TECHNICAL REPORT

Productive Petascale Computing: Requirements, Hardware, and Software

Michael L. Van De Vanter, Alan Wood,
Christopher Vick, Stuart Faulk,
Susan Squires, and Lawrence G. Votta, Jr.

Sun microsystems

> Sun Microsystems Laboratories

Productive Petascale Computing: Requirements, Hardware, and Software

Michael L. Van De Vanter, Alan Wood, Christopher Vick, Sun Microsystems, Menlo Park, California

Stuart Faulk, University of Oregon, Dept. of Computer and Information Science, Corvallis, Oregon

Susan Squires, Technology for Independent Living Centre, Trinity College Dublin, Ireland

Lawrence G. Votta, Jr., Brincos, Inc., Sammamish, Washington

SMLI TR-2009-183 July 2009

Abstract:

Supercomputer designers traditionally focus on low-level hardware performance criteria such as CPU cycle speed, disk bandwidth, and memory latency. The High-Performance Computing (HPC) community has more recently begun to realize that escalating hardware performance is, by itself, contributing less and less to real productivity—the ability to develop and deploy high-performance supercomputer applications at acceptable time and cost.

The Defense Advanced Research Projects Agency (DARPA) High Productivity Computing Systems (HPCS) initiative challenged industry vendors to design a new generation of supercomputers that would deliver a 10x improvement in this newly acknowledged but poorly understood domain of real productivity. Sun Microsystems, choosing to abandon customary evolutionary approaches, responded with two revolutionary decisions. The first was to investigate the nature of supercomputer productivity in the full context of use, which includes people, organizations, goals, practices, and skills as well as processors, disks, memory, and software. The second decision was to rethink completely the design of supercomputing systems, informed by productivity-based requirements and driven by recent technological breakthroughs. Crucial to the implementation of these decisions was the establishment of multidisciplinary, closely collaborating teams that conducted research

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
michael.vandevanter@sun.com
alan.wood@sun.com
christopher.vick@sun.com
faulk@cs.uoregon.edu
susan.squires@acelere.net
votta@alum.mit.edu

into productivity and developed the many closely intertwined design decisions needed to meet DARPA's challenge

Among the most significant results from Sun's productivity research was a detailed diagnosis of software development as the dominant barrier to productivity improvements in the HPC community. The level of expertise required, combined with the amount of effort needed to develop conventional HPC codes, has already created a crisis of productivity. Even worse, there is no path forward within the existing paradigm that will significantly increase productivity as hardware systems scale up. The same issues also prevent HPC from "scaling out" to a broader class of applications. This diagnosis led to design requirements that address specific issues behind the expertise and effort bottlenecks.

Sun's design teams explored complex, system-wide tradeoffs needed to meet these requirements in all aspects of the design, including reliability, performance, programmability, and ease of administration. These tradeoffs drew on technological advances in massive chip multithreading, extremely high-performance interconnects, resource virtualization, and programming language design. The outcome was the design for a machine to operate at petascale, with extremely high reliability and a greatly simplified programming model. Although this design supports existing codes and software technologies—crucial requirements—it also anticipates that the greatest productivity breakthroughs will follow from dramatic changes in how HPC codes are developed, changes that require a system of the type designed by Sun's HPCS team.

Table of Contents

List of Figures

List of Tables

1 Introduction

Supercomputing seeks solutions to computational problems whose scale, in both complexity and size, stretches the limits of our technology. Typical supercomputer applications[1] include weather modeling, crash simulation, hydrodynamic modeling, and encryption [12]. Requirements for new supercomputers traditionally focus on hardware performance issues: CPU cycle speed, storage I/O, and memory latency.

The High-Performance Computing (HPC) community has begun to realize, however, that evolutionary hardware improvement, although necessary, is not enough to meet the community's goals. Hardware improvements are now delivering less gain in *real* productivity: *the ability to develop and deploy high-performance supercomputer applications at acceptable time and cost* [30]. Equally challenging is an apparent inability to *scale out* to a wider class of problems for which supercomputing would be valuable. The Defense Advanced Research Projects Agency (DARPA) High Productivity Computing Systems (HPCS) program addresses these problems by funding research and development of more cost-effective supercomputers. In HPCS Phase II, DARPA challenged Sun and other vendors[2] to develop designs for revolutionary petascale computing systems that will not only compute significantly faster than the current generation, but will also deliver at least *10 times more productivity*.

HPCS program goals acknowledge that productivity is poorly understood, but suggest that it is some combination of factors in the areas of performance, robustness, programmability, and portability.[3,4] This characterization demands an extraordinary expansion of the design space for computing systems:

- Programmability, and thus productivity in general, is critically dependent on the context of a system's use, a context that includes challenging issues concerning people, skills, practices, and organizations.
- Programmability and portability are acknowledged to be *whole system properties*; achieving them requires systems engineering of the sort traditionally applied only to performance and robustness.
- Engineering tradeoffs can be made among at least four system properties with requirements that sometimes conflict. There is some precedent for such explicit tradeoffs between performance and robustness, but much less so among the other factors.

It is equally challenging to develop design metrics for real-world productivity. The HPC community traditionally uses machine utilization as a proxy for productivity, where utilization

[1] The HPC community refers to software written to perform scientific computations as *codes*; in this paper, they are referred to interchangeably as *HPC applications* or simply *applications*.

[2] HPCS Phase II vendors were Sun, Cray and IBM.

[3] Considerable expense in the lifetime of HPC applications derives from platform-specific optimizations that confound porting to new platforms.

[4] Sun added *administration* to DARPA's list of productivity factors – see Section 3.9.

denotes that portion of the potentially available floating point operations consumed by an application during execution. This has the advantage of being easily measured and the disadvantage of addressing only some aspects of HPC's cost. An even more severe disadvantage is that it completely ignores the value side of any productivity equation. Utilization as a proxy for productivity is useful only in environments where almost nothing changes. This renders utilization useless in the face of the extraordinary improvement sought by DARPA, through which almost everything changes.

Sun responded to the HPCS challenge by committing to a complete rethinking of supercomputer design, attending not only to hardware performance but also to requirements derived from the best possible understanding of real productivity. The project began with the guiding aphorism:

Productivity must be built in; it cannot be added on.

Two general decisions guided this work:

1. *Broaden the scope.* In addition to traditional hardware and software concerns, decisions were based on human and organizational issues concerning software development, system administration, and the scientists for whom the computer is intended. The team drew on expertise in hardware and software architecture, computational science, software engineering, programming languages and tools, physics, and cultural anthropology. The approach was empirical, driven by data on a wide variety of issues, many of which are messy, historically ignored, and subject to unstated assumptions.
2. *Identify bottlenecks*. Opportunities for dramatic change in complex systems, including human organizations, are best prioritized in terms of constraints that limit progress toward goals [16]. These constraints often manifest most clearly at system boundaries. The team's research focused on identifying the most significant bottlenecks that inhibit system-wide productivity. For example, analysis revealed a crucial programming bottleneck created by the extraordinary amount of both effort and expertise required to develop and maintain HPC applications. This is caused by the failure of current systems to isolate programmers from machine details (for example, memory models and failure modes). Current systems also do not permit applications to be expressed in ways that are intelligible to the domain scientists for whom the computations are performed. An example of an administrative bottleneck is lost availability of machine resources during switchover between capability and capacity modes of operation.

Three additional decisions guided Sun's approach, informed by productivity-based requirements and supported by recent technological breakthroughs:

3. *Focus on whole system properties*. Experts in performance and robustness understand that these are emergent properties of complex systems, requiring design skills that span every aspect from hardware to application software. The team observed that the less-studied areas of programmability, portability, and administration also have this nature.
4. *Rethink system layers*. Decomposing systems into layers with distinct concerns is one of the most powerful intellectual tools available for constructing complex systems. Traditional strategies for redesigning system boundaries include moving functionality down, aligning interfaces with more appropriate abstractions, distributing new functionalities across layers, and making global tradeoffs in the name of overall cost and complexity. The extended scope

of this design considers people and organizations as parts of the system, for example, as additional layers in a system stack, inviting analogous tradeoffs—for example, between human and machine effort.

5. *Leverage new technologies*. Emerging technologies, such as those in the areas of chip interconnect and resource virtualization, create new opportunities to rethink established designs around known system bottlenecks.

This report describes *Hero*, the revolutionary petascale supercomputer, the design of which arose from this work. It also describes Sun's interdisciplinary, highly collaborative design process, without which the goals could not have been achieved. Sun's HPCS productivity team, of which the authors were members, played several roles in that process. The team:

- Conducted research as part of the HPCS extended productivity program.
- Gathered empirical data from the HPCS mission partners, the organizations for which the HPCS supercomputers are intended.
- Applied research results to development of system hardware and software requirements.
- Worked closely with design teams to create the alignment needed to meet the productivity challenge.

Section 2 describes how the team constructed an interdisciplinary perspective on the wide range of significant issues faced by the supercomputing community as systems and problems continue to increase in scale and the requirement to scale out to a wider audience becomes more important. It includes an informal, operational definition of productivity that is consistent with a more precise mathematical model developed during the HPCS program [40], as well as a discussion of the role of quantitative metrics.

Section 3 describes productivity bottlenecks identified by this research and reviews the multiple methodologies used to understand them, including observational experiments, case studies, and literature reviews [54].

The productivity research described in Sections 2 and 3 informed the platform development strategy described in Section 4. Drawing on well-understood principles from other computational domains, this development strategy relies on the application of *abstraction* and *automation* to a wide range of productivity-related issues. Application of this development strategy leads to the productivity requirements in Section 5. These requirements touch on the system memory model, application portability, tools, libraries, programming languages, resource monitoring and management, checkpointing, failure recovery, and many others.

Section 6 describes the transition from productivity research and analysis to a design that treats productivity as a property of the whole system. It introduces an interdisciplinary, collaborative process for making high-level design decisions, and notes the interplay among those decisions in rethinking system layers to meet productivity-driven requirements.

Section 7 provides an overview of the Hero system. Each of the subsequent five sections describes a Hero system layer and how it interacts with other system layers to support productivity as a whole system property. These sections also introduce the emerging technologies that enabled the team to rethink system design:

- *Compute nodes* (Section 8): reliance on massive chip multithreading (CMT)

- *System Interconnect* (Section 9): proximity communication and silicon photonics
- *Execution model* (Section 10): single address space memory
- *System software* (Section 11): layers, virtualization, system checkpointing, file and network I/O, and administrative support
- *Development support* (Section 12): languages, tools, and skills

Section 13 revisits the theme, discussed briefly in Section 6, of interdependent design decisions that must address multiple requirements and have implications across many system layers. As an illustrative example, the Hero memory model is chosen for more detailed discussion. Finally, Section 14 re-emphasizes key lessons from Sun's productivity research and Hero system design.

Sections 15-19 are appendices that include acknowledgements, references, a glossary, and documentation associated with the HPCS Phase II program. Section 17 contains a detailed example of the difference between programming in highly productive and standard HPC languages.

2 Characterizing productivity in High-Performance Computing (HPC)

The productivity team began with the definition of real productivity set forth in the HPCS program goals: *the ability to develop and deploy high-performance supercomputer applications at acceptable time and cost* [30]. The range of issues implicit in this precise but general definition makes it difficult to create a common understanding of the domain within which design work can proceed.

This section describes several vantage points from which to begin developing an operational understanding of HPC productivity, starting with characteristics that set it apart from other areas of computation (Section 2.1). It considers traditional layered models used to manage the complexity of computer systems, observing that they exclude many significant productivity-related issues. Drawing on two additional perspectives, software engineering and anthropology, the layered model is extended to describe the whole supercomputer system: hardware, software and the full context of use (Section 2.2). Design metrics depend on quantitative models, for which the utilization-based approach customary in the HPC community turns out to be inadequate (Section 2.3). Finally, none of these models adequately address the desire to scale out HPC to a wider range of practitioners and problems (Section 2.4).

These models represent a starting point for the research data and analysis presented in Section 3 and for the design decisions described in subsequent sections.

2.1 Unique characteristics of HPC

Although software productivity has been studied in other computing domains for years (for example, Boehm's software engineering economics [5]), these results have proven difficult to transfer to HPC applications. There are certainly attributes in common, but there are also ways in which the context of HPC application development differs significantly from general computation.

Capability trumps cost-effectiveness: By definition, supercomputing takes place at the frontier of what is possible in hardware scale and performance, well beyond the cost-effective "sweet spots" available to consumers of commodity computing. Time to solution is often more important than cost.

Software requirements are not well understood: HPC applications are often undertaken to exploit new scientific insights or newly developed numerical and computational techniques, but the software's actual design requirements are created by trial and error [50]: this puts HPC development in the special software development category of "exploratory programming".

Parallel programming is essential: Applying the hardware resources of a supercomputer to large problems demands that applications execute with as much parallel computation as possible.

Application life cycles are long: Figure 1 describes a typical HPC application life cycle; it spans up to 35 years, well beyond the lifetime of applications in other domains. Initial deployment often follows three to five years of development, and maturity may not be reached until years beyond that. Steady-state production support (known as maintenance and evolution in the software engineering community) continues for decades [50].

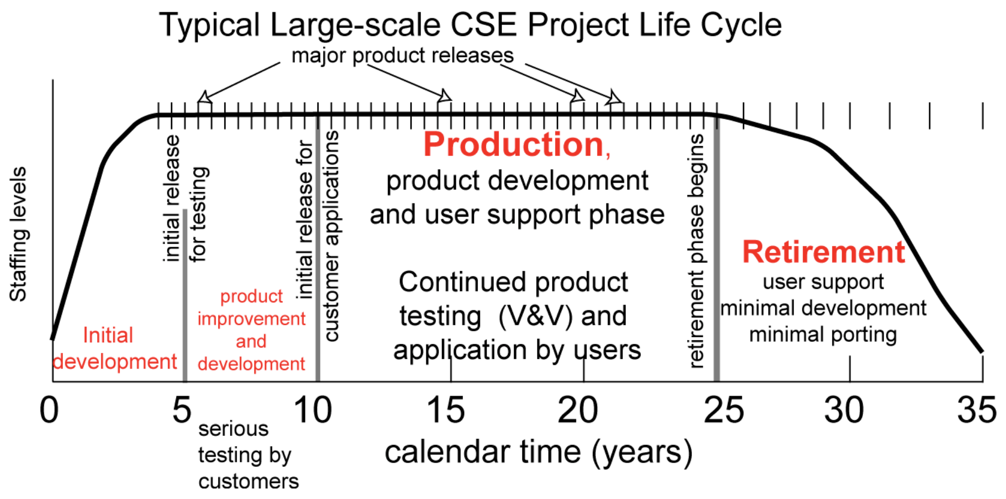


Figure 1: Typical large-scale computational science and engineering (CSE) project life cycle[5]

Ports are frequent: An application is likely to need porting every three to four years during its life cycle, usually at least once during initial development, often at the cost of significant code modification. Because FORTRAN 77 is the only language universally supported with high-quality compilers, it is still considered the safest choice at the outset of new projects, even though other languages (including Fortran 90/95) may shorten initial development.

Validation and verification are expensive: Correctness constraints on complex scientific calculations are stringent [7] [49] [53], but the main approach to testing relies on users, who may also be developers.

Mainstream computing technologies are considered irrelevant: Advances in mainstream software engineering technologies have taken place outside the extraordinary constraints and priorities of HPC. Very little supercomputing is done by people with backgrounds in computer science, because they lack the training, experience, and attitude expected by the HPC community.

2.2 Design models and strategies

HPCS program goals explicitly call out components of productivity (especially portability and programmability) that are sensitive to issues such as software product life cycles, technology replacement cycles, skills, experience, and many more. In other words, the value of a supercomputer depends not only on the hardware but also on the *context of its use*. This is a broad scope for study. An immediate challenge was to derive a framework for relating research results and system design choices, complicated by the breadth of issues under study and the interdisciplinary team's diverse viewpoints.

This section describes an informal model of a whole HPC system that was developed to meet this need. The model begins with the layered system models ubiquitous in computer systems (Section

[5] Figure reproduced with permission from "Large-Scale Computational Scientific and Engineering Code Development and Production Workflows" [50].

2.2.1); these are powerful intellectual tools, but they are too narrowly focused for this study's purposes. The team drew on models from two additional disciplines, software engineering (Section 2.2.2) and anthropology (Section 2.2.3), leading to a synthesized model that captures the notion of a whole supercomputer system (Section 2.2.4). Design strategies familiar to traditional computer system design can be applied within this extended model, after the appropriate data are gathered.

Hero's expanded set of productivity-driven requirements fundamentally changes the nature of computer system design. Issues that might once have been considered simple constraints become opportunities for tradeoffs. These may involve not only hardware and software layers, but also legacy applications, people, and organizations.

2.2.1 A computer system design perspective

Asking a computer system designer for a simple description of a complex system often yields a diagram such as the one appearing in Figure 2. Layered models represent a powerful intellectual tool for managing complexity, and they can be applied at many levels of granularity. For example, Dijkstra's seminal 1968 paper on operating system design described the advantages of a layered approach to system software and the useful "separation of concerns" it permits [13].

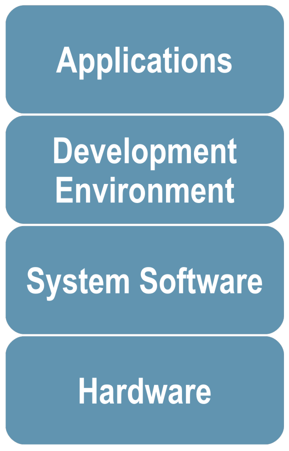


Figure 2: Simple layered model of a computer system

As a design strategy, the layered model presents two important implications. First, it casts system design as an exercise in defining boundaries between layers, also called interfaces in some contexts. A boundary serves to divide the work of designing and building a system, and it defines what is to be done in each layer. From the perspective of each layer's design:

- The layer below offers resources with which to carry out the work of the layer.
- The layer above is a client or consumer of the results created by the layer.

Proper layering creates boundaries that are both maximally useful to the layer above and most easily implemented by resources available in the layer below. Achieving a balance between these two goals can require complex tradeoffs in the definition of the boundaries—tradeoffs that change as both technology and goals evolve. For example, increases in hardware capability have permitted a great downward migration of functionality, and two historic revolutions in computer system design, the virtualization of time (time sharing) and storage (virtual memory), eventually led to redesign at every level.

A second implication of such a diagram is that it defines the scope of the system designer's concerns:

- A designer ignores (or, more accurately, treats as a fixed constraint) anything below the lowest layer in the model, for example the fabrication of processor chips.
- A designer ignores (or, more accurately, treats as a fixed requirement) anything above the highest layer in the model, for example, the skills of application programmers or what tools they use.

This traditional layered model of computer systems, although essential, fails to encompass the range of issues raised by productivity-based requirements. This model was extended by drawing on perspectives from other disciplines, starting with software engineering.

2.2.2 A software engineering perspective

The HPC Productivity Research community proposed describing application development activities as workflows within a single cycle of the sort summarized in Figure 1 on page 12. Although HPC life cycles are extraordinarily long, a software engineering analysis reveals that focusing only on a single application workflow fails to account for the complete context of HPC software development. Figure 3 presents a more complete model, in which the single application workflow is represented in the context of surrounding workflows.

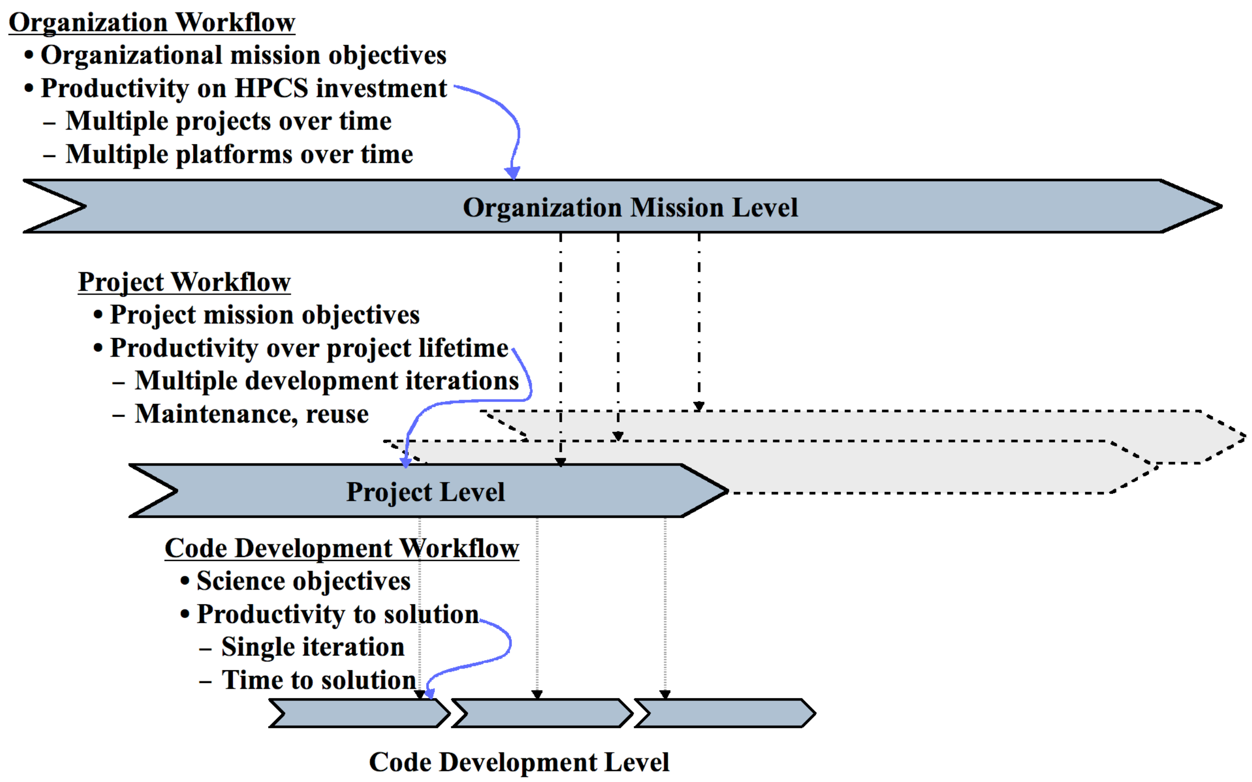

Organization Workflow
• Organizational mission objectives
• Productivity on HPCS investment
– Multiple projects over time
– Multiple platforms over time
Organization Mission Level
Project Workflow
• Project mission objectives
• Productivity over project lifetime
– Multiple development iterations
– Maintenance, reuse
Project Level
Code Development Workflow
• Science objectives
• Productivity to solution
– Single iteration
– Time to solution
Code Development Level

Figure 3: Software engineering perspective on workflows

This model derives from research into general software development, but it applies equally well in environments such as those targeted by the HPCS program. Objectives and resources differ from one level to the next, which means that any notion of productivity must also vary. The three workflows in this model include:

- *Code Development*: A typically small team develops an HPC application with a well-articulated scientific objective, for example modeling thermal dispersion under certain circumstances; such a team is focused on a timely solution.
- *Project*: A project often has a larger goal, for example studying the thermal stability of nuclear weapons, and takes a longer view than a single application. For example, development of a modern replacement for an important application might begin midway through the life cycle of its predecessor. Important issues at this level include strategic reuse of existing code, and decisions about new technology adoption.[6]
- *Organization*: Human organizations, such as government-sponsored laboratories, have missions that are both longer-term and broader than any constituent project, for example strategic stewardship of the nuclear arsenal. Important issues at this level include

[6] Significant decisions to adopt new technologies are almost always taken only at the beginning of new application development, although experimentation may be underway in parallel (Section 3.10).

collective cost effectiveness, the project portfolio in the face of evolving mission objectives, and securing funding.

Many software engineering productivity problems and opportunities lie outside the scope of a single code development cycle, but within a single cycle for a project or organization. In practice, programming productivity is often affected by past decisions such as tool investments, reusable code development, and hiring policies. Problems of this sort appeared during the team's research on the topic of specialized programming tools (Section 3.8). A software engineering perspective enabled the team to understand that the root cause of an apparent failure in the HPC community was misalignment across levels (Section 5.1.5).

As with computer systems, tradeoffs among software engineering levels (layers) can have significant impact. For example, programmer training might take place on the job (code development) or through a wider investment in education (project or organization). Investment in extensible and reusable software application architectures, such as those explored in the "product line" approach [3], is counterproductive for a development team, but it might be vital to an organization's survival.

Finally, the team observed an interesting alignment: the lowest level of the software engineering model (code development) corresponded to the highest level in the computer system model characterized in Figure 2. From the system designer's perspective, application development is out of scope (a fixed requirement). From the software engineering perspective, computer system characteristics are out of scope (a fixed constraint). There is more discussion of this observation following mention of yet another perspective.

2.2.3 An anthropological perspective

The interdisciplinary team also included a cultural anthropologist, skilled at the systematic study of *culture*: a complex phenomenon that includes interrelated economic systems, political systems, social organizations, and belief systems.[7] Observed behavior patterns in people and organizations reflect the combined interrelationships of all these systems.

As the team began to collect data from and about HPCS mission partners, it became clear that the development of HPC applications could be understood only in a much larger context. This led to many questions, starting with those immediately facing the individual programmer and including issues that eventually encompassed the mission's fundamental nature:

- What tasks do programmers perform?
- What training and skills do programmers need?
- How are programmers assigned to projects?
- How does a programmer gain access to computing resources?
- How does a programmer make economic tradeoffs with available resources?
- How do careers in HPC programming develop?

[7] Franz Boas, the "father of American anthropology" received his doctorate in physics and is known for applying the scientific method to the study of people and societies.

- How are projects managed?
- How are computing resources allocated to projects?
- How is access to resources managed on a day-to-day (or minute-to-minute) basis?
- What external products (compilers, for example) are essential, and who pays for them?
- How and when are new programming technologies adopted?
- How is the purchase of a supercomputer justified?
- Who pays for a supercomputer and under what circumstances?
- How is a supercomputer vendor chosen, and what does the purchase include?
- How are laboratories funded?

An anthropologist frames such questions in the multiple contexts in which work gets done: for example individual, group, organization, subculture, and culture. A person makes decisions and acts in every one of those contexts simultaneously; each context has its own systematic structure and rules.

HPC, in the context of the mission partners, is visualized in Figure 4 by mapping these anthropological categories to specific organizational structures. An HPC programmer is an individual (with particular skills), but is also a group member (with short- to medium-term goals), member of a lab or other organization (with career goals), member of the scientific programming community and a citizen.

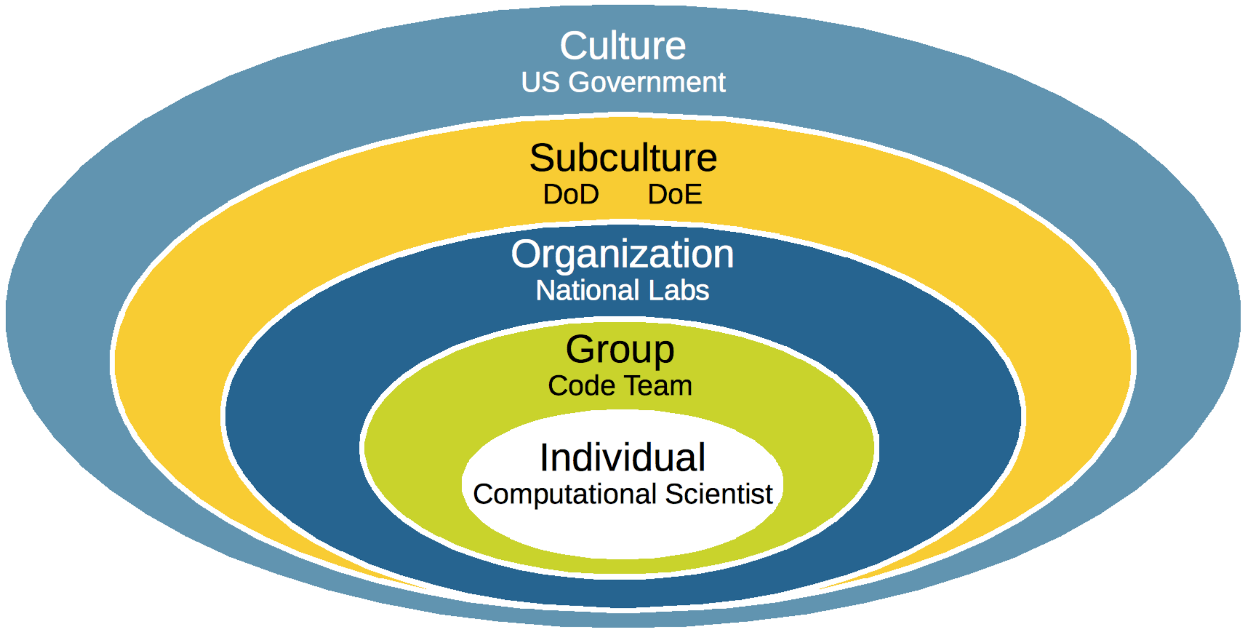


Figure 4: Anthropological perspective on HPCS mission partners

Finally, observe another useful alignment: the layers of the anthropologist's view are similar to the three workflows described in the software engineering perspective depicted in Figure 3. Similar questions arise, and objectives and resources differ from level to level. For example, the decision (and funding) to purchase a supercomputer is an act with national policy considerations and also has a different kind of impact on every level of HPC, down to the individual. The mission's nature has a profound impact on the people who are attracted to the environment, as does the design (and prestige) of the supercomputer around which work gets done.

2.2.4 An interdisciplinary perspective

Drawing on these diverse perspectives, the team extended the conventional but limited layered system model to include explicitly the context of the system's use, as shown in Figure 5.

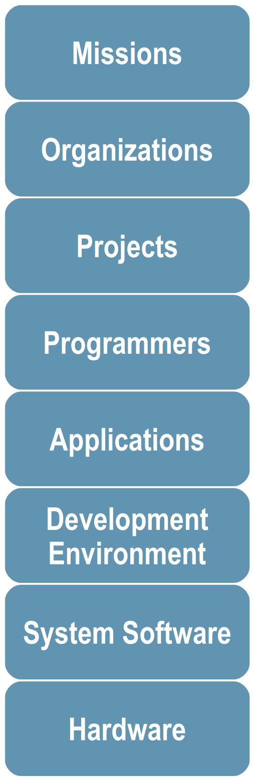


Figure 5: Interdisciplinary view of a whole HPC system

This is the "whole system" whose productivity the team addressed. It is described in terms of layers that embody separate concerns, which interact at boundaries. These layers are subject to design decisions that might redefine boundaries, migrate functionality, and make tradeoffs. Extending the model in this way brings more design parameters into scope for investigation and design. For example, programmer skills and project organization are no longer fixed requirements for software and hardware system design, but opportunities for rethinking and redesign. A machine's architecture is no longer a fixed constraint for developing application software, but an opportunity to make different tradeoffs in the name of overall productivity.

General strategies for design apply, including *abstraction* (making the output, or work result, of each layer as useful as possible to the layer above) and *automation* (arranging for as much of the work as possible at each layer to be subsumed efficiently by a lower layer). Each of the HPCS productivity components (performance, reliability, programmability, portability and administration) is a property of the whole system, and tradeoffs among them must also be considered, such as performance versus programmability.

Each of these layers is an area for research by the productivity team, as well as an area for close collaboration with system designers. Insights about many kinds of bottlenecks, which are obstacles to real productivity, can be drawn into the design process.

A significant challenge is that established boundaries resist change. A particular decomposition can become so embedded into the skills, practices, organizations, careers, and even culture that change becomes very difficult and expensive. This observation applies as much to the organizational context of the mission partners as it does to the organizational context in which computers are typical designed and built.

HPCS productivity goals, however, demand change of this magnitude. Critical bottlenecks are likely to be found in boundaries whose design is obsolete; productivity breakthroughs require significant redesign at many levels.

2.3 Quantitative productivity models

Productivity-based evaluation criteria for design decisions are especially problematic for supercomputers, both in hardware aspects and application development. Commoditized computing affords relatively straightforward tradeoffs, such as when to add processors to a web server farm or how to predict software engineering costs in familiar domains [5]. By definition, supercomputing operates beyond the sweet spot of commodity computing (Section 2.1):

- Hardware is expensive and quickly superseded by better, faster hardware.
- Parallel programming, especially at large scale, is extremely difficult, poorly supported by tools, long-lived, and thus quite expensive.
- Missions can be of critical importance.

This section presents an informal view of quantitative drivers for HPC productivity. It is consistent with a more formal mathematical model developed by Sun's productivity team [40].

2.3.1 The basic productivity equation

Basic economics defines Productivity as a dimensionless ratio of Value (or output) relative to Cost (of producing the output).

$$P = V / C$$

For instance, producing \$100 worth of goods at a cost of \$50 is described as having productivity of two. If the value of goods drops below the cost of producing them, then productivity is less than unity. Furthermore, the value of products, including scientific results, typically varies with time; for example, value may decrease due to competition, or it may increase because of rising demand and short supply, or it may drop precipitously when there are deadlines. Costs may also vary over time.

Supercomputing costs include application programming, porting, administration, hardware purchase, maintenance, space, and (increasingly) energy. Some costs are easy to understand and assign to projects, but others can be difficult to amortize or distribute precisely.

Supercomputing value is much harder to quantify. For instance, fluid simulations are used to model the flow of materials into complex molds to ensure good distribution. Such simulations reduce the need to build and experiment with physical models. Similarly, simulations are used to model airflow, reducing the need for wind tunnel tests. The money thus saved could be taken as the value of the simulations, and manufacturers might provide a realistic estimate. By contrast, it is hopeless to determine the monetary value of research, where scientific publications constitute the main tangible product. Publication counts or citations could be substituted for the value of research, but some papers prove to be more valuable than others, and citation numbers are impossible to predict.

2.3.2 Utilization as a proxy for productivity

The supercomputing community traditionally backs away from these challenges, choosing instead to evaluate system designs with a proxy for productivity that can be measured with some certainty. *Utilization*, expressed as a percentage, measures the portion of the potentially available floating-point operations that a supercomputer application actually consumes during a span of execution. Among a collection of early papers by HPCS program participants, nearly every one characterized productivity gains as improvements to utilization [30].

This approach baffles the outsider; as a simple thought experiment, scenarios can be created where utilization falls in the presence of clear productivity improvements. The superficial problem is that utilization is a cost-based metric that takes no account of the value of computational results. By way of contrast, no owner of a personal computer gives a thought to unused computational cycles.

The community's persistent commitment to this metric derives from the singular nature of HPC in general:

- Utilization can be measured objectively, consistent with the way that physical scientists the traditional consumers of supercomputing, conduct their own research.
- Unlike general computing, supercomputing is dominated by numerical calculations.
- The cost of operating hardware is disproportionately high.
- The *capability*[8] of a supercomputer is the paramount justification for its purchase.
- Time to solution is understood to be more important than cost-effectiveness.

An unstated assumption behind the use of utilization, however, is that comparisons rest at particular operating points in which there is very little change in the environment, called the *context of use*. This assumption holds for evolutionary improvement of the kind often

[8] The HPC community distinguishes between *capability* computing, for which the extraordinary resources of a supercomputer are required, and *capacity* computing, where it is the throughput of significantly smaller computations that matters; supercomputers typically alternate between the two modes of operation.

experienced by the supercomputer community. It does not hold in the face of the magnitude of change needed for a 10x productivity improvement. Such a change will necessarily affect the whole system: people, skills, practices, and organizations, as well as hardware and software.

2.3.3 Toward value-based metrics

These challenges suggest that it is not possible to produce a general quantitative model of productivity: value will vary according to a variety of factors including organizational needs, application domain, and time. However, it is possible to develop quantitative valuation frameworks. This is what the productivity team did during the program. Such frameworks require that stakeholders customize the valuation model with domain-specific parameters that can be difficult to determine.

In the absence of a general characterization in the numerator, initial quantitative focus must be on the denominator, combined with a search for ways to keep the availability and utilization of computer equipment high. In other words, a first step toward improving productivity requires dramatically reducing costs for hardware, energy, computation runtime, software development, and service and maintenance. Even without precise quantitative valuation, considerable insight can be gained by searching for bottlenecks that prevent improvement in these factors (as reported in Section 3).

2.4 Productivity at a wider scale

Even as hardware costs drop and the notion of a small supercomputer becomes practical, supercomputing is still perceived by the larger community as too hard for “the rest of us” [1]. Just as DARPA desires to scale up supercomputers dramatically, others envision broadening the class of problems to which they can be applied in practice [20].

A ratio-based measure, such as the classical productivity formula discussed above, is silent on total output. In fact, scaling out might, in some scenarios, come at the cost of lower utilization, which the traditional HPC community would consider reduced productivity. On the other hand, such a shift would recapitulate the historical shift from mainframe computing to the desktop, migrating to computers whose utilizations would be unacceptable in mainframe shops. Once again, the challenge is to adequately value computer output, a topic finessed by an exclusive focus on utilization.

On the other hand, if the skills needed to use supercomputers were reduced, for instance by requiring less-detailed programming, this would not only improve productivity by speeding up software development and increasing equipment utilization, but also increase total output. Bottlenecks of these sorts are also the topic of research covered in the following section.

3 Productivity bottlenecks in HPC

Within the conceptual framework described in the previous section, the productivity team created a research program that would generate goals and requirements for a new kind of supercomputer: 10x more productive than the status quo. This is an extraordinary requirement, especially for such a poorly understood phenomenon, and it calls for dramatic breakthroughs.

Two observations distinguished the team's approach:

- *Ten times anything is disruptive*. Under the best of circumstances, a 10x change invalidates existing models and assumptions. This is especially true of the incremental, utilization-based quantitative models favored by the HPC community. A disruptive, qualitative change is required, so the research program sought opportunities for such shifts; opportunities for which qualitative insight is as important as quantitative. These opportunities emerged through the identification of *bottlenecks*: aspects of the HPC environment that prevent dramatic breakthroughs in productivity. Bottlenecks may or may not be evident to practitioners, whose daily work embeds them in accepted, non-reflective routines; a well-founded scientific inquiry was required.
- *Context matters*. HPCS criteria imply that a supercomputer's productivity can be understood only in the context of its use. The team's conceptual framework for investigating both computer and context, depicted in Figure 5 on page 18, includes phenomena far beyond the expertise and traditions of the existing HPC community (phenomena that concern people, groups, organizations, and culture,) as well as issues closer to home that are often ignored, such as software engineering and programming language design. The inquiry must be interdisciplinary, and it must be conducted *in situ*.

Section 3 describes the methodology, process, and selected findings from this search for bottlenecks in HPC application development. The bottlenecks can be validated with available data and, although difficult to quantify precisely, can be estimated to have very large effects on productivity.

These studies by the productivity team focus on several of the upper levels depicted in Figure 5, which have historically received little attention. Extensive research by other teams was conducted at many system levels, including processor design, chip interconnect, memory models, and system software. The details of those studies are beyond the scope of this report, although the resulting design decisions, informed by productivity-based requirements, are described in later sections.

Section 3.1 begins with a summary of the interdisciplinary methodology created for these studies. A three-stage adaptation of the scientific method provides a roadmap for the transition from qualitative to quantitative studies and helps avoid common mistakes made by researchers operating outside their fields of expertise, for example naively gathering precise data without an evidence-based theory in which to interpret and validate results.

Section 3.2 describes a set of collaborative case studies conducted by teams from the HPCS vendors, mission partners, and HPCS-funded university projects. Each case study gathered baseline data about a representative HPC application development. Common themes extracted from the data addressed quite explicitly some issues associated with the upper (contextual) layers of the conceptual framework.

Section 3.3 introduces the *expertise hypothesis*, an anecdotal but substantial perception already present in the HPC community that (a) the productivity of HPC application development was collectively stuck at a point of diminishing returns, and (b) that the root cause has something to do with expertise [52]. Starting with this perception, the team reevaluated the collaboratively collected data, gathered more of its own, and created a working hypothesis about the role of expertise in HPC application development.

The HPCS program called for collaborative development of workflows, described in Section 3.4, that would describe the tasks and procedures in HPC application development. The team discovered that the resulting workflows, while useful, could not help refine and validate its hypothesis: they contained more detail than could be validated with the available data, and they were silent on issues that were felt to be significant.

Section 3.5 describes some of the analysis that helped refine and validate the team's understanding of the expertise (skills) and effort (time) that goes into HPC application development. The resulting quantitative data also helped sort out those parts of the process that are *essential* to application goals from those parts that are *accidents* of the technologies being used.[9] The team found that a substantial amount of effort and expertise were not fundamentally essential to the projects' goals. These were cited as bottlenecks where dramatic improvement in productivity might be found. So attention was turned to three areas of software engineering to develop estimates of what improvements might be possible:

- *Software libraries* (Section 3.6), which continue to save effort through code reuse, but offer little promise of a breakthrough in expertise requirements;
- *Programming languages and techniques* (Section 3.7), where several data points suggest that a dramatic increase in productivity is possible; and
- *Development tools* (Section 3.8), where a dramatic breakthrough is possible and the need for change is widely perceived. However, the solution lies beyond the resources or mission of any stakeholders other than funding agencies at the top level (mission) of the extended system model.

In parallel with the productivity team's investigation of programmability and portability requirements, another team gathered data and analyzed the administration requirement. Section 3.9 summarizes their findings.

A critical requirement for any potential breakthrough in productivity is that the HPC community must actually be able to adopt whatever technologies and practices are needed to break the

[9] Fred Brooks draws this distinction in his essay, "No Silver Bullet – Essence and Accident in Software Engineering," and argues that modern software engineering has already achieved dramatic reductions of the *accidental*, leaving the unavoidably *essential* complexity of the problems that were supposed to be solved with software [6]. The team found that Brooks' argument does not yet apply to HPC programming.

bottleneck. Section 3.10 explains how the unique characteristics of HPC environments constrain such adoption, some of which are bottlenecks identified in earlier sections.

Finally, Section 3.11 summarizes findings about bottlenecks in HPC productivity that promise possible breakthroughs. These findings directly drove Hero's design goals and requirements (Section 5) and their impact can be seen in the design work reported in Sections 7 through 13.

3.1 Research methodology

This expansive program of productivity research required a corresponding expansion of methodology conducted by an interdisciplinary team with experience in the social sciences, physics, numerical computation, and software engineering. The team committed to the soundest scientific basis possible, drawing on well-established research methodologies from relevant fields, many of which are unfamiliar within the HPC community (and are sometimes reinvented badly). This section summarizes the team's methodology [54].

Social scientists have developed methods that are both verifiable and reproducible in many contexts, and which can do so with surprisingly few data points in some circumstances. However, the sheer number of methodological options made it crucial that each project begin with clear research goals to identify the most effective combinations of concepts, research designs, information sources, and methods. Table 1 summarizes a three-stage framework, based on the scientific method, that guides these goals; it is grounded in empirical data, validated by multiple approaches (triangulation), and applied to practicing HPC professionals who actually perform the work under study.[10]

Table 1: Research framework

Stage	Goals	Methods
1. Explore and discover	Develop hypotheses	Qualitative
2. Test and define	Test and refine models	Qualitative and quantitative
3. Evaluate and validate	Replicate and validate findings	Quantitative

The stages are necessarily sequential: each provides a foundation for methods in the next. This framework is broadly analogous to realizations of the scientific method used in many disciplines, although the correspondence can be obscured by differences among the phenomena being studied and the methods appropriate to their study.

- *Explore and discover*. At the outset, researchers may not know the appropriate questions to ask or issues to address, let alone have a coherent theory of the phenomena being studied. The first stage is mainly qualitative, open-ended, and designed to produce

[10] It is possible to collect data much less expensively using students rather than professionals. Unfortunately, this presents threats to validity that cannot be assessed in early-stage research on poorly understood phenomena. This danger is acute in a domain where competence is widely understood to require years of experience [52].

insights for hypothesis generation. Such insights can be considered explicit models of the phenomena, analogous to the paradigm in which scientific inquiry is generally undertaken and through which experimental data are interpreted. The most important techniques in this phase permit a researcher to identify and neutralize well-established assumptions, the so-called conventional wisdom of a community under study. This starts the essential process of distinguishing the conventional wisdom that is true from that which is not.

- *Test and define*. Rigor is added in a second stage, using methods that produce additional data surrounding theories generated in the first. This provides feedback on insights, supplies concrete data for model refinement, and leads to deeper understanding of what can be measured and how those measurements can be interpreted. This is the necessary bridge between theory and experimentation.
- *Evaluate and validate*. Finally, more focused and mostly quantitative techniques are precisely applied to collect data, interpret results, and validate outcomes.

An important benefit of this framework is a reliable roadmap for moving from qualitative data collection and analysis, which are appropriate for discovery, to more quantitative data collection and analysis methods, which are appropriate to definitional and evaluative research. In particular, it addresses the challenge of determining exactly when, how, and why to use certain methods to draw out implications of experimental data. Table 2 summarizes many of the possibilities.

A research program without such a roadmap runs a great risk of collecting data using later stage techniques, for example surveys and experiments, without sound hypotheses to inform experimental design; in such cases, conventional wisdom may fill the gap and obscure actual phenomena.

Table 2: Examples of research methods

Stage	Character	Examples of Methods
1. Explore and discover	Open	Ethnography *Case study*[11] *Contextual observation* *Semi-structured interview* Participation Document review Language patterning
2. Test and define	Focused	*Quasi-experimental study* Concept mapping *Structured interview* *Questionnaire* Comparative study Focus group Semiotic analysis[12]
3. Evaluate and validate	Structured	Social network analysis *Survey* Controlled experiment Product testing User experience simulation Human testing Quality measurement

3.2 Collaborative case studies

An important method for first-stage research is the *case study* [66], which draws on many techniques, including document reviews, observation, collection of contextual artifacts, self-reporting, and interviews. In addition to its own research, the team participated in an extensive series of studies at mission partner sites, conducted in collaboration with representatives of national laboratories and other HPCS vendors. The stated objectives of these studies were to:

- Identify critical success factors
- Identify issues that must be addressed by hardware and software vendors to improve productivity of the code development process
- Develop a reference body of case studies for the computational science and engineering community
- Document lessons learned from the analysis and personal team interviews [24] [25] [26] [27] [28] [29] [48] [56]

[11] Italicized methods are those used by the productivity team during this research program.

[12] Semiotics is the study of how symbols and signs are given meaning and understood by people.

These studies collected *quantitative* data using a structured pre-interview survey and *qualitative* data using semi-structured interviews with individual stakeholders and in structured group sessions with code teams. Each study followed a standardized protocol:

- Identify the project and sponsors.
- Negotiate the case study with team and sponsors.
- Complete the pre-interview questionnaire process.
- Analyze the questionnaire and plan onsite interviews.
- Conduct an onsite interview with the team.
- Analyze the onsite interview and integrate with questionnaire.
- Conduct follow-up to resolve unanswered questions.
- Write a report and iterate with code team and sponsor.
- Publish the report.

The collaborators identified common themes [7]:

- Verification and validation are very difficult in this environment.
- A project's primary language typically does not change over time.
- The use of higher-level languages is low.
- Developers prefer the flexibility of UNIX® command lines over integrated development environments (IDEs).
- Externally developed software is a risk.
- Performance competes with other important goals.
- Agile methodologies are better accepted by scientific and engineering code developers than more traditional methodologies.
- Multidisciplinary teams are important to a project's success.
- A project's success or failure depends on keeping customers satisfied (in addition to sponsors).

Baseline data collected by these studies provided valuable first-phase research data, much more than could be reported in published reports. In fact, the choice of what data should be analyzed and reported reflected a certain amount of preconception by participants that, in hindsight, overlooked important contextual data. This is not unusual in early-phase studies.

An outcome from these studies was validation of the extended system model, which adds programmers, projects, organizations, and missions to the essential context of productivity. All of these appeared in themes mentioned in the research summary.

3.3 The expertise hypothesis

Although not reported in findings from the collaborative case studies listed above, the team noticed themes about potential bottlenecks emerging from interview data. For example, an HPC program manager commented on the difficulty of finding people with the right combination of skills to exploit existing machines. In addition, informal evidence showed that some supercomputer sites experience difficulty developing enough highly parallelized applications to keep the machines busy in *capability* mode for substantial amounts of time. This is the mode of operation for which supercomputers are designed and which justifies their expense.

These insights were consistent with early anecdotal evidence from DARPA and HPCS mission partners concerning an "expertise gap" that might lie at the heart of the HPC application crisis [52]. The team began looking more deeply into this question, among others, with additional first-stage research:

- Raw data collected during collaborative case studies was revisited and reinterpreted.
- Additional rapid ethnographic assessments [60] were conducted, drawing on contextual observations and semi-structured interviewing methods at HPC workplaces during subsequent visits to mission partner sites.

Patterns emerged that centered on expertise. In every study, at least one founding team member had been recruited for special knowledge of science; in each case the scientist was not an HPC programmer and may have had little or no experience using Fortran or C++. The scientist was required at the outset to undertake a significant effort outside the domain of science: either learn a programming language or build an effective working relationship with someone who already knew it. In either case, the educational process took considerable time before the individual/pair were judged to be productive. The role of project manager, typically assumed by another person, was to run interference by keeping the sponsor happy and obtaining computing resources. In this context, teams typically require four to six years to produce a working application. Success is commonly attributed to having the right mix of expertise. Teams succeeded only with the appropriate mix of knowledge in four broad areas of expertise:

- Domain science
- Numerical programming
- Optimizing and scaling (parallelization)
- Project management and resource provision

Furthermore, the problems are so demanding that the mere presence of the necessary knowledge does not suffice: there must be overlap. In other words, every member of a successful team must have some expertise in more than one area and be skilled at communication and collaboration.

It appeared that these patterns might explain why HPC expertise is so scarce. A hypothesis was developed postulating that overlapping, domain-specific expertise in at least four different areas is needed to exploit highly parallel machines. Very few people possess these skill sets, and as machines grow in size and complexity, the pool of experts narrows.

Team approaches, as described, are felt to be the best strategy at the moment, but a variety of opinions were expressed about how development might become more productive. A common suggestion was that education for HPC programming should be improved. However, other data shows that it takes many years for programmers—even with the best education anybody could describe—to be considered fully productive. This is an example of conventional wisdom that, upon examination, could not be validated.

The next research step was to craft a more focused study to test the hypothesis and to understand in more detail when and how the various areas of expertise were used; this took place in the research framework's second stage, during which a variety of studies were undertaken to explore several of the upper layers in the extended, interdisciplinary model of supercomputing (Figure 5 on page 18).

3.4 HPCS workflows

A research outcome requested by HPCS program sponsors was the collaborative development of standard *workflows*, intended to identify tasks and procedures carried out by HPC programmers. HPC team members extracted an extensive list of activities from case study data (Section 3.2) and other sources, from which they proposed workflows describing the structure of standard tasks. The workflow in Figure 6 is one example; it is intended to characterize the development of large complex applications and highlight the many iterative paths [50].

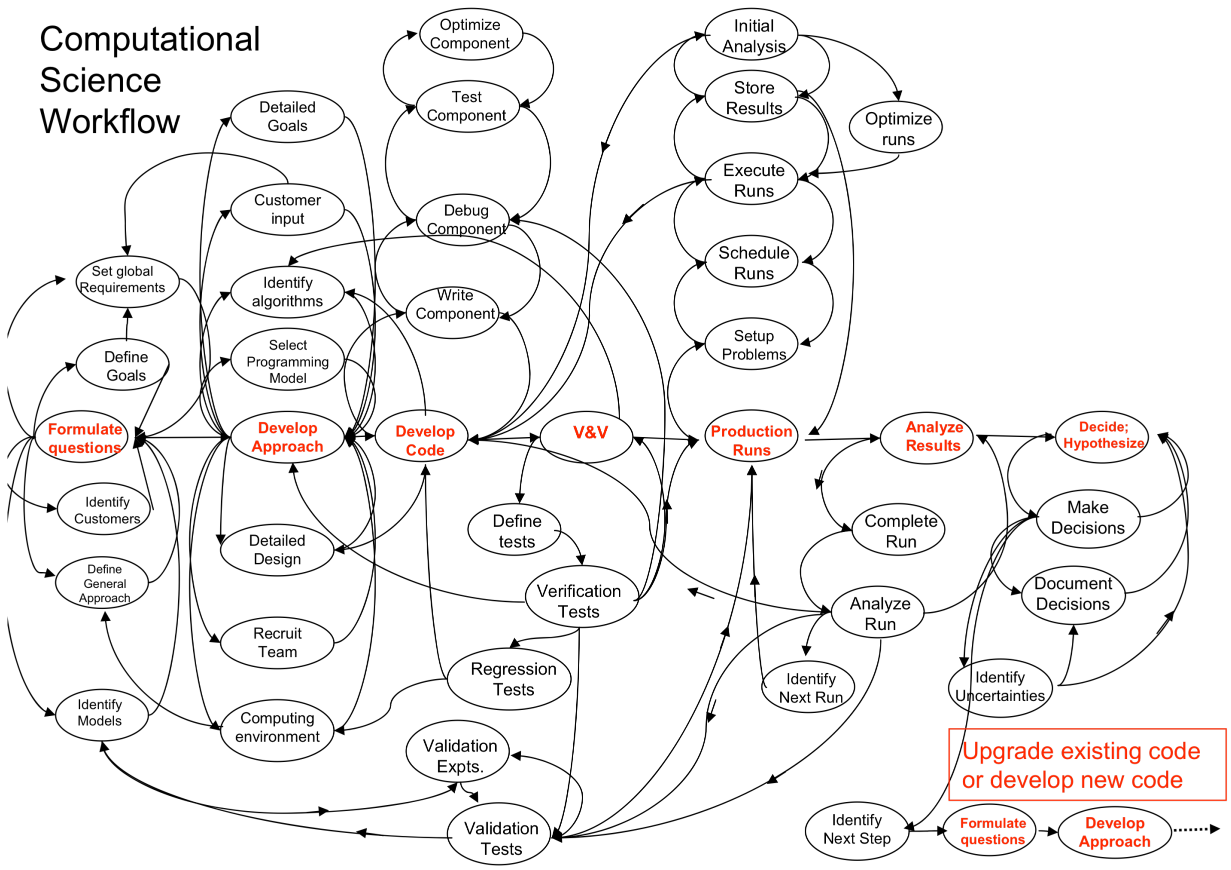


Figure 6: Complex HPC workflow[13]

The proposed workflows represent interesting and useful compilations of data from informant interviews, describing in considerable detail how they think about their work at the programming

[13] Figure reproduced with permission from “Large-Scale Computational Scientific and Engineering Code Development and Production Workflows” [50].

and project layers of the extended system model. As working hypotheses for a research strategy, however, they are overspecified: that is, they contain more detail than can be validated at this early phase of research. Without additional data behind these workflow models, it is not possible to identify significant obstacles to productivity. Addressing the expertise hypothesis would take further second-stage research and require more details.

3.5 Expertise, effort, and intellectual workflows

An important hypothesis from first-phase research was that an important factor limiting HPC software development productivity is the level and range of expertise needed. A second-phase research goal was to validate the hypothesis concerning the different areas of expertise needed and seek significant bottlenecks whose removal might have a dramatic positive impact on productivity.

In this phase, the strategy was to construct a workflow for HPC programming that, in contrast to HPCS workflows, would emphasize only *generalized activities* and the *expertise needed* to support them. This workflow could then be validated and estimates could be constructed, correlating the results with data from other sources. Candidate bottlenecks would appear as areas requiring considerable effort and/or expertise that could be seen as not essential to the scientific objective, but rather accidents of the technologies being used.

The first step toward constructing an *intellectual workflow* was a set of observational studies using professional programmers. During these studies, a variety of data was collected, both quantitative (automated measurement of time spent interacting with programming tools) and qualitative (subject journaling and informal interviews).

Hackystat, an in-process software engineering measurement and analysis tool [24], unobtrusively recorded hours of event traces from instrumented software development tools. A small excerpt, appearing in Table 3, exemplifies Hackystat data collected during a five-hour work session. It includes the session date, total time the programmer spent working with particular files and a detailed event log (first 10 minutes). Hackystat collects data from multiple streams on a server and can produce many kinds of activity reports and analyses spanning different time frames [23].

Table 3: Examples of Hackystat detail

Date	Time in Each File	Time /Activity Detail[14]
16-Mar-2005	chargee.f90 (0.4hrs) chargei.f90 (0.7hrs) field.f90 (0.1hrs) int_frac_parts (1.3hrs) module.f90 (0.4hrs) poisson.f90 (0.5hrs) pushe.f90 (0.2hrs) pushi.f90 (0.2hrs) setup.f90 (0.6hrs) smooth.f90 (0.2hrs) snapshot.f90 (0.1hrs) summation_notation.f90 (0.2hrs) transactional_memory_example.f90 (0.2hrs)	06:30 AM 2 06:30 ls 06:30 history 06:31 vid 06:31 ls 06:31 wc -l *.f90 06:31 vim ~/ /diary 06:32 jobs 06:32 grep qtinv * 06:32 vim chargei.f90 06:34 vid chargei.f90 06:35 jobs 06:35 vim chargei.f90 06:36 grep istep * 06:36 vim chargei.f90 06:37 grep jtion * 06:37 grep kzion * 06:38 vim pushi.f90 06:39 jobs 06:39 vid 06:40 AM 06:40 ls 06:40 jobs 06:40 more main.f90 …

Although Hackystat data is invaluable for its detail and objectivity, it is necessarily incomplete. For example, it does not account for time spent on the phone, walking down the hall to speak with a colleague, or just sitting and thinking. Figure 7 shows the total daily time (interactions with programming tools) captured by Hackystat for a professional HPC programmer over several weeks. It is understood in general software development that there are many important activities away from the edit-debug loop, and this is clearly the case for HPC programming, as well.

[14] Entries in the Time/Activity Detail column are commands typed by the user into a UNIX shell (command interpreter). For example, "ls" enumerates file names in a particular file system context, "wc – l" counts the lines in file, and "vim" opens an editing session on a file.

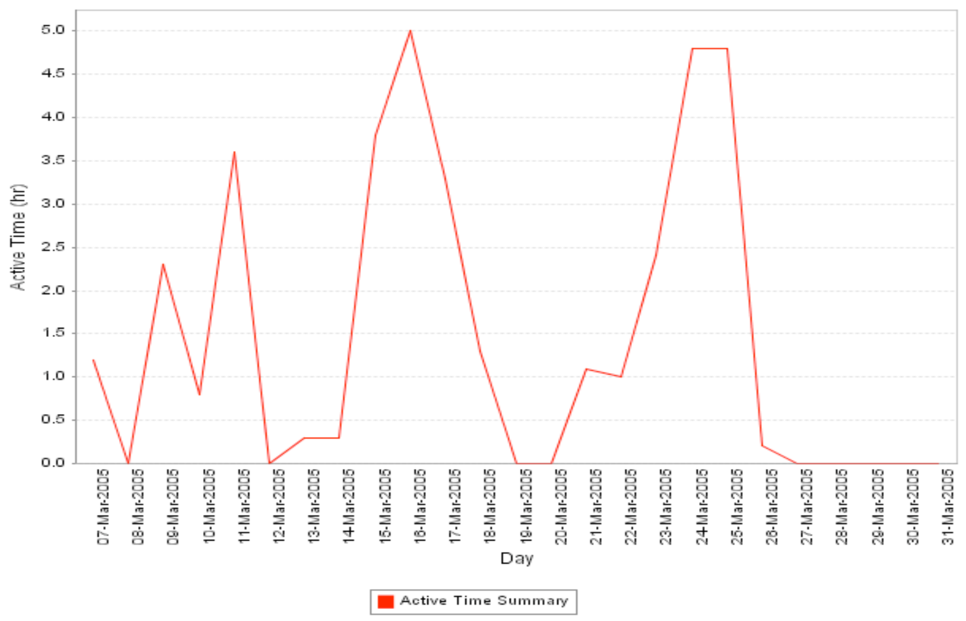


Figure 7: Time per day spent interacting with programming tools

Hackystat also cannot record *why* certain tools were used, for example whether using the telephone or a web browser concerns a requirements task, programming task, debugging, or whether the programmer is on task at all. The Hackystat data was complemented with qualitative data in near real-time: time-stamped journals written by the professional subjects who agreed to record personal narratives as they worked. Finally, additional qualitative data was collected during informal interviews.

Manual coding of the qualitative data yielded important insights in how to interpret the Hackystat data. For example, it confirmed that a great deal of the time away from instrumented tools, during which Hackystat records no data, is in fact on task: most often it was spent trying to understand difficult aspects of the work.

Furthermore, some of the time spent with programming tools was actually used for executing small experiments, such as downloading and playing with code imported from elsewhere. Such experiments sometimes produced insight or answered questions, but the code written during these episodes was often discarded. Such periods should be recorded as part of an understanding phase, rather than coding, a distinction that cannot be drawn from Hackystat data alone.

By triangulating data from many sources, such as Hackystat telemetry data, journals, interviews, and contextual information from the earlier case studies, it was possible to refine the team's understanding of the work and the significance of expertise in carrying it out. An alternate model of the workflow was created, as shown in Figure 8, which emphasized generalized activities at a granularity more in keeping with available data. This model is divided into stages of different kinds of activities that require different skill sets. These activities are defined less in terms of concrete actions, as were those in the HPCS workflows (Section 3.4), and more in terms of the goals and intellectual activities relevant to the team's hypothesis about expertise from first-phase research. In practice, stages overlap and iterate, with only a broadly sequential relationship.

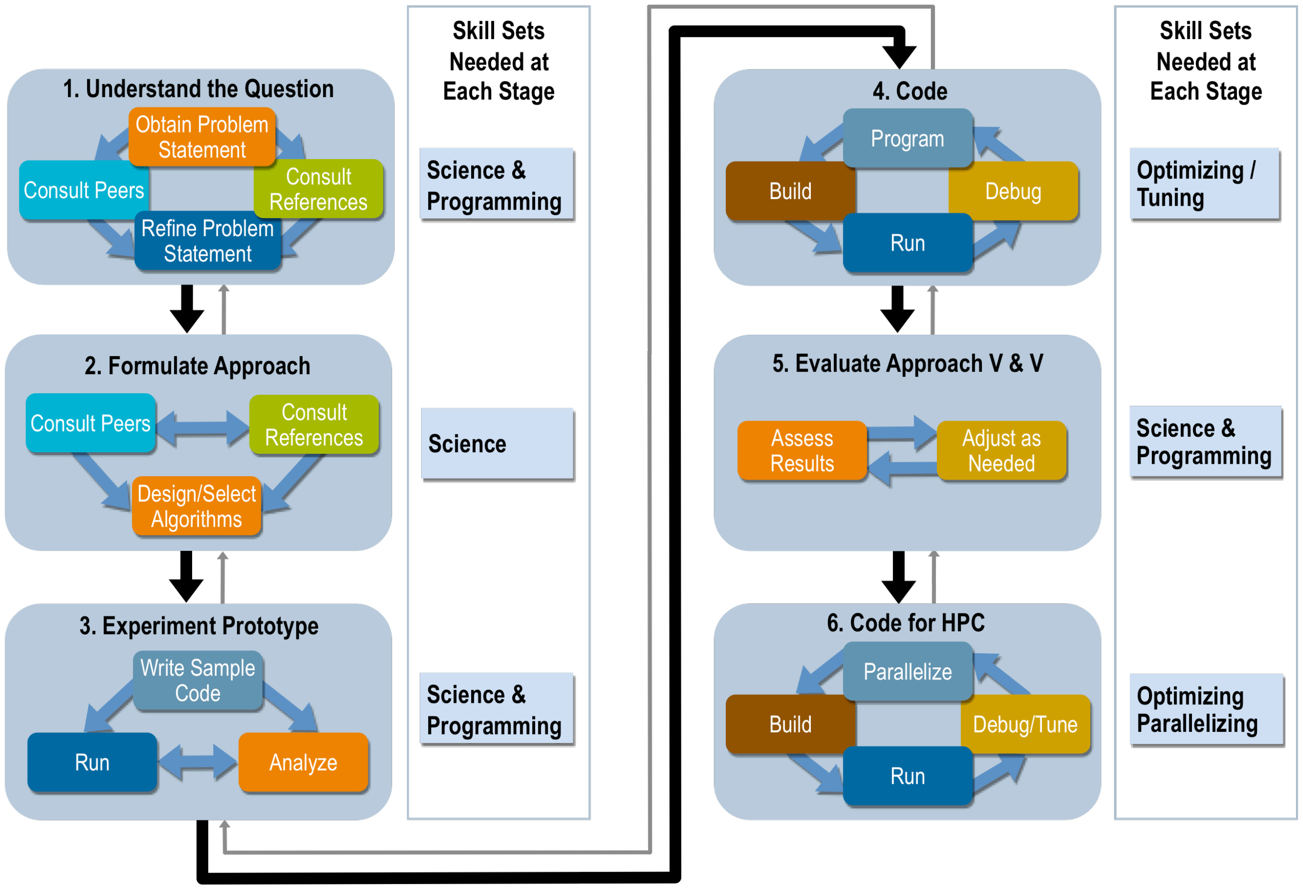


Figure 8: HPC development stages and skill sets

Then, activities were weighted with other available data. This included fine-grained data, such as the Hackystat logs for individual programmers, as well as very coarse-grained data from other sources.

For example, collaborative case studies (Section 3.2) solicited estimates of total time spent on specific *activity sets*: categories created by interviewers at the beginning of the study. These categories, along with the results, are summarized in Table 4. They do not align with the stages identified in the workflow shown above in Figure 8, since the study's methodology allowed the categories (models of the phenomena) to emerge from first-stage data. This is a clear example of how initial assumptions about phenomena under study can determine what type of data is collected. Interpreting the relationship between case study data and the workflow required additional analysis that drew on other contextual data.

Table 4: Total time spent on key activity sets in case study projects

Project	Analysis and Design	Implementation	Testing	Maintenance
Falcon	25%–35%	25%–35%	15%–30%	10%–30%
Hawk	25%	40%	20%	15%
Condor	15%	55%	15%	15%
Eagle	25%	55%	15%	15%
Nene	35%	45%	15%	5%

Yet another source of data appeared in the form of project staffing timelines. Table 5 summarizes this data for one of the collaborative case studies. Management data allowed development to be divided into two broad categories: the first leading to a working serial application with the desired scientific properties; the second leading to a rewritten version of the same code that is parallelized and tuned for performance. The resulting application contained approximately 134,000 lines of executable code [25]. As before, other analysis and contextual information helped estimate the relationship between this data and the workflow stages.

Table 5: HPC application project timeline and staffing

	Understand, Formulate Experiment, Prototype	Code for HPC	Summary
Timeline	4 years	3 years	7 years
Staff FTE	2 people	3 people	
Effort	8 person-years	9 person-years	17 person-years

Among the many insights derived from these studies came validation of the workflow's structure (Figure 8). In addition, estimates of the amount of resources and expertise required in each stage were developed.

Finally, in order to validate such findings from the case studies and developer instrumentation, closed-form surveys (typical of third-phase research) were conducted across larger samples of mission partner teams using a standardized closed-answer questionnaire. Statistical analysis of the data supported the team's conclusions in this area [55].

The team concluded that the estimates were more than precise enough to classify groups of activities that consume the most resources. These are:

- *Developing correct scientific programs*: activities associated with translating an understanding of the scientific problem that must be solved (for example, a predictive weather model) into working code, often starting with or including code from other projects; this corresponds to the first three phases in the workflow.
- *Code optimization and tuning*: activities associated with refining a serial version of the code to ensure correctness and achieve desired levels of accuracy and efficiency; this corresponds to the fourth and fifth phases in the workflow.

- *Code parallelization and optimization*: activities associated with parallelizing the code and tuning to achieve high machine utilization and rapid execution; this corresponds to the final phase in the workflow.
- *Porting*: where a solution exists, this comprises activities associated with translating the existing solution to a representation appropriate for a new platform. This activity was not an explicit part of the studies conducted at that time, although its importance and relationship to the activities became clearer as the studies progressed.

The data show that these activities are expensive, not only in total amount of effort (Table 5), but also in expertise needed. Three distinct skill sets, in addition to project management, are required:

- Mastery of the application domain's science
- Scientific application programming
- Tuning or optimization of programs for efficient execution on a particular parallel platform

Annotations in Figure 8 summarize the skills needed at each stage of the workflow. Because these skills are scarce, this was a significant constraint on some of the projects studied. Finally, overlapping skills and effective collaboration are essential, raising the bar even higher.

From this, two interrelated bottlenecks to productivity in HPC application development were identified. These obstacles were of sufficient magnitude that, if addressed, they would contribute significantly to the HPCS productivity goal:

- *Programming effort*. Of the four activity groups that consume the most resources, only the first (developing correct scientific programs) is essential to the scientific objectives. A significant reduction in the need for the others would dramatically increase productivity, not only lowering cost, but also reducing time to solution.
- *Expertise*. In general, human resources are not fungible—that is, expertise in one area does not imply expertise in another—so people cannot be arbitrarily reallocated to critical tasks. Currently, there is a shortage of experts skilled in scientific application programming and tuning for parallel platforms. Sarkar et al postulate an expertise gap in these areas, exacerbated by the increasing complexity of both scientific applications and parallel platforms [52], a gap validated by the team's data [55]. A reduction of skill level needed for any activity group would dramatically increase productivity, not only by lowering cost and reducing time to solution, but also by increasing the number of HPC applications that could be produced with the available pool of experts.

3.6 Code libraries and expertise

A historically important approach to improving programming productivity in HPC and general programming communities has been the use of *libraries*: collections of well-documented and tested code that can be reused without intimate knowledge of their construction. A good library can reduce both the amount of effort and expertise associated with application development, making library construction a possible strategy for a productivity breakthrough.

In fact, a number of informants in the HPC community offered this prospect as a possible solution to the productivity problem. This view is highly plausible: libraries play a very important role in the history of HPC programming and will clearly continue to do so. However, one of the team's experiments casts doubt on this particular conventional wisdom.

Two experienced programmers were instructed to study one of the proposed benchmark applications[15] under development by other groups in the HPCS research community. Each subject, working separately, was asked to port that application from its "executable specification" written in MATLAB [37] into the Java™ programming language [17]. Little further direction was given. One subject had HPC programming experience but was new to Java; the other was familiar with Java but new to HPC programming. Data were collected using journaling, occasional observation, interviews, and code inspections.

Among the findings was that neither subject adopted any of the readily available libraries, even though that would have shortened their tasks significantly. Each subject did spend time actively seeking suitable libraries. However, in every instance, the subject had too little confidence in the choice to risk commitment to something they understood so little as the task itself. Both ended up completing the application port without supporting libraries.

This outcome suggests that libraries, although they can reduce the level of effort needed in many cases, require a high level of expertise to be selected with confidence, used appropriately, and chosen early enough in a development task to make a difference. The conclusion was reached that libraries can reduce effort and, to a lesser extent, the need for expertise, but not with sufficient magnitude to create a productivity breakthrough.

3.7 High-programmability code

Another historically important approach to improving programming productivity in both the HPC and general programming communities has been the development of better programming languages and techniques. The team investigated the prospects for a significant productivity breakthrough in this area, drawing on the study's experiments and case study data. The conclusion was that significant breakthroughs are possible.

The experiments were conducted by an experienced professional HPC programmer. In addition, data collected from the mission partners was revisited for insight into the same questions. The results strongly suggest that significant improvement is possible with existing technologies and that more improvement could be gained through investment in improving currently available technologies, as well as developing new programming languages designed with these issues in mind.

3.7.1 NAS benchmark improvement

One experiment explored the character of HPC applications. In particular, the team investigated how applications written using conventional HPC programming models might be transformed into a style that would lead to more productive programming. This might be possible through application of software engineering techniques well understood in the general programming

[15] Scalable Synthetic Compact Application #2 (Graph Theory) [4].

community, but considered inappropriate by HPC programmers because of their cost in runtime performance.

To gather baseline data, a programmer rewrote well-known HPC code examples, including the NASA Advanced Supercomputing (NAS) parallel benchmarks [41] and a few others, with clarity and compactness of the code as paramount goals. The code was stripped of explicit Message Passing Interface (MPI) data decomposition and distribution, manual optimizations were removed, and the more expressive (and abstract) array syntax of Fortran 90 was adopted in place of the customary FORTRAN 77 operations. The programmer endeavored to align the code with the published specification as much as possible.[16] The modified code turned out to be as much as ten times smaller and vastly easier to read and understand.

The code excerpt in Figure 9, taken from the benchmark suite rewritten in the experiment, shows how improved abstraction can make code dramatically easier to read and understand than its conventional counterpart. A significant and somewhat unexpected result was that with thoughtful programming, language features available in Fortran 90 could be exploited to produce code that was visibly more aligned with underlying mathematics and free of language artifacts that were irrelevant to the problem.

[16] No useful measures or breakdowns of time and effort were recorded in this experiment; the focus was on the plausibility of code written in this nonstandard fashion, not on the cost of reengineering legacy code. In fact, the original experimental design did not even require rewriting the benchmarks into an existing language.

```
call resid(u,v,r,n1,n2,n3,a,k)
callnorm2u3(r,n1,n2,n3,rnm2,rnmu,nx(lt),ny(lt),nz(lt))
old2 = rnm2
oldu = rnmu
do  it=1,nit
    call mg3P(u,v,r,a,c,n1,n2,n3,k)
    call resid(u,v,r,n1,n2,n3,a,k)
enddo
call norm2u3(r,n1,n2,n3,rnm2,rnmu,nx(lt),ny(lt),nz(lt))
```

(a) Original FORTRAN 77

Each of the four iterations consists of the following two steps,
r = v - A u   (evaluate residual)
u = u + Mk r (apply correction)
...
Start the clock before evaluating the residual for the first time, ...
Stop the clock after evaluating the norm of the final residual.

(b) Specification

```
do iter = 1, niter
  r = v - A(u)    ! evaluate residual
  u = u + M(r)    ! apply correction
enddo
r = v - A(u)    ! evaluate residual
L2norm = sqrt(sum(r*r)/size(r))
```

(c) High-programmability style with Fortran 90

Figure 9: Improved code excerpt from NAS MG benchmark (timed portion)

This experiment also produced a dramatic reduction in code size across the benchmark suite and some other examples, summarized in Figure 10. Approximately half of the size reduction was due to removal of explicit data decomposition and distribution, a result consistent with other studies on the contribution made by MPI to code size [9] [31].

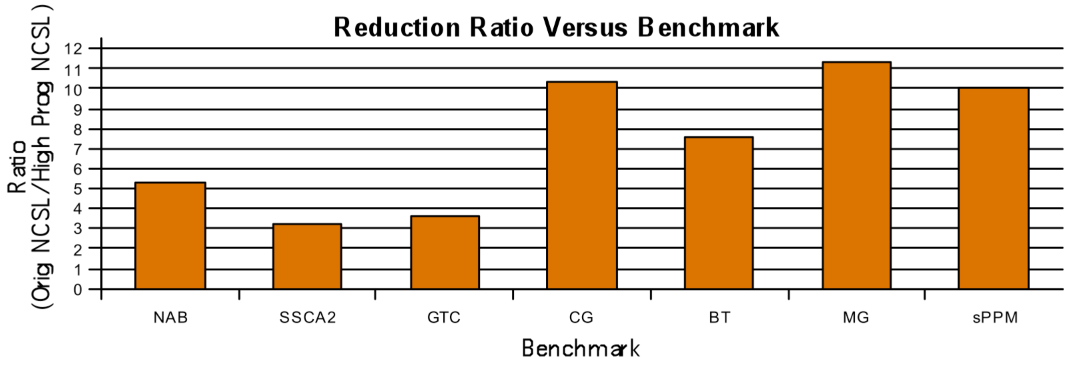


Figure 10: Benchmark code improvement: size reduction [35]

It is well-understood in the software engineering community that software's lifetime maintenance cost is very strongly correlated to code size, suggesting that this approach to HPC programming can produce significant reduction in effort over an application's life cycle.

3.7.2 NAS BT I/O code modification

In a closely related experiment, an experienced HPC programmer was asked to add new functionality to an established benchmark (NAS block-tridiagonal or BT I/O). This exercise was part of the original benchmark specification and was reflective of the maintenance phase of an application's life cycle. The starting point was the version of the benchmark rewritten in the previous experiment. The task was to add checkpoint-style I/O and perform the experiment several times, each time using a different programming model. To summarize the results from four of these experiments:

- *High-programmability style with Fortran 90*: Exploiting simplifications made during the previous experiment, the important application state was contained in a single array, for which high-level Fortran 90 array operations are available.
- *Serial FORTRAN 77*: Standard FORTRAN I/O for reading and writing the array contents.
- *MPI – simple*: Parallel I/O with naïve use of the MPI I/O API.
- *MPI – optimized*: Parallel I/O optimized use of the MPI I/O API, for example using collective I/O operations.

The task required implementing four operations: setup, write, read and close. Table 6 shows the amount of code required for the task using each of the four programming models. The total lines of code (LOC) required for each model gives a rough measure of the code's complexity and expected lifetime maintenance cost for this segment.

Table 6: NAS BT I/O code modification - lines of code

Programming Model	Setup	Write	Read	Close	Total LOC
High programmability with F90	1	1	1	1	4
Serial FORTRAN 77	7	19	20	1	47
MPI - simple	25	22	23	1	71
MPI - optimized	144	12	13	1	170

Two representative examples of the added code appear in Figure 11: the write operation, as written in both the high-programmability style with Fortran 90 and naïve MPI programming models, respectively. A complete listing of code from this experiment appears in Appendix 17.

```
write(20) u   ! write entire array, including boundary cells
```

(a) Write: high-programmability style with Fortran 90

```
do cio=1,ncells
     do kio=0, cell_size(3,cio)-1
         do jio=0, cell_size(2,cio)-1
             iseek=5*(cell_low(1,cio) +
$                   PROBLEM_SIZE*((cell_low(2,cio)+jio) +
$                   PROBLEM_SIZE*((cell_low(3,cio)+kio) +
$                   PROBLEM_SIZE*idump)))

             count=5*cell_size(1,cio)

             call MPI_File_write_at(fp, iseek,
$                  u(1,0,jio,kio,cio),
$                  count, MPI_DOUBLE_PRECISION,
$                  mstatus, ierr)

             if (ierr .ne. MPI_SUCCESS) then
                 print *, 'Error writing to file'
                 stop
             endif
         enddo
     enddo
 enddo
```

(b) Write: MPI - simple

Figure 11: NAS BT I/O code modification - code samples

3.7.3 Performance penalty

When simplifying code, the HPC community's first concern is the cost in performance, including parallel speedup. This concern is justified, but substantial grounds for hope were found.

First, it was acknowledged that experiences vary tremendously. In the experiments, however, programmers found the performance shortfall of high-programmability applications to be rather tolerable. Approximately 2x performance loss was representative for most of the cases considered, even when scaling up to 100 threads. These results were achieved using commercial, automatically parallelizing compilers and large-scale, shared-memory hardware [35].

Many opportunities were found for on-going product improvements. For example, compilers could produce better performance and automatic parallelization such as faster algorithms for implementing array syntax and more-aggressive exploitation of concurrency. Much compiler development, however, is driven by benchmarks that are written in traditional styles, which stunts the development of a high-programmability application ecosystem.

On the hardware side, better support for high-end shared-memory systems and latency hiding would help software developers deliver high parallel performance for high-programmability applications, compilers, and middleware.

On one hand, HPC conventional wisdom was confirmed: programming to minimize software development costs can be at odds with programming to maximize hardware performance. On the other hand, the great potential to reduce code volumes (nominally, 10x) and relatively tolerable cost in performance (nominally, 2x) was surprising.

3.7.4 Case study data

Analysis of raw data from the case studies provided additional validation of the significance of programming language on productivity. An alternate style of HPC code development was observed in use at some sites, using MATLAB in an early phase of development instead of traditional choices like Fortran and C++. The MATLAB programming language, although closely related to Fortran, is characterized by extensive use of higher level (more abstract) library functions that are specialized for the kind of scientific and numerical programming common in HPC applications. Although MATLAB did not offer parallel scalability at the time of those studies (efforts to correct this are underway), and although it incurred a considerable performance penalty when compared to serial (nonparallel) Fortran, many projects chose MATLAB over the alternatives.

Two case studies were selected for comparison: one used MATLAB in early phases and one used Fortran. In both cases, serial programs developed in early stages were eventually rewritten into C++ for parallel scalability. The resulting C++ programs were of roughly comparable size.

Table 7 summarizes aggregate staffing costs for these two projects. The "Code for HPC" phase, in which serial codes were rewritten for parallel scalability, incurred similar costs in both examples, confirming the notion that the two projects were in some way comparable. On the other hand, the earlier phases "Understand, Formulate, Experiment, Prototype," in which a serial program was eventually produced that captured an understanding of domain and numerical solution methods, differed dramatically in cost between the two projects.

Table 7: Case study development costs, two examples

Project	Domain	Understand, Formulate Experiment, Prototype	Code for HPC	Costs
Hawk	Fluid Dynamics	Time: 4 years FTE: 2 people Total: 8 person-years Language: **F90**	Time: 3 years FTE: 3 people Total: 9 person years Language: C++	Time-to-solution: 7 years Effort: 17 person years
Eagle	Signal Processing	Time: 1 year FTE: 1 person Total: 1 person-year Language: **MATLAB**	Time: 3 years FTE: 3 people Total: 9 person-years Language: C++	Time-to-solution: 4 years Effort: 10 person years

The results from these studies of programming languages and technologies appeared in a requirement for ongoing improvement in programming languages and their use (Section 5.1.4), including the development of experimental new languages such as Sun's Fortress [2] (Section 12.4).

3.8 Software development tools

Another possible opportunity for productivity breakthroughs, almost by definition, concerned tools used by HPC programmers, identified as the Development Environment layer in the system model (Figure 5 on Page 18). These investigations produced findings as dramatic as those reported for programming languages and techniques in the previous section. In this area, however, the findings were consistent with the community's conventional wisdom, at least to the extent that tools were a problem [62].

HPC programmers use many tools in common with wider computing communities, such as editors and source code control systems.[17] However, they do not use IDEs, which have contributed greatly to productivity among C++ and Java programmers. Conventional wisdom holds that HPC programming languages and practices are not well supported by IDEs; no data was found to contradict this opinion.

The tools that matter, the ones called out for complaint by HPC programmers, are those designed for the unique requirements of highly parallel, scientific computing, such as specialized compilers and performance analyzers. Common complaints included:

- Tools are hard to learn.
- Tools do not scale (in problem size or parallelism).
- Tools differ across platforms.
- Tools are slow to appear on new platforms.
- Tool support is inadequate.
- Tools are hard to test.
- Tool availability is uncertain.
- Tools are often too expensive for universities.
- Tools are seen as a risk to project success.

The last complaint, typically heard from project managers, was the most startling: rather than being seen as a source of productivity and opportunity for productivity growth, specialized HPC tools were seen as a risk to project success. Given the essential role played by tools in any kind of productivity, this situation was identified by the team for further investigation as a potential bottleneck.

The most significant fact is that programmers perceive a problem, implicitly believing that they would be more productive if they had better tools. A 1997 technical report promoting HPC software standards began:

> Although the number and variety of high-performance computing (HPC) systems has grown dramatically over the last decade, the quality of system

[17] Data taken from case studies reported in Section 3.2 suggest that the HPC community's use of general-purpose tools is several decades behind the wider programming community, evidently because those tools, although useful, do not address dominant problems confronting HPC programmers.

software and tools remains far below the expectations of the user community. [47]

More evidence of this belief and its intensity comes from several attempts by HPC practitioners to do something about the problem, most notably the now-defunct Parallel Tools Consortium [46]. Bitterness about those failures lingers.

As the team dug deeper, revisiting earlier data and conducting additional unstructured interviews, they discovered that a surprising amount of effort was dedicated to tool development, effort that did not directly address the scientific problems at hand. Frustration about this situation was observed at the project and laboratory management levels. The effort showed up in several ways:

- *Retooling*. A major scientific project lost a year's progress when the optimizing compiler became unavailable; the supplier, a very small software company, was purchased by a large corporation that removed the compiler from the market. Also, useful tools sometimes emerge out of university or other research environments but become unavailable when the developer graduates, migrates, or otherwise moves on.
- *Relearning*. HPC tools tend to be complex: they address difficult problems, and the small market size dictates against large investment in usability. Furthermore, some important tools are supplied by platform vendors and are thus unique. Learning to use new tools imposes a considerable delay when a programmer moves to a different environment.
- *Migrating*. One software support group estimated that it typically takes one year before a new supercomputer becomes fully productive. This is a significant loss of value for machines with useful lives of four years. Reasons cited included: new platforms often arrive without critical software such as tools, new software is very buggy, and it takes a long time for programmers to become productive with new tools.
- *Negotiating*. Some support groups reported pushing tool vendors toward open source, even when they were willing to pay for support. The reason given was neither financial nor ideological, but rather indemnification against loss of critical tools.
- *Reinventing*. Several HPC groups reported developing their own tools, often duplicating functionality available in commercial tools, and putting the results into open source. The choice of open source serves both as indemnification, as mentioned above, and a way to share with other HPC programmers (the publicly funded laboratories had been doing this for many years before the formal existence of open source).

Analysis of these data began with two general observations. The first is that in HPC environments, tools used to develop an application become an *essential part of the application for its lifetime*. Although this is also somewhat true in general computing environments, it is especially notable in the HPC environment where the cost and complexity of specialized tools, combined with the small market size and poorly funded consumers, makes the business environment for them extremely fragile. Unfortunately, HPC applications can depend on those specialized tools for 30 or more years. The problem's shape became clear: adopting a tool, including a programming language, at a project's outset amounts to placing a bet that the tool will be available and effective on all the platforms where the application will be ported over its lifetime. In other words, tools become a risk.

A second general observation concerned the apparent intractability of the tool problem*,* noted especially by participants in failed attempts at solutions. Data was reviewed that described the contexts in which the evident stakeholders operated: hardware vendors, researchers, independent tool companies, customers, voluntary collaborators, and community heroes. The conclusion:

> None of these stakeholders have the primary mission, resources, or longevity to produce what's needed: to create a complete productivity infrastructure for the HPC community. [62]

This means that the real issue is not an individual's or group's productivity, but rather the HPC community's productivity as a whole.

Opinions from informants about underlying causes of this problem or prospects for a solution were diverse and unconvincing. The analysis, based on the model described in Section 2.2.2 and using social science methodologies, identified a serious misalignment of responsibilities that crossed many layers of the extended system model (Figure 5): programmers, projects, organizations, and missions. In particular, it was observed that the need for high-quality specialized tools, experienced most painfully at programmer and project levels, was not acknowledged, understood, or funded at the mission level, which is the only level where sufficient resources and longevity were available to create general solutions [62]. This finding was consistent with, and explained the need for, recommendation 4 from the 2003–2004 National Research Council study on the future of supercomputing in the United States:

> The creation and long-term maintenance of the software that is key to supercomputing *requires the support of those agencies that are responsible for supercomputing R&D*. That software includes operating systems, libraries, compilers, software development and data analysis tools, application codes, and databases [18] [emphasis added].

The analysis further identified critical characteristics of the software in the productivity infrastructure: common toolset, functionally complete, multiplatform, specialized, widely available, enduring, open to research and financially viable [62]. With such an infrastructure, the entire community's productivity would improve. Individuals and groups would be more productive with better tools, less time would be spent on tool-related efforts and the pool of capable HPC programmers would grow.

Although a significant breakthrough in this area was beyond the reach of any single participant in the HPCS program, this analysis informed tool requirements (Section 5.1.5) and Sun's tool strategy (Section 12).

3.9 Administration

Sun's comprehensive view of productivity led to the addition of *administration* to the HPCS program's list of productivity factors (performance, robustness, programmability, and portability). The administration category includes general activities typically associated with making a supercomputer useful for its intended purpose: operations, administration, maintenance, and provisioning (OAM&P). These activities play a significant role in supercomputer productivity. Administration costs contribute significantly to the cost of running a supercomputer, and effective administration can make a significant impact on the value derived by scientists and programmers.

A separate team, working in collaboration with the core productivity team, conducted research into productivity factors associated with administration. Details of those results are beyond the scope of this report, but the findings that made an impact on Hero's design are summarized here.

Analysis of data gathered by the administration team revealed bottlenecks in two general areas: availability and effective resource utilization.

Availability refers to the percentage of time that a machine is in operation and able to process user jobs. Productivity bottlenecks in this area include:

- Manually restarting failed jobs after hardware failures have been diagnosed and corrected.
- Diagnosing faults, in particular locating failed hardware components in a timely way, a problem likely to become acute as systems grow to petascale.
- Performing software upgrades, especially urgent upgrades needed for security purposes, another problem likely to have a growing impact on availability as system size increases.

Effective resource utilization covers the degree to which available computing resources are effective in helping users achieve their goals. It has several aspects:

- Computing jobs vary considerably in type of computing resources needed: memory, CPU cycles, disk I/O bandwidth, or disk space. It is difficult or impossible to configure these resources precisely, without waste, in most systems; it is even more difficult to characterize the needs of certain applications in advance or on the fly as they change computational phases.
- In addition to resource availability, job scheduling is sensitive to administrative concerns such as priority and urgency; well-defined scheduling policies are usually in place but are often manipulated by users to secure better responsiveness, sometimes by exercising political influence.
- A difficult scheduling issue, unique to supercomputers, is the cost (in resources and availability) of switching from capacity mode into capability modes. An effective job scheduler operating in capacity mode may have a large number of jobs running simultaneously, with a wide distribution of completion times. Switching into capability mode requires clearing out capacity jobs so that the entire machine can be dedicated to a single computation. The transition can produce long delays (when jobs are allowed to complete) or unhappy users (when jobs are terminated prematurely).
- A general concern in resource utilization is the ability of administrators to gather and visualize enough information to make effective, sometimes urgent decisions.

Requirements informed by this analysis appear in Section 5.4 and some solutions proposed for Hero are in Section 11.4.

3.10 Technology adoption

The findings reported so far represent a snapshot of the status quo in the supercomputing milieu across the many levels depicted in Figure 5 (on Page 18). As suggested earlier, a change of the magnitude proposed by DARPA (10x more productive) demands a fundamental shift in the way things are done, a change that likely pervades every layer of the system, from semiconductors to mission sponsors. Hero succeeds only if the community can make the transition, a steep challenge for a community whose software development practices appear to evolve at a glacial pace.

The team investigated how the HPC community adopts new technology, drawing on many sources. It found that the pace of adoption for software technologies is indeed glacial but not completely static:

- The structure and pace of technology adoption follow naturally from characteristics of the existing HPC environment (Section 2.1).
- A technology adoption process exists and is best described in terms of three stages: established, emerging, and future.
- Organizations typically engage in all three stages simultaneously, differing mainly in distribution of effort and mission-driven time constraints.

A closer look at these three stages added a different dimension to the understanding of productivity, and in particular to bottlenecks in the adoption of any new technologies, no matter how promising. In particular, it became clear that successful adoption of Hero requires support in all three stages (Section 5.1.4).

3.10.1 Established technologies

Widely established HPC technologies have ubiquitous and reliable supporting software infrastructure, drawing on programmers' expertise in FORTRAN 77, Fortran 90/95, C++, and MPI-1 [38]. Most of these applications already exist and use established technologies that will require support for at least 30 years into the future.

The window of opportunity for software technology adoption comes mainly at the beginning of new projects. The cost of rewriting existing code is so great that projects seldom adopt new technologies once underway. Support teams experiment with new technologies in noncritical ways, but adoption takes place only if a new technology fits well, which means it offers immediate and substantial benefit, requires little or no code change, avoids the cost of reverification and revalidating and is expected to be widely supported into the indefinite future without substantial additional cost or fear of disappearance. This latter concern, based on painful experience, is increasingly addressed through reliance on open source software, not out of ideology but as a form of indemnification against the kinds of disasters that cause tools to be seen as risks (Section 3.8) [62].

3.10.2 Emerging technologies

The limitations of established technologies are well understood: thoughtful members of the community believe that a transition to more productive technologies is necessary. Many HPC sites participate actively in the development of new technologies.

Even so, the window of opportunity at the beginning of new projects is exceedingly narrow. Project managers adopt new technologies only if they have great familiarity and high confidence in the future outlook for cross-platform standardization and support. Examples of emerging technologies include the three partitioned global address space (PGAS) languages - Co-Array Fortran [42], Unified Parallel C (UPC) [61] and Titanium [19], as well as MATLAB [37] and scripting languages. Some of these technologies have been emerging for 10 years. MPI-2 [39] also falls in this category, but it is mainly a transitional technology leading to PGAS languages.

Time lines for technology adoption vary considerably, ranging from sites with no intention of abandoning established technologies in the foreseeable future ("MPI forever" is a ubiquitous sentiment in some organizations) to those already heavily committed to emerging technologies ("can't live without MATLAB or UPC").

3.10.3 Future technologies

A few members of the HPC community engage in research beyond the emerging technologies. However, most see these as irrelevant or too far out. Current examples of future technologies include the high-productivity computing systems (HPCS) languages (Chapel [8], X10 [22] and Fortress [2], see Section 12.4), as well as parallel versions of MATLAB. Even these visionary languages are expected to interoperate with existing and emerging technologies; examples include parallel MATLAB with MATLAB and Fortress with Fortran.

3.11 Summary of findings

This section summarizes findings concerning significant bottlenecks that constrain HPC application development in environments studied during the DARPA program, in particular the mission partners. The framework within which the findings were interpreted is the workflow in Figure 8, expressed in terms of skills (expertise) and tasks (effort) across several stages.

Some of the findings were general; some described conventional wisdom about productivity that turned out to be inaccurate. However, some identified bottlenecks so significant that their resolution could move the community toward the 10x DARPA goal. These findings shaped the goals and informed the requirements described in following sections.

General findings about HPC in the environments studied are listed below:

- HPC programming is extraordinarily expensive as measured in labor cost, computational resources, and time to solution.
- HPC programming is on an evolutionary path that will not lead to productivity improvements; the reverse may actually be true.
- The level of skill and domain understanding among HPC programmers is very high.
- Project success depends on good management in combination with skills in several areas: the scientific problem domain, numerical programming, optimization and scaling, or optimizing for highly parallel execution.
- Productive HPC programmers are in short supply.
- Software development tools specialized for HPC are often perceived as risks to project success, rather than generators of productivity.

A number of possible strategies were investigated, some originating in the community's conventional wisdom, that did not represent opportunities for breakthroughs of the magnitude needed for DARPA's goals:

- *Better education* is always appealing, but the imagined advantage was not supported by the data. HPC programming is essentially a craft that is learned on the job. Even with a PhD-level education in science or engineering, it requires years of work to become highly productive in the current HPC programming environment [52].
- *Bigger teams* are not practical as a general strategy. Necessary skills are in short supply. The work requires overlapping skills and collaboration, which depends on collaborative relationships that take time to build and sometimes fail. Finally, scientific programming is essentially exploratory, which means that concurrent development is especially fraught with difficulty [50].
- *Better code libraries* would make a difference, but this strategy for code reuse has already been pursued for years. Good libraries reduce programming effort significantly, but they do not dramatically reduce the need for scarce expertise (Section 3.6).
- *Early workflow stages* are focused on exploration of both science and numerical methods in the problem's domain. This exploration is the essential part of a project; there is no reason to expect a breakthrough here.

On the other hand, the focus on an expertise hypothesis led to bottlenecks where breakthroughs big enough to contribute to DARPA's goals are possible:

- Early workflow stages are constrained by the need to produce working serial codes; this requires considerable effort as well as scarce expertise in numerical programming, obtained through collaboration and/or multi-skilled personnel. Experiences reported by programmers and teams using higher level, more-abstract languages during early stages suggest that considerable advantage can be gained (Section 3.7.4). In fact, the greatest hypothetical breakthrough could come from allowing scientists to carry out this exploration purely in terms of the science and numerical methods, not programming.
- Later workflow stages consist mainly of tasks not directly related to the science objective: optimizing serial codes, rewriting codes for parallel execution, tuning the parallel codes for scalability, and eventually porting. Any of these tasks, and the enormous expertise and effort they require, could in principle be partially automated or eliminated to great advantage.
- Programming languages and techniques can have a dramatic impact on software development productivity (Section 3.7). Thoughtful use of existing languages such as Fortran 90 can produce benefit only with sufficient investment in high-quality compilers and guarantees of future portability (a tools problem). However, the languages available at this time, including emerging PGAS languages, do not support parallelism at the scale needed for petascale systems. As with early-stage tasks, aligning the implementation language abstraction with the domain of science would produce benefits, including reduction of effort through automation of low-level and platform-dependent tasks.
- Programmers are frustrated by inadequate support software, especially the specialized development tools critical to HPC programming. This failure cannot be corrected, despite

persistent effort, at project and lab levels, nor is a solution with the reach of any single platform vendor (Section 3.8). A significant breakthrough is possible in this area, but the relevant funding agencies apparently lack the knowledge, wisdom, or priorities to do so. However, platform vendors can steer tool efforts in more productive directions.

- Nowhere is the lack of advanced tool support more noticeable than in the need for validation and verification; although increasingly important [49], typical approaches are informal.

Selected findings reported by the administration team (Section 3.9) are summarized below:

- The scale of Hero's hardware makes it more difficult to keep the machine functioning correctly and highly available.
- The scale of Hero's processing power increases the challenge of understanding and managing job processing.
- The opportunities for significant productivity breakthroughs in administration are, in many ways, similar to those for programming languages and tools. Effective tools operating at petascale must enable system administration to be highly automated, so staff can make policy decisions at a more abstract level than is now supported.

Finally, a summary of cautions—possible obstacles to achieving the potential productivity breakthroughs identified by the team:

- The change needed to achieve DARPA's challenge of 10x productivity improvement is dramatic and pervasive. The biggest obstacle to change is the evolution of the community's expectations and practices.
- A successful supercomputer must simultaneously support programming technologies in all stages of the adoption model (established, emerging, and future), with special attention to incremental migration (Section 3.10).
- A successful supercomputer must rely on programming technologies (languages, compilers, performance analyzers, and many others) whose ubiquity, future availability, and ongoing evolution are not in doubt. These must be guaranteed by a stakeholder with the appropriate mission, budget, and longevity to convince project management that tools are no longer risks, but keys to a more productive future.

4 Strategic development goals for Hero

As summarized in the preceding section, the key to increasing HPC productivity is to reduce the programming effort and level of expertise required to achieve a given level of machine utilization. The project systematically applied the engineering principles of abstraction and automation to remove these bottlenecks. Although these principles have been extensively applied in other domains, they have been largely eschewed by HPC programmers in favor of hands-on control over performance. However, it is precisely the exercise of this low-level control that leads to programming effort and expertise bottlenecks.

A long-term goal must be to relieve HPC programmers from considering platform dependent details. The team's studies of programmability demonstrated that explicitly parallelizing and optimizing applications for a given machine architecture can cause program size to grow by a factor of ten over a formulation that abstracts from such details (Section 3.7). Also, it is known that total software development time and effort rise super-linearly with code size [5].

Ideally, scientists and programmers would develop application programs using syntax and semantics close to the application domain, without regard for underlying machine architecture. However, empirical data shows that productivity on existing platforms is constrained by both the effort and expertise required to adapt solutions to particular parallel platforms (this includes coding, tuning, parallelization, and optimization). Current HPC environments fall short in providing effective abstractions; in fact, common parallel programming models such as MPI [38] and OpenMP [44] break abstraction by forcing users to explicitly manage process parallelism and communication. Moreover, when programs must be written with additional mechanisms to compensate for machine failures (for example, explicit checkpointing), another important abstraction is broken.

The Hero design strategy is to create a *virtual machine* that hides platform details such as instruction sets, number of processors, and memory architecture. The virtual machine is also designed to be reliable, even if underlying hardware is not. If machine components fail, programs must continue to run, though perhaps with reduced performance. The virtual machine is not provided by any single component architecture, but necessarily results from a synthesis of hardware and software capabilities.

A second strategic platform goal is to *automate* many labor-intensive tasks such as parallelization and optimization. Creating effective abstraction without sacrificing efficiency (especially machine utilization) requires automating many programming and optimization tasks that are now done by hand. Where the scientist or programmer's interface abstracts away the details of hardware management, a development platform must provide capabilities to effectively automate significant parts of code parallelization, memory management, fault recovery, and other platform-dependent details that are now managed explicitly.

These high-level goals are necessarily strategic because it may not be possible to fully satisfy them with current technology. Creating effective domain-specific abstractions (for example, in domain-specific languages) is an open area of research. So too is the technology needed to automate mapping of domain-specific constructs and other abstractions onto massively parallel platforms with high utilization and performance. Even if these research issues can be addressed, the cost of incorporating solutions in any particular platform is still unknown.

Nonetheless, it is important to identify and characterize these strategic platform goals. Addressing productivity problems in the long term requires meeting these goals. Preserving this as a possibility in any future platform means that interim solutions should be waypoints on a path toward those goals. This ensures that requirements and design are given sufficient forethought, so they can be adapted to accommodate future platform changes that address productivity needs.

5 Productivity requirements for Hero

From the results of productivity studies with HPC stakeholders, reported in Section 3, and the strategic goals summarized in the preceding section, requirements were derived that address productive use of human and computer resources. These requirements include the need to reduce the effort and level of expertise required to develop, run, maintain, or port HPC application software. They also include the need to keep a supercomputer as busy as possible with useful computation because an underused, idling, or unavailable supercomputer is an unproductive use of significant capital expenditure.

5.1 Requirements for programmability and portability

Studies of HPC developers establish that two tasks consuming large amounts of time and effort are (1) translating a computational concept into a correctly operating, parallel program and (2) porting the program to a different machine [7] [25] [26] [27] [28] [29] [48] [56]. Moreover, developers cannot reduce development time by adding people to a project. For the foreseeable future, there will be a shortage of personnel with requisite skills. For these reasons, many of the greatest opportunities for productivity improvements will come from improving programmability and portability.

5.1.1 Shared memory model

The application development environment must provide some form of shared memory model. There is strong evidence (including this project's own experiments, Section 3.7) that message passing and other forms of explicit memory management in parallel applications require a significant amount of programming effort and expertise. For example, some scientists claim that writing MPI code is the most difficult aspect of their work [56]. These studies also showed that MPI increases code size by a factor of 1.5 to 2.0, with a corresponding downstream burden on maintenance. The general consensus is that a shared memory model (that is, one in which the programmer does not need to consider where in memory data is stored) is much simpler to program.

The difficulty is in providing a shared memory abstraction while also achieving adequate memory latency and bandwidth. HPC programmers invest time and effort in memory management precisely because different memory allocation strategies result in significant performance differences across different hardware platforms. Programmers will release direct control over memory management only when the system can provide a shared memory model that does not significantly retard execution efficiency. This requires advances in hardware implementation of parallel platforms and memory management provided by system software.

Progress in these areas suggests this will be possible in new machines. For example, the Hero system was architected to provide memory bandwidth and latency sufficient to keep up with the processors for large address spaces, relative to the amount of data accessed by typical parallel computations. While such approaches do not remove all memory latency issues, they can reduce them below the threshold of concern for a large class of applications and enable effective support for legacy MPI applications.

5.1.2 Problem-oriented abstraction

Adequately addressing the expertise gap [55] requires programming environments that support problem-oriented abstraction. Substantial effort throughout the development cycle is expended to manage the interface between application science and parallel programming. Since there is a community-wide shortage of developers who are conversant in both disciplines [52], this problem is expected to persist. In fact, the increasing complexity of new applications, coupled with the increasing complexity of supercomputer platforms, suggests that the expertise gap will only widen unless steps are taken to manage it.

Since handoff and translation problems are inherent in the division of labor between scientists and programmers (Section 3.3), truly addressing this problem will require reducing the level of programming expertise needed—ideally to a point where domain experts can develop most of their own applications. Supercomputer platforms need to provide programming environments that support more problem-oriented abstractions, so developers can think and write applications in terms of the problem domain (rather than the parallel computing domain).

While providing specific problem-oriented abstractions (such as programming constructs for weather simulation) is beyond the scope of any hardware vendor, such systems should provide core capabilities (extensible libraries, languages, performance analysis and prediction, interactive graphical visualization, automatic checkpointing, and so on) on which problem-oriented abstraction layers can be constructed.

Verification and validation of scientific codes has also been identified as a significant bottleneck [49]. If programs are written at a higher level of abstraction, expressed in terms closer to the problem's mathematics and structure, verification and validation is easier than when dealing with low-level implementation details, as the experiment in code rewriting showed (Figure 9).

5.1.3 Portability layer

Significant development effort is expended in porting applications to new machines. Because HPC applications can outlive multiple machine generations, they are often ported several times. To the extent that the application has been optimized to its current platform, that optimization must be undone, and the application must be optimized on the new machine.

This problem can be addressed by providing a *portability layer*: an abstract machine layer that can run on different hardware, provide a uniform target for translators, and automate efficient mapping to the underlying machine. For example, the Hero approach is to provide a portable, intermediate abstract machine layer (as is done for the Java programming language [17]) optimized across a family of similar (single address space) hardware implementations.

5.1.4 Programming language support

Stakeholder perceptions differ on the requirements for programming language support. In some cases, there is a strong desire for platforms that support established languages, in particular Fortran and C++ with MPI. However, others feel that existing languages do not offer the capabilities to address language-related productivity issues (for example, the capability to provide adequate abstraction or automation). These stakeholders (including DARPA) are currently supporting several efforts to develop new parallel programming languages, but it is clear that a range of language support is needed in the foreseeable future (Section 3.10).

Specific requirements for established languages (Section 3.10.1) include backward compatibility as well as execution efficiency for applications written with MPI, OpenMP and MPI I/O. For the Hero architecture, this requirement was interpreted to mean that legacy applications written for distributed memory machines had to run with acceptable efficiency when mapped to Hero's memory architecture.

Emerging languages such as the PGAS family (Section 3.10.2) also had to be supported; these are not yet in widespread use, but consensus in much of the community suggests that this is the near-term trend. Sites differ in pace and timing when adopting these new languages, and effective interoperation with the older languages is understood to be a significant requirement for incremental adoption.

Finally, there are efforts to develop new parallel programming languages that provide both a more effective parallel programming paradigm and more efficient implementation on new hardware architectures (Section 3.10.3). Each of the HPCS Phase II vendors has been developing some form of new programming language. Sun's experimental language Fortress [2] (Section 12.4) is strongly focused on abstraction and on language extensions tuned for particular kinds of applications.

Some studies also suggest that significant productivity gains can be accomplished with better use of existing programming languages. This includes experimental rewriting of a standard benchmark into a high-programmability style using Fortran 90, which produced code that ran about half as fast using currently available language support (Section 3.7). It is likely that much of this loss could be regained through improved optimization techniques. Further, the effects on human productivity (for example, in maintenance or porting) may outweigh losses in runtime efficiency. None of this can be achieved, however, without increased investment in compilers and guarantees that high quality support will be widely available well into the future (a tools issue, discussed in the next section).

5.1.5 HPC tools and libraries

Although much of the development of HPC-specific tools and libraries may be conducted by third parties, their availability is a critical requirement if new supercomputer platforms are to have a major and continuing impact on productivity. The availability of good HPC tools has unfortunately been trending in the opposite direction, creating a perception that tools represent a significant project risk (Section 3.8).

Tools that address key development areas include software configuration management tools, build tools, parallel debuggers, program monitors, profilers, and tools for detecting deadlocks and race conditions. Data visualizers are also needed to help interpret the output, and program state visualizers are important debugging aids.

Despite the potential of tools, current developers are often unwilling to embrace them, because they are unsure whether the tools will be available on future platforms. In some cases, this also results in tools being developed and maintained in-house at considerable effort and expense.

An assessment of stakeholders' tool needs suggests that a long-term tool strategy is required, based on collaborative development or open source, to ensure that tools can be ported to different machines even if the original developers have lost interest. A more complete analysis and set of recommendations appear elsewhere [62].

Ongoing improvement in libraries supporting parallel programming constructs, as well as specific application domains, is also needed. Libraries have been demonstrated to save significant amounts of development time in HPC domains by providing reusable solutions for recurring problems. The development environment should provide libraries for common parallel vector and matrix operations, parallel versions of common data structures, as well as flexible visualizations of these data structures. It should also provide capabilities supporting user extensions that enable development of organized classes of domain specific objects. This can be viewed as a steppingstone toward a more problem-oriented development environment expressed at a more suitable level of abstraction.

5.1.6 Continuous monitoring, measurement, and improvement

Developers should be able to continue identifying and removing productivity bottlenecks after system delivery. Productivity bottlenecks act as a set of constraints on software development. A new platform that satisfies these requirements will remove many of the constraints discussed in this paper. However, the theory of constraints tells us that removing one process constraint will likely expose other constraints that were previously masked [15] (much as fixing the weakest link in a chain exposes the next weakest link). Continuous productivity improvement requires identifying and addressing new constraints as they become manifest.

New platforms should support such continuous monitoring and improvement by offering support for collecting data on how individuals and teams of programmers spend their time on real development efforts. The studies mentioned in this paper (Section 3.5), as well as others in the HPCS program, used the monitoring tool Hackystat [24] to gather real-time data on developer activities. This work has shown that such tools can unobtrusively collect data needed to profile development activities by monitoring computer activities associated with programming, testing, and debugging. Such data can provide the basis for suggesting further productivity improvements such as better tools or programming methods. Additional research in this area is needed to understand exactly what data should be collected and how to facilitate its interpretation.

5.2 Requirements for performance

Performance is the *raison d'être* for supercomputers. Further, ever-increasing performance allows ever-larger problems to be solved in the available time. Performance can even be an all-or-nothing criterion. For example, the results of a weather simulation are useless if its predictions arrive after the fact. Performance can thus influence productivity in two ways: first, by achieving a computational result quickly. If the result is achieved too late because of hardware or software that is too slow, the value of this result may be greatly reduced, perhaps to zero.

The second contribution to productivity comes from making scientists more productive. Assuming that a parallel program has been developed and is being used in production mode, if scientists get their results back more quickly, they spend less time waiting or doing less-productive work. It is well established that interruptions in work are disruptive; a faster computer can return results sooner and shorten these disruptions, so work on the next problem can start sooner. Thus, a faster computer produces results of higher value and helps scientists use their own time more productively, as well.

For supercomputer applications, performance is much more than just a simple measure of microprocessor floating point operations per second (FLOPS). DARPA has defined a set of HPC challenge benchmarks [36] that include measurements of FLOPS, memory bandwidth, bisection bandwidth, and remote memory access latency. In addition, with the increase in the cost of energy, measurements such as FLOPS per Watt have become very important in evaluating operations cost.

5.3 Requirements for robustness

Due to the large number of hardware components in a petascale system, the likelihood of a component failure is high. For example, reliability models for the Hero hardware, which consists largely of commodity components, predict the occurrence of several hardware faults *per day*. Since HPC applications typically run for hours, days, or even longer, multiple component failures during execution must be anticipated. Clearly, shutting down the entire system, repairing it, and restarting the interrupted applications causes unacceptably low productivity.

Next-generation systems must be designed to continue computation in the face of component failure, a goal expressed as:

Compute Correctly Through Failure

First, in the event of component failures, unaffected components should continue to provide service, even as total system capacity and performance diminishes. Repair or replacement of failed components should be possible while the rest of the machine continues operating. Applications using the failed components should continue unabated. Second, applications using the failed component should not lose results computed prior to the failure. Many failures can be made transparent to the application by using redundancy or restarting failed processes. To protect against failures that cannot be made transparent, it must be possible to collect and store frequent checkpoints for applications at runtime, then restore an application to a safe state and resume computing, without compromising program execution time or overall system throughput.

For example, Hero is designed to provide an automatic checkpointing facility, allowing applications to be rolled back and restarted at a checkpoint. It also provides facilities for monitoring the health of all components and proactively migrating computations away from components that are in danger of failing. The benefits of such approaches are discussed elsewhere [63] [65].

By providing automated checkpointing, migration, and continued operation through failures, developers can be relieved from programming their own robustness features. The resulting codes will be smaller and less expensive to maintain.

5.4 Requirements for administration

Scheduling jobs of varying sizes and priorities on a supercomputer is complex, and it is becoming even more challenging as machines such as Hero grow to petascale (Section 3.9). This task should be automated as far as possible because automation reduces administration costs and errors. Resource monitors and visualization should provide administrators, scientists, and programmers with up-to-date status and usage information at several levels of detail.

Applications also should not waste resources. Many parallel applications exhibit a point of diminishing returns, where adding processors causes poor machine utilization or even an increase in computation time. In that case, it is more productive to allocate fewer resources and allow applications to run concurrently. This implies that the system should provide automated monitors for detecting and mitigating resource waste. It must also be possible to change the job mix quickly (for example, between large numbers of smaller jobs and fewer large jobs) as a response to project demands and priorities. Minimizing delay and idle resources while changing the job mix requires that the system checkpoint and suspend jobs quickly, and restart them without significant loss of progress.

Efficient use of a supercomputer system also means that the machine must be configurable to the intended workload. Underutilized components are unproductive. The number of processors, memory per processor, number of I/O nodes, and number of various types of storage devices should be configurable on an as-needed basis. This is because data analysis and computation-intensive applications differ in their processing, communication, and storage requirements.

In particular, to support many different types of workloads at national laboratories and other supercomputer centers, the machine must be able to transition seamlessly between capability and capacity modes (Sections 2.3 and 3.9). Current supercomputers often run only in one or the other; for example, the newest, most capable machine often runs in capability mode while older machines run mainly in capacity mode. For a system the size of Hero, few applications are computationally intense enough to consume all its resources, so the machine must be able to satisfy other computing demands when the few large applications are not running.

6 From requirements to design

The requirements set forth in the previous section call for an innovative system architecture and design. The challenge to innovate requires the synergy that results from combining individuals with expertise in multiple areas on every functional design team and having key design leaders participate in all design phases. This meant that the consequences of each design decision were evaluated from many perspectives and that the full benefit could be extracted from any particular innovation. For example, by including the lead designer for reliability, availability, and serviceability (RAS) in all key hardware and software decisions, and by enabling direct participation of hardware and software leads in the RAS team, RAS considerations permeated the entire design at every level. This ensured that the design requirements were met and key RAS features such as hardware virtualization could be supported in the most efficient possible way, using both hardware and software design components.

Likewise, the direct participation of all design leaders for both hardware and system software features enabled the creation of a flexible and powerful global shared address space programming model, using both hardware and software components, while avoiding the excessive costs associated with global hardware cache coherence. The flexibility to implement any required feature in the most efficient manner by combining the best mechanisms from hardware and software technologies was crucial to the production of a revolutionary new design, not just a simple evolution of current designs.

Section 6.1 describes the design process that allowed the team to break through organizational barriers and avoid obstacles to innovation. Section 6.2 illustrates the results of the process at a high level—how productivity goals were supported by innovations at all layers of the system, the subject of sections 8-12. Section 6.3 contains a detailed mapping of productivity goals to design requirements and of design requirements to enabling innovations. These mappings demonstrate the flow of productivity goals into the design. Later in this document, Section 13 returns to the theme of interdisciplinary design and provides a specific example of how the design process enabled the tradeoffs needed to create a feasible design in the presence of seemingly intractable requirements.

6.1 System Exploration Model

Sun developed a collaborative design approach for the HPCS Phase II program: the System Exploration Model. In contrast to other design approaches, which assemble existing or favored component architectures, the System Exploration Model started with the requirements and did not constrain the system architecture required to fulfill those requirements in any way. A cross-functional team of component technology experts worked together to understand what is possible and used simulation and analysis tools to evaluate the possibilities. The output of the process was an optimized system design, jointly proposed and agreed to by the component technology experts. Sun's System Exploration Model is fundamentally a concurrent process, in contrast to sequential models where separately optimized components are merged into system-level designs and then packaged into products.

The central management construct in the System Exploration Model is the system architecture team, which provides a central unifying role during the system exploration process. This team is cross-functional; it includes thought leaders from component teams and becomes an interdisciplinary group of experts, addressing global objectives by stepping through learning stages that include:

- Prior art investigation
- Workload characterization
- Assessment of key market and technology limiters and opportunities
- Technology trend analysis
- Component and system modeling
- Design space exploration
- Generation of straw man alternatives
- Generation of system reference designs
- Component, subsystem, and system-level prototyping

The system exploration process is summarized in Figure 12. Component area experts begin by explaining the technology trends in their areas, rather than specific components or design attributes. For example, a microprocessor architect describes technology trends, such as multithreading, that allow microprocessors to continue on the trend prescribed by Moore's law, rather than describing the operating frequency and floating-point pipeline of current microprocessors.

Technology trends feed a sequence of progressively more detailed system models that permit the system architecture team to identify areas that may require innovation beyond current technology trends. For example, interconnect trends did not provide the internode bandwidth required to meet performance goals, so the innovations described in Section 9 were necessary. Models are often just simple component performance models, which would not have allowed the satisfaction of a wide range of productivity goals. Therefore, the HPCS Phase II program measurement and analysis frameworks were developed for end-to-end performance, productivity, robustness, and security projection, as well as system-level cost, power, thermal, and mechanical models. The modeling approach was hierarchical, based on detailed component-simulation models, which were then factored into system-level analytic and simulation models. These analyses became progressively more accurate as prototyping and implementation progressed.

In addition to system models, a detailed understanding of the workload characteristics is essential. A system optimized for a particular workload such as random traffic may perform poorly under other workloads that have hot spots. The DARPA challenge benchmarks [36] provided a set of workloads that stressed the system design in multiple dimensions, including peak processor performance, memory access latency, and bisection bandwidth. The Hero system architecture had to satisfy the requirements of all workloads, rather than just being optimized for a single characteristic.

With such a challenging set of requirements, the system architecture exploration shown in the center of Figure 12 was an iterative process, informed by workload simulation of proposed system alternatives. Component teams could make proposals, but the interplay of proposals with other components had to be evaluated in the context of an entire system to create an optimal system design, rather than a set of optimal component designs. For example, the memory subsystem had to satisfy system checkpoint requirements as well as microprocessor requirements for memory bandwidth and latency. This concept is somewhat foreign to product teams accustomed to spending most of their time in detailed design rather than iterative system architecture, and management had to work hard to ensure that the architecture team remained focused on system optimization.

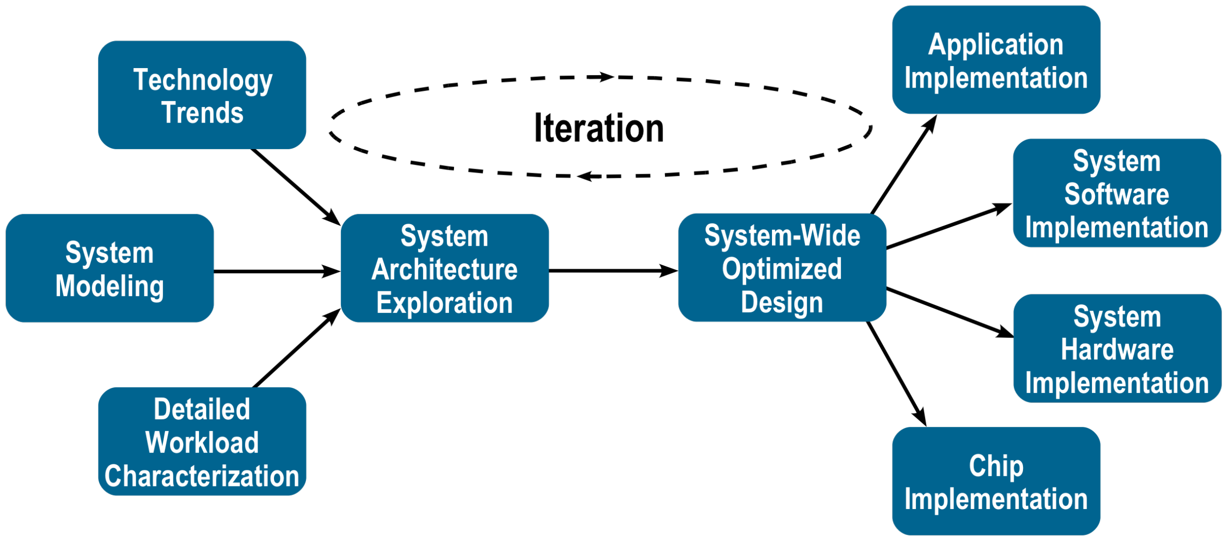

Technology Trends
System Modeling
Detailed Workload Characterization
System Architecture Exploration
Iteration
System-Wide Optimized Design
Application Implementation
System Software Implementation
System Hardware Implementation
Chip Implementation

Figure 12: System Exploration Model

System-level metrics visibility keeps everything in perspective. The system exploration process identifies the system-level contribution of each component, the critical factors and risks, and the system metrics comparison of alternative system designs. It thus enables the derivation of optimal reference designs for each market segment. These quantified reference designs allow the product definition team to assess and define appropriate products as the program progresses. Ongoing visibility into system and component architectures provides program management with the basis for decision-making and flexibility in adjusting areas of focus and investment.

The System Exploration Model (Figure 12) represents a pivot point in Hero's design and this report. The productivity research and analysis described in Sections 2-5 produces design requirements, and the innovative designs described in Sections 7-13 satisfy those requirements. In practice, of course, the division between requirements and design is not nearly so cut and dried. There were many iterations and compromises that are not adequately described in this document, but whose existence is understood by anyone who has worked on a complex project. However, throughout the entire iterative process, the emphasis was on creating an entire system that achieved a 10x productivity improvement.

6.2 Enabling innovations

The Hero design rests firmly upon five technical foundations, described in more detail in following sections. At the processor level, extensive chip multithreading (CMT) provided an efficient way to keep all hardware resources busy performing useful computation (Section 8). At the interconnect level (Section 9), a combination of proximity communication and silicon photonics enabled key hardware and software breakthroughs by providing new levels of bandwidth at low latency. Also at the interconnect level, the hardware and software supporting the global address space enabled not only simpler programming models and new system software and virtualization strategies, but also delivered simpler hardware designs for nodes (Section 10). At the system software level, creating the abstraction of a single, partially coherent, shared memory machine using virtualization allowed simpler OS designs while opening new opportunities for high-level programming (Section 11). Finally, at the tools and languages levels, technologies in the Fortress language [2], together with associated parallel tools, permitted full and efficient utilization of all the other innovations by users of the system (Section 12).

These technical innovations were the result of the process outlined in the previous section. The linkage from overarching productivity goals to design requirements to technical innovations is shown in Figure 13 with details of the links in Section 6.3. These linkages provide traceability from the design back to the original productivity goals.

The left side of Figure 13 shows the linkage from productivity goals to design requirements, while the right side shows the linkage from design requirements to enabling innovations in the design. For example, the design requirement labeled “Compute Correctly Through Failure” is a paramount requirement for satisfying the robustness goal, but it also serves the following goals:

- *Programmability*: Eliminating the need for defensive programming improves application development productivity.
- *Performance*: Minimizing fault tolerance overhead and recovery time from failures significantly increases how much time is spent performing useful computation.
- *Administration*: Automating failure handling and recovery improves administrative productivity and reliability.

As shown on the right side of Figure 13, several design innovations enable Compute Correctly Through Failure. The virtualization-based software stack allows reallocation of compute resources for efficient recovery; the global address space allows reallocation of memory resources for efficient recovery; and proximity communication and silicon photonics provide sufficient communication bandwidth to support system checkpointing effectively.

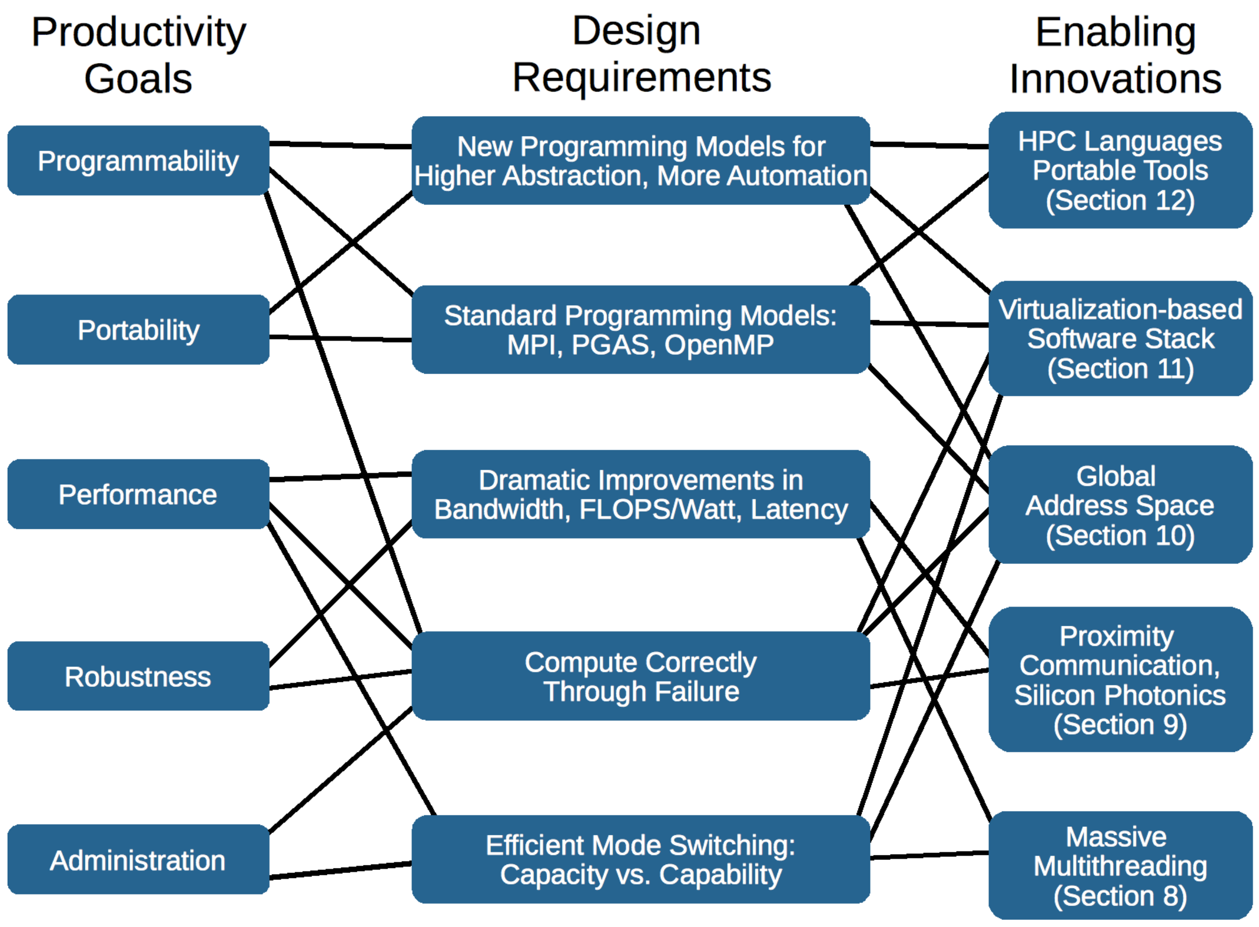

Productivity Goals
Design Requirements
Enabling Innovations
Programmability
Portability
Performance
Robustness
Administration
New Programming Models for Higher Abstraction, More Automation
Standard Programming Models: MPI, PGAS, OpenMP
Dramatic Improvements in Bandwidth, FLOPS/Watt, Latency
Compute Correctly Through Failure
Efficient Mode Switching: Capacity vs. Capability
HPC Languages Portable Tools (Section 12)
Virtualization-based Software Stack (Section 11)
Global Address Space (Section 10)
Proximity Communication, Silicon Photonics (Section 9)
Massive Multithreading (Section 8)

Figure 13: Innovations that enable achievement of productivity goals

6.3 Requirements traceability

Figure 13 illustrates high-level linkages among productivity goals, design requirements, and enabling innovations. The following tables explain the rationale for these linkages, which are key to requirements traceability. Table 8 traces productivity goals to design requirements, and Table 9 traces design requirements to enabling innovations. Although the traceability is described as flowing from goals to innovations, the process to develop that flow is highly iterative as described in Section 6.1.

Table 8: Productivity goals to design requirements traceability

Productivity Goals	Design Requirements	Rationale
Programmability	New Programming Models	Address key bottlenecks around expertise and effort with programming models that are more abstract (less non-domain knowledge needed) and embody more automation (reducing level of effort).
Programmability	Standard Programming Models	There is a huge legacy of HPC codes and skills, and the community will evolve slowly; there must be excellent support for legacy technologies as well as transitional steps toward new programming models such as OpenMP and PGAS.
Programmability	Compute Correctly Through Failure	Guarding against system failure imposes a burden on application development in the expertise and effort needed to implement application checkpointing; computing correctly through failure eliminates this need in legacy and new programming models.
Portability	New Programming Models	The abstraction of new programming models by definition excludes machine-specific programming requirements.
Portability	Standard Programming Models	Even within legacy programming models, a more abstract execution model reduces the need for machine-specific optimizations.
Performance	Dramatic Improvements in Bandwidth, FLOP/Watt, Latency (Raw Performance)	Internode bandwidth and latency are the key performance in current distributed memory machines. Energy costs are becoming a primary limit to scalability of the largest machines.
Performance	Compute Correctly Through Failure (Effective Performance)	The amount of time performing useful computation is significantly increased by integrating system fault tolerance, minimizing both the overhead costs and recovery time from failures.
Performance	Efficient Mode Switching (Effective Performance)	System checkpointing and hardware resource virtualization make it possible for jobs to be suspended, relocated, and reprovisioned efficiently; this makes possible much more efficient job management and increases total system throughput.
Robustness	Dramatic Improvements in Bandwidth, FLOP/Watt, Latency	Bandwidth and latency improvements, combined with hardware virtualization technologies, enable dynamic node sparing for hardware fault recovery.
Robustness	Compute Correctly Through Failure	Robustness in the face of inevitable hardware failure has become a critical productivity problem for massive high-end systems; the ability to compute correctly and continuously through failure is a requirement for increasing the scale of these systems.
Administration	Compute Correctly Through Failure	Automated failure handling and recovery reduces the human cost of administration.
Administration	Efficient Mode Switching	Allows more flexibility in machine configuration and job management.

Table 9: Design requirements to enabling innovations traceability

Design Requirements	Enabling Innovations	Roles
New Programming Models	HPC Languages; Portable Tools	A new generation of languages embodies the strategies of abstraction and automation.
New Programming Models	Virtualization-Based Software Stack	Virtualization provides a higher level of abstraction at the system software layer, making it much easier to build modern programming languages and tools.
New Programming Models	Global Address Space	A single address space is a key part of system virtualization, supporting simplified memory abstraction.
Standard Programming Models	HPC Languages; Portable Tools	Portable tools make it possible to support legacy codes, skills, and practices.
Standard Programming Models	Virtualization-Based Software Stack	Virtualization enables support of OpenMP on multi-node systems and efficient support of PGAS languages.
Standard Programming Models	Global Address Space	Even within legacy programming models, a more abstract execution model reduces the need for machine-specific optimizations
Dramatic Improvements in Bandwidth, FLOP/Watt, Latency	Proximity; Silicon Photonics	Proximity and silicon photonics address key hardware bottlenecks in providing high-bandwidth, low-latency, energy-efficient internode communication.
Dramatic Improvements in Bandwidth, FLOP/Watt, Latency	Massive Multithreading	Reduces the intra-node cost of computation in energy efficiency and latency.
Compute Correctly Through Failure	Virtualization-based Software Stack	Allows reallocation of compute resources for efficient recovery.
Compute Correctly Through Failure	Global Address Space	Allows reallocation of memory resources for efficient recovery.
Compute Correctly Through Failure	Proximity; Silicon Photonics	Provides the communication bandwidth needed to effectively support system checkpointing.
Efficient Mode Switching	Virtualization-Based Software Stack	Allows reallocation of compute resources for efficient suspension and resumption.
Efficient Mode Switching	Global Address Space	Allows reallocation of memory resources for efficient suspension and resumption.

7 Design overview

This section provides an overview of the innovative design that was developed, based on the productivity research described in previous sections. Sections 8–12 provide more detail of hardware and software inventions that enable the design. Specifically:

- Section 8: Massive chip multithreading, a processor architecture that provides the ability to saturate many on-chip cores, even in the presence of irregular high-latency memory operations.
- Section 9: Proximity communication and silicon photonics, technical innovations that provide the massive system bandwidth required to support the productivity of a global shared-memory model.
- Section 10: A system coherence model that supports legacy applications and enables new highly productive languages and programming models.
- Section 11: System software that provides a robust computing environment across thousands of nodes.
- Section 12: Development support for highly productive programming and Fortress, a new programming language that provides a higher level of abstraction more closely resembling scientific and mathematical expression.

7.1 Hardware overview

Figure 14 depicts a high-level view of the Hero system hardware architecture, which consists of a large number of interconnected compute and I/O nodes supporting a global shared-memory address space. Memory within each compute nodes is fully coherent; global shared memory with managed coherence (Section 10.1) is supported across all nodes in the system. Both the compute and I/O nodes are based on standard commercial components augmented with optical interconnect and special hardware, a scalability interface (SIF) that assists with remote memory operations. Compute nodes have extra memory bandwidth for higher compute performance, while I/O nodes provide external storage and network connectivity, including an interface to specialized analysis functions such as visualization.

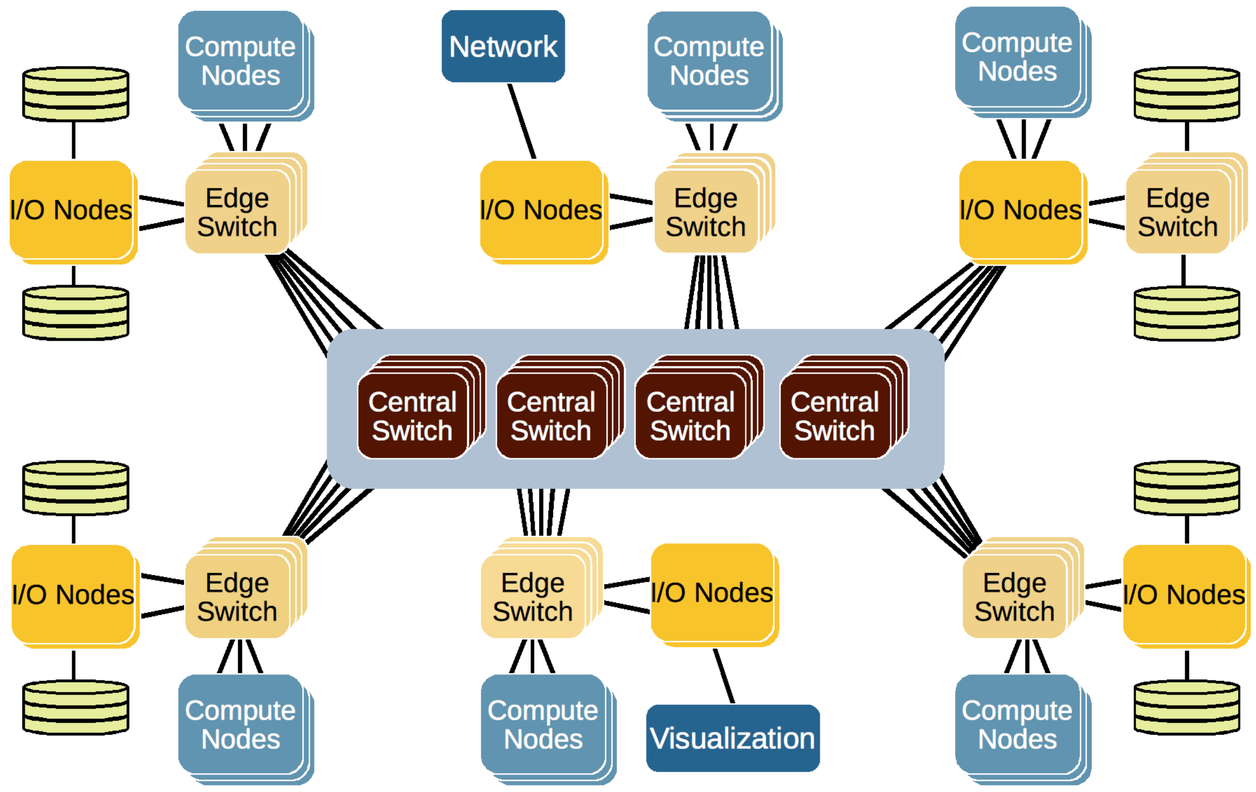


Figure 14: Hero hardware architecture

Compute and I/O nodes are interconnected by a multi-stage switching fabric, for example, a three-stage Clos network of Hero Switches.[18] The edge switches and central switches shown in Figure 14 are both physically Hero Switches; they differ only in topology. An example path in a three-stage network is shown in Figure 15. A packet originating in the source node for a global load/store operation uses the SIF to manage remote memory access. The example packet traverses four optical links and three switches on its way to the destination node. Silicon photonics connects nodes and switches optically; proximity communication connects elements within the switch. These innovative technologies provide the low latency, high-bandwidth switch fabric necessary for high-performance remote memory operations among thousands of nodes.

[18] Clos networks [11] are a popular HPC switching fabric due to advantages such as high bisection bandwidth, low switching delays, high reliability, low-complexity deadlock avoidance, and efficient routing. The number of stages is the maximum number of switches that must be traversed to get from one node to any other node. Two nodes connected to the same switch need only traverse that switch, but a typical path among nodes shown in Figure 15 traverses two edge switches and a central switch. For a larger network, another set of edge switches could be inserted before the central switches to make a five-stage network. Technically, the network described in this report is a folded Clos or fat-tree network [34], because the same edge switches are used for traffic entering and exiting the central switch.

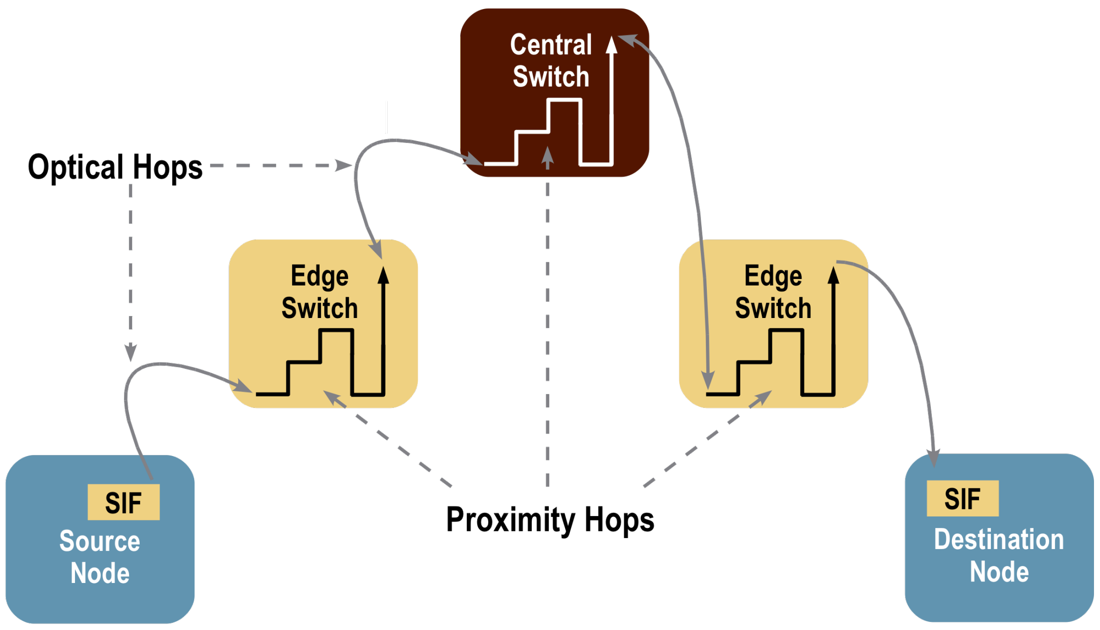


Figure 15: Path from source to destination node

7.2 Software overview

Hero's software architecture, shown in Figure 16, supports highly productive programming by leveraging global shared memory over a high-bandwidth, low-latency switch fabric, and by introducing a new multi-node execution environment, called a *SuperZone*, which is built on extensions to the OpenSolaris™ Zones technologies. Acting in concert, Hero hardware and software enable a new execution model that provides a global address space and coherence features that free programmers from low-level memory management. By eliminating the necessity to manage memory locality, Hero provides revolutionary improvements in programmability and supports all existing and emerging HPC programming models, including OpenMP, Autoparallelized Fortran, PGAS, and MPI. Additional productivity gains are provided by extensive fault tolerance features added throughout the Hero software stack and the advanced administrative model supported by the Administrative Environment. These features combine with the Hero File System to support the massive external bandwidth provided by Hero hardware.

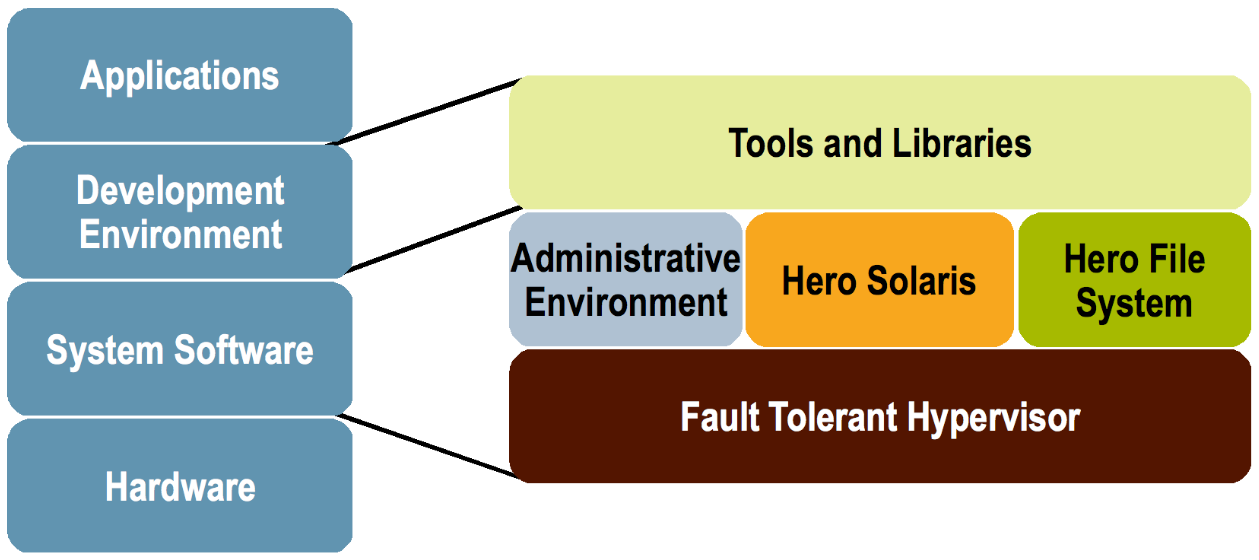


Figure 16: Hero software architecture

The Hero software stack comprises the following, described in more detail in later sections:

- *Fault Tolerant Hypervisor*: Based on the current Sun4v hypervisor [45] with extensions for scaling and fault tolerance. It provides full device virtualization and support for automated checkpoint and restart.
- *Hero Solaris™*: Based on the current OpenSolaris operating system [45] with extensions for multi-node semantics, multi-node execution, predictive self-healing, and automated checkpoint and restart. It supports multiple file systems (Hero File System, Lustre, ZFS, NFS, and pNFS) and a broad array of networking options.
- *Hero File System*: An object-based parallel file system, implementing the T10 OSD standard with NFS and pNFS support.
- *Tools and Libraries*: C, C++, Fortran, and UPC compilers, augmented with debugging and performance monitoring tools and supported by libraries tailored to the global shared-memory environment of Hero.
- *Administrative Environment*: Based on the current OpenSolaris management suite, extended to handle global shared memory, larger node counts and Hero Switch fabric.

For application development Sun created Fortress: a revolutionary new HPC programming language with a higher level of programming abstraction. Fortress is designed for high-performance computing with high programmability, supporting new features such as transactions, specification of locality, implicit parallel computation, and the ability to integrate third-party libraries into Fortress as if they were designed integrally with the language.

8 Compute node: massive chip multithreading (CMT)

Based on chip multithreading (CMT), described in Section 8.1, Hero's compute node architecture keeps hardware resources busy even in the presence of stalls, high-latency memory operations, and other processing irregularities. It provides dramatic performance improvements for HPC codes with long cross-system memory latencies. Even more important for productivity, it frees programmers from hand-tuning applications to the machine's exact processor and memory architecture each time the application ports to a new platform. As described in Section 8.2, the compute node also provides hardware infrastructure for the global address space and execution model (Section 10). Finally, as described in Section 8.3, the use of commercial components minimizes cost.

8.1 Chip multithreading (CMT)

By the beginning of the HPCS program, it had become clear that improvements in microprocessor clock frequency, longer pipelines, and larger cache sizes would be insufficient for microprocessor performance to keep up with Moore's Law. As a result, a new design approach that took advantage of increasing silicon transistor density to place multiple microcores on a single die was developed. This industry—chip multiprocessor (CMP) microprocessors—is made possible by semiconductor process improvements, and allows microprocessor performance to keep up with Moore's Law.

At the outset of the Phase II program Sun led the industry in massive chip multithreading (CMT) microprocessors. CMT microprocessors combine CMP microprocessors with multithreading (MT), in which hardware resources are dynamically allocated to whatever threads need them at the time. During the program, Sun shipped the Niagara processor with eight cores supporting four threads each. Sun has maintained the industry lead since the end of Phase II and is now shipping the Niagara 2 processor with eight cores supporting eight threads each [33].

Hero is based on a variant of the Rock microprocessor [32] [58], which implements a third-generation checkpoint-based CMT architecture. In this context, a checkpoint is a copy of the full microprocessor register file. A checkpoint is stored when the executing thread encounters an unresolved data dependency due to a previous cache miss. Since the state of execution is preserved, the thread can continue to speculatively "execute ahead" by committing instructions in a different register file copy. In *execute-ahead mode*, the processor continues beyond operations with unknown operands (which are waiting for memory). It speculatively retires the instructions that can be executed, while it defers instructions with unknown operands for execution later. When memory operations return, the processor goes back and executes the deferred instructions. When all deferred instructions have been executed successfully, the speculative portion is committed and execution proceeds normally until a new execute-ahead opportunity arises. If the speculation fails, execution is rolled back to the state preserved in the checkpoint. Execution then resumes, now with many of the outstanding memory operands present in caches. If the store queue fills up during speculation, the Rock processor still continues the speculative execution to launch as many memory operations (prefetches) as possible, but without the ability to commit these speculative instructions later. This noncommitting speculation is referred to as *scout execution*. It is even possible to do *simultaneous scout execution*, in which execution proceeds simultaneously at two different points of the same thread [58].

Rock's checkpoint-based architecture allowed the development of an innovative out-of-order instruction pipeline. Instructions are not only executed out of order but also retired out of order. This resolves issues associated with standard out-of-order pipelines where instructions must be reordered prior to retirement, which requires complex content-addressable memories (CAMs) to hold out-of-order instructions. Because it is not limited by the size and power requirements for complex CAM structures, Rock's pipeline enables much deeper speculation than conventional out-of-order architectures: thousands of instructions rather than 32 or 64, which improves single thread performance and CMT throughput.

Another innovation in Rock's checkpoint-based architecture is *transactional memory* [59]: the ability to perform a set of instructions as a single atomic unit. This means that either all instructions in the set (transaction) complete fully, or none of them complete. New instructions such as checkpoint and commit, as well as enhanced micro-architecture structures, enable the execution of transactions without expensive synchronization instructions. The checkpointing mechanism is used to checkpoint a thread's state prior to starting a transaction and to restore it in case of transaction failure.

Transactional memory enables multiple threads to simultaneously enter a critical code section and allows programmers to replace complex locks in applications with transactions. This greatly enhances parallel programming productivity because the system handles all low-level concurrency control issues and synchronizes concurrent access to shared memory by multiple threads.

The use of CMT microprocessors strongly influenced the system design, because massive multithreading:

- Uses all processor resources efficiently by providing multiple threads that can be freely interchanged on a core, enabling excellent performance/chip and performance/watt.
- Improves performance by dedicating threads to events such as asynchronous system events and direct message handling, which avoids costly context switches and interrupt handling.
- Provides concurrency control, making sure that the right operations are taking place at the right time, without interference or disruption, at high performance.
- Provides the infrastructure that permits a novel software technique to minimize application jitter (Section 11.3.4).
- Tolerates irregular application behavior (Section 8.1.1).
- Allows the offloading of maintenance and I/O functions (Section 8.1.2) because the microprocessor hardware can be fully utilized running applications.

8.1.1 Irregular application behavior

To maximize performance, microprocessor hardware must be kept fully utilized. Stalls for context switches, cache misses, disk access, and the like significantly decrease performance. To keep the microprocessor busy when a stall occurs, CMT threads can be used both for speculation on a stalled task (Section 8.1) and for different tasks. Massive multithreading makes it possible to saturate pipelines even in the presence of irregular, high-latency memory operations. It therefore tolerates irregular application behavior and long cross-system memory latencies, meaning that programmers do not have to structure their programs to avoid these issues, which has a potentially profound effect on application programmer productivity.

Highly optimized HPC applications are carefully tuned to squeeze the last usable FLOP out of a pipeline, regularize memory access patterns, maximize locality, and carefully structure communication patterns to avoid latency. This improves application performance on current supercomputers optimized for peak FLOPS, but the programming required for manual tuning can grow the code's size by an order of magnitude [35]. Massive multithreading supports efficient execution of applications with irregular behavior, enabling higher productivity programming by eliminating manual code optimization. It also minimizes the effort required in the software stack to mitigate irregular application behavior and the tools required to help programmers regularize application behavior.

8.1.2 Offloading maintenance and I/O functions

Maintenance and I/O processing could have been performed by compute microprocessors, perhaps using separate threads. However, massive multithreading provides a way to fully utilize CPU hardware for application processing, so maintenance and I/O functions were offloaded to other hardware. Application processing is performed on compute nodes, I/O functions are performed on I/O nodes and special application-specific integrated circuits (ASICs), and service processors are incorporated into nodes to perform maintenance functions.

An I/O node (see Figure 17) similar to planned commercial products performs I/O functions such as disk and network access. The use of commercial components reduces costs and provides a familiar development platform for I/O software developers. Remote memory access is performed by an ASIC specifically designed for that purpose (Section 8.2).

A dedicated service processor using an out-of-band network performs maintenance processing. The service processor network provides diagnostics and control to help the system detect, isolate, and recover from failures. A petascale system contains so much hardware and software that failures are frequent. This creates requirements for checkpointing and virtualization of hardware resources to permit the system to compute correctly through failures.

8.2 Hardware support for global shared memory

At the beginning of the HPCS Phase II program, memory coherence was thought to be an HPC application requirement, necessary to improve productivity by making the system easier to program. Although providing hardware memory coherence is relatively simple within a single compute node, hardware memory coherence across a large system demands great hardware complexity. Also, the non-uniform memory access (NUMA)[19] nature of petascale computing would have made it perform very poorly. Due to the infeasibility of a purely hardware implementation, the team revisited the requirements; in particular, the actual coherence

[19] Memory accesses on a petascale computer are inherently non-uniform because messages from remote memory take a relatively long transit time, even at the speed of light.

requirements of applications (Sections 10.1 and 13). The team's research demonstrated that global coherence is necessary only at certain points in a typical HPC application; for example, at the start and end of parallel regions. This central requirement became a fundamental basis of the design: global load/store semantics supported by a shared global address space with coherence barriers, known as global shared memory with managed coherence.

The hardware requirements for global shared memory with managed coherence are much simpler than for full coherence. The Scalability Interface (SIF) ASIC was created to manage remote memory access and provide global addressability (Section 10.1). The SIF performs the function of a memory controller for all physical memory addresses that are not resident on the node where the CPU issuing the memory request resides. It forwards remote memory requests—including remote prefetch and remote compare and swap operations—that are performed by the node's processors or I/O devices. This requires translation of the address to a global memory address and creation of a transaction that is forwarded to the appropriate compute node. The SIF also services memory and I/O requests from remote processors. SIF functionality helps hide the latency penalty of random access fetches across the system interconnect and saturates the system interconnect bandwidth with random loads and stores.

The SIF handles remote memory requests by using internal address tables to forward each request to the node where the actual physical memory resides (see Section 10.2.1). The presence of these additional translation tables also permits the system to virtualize memory space by using virtual node IDs. This permits the system to remap physical memory from one node to another, after an application restart, for example. This mechanism enables graceful handling of node failures and enhances the system configuration flexibility. These features all contribute to the primary goal of improving productivity by supporting the requirements described in Section 5.

Active messages are an asynchronous mechanism in which a message is delivered to a remote node and then executed on a thread in the receiving node. Active messages were implemented for Hero because they allow instructions to be sent to the node with the data for execution, reducing data transfer and improving performance. They also improve MPI message performance by implementing a hardware mailbox facility. The SIF provides a user space interface to hardware that generates, receives, and locally dispatches active messages. It also implements support for buffering active messages, end-to-end flow control, and dispatch management to the processor.

8.3 Compute and I/O nodes

To minimize development and production cost, Hero's compute and I/O nodes are designed for similarity with Sun's standard commercial products. Hero components, such as microprocessors and memory controllers, are the latest-generation components used in commercial products. The mechanical, power, and cooling infrastructure is essentially identical to standard commercial nodes.

A high-level diagram of Hero's compute and I/O nodes appears in Figure 17. Each node is a standard cache-coherent symmetric multiprocessing (SMP) machine with a CMT microprocessor, shared L3 cache, memory controllers, and dual inline memory modules (DIMMs). Each node contains a SIF (Section 8.2) and a fiber optic interface for internode communication (Section 9). In order to accommodate local and remote memory bandwidth requirements for HPC applications, the Hero compute node has more memory controllers and internode communication capability than the I/O node. The Hero I/O node includes PCI Express (PCIe) network connectivity for the storage, network, and visualization interfaces shown in Figure 14 (on Page 66). The service processor is not shown in either node.

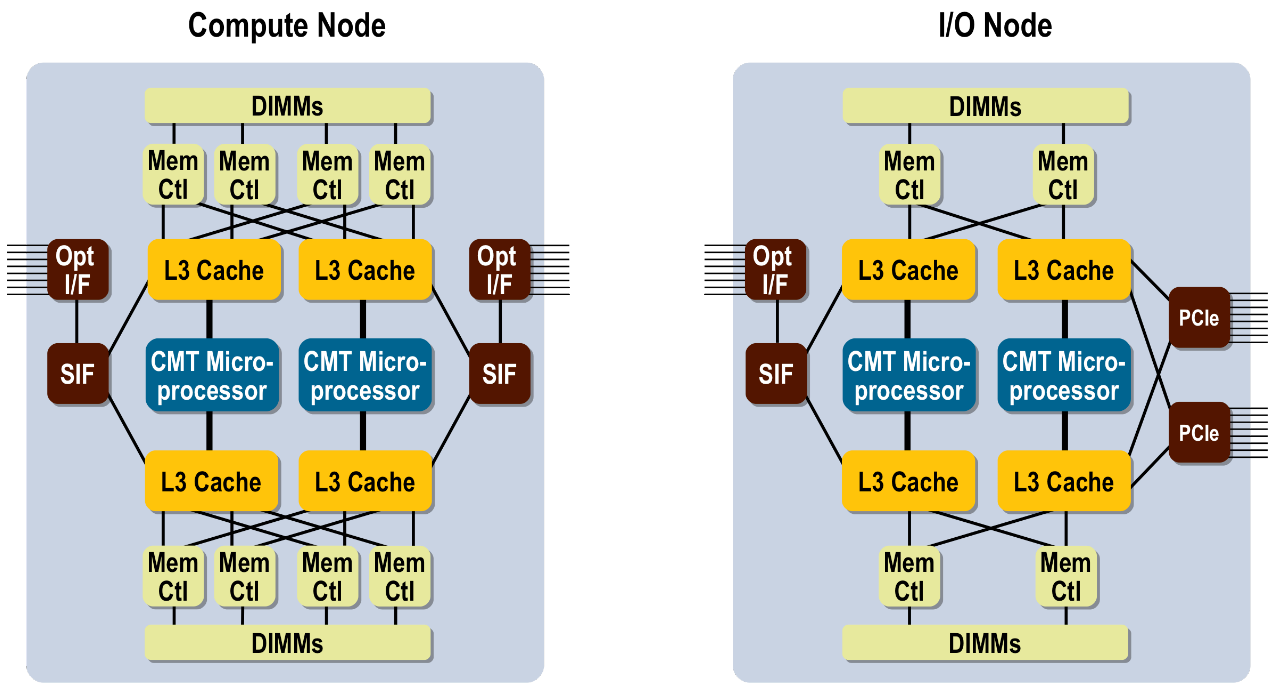


Figure 17: Hero nodes

9 Interconnect: proximity communication and silicon photonics

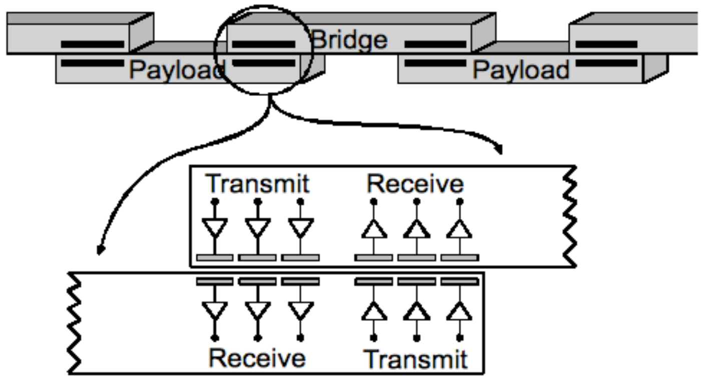
For the many HPC applications that can consume the compute resources of an entire petascale system (the so-called *capability* codes), fast node-to-node access is critical for performance, both for memory access and computational results. Even massively threaded microprocessors eventually stall when remote access latencies are large. Without fast remote access, programmers revert to hand-tuning applications to minimize remote communication (Section 3.10.1). The productivity benefits of a globally shared memory architecture are lost if remote access is so slow that it requires programmers to expend significant effort maximizing locality.

Fast remote access requires a high-bandwidth, low-latency interconnect among compute nodes. Low latency frees programmers from using programming models that are based on latency hiding. High bandwidth effectively contributes to low latency by reducing contention for network resources; it also provides the infrastructure necessary for system checkpoints, essential components of the RAS strategy.

As the HPCS Phase II program began, Sun was developing two new technologies for fast interconnect: proximity communication (Section 9.1) and wavelength division multiplexing (WDM) silicon photonics (Section 9.2). Both provide enormous bandwidth and low latency, which enable higher productivity execution and programming models (Section 10). Proximity communication and WDM silicon photonics are naturally complementary technologies: proximity communication provides high-bandwidth, on-module connectivity, and WDM silicon photonics provides high-bandwidth, off-module connectivity. Together, they make it possible to build very large systems with uniformly high bandwidth.

9.1 Proximity communication

Proximity communication [14] [21] uses capacitive coupling to enable low latency, high-bandwidth communication between pairs of neighboring chips. As shown in Figure 18, metal plates on separate chips create a chip-to-chip capacitor when the chips are placed in close, face-to-face proximity. The sending side drives one capacitor plate, inducing a small voltage swing on the receiving capacitor plate; there is no physical connection between the two chips. Because



Proximity communication sends signals between a face-up chip and a face-down chip using capacitive coupling. The face- up chips are payload chips, such as microprocessors, switches, and caches. In order to communication between two payload chips, a face-down chip, called a bridge, is placed on top of neighboring payload chips. A signal sent by a payload chip to a neighboring payload chip crosses two proximity hops and a bridge chip.

Figure 18: Proximity communication using capacitive coupling

capacitive coupling enables high signal density, proximity communication offers much higher performance and consumes significantly less power and area when compared to traditional interconnects that depend on direct physical contact, such as bonding wires, printed circuit board traces, and solder balls.

The advantages of this technology arise because the plates on each chip are small: 20 to 30 microns on a side. By comparison, typical C4 solder balls are spaced at least 200 microns apart. This density advantage dramatically increases chip-to-chip bandwidth for the same area and significantly reduces power cost compared to current serialized I/O technology. Alternately, the bandwidth advantage can be used to transmit data that is wide, parallel, and running at the chip's frequency, rather than narrow, serial, and overclocked. This avoids the latency, complexity, area, and power overhead of using serializer/deserializer (SerDes) circuits to communicate between chips, while providing the same or greater bandwidth than SerDes-based I/O at far lower area, energy, and cost.

Proximity communication presents a number of packaging challenges that were the subject of extensive investigation during the HPCS Phase II program. To send data reliably at full bandwidth, plates on the face-to-face proximity connections must be aligned in the X-, Y- and Z-dimensions within one-third of a plate's length [21], for example, eight microns for a 24x24-micron plate. They must remain within these bounds in the face of thermal expansion and vibration. The primary mechanical alignment mechanism is the use of sapphire microballs that rest in pyramidal pits on the bridge and payload chips, as shown in Figure 19. The pits are lithographically etched into the silicon during wafer manufacturing, and the balls are placed in the pits during assembly. The ball and pit mechanism, along with other mechanical innovations [57] [67], keeps chip alignment well within required tolerances. The inset in Figure 19 is a scanning electron micrograph picture of a sapphire microball embedded in an etched silicon pit fabricated by Sun.

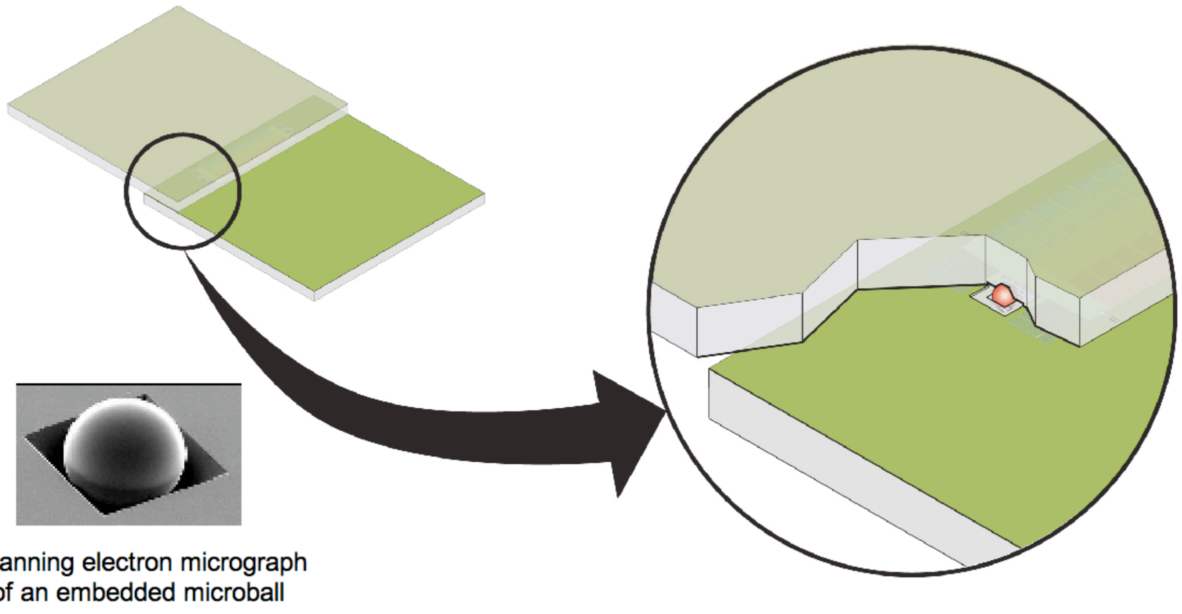


Figure 19: Balls and pits used as a proximity communication alignment mechanism

9.2 WDM silicon photonics

The essential internode interconnect technology is wavelength division multiplex (WDM) silicon photonics, which performs transmission, amplification, detection, modulation, and switching of multiple wavelengths (colors) of light on silicon. Each wavelength can carry a separate signal, allowing multiple data packets to traverse a single fiber concurrently and resulting in much higher bandwidth than current technology. The use of silicon photonics and fiber optic cables offers significant cost, latency, bandwidth, range, and reliability advantages over copper.

Several basic building blocks are needed to build optical transceiver cores into a CMOS die. These include fiber coupling interfaces, waveguides, wavelength multiplex/demultiplexers, optical modulators, and optical detectors. Luxtera, an HPCS Phase II partner, had these components and demonstrated the prototype shown in Figure 20 at the 2005 Supercomputing Conference (SC|05). This chip has fibers directly attached to the center and can communicate via proximity communication to other chips. Silicon photonics packaging is described in [51] and [67].

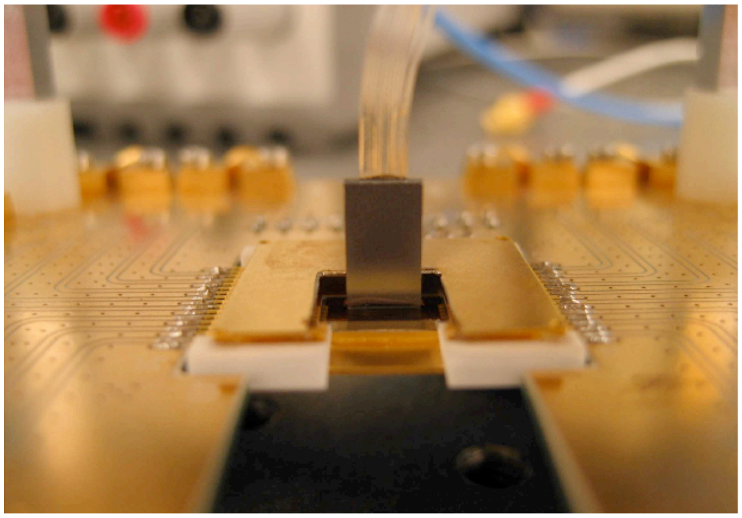


Figure 20: Packaged silicon photonics chip

9.3 Hero Switch

The Hero Switch incorporates both proximity communication and silicon photonics. It minimizes remote access latency, maximizes bisection bandwidth, and allows configurations to scale to thousands of nodes. It consists of a two-dimensional grid of switch chips, like the 4x4 grid shown in Figure 21, interconnected via proximity communication. Each switch port is implemented using a fiber optic pair connected to the switch chip using silicon photonics. Up to eight fiber optic pairs can attach to a switch chip. Each switch chip contains a small low-latency, cut-through routing switch that steers packets to one of its neighboring switch chips or to one of the optical fiber pairs connected to the switch chip. A data packet enters the switch via fiber optics and is routed along a minimum distance path in the proximity communication grid of switch chips to its fiber optic exit to another switch or node (see Figure 15). WDM silicon photonics provides high-bandwidth, low-latency communication among switches and nodes. Proximity communication provides high-bandwidth, low-latency communication within the switch.

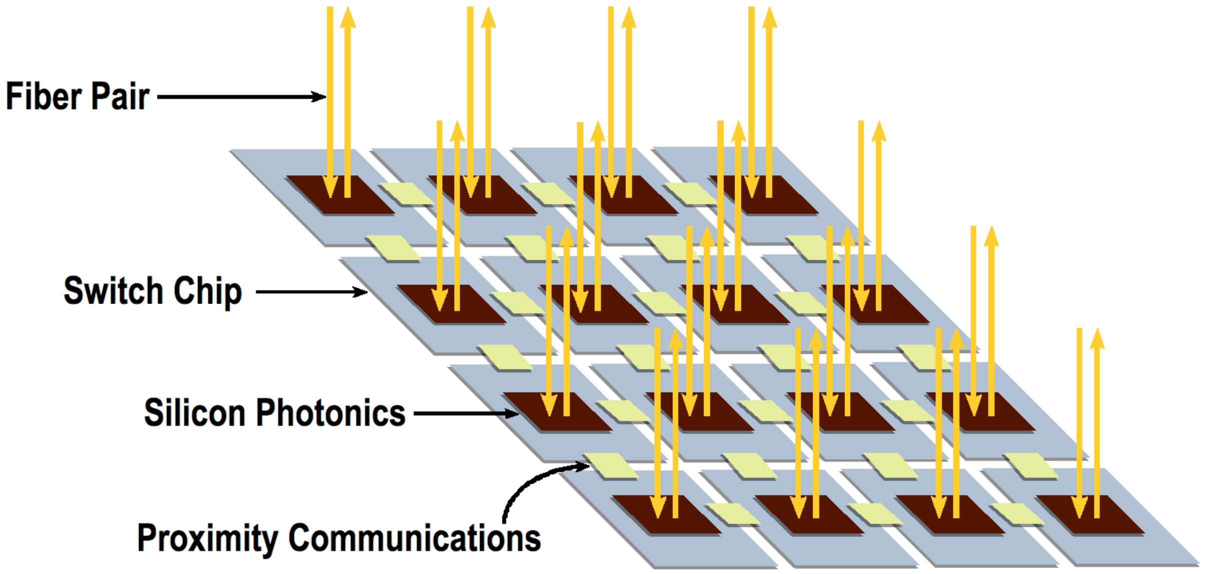


Figure 21: Example of a Hero Switch

As described in the next section, a high port count switch is advantageous to minimize the number of switch hops that a packet must traverse, which minimizes remote access latency. By using a large grid of switch chips, the Hero Switch can be almost arbitrarily large. For the 4x4 grid shown in Figure 21, the Hero Switch can support 64 ports with four fiber pairs (4x4 chips x four ports per chip) or 128 ports with eight fiber optic pairs. With a larger 8x8 grid and eight fiber optic pairs, the Hero Switch can accommodate 512 ports. The port speed is 8x Quad Data Rate (QDR), which is 8-GB/s in and 8-GB/s out, matching the fiber optic link's data rate. Thus, a 512-port Hero Switch has a total bandwidth of 4-TB/s in and 4-TB/s out. An on-chip interconnect technology such as SerDes cannot provide the bandwidth and latency necessary for such a switch, but proximity communication can. With a few nanoseconds per hop [57], the worst-case latency for proximity communication hops through even an 8x8 switch is a few tens of nanoseconds.

A lightweight routing protocol is needed to take advantage of the low latency switch fabric afforded by proximity communication and silicon photonics. The switch fabric protocol defines the switch entrance and exit ports selected to route packets from a source compute node to a destination compute node (called source-routing because the ports are preselected at the source node). In the three-stage Clos network (Section 7.1), it defines three sets of entrance/exit fibers: two for the edge switches connecting to nodes and one for a central switch that connects edge switches (Figure 15). The switch fabric protocol minimizes latencies and always selects a shortest path through the switch fabric.

The internal switch provides low-latency, high-bandwidth wormhole routing[20] throughout the grid of switch chips. It is deadlock-free by design and provides multiple paths to route around failures. When a packet arrives via a particular fiber in a Hero Switch, it is routed through the switch as follows:

- The exit fiber is retrieved from the packet destination source routing;
- The path from entrance fiber to exit fiber through the switch chip grid is determined by table lookup (the table can be updated to reflect topology changes such as partial failures and can contain multiple paths for fault tolerance and load balancing);
- A header with the proximity communication path through the switch chip grid is prepended to the packet (the header is a series of directions through the grid, for example: go east, then south, then east, then exit on fiber);
- The packet immediately begins wormhole routing to the exit fiber using the first switching command in the prepended packet header (meaning that it selects a proximity communication or fiber optic exit from the switch chip); and
- Upon reaching the switch chip containing the exit fiber, the packet header is stripped, and the packet is sent out the exit fiber.

9.4 System configurations

By using proximity communication as a chip-to-chip interconnect, a very high port-count Hero Switch can be built without a significant latency penalty. A large switch can connect to many compute nodes and other switches, allowing the switch fabric topology to be flattened, which minimizes the number of switching stages. For example, a 3-stage Clos network could connect thousands of compute nodes using this switch. This is important because each switching stage adds latency and increases switch hardware cost. Minimizing the number of switch hops also provides a smooth latency curve in the switch fabric to minimize the impact of locality.

A high-level system architecture using the switch fabric is shown in Figure 22 (same as Figure 14, repeated for convenience). Local compute nodes can communicate through a single edge switch, while central switches provide access to the entire system.

This hardware architecture can efficiently scale to thousands of nodes for many different network topologies. Table 10 lists the number of nodes that can be supported for different size switches in various Clos network configurations. Nodes with multiple ports can also be connected to multiple networks (sometimes called rails); for example a node with four ports could be connected to four different Clos networks, increasing the maximum number of nodes and switches shown in Table 10. Traversing a switch adds latency, so reducing the number of switching stages also reduces remote memory access latency. The Hero Switch is a single-stage switch that can scale to 512 or more ports, far in excess of port counts for current single-stage switches.

[20] Wormhole routing/switching is an efficient flow control mechanism that allows the packet's head to leave a switch chip before the tail arrives. Thus, a packet could be spread across a number of switch chips in the grid, creating a worm-like image.

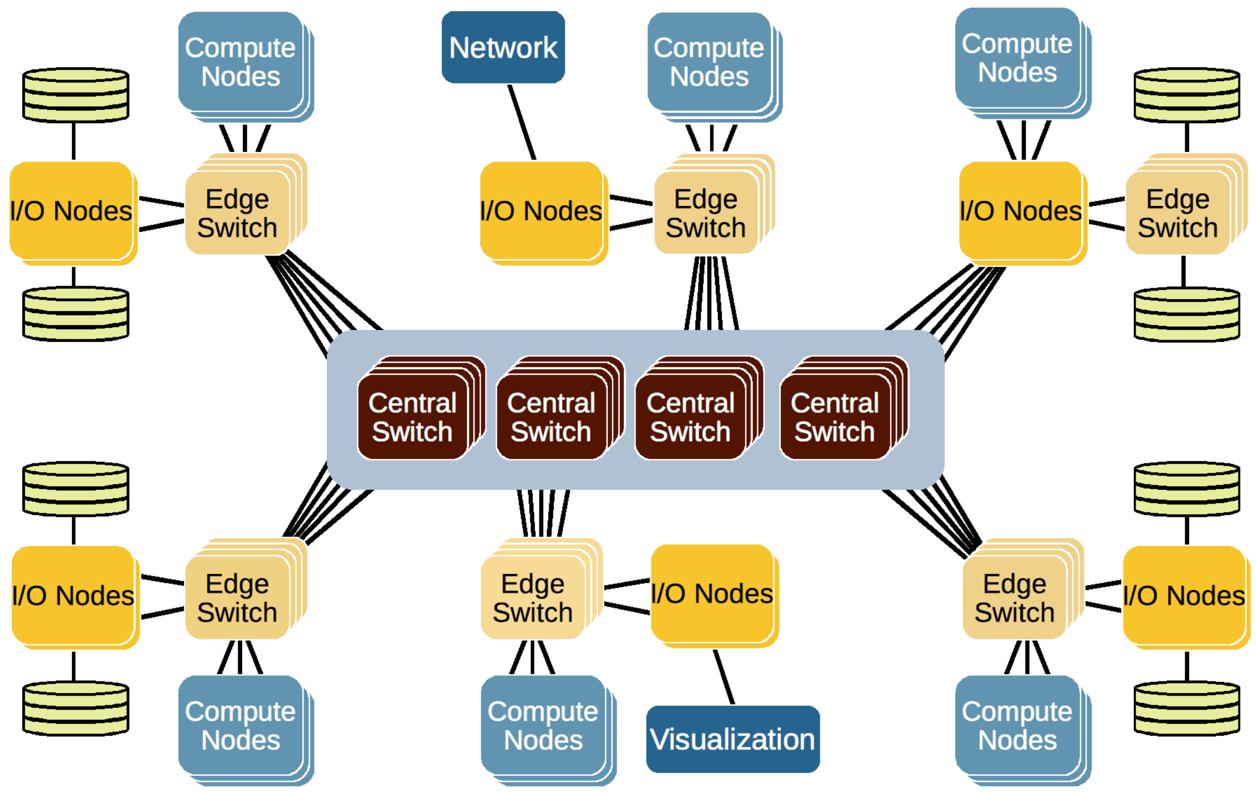


Figure 22: Hero system architecture

Table 10: System configuration examples using a Clos network

Ports per Switch	Number of Stages	Number of Switches	Maximum Number of Node Ports	Maximum Number of Nodes with 4 Ports Each
16	1	1	16	4
16	3	24	128	32
16	5	320	1,024	256
64	1	1	64	16
64	3	96	2,048	512
64	5	5,120	65,536	16,384
512	1	1	512	128
512	3	768	131,072	32,768

10 Execution model

An execution model describes how hardware architecture directly supports one or more programming models. It describes what the machine sees, as opposed to what the programmer writes. As shown in Figure 23, the execution model is at the boundary between hardware and system software layers, while the programming model is at the boundary between system software and development environment layers.

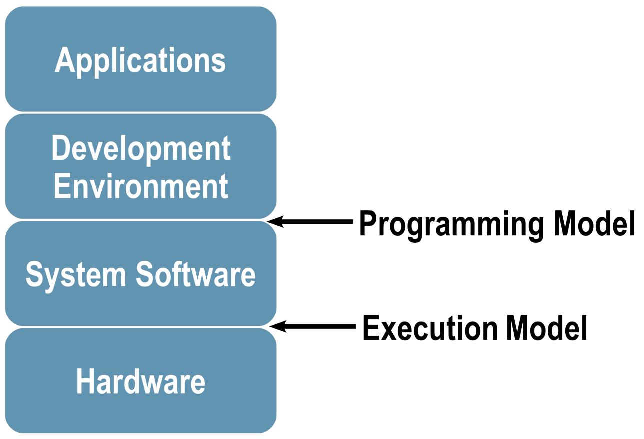


Figure 23: Execution versus programming models

The Hero execution model (Section 10.1) does not support all programming models, but it does provide a global address space and other key features that support established HPC programming models (Section 3.10.1) and enable newer, higher productivity programming models (Sections 3.10.2 and 3.10.3). The execution model, *global shared memory with managed coherence,* is sufficiently abstract that it can be ported across platforms and sufficiently powerful to virtualize hardware resources. It supports a simplified memory abstraction that frees programmers from managing memory locality. The Hero execution model provides an execution environment that is familiar, meaning that it looks, feels, and acts as though Hero is a single SMP machine. In essence, it is an abstraction that allows programmers to concentrate on algorithms instead of resource management.

10.1 Global shared memory with managed coherence

Memory coherence was initially thought to be an HPC application requirement. However, a purely hardware implementation of memory coherence across a system of this size is not feasible (Sections 8.2 and 13.1). This led the team to revisit the requirements and, in particular, to analyze in more detail the actual memory requirements of applications expected to run on Hero. Typical HPC applications alternate between serial and parallel regions, with the vast majority of time and computation spent in parallel regions in order to take advantage of a supercomputer's huge compute resources. The global memory state must be universally visible and consistent when transitioning from parallel to serial region and vice versa, but not in between. Also, the transitions are expected to be infrequent. The team considered how Hero's hardware architecture could support these transitions, while also satisfying the requirement for a shared memory model (Section 5.1.1).

Three features of Hero's hardware architecture determine what execution models it can support directly:

- *Global addressability:* Every CPU can directly address and access any memory location in the system using standard load/store instructions. This access is made efficient by a high-bandwidth, low-latency interconnect. Its efficiency depends upon the virtual memory support described in Section 10.2.
- *Remote load/store* semantics have specific ordering properties. Individual nodes impose total store ordering (TSO) on all stores from local processors to local memory. In addition, all remote stores are seen in program order by the issuing thread. However, other threads may see stores out of program order.
- *Cache coherence:* Each node locally supports full hardware cache coherence. There is no support for full hardware cache coherence at the multi-node system level, which would be complex and expensive. However, there is combined hardware and software support that enforces full coherence at designated program points using a special barrier mechanism, a *coherence fence*, which is explored more fully in Section 10.3. Hero's ability to enforce full coherence at designated points in a program is called *managed coherence*.

The extent to which these three features satisfy the requirements of four common parallel programming models is summarized in Table 11. The programming models are cache-coherent shared-memory, OpenMP [44], partitioned global address space (PGAS) [19] [42] [61] and message passing interface (MPI) [38]. The cache-coherent shared-memory programming model is not a formal standard, but is used to represent the commonly understood set of properties provided by most SMP machines and relied upon for correct program execution using a form of threaded concurrency. The other three programming models are current HPC programming models.

Table 11: Programming model requirements

	Cache-Coherent Shared-Memory	OpenMP	PGAS	MPI
Global Addressability	Required	Required	Required	Local
Remote Load/Store Semantics	Required*	Required*	No**	No***
Cache Coherence	Full Global	Fence^	Fence^	Fence^

* - Full remote load/store semantics are required, because no messaging semantics are available
** - Remote load/store semantics are convenient but not required, because messaging semantics are available
*** - Remote load/store may improve the implementation's simplicity and performance
^ - Coherence required at explicit locations where a coherence fence may be inserted

The cache-coherent shared-memory programming model is not directly supported by Hero hardware due to a lack of full global cache coherence and global store ordering guarantees, but direct support for this programming model is not a requirement. However, it is a requirement to support the other three programming models and future highly productive programming models (Section 5.1.4). The OpenMP model is directly supported by hardware due to system-wide direct addressability and direct processor load/store support, along with global coherence fences for transitions between parallel and serial regions. Auto-parallelized Fortran, the high-programmability style with Fortran 90 demonstrated in HPCS Phase II research [35] (Section 3.7), offers a model effectively identical to OpenMP, so it is also directly supported. PGAS and MPI models are also directly supported by Hero hardware, as shown in Table 11. The productivity gains targeted in the DARPA program require the high-programmability style presented in HPCS Phase II work (Section 12.3), which is fully supported by the Hero system.

The determination that global coherence is necessary only at certain points in a typical HPC application became a fundamental basis of Hero's design. This execution model is called:

> **Global shared memory with managed coherence**—global load/store semantics supported by a shared global address space with coherence fences.

This execution model is feasible to implement and directly satisfies the shared memory model (Section 5.1.1) and programming model support (Section 5.1.4) requirements.

10.2 Virtual memory

The execution model provides a foundation for global shared memory access, and the high-bandwidth, low-latency interconnect enables global shared memory. However, in order to provide efficient, system-wide load/store access, the Hero system must also provide efficient mechanisms for virtual memory, address translation, and memory protection.

The system software on each node owns and manages that node's local memory, with shared memory mappings across nodes negotiated by system software on the respective nodes. All nodes have the same view of the entire shared memory region. However, different nodes can have different page mappings for a memory region, such as where shared code is replicated locally for performance. Node-local memory mappings are private.

A single SMP machine has three different address space views: the usual physical address space for physical memory, the (process) virtual address space to which application programmers write, and the (system) virtual address space that system programmers care about. The system virtual address space translates from virtual addresses in applications to physical memory locations and manages constructs such as page tables and swap space. In order to make Hero a familiar environment with standard shared-memory programming semantics for application and system programmers, it was necessary to make the same three distinct address spaces appear to programmers as though they are on a single machine, even though the spaces are actually spread across many nodes.

- Each Hero node has its own standard physical address space. The *system physical address space* is the usual combination of all node physical address spaces; that is, all of system memory.

- The *multi-node process virtual address space* describes system memory from the view of a single process. This address space is usually spread across portions of many different nodes throughout the system. There are as many of these spaces on the system as there are processes that run on multiple nodes. As usual, portions of a virtual address space may reside in memory, while other portions may reside in other storage such as a disk.
- The *global system virtual address space* describes a view of the entire Hero system memory, represented as tuples of <*virtual node id : virtual offset*>. Virtual node id 0 is reserved for the local node, so there is no overlap between the global system virtual address space and the local physical address space of a node. This space provides each process with a single view of global memory, facilitating shared-memory programming semantics as described below.

As an example of Hero memory address space mapping, Figure 24 shows the address spaces for a Hero system that is physically partitioned into two sets of nodes. These partitions have their own separate physical and virtual address spaces, so *global* and *system* in the above definitions really mean "within a partition." Although there are some administrative constructs to manage the assignment of nodes to partitions, application and system programmers view a partition as a system. Therefore, for the remainder of this document, the Hero system is described as though it were a single partition, and the terms *global* and *system* are used in their natural sense.

There are *k* processes and *m* nodes in Partition A. The mapping of processes to nodes is many to many; any process can be mapped to any set of nodes. The global system virtual address space manages the many-to-many assignment. In Figure 24:

- Process 1 is assigned to nodes 1, 2, and *m*.
- Process 2 is assigned to a set of nodes located between nodes 2 and *m*, disjoint from Process 1.
- Process *k* is assigned to nodes 1 and 2.

The facilities that allow processes to overlap and share nodes are described in Section 11.2.

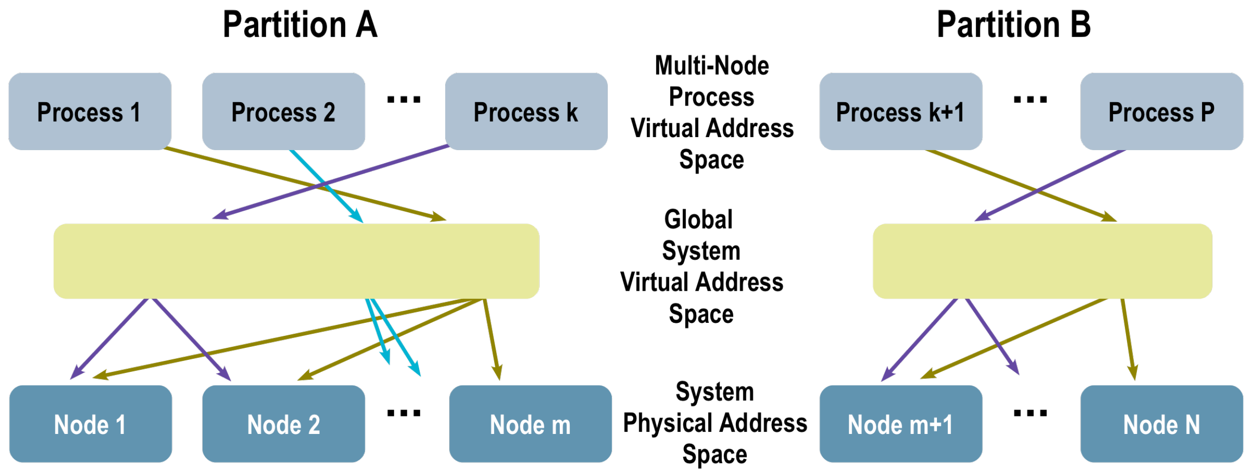


Figure 24: Hero memory address space mapping

Figure 25 shows details of the address space mapping for a single multi-node process virtual address space that uses memory in three nodes. The physical address space in a node is divided into private memory (Node Reserved Page in Figure 25) and memory available to processes (Home Node Data Space, Global Shared Text Page, and Global Distributed Data Space in Figure 25).

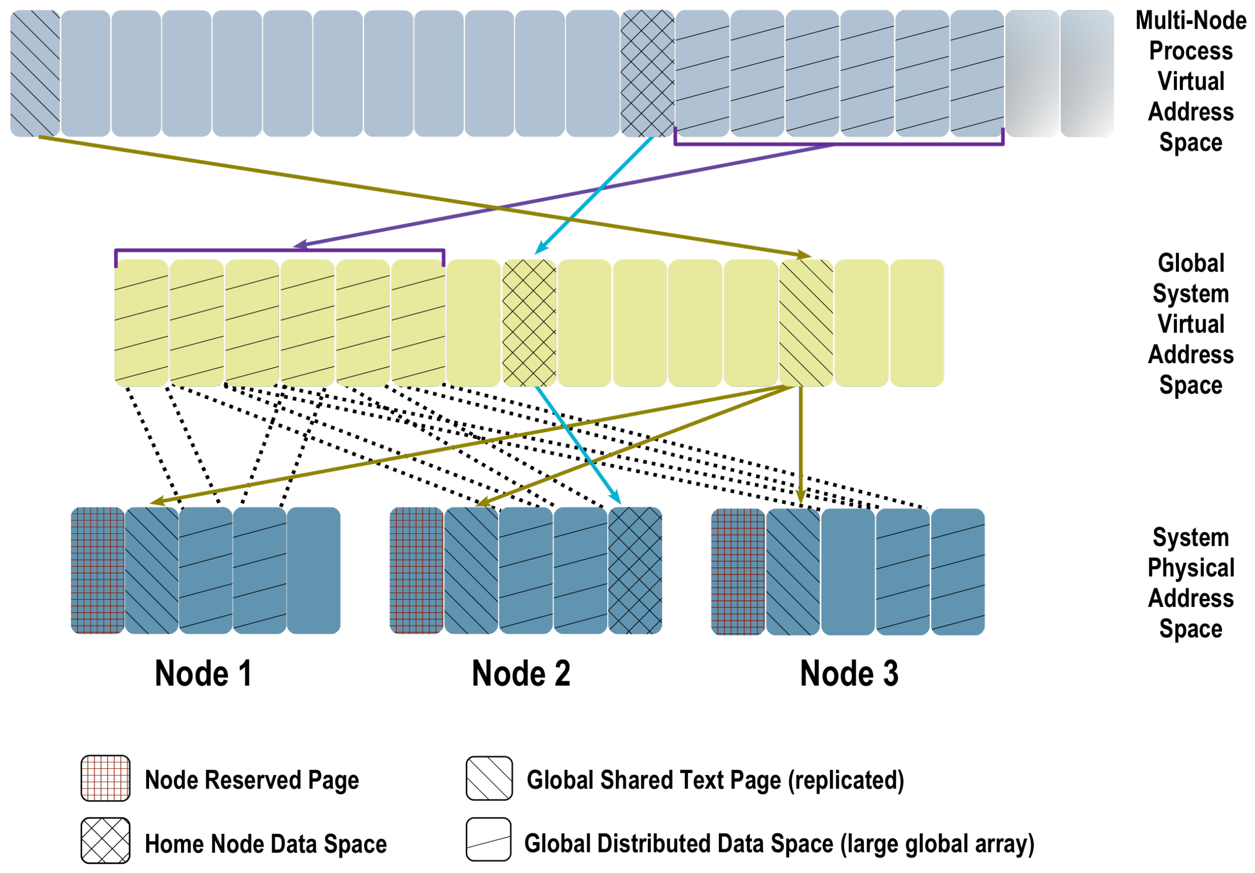

Multi-Node Process Virtual Address Space
Global System Virtual Address Space
System Physical Address Space
Node 1
Node 2
Node 3
Node Reserved Page
Global Shared Text Page (replicated)
Home Node Data Space
Global Distributed Data Space (large global array)

Figure 25: Hero memory address space mapping details

The four memory structures shown in Figure 25 are defined as follows:

- The node uses the *Node Reserved Page* to run its OS and other private memory allocations and does not share it with other nodes.
- The *Home Node Data Space* contains data that resides only in the home node (Node 2 in Figure 25). The home node is the node in which the process originated, that is, the node that received a system call from the process. The home node manages global memory mapping (Section 11.3.1).
- The *Global Shared Text Page* contains data that all nodes need for the multi-node process, such as application code. The page is replicated on all nodes to improve performance by minimizing page mapping over the network. Global Shared Text Page

data should not be modified because broadcasting global updates would significantly hurt performance.

- The *Global Distributed Data Space* contains data, such as application data, that is not replicated and is distributed across separate nodes.

The multi-node process virtual address space is the customary user virtual address space, for example 64 bits. Because 64 bits is a very large address space, some portions would be mapped into memory, some portions would be mapped into disk or other remote storage, but most of the address space would be unmapped. The global system virtual address space is an innovation that resides between physical memory and the multi-node process virtual address space; it acts as a single federated view of global memory for the multi-node process. It enables shared-memory programming semantics by unifying all process data spread across the nodes.

10.2.1 Address translation

To make Hero a highly productive programming environment, programmers must be able to use standard load/store semantics and interfaces, for example, by simply writing a load instruction and a memory reference. The Hero virtual memory system transparently performs the address translation and handles the request locally or remotely, depending on where the requested data is located.

The key innovation for transparent remote address translation is the scalability interface (SIF) ASIC (Section 8.2). The SIF sits on a node's memory bus and acts like a local memory controller, but actually connects to the Hero interconnect fabric and provides an interface to remote memory. When a memory access is requested, the processor's memory controller performs its normal functions, but produces an address in the global system virtual address space of the form <*virtual node id : virtual offset*>. If the virtual node id is 0, the memory address is on the local node and is handled normally by the local node memory controller. If the virtual node id is greater than 0, the memory address is on a remote node, and the SIF acts as a memory controller for the remote memory request.

The SIF contains tables for mapping a remote virtual address into a physical node id and a virtual offset for the remote node corresponding to the virtual address. The SIF on the local node forwards a memory request to the indicated remote node; a receiving SIF on the remote node also uses table lookups to translate the virtual offset into a physical page id and a physical offset. The request is then forwarded to local memory controllers on the remote node to be satisfied in the normal fashion. Finally, the remote SIF returns the result to the SIF on the requesting node.

Figure 26 illustrates address translation for a remote memory request, for example, a store operation. The memory controller sends a virtual address to its local CPU translation look-aside buffer (TLB)[21], which accesses local memory if the virtual address maps to a local physical address. In this example, however, the operation is a remote request, and the CPU TLB sends the virtual address to the SIF, which uses its route table to determine an appropriate remote node and then routes the request to that node. The SIF on the remote node acts as a TLB in translating the

[21] A translation lookaside buffer (TLB) is a CPU cache used to improve the speed of virtual address translation.

virtual address to a physical address and then forwards the memory request to the memory controller on the remote node to perform the indicated operation.

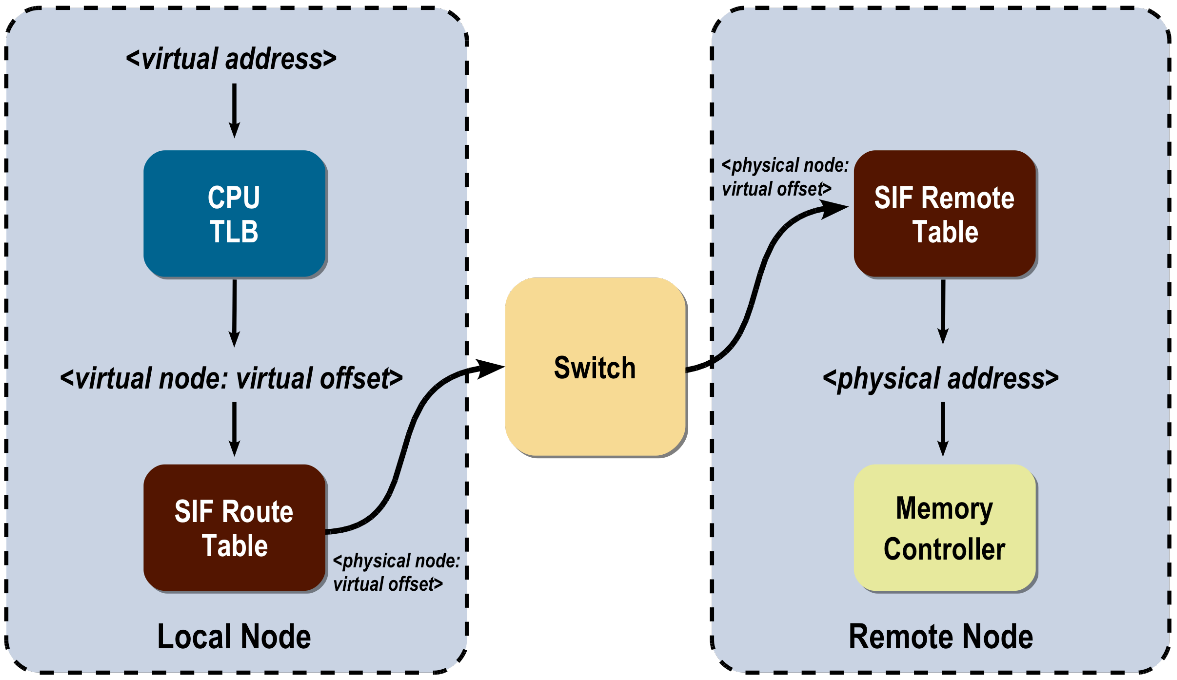


Figure 26: Address translation for remote memory request

10.2.2 Memory protection

Memory protection for remote memory access is a two-phase operation. At the node where the memory request is made, a local memory controller checks the protection bits for the requested page. If that check fails, the error is handled normally. If the check succeeds, the memory request is passed to the remote node containing the physical memory, as shown in Figure 26. Then, at the SIF on the remote node, an additional check is performed to see if the incoming memory request is from an approved node. This prevents nonparticipating rogue nodes from corrupting memory on a remote node. Such faults are reported asynchronously, but should happen only for cases where the OS of the node that made the memory request is either malicious or corrupted.

To create a multi-node process memory allocation, the local OS on each remote node requests a mapping from the home node (the node where the process originated). Each remote node maps its memory and notifies the home node of mapped addresses. Each remote node also updates its memory protection table in its SIF to indicate that all nodes involved with the multi-node process are legal sources for memory accesses on this node. When the mapping takes place, all nodes involved with the multi-node process establish a common mapping; the control domains for each node assigns pages for that mapping, with protection information for the local process ID corresponding to the global process initiating the mapping.

10.3 Synchronization mechanisms

Synchronization mechanisms are used to implement managed coherence. The standard HPC synchronization mechanism is a *barrier*: a construct that enforces the property that all threads must enter the barrier before any thread leaves the barrier. The *coherence fence* is a special form of barrier with the following additional property: all stores from all threads must complete and become visible to all threads before any thread exits the fence. The coherence fence is the synchronization mechanism used to enforce coherence at designated points in the application—for example, before and after a parallel execution region. A simple implementation flushes all dirty cache entries and invalidates all cached data before any thread leaves the fence. To implement the coherence fence, Hero supports remote cache flush and cache invalidate operations.

When an application arrives at a coherence fence, the application calls the system software, triggering a notification to all nodes on which the application is running. When all application threads have arrived at the fence, the system software initiates a flush-and-invalidate operation on every CPU where the application is running. When those operations complete, the system software returns from the fence and allows threads to continue execution. The entire application is in a fully cache-coherent state at the point where execution resumes. A coherence fence can have a significant performance impact, but HPC applications are expected to use this construct infrequently.

11 System software

To support productivity at petascale, Hero software provides a familiar environment for applications and system software, allowing programmers to write software as if for a single SMP machine when they are actually controlling thousand of nodes in a supercomputer. Resource virtualization is the key enabling technology; it provides the essential flexibility and scalability necessary to create such an environment. It permits Hero software to support both high programmability and legacy applications, scaling from single nodes to very large, multi-node systems. System resource virtualization permits programmers to write logical constructs without concern for managing physical resources, and it allows the system to change the physical resources dedicated to an application. It is a very powerful abstraction that enables efficient mode switching (such as transitions between capability and capacity modes) and hardware reconfiguration transparent to the applications (such as resource reallocation due to a hardware failure).

Software for the Hero system, summarized in Figure 27, consists of Hero system software and the Hero development environment. The Hero system software includes the fault tolerant hypervisor, Hero Solaris™ OS [45], Hero file system, and Administrative Environment. The Hero development environment includes compilers, parallel and serial debuggers, and other programming tools, as well as a variety of performance, I/O, and other libraries and visualization software. This section describes the Hero system software, starting with a description of its basic structure and features in Section 11.1. The remainder of Section 11 describes key enabling technologies that cross system software structural boundaries and work together to support multi-node applications. Section 12 describes the Hero development environment.

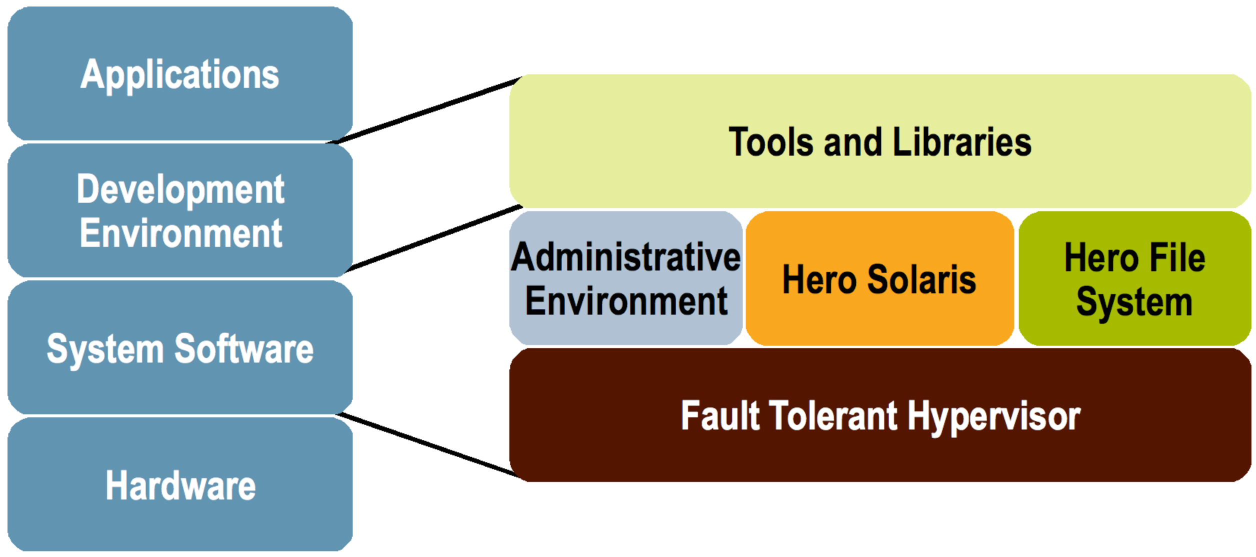


Figure 27: Hero system software

11.1 System software stack

Running a single OS across a system the size of Hero is infeasible—scheduling is a bottleneck, kernel algorithms break, 32-bit counters overflow, and so forth. Therefore, the system software approach is to run the software needed for node resources and home node functionality on every node, but to run the software needed for multi-node application and system functionality only on a small, replicated subset of nodes. This approach mitigates some of the most severe scaling challenges for system software by minimizing system-wide services; it maximizes robustness by providing extensive redundancy of functionality throughout the system.

Every node hosts instances of a fault tolerant hypervisor, the Hero Solaris OS, and Administrative Environment services modules. Only a subset of nodes hosts instances of the Hero File System service modules and system-wide Administrative Environment service modules. These components collaborate to provide a multi-node application execution environment that supports highly productive programming models, while also providing independent system support for each node in the Hero system.

The following subsections describe the hypervisor and operating system, which run on every node, as well as portions of the file system and administrative components, which run on a subset of nodes. Sections 11.2 and 11.3 describe system software functionality that provides the multi-node application execution environment. System-wide administrative functions are described in Section 11.4.

11.1.1 Fault tolerant hypervisor—virtual machine support

The fault tolerant hypervisor provides the foundation for Hero system software. It is based upon the current OpenSolaris Sun4v hypervisor, with extensions to provide fault tolerance and scalability features. It provides full hardware device virtualization to insulate software layers from hardware failures. The fault tolerant hypervisor also provides state replication for its own state in order to improve overall robustness in the face of hardware faults. It supports migration of virtual machines (OS instances and everything running on top of the OS instances) within a node and across node boundaries and provides support for checkpointing virtual machines. Virtual machine migration and checkpointing are both essential elements of Hero's global fault tolerance strategy.

11.1.2 Hero Solaris™ OS

The Hero operating system is based on OpenSolaris, enhanced with features to increase robustness and with a limited set of multi-node semantics for applications. OpenSolaris is an open source project based initially on Sun's Solaris 10 code base [45]. OpenSolaris is a version of the UNIX System V operating system, with a long track record of excellence in the areas of robustness and support for concurrency; it is used extensively on both SPARC® technology-based and X64-based systems in a wide variety of commercial, industrial, government, and scientific applications.

Hero Solaris includes new robustness features that extend the current OpenSolaris Fault Management Architecture to support predictive self-healing over a multi-node system and to support a new, automated checkpointing technology. Multi-node semantics in Hero Solaris (Section 11.2) supports memory management, process and thread management, as well as file and network I/O across multiple nodes. This support is built on the OpenSolaris Zones and BrandZ technologies [45]. The intent is that Hero Solaris features will be released under open source as part of the OpenSolaris environment.

11.1.3 File system

The Hero File System utilizes object-based storage as defined by the T10 object-based storage device (OSD) standard. Object-based storage helps meet performance and scalability requirements by distributing space allocation and layout decisions to storage devices and by eliminating the need for locking to handle multiple writers to a file. Block disk and tape devices are supported in the Hero File System by using object storage servers. Hero also supports legacy file systems, including Lustre, NFS, and Parallel NFS (pNFS), along with standard OpenSolaris file systems such as ZFS.

11.1.4 Administrative Environment

The Administrative Environment is at the boundary of the Hero system software and Hero development environment; it includes tools that help manage the system and improve application execution. The Administrative Environment provides standard administrative functions—including active management of thousands of physical nodes, resource allocation and scheduling, resource utilization tracking, firmware management, power sequencing, operating system install, update and boot, host virtualization (including host-level fault management), and network management. It can define groups of nodes flexibly, as well as implement management activities and policies in a hierarchical manner.

An important abstraction for a system of Hero's size is the use of automated policy-based resource management, including administrative control over the weight assigned to different resource management policies. The Administrative Environment supports automatically triggered system/application checkpoints, automatic policy-based preemption, and automated triggering of resource reassignment using various policies at different levels of sophistication.

During application execution, the Administrative Environment provides troubleshooting tools for log viewing and event tracking, as well as a telemetry data mining service that analyzes historical system data to identify patterns, trends, and anomalies in system behavior over time. These include visual tools to help the administrator monitor systems and track historical system data, as well as view, search, and associate multiple log files from multiple systems.

11.2 Multi-node support

All layers of Hero system software work together to create a multi-node set of computational resources for Hero applications. The fault tolerant hypervisor virtualizes hardware resources. Hero Solaris and the Hero file system provide multi-node containers (SuperZones) in which applications execute. The Administrative Environment presents the SuperZone as a single logical computational unit to the application.

11.2.1 Zones

Zones are an operating system abstraction for partitioning systems, allowing multiple applications in a single operating system instance to run in isolation from one other. This isolation prevents processes running within a zone from monitoring or affecting processes running in other zones, accessing each other's data, or manipulating underlying hardware. Zones, together with the hypervisor, also provide an abstraction layer that separates applications from physical attributes of the machine where they are deployed, such as physical device paths and network interface names.

BrandZ is an OpenSolaris framework that extends the OpenSolaris Zones infrastructure to create branded zones that contain non-native operating environments. The term "non-native" is intentionally vague, as the infrastructure allows for the creation of a wide range of operating environments. Each operating environment is provided by a brand that plugs into the BrandZ framework. A brand may be as simple as an environment with standard OpenSolaris utilities replaced by their GNU equivalents, or as complex as a complete Linux user space. Hero Solaris includes two innovative extensions to the OpenSolaris BrandZ zones framework that make a multi-node execution environment possible: Unification Zones and SuperZones.

Unification Zones (UZ in Figure 28) are branded zones where the brand supplies a limited set of system calls and services with multi-node semantics. Multi-node support is provided through interposition of the brand on system calls, kernel upcalls, and signals. Operations requiring the participation of more than one node invoke interzone communication mechanisms that allow Unification Zones on multiple nodes to fulfill the request cooperatively.

SuperZones are multi-node sets of computational resources established by the Administrative Environment. They act as containers for multi-node applications. A SuperZone is instantiated by notifications to system software on each node in the SuperZone, and the establishment on each node of a Unification Zone where the multi-node application's elements may execute. A SuperZone may include Unification Zones on one or more nodes and could even comprise all resources in a system. Figure 28 illustrates an example mapping of Unification Zones, each running within a node, to a SuperZone running across multiple nodes. The mapping is extremely flexible: a node could have multiple Unification Zones all contained in the same SuperZone. Note that the Unification Zone and SuperZone do not extend into the hypervisor layer, because they do not interpose multi-node semantics on hypervisor calls, although the hypervisor provides physical resource abstraction for Unification Zones.

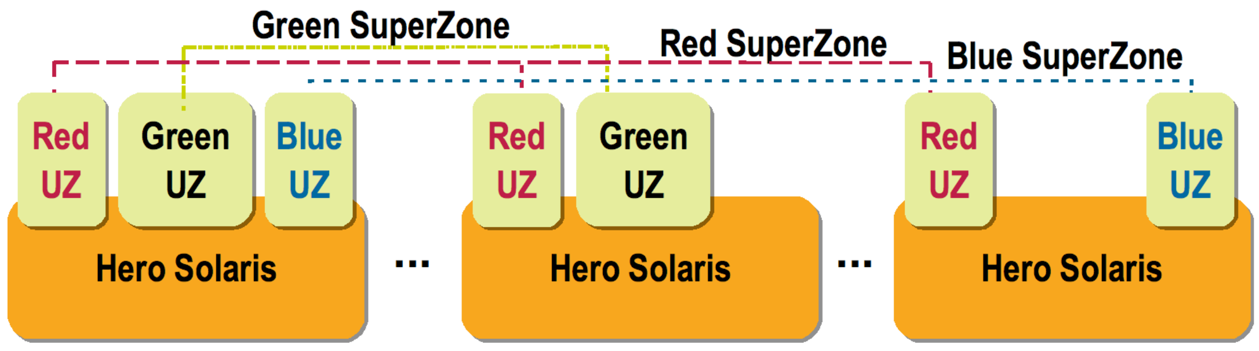
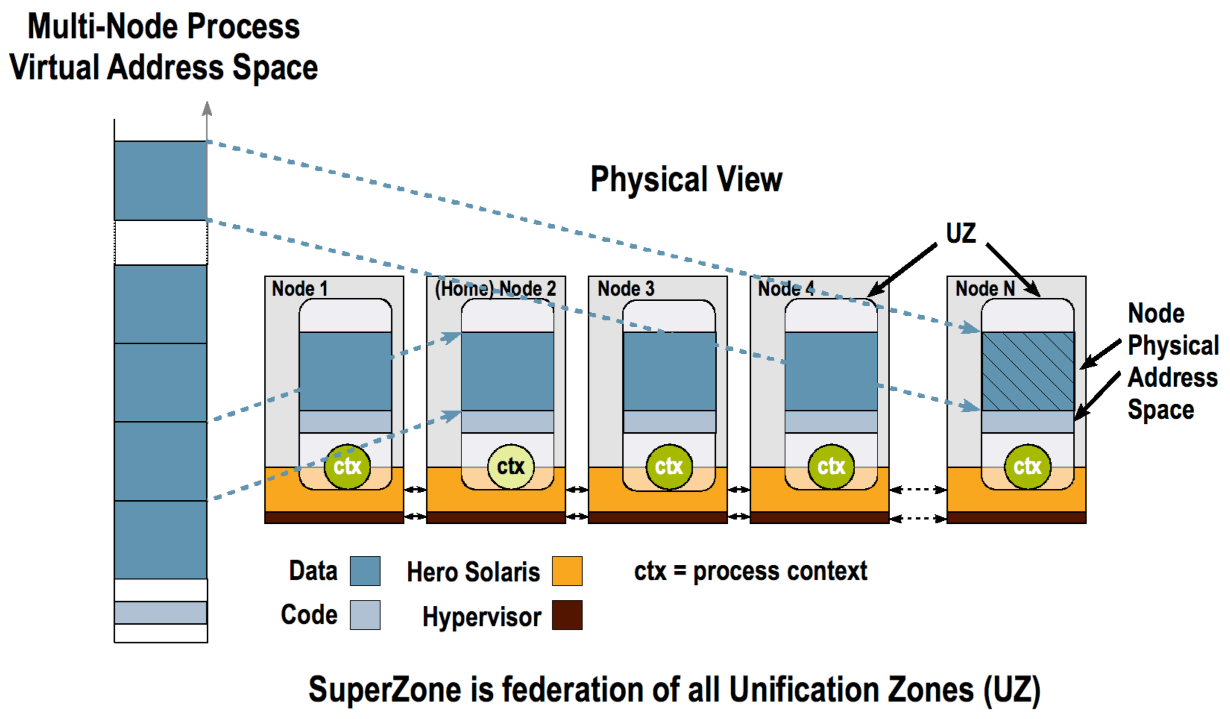


Figure 28: Mapping Unification Zones (UZ) to SuperZones

Unification Zones and SuperZones support allocation and mapping of both local and remote memory, as shown in Figure 29. This permits applications to allocate data structures that are too large to fit on a single node, or distribute a data structure across a group of nodes. To maximize performance predictability, the SuperZone attempts to distribute remote allocations evenly across nodes assigned to the application.

In Figure 29, the SuperZone comprises Unification Zones residing in nodes 1 through N, and multi-node process data structures are distributed across those nodes. Application code and process context are also replicated in those nodes. Communication at hypervisor and OS levels (shown by arrows between Hero Solaris and the hypervisor on each node in Figure 29) is necessary to establish a SuperZone.



Multi-Node Process Virtual Address Space
Physical View
UZ
Node 1
(Home) Node 2
Node 3
Node 4
Node N
ctx
Node Physical Address Space
Data
Code
Hero Solaris
Hypervisor
ctx = process context
SuperZone is federation of all Unification Zones (UZ)

Figure 29: Application data distribution throughout a SuperZone

To create a SuperZone, a node—usually the node in which a system call occurred (Node 2 in Figure 29)—is selected as the *process home*. This home node replicates the process context to other nodes in the SuperZone (Section 11.3.1). It determines that a multi-node memory allocation is requested and spreads the mapping of the requested memory (application code and data) over nodes in its SuperZone. Then, it communicates the proposed mapping to each Unification Zone in the SuperZone, requesting that they map their portion of the request. Upon receiving successful responses from all Unification Zones, the proposed mapping is committed, and the newly mapped memory is available for use by the application. Similar techniques are used to support process and thread management in a SuperZone.

11.2.2 File and network I/O support

Hero Solaris provides two basic types of file and network I/O support—a single-system view or single-node view. The SuperZone construct provides applications with a single view of the Hero system, meaning applications see a single IP address and a global name space. This allows programmers to use standard file and network I/O constructs and programming techniques. Because each node runs Hero Solaris, applications also have a single-node view using a Unification Zone on a node for independent file and network I/O access. This could be useful for improved device performance or legacy application support. In either case, I/O support is provided through I/O nodes (Section 8.3). A virtual Ethernet interface connects compute nodes and I/O nodes over the Hero interconnect to provide a robust communication channel for legacy and external communication.

Two principal forms of support are provided for network I/O. To allow legacy codes to run mostly unmodified, network I/O can be executed using standard interfaces; traffic is routed to the SuperZone's home node and the TCP/IP stack is executed there. This provides a single central IP address for the SuperZone and gives the application—and any outside entity communicating with it—the appearance that the application is running on a single system for the purposes of network I/O. Alternatively, Hero Solaris supports network I/O using one or more IP addresses per Unification Zone, with local TCP/IP stacks. This alternative model enables high degrees of parallelism in the network I/O arena and is well suited to PGAS and MPI programming models. Hero tools and libraries on each node also support standard local networking.

Similarly, for file I/O, Hero Solaris provides an interface for legacy single-system semantics—using the home node for naming—and a more parallel interface where files are accessed independently from each node in the SuperZone. The system supports a flexible layout, with no need for direct attached storage at every node, and the capability to isolate the file system software from application software. To support legacy semantics of various I/O facilities, each Unification Zone in a SuperZone provides a global context mechanism that allows process context for a multi-node process to be replicated across all nodes in the SuperZone. This enables fast name lookups in a multi-node name space and makes it possible to provide multi-node applications with the appearance of global name spaces for a variety of purposes.

11.2.3 Robustness features

In support of increased system availability, extended OpenSolaris™ Fault Management Architecture facilities allow Hero to provide a form of federated fault management. This extends current Sun predictive self-healing technologies to multi-node systems like Hero by providing facilities for aggregating fault information across multiple nodes and handling faults above the node level. Federated fault management enables multiple Hero Solaris instances to participate in the prediction, diagnosis, and healing of system faults. The Fault Management Architecture saves process state redundantly and restarts processes on failure, making it internally fault tolerant. This state replication mechanism will also be used to provide fault tolerance for other Hero system software. In addition, Hero Solaris provides mechanisms that support automated system checkpoints, described in greater detail in Section 11.3.3.

11.3 Runtime environment

As mentioned above, the SuperZone provides the core abstraction for running multi-node applications on Hero. It is a set of one or more Unification Zones that provide high-programmability applications the limited appearance of a single system. From the perspective of the Administrative Environment, the SuperZone is a multi-node set of hardware and software resources that can execute a multi-node application. It is the fundamental unit of logical resource partitioning for the Hero system and serves as the basic unit for automated system checkpointing.

It is important to differentiate between the standard notion of a single-system image and the single-system view provided by Hero. A single-system image provides applications with a single view of all system services. In a multi-node system, this usually means that a single OS image runs across the entire system. Hero's single-system view, instantiated as a SuperZone, provides applications with a single view of all computational resources—memory, files, and I/O—but it does not provide a single view of all system services. For example, there is neither global console nor global visualization support. Individual nodes must stream data separately to visualization engines, rather then sending a single stream controlled by a SuperZone.

11.3.1 Multi-node application support

Within a SuperZone, the Administrative Environment can initiate the execution of a multi-node application. Examples of such applications include a high-programmability application written in Fortran, automatically parallelized, and an OpenMP Fortran program. Programming tools mark these as multi-node applications, which instructs the Administrative Environment to initiate such an application in a SuperZone.

To create a SuperZone, a multi-node process is created on the home node and then propagated to other nodes. In Figure 30, Node 2 is the home node and it creates a process in response to a system call. The system software creates a process on each Unification Zone in the SuperZone (Nodes 1-N in Figure 30) and replicates the multi-node context from the home node to each of these processes. This allows the system to create the appearance of multi-node name spaces without forcing identical naming across the entire system. Using the mechanisms shown in Figure 29, the application can allocate local and remote memory, including very large allocations that do not fit on a single node. The home node propagates system calls, as well as signals and Kernel upcalls, to other nodes in the SuperZone. For example, it can propagate a call to open a file with a file handle. Internode communication may be at the hypervisor level as well as the OS level if hardware is involved, such as when a network port is requested.

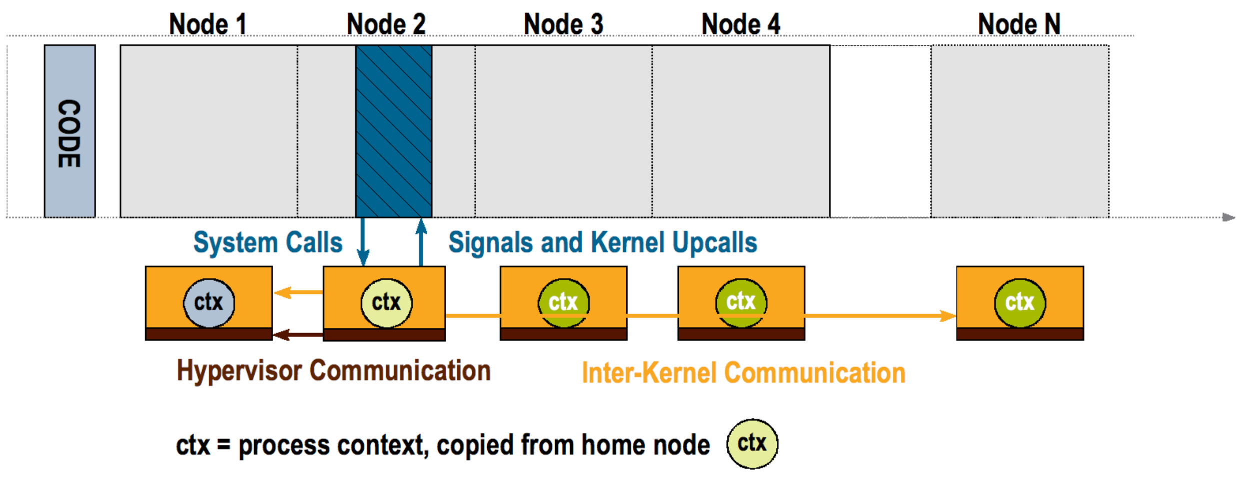

Node 1
Node 2
Node 3
Node 4
Node N
CODE
System Calls
Signals and Kernel Upcalls
ctx
ctx
ctx
ctx
ctx
Hypervisor Communication
Inter-Kernel Communication
ctx = process context, copied from home node
ctx

Figure 30: Multi-node job creation from the application point of view

After establishing the SuperZone, the application can allocate and use memory, including large allocations that span multiple nodes. Figure 31 shows an example of memory allocation from the application's point of view (see Figure 29 for the physical memory allocation view). The shaded area in the Unification Zones is actively being used for application data by load/store operations; the unshaded area in the Unification Zones on the right and left is allocated to the multi-node process, but not currently used. Each Unification Zone is contained in a single node and is in a cache-coherence domain. The SuperZone is an aggregation of Unification Zones and is a managed-coherence domain (Section 10.1). The multi-node process virtual address space shown extending to the right in Figure 31 is usually much larger than the SuperZone, but most of it is never allocated (for example, a 64 bit virtual address space).

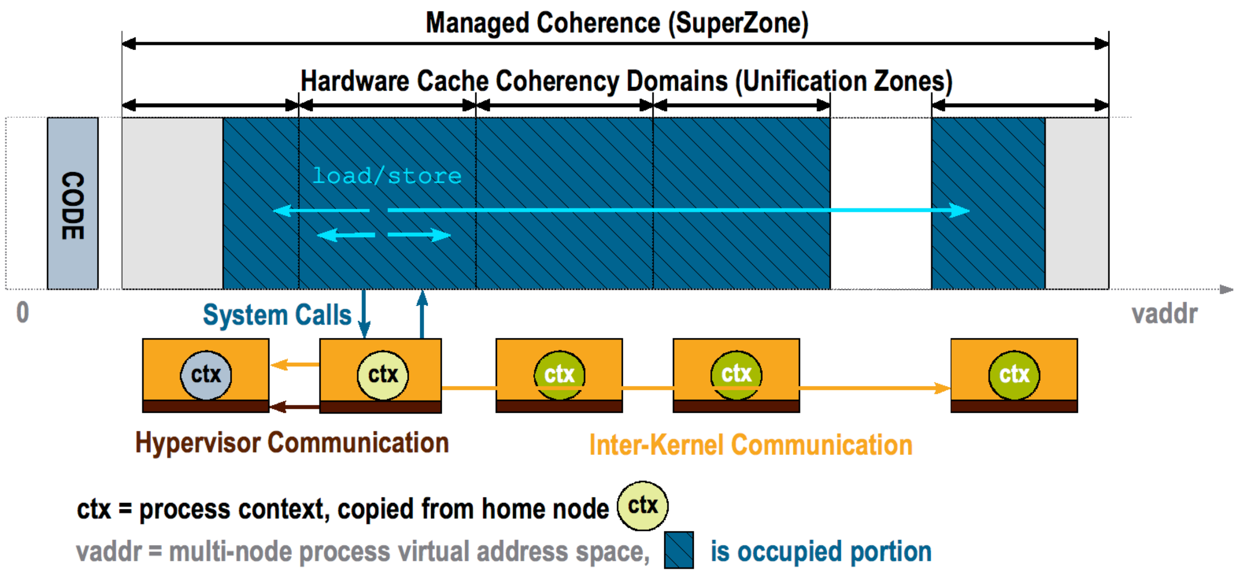

Managed Coherence (SuperZone)
Hardware Cache Coherency Domains (Unification Zones)
CODE
load/store
0
vaddr
System Calls
ctx
ctx
ctx
ctx
ctx
Hypervisor Communication
Inter-Kernel Communication
ctx = process context, copied from home node
ctx
vaddr = multi-node process virtual address space,
is occupied portion

Figure 31: Multi-node memory allocation from the application's point of view

Threads are used throughout the SuperZone to execute parallel regions in the application. Each thread has full access to all memory allocated by the application, but allocations that are neither too large nor marked as remote are made locally for improved latency and performance. Because the semantics of these programming models requires that parallel regions have no memory dependencies, all threads can execute both local and remote load/store operations; cache coherence need not be maintained. At the end of a parallel region, code generators insert a coherence fence operation to preserve programming model semantics (see "managed coherence," Section 10.1). Multi-node applications have access to the special file and network I/O facilities described in prior sections.

Hero also supports multi-node applications using MPI or PGAS programming models. These applications can be run in a SuperZone or as normal processes on individual nodes. Running in a SuperZone permits the communication and I/O libraries that implement the programming model to utilize full multi-node capabilities, including remote allocation and remote load/store operations. Running as normal processes on individual nodes and communicating using cluster-like messaging libraries, applications can still benefit from Hero's low-latency, high-bandwidth interconnect.

11.3.2 Legacy software support

Hero supports legacy software in several ways. Individual Hero nodes are standard SMP systems with full hardware cache coherence and many hardware threads. These nodes directly support applications written for SMP systems, when compiled for the OpenSolaris environment. Hero also supports MPI and PGAS programs compiled using Hero programming tools and using libraries provided with the system. As detailed in later sections, Hero offers a variety of communication and performance libraries to support legacy and high-programmability languages.

11.3.3 Automated system checkpointing

Checkpointing is the storing of an entity's state for use in error recovery or other resource remapping. Hero checkpoints can be defined at the application, node, or system level. Application checkpoints are a standard HPC defensive programming technique, but their creation consumes valuable programmer resources and adds complexity to applications. Hero offers standard support for application and node checkpoints; but Hero Solaris, along with its programming tools, libraries, and Administrative Environment, also supports a new technology for automated system checkpointing. This new facility enables Hero to recover quickly from failures by remapping failed hardware resources to other hardware and restarting from the most recent checkpoint. It can also be used by operators to temporarily suspend applications when a critical need for compute resources arises. The same automated checkpointing facility applied at the application level can obviate the need for programming application checkpoints.

To facilitate system checkpoints, Hero's code-generation tools automatically insert special code *(safepoints)* into Hero applications at locations where it is safe for the application to stop, as determined by global data-flow and control-flow analysis. At a safepoint, it is possible to take a snapshot of the entire machine's state or to checkpoint a subset of the machine, such as an application running in a SuperZone. Safepoints inserted into the Hero MPI library and other libraries can accomplish the same goal.

The checkpoint mechanism is based on the Java HotSpot™ Virtual Machine (VM) mechanism [43] used to stop the VM for garbage collection. The Administrative Environment sets policies for the frequency or conditions for checkpointing, It notifies the Hero Solaris image on each node at what time and for which threads a checkpoint is requested. Hero Solaris sets a flag that notifies each thread to stop execution when reaching a safepoint. Each thread stopping at a safepoint notifies the OS that it is stopping (the OS may be a guest OS running on a single node, rather than Hero Solaris). After all threads have stopped, the OS notifies the Administrative Environment that the node is ready to checkpoint. After all requested nodes have stopped, the Administrative Environment initiates a snapshot of the state of all nodes (or a subset, such as a SuperZone) and coordinates storage and tracking of the snapshots. The hypervisor on each node manages replication of the virtual node or guest OS state and notifies the Administrative Environment that the state has been saved. Upon notification of successful state saves for each node, the Administrative Environment notifies the OS images on each node to continue execution and notifies Hero Solaris images to restart all their threads.

When a fault occurs, the Administrative Environment can restore a SuperZone's checkpoint onto a set of nodes that is different from the set originally running the application. This new set of nodes must have the same cardinality as the original set, as well as access to the same external resources. Hero's virtualized I/O and nonlocal file system support provides an extremely flexible tool for efficiently performing this hardware remapping. In addition, these mechanisms are available to the Administrative Environment so it can provide facilities to migrate running applications off and back onto the system, resulting in a very flexible utilization of the large pool of resources represented by a Hero system.

Figure 32 shows the creation and restoration of a SuperZone checkpoint. The machine state from each Unification Zone in Nodes 1 through N is saved to memory at a safepoint, then sent to disk as a background task while the application continues to run. Alternatively, the safepoint can be sent directly to disk. This requires less memory, but causes a longer application stall [65]. When a node fails, the checkpoint can be recovered from memory or disk and mapped to a new set of nodes, in essence redefining the set of nodes contained in the SuperZone. In Figure 32, Node 1 fails and is replaced by Node N+1. Hero Solaris establishes communication with Node N+1 to reconstitute the SuperZone, prior to mapping data from Node 1 into Node N+1.

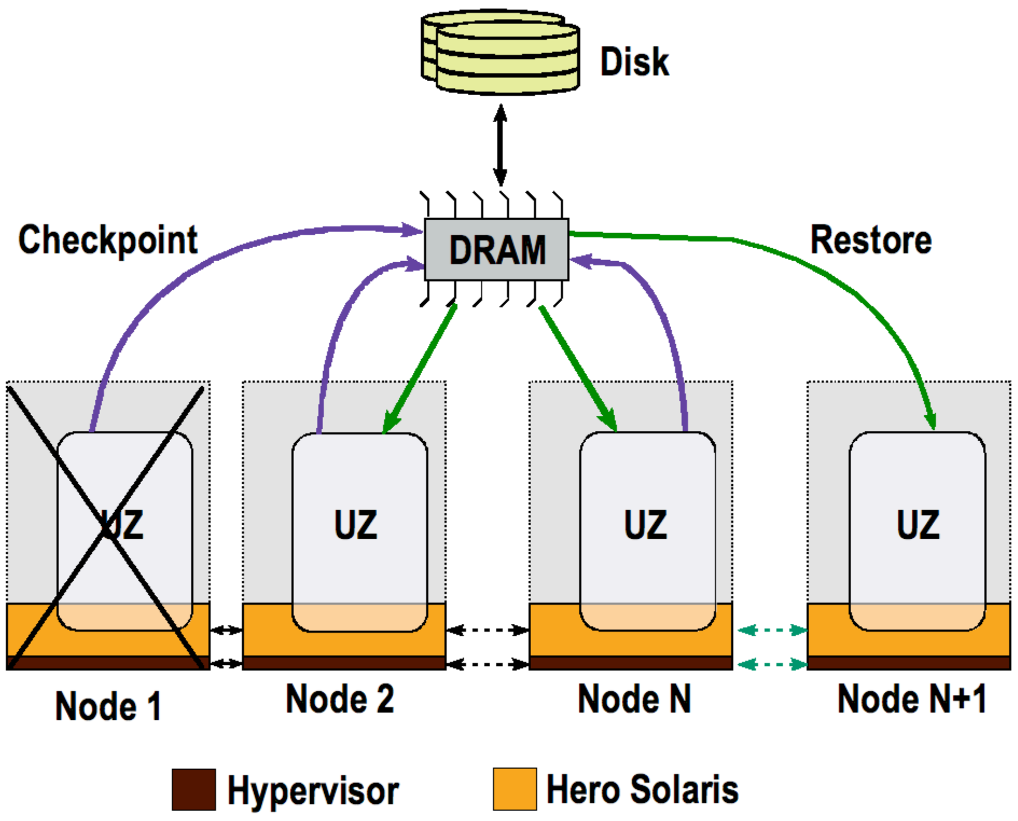


Figure 32: Automated checkpoint creation and recovery

11.3.4 Application isolation

A challenging problem encountered by applications on very large systems is *application jitter*, a performance degradation that occurs when large numbers of concurrently executing threads attempt to synchronize too frequently. The entire application must wait until the slowest thread reaches the synchronization point. Typically, the threads are actually running the same code; in theory, all threads should reach the synchronization point simultaneously. However, asynchronous behaviors in the system cause context switches or other stalls in some threads. This can lead to significant performance degradation, as shown in Figure 33. A common solution to this problem is to minimize these events and make them synchronous, causing the OS to queue up events until a timer elapses. Then, all CPUs stop executing application threads, in order to handle the event queues.

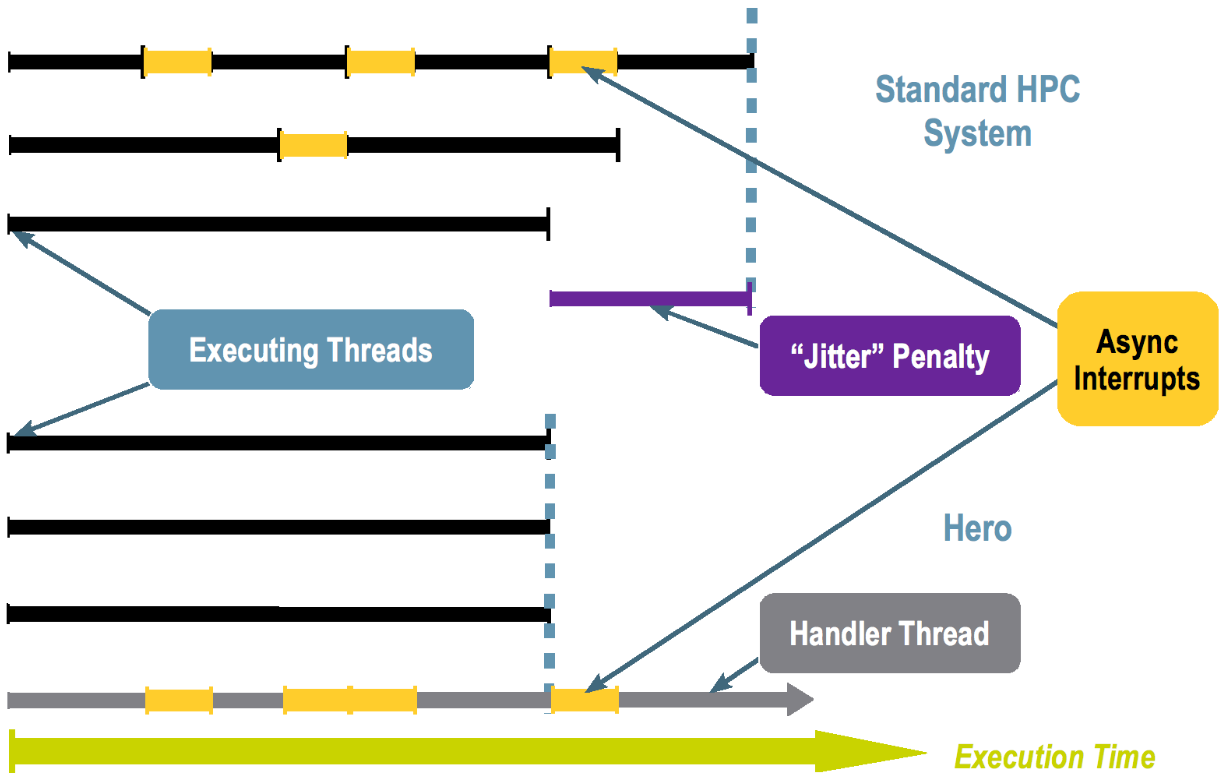


Figure 33: Application jitter

As shown in Figure 33, Hero approaches application jitter in a different fashion. Hero uses space sharing instead of time sharing, relying on its many execution threads, abundant memory, and bandwidth resources. All asynchronous system events can be handled by OS threads running on a dedicated microcore. With handler threads on a dedicated microcore, application threads are not preempted and continue to run. In addition, since each microcore has its own L1 cache, application threads do not suffer from the nondeterministic effects of periodic L1 cache pollution. Further, since Hero has more threads (integer execution units) than floating point units, the performance impact of the space-sharing approach caused by resource consumption is minimized. In the worst case, where a full microcore must be dedicated to handling events on each node, this represents a very small percentage of the compute resources in a system with hundreds or thousands of nodes. Hero can also support the traditional time-sharing approach to minimizing jitter, if needed.

11.4 Administrative support

Automation and virtualization (a form of abstraction) throughout the Hero software stack enable much more effective management of system resources than has been historically possible for machines at extreme scale. The Administrative Environment leverages the Hero system's fundamental properties to address traditional bottlenecks. These mechanisms are managed by tools that are designed to reduce staff burden and increase overall system utilization.

As with the system software, the intent of the administrative software is to create a familiar management environment for system operators. For example, the decision to run a complete version of Hero Solaris at every node, rather than attempting a single system image approach, means that many existing tools for managing large, multi-node OpenSolaris systems can be easily adapted for Hero. This same decision also makes developing new administrative tools simpler than other approaches such as managing the system's extreme scale through hierarchical decomposition.

The Administrative Environment can produce very high application completion rates because of resource virtualization, predictive self-healing mechanisms, and automatic system checkpointing for dealing with hardware failures. Automated failure recovery reduces software development costs associated with application-level checkpointing and eliminates administrative costs associated with restarting large jobs.

In much the same way as it handles failures, the Administrative Environment leverages underlying resource virtualization to support job configuration with minimal operator intervention. These underlying system services enable a dynamic job roll-in/roll-out mechanism that can suspend, relocate, and restart applications under automatic control in the name of improved overall system resource utilization. For example, large overheads associated with draining job queues when switching between capacity and capability modes can be replaced by highly dynamic job management that adapts to changing workloads without loss of resource utilization.

Job scheduling procedures can be highly automated in the Administrative Environment. In fact, they must be automated to allow administrative staff to manage job streams at the scale supported by a machine of this size. Because job and resource management are efficient and dynamic, job scheduling can be implemented as a hybrid of highly automated scheduling based on relatively static policies, as well as contingent scheduling based on management decisions, job progress, and other externalities.

Finally, the Administrative Environment can access system performance data at every level of the stack, which supports monitoring of nearly every aspect of system performance. Such continuous feedback creates an environment of ongoing improvement of resource utilization for designing application software, job management algorithms, and policy management.

12 Development support: languages, tools, skills

From the perspective of technology change (Section 3.10), Hero is a transitional system. It is designed to carry the HPC community through a fundamental shift in the way software is created, applied, and maintained. This is a difficult but necessary shift, without which the dramatic productivity gains anticipated by the HCPS program cannot be realized.

This shift starts with the status quo: an extremely skilled community with a decades-long legacy of applications that were extraordinarily expensive to develop and are increasingly expensive to maintain as computing platforms evolve. Much of the expense associated with these applications derives from concerns that are not essential to the scientific problems being addressed, but are instead related to the details of specific machine architectures (for example, distributed memory models and parallelism) and to external contingencies (for example, application-level checkpointing as a defense against frequent system failure).

The end point of this shift is a computational environment in which the execution model is abstract enough that it can survive many machine generations and coherent enough that a great many computational requirements can be automated. In this environment, tools can be constructed that permit problems to be solved in terms of the scientific mission and relevant mathematics. Furthermore, the execution model and development tools are portable across platforms and across time, which amplifies the value of the skills of people using them.

12.1 Legacy applications and languages

Hero is an ideal platform for technology migration. The global shared memory model, including system-wide load/store (Section 10.1), combined with services provided by Unification Zones and SuperZones, enables the system to directly support established distributed programming models such as MPI (Section 11.3.2). The same characteristics also provide excellent support for emerging PGAS languages and ultimately, for high-programmability programming models.

Applications based on established technologies, such as Fortran or a mixture of Fortran and C/C++ (Section 3.10.1), can be run in a SuperZone as though in a very large shared-memory processor. Software ensures the memory coherence required by the language. Parallelism can be introduced by the compiler or with OpenMP directives. In this model, data and thread placement are entirely under the control of runtime software with guidance from compiler directives. Object code for access to a remote memory location is identical to that for local memory, so code can be loaded and run on any portion of the system, from a single node to a SuperZone.

The emerging PGAS family of languages (Co-Array Fortran [42], UPC [61], and Titanium [19], Section 3.10.2) represents an intermediate step in HPC programming language evolution, since these languages continue to express data sharing and placement explicitly. Hero architecture is heavily influenced by the design of PGAS languages, and consequently allows them to be implemented directly rather than through a communication library. Access to shared variables is accomplished by direct addressing, while collective and barrier operations are performed using active messages. Any operation that requires coherence is somewhat inefficient, but such operations are likely to be infrequent because they are mostly used for debugging and other special cases. PGAS implementations can also run using Hero as a standard cluster. The transformation between SuperZone model and cluster model can be handled at load time using historical data and a binary optimizer, although this involves more substantial changes to object code than incorporating specialized libraries.

12.2 Standard development tools

The transition to new computing platforms depends crucially on tool support. For many of the reasons mentioned in the previous section, existing software development tools can run in a Hero SuperZone, which presents a complete Hero Solaris execution environment on a (virtual) shared-memory processor.

For example, existing OpenSolaris compilers share a common infrastructure that provides highly effective optimization, code generation, and tool support. They all support the following:

- Auto-parallelization of computational loops
- OpenMP support
- Cross-file interprocedural optimization, including automatic inlining
- Debugging support
- Post-link optimization
- Automatic tuning system for choosing optimizations
- Feedback-directed optimization
- Compiler commentary to guide performance tuning

Additional support necessary to ensure memory coherence across node boundaries (managed coherence, Section 10.1) is supplied automatically by compilers in the form of additional support libraries that are transparent to programmers. Ongoing performance and scalability improvement in legacy application support derives from continuing work in refinement of support libraries and increasing sophistication in collaboration between tools and virtual software operating environments.

Standard support for debugging in the transitional Hero environment comes from two sources. The Sun™ Studio dbx debugger provides multithreaded debugging within a node or a SuperZone; it offers the expected debugger features, including the following:

- Conditional breakpoints
- Watch points (variable change)
- Runtime checking (stray memory reference)
- Fix and continue (modify a function and keep running without restarting)

The dbx debugger works well with any shared memory model, including OpenMP and auto-parallelized programs. The second debugging resource comprises existing debugging tools for distributed memory programming that can also be easily adapted to run on the Hero MPI compatibility platform.

12.3 Highly productive programming

A highly productive programming model requires application codes that are:

- Domain-focused

- Inherently independent of underlying hardware and software architectures
- Free of difficult, error-prone distractions such as optimization, decomposition, parallelization, and checkpointing

As with more general software engineering challenges, the key strategies are abstraction and automation.

In the HPC community, examples can be found of programming technologies that move in this direction, though none are sufficient to achieve the necessