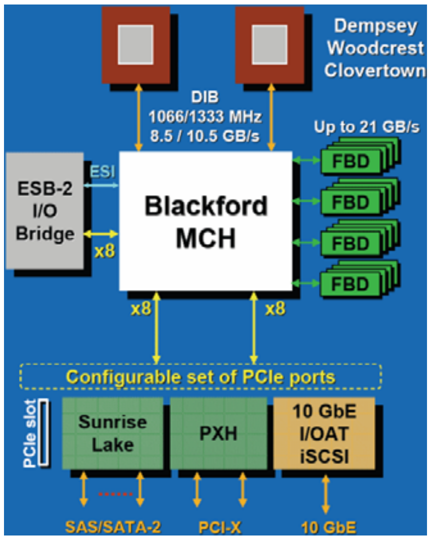


Figure 3.1. The Bensley Platform (Courtesy of Intel)

Both Clovertown and Woodcrest systems are based on Intel's Bensley platform (shown in Figure 3.1) for Xeon processors, which is expected to have sufficient capacity to handle the overheads of additional cores. The Blackford Memory Controller Hub (MCH) has dual, independent front-side busses, one for each CPU socket, and those FSBs run at 1333MHz when coupled with the fastest Xeons, rated at approximately 10.5GB/s per socket. Also, four memory channels of the Blackford MCH can host Fully Buffered

[2] http://www.dwheeler.com/sloccount/

DIMMs (FB-DIMMs) at clock speeds up to 667MHz. The twin chips inside of the Clovertown processor have no real provision for communicating directly with one another. Instead, the two chips share the front-side bus (FSB) with the Intel Blackford MCH, or north bridge.

The microarchitecture for both the Woodcrest and Clovertown processors supports Intel's Streaming SIMD Extension (SSE) instructions that operate on data values packed into 128-bit registers (e.g., four 32-bit values or two 64-bit values) in parallel. The microarchitecture used in the Clovertown processor can execute these SIMD instructions at a rate of one per cycle, whereas the previous-generation microarchitecture was limited to half that rate. The microarchitecture includes both a floating-point multiply unit and a floating-point add unit. Using SIMD instructions, each of these units can operate on two packed double-precision values in each cycle. Thus, each Clovertown core is capable of producing four double-precision floating-point results per clock cycle, for a theoretical maximum rate of sixteen double-precision floating-point results per clock cycle per socket in a Clovertown-based system.

3.1.2 Cell Broadband Engine

The Cell Broadband Engine processor is a heterogeneous multicore processor, with one 64-bit Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs), as shown in Figure 1.2. The PPE is a dual-threaded Power Architecture core containing extensions for SIMD instructions (VMX) **Error! Reference source not found.**. The SPEs are less traditional, in that they are lightweight processors with a simple, heavily SIMD-focused instruction set, with a small (256KB) fixed-latency local store (LS), a dual-issue pipeline, no branch prediction, and a uniform 128-bit, 128-entry register file **Error! Reference source not found.**. The SPEs operate independently from

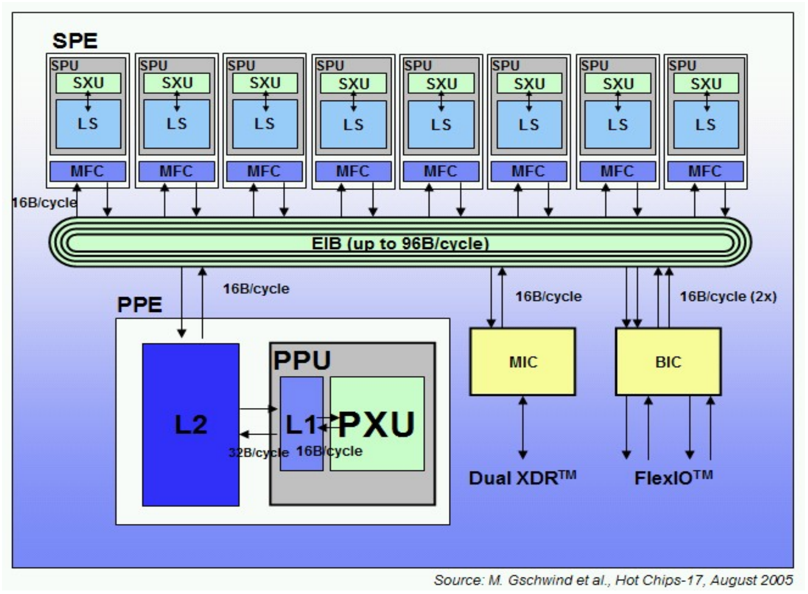


Figure 1.2. Design components of the Cell BE [http://www.research.ibm.com/cell/heterogeneousCMP.html].

the PPE and from each other, have an extremely high bandwidth DMA engine for transferring data between main memory and other SPE local stores, and are heavily optimized for single-precision vector arithmetic. Regrettably, these SPEs are not optimized for double-precision floating point calculations, limiting Cell's applicability for a large number of scientific applications. The Cell processor architecture enables great flexibility with respect to programming models [27].

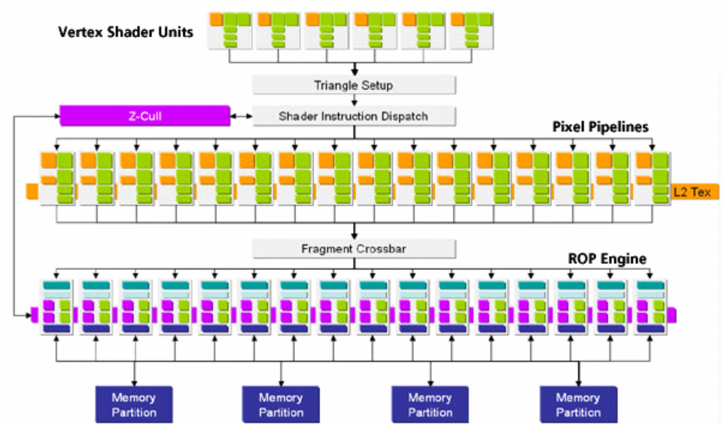


Figure 3.3. The NVIDIA GeForce 6800 GPU

3.1.3 Graphics Processing Units

The origin of Graphics Processing Units, or GPUs, is in accelerating the real-time graphics rendering pipeline. As developers demanded more power and programmability from graphics cards, these cards became appealing for general-purpose computation, especially as mass markets began to force even high-end GPUs into low price points [25]. The high number of FLOPS in GPUs comes from the parallelism in the architecture. Figure 3.3 shows an earlier-generation high-end part from NVIDIA, with 16 parallel pixel pipelines. It is these programmable pipelines that form the basis of general-purpose computation on GPUs. Moreover, in next-generation devices, the parallelism increased. The subsequent generation from NVIDIA contained up to 24 pipelines. Typical high-end cards today have 512MB of local memory or more, and support from 8-bit integer to 32-bit floating point data types, with 1, 2, or 4 component SIMD operations.

There are several ways to program the parallel pipelines of a GPU. The most direct way is to use a GPU-oriented assembler or a compiled C-like language with graphics related intrinsics, such as Cg from NVIDIA [30]. Accordingly, as GPUs are coprocessors, they require interaction from the CPU to handle high-level tasks such as moving data to and from the card, and setting up these "shader programs" to execute on the pixel pipelines.

Inherently, GPUs are stream processors, as a shader program cannot read and write to the same memory location. Thus, arrays must be designated as either input or output, but not both. There are technical limitations on the number of input and output arrays addressable in any shader program. Together, these restrictions form a set of design challenges for accelerating a variety of algorithms using GPUs.

3.1.4 Cray MTA-2

Cray's Multi-Threaded Architecture (MTA) uses a high degree of multi-threading instead of data caches to address the gap between the rate at which modern processors can execute instructions and the rate at which data can be transferred between the processor and main memory. An MTA-2 system consists of a collection of processor modules and a collection of memory modules, connected by an interconnection network. The MTA processors support a high degree of multi-threading when compared to current commercial off-the-shelf processors (as shown in Figure 3.4). These processors tolerate memory access latency by supporting many concurrent streams of execution (128 in the MTA-2 system processors). A processor can switch between each of these streams on each clock cycle. To enable such rapid switching between streams, each processor maintains a complete thread execution context in hardware for each of its 128 streams. Unlike conventional designs, an MTA-2 processor module contains no local memory; it does include an instruction stream shared between all of its hardware streams.

The Cray MTA-2 platform is significantly different from contemporary, cache-based microprocessor architectures. Its differences are reflected in the MTA-2 programming model and, consequently, its software development environment [24]. The key to obtaining high performance on the MTA-2 is to keep its processors saturated, so that each processor always has a thread whose next instruction can be executed. If the collection of threads presented to a processor is not large enough to ensure this condition, then the processor will be under-utilized.

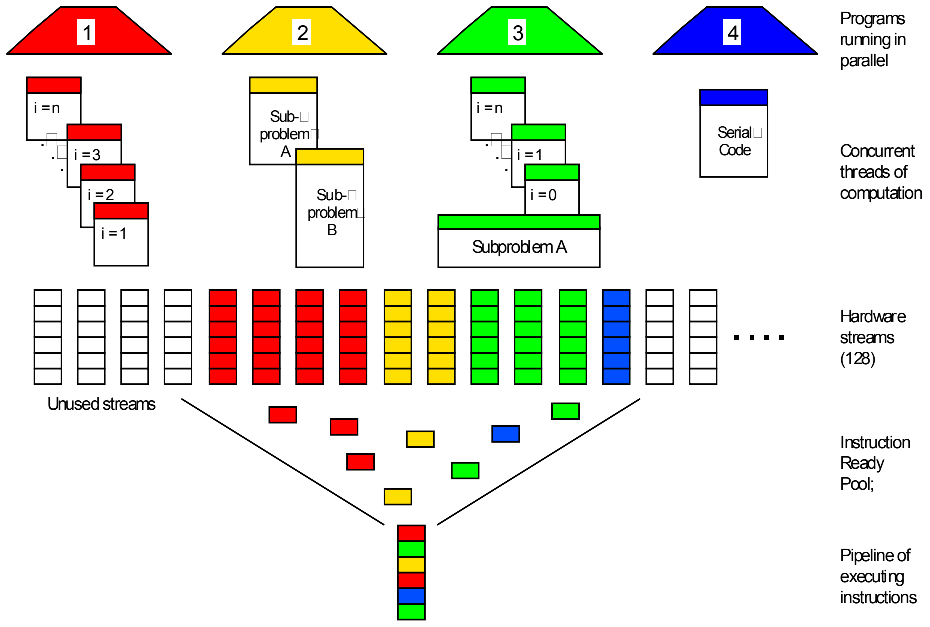


Figure 3.4. Block diagram of the MTA-2 system

The MTA-2 is no longer an active product in the Cray product line. However, Cray has announced an extreme multi-threaded system – the Cray XMT system. Although the XMT system uses multithreaded processors similar to those of the MTA-2, there are several important differences in the memory and network architecture. The XMT will not have the MTA-2's nearly uniform memory access latency, so data placement and access locality will be important considerations when programming these systems.

3.2 Target Workloads

We target two algorithms for our initial evaluations: an imaging application and a floating-point-intensive scientific algorithm.

3.2.1 Hyperspectral Imaging (HSI)

A covariance matrix is created in a number of imaging applications, such as hyperspectral imaging (HSI). HSI, or image spectroscopy, can be described as the capture of imagery either with a large number of wavelengths or across a large number of pixels. Whereas black and white images are captured at one wavelength, and color images at three (red, green, and blue), hyperspectral images are captured in hundreds of wavelengths simultaneously. If a HSI data cube is N by M pixels, with L wavelengths, the covariance matrix is an L×L matrix, where the entry $Cov_{a,b}$ at row a and column b in the covariance matrix can be represented as:

$$Cov_{a,b} = \sum_{i=1}^{N}\sum_{j=1}^{M} input_{i,j,a} \times input_{i,j,b} .$$

3.2.2 Molecular Dynamics

The biological processes within a cell occur at multiple lengths and time scales. The processing requirements for bio-molecular simulations, particularly at large time scales, far exceed the available computing capabilities of the most powerful computing platforms today. Molecular dynamics (MD) is a computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating the equations of motion [45]. In the Newtonian interpretation of dynamics, the translational motion of a molecule is caused by force exerted by some external agent. The motion and the applied force are explicitly related through Newton's second law: $F_i = m_i a_i$. m_i is the atom's mass, $a_i = \frac{d^2 r_i}{dt^2}$ is its acceleration, and F_i is the force acting upon it due to the interactions with other atoms. MD techniques are extensively used in many areas of scientific simulations, including biology, chemistry, and materials.
MD simulations are computationally very expensive. Typically, the computational cost is proportional to N^2, where N is the number of atoms in the system. In order to reduce the computational cost, a number of algorithm-oriented techniques such as a cutoff limit are used. It is assumed that atoms within a cutoff limit contribute to the force and energy calculations on an atom. As a result, the MD simulations do not exhibit a cache-friendly memory access pattern.

Our MD kernel contains two important parts of an MD calculation: force evaluation and integration. Calculation of forces between bonded atoms is straightforward and less computationally intensive, as there are only very small numbers of bonded interactions as compared to the non-bonded interactions. The effects of non-bonded interactions are modeled by a 6-12 Lennard-Jones (LJ) potential model: $V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$.

The Verlet algorithm uses positions and acceleration at time *t* and positions from time $t + \delta t$ to calculate new positions at time $t + \delta t$. The pseudo code for our implementation is given in Figure 3.5. Steps are repeated for *n* simulation time steps.

```
1. advance velocities
2. calculate forces on each of the N atoms
   compute distance with all other N-1 atoms
       if(distance within cutoff limits)
         compute forces
3. move atoms based on their position,
   velocities & forces
4. update positions
5. calculate new kinetic and total energies
```

Figure 3.5. MD kernel implemented on MTA-2

The most time-consuming part of the calculation is step 2, in which an atom's neighbors are determined using the cutoff distance, and subsequently the calculations are performed (N^2 complexity). We attempt to optimize this calculation on the target platforms, and we compare the performance to the reference single-core system. We implement single-precision versions of the calculations on the Cell BE and the GPU accelerated system, while Intel OpenMP and MTA-2 implementation are in double-precision.

3.3 Evaluation

3.3.1 Homogeneous Multi-core Systems

Optimization Strategies

Since there are shared memory resources in Intel dual- and quad-core processors, we consider OpenMP parallelism in this study. Note that an additional level of parallelism is also available within individual Xeon cores in the form of SSE instructions. The Intel compilers are capable of identifying this parallelism with optimized flags, including such as `–fast -msse3 -parallel (-fast= -O3, -ipo, –static).`

Molecular Dynamics

We inspected compiler reports and identified that a number of small loops in the initialization steps and subsequent calculations are automatically vectorized by the compiler. The complex data and control dependencies in the main phases of the calculation prevented the generation of optimized SSE instruction by the compiler. The next step was to introduce OpenMP parallelism in the main calculation phases. Figure 3.6 shows results of an experiment with 8,192 atoms. Overall, the performance of the Woodcrest system is higher than that of the Clovertown system, which could be attributed to the higher clock of the Woodcrest system, the shared L2 cache between the two cores, and shared FSB between the two sockets in the Clovertown processor. We observe that the speedup increases with workload size (number of atoms). As a result, the runtime performance of 8 threads on Clovertown exceeds the performance of 4 threads on the Woodcrest system.

HSI Covariance Matrix

The OpenMP optimization applied for the covariance matrix calculations is similar to the MD optimization. The main loop is composed of largely data-independent calculations. We modified the innermost loop where a reduction operation is performed and then applied OpenMP parallel for construct. Figure 3.7 shows results on a covariance matrix creation for a 256^3 data cube. Results are qualitatively similar to those from the molecular dynamics kernel: the Woodcrest system outperforms the Clovertown on the same number of threads, likely due to a higher clock speed and other architectural differences. In this case, the 8-thread Clovertown result is a great improvement over the 4-thread Clovertown runtime, although it only exceeds the performance of the 4-thread Woodcrest implementation by a very small margin.

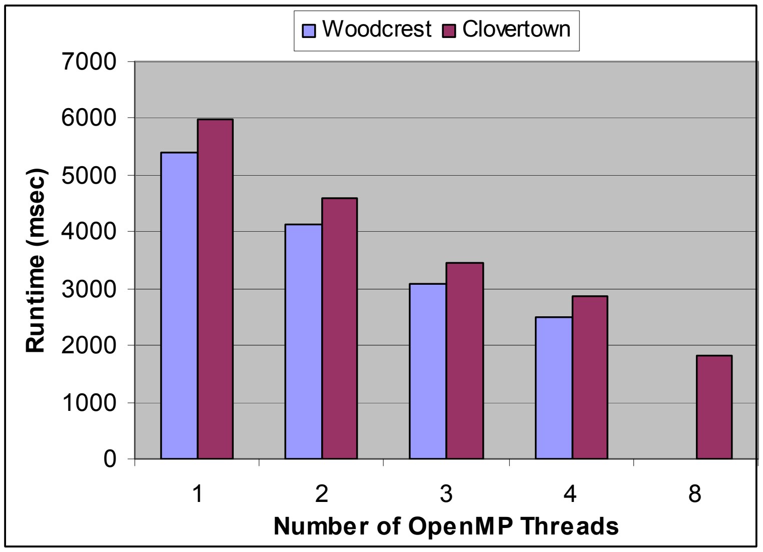


Figure 3.6. MD experiments with 8,192 atoms

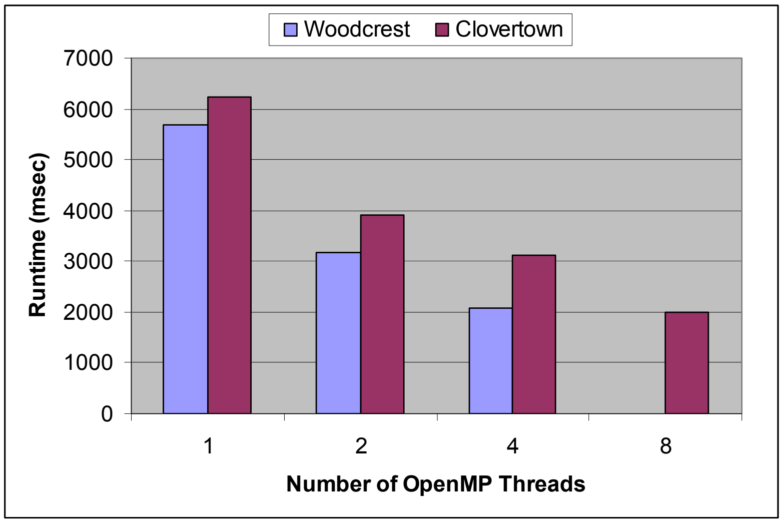


Figure 3.7. Experiments with 256^3 HSI covariance matrix

3.3.2 Cell Broadband Engine

Molecular Dynamics

Our programming model for the Cell processor involves finding time-consuming functions that map well to the SPE cores, and instead of calculating these functions on the PPE, we launch “threads” on the SPEs to read the necessary information into their local stores, perform the calculations, and write the results back into main memory for the next calculation steps on the PPE. Because of its high percentage of the total runtime, the MD acceleration function alone was offloaded to SPEs.

The MD calculation deals with three-dimensional positions, velocities, and forces, so the most natural way to make use of the 4-component SIMD operations on the SPE is to use the first three components of the inherent SIMD data types for the x, y, and z components of each of these arrays. The communication between the PPE and SPEs is not limited to large asynchronous DMA transfers; there are other channels (“mailboxes”) that can be used for blocking sends or receives of information on the order of bytes. As we are offloading only a single function, we can launch the SPE threads only on the first time step, and signal them using mailboxes when there is more data to process. Hence, the thread launch overhead is amortized across all time steps.
Runtime results are listed in Table 3.1 for a 4096-atom experiment that runs for 10 simulation time steps. Due the extensive use of SIMD intrinsics on the SPE, even a single SPE outperforms the PPE on the Cell processor by a significant margin. Further, with an efficient parallelization using all 8 SPEs, the total runtime is

Number of Atoms	**4096**
Cell, PPE only	4.664 sec
Cell, 1 SPE	2.958 sec
Cell, 8 SPEs	0.448 sec

Table 3.1. Performance comparison on the Cell processor.

approximately 10 times faster than the PPE alone.

HSI Covariance Matrix

The covariance matrix creation routine transfers more much data through the SPEs for the amount of computation performed than the MD application does. Specifically, the data set and tiling sizes used resulted in a total of 16 chunks to process, and this implementation must still stream the data set through each SPE to create each of the output chunks. Therefore, the time spent during data transfer has the potential for a noticeable impact on total performance. In this example, optimization of the thread launching, DMA overlapping, and synchronization resulted in considerable improvement of the routine.

Table 3.2 shows the performance improvement on the Cell processor. The PPE shows its disadvantage as a computational processor by running 18x slower than a single SPE. Additionally, parallelization across SPEs was very effective, providing a 7.5x speedup on 8 SPEs.

Covariance Matrix	**256x256x256**
Cell, PPE only	88.290 sec
Cell, 1 SPE	5.002 sec
Cell, 8 SPEs	0.662 sec

Table 3.2. Performance comparison on the Cell processor.

3.3.3 GPU

Molecular Dynamics

As with the Cell, implementation for the GPU focused on the part of the algorithm that calculates new accelerations from only the locations of the atoms and several constants. For our streaming processor, then, the obvious choice is to have one input array comprising the positions, and one output array comprising the new accelerations. The constants were compiled into the shader program source using the provided JIT compiler at program initialization.

We set up the GPU to execute our shader program exactly once for each location in the output array. That is, each shader program calculates the acceleration for one atom by checking for interaction with all other atoms and accumulating forces into a single acceleration value for the target atom. Instruction length limits prevent us from searching more than a few thousand input atoms in a single pass, and so with 4096 atoms or more, the algorithm switches to use multiple passes through the input array. After the GPU is finished, the resulting accelerations are read back into main memory, where the host CPU proceeds with the current time step. At the next time step, the updated positions are re-sent to the GPU and new accelerations computed again.

Figure 3.8 shows performance using an NVIDIA GeForce 7900GTX GPU. This figure includes results from the GPU's host CPU (a 2GHz Opteron) as a reference for scaling comparisons. The readily apparent change in slope for the GPU below 1,024 atoms shows the point at which the overheads associated with offloading this acceleration computation to the GPU become a more significant fraction of the total runtime than the $O(N^2)$ calculation itself. These constant and $O(N)$ costs for each time step include sending the position array and reading the acceleration array across the PCIe bus at every time step, and these results show that there is a lower bound on problem size where a GPU will not be faster than a CPU. However, the massive parallelism of the GPU helps it maintain a consistent speedup above 1000 atoms.

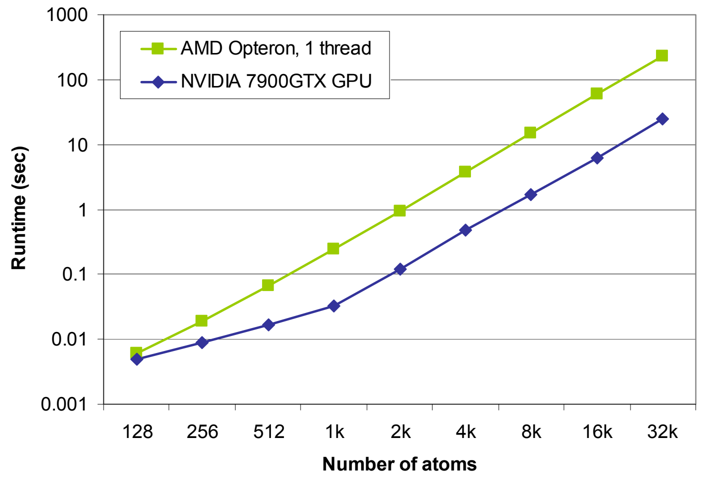


Figure 3.8 Performance scaling results on GPU with CPU scaling for comparison.

HSI Covariance Matrix

The GPU architecture is generally well suited to image operations, and to some degree this extends to hyperspectral image data. However, as the SIMD nature of the pipelines in a GPU is well oriented toward 4-component images, this is not a direct match with an image with many more than four components. However, the regular nature of the data does have an impact, and the SIMD nature of the GPU can be exploited to take advantage of the operations in the covariance matrix creation routine. Figure 3.9 shows the runtime of the GPU on the 256^3 covariance matrix creation benchmark under several implementations. The implementation exploiting none of the SIMD instructions on the GPU naturally performs the worst. The fully SIMDized implementation is a drastic improvement, running several times faster than the unoptimized implementation.

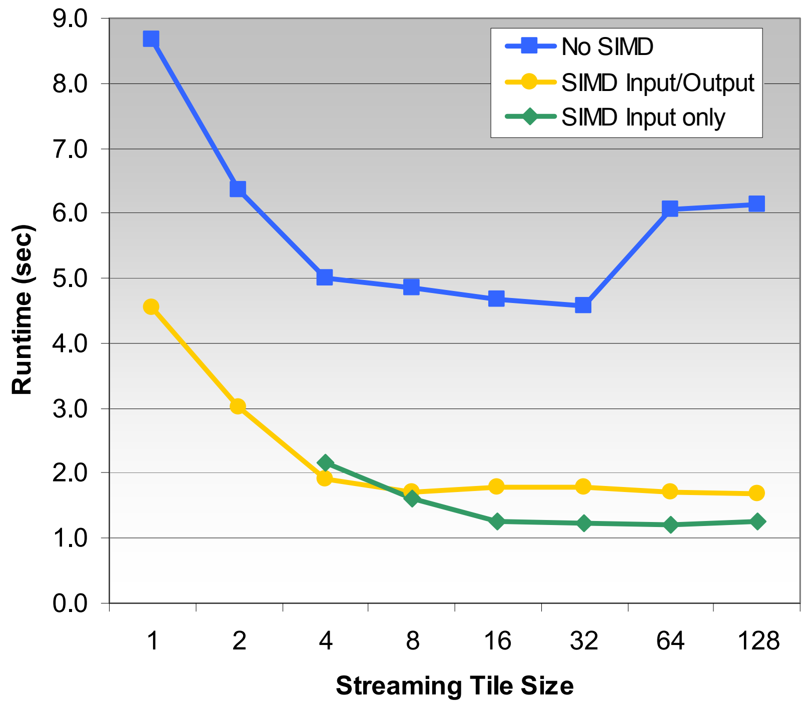


Figure 3.9. Performance of the GPU on the 256^3 covariance matrix creation under a variety of SIMDization optimizations and streaming tile sizes.

Figure 3.9 also shows the effect of tile size on total runtime. With larger tile sizes, fewer passes need to be made over the same data. However, the larger the tile size, the longer the stream program which needs to run, and this can adversely impact memory access patterns. The competing effects lead to a moderate value for the ideal streaming tile size.

3.3.4 MTA-II

Molecular Dynamics

The MTA-2 architecture provides an optimal mapping to the MD algorithm because of its uniform memory latency architecture. In other words, there is no penalty for accessing atoms outside the cutoff limit or the cache boundaries, in an irregular fashion, as is the case in the microprocessor-based systems. In order to parallelize calculations in step 2, we moved the reduction operation inside the loop body. Figure 3.10 shows the performance difference before and after adding several pragmas to the code to remove phantom dependencies from this loop.

In

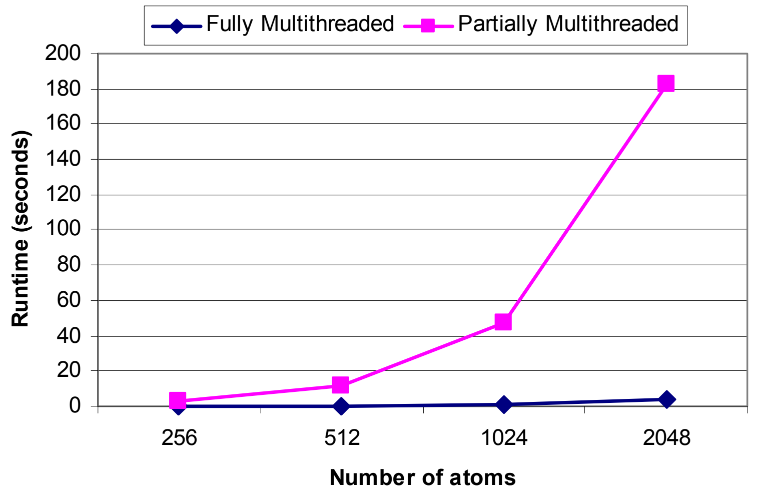


Figure 3.10. Performance comparison of fully vs. partially multithreaded versions of the MD kernel for the MTA-2.

order to explore the impact of the uniform memory hierarchy of the MTA-2 architecture, we compare its performance with the OpenMP multi-threaded implementation on the Intel quad-core Clovertown processor. Figure 3.11 shows runtime in milliseconds on the two systems when increasing the number of OpenMP threads (number of Clovertown cores) and number of MTA-2 processors for three workload sizes. Although the performance of the Clovertown processor (released in November 2006) is significantly higher than the MTA-2 system (released in early 2002) for the same number of cores/MTA-2 processors, the MD workload scales almost linearly on the MTA-2 processor. Note that the clock frequency of MTA-2 is 200 MHz while the Clovertown operates at 2.4 GHz clock frequency. Speedup (runtime on a single execution unit/runtime on n execution units) shown in Figure 3.12 confirms that the fine-grain multi-threading on the MTA-2 system provides relatively high scaling as compared to the OpenMP implementation.

HSI Covariance Matrix

Like the MD application, the covariance matrix calculation was only partially optimized by the MTA-2 compiler. To enable full multithreading, we moved the reduction operation outside the inner loop and introduced an array element. A full multithreaded version of the application is then produced by the MTA-2 compiler. Figure 3.13 and Figure 3.14 compare performance and relative speedup of the OpenMP multithreaded implementation and MTA-2 multithreaded implementation, respectively. We observe that although the time-to-solution is much faster on the recent quad-core Clovertown system, the MTA-2 system provides high parallel efficiency on up to 16 threads.

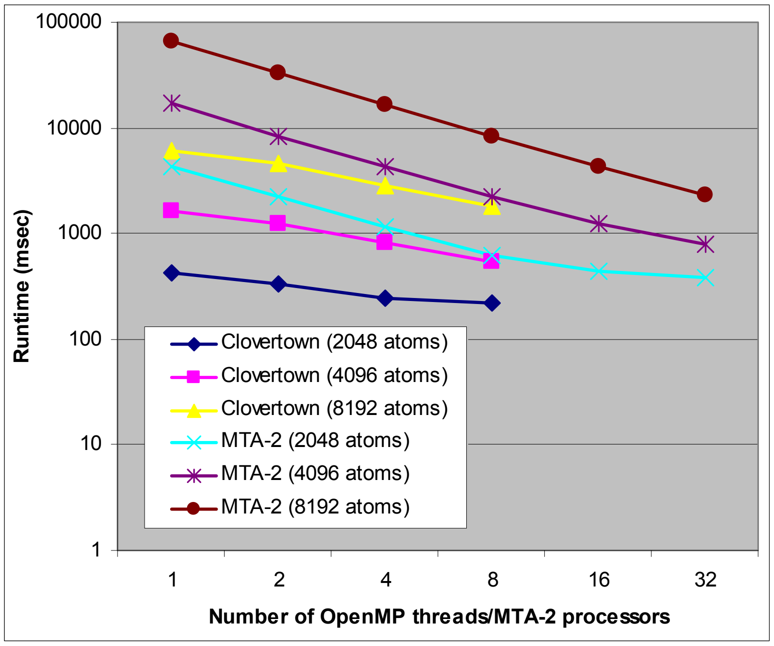


Figure 3.11. Runtime in milli-seconds with multi-threaded implementation

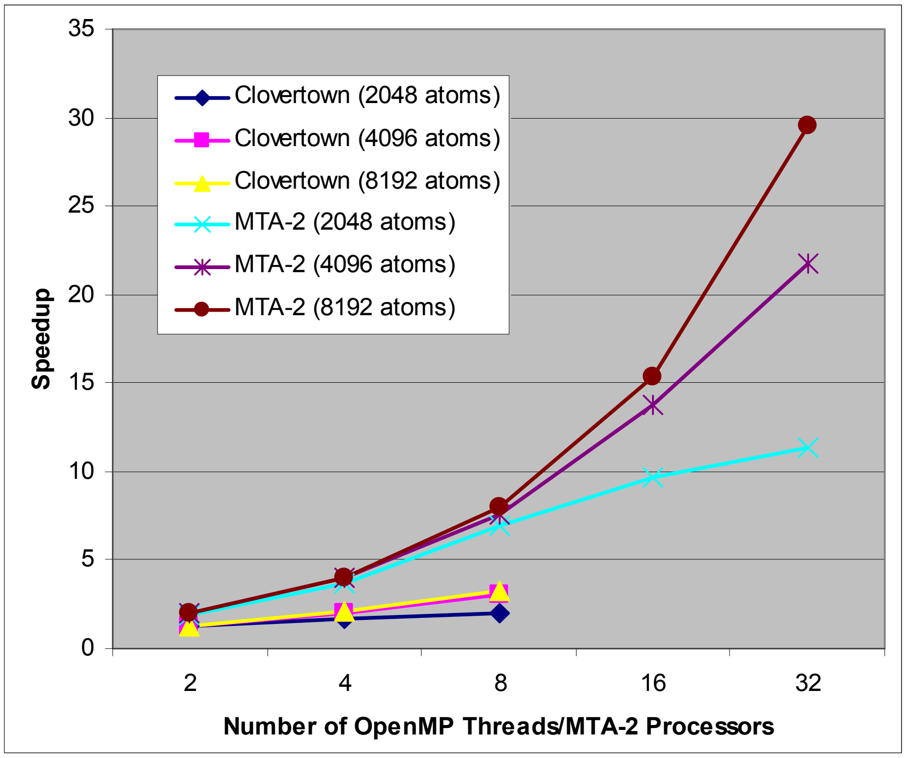


Figure 3.12. Parallel efficiency on Clovertown cores (2 sockets) and MTA-2 processors

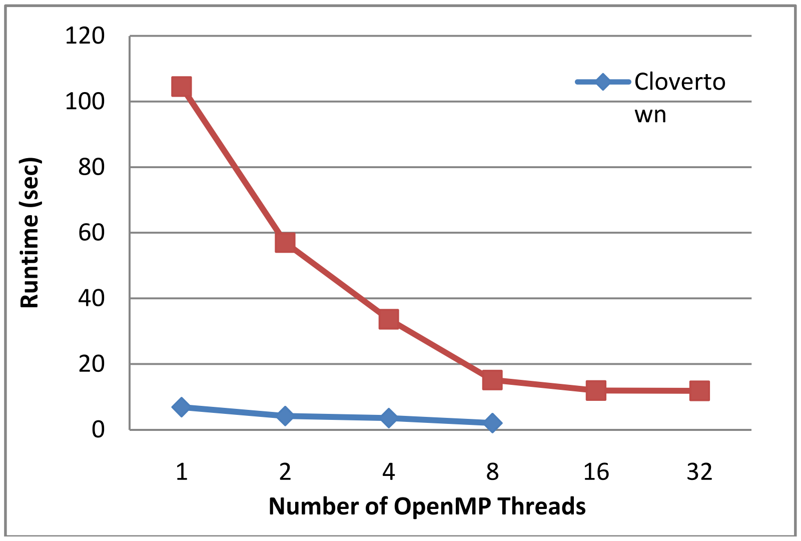


Figure 3.13. Runtime in milli-seconds with multi-threaded implementation (HSI)

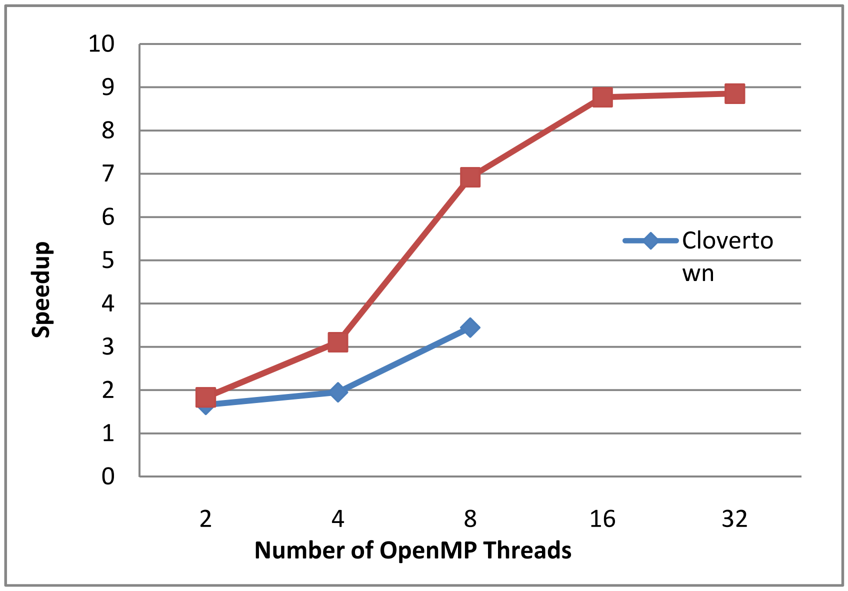


Figure 3.14. Parallel efficiency on Clovertown cores and MTA-2 processors (HSI)

3.4 Productivity

The preceding section reviewed a crucial aspect of the HPCS program's productivity goal – sustained performance on real-world codes – and presented performance and scaling results in terms of runtimes and parallel speedups. But another critical contributor to HPC productivity occurs before the codes are ever run. In this section, we discuss and compare relative performance improvement, and performance-to-productivity ratios, by taking into account the level of effort involved in optimizing the same molecular dynamics (MD) algorithm on the target architectures. Since it is not trivial to quantify the code development effort, which depends on a number of factors including the level of experience of the code developer, the degree of familiarity with the programming model, and the attributes of the target system, we measure a quantifiable value called Source Lines of Code (SLOC). We recognize that SLOC does not encapsulate and fully represent the code optimization effort, but it has been extensively used and reported in a large number of productivity studies on parallel and high performance computing systems. We further divide the SLOC into *effective* SLOC and *boilerplate* SLOC. The distinction is intended to quantify the learning curve associated with the unique architectures investigated here, as there is some amount of boilerplate code that one must write to get any application working but which will typically be re-used on further applications. For example, code for SPE thread launches and DMA transfers on the Cell is highly reusable, and on the GPU this might include initializing OpenGL and other graphics primitives. So the ‘total’ SLOC is close to what one might expect when presented with the architecture for the first time, and the ‘effective’ SLOC (with boilerplate code discounted) approximates what someone experienced with the platform can expect.

We measure the relative performance of the optimized, SSE-enabled microprocessor (single-core Intel Woodcrest) version and the optimized implementation as $Performance_{ratio} = \frac{Runtime_{Woodcrest-core}}{Runtime_{emerging-architecure}}$, and we measure the productivity by comparing the SLOC ratio of the test suite as $SLOC_{ratio} = \frac{SLOC_{optimized-implementation}}{SLOC_{serial-implementation}}$. Although no code modification is performed in the serial version, we extensively studied the impact of various compile-time and runtime optimization flags offered by the Intel C compiler (version 9.1). These optimizations included operations such as Inter Procedural Optimization (IPO), auto-parallelization, and SSE-enabled code generation/vectorization. Hence, our reference runtime results are optimal for the single-core Woodcrest platform. In order to quantify the tradeoffs between time spent tuning a code versus the benefit seen via shorter runtimes, we introduce the concept of “Relative Productivity,” in the form of the relative development time productivity (RDTP) metric, defined as speedup divided by relative effort, i.e., the ratio of the first two metrics [42].

Some might observe that additional optimization can often result in fewer lines of code, and thus suggest it is misleading to use SLOC as a productivity metric. While potentially true, that effect is greatly mitigated in this study. First, these devices ostensibly require adding code, not subtracting it, simply to get these devices to function as accelerators. Hence, an increase in the lines of code, when compared to the single-threaded CPU version, almost certainly requires an increase in the amount of effort. This is in dramatic contrast to other kinds of optimizations, such as within homogeneous CPU code and loop reordering, which can entail considerable effort with no increase in SLOC. Secondly, the fact that the added code was subject to these kinds of statement-level-optimizations, which have a nonlinear impact on SLOC, does not invalidate the comparison, because the original single-threaded code was subject to these same transformations. In other words, because we are comparing optimized code to optimized code, SLOC remains a useful metric for the optimization effort.

	SLOC Ratio (Total)	SLOC Ratio (Effective)	Performance Ratio	Relative Productivity
OpenMP (Woodcrest, 4 cores)	1.07	1.059	1.713	1.618
OpenMP (Clovertown, 8 cores)	1.07	1.059	2.706	2.556
Cell (8 SPEs)	2.27	1.890	2.531	1.339
GPU (NVIDIA 7900GTX)	3.63	2.020	2.502	1.239
MTA-2 (32 processors)	1.054	1.054	1.852	1.757

Table 3.3. Performance and productivity of an MD calculation on the emerging architectures relative to the single-core, SSE-enabled Woodcrest implementation. Note that the performance of OpenMP and MTA implementations is gathered on multiple cores/processors.

	SLOC Ratio (Total)	SLOC Ratio (Effective)	Performance Ratio	Relative Productivity
penMP (Woodcrest, 4 cores)	1.070	1.047	2.746	2.624
penMP (Clovertown, 8 ores)	1.070	1.047	2.859	2.732
ell (8 SPEs)	5.605	3.442	8.586	2.495
PU (NVIDIA 7900GTX)	11.302	1.977	4.705	2.380
TA-2 (32 processors)	1.093	1.093	0.481	0.440

Table 3.4. Performance and productivity of a covariance matrix creation on the emerging architectures relative to the single-core, SSE-enabled Woodcrest implementation. Note that the performance of OpenMP and MTA implementations is gathered on multiple cores/processors.

In Table 3, we list the ratio of source lines of code (SLOC), both in terms of effective SLOC and total SLOC, and the relative performance increase on our target platforms. The "Performance Ratio" column in Table 3 is the speedup relative to the reference single-threaded implementation running on a 2.67GHz Woodcrest Xeon compiled with the Intel 9.1 compiler, using the flags that achieved the best performance, including SSE3 instructions and inter-procedural optimization. The final column, "Relative Productivity," is the relative development time productivity (RDTP) metric defined above, where a higher number indicates more benefit for less code development effort. The RDTP metric presented in Table 3 is calculated using effective SLOC, as the boilerplate code required no additional development effort.

First, we calculate RDTP for the OpenMP implementation on the Intel multi-core platforms. Since there is very little boilerplate code for the OpenMP implementation, the SLOC ratio compared to the serial implementation is not high. Performance is measured for 32K-atoms runs on the reference Woodcrest core against the 4 OpenMP threads and 8 OpenMP threads runs on Woodcrest and Clovertown, respectively. The RDTP for the Woodcrest is well over one and for the 8 cores, the ratio is more than 2. We therefore conclude that the OpenMP implementation does not have a negative performance-to-productivity ratio, as this implementation can utilize the target system resources effectively.

For the Cell processor, our final performance numbers were obtained using the latest XLC compiler. Our implementation limited us to a comparison at 4K atoms, but effective parallelization and use of SIMD instructions nevertheless resulted in good performance, even at this smaller problem size. However, manually handling the decomposition and the various aspects of SPE coding did result in a noticeable increase in SLOC, and as such the RDTP for the Cell processor was about 1.3.

The GPU had comparable performance to the Cell – though the parallelization is handled at a finer granularity, the SIMD instructions were utilized in a similar fashion. Though we rely on the GPU to handle the distribution of the parallel work among the shader units, collecting the results and setting up the graphics primitives in a way the GPU can process still takes some coding effort, even after most boilerplate routines are discounted. As such, the effective SLOC ratio is the highest of all platforms, and the RDTP, though still greater than one, is the lowest among the platforms.

Finally, we compare the fine-grain multi-threaded implementation on the MTA-2 system. Due to a highly optimizing compiler, very few code modifications are required to optimize the time-critical loop explicitly; the remaining loops were automatically optimized by the compiler, resulting in very low code development overheads. We compare performance of a 32K-atoms run on 32 MTA-2 processors. Note that the uniform memory hierarchy for 32 processors provides for good scaling and somewhat compensates for the difference in the clock rates of the two systems. Like the OpenMP implementation, the RDTP for the fine-grain multithreading on MTA-2 is greater than 1; in fact, it is close to the dual-core (4 cores in total) Woodcrest system's RDTP.

Note that we have not utilized multiple sockets and processors for all our target devices. However, it is possible to utilize more than one Cell processor in parallel, as blade systems have two sockets on the board, and similarly one can place two GPUs in a node with more than one PCI-Express slot. So, with additional effort one could include an additional level of parallelism for these platforms. As such, we introduce Table to show how performance and productivity compare when limited to that of a single "socket" or "processor," with as much inherent parallelism (cores, shaders, SPEs) as this implies. For Woodcrest, this means we are limited to one socket, and thus two cores; for Clovertown, four cores; and for MTA-2, 128 streams. In these comparisons, only the Clovertown, Cell, and GPU sustained RDTP metrics greater than one. The Cell and the GPU implementation, on the other hand, provide over 2x speedup over the reference optimized serial implementation.

	Performance Ratio	**Relative Productivity**
OpenMP (Woodcrest, 2 cores)	1.031	0.973
OpenMP (Clovertown, 4 cores)	1.407	1.329
Cell (8 SPEs)	2.531	1.339
GPU (NVIDIA 7900GTX)	2.367	1.172
MTA-2 (1 processor, 128 streams)	0.0669	0.063

Table 3.5. Performance and productivity of a 4K-atom MD calculation (single socket/processor comparisons).

	Performance Ratio	Relative Productivity
OpenMP (Woodcrest, 2 cores)	1.794	1.714
OpenMP (Clovertown, 4 cores)	1.823	1.742
Cell (8 SPEs)	8.586	2.495
GPU (NVIDIA 7900GTX)	4.705	2.380
MTA-2 (1 processor, 128 streams)	0.054	0.050

Table 3.6. Performance and productivity for a 256^3 HSI data cube (single socket/processor comparisons).

We would like to emphasize that the results presented in Table 3 and Table should not be considered the absolute performance measures of the targeted devices. First, the level of maturity in the software stack for scientific code development is not consistent across all platforms. Second, some target devices presented in the paper do not represent the state-of-the-art in the field, while others were released very recently. For instance, the Clovertown and Woodcrest systems are the most recent releases among all the platforms. In contrast, the 2.4GHz Cell processor used here is over one year old, the 7900GTX has already been supplanted by its successor GPU from NVIDIA, and the MTA-2 system was released in early 2002 and is not an active product from Cray.

4. The DARPA HPCS Language Project

. Another important goal of the HPCS project has been to improve the productivity of software designers and implementers by inventing new languages that facilitate the creation of parallel, scalable software. Each of the three Phase II vendors proposed a language—Chapel from Cray, X10 from IBM, Fortress from Sun—for this purpose. (Sun was not funded in Phase III, so Fortress is no longer being supported by DARPA. Nonetheless, it remains a significant contribution to the HPCS goals, and we include it here.) Before we consider these “HPCS languages” themselves, we provide the context in which this development has taken place. Specifically, we discuss current practice, compare some early production languages with the HPCS languages, and comment on previous efforts to introduce new programming languages for improved productivity and parallelism.

4.1 Architectural Developments

Language development for productivity is taking place at a time when the architecture of large-scale machines is still an area of active change. Innovative network interfaces and multiprocessor nodes are challenging the ability of current programming model implementations to exploit the best performance the hardware can provide, and multicore chips are adding another level to the processing hierarchy. The HPCS hardware efforts are at the leading edge of these innovations. By combining hardware and languages in one program, DARPA is allowing language designs that may take advantage of unique features of one system, although this design freedom is tempered by the desire for language ubiquity. The new HPCS programming models and languages will be expected to exploit the full power of the new architectures, while still providing reasonable performance on more conventional systems.

Current Practice

Most parallel programs for large-scale parallel machines are currently written in a conventional sequential language (Fortran-77, Fortran-90, C, or C++) with calls to the MPI message-passing library. The MPI standard [63, 64] defines bindings for these languages. Bindings for other languages (particularly Java) have been developed and are in occasional use but are not part of the MPI standard. MPI is a realization of the message-passing model, in which processes with completely separate address spaces communicate with explicit calls to `send` and `receive` functions. MPI-2 extended this model in several ways (parallel I/O, remote memory access, and dynamic process management), but the bulk of MPI programming utilizes only the MPI-1 functions. The use of the MPI-2 extensions is more limited, but usage is increasing, especially for parallel I/O.

The PGAS Languages

In contrast with the message-passing model, the Partitioned Global Address Space (PGAS) languages provide each process direct access to a single globally addressable space. Each process has local memory and access to the shared memory.[3] This model is distinguishable from a symmetric shared-memory model in that shared memory is logically partitioned, so there is a notion of near and far memory explicit in each of the languages. This allows programmers to control the layout of shared arrays and of more complex pointer-based structures.

The PGAS model is realized in three existing languages, each presented as an extension to a familiar base language: UPC (Unified Parallel C) [58] for C; Co-Array Fortran (CAF) [4.13] for Fortran, and Titanium [68] for Java. The three PGAS languages make references to shared memory explicit in the type system, which means that a pointer or reference to shared memory has a type that is distinct from references to local memory. These mechanisms differ across the languages in subtle ways, but in all three cases the ability to statically separate local and global references has proven important in performance tuning. On machines lacking hardware support for global memory, a global pointer encodes a node identifier along with a memory address, and when the pointer is dereferenced, the runtime must deconstruct this pointer representation and test whether the node is the local one. This overhead is significant for local references, and is avoided in all three languages by having expressions that are statically known to be local. This allows the compiler to generate code that uses a simpler (address-only) representation and avoids the test on dereference.

[3]Because they access shared memory, some languages use the term "thread" rather than "process."

These three PGAS languages share with the strict message-passing model a number of processes fixed at job start time, with identifiers for each process. This results in a one-to-one mapping between processes and memory partitions and allows for very simple runtime support, since the runtime has only a fixed number of processes to manage and these typically correspond to the underlying hardware processors. The languages run on shared memory hardware, distributed memory clusters, and hybrid architectures.
Each of these languages is the focus of current compiler research and implementation activities, and a number of applications rely on them. All three languages continue to evolve based on application demand and implementation experience, a history that is useful in understanding requirements for the HPCS languages. UPC and Titanium have a set of collective communication operations that gang the processes together to perform reductions, broadcasts, and other global operations, and there is a proposal to add such support to CAF. UPC and Titanium do not allow collectives or barrier synchronization to be done on subsets of processes, but this feature is often requested by users. UPC has parallel I/O support modeled after MPI's, and Titanium has bulk I/O facilities as well as support to checkpoint data structures, based on Java's serialization interface. All three languages also have support for critical regions and there are experimental efforts to provide atomic operations.

The distributed array support in all three languages is fairly rigid, a reaction to the implementation challenges that plagued the High Performance Fortran (HPF) effort. In UPC distributed arrays may be blocked, but there is only a single blocking factor that must be a compile-time constant; in CAF the blocking factors appear in separate "co-dimensions;" and Titanium does not have built-in support for distributed arrays, but they are programmed in libraries and applications using global pointers and a built-in all-to-all operation for exchanging pointers. There is an ongoing tension in this area of language design, most visible in the active UPC community, between the generality of distributed array support and the desire to avoid significant runtime overhead.

The HPCS Languages

As part of Phase II of the DARPA HPCS Project, three vendors—Cray, IBM, and Sun—were commissioned to develop new languages that would optimize software development time as well as performance on each vendor's HPCS hardware, which was being developed at the same time. Each of the languages—Cray's Chapel [57], IBM's X10 [66], and Sun's Fortress [53]—provides a global view of data (similar to the PGAS languages), together with a more dynamic model of processes and sophisticated synchronization mechanisms.
The original intent of these languages was to exploit the advanced hardware architectures being developed by the three vendors, and in turn to be particularly well supported by these architectures. However, in order for these languages to be adopted by a broad sector of the community, they will also have to perform reasonably well on other parallel architectures, including the commodity clusters on which much parallel software development takes place. (The advanced architectures will also have to run "legacy" MPI programs well in order to facilitate the migration of existing applications.)

Until recently, the HPCS languages were being developed quite independently by the vendors; however, DARPA also funded a small, academically based effort to consider the languages together, in order to foster vendor cooperation and perhaps eventually develop a framework for convergence to a single high-productivity language [61]. (Recent activities on this front are described below).

Cautionary Experiences

The introduction of a new programming language for more than research purposes is a speculative activity. Much effort can be expended without creating a permanent impact. We mention two well-known cases.

In the late 1970s and early 1980s, an extensive community effort was mounted to produce a complete, general-purpose language expected to replace both Fortran and COBOL, the two languages in most widespread use at the time. The result, called Ada, was a large, full-featured language and even had constructs to support parallelism. It was required for many U.S. Department of Defense software contracts, and a large community of programmers eventually developed. Today, Ada is used within the defense and embedded systems community, but it did not supplant the established languages.
A project with several similarities to the DARPA HPCS program was the Japanese “5th Generation” project of the 1980s. Like the DARPA program, it was a ten-year project involving both a new programming model, presented as a more productive approach to software development, and new hardware architectures designed by multiple vendors to execute the programming model efficiently and in parallel. The language realizing the model, called CLP (Concurrent Logic Programming), was a dialect of Prolog specifically engineered for parallelism and high performance. The project was a success in training a generation of young Japanese computer scientists, but it has had no lasting influence on the parallel computing landscape.

Lessons

Programmers *do* value productivity, but reserve the right to define it. Portability, performance, and incrementality seem to have mattered more in the recent past than elegance of language design, power of expression, or even ease of use, at least when it came to programming large scientific applications. Successful new sequential languages have been adopted in the past twenty-five years, but each has been a modest step beyond an already established language (from C to C++, from C++ to Java). While the differences between each successful language have been significant, both timing of the language introduction and judicious use of familiar syntax and semantics were important. New “productivity” languages have also emerged (Perl, Python, and Ruby); but some of their productivity comes from the interpreted nature, and they are neither high-performance nor particularly suited to parallelism.

The PGAS languages, being smaller steps beyond established languages, thus present serious competition for the HPCS languages, despite the advanced, and even elegant, features designed into Chapel, Fortress, and X10. The most serious competition, however, comes from the more established message-passing interface, MPI, which has been widely adopted and provides a base against which any new language must compete. In the next section we describe some of the “productive” features of MPI, as a way of setting the bar for the HPCS languages, and some of the opportunities to improve over MPI as we move forward. New approaches to scalable parallel programming must offer a significant advantage over MPI, and cannot omit critical features that have proven useful in the MPI experience.

4.2 The HPCS Languages as a Group

The detailed, separate specifications for the HPCS languages can be found at [60]. In this section we consider the languages together and compare them along several axes in order to present a coherent view of them as a group.

Base Language
The HPCS languages use different sequential bases. X10 uses an existing object-oriented language, Java, inheriting both good and bad features. It adds to Java support for multidimensional arrays, value types, and parallelism, and it gains tool support from IBM’s extensive Java environment. Chapel and Fortress use their own, new object-oriented languages. An advantage of this approach is that the language can be tailored to science (Fortress even explores new, mathematical character sets), but the fact that a large intellectual effort is required in order to get the base language right has slowed development and may deter users.

Creating Parallelism
Any parallel programming model must specify how the parallelism is initiated. All three HPCS languages have parallel semantics; that is, there is no reliance on automatic parallelism, nor are the languages purely data parallel with serial semantics, like the core of HPF. All of them have dynamic parallelism for loops as well as tasks, and encourage the programmer to express as much parallelism as possible, with the idea that the compiler and runtime system will control how much is actually executed in parallel. There are various mechanisms for expressing different forms of task parallelism, including explicit spawn, asynchronous method invocation, and futures. Fortress is unusual in making parallelism the default semantics for both loops and for argument evaluation; this encourages programmers to “think in parallel,” which may result in more highly parallel code, but could also prove surprising to programmers.
The use of dynamic parallelism is the most significant semantic difference between the HPCS language and the existing PGAS languages with their static parallelism model. It presents the biggest opportunity to improve performance and ease of use relative to these PGAS languages and MPI. Having dynamic thread support along with data parallel operators may encourage a higher degree of parallelism in the applications, and allows the simplicity of expressing this parallelism directly rather than mapping it to a fixed process model in the application. The fine-grained parallelism can be used to mask

communication latency, reduce stalls at synchronization points, and take advantage of hardware extensions such as SIMD units and hyperthreading within a processor.

The dynamic parallelism is also the largest implementation challenge for the HPCS languages, since it requires significant runtime support to manage. The experience with Charm++ shows the feasibility of such runtime support for a class of very dynamic applications with limited dependencies [62]. A recent UPC project to apply multithreading to a matrix factorization problem reveals some of the challenges that arise from more complex dependencies between tasks. In that UPC code, the application-level scheduler manages user level threads on top of UPC's static process model: it must select tasks on the critical path to avoid idle time, delay allocating memory for noncritical tasks to avoid running out of memory, and ensure that tasks run long enough to gain locality benefits in the memory hierarchy. The scheduler uses application-level knowledge to meet all of these constraints, and performance depends critically on the quality of that information; it is not clear how such information would be communicated to one of the HPCS language runtimes.

Communication and Data Sharing

All three of the HPCS languages use a global address space for sharing state, rather than an explicit message-passing model. They all support shared multidimensional arrays as well as pointer-based data structures. X10 originally allowed only remote method invocations rather than direct reads and writes to remote values, but this restriction has been relaxed with the introduction of "implicit syntax," which is syntactic sugar for a remote read or write method invocation.

Global operations such as reductions and broadcasts are common in scientific codes, and while their functionality can be expressed easily in shared memory using a loop, they are often provided as libraries or intrinsics in parallel programming models. This allows for tree-based implementations and the use of specialized hardware that exists on some machines. In MPI and some of the existing PGAS languages, these operations are performed as "collectives": all processes invoke the global operation together so that each process can perform the local work associated with the operation. In data parallel languages global operations may be converted to collective operations by the compiler. The HPCS languages provide global reductions without explicitly involving any of the other threads as a collective: a single thread can execute a reduction on a shared array. This type of one-sided global operation fits nicely in the PGAS semantics, as it avoids some of the issues related to processes modifying the data involved in a collective while others are performing the collective [58]. However, the performance implications are not clear. To provide tree-based implementation and allow work to be performed locally, a likely implementation will be to spawn a remote thread to reduce the values associated with each process. Since that thread may not run immediately, there could be a substantial delay in waiting for such global operations to complete.

Locality

The HPCS languages use a variation of the PGAS model to support locality optimizations in shared data structures. X10's "places" and Chapel's "locales" provide a logical notion of memory partitions. A typical scenario maps each memory partition at program startup to a given physical compute node with one or more processors and its own shared memory. Other mappings are possible, such as one partition per processor or per core. Extensions to allow for dynamic creation of logical memory partitions have been discussed, although the idea is not fully developed. Fortress has a similar notion of a "region," but regions are explicitly tied to the machine structure rather than virtualized, and regions are hierarchical to reflect the design of many current machines.

All three languages support distributed data structures, and in particular distributed arrays that include user-defined distributions. These are much more general than in the existing PGAS languages. In Fortress the distribution support is based on the machine-dependent region hierarchy and is delegated to libraries rather than being in the language itself.

Synchronization among Threads and Processes

The most common synchronization primitives used in parallel applications today are locks and barriers. Barriers are incompatible with the dynamic parallelism model in the HPCS languages, although their effect can be obtained by waiting for a set of threads to complete. X10 has a sophisticated synchronization mechanism called "clocks," which can be thought of as barriers with attached tasks. Clocks provide a very general form of global synchronization that can be applied to subsets of threads.

In place of locks, which are viewed by many as cumbersome and error prone, all three languages support atomic blocks. Atomic blocks are semantically more elegant than locks, because the syntactic structure forces a matching "begin" and "end" to each critical region, and the block of code is guaranteed to be atomic with respect to all other operations in the program (avoiding the problems of acquiring the wrong lock, or deadlock). Atomic blocks place a larger burden on runtime support: one simple legal implementation involves a single lock to protect all atomic blocks[4], but the performance of such an implementation is probably unacceptable. More aggressive implementations will use speculative execution and rollback, possibly relying on hardware support within shared memory systems. The challenge comes from the use of a single global notion of atomicity, whereas locks may provide atomicity on two separate data structures using two separate locks. The information that the two data structures are unaliased must be discovered dynamically in a setting that relies on atomics. The support for atomics is not the same across the three HPCS languages. Fortress has abortable atomic sections, and X10 limits atomic blocks to a single place, which allows for a lock-per-place implementation.

The languages also have some form of a "future" construct that can be used for producer-consumer parallelism. In Fortress if one thread tries to access the result of another spawned thread, it will automatically stall until the value is available. In X10 there is a distinct type for the variable on which one waits and its contents, so the point of potential

[4]This assumes atomic blocks are atomic only with respect to each other, not with respect to individual reads and writes performed outside and atomic block.

stalls is more explicit. Chapel has the capability to declare variables as “single” (single writer) or “sync” (multiple readers and writers).

Moving Forward

Recently a workshop was held at Oak Ridge National Laboratory, bringing together the three HPCS language vendors, computer science researchers representing the PGAS languages and MPI, potential users from the application community, and program managers from DARPA and the Department of Energy’s Office of Advanced Scientific Computing. In this section we describe some of the findings of the workshop at a high level and present the tentative plan for further progress that evolved there.

The workshop was organized in order to explore the possibility of converging the HPCS languages to a single language. Briefings were presented on the status of each of the three languages and an effort was made to identify the common issues involved in completing the specifications and initiating the implementations. Potential users offered requirements for adoption, and computer science researchers described recent work in compilation and runtime issues for PGAS languages. One high-level finding of the workshop was the considerable diversity in the overall approaches being taken by the vendors, the computer science research relevant to the HPCS language development, and the application requirements.

Diversity in Vendor Approaches

Although the three vendors are all well along the path to completing designs and prototype implementations of these languages that are intended to increase the productivity of software developers, they are not designing three versions of the same type of object. X10, for example, is clearly intended to fit into IBM’s extensive Java programming environment. As described earlier, it uses Java as a base language, allowing multiple existing tools for parsing, compiling, and debugging to be extended to the new parallel language. Cray’s approach is more revolutionary, with the attendant risks and potential benefits; Chapel is an attempt to design a parallel programming language from the ground up. Sun is taking a third approach, providing a framework for experimentation with parallel language design, in which many aspects of the language are to be defined by the user and many of the features are expected to be provided by libraries instead of by the language itself. One novel feature is the option of writing code with symbols that, when displayed, can be typeset as classical mathematical symbols, improving the readability of the “code.”

Diversity in Application Requirements

Different application communities have different expectations and requirements for a new language. Although only a small fraction of potential HPC applications were represented at the workshop, there was sufficient diversity to represent a wide range of positions with respect to the new languages.

One extreme is represented by those applications for which the current programming model—MPI together with a conventional sequential language—is working well. In many cases MPI is being used as the MPI Forum intended: the MPI calls are in libraries written by specialists, and the application programmer sees these library interfaces rather than MPI itself, thus bypassing MPI's ease-of-use issues. In many of the applications content with the status quo, the fact that the application may have a long life (measured in decades) amortizes the code development effort and makes development convenience less of an issue.

The opposite extreme is represented by those for whom rapidity of application development is *the* critical issue. Some of these codes are written in a day or two for a special purpose and then discarded. For such applications the MPI model is too cumbersome and error prone, and the lack of a better model is a genuine barrier to development. Such applications cannot be deployed at all without a significant advance in the productivity of programmers.

Between these extremes is, of course, a continuously variable range of applications. Such applications would welcome progress in the ease of application development and would adopt a new language in order to obtain it, but any new approach must not come at the expense of qualities they find essential in the status quo: portability, completeness, support for modularity, and at least some degree of performance transparency.

Diversity in Relevant Computer Science Research

The computer science research most relevant to the HPCS language development is that being carried out in the various PGAS language efforts. Like the HPCS languages, PGAS languages offer a global view of data, with explicit control of locality in order to provide performance. Their successful implementation has involved research into compilation techniques that are likely to be useful as the HPCS languages finalize language design and begin developing production compilers, or at least serious prototypes. The PGAS languages, and associated research, also share with the HPCS languages the need for runtime systems that support lightweight communication of small amounts of data. Such portable runtime libraries are being developed in both the PGAS and MPI implementation research projects [55, 56].

The PGAS languages are being used in applications to a limited extent, while the HPCS languages are still being tested for expressibility on a number of example kernels and benchmarks.

The issue of the runtime system is of particular interest, because standardizing it would bring multiple benefits. A standard syntax and semantics for a low-level communication library that could be efficiently implemented on a variety of current communication hardware and firmware would benefit the entire programming model implementation community: HPCS languages, PGAS languages, MPI implementations, and others. A number of such portable communications exist now (GASNet [54], ADI-3, ARMCI), although most have been developed with a particular language or library implementation in mind.

Immediate Needs

Despite the diverse approaches to the languages being taken by the vendors, some common deficiencies were identified in the workshop. These are areas that have been postponed while the initial language designs are being formulated, but now really need to be addressed if the HPCS languages are to catch the attention of the application community.

Performance
The Fortress [59] and X10 [67] implementations are publicly available and an implementation of Chapel exists, but is not yet released. So far these prototype implementations have focused on expressivity rather than performance. This direction has been appropriate up to this point, but now that one can see how various benchmarks and kernels can be expressed in these languages, one wants to see how they can be compiled for performance competitive with existing approaches, especially for scalable machines. While the languages may not reach their full potential without the HPCS hardware being developed by the same vendors, the community needs some assurance that the elegant language constructs, such as those used to express data distributions, can indeed be compiled into efficient programs for current scalable architectures.

Completeness
The second deficiency involves completeness of the models being presented. At this point none of the three languages has an embedded parallel I/O model. At the very least the languages should define how to read and write distributed data structures from and into single files, and the syntax for doing so should enable an efficient implementation that can take advantage of parallel file systems.

Another feature that is important in multiphysics applications is modularity in the parallelism model, which allows subsets of processors to act independently. MPI has “communicators” to provide isolation among separate libraries or separate physics models. The PGAS languages are in the process of introducing “teams” of processes to accomplish the same goals. Because the HPCS languages have a dynamic parallel operator combined with data parallel operators, this form of parallelism should be expressible, but more work is needed to understand the interactions between the abstraction mechanisms used to create library interfaces and the parallelism features.

A Plan for Convergence
On the last day of the workshop a plan for the near future emerged. It was considered too early to force a convergence on one language in the near term, given that the current level of diversity seemed to be beneficial to the long-term goals of the HPCS project, rather than harmful. The level of community and research involvement could be increased by holding a number of workshops over the next few years in specific areas, such as memory models, data distributions, task parallelism, parallel I/O, types, tools, and interactions with other libraries. Preliminary plans were made to initiate a series of meetings, loosely modeled on the MPI process, to explore the creation of a common runtime library specification.

An approximate schedule was proposed at the workshop. In the next eighteen months (i.e., by the end of the calendar year 2007) the vendors should be able to freeze the syntax of their respective languages. In the same time frame, workshops should be held to address the research issues described above. Vendors should be encouraged to establish “performance credibility” by demonstrating the competitive performance of some benchmark on some high-performance architecture. This would not necessarily involve the entire language, nor would it necessarily demand better performance of current versions of the benchmark. The intent would be to put to rest the notion that a high-productivity language precludes high performance. Also during this time, a series of meetings should be held to determine whether a common communication subsystem specification can be agreed upon.

The following three years should see the vendors improve performance of all parts of their languages. Inevitably, during this period the languages will continue to evolve independently as experience is gained. At the same time, aggressive applications should get some experience with the languages. After this period, when the languages have had an opportunity to evolve in both design and implementations while applications have had the chance to identify strengths and weaknesses of each, would come the consolidation period. At this point (2010-2011) an MPI forum-like activity could be organized to take advantage of the experience now gained, in order to cooperatively design a single HPCS language with the DARPA HPCS languages as input (much as the MPI Forum built on, but did not adopt any of, the message-passing library interfaces of its time).

By 2013, then, we could have a new language, well vetted by the application community, well implemented by HPCS vendors and even open-source developers, that could truly accelerate productivity in the development of scientific applications.

Conclusion
The DARPA HPCS language project has resulted in exciting experimental language research.. Excellent work is being carried out by each of the vendor language teams, and it is to be hoped that Sun’s language effort will not suffer from the end of Sun’s hardware development contract with DARPA. Now is the time to get the larger community involved in the high-productivity programming model and language development effort, through workshops targeted at outstanding relevant research issues and through experimentation with early implementations of all the “productivity” languages. In the long run, acceptance of any new language for HPC is a speculative proposition, but there is much energy and enthusiasm for the project, and a reasonable plan is in place by which progress can be made. The challenge is to transition the HPCS language progress made to-date into the future community efforts.

5. Research on Defining and Measuring Productivity

Productivity research under the HPCS program has explored better ways to define and measure productivity. The work that was performed under Phase 1 and Phase 2 of the HPCS program had two major thrusts. The first thrust was in the study and analysis of software development time. Tools for accomplishing this thrust included surveys, case studies, and software evaluation tools. The second thrust was the development of a productivity figure of merit, or metric. In this section, we will present the research that occurred in Phase 1 and Phase 2 of the HPCS program in these areas. This research has been described in depth in [69] and [70]. This section will provide an overview of the research documented in those publications and provide pointers to specific articles for each topic.

5.1 Software Development Time

Much of architectural development in the past decades has focused on raw hardware performance and the technologies responsible for it, including faster clock speeds, higher transistor densities, and advanced architectural structures for exploiting instruction-level parallelism. Performance metrics, such as GFLOPS/second and MOPS/watt, were exclusively employed to evaluate new architectures, and reflect this focus on hardware performance.

A key, revolutionary aspect of the HPCS program is its insistence that software be included in any performance metric used to evaluate systems. The program participants realized early on that true time to solution on any HPC system is not just the execution time, but is in fact the sum of development time and execution time. Not only are both important, but the piece of the puzzle that has always been ignored, development time, has always dominated time-to-solution – usually by orders of magnitude. This section describes the ground-breaking research done in understanding and quantifying development time and its contribution to the productivity puzzle.

5.1.1 Understanding the Users[5]

Attempts to evaluate the productivity of an HPC system require an understanding of what productivity means to all its users. For example, researchers in computer science work to push the boundaries of computational power, while computational scientists use those advances to achieve increasingly detailed and accurate simulations and analysis. Staff at shared resource centers enable broad access to cutting-edge systems while maintaining high system utilization. While each of these groups use HPC resources, their differing needs and experiences affect their definitions of productivity.

Computational scientists and engineers face many challenges when writing codes for high-end computing systems. The HPCS program is developing new machine architectures, programming languages, and software tools to improve the productivity of scientists and engineers. Although the existence of these new technologies is important for improving productivity, they will not achieve their stated goals if individual scientists and engineers are not able to effectively use them to solve their problems. A necessary first step in determining the usefulness of new architectures, languages and tools is to gain a better understanding of what the scientists and engineers do, how they do it, and what problems they face in the current high-end computing development environment. Because the community is very diverse, it is necessary to sample different application

[5] Material taken from [73] and [76].

domains to be able to draw any meaningful conclusions about the commonalties and trends in software development in this community.

Two important studies were carried out during Phases 1 and 2 of the HPCS program to better identify the needs and characteristics of the user and application spaces that are the targets for these new architectures. The first team worked with the San Diego Supercomputer Center (SDSC) and its user community.[73]. This team analyzed data from a variety of sources, including SDSC support tickets, system logs, HPC developer interviews, and productivity surveys distributed to HPC users, to better understand how HPC systems are being used, and where the best opportunities for productivity improvements are. The second team analyzed 10 large software projects from different application domains to gain deeper insight into the nature of software development for scientific and engineering software.[76]. This team worked with ASC-Alliance projects, which are DOE-sponsored computational science centers based at five universities across the country, as well as codes from the HPCS mission partners.

Although the perspectives and details of the two studies were quite different, a number of common conclusions emerged with major relevance for the community of HPCS tool developers. Table and Table summarize the conclusions of the two studies.

HPC users have diverse concerns and difficulties with productivity.
Users with the largest allocations and most expertise are not necessarily the most productive.
Time to solution is the limiting factor for productivity on HPC systems, not computational performance.
Lack of publicity and education are not the main roadblocks to adoption of performance and parallel debugging tools – ease of use is more significant.
HPC programmers do not require dramatic performance improvements to consider making structural changes to their codes.
A computer science background is not crucial to success in performance optimization.
Visualization is key to achieving high productivity in HPC in most cases.

Table 5.1. SDSC Study Conclusions

Goals and Drivers of Code Development
• Code performance is not the driving force for developers or users; the science and portability are of primary concern. • Code success depends on customer satisfaction.
Actions and Characteristics of Code Developers
• Most developers are domain scientists or engineers, not computer scientists. • The distinction between developer and user is blurry. • There is high turnover in the development team.
Software Engineering Process and Development Workflow

• There is minimal but consistent use of software engineering practices. • Development is evolutionary at multiple levels. • Tuning for a specific system architecture is rarely done, if ever. • There is little reuse of MPI frameworks. • Most development effort is focused on implementation rather than maintenance.
Programming Languages
• Once selected, the primary language does not change. • Higher level languages (e.g., Matlab) are not widely adopted for the core of applications.
Verification and Validation
• Verification and validation are very difficult in this domain. • Visualization is the most common tool for validation.
Use of Support Tools during Code Development
• Overall, tool use is lower than in other software development domains. • Third party (externally developed) software and tools are viewed as a major risk factor.

Table 5.2. ASC/HPCS Project Survey

Common themes from these two studies are that end results are more important than machine performance (we're interested in the engine, but we drive the car!); visualization and easy-to-use tools are key; and HPC programmers are primarily domain experts driven by application needs, not computer scientists interested in fast computers. The implications for HPCS affect both the types of tools that should be developed and how productivity should ultimately be measured on HPCS systems from the user's perspective.

5.1.2 Focusing the Inquiry[6]

Given the difficulty in deriving an accurate general characterization of HPC programmers and their productivity characteristics, developing a scientific process for evaluating productivity is even more difficult. One team of researchers responded with two broad commitments that could serve more generally to represent the aims of the productivity research team as a whole: [79]

1. Embrace the broadest possible view of productivity, including not only machine characteristics but also human tasks, skills, motivations, organizations, and culture, to name just a few; and

2. Put the investigation of these phenomena on the soundest scientific basis possible, drawing on well-established research methodologies from relevant fields, many of which are unfamiliar within the HPC community.

[6] Material taken from [79] and [88].

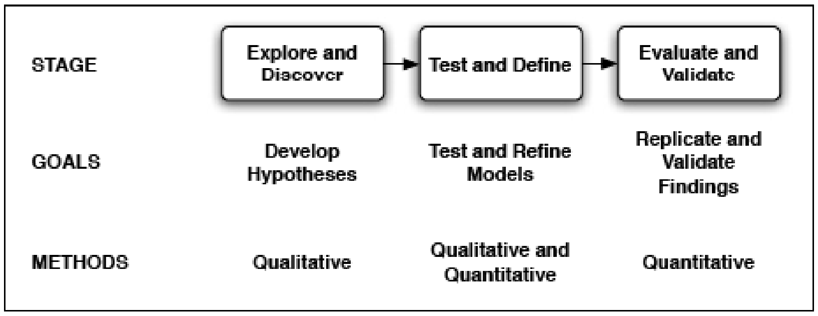


Figure 5.1. Research Framework

This team of researchers outlined a three-stage research design shown in Figure 5.1. For the first stage, exploration and discovery, case studies and other qualitative methods are used to produce the insights necessary for hypothesis generation. For the second stage, qualitative and quantitative methods are combined to test and refine models. The quantitative tool used by this team in the second stage was HackyStat, an in-process software engineering measurement and analysis tool. Patterns of activity were used to generate a representative workflow for HPC code development. In the third stage, the workflows were validated via quantitative models that were then used to draw conclusions about the process of software development for HPC systems.

A number of case studies were produced in the spirit of the same framework, with the goal of defining a workflow that is particular to large-scale computational scientific and engineering projects in the HPC community[88]. These case studies identified seven development stages for a computational science project:

1. Formulate questions and issues
2. Develop computational and project approach
3. Develop the program
4. Perform verification and validation
5. Make production runs
6. Analyze computational results
7. Make decisions

These tasks strongly overlap each other, with a lot of iteration among the steps and within each step. Life cycles for these projects are very long, in some cases 30-40 years or more, far longer than typical IT projects. Development teams are large and diverse, and individual team members often don't have working experience with the modules and codes being developed by other module sub-teams, making software engineering challenges much greater than those of typical IT projects. A typical project workflow is shown in Figure 5.2.

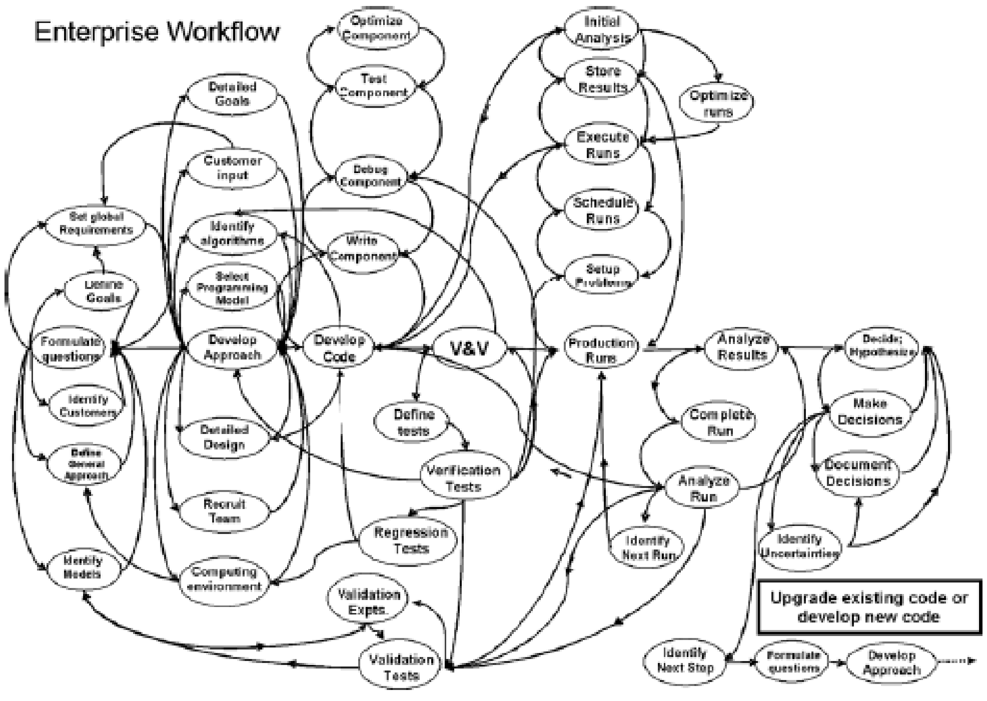


Figure 5.2. Comprehensive workflow for large-scale CSE project

With these projects, we begin to see the development of frameworks and representations to guide the productivity evaluation process. The derivation of workflows is key to understanding any process, and a disciplined understanding is necessary before any improvements can be made or assessed. The workflows for software development could be as complex as the projects they reflect and as diverse as the programmers who implement them. In the next section, we will see some examples of the types of tools that can be used once formal representations, specific measurements, and quantifiable metrics have been defined.

5.1.3 Developing the Evaluation Tools[7]

A prerequisite for the scientific and quantified study of productivity in HPC systems is the development of tools and protocols to study productivity. As a way to understand the particular needs of HPC programmers, prototype tools were developed in Phases 1 and 2 of the HPCS program to study productivity in the HPC community. The iInitial focus has been on understanding the effort involved in coding for HPC systems and the defects that occur in developing programs. Models of workflows that accurately explain the process that HPC programmers use to build their codes were developed. Issues such as time involved in developing serial and parallel versions of a program, testing and debugging of the code, optimizing the code for a specific parallelization model (e.g., MPI, OpenMP) and tuning for specific machine architectures were all topics of study. Once those models are developed, the HPCS system developers can then work on the

[7] Material taken from [75], [78], and [82].

more crucial problems of what tools and techniques will better optimize a programmer's ability to produce quality code more efficiently.

Since 2004, studies of programmer productivity have been conducted, in the form of human-subject experiments, at various universities across the U.S. in graduate level HPC courses.[75] Graduate students in HPC classes are fairly typical of novice HPC programmers who may have years of experience in their application domain but very little in HPC-style programming experience. In the university studies, multiple students were routinely given the same assignment to perform, and experiments were conducted to control for the skills of specific programmers (e.g., experimental meta-analysis) in different environments. Due to their relatively low cost, student studies are an excellent environment for debugging protocols that might be later used on practicing HPC programmers. Limitations of student studies include the relatively short programming assignments, due to the limited time in a semester, and the fact that these assignments must be picked for their educational value to the students as well as their investigative value to the research team.

Using the experimental environment developed under this research, various hypotheses about HPC code development can be tested and validated (or disproven!). Table shows some sample hypotheses and how they would be tested using the various tools that have been developed. In addition to verifying hypotheses about code development, the classroom experiments have moved beyond effort analysis and started to look at the impact of defects (e.g., incorrect or excessive synchronization, incorrect data decomposition) on the development process. By understanding how, when, and what kinds of defects appear in HPC codes, tools and techniques can be developed to mitigate these risks and improve the overall workflow. Automatically determining workflow is not precise, so these studies involved a mixture of process activity (e.g., coding, compiling, executing) and source code analysis techniques.

Hypothesis	Test Measurement
The average time to fix a defect due to race conditions will be longer in a shared memory program compared to a message-passing program.	Time to fix defects due to race conditions
On average, shared memory programs will require less effort than message passing, but the shared memory outliers will be greater than the message passing outliers.	Total development time
There will be more students who submit incorrect shared memory programs compared to message-passing programs.	Number of students who submit incorrect solutions
An MPI implementation will require more code than an OpenMP implementation.	Size of code for each implementation

Table 5.3. HPC Code development Hypotheses

Some of the key results of this effort include:

- Productivity measurements of various workflows, where productivity is defined as relative speedup divided by relative effort. Relative speedup is reference (sequential) execution time divided by parallel execution time, and relative effort is parallel effort divided by reference (sequential effort). Results of student measurements for various codes show that this metric behaves as expected, i.e., good productivity means lower total effort, lower execution time and higher speedup.

- Comparison of XMT-C (a PRAM-like execution model) to MPI-based codes in which, on average, students required less effort to solve the problem using XMT-C compared to MPI. The reduction in mean effort was approximately 50%, which was statistically significant according to the parameters of the study.

- Comparison of OpenMP and MPI defects did not yield statistically significant results, which contradicts a common belief that shared memory programs are harder to debug. Since defect data collection was based on programmer-supplied effort forms, which are not accurate, more extensive defect analysis is required.

- Collection of low-level behavioral data from developers in order to understand the workflows that exist during HPC software development. A useful representation of HPC workflow could both help characterize the bottlenecks that occur during development and support a comparative analysis of the impact of different tools and technologies upon workflow. A sample workflow would consist of five states: serial coding, parallel coding, testing, debugging, and optimization.

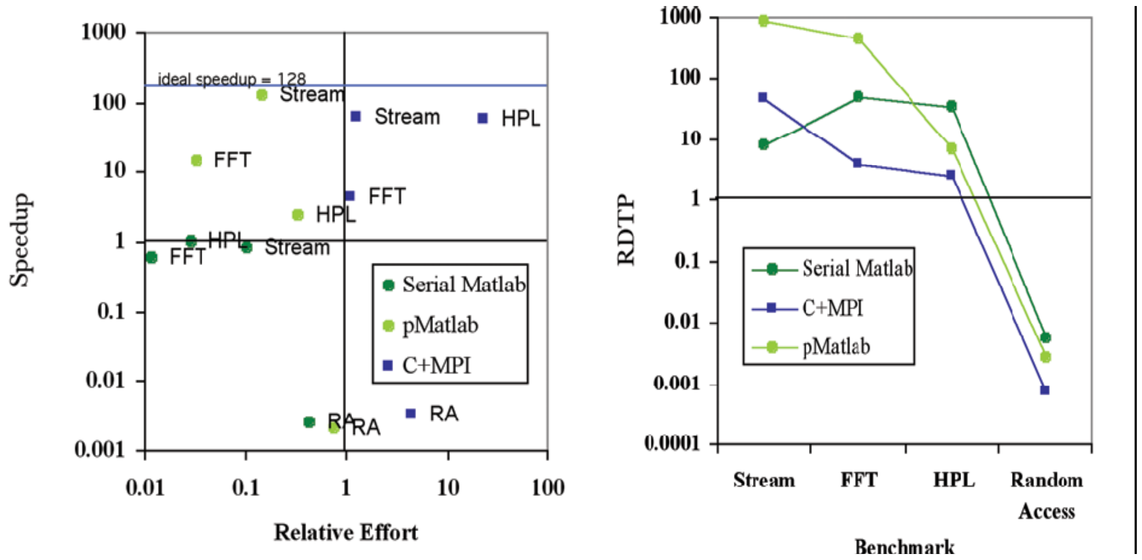
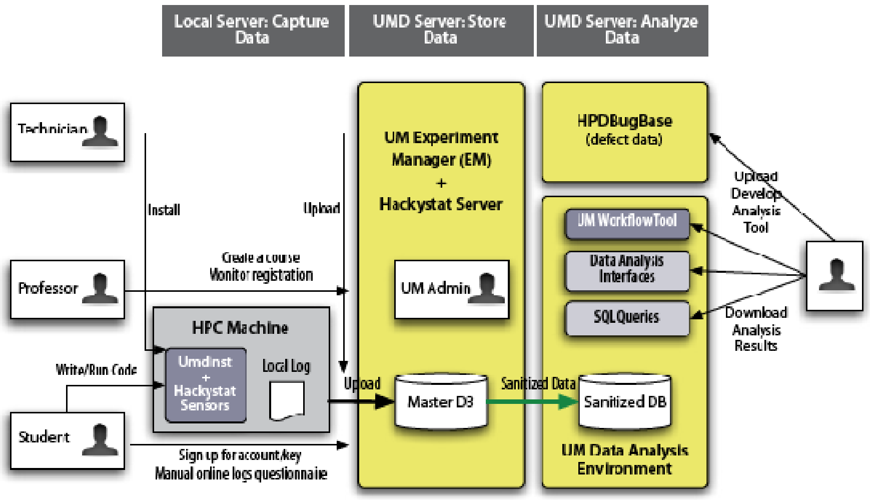


Figure 5.3. Speedup vs. Relative Effort and RDTP for HPC Challenge

Error! Reference source not found. presents results of the relative development time productivity metric, using the HPCChallenge benchmark described earlier in this chapter. With the exception of Random Access (the implementation of which does not scale well on distributed memory computing clusters), the MPI implementations all fall into the upper-right quadrant of the graph, indicating that they deliver some level of parallel speedup, while requiring greater effort than the serial code. As expected, the serial Matlab implementations do not deliver any speedup, but all require less effort than the serial code. The pMatlab implementations (except Random Access) fall into the upper-left quadrant of the graph, delivering parallel speedup while at the same time requiring less effort.

In another pilot study[78], students worked on a key HPC code using C and PThreads in a development environment that included automated collection of editing, testing, and command line data using Hackystat. The "serial coding" workflow state was automatically inferred as the editing of a file not containing any parallel constructs (such as MPI, OpenMP, or PThread calls), and the "parallel coding" workflow state as the editing of a file containing these constructs. The "testing" state was inferred as the occurrence of unit test invocation using the CUTest tool. In the pilot study, the debugging or optimization workflow states could not be inferred, as students were not provided with tools to support either of these activities that we could instrument. Based on these results, researchers concluded that workflow inference may be possible in an HPC context and hypothesize that it may actually be easier to infer these kinds of workflow states in a professional setting, since more sophisticated tool support is often available that can help support conclusions regarding the intent of a development activity. It is also possible that a professional setting may reveal that the five states initially selected are appropriate for all HPC development contexts. It may be that there is no "one size fits all" set of workflow states, and that custom sets of states for different HPC organizations will be required in order to achieve the goal of accurately modeling the HPC software development process.



Local Server: Capture Data
UMD Server: Store Data
UMD Server: Analyze Data
Technician
Install
Upload
UM Experiment Manager (EM) + Hackystat Server
HPDBugBase (defect data)
Upload Develop Analysis Tool
UM WorkflowTool
Data Analysis Interfaces
SQL Queries
Download Analysis Results
Professor
Create a course Monitor registration
UM Admin
HPC Machine
UMDinst + Hackystat Sensors
Local Log
Write/Run Code
Upload
Master DB
Sanitized Data
Sanitized DB
Student
Sign up for account/key Manual online logs questionnaire
UM Data Analysis Environment

Figure 5.4. Experiment Manager Structure

To support the tools and conclusions described in this section, an Experiment Manager was developed to more easily collect and analyze data during the development process. It includes effort, defect and workflow data, as well as copies of every source program used during development. Tracking effort and defects should provide a good data set for building models of productivity and reliability of high-end computing (HEC) codes. Figure 5.4 shows the components of the Experiment Manager and how they interact.

Another key modeling tool developed in Phases 1 and 2 of the HPCS program involves analysis of an HPC development workflow using sophisticated mathematical models.[82] This work is based on the observation that programmers go through an identifiable, repeated process when developing programs, which can be characterized by a directed graph workflow. Timed Markov Models (TMMs) are one way to quantify such directed graphs . A simple TMM was developed that captures the workflows of programmers working alone on a specific problem. An experimental setup was constructed in which the student homework in a parallel computing class was instrumented. Tools were developed for instrumentation, modeling, and simulating different what-if scenarios in the modeled data. Using our model and tools, the workflows of graduate students programming the same assignment in C/MPI4 and UPC5 were compared — something that is not possible without a quantitative model and measurement tools. Figure shows the workflow used, where:

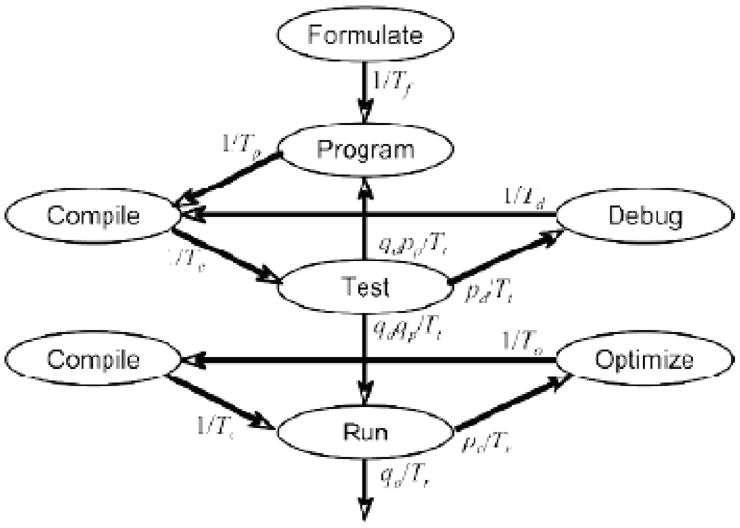


Figure 5.5. Lone programmer workflow

- Tf represents the time taken to formulate the new algorithmic approach.
- Tp is the time necessary to implement the new algorithm in a program.
- Tc is the compile time.
- Tt is the time necessary to run a test case during the debugging phase.
- Td is the time the programmer takes to diagnose and correct the bug.
- Tr is the execution time for the performance tuning runs. This is the most obvious candidate for a constant that should be replaced by a random variable.
- To is the time the programmer takes to identify the performance bottleneck and program an intended improvement.
- Pp is the probability that debugging will reveal a necessity to redesign the program.
- Pd is the probability that more debugging will be necessary.
- Po is the probability that more performance optimization will be necessary.

- qp, qd, and qo are 1 - Pp, 1 - Pd, and 1 - Po, respectively.

Using the TMM, the workflow of UPC programs was compared to that of C/MPI programmers on the same problem. The data collection process gathers enough data at compile time and run time so that programmer experience can be accurately recreated offline. A tool for automatic TMM generation from collected data was built, as well as a tool for representing and simulating TMMs. This allowed replay of the sequence of events (every compile and run) and collection-specific data that may be required by the modeling process but was not captured while the experiment was in progress. The resulting data showed that a "test" run is successful 8% of the time for C/MPI and 5% of the time for UPC; however, in the optimization cycle, 28% of C/MPI runs introduced new bugs compared to only 24% of UPC runs. It is not clear whether these differences are significant, given this small sample size. A programmer spends much longer to attempt an optimization (763 seconds for UPC and 883 seconds for C/MPI) than to attempt to remove a bug (270-271 seconds). The time to optimize UPC (763 seconds) is smaller that for MPI (883 seconds), suggesting perhaps that UPC optimization is carried out in a more small-granularity, rapid-feedback way.

The research presented in this section takes the formalisms and representations developed under productivity research and begins the scientific process of gathering measurements, building and verifying models, and then using those models to gain insight into a process, either via human analysis or via formal, mathematical methods. This is where the leap from conjecture to scientific assertion begins, and the measurements and insights presented here represent a breakthrough in software engineering research. The immediate goal for the HPCS program is to use these analytical tools to compare current HPC systems with those being developed for HPCS. However, we see a broader applicability for these types of methods in the computing industry. Future work in this area has the potential to take these models and use them not only to gain insight, but also to predict the performance of a given process. If we can build predictive models, and tailor those models to a particular user base or application class, the gains in productivity could eventually outstrip the capability to build faster machines, and may have a more lasting impact on software engineering for HPC as a whole.

5.1.4 Advanced Tools for Engineers

Several advanced tools were developed under HPCS productivity research that should be mentioned here but cannot be described in detail because of space constraints. These are briefly described in the following paragraphs; more detailed information can be found in [70].

<u>Performance Complexity (PC) Metric [81]:</u> an execution-time metric that captures how complex it is to achieve performance and how transparent performance results are. PC is based on performance results from a set of benchmark experiments and related performance models that reflect the behavior of a program. Residual modeling errors are used to derive PC as a measure for how transparent program performance is and how complex the performance appears to the programmer. A detailed description for calculating compatible P and PC values is presented and uses results from a parametric

benchmark to illustrate the utility of PC for analyzing systems and programming paradigms.

Compiler-guided Instrumentation for Application Behavior Understanding:[83] an integrated compiler and runtime approach that allows the extraction of relevant program behavior information by judiciously instrumenting the source code and deriving performance metrics such as range of array reference addresses, array access stride information or data reuse characteristics. This information ultimately allows programmers to understand the performance of a given machine in relation to rational program constructs. The overall organization of the compiler and run-time instrumentation system is described and preliminary results for a selected set of kernel codes are presented. This approach allow programmers to derive a wealth of information about the program behavior with a run-time overhead of less than 15% of the original code's execution time, making this approach attractive for instrumenting and analyzing codes with extremely long running times where binary-level approaches are simply impractical.

Symbolic Performance Modeling of HPCS:[84] a new approach to performance model construction, called modeling assertions (MA), which borrows advantages from both the empirical and analytical modeling techniques. This strategy has many advantages over traditional methods: isomorphism with the application structure; easy incremental validation of the model with empirical data; uncomplicated sensitivity analysis; and straightforward error bounding on individual model terms. The use of MA is demonstrated by designing a prototype framework, which allows construction, validation, and analysis of models of parallel applications written in FORTRAN or C with the MPI communication library. The prototype is used to construct models of NAS CG, SP benchmarks and a production-level scientific application called Parallel Ocean Program (POP).

Compiler Approaches to Performance Prediction and Sensitivity Analysis:[86] the Source Level Open64 Performance Evaluator (SLOPE) approaches performance prediction and architecture sensitivity analysis by using source level program analysis and scheduling techniques. In this approach, the compiler extracts the computation's high-level data-flow-graph information by inspection of the source code. Taking into account the data access patterns of the various references in the code, the tool uses a list-scheduling algorithm to derive performance bounds for the program under various architectural scenarios. The end result is a very fast prediction of what the performance could be and, more importantly, why the predicted performance is what it is. This research experimented with a real code that engineers and scientists use. The results yield important qualitative performance sensitivity information. This can be used to allocate computing resources to the computation in a judicious fashion, for maximum resource efficiency and to help guide the application of compiler transformations such as loop unrolling.

5.2 Productivity Metric[8]

Another key activity in the HPCS productivity research was the development of productivity metrics that can be used to evaluate both current and future HPCS systems. The former is necessary to establish a productivity baseline against which to compare the projected improvements of the latter. In either case, the metric must be quantifiable, measurable, and demonstrable over the range of machines competing in the program.

Establishing a single, reasonably objective quantitative framework to compare competing high productivity computing systems has been difficult to accomplish. There are many reasons for this, not the least of which is the inevitable subjective component of the concept of productivity. Compounding the difficulty, there are many elements that make up productivity and these are weighted and interrelated differently in the wide range of contexts into which a computer may be placed. But because significantly improved productivity for high performance government and scientific computing is the key goal of the HPCS program, evaluating this critical characteristic across these contexts is clearly essential.

This is not entirely a new phenomenon. Anyone who has driven a large-scale computing budget request and procurement has had to address the problem of turning a set of preferences and criteria, newly defined by management, into a budget justification and a procurement figure of merit that will pass muster with agency (and OMB) auditors. The process of creating such a procurement figure of merit helps to focus the mind and cut through the complexity of competing user demands and computing options.
The development of productivity metrics was addressed from both a business and a system-level perspective in Phase 1 and Phase 2 research. The results of both phases are summarized in the following sections.

5.2.1 Business Perspective[9]

High performance computing (HPC), also known as supercomputing, makes enormous contributions not only to science and national security, but also to business innovation and competitiveness — yet senior executives often view HPC as a cost, rather than a value investment. This is largely due to the difficulty businesses and other organizations have had in determining the return on investment (ROI) of HPC systems.

Traditionally, HPC systems have been valued according to how fully they are utilized (i.e., the aggregate percentage of time that each of the processors of the HPC system is busy); but this valuation method treats all problems equally and does not give adequate weight to the problems that are most important to the organization. With no ability to properly assess problems having the greatest potential for driving innovation and competitive advantage, organizations risk purchasing inadequate HPC systems or, in some cases, foregoing purchases altogether because they cannot be satisfactorily justified.

[8] Material drawn from [72], [77], and [80].
[9] Material drawn from [72] and [77].

This stifles innovation within individual organizations and, in the aggregate, prevents the U.S. business sector from being as globally competitive as it could and should be. The groundbreaking July 2004 “Council on Competitiveness Study of U.S. Industrial HPC Users,” sponsored by the Defense Advanced Research Projects Agency (DARPA) and conducted by market research firm IDC, found that 97% of the U.S. businesses surveyed could not exist, or could not compete effectively, without the use of HPC. Recent Council on Competitiveness studies reaffirmed that HPC typically is indispensable for companies that exploit it.

It is increasingly true that to out-compete, companies need to out-compute. Without a more pragmatic method for determining the ROI of HPC hardware systems, however, U.S. companies already using HPC may lose ground in the global competitiveness pack. Equally important, companies that have never used HPC may continue to miss out on its benefits for driving innovation and competitiveness.

To help address this issue, we present an alternative to relying on system utilization as a measure of system valuation, namely, capturing the ROI by starting with a benefit-cost ratio (BCR) calculation. This calculation is already in use at the Massachusetts Institute of Technology, where it has proven effective in other contexts.

As part of the HPCS productivity research, two versions of the productivity metric were developed based on benefit-to-cost ratios. Numerical examples were provided to illustrate their use. The goal is to use these examples to show that HPC assets are not just cost items, but that they can contribute to healthy earnings reports as well as more productive and efficient staff. Detailed results are described in [72].

Another important barrier preventing greater HPC use is the scarcity of application software capable of fully exploiting current and planned HPC hardware systems. U.S. businesses rely on a diverse range of commercially available software from independent software vendors (ISVs). At the same time, experienced HPC business users want to exploit the problem-solving power of contemporary HPC hardware systems with hundreds, thousands or (soon) tens of thousands of processors to boost innovation and competitive advantage. Yet few ISV applications today can exploit (“scale to”) even 100 processors, and many of the most popular applications scale to only a few processors in practice.

Market forces and technical challenges in recent years have caused the ISVs to pull away from creating new and innovative HPC applications, and no other source has arisen to satisfy this market need. For business reasons, ISVs focus primarily on the desktop computing markets, which are much larger and therefore promise a better return on R&D investments. ISVs can sometimes afford to make modest enhancements to their application software so that it can run faster on HPC systems, but substantially revising existing applications or creating new ones typically does not pay off. As a result, the software that is available for HPC systems is often outdated and incapable of scaling to the level needed to meet industry’s needs for boosting problem-solving performance. In some cases, the applications that companies want simply do not exist.

This need for production-quality application software and middleware has become a soft spot in the U.S. competitiveness armor; a pacing item in the private sector's ability to harness the full potential of HPC. Without the necessary application software, American companies are losing their ability to aggressively use HPC to solve their most challenging problems and risk ceding leadership in the global marketplace. Market and resource barriers are described in detail in [77].

5.2.2 System Perspective[10]

Imagining that we were initiating a procurement in which the primary criterion would be productivity, defined as utility/cost, we developed figure of merit for total productivity. This framework includes such system measurables as machine performance and reliability, developer productivity, and administration overhead and effectiveness of resource allocation. These are all applied using information from the particular computing site that is proposing and procuring the HPCS computer. This framework is applicable across the broad range of environments represented by HPCS mission partners and others with science and enterprise missions that are candidates for such systems. The productivity figure of merit derived under this research is shown in **Error! Reference source not found.**. As a convention, the letters U, E, A, R, C are used to denote the variables of utility, efficiency, availability, resources, and cost, respectively. The subscripts indicate the variables that address system level (including administrative and utility) and job level factors.

$$P = \frac{U_{sys} E_{proj} E_{adm} E_{job} A_{sys} R}{C}$$

Figure 5.6. System-wide Productivity Figure of Merit

As is evident from this formulation, some aspects of the system level efficiency will never be amenable to measurement and will always require subjective evaluation. Only subjective evaluation processes can address the first two variables in the utility numerator, for example. In principle one can measure the last four variables, and the HPCS research program is addressing such measurements. A description of the steps required to using the overall system-level productivity figure of merit can be found in [80].

5.3 Conclusions

In this section, we have given a broad overview of the activities performed under Phase 1 and Phase 2 HPCS Productivity Research, in enough detail to communicate substantial results without misrepresenting the inherent complexity of the subject matter. In reality, the productivity of HPC users intrinsically deals with some of the brightest people on the planet, solving very complex problems, using the most complex computers in the world.

[10] Material drawn from [80].

The HPCS program has performed ground-breaking research into understanding, modeling, quantifying, representing, and analyzing this complex and difficult arena; and although much has been accomplished, the surface of productivity research has barely been scratched. The tools and methodologies developed under the HPCS program are an excellent base on which to build a full understanding of HPC productivity; but in the final analysis, those tools and methodologies must be applied to an area of human endeavor that is as diverse, specialized, individualistic, inconsistent, and even eccentric as the people who are its authors and creators. If HPCS productivity research is to have a hand in transforming the world of high-performance computing, it must evolve from a set of tools, equations, and experiments into a comprehensive understanding of HPC software development in general. Such understanding will require enlarging its experimental space, both for statistical reasons and also for the purpose of refining, deepening, and maturing the models and assumptions inherent in the experimental methodologies. It will require clever engineering of test conditions to isolate factors of interest and demonstrate true cause and effect relationships. It will require long-term investment in research, since experiments are difficult to “set up” and take a long time to produce results. Finally, it will require a new generation of researchers who grasp the vital importance of understanding and improving HPC software productivity and are committed to creating a legacy for future generations of HPC systems and their users. It is the hope of the authors? and all who participate in HPCS productivity research that such a vision will come into being.

6. The HPC Challenge Benchmark Suite

As noted earlier, productivity – the main concern of the DARPA HPCS program – depends both on the programming effort and other "set up" activities that precede the running of application codes, and on the sustained, runtime performance of the codes. Approaches to measuring programming effort were reviewed in a prior section of this work. This section discusses the HPC Challenge (HPCC) benchmark suite, a relatively new and still-evolving tool for evaluating the performance of HPC systems on a various types of tasks that form the underpinnings for most HPC applications.

The HPC Challenge[11] benchmark suite was initially developed for the DARPA HPCS program, [89] to provide a set of standardized hardware probes based on commonly occurring computational software kernels. The HPCS program involves a fundamental reassessment of how we define and measure performance, programmability, portability, robustness and, ultimately, productivity across the entire high-end domain. Consequently, the HPCC suite aimed both to give conceptual expression to the underlying computations used in this domain, and to be applicable to a broad spectrum of computational science fields. Clearly, a number of compromises needed to be embodied in the current form of the suite, given such a broad scope of design requirements. HPCC was designed to approximately bound computations of high and low spatial and temporal locality (see Figure 6.1, which gives the conceptual design space for the HPCC component tests). In

[11] This work was supported in part by the DARPA, NSF, and DOE through the DARPA HPCS program under grant FA8750-04-1-0219 and SCI-0527260.

addition, because the HPCC tests consist of simple mathematical operations, HPCC provides a unique opportunity to look at language and parallel programming model issues. As such, the benchmark is designed to serve both the system user and designer communities [90].

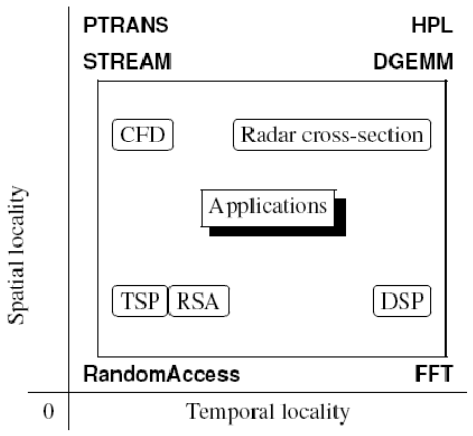


Figure 6.1 The application areas targeted by the HPCS Program are bound by the HPCC tests in the memory access locality space.

Rank	Name	Rmax	HPL	PTRANS	STREAM	FFT	Random Access	Lat.	B/w
1	BG/L	280.6	259.2	4665.9	160	2311	35.47	5.92	0.16
2	BG W	91.3	83.9	171.5	50	1235	21.61	4.70	0.16
3	ASC Purple	75.8	57.9	553.0	44	842	1.03	5.11	3.22
4	Columbia	51.9	46.8	91.3	21	230	0.25	4.23	1.39
9	Red Storm	36.2	33.0	1813.1	44	1118	1.02	7.97	1.15

Table 6.1 All of the top-10 entries of the 27th TOP500 list that have results in the HPCC database.

Figure 6.2 shows a generic memory subsystem and how each level of the hierarchy is tested by the HPCC software, along with the design goals for the future HPCS system (i.e., the projected target performance numbers that are to come out of the wining HPCS vendor designs).

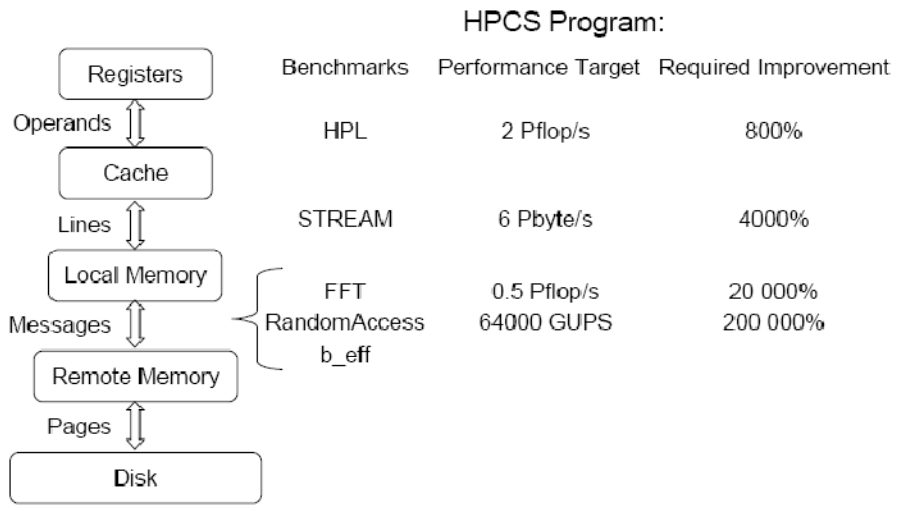


Figure 6.2 HPCS program benchmarks and performance targets.

6.1 The TOP500 Influence

The most commonly known ranking of supercomputer installations around the world is the TOP500 list [91]. It uses the equally well-known LINPACK benchmark [92] as a single figure of merit to rank 500 of the world's most powerful supercomputers. The often-raised question about the relation between the TOP500 list and HPCC can be addressed by recognizing the positive aspects of the former. In particular, the longevity of the TOP500 list gives an unprecedented view of the high-end arena across the turbulent era of Moore's law [93] rule and the emergence of today's prevalent computing paradigms. The predictive power of the TOP500 list is likely to have a lasting influence in the future, as it has had in the past. HPCC extends the TOP500 list's concept of exploiting a commonly used kernel and, in the context of the HPCS goals, incorporates a larger, growing suite of computational kernels. HPCC has already begun to serve as a valuable tool for performance analysis. Table 6.1 shows an example of how the data from the HPCC database can augment the TOP500 results.

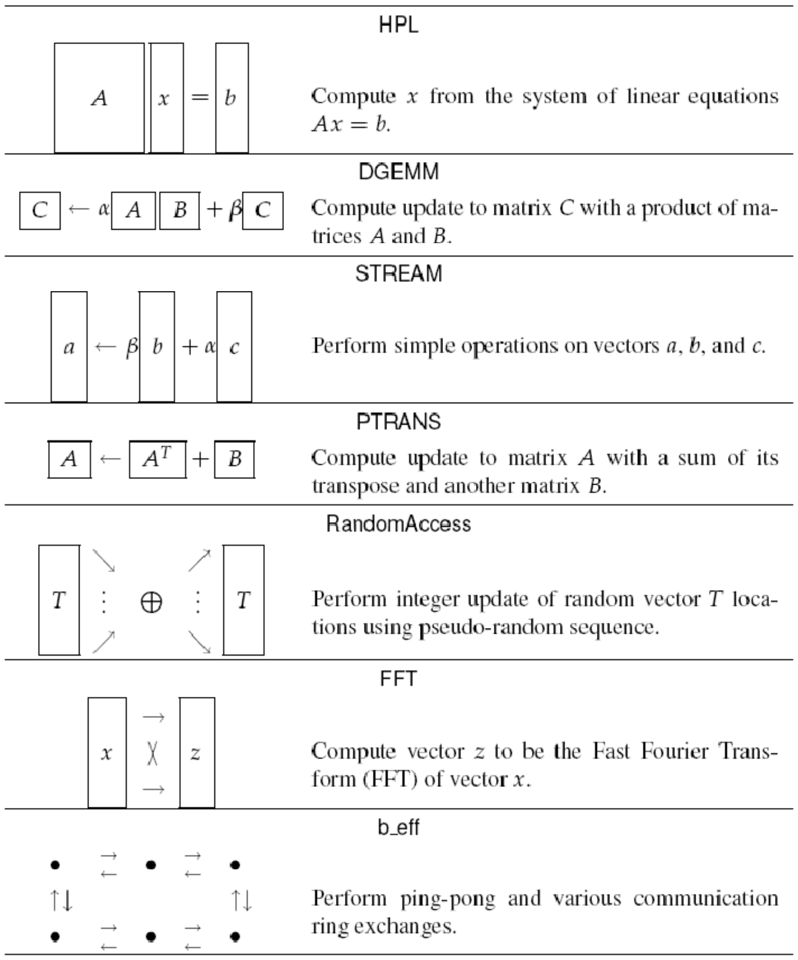
6.2 Short History of the Benchmark

The first reference implementation of the HPCC suite of codes was released to the public in 2003. The first optimized submission came in April 2004 from Cray, using the then-recent X1 installation at Oak Ridge National Lab. Ever since then, Cray has championed the list of optimized HPCC submissions. By the time of the first HPCC birds-of-a-feather session at the Supercomputing conference in 2004 in Pittsburgh, the public database of results already featured major supercomputer makers – a sign that vendors were participating in the new benchmark initiative. At the same time, behind the scenes, the code was also being tried out by government and private institutions for procurement and marketing purposes. A 2005 milestone was the announcement of the HPCC Awards contest. The two complementary categories of the competition emphasized performance and productivity – the same goals as the sponsoring HPCS program. The performance-emphasizing Class 1 award drew the attention of many of the biggest players in the supercomputing industry, which resulted in populating the HPCC database with most of the top10 entries of the TOP500 list (some exceeding their performances reported on the TOP500 -- a tribute to HPCC's continuous results update policy). The contestants competed to achieve the highest raw performance in one of the four tests: HPL, STREAM, RANDA, and FFT. The Class 2 award, by solely focusing on productivity, introduced a subjectivity factor into the judging and also into o the submission criteria, regarding what was appropriate for the contest. As a result, a wide range of solutions were submitted, spanning various programming languages (interpreted and compiled) and paradigms (with explicit and implicit parallelism). The Class 2 contest featured openly available as well as proprietary technologies, some of which were arguably confined to niche markets and some that were widely used. The financial incentives for entering turned out to be all but needless, as the HPCC seemed to have gained enough recognition within the high-end community to elicit entries even without the monetary assistance. (HPCwire provided both press coverage and cash rewards for the four winning contestants in Class 1 and the single winner in Class 2.) At the HPCC's second birds-of-a-feather session during the SC07 conference in Seattle, the former class was dominated by IBM's BlueGene/L at Lawrence Livermore National Lab, while the latter class was split among MTA pragma-decorated C and UPC codes from Cray and IBM, respectively.

6.2.1 The Benchmark Tests' Details

Extensive discussion and various implementations of the HPCC tests are available elsewhere [94, 95, 96]. However, for the sake of completeness, this section provides the most important facts pertaining to the HPCC tests' definitions.

All calculations use *double precision* floating-point numbers as described by the IEEE 754 standard [97], and no mixed precision calculations [98] are allowed. All the tests are designed so that they will run on an arbitrary number of processors (usually denoted as *p*). Figure 6.3 shows a more detailed definition of each of the seven tests included in HPCC. In addition, it is possible to run the tests in one of three testing scenarios to stress various hardware components of the system. The scenarios are shown in Figure 6.4.



HPL

Compute x from the system of linear equations $Ax = b$.

DGEMM

Compute update to matrix C with a product of matrices A and B.

STREAM

Perform simple operations on vectors a, b, and c.

PTRANS

Compute update to matrix A with a sum of its transpose and another matrix B.

RandomAccess

Perform integer update of random vector T locations using pseudo-random sequence.

FFT

Compute vector z to be the Fast Fourier Transform (FFT) of vector x.

b_eff

Perform ping-pong and various communication ring exchanges.

Figure 6.3. Detail description of the HPCC component tests (*A*, *B*, *C* - matrices, *a*, *b*, *c*, *x*, *z* - vectors, α, β - scalars, *T* - array of 64-bit integers).

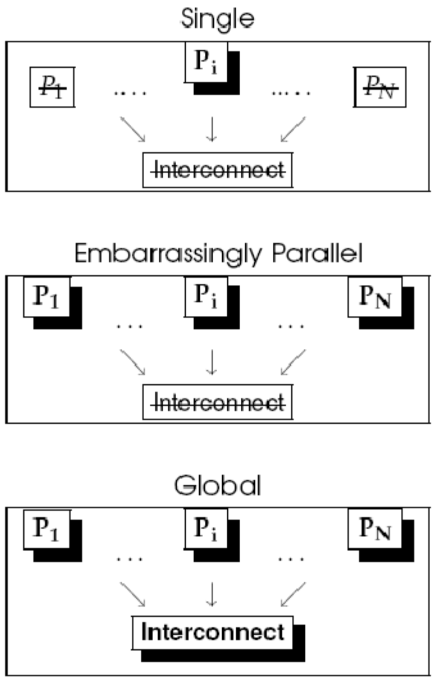


Figure 6.4. Testing scenarios of the HPCC components.

6.2.2 Benchmark Submission Procedures and Results

The reference implementation of the benchmark may be obtained free of charge at the benchmark's web site[12]. The reference implementation should be used for the base run: it is written in a portable subset of ANSI C [99] using a hybrid programming model that mixes OpenMP [100, 101] threading with MPI [102, 103, 104] messaging. The installation of the software requires creating a script file for Unix's `make(1)`utility. The distribution archive comes with script files for many common computer architectures. Usually, a few changes to any of these files will produce the script file for a given platform. The HPCC rules allow only standard system compilers and libraries to be used through their supported and documented interface, and the build procedure should be described at submission time. This ensures repeatability of the results and serves as an educational tool for end users who wish to use a similar build process for their applications.

After a successful compilation, the benchmark is ready to run. However, it is recommended that changes be made to the benchmark's input file that describes the sizes of data to use during the run. The sizes should reflect the available memory on the system and the number of processors available for computations.

There must be one baseline run submitted for each computer system entered in the archive. An optimized run for each computer system may also be submitted. The baseline run should use the reference implementation of HPCC, and in a sense it represents the scenario when an application requires use of legacy code – a code that cannot be changed. The optimized run allows the submitter to perform more aggressive optimizations and use system-specific programming techniques (languages, messaging libraries, etc.), but at the same time still includes the verification process enjoyed by the base run.

All of the submitted results are publicly available after they have been confirmed by email. In addition to the various displays of results and exportable raw data, the HPCC website also offers a kiviat chart display to visually compare systems using multiple performance numbers at once. A sample chart that uses actual HPCC results data is shown in Figure 6.5.

[12] http://icl.cs.utk.edu/hpcc/

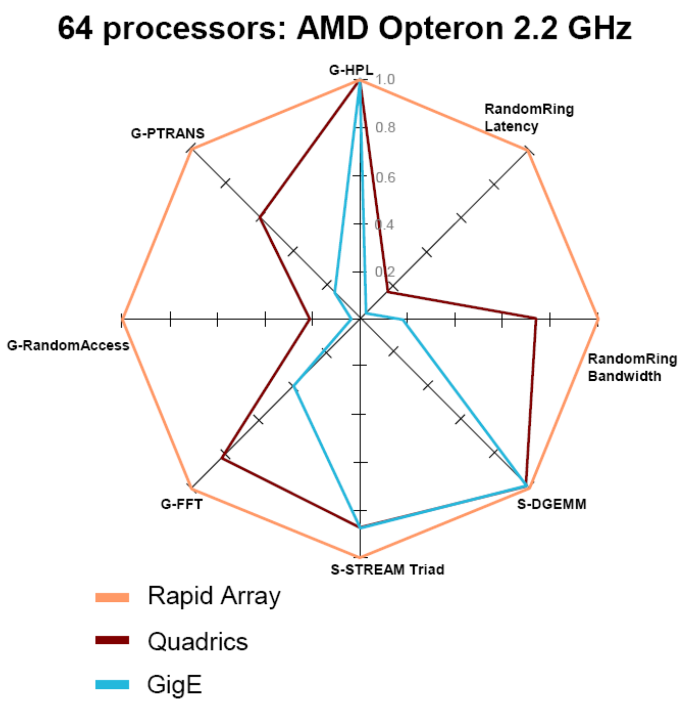


Figure 6.5. Sample Kiviat diagram of results for three different interconnects that connect the same processors.

Figure 6.6 show performance results of some currently operating clusters and supercomputer installations. Most of the results come from the HPCC public database.

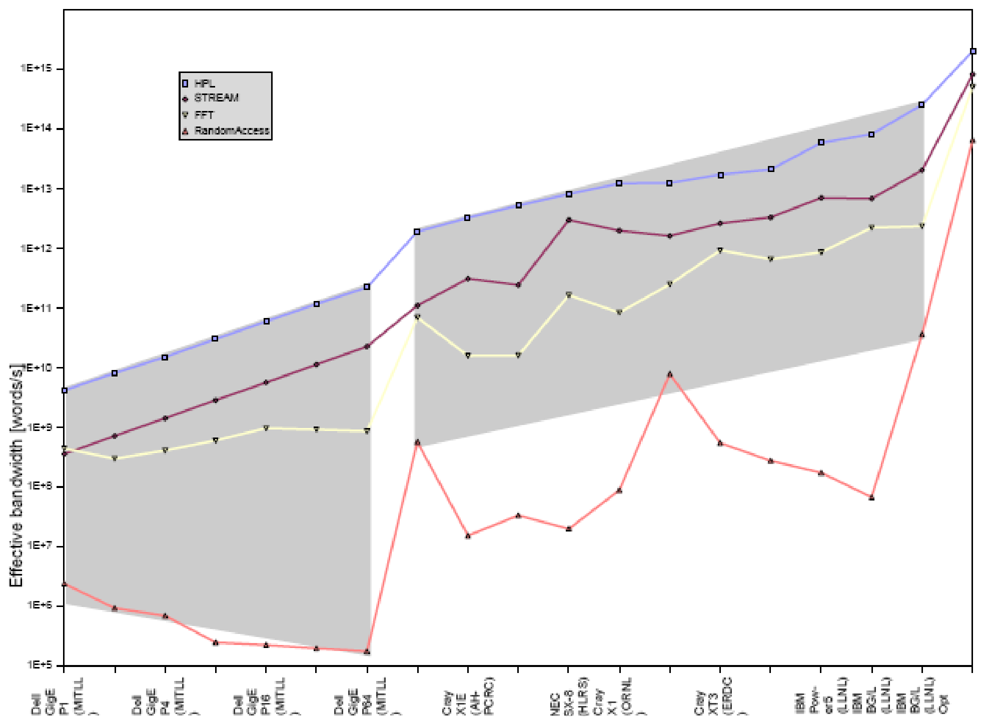


Figure 6.6. Sample interpretation of the HPCC results.

6.2.3 Scalability Considerations

There are a number of issues to be considered for benchmarks such as HPCC that have scalable input data. These benchmarks need to allow for arbitrary sized systems to be properly stressed in the benchmark run. The time to run the entire suite is a major concern for institutions with limited resource allocation budgets. With these considerations in mind, each component of HPCC has been analyzed from the scalability standpoint, and Table 11 shows the major time complexity results. In the following tables, it is assumed that:

- *M* is the total size of memory,
- *m* is the size of the test vector,
- *n* is the size of the test matrix,
- *p* is the number of processors,
- *t* is the time to run the test.

Name	Generation	Computation	Communication	Verification	Per-processor data
HPL	n^2	n^3	n^2	n^2	p^{-1}
DGEMM	n^2	n^3	n^2	**1**	p^{-1}
STREAM	m	m	**1**	m	p^{-1}
PTRANS	n^2	n^2	n^2	n^2	p^{-1}
RandomAccess	m	m	m	m	p^{-1}
FFT	m	$m \log_2 m$	m	$m \log_2 m$	p^{-1}
b_eff	**1**	**1**	p^2	**1**	**1**

Table 6.2. Time complexity formulas for various phases of the HPCC tests (*m* and *n* correspond to the appropriate vector and matrix sizes, *p* is the number of processors.)

Clearly, any complexity formula that grows faster than linearly for any system size raises concerns about the time-scalability issue. The following HPCC tests have had to be looked at with this concern in mind:

- HPL, because it has computational complexity $O(n^3)$.
- DGEMM, because it has computational complexity $O(n^3)$.
- b_eff, because it has communication complexity $O(p^2)$.

The computational complexity of HPL of order $O(n^3)$ may cause excessive running time because the time will grow proportionately to a high power of total memory size:

Equation 1 $t_{HPL} \sim n^3 = (n^2)^{3/2} \sim M^{3/2} = \sqrt{M^3}$

To resolve this problem, we have turned to the past TOP500 data and analyzed the ratio of Rpeak to the number of bytes for the factorized matrix for the first entry on all the lists. It turns out that there are on average 6±3 Gflop/s for each matrix byte. We can thus conclude that the performance rate of HPL remains constant over time ($r_{HPL} \sim M$) which leads to a formula much better than Equation 1:

Equation 2 $t_{HPL} \sim n^3/r_{HPL} \sim \sqrt{M^3}/M = \sqrt{M}$

There seems to be a similar problem with the DGEMM, as it has the same computational complexity as HPL; but fortunately, the *n* in the formula is related to a single process memory size rather than the global one, and thus there is no scaling problem.

The b_eff test has a different type of problem: its communication complexity is $O(p^2)$, which is already prohibitive today as the number of processes of the largest system in the HPCC database is 131072. This complexity comes from the ping-pong component of b_eff that attempts to find the weakest link among all nodes and thus, theoretically, needs to look at all possible process pairs. The problem was remedied in the reference implementation by adapting the runtime of the test to the size of the system tested.

6.3 Conclusions

No single test can accurately compare the performance of any of today's high-end systems, let alone those envisioned by the HPCS program in the future. Thus, the HPCC suite stresses not only the processors, but the memory system and the interconnect. It is a better indicator of how a supercomputing system will perform across a spectrum of real-world applications. Now that the more comprehensive HPCC suite is available, it can be used in preference to comparisons and rankings based on single tests. The real utility of the HPCC benchmarks is that it can describe architectures with a wider range of metrics than just flop/s from HPL. When only HPL performance and the TOP500 list are considered, inexpensive build-your-own clusters appear to be much more cost effective than more sophisticated parallel architectures. But the tests indicate that even a small percentage of random memory accesses in real applications can significantly affect the overall performance of that application on architectures not designed to minimize or hide memory latency. The HPCC tests provide users with additional information to justify policy and purchasing decisions. We expect to expand the HPCC suite (and perhaps remove some existing components) as we learn more about the collection and its fit with evolving architectural trends.

7. Summary: The DARPA HPCS Program

This document reviews the historical context surrounding the birth of the High Productivity Computing Systems (HPCS) program, including DARPA's motivation for launching this long-term high performance computing initiative. It discusses HPCS-related technical innovations, productivity research, and the renewed commitment by key government agencies to advancing leadership computing in support of national security, large science, and space requirements at the start of the 21st century.

To date, the HPCS vision of developing economically viable high productivity computing systems, as originally defined in the HPCS white paper, has been carefully maintained . The vision of economically viable—yet revolutionary—petascale high productivity computing systems led to significant industry and university partnerships early in the program and to a heavier industry focus later in the program. The HPCS strategy has been to encourage the vendors to not simply develop evolutionary systems, but to attempt bold *productivity* improvements, with the government helping to reduce the risks through R&D cost sharing. Productivity, by its very nature, is difficult to assess because its definition depends upon the specifics of the end user mission, applications, team composition, and end use. Based on the productivity definition outlined in this work, specific research results were presented, performed by multi-agency/university HPCS productivity team, that address the challenge of providing some means of predicting, modeling and quantifying the end value "productivity" of complex computing systems to end users. The productivity research to date represents the beginning and the not the end of this challenging research activity.

History will ultimately judge the progress made under HPCS during this period, but will no doubt concede that these years produced renewed public/private support and recognition for the importance of supercomputing and the need for a better path forward. It has become abundantly clear that theoretical ("peak") performance can no longer suffice for measuring computing leadership. The ability to use supercomputing to improve a company's bottom line, enhance national security, or accelerate scientific discovery has emerged as the true standard for technical leadership and national competitiveness. The battle for leadership is far from over, however. Programming large-scale supercomputers has never been easy, and the near-term prospect of systems routinely having 100,000 or more processors has made the programming challenge even more daunting.

A number of agencies, including the DOE Office of Science, National Nuclear Security Agency (NNSA), National Security Agency (NSA), and National Science Foundation (NSF) now have active programs in place to establish and maintain leadership-class computing facilities. These facilities are preparing to meet the challenges of running applications at sustained petaflop speeds (one quadrillion calculations per second) in areas ranging from national security to data analysis and scientific discovery.

The challenge through this decade and beyond is to continue the renewed momentum in high-end computing and to develop new strategies to extend the benefits of this technology to many new users, including the tens of thousands of companies and other organizations that have not moved beyond desktop computers to embrace HPC. Meeting this challenge would not only boost the innovation and competitiveness of these companies and organizations, but in the aggregate would advance the economic standing of the nation.

References

1. R. Badia, G. Rodriguez, and J. Labarta. Deriving analytical models from a limited number of runs. In Parallel Computing: Software Technology, Algorithms, Architectures, and Applications (PARCO 2003), pages 769?776, Dresden, Germany, 2003.

2. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K.Weeratunga. The NAS Parallel Benchmarks. The International Journal of Supercomputer Applications, 5(3):63?73, Fall 1991.

3. L. Carrington, M. Laurenzano, A. Snavely, R. Campbell, and L. Davis. How well can simple metrics predict the performance of real applications? In Proceedings of Supercomputing (SCé05), November 2005.

4. L. Carrington, A. Snavely, N.Wolter, and X. Gao. A performance prediction framework for scientific applications. In Proceedings of the International Conference on Computational Science (ICCS 2003), Melbourne, Australia, June 2003.

5. Department of Defense High Performance Computing Modernization Program. Technology Insertion - 06 (TI-06). http://www.hpcmo.hpc.mil/Htdocs/TI/TI06, May 2005.

6. European Center for Parallelism of Barcelona. Dimemas. http://www.cepba.upc.es/dimemas.

7. X. Gao, M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic traces. In International Symposium on Workload Characterization (ISWC05), September 2005.

8. J. L. Gustafson and R. Todi. Conventional benchmarks as a sample of the performance spectrum. The Journal of Supercomputing, 13(3):321?342, 1999.

9. C. L. Lawson and R. J. Hanson. Solving least squares problems, volume 15 of Classics in applied mathematics. SIAM, Philadelphia, PA, 1995. An unabridged, revised republication of the original work published by Prentice-Hall, Englewood Cliffs, NJ, 1974.

10. P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC challenge benchmark suite. Available at http://www.hpccchallenge.org/pubs/, March 2005.

11. G. Marin and J. Mellor-Crummey. Crossarchitecture performance predictions for scienti[1]c applications using parameterized models. In Proceedings of SIGMETRICS/Performance'04, New York, NY, June 2004.

12. J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Technical Committee on Computer Architecture Newsletter, December 1995.
13. C.L. Mendes and D.A. Reed. Performance stability and prediction. In IEEE /USP International Workshop on High Performance Computing, 1994.

14. C.L. Mendes and D.A. Reed. Integrated compilation and scalability analysis for parallel systems. In IEEE PACT, 1998.

15. R.H. Saavedra and A.J. Smith. Measuring cache and tlb performance and their effect on benchmark run times. In IEEE Transactions on Computers, vol. 44:10 pp. 1223-1235, 1995.

16. R.H. Saavedra and A.J. Smith. Performance characterization of optimizing compilers. In TSE21, vol. 7, pp. 615-628, 1995.

17. R.H. Saavedra and A.J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. In TOCS14, vol. 4, pp. 344-384, 1996.

18. J. Simon and J. Wierum. Accurate performance prediction for massively parallel systems and its applications. In Proceedings of 2nd International Euro-Par Conference, Lyon, France, August 1996.

19. A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In Proceedings of Supercomputing (SC2002), Baltimore, MD, November 2002.

20. SPEC: Standard performance evaluation corporation. http://www.spec.org, 2005.

21. L. Svobodova. Computer system performance measurement and evaluation methods: Analysis and applications. In Elsevier N.Y., 1976.

22. TI-06 benchmarking: rules & operational instructions. department of defense high performance computing modernization program. http://www.hpcmo.hpc.mil/Htdocs/TI/TI06/ti06 benchmark inst.May 2005

23. Cray Inc. Cray XMT Platform, available at http://www.cray.com/products/xmt/index.html

24. Cray Inc., Cray MTA-2 Programmer's Guide, Cray Inc. S-2320-10, 2005.

25. GPGPU, General Purpose computation using GPU hardware, http://www.gpgpu.org/

26. GROMACS, http://www.gromacs.org/

27. International Business Machines Corporation, Cell Broadband Engine Programming Tutorial Version 1.0, 2005.
28. LAMMPS, http://lammps.sandia.gov/

29. NAMD, http://www.ks.uiuc.edu/Research/namd/

30. NIVDIA, http://www.nvidia.com

31. OpenMP specifications, version 2.5, `http://www.openmp.org/drupal/mp-documents/spec25.pdf`.

32. S. R. Alam, et. al., Performance Characterization of Bio-molecular Simulations using Molecular Dynamics, ACM Symposium of Principle and Practices of Parallel Programming, 2006.

33. D. A. Bader, et.al, On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking, IEEE Int. Parallel and Distributed Processing Symp. (IPDPS),

2007.

34. S. Bokhari and J. Sauer, Sequence alignment on the Cray MTA-2, Concurrency and Computation: Practice and Experience (Special issue on High Performance Computational Biology), 16(9):823–39, 2004.

35. [J. Bower, et. al., Scalable Algorithms for Molecular Dynamics. Simulations on Commodity Clusters, ACM/IEEE Supercomputing Conference, 2006.

36. F. Blagojevic, et. al., RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine, IPDPS, 2007.

37. I. Buck, Brook-Data Parallel Computation on Graphics Hardware, Workshop on Parallel Visualization and Graphics, 2003.

38. S. Faulk, et al., Measuring HPC productivity, International Journal of High Performance Computing Applications, 2004. 18(4): p. 459-473.

39. J. Feo, et al., ELDORADO, Conference on Computing Frontiers. Italy: ACM Press, 2005.

40. B. J. Fitch, et.al., Blue Matter: Approaching the Limits of Concurrency for Classical Molecular Dynamics, ACM/IEEE Supercomputing Conference, 2006.

41. B. Flachs, et al., The microarchitecture of the synergistic processor for a cell processor, IEEE Journal of Solid-State Circuits, 41(1):63-70, 2006.

42. A. Funk et. al., Analysis of Parallel Software Development using the Relative Development Time Productivity Metric, CTWatch Quarterly, 2006. 2(4A).

43. Hwa-Joon, S.M. Mueller et al., A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor, IEEE Journal of Solid-State Circuits, 41(4):759-71, 2006.

44. J. A. Kahle, et al., Introduction to the Cell Microprocessor, IBM J. of Research and Development, 49(4/5):589-604, 2005.

45. A. R. Leach, Molecular modeling: principles and applications, 2nd ed: Prentice Hall, 2001.

46. W. Liu, et. al. Bio-Sequence Database Scanning on a GPU, IEEE Int. Workshop on High Performance Computational Biology, 2006.

47. Y. Liu, et. al. GPU Accelerated Smith-Waterman. International Conference on Computational Science, 2006.

48. S. Oliver, et.al., Porting the GROMACS Molecular Dynamics Code to the Cell Processor, Proc. of 8th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-07), 2007.

49. F. Petrini, et. al. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine, IPDPS, 2007.

50. R. M. Ramanthan, Intel Multi-core Processors: Making the move to Quad-core and Beyond, white paper available at http://www.intel.com/technology/architecture/downloads/quad-core-06.pdf

51. Villa, et. al. Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-core Processors, IPDPS, 2007.

52. M. Zelkowitz, et al., Measuring Productivity on High Performance Computers, 11th IEEE International Symposium on Software Metric, 2005.

53. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification. Available from http:// research.sun.com/projects/plrg/.

54. D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002. Instructions for Typesetting Camera-Ready Manuscripts

55. D. Buntinas and W. Gropp. Designing a common communication subsystem. In Beniamino Di Martino, Dieter Kranzlu¨uller, and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume LNCS 3666 of Lecture Notes in Computer Science, pages 156–166. Springer, September 2005. 12th European PVM/MPI User's Group Meeting, Sorrento, Italy.

56. D. Buntinas and W. Gropp. Understanding the requirements imposed by programming model middleware on a common communication subsystem. Technical Report ANL/MCS-TM-284, Argonne National Laboratory, 2005.

57. Chapel: The Cascade high productivity language. http://chapel.cs.washington. edu/.

58. UPC Consortium. UPC language specifications v1.2. Technical report, Lawrence Berkeley National Lab, 2005.

59. Project Fortress code.

60. HPCS Language Project Web Site. http://hpls.lbl.gov/.

61. HPLS. http://hpls.lbl.gov.

62. L.V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, Sep-Oct 1993. ACM Sigplan Notes, Vol. 28, No. 10, pp. 91-108.

63. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. International Journal of Supercomputer Applications, 8(3/4):165–414, 1994.

64. Message Passing Interface Forum. MPI2: A Message Passing Interface standard. International Journal of High Performance Computing Applications, 12(1–2):1–299, 1998.

65. R. Numrich and J. Reid. Co-Array Fortran for parallel programming. In ACM Fortran Forum 17, 2, 1-31., 1998.
66. The X10 programming language. http://www.research.ibm.com/x10.

67. The X10 compiler. http://x10.sf.net.

68. K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. Concurrency: Practice and Experience, 10:825–836, 1998

69. Cyberinfrastructure Technology Watch (CTWatch) Quarterly, http://www.ctwatch.org, Volume 2, Number 4A, November 2006: High Productivity Computing Systems and the Path Towards Usable Petascale Computing, Part A: User Productivity Challenges, Jeremy Kepner, guest editor.

70. Cyberinfrastructure Technology Watch (CTWatch) Quarterly, http://www.ctwatch.org, Volume 2, Number 4B, November 2006: High Productivity Computing Systems and the Path Towards Usable Petascale Computing, Part B: System Productivity Technologies, Jeremy Kepner, guest editor.

71. J. Kepner, Introduction: High Productivity Computing Systems and the Path Towards Usable Petascale Computing, MIT Lincoln Laboratory, in [69] p. 1.

72. Making the Business Case for High Performance Computing: A Benefit-Cost Analysis Methodology Suzy Tichenor (Council on Competitiveness) and Albert Reuther (MIT Lincoln Laboratory), in [69], pp. 2-8.

73. N. Wolter, M. O. McCraacken, A. Snavely, L. Hochstein, T. Nakamura and V. Basili, What's Working in HPC: Investigating HPC User Behavior and Productivity, in [69] pp. 9-17

74. P. Luszczek, J. Dongarra, and J. Kepner, Design and Implementation of the HPC Challenge Benchmark Suite, in [69] pp. 18-23.

75. L. Hochstein , T. Nakamura, V. R. Basili, S. Asgari, M. V. Zelkowitz, J. K. Hollingsworth, F. Shull, J. Carver, M. Voelp, N. Zazworka, and P. Johnson, Experiments to Understand HPC Time to Development, in [69] pp. 24-32.

76. J. Carver, L. M. Hochstein, R. P. Kendall, T. Nakamura, M. V. Zelkowitz, V. R. Basili, and D. E. Post, Observations about Software Development for High End Computing, in [69] pp. 33-38.

77. S. Tichenor, Application Software for High Performance Computers: A Soft Spot for U.S. Business Competitiveness, in [69] pp. 39-45.

78. A. Funk, V. Basili, L. Hochstein, and J. Kepner, Analysis of Parallel Software Development using the Relative Development Time Productivity Metric, in [69] pp. 46-51.

79. S. Squires, M. L. Van de Vanter and L. G. Votta, Software Productivity Research In High Performance Computing, in [69] pp. 52-61.

80. D. Murphy, T. Nash, L. Votta and J. Kepner, A System-wide Productivity Figure of Merit, in [70] pp. 1-9.

81. E. Stromaier, Performance Complexity: An Execution Time Metric to Characterize the Transparency and Complexity of Performance, in [70] pp. 10-18.

82. A. Funk, J. R. Gilbert, D. Mizell and V. Shah, Modelling Programmer Workflows with Timed Markov Models, in [70] pp. 19-26.

83. P. C. Diniz and T. Krishna, A Compiler-guided Instrumentation for Application Behavior Understanding, in [70] pp. 27-34.

84. S. R. Alam, N. Bhatia and J. S. Vetter, Symbolic Performance Modeling of HPCS Applications, in [70] pp. 35-40.

85. D. A. Bader, K. Madduri, J. R. Gilbert, V. Shah, j. Kepner, T. Meuse, and A. Krishnamurthy, Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems, in [70] pp. 41-51.

86. P. C. Diniz and J. Abramson, SLOPE - A Compiler Approach to Performance Prediction and Performance Sensitivity Analysis for Scientific Codes, in [70] pp. 52-48.

87. T.-Y. Chen, M. Gunn, B. Simon, L. Carrington, and A. Snavely, Metrics for Ranking the Performance of Supercomputers, in [70] pp. 46-67.

88. D. E. Post and R. P. Kendell, Large-Scale Computational Scientific and Engineering Project Development and Production Workflows, in [70] pp. 68-76 in .

89. J. Kepner. HPC productivity: An overarching view. International Journal of High Performance Computing Applications, 18(4), November 2004.

90. W. Kahan. The baleful effect of computer benchmarks upon applied mathematics, physics and chemistry. The John von Neumann Lecture at the 45th Annual Meeting of SIAM, Stanford University, 1997.

91. H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon. TOP500 Supercomputer Sites, 28th edition, November 2006. (The report can be downloaded from http://www.netlib.org/benchmark/top500.html).

92. J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience, 15:1–18, 2003.

93. G. E. Moore. Cramming more components onto integrated circuits. Electronics, 38(8), April 19 1965.

94. J. Dongarra and P. Luszczek. Introduction to the HPC Challenge benchmark suite. Technical Report UT-CS-05-544, University of Tennessee, 2005.

95. P. Luszczek and J. Dongarra. High performance development for high end computing with Python Language Wrapper (PLW). International Journal of High Perfomance Computing

Applications, 2006. Accepted to Special Issue on High Productivity Languages and Models.

96. N. Travinin and J. Kepner. pMatlab parallel Matlab library. International Journal of High Perfomance Computing Applications, 2006. Submitted to Special Issue on High Productivity Languages and Models.

97. ANSI/IEEE Standard 754-1985. Standard for binary floating point arithmetic. Technical report, Institute of Electrical and Electronics Engineers, 1985.

98. J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy. In Proceedings of SC06, Tampa, Florida, November 11-17 2006. See http://icl.cs.utk.edu/iter-ref.

99. B. W. Kernighan and D. M. Ritchie. The C Programming Language. Prentice-Hall, Upper Saddle River, New Jersey, 1978.

100. OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org/

101. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. Parallel Programming in OpenMP. Morgan Kaufmann Publishers, 2001.

102. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. The International Journal of Supercomputer Applications and High Performance Computing, 8, 1994.

103. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1), 1995. Available at: http://www.mpi-forum.org/.

104. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 18 July 1997. Available at http://www.mpi-forum.org/docs/mpi-20.ps.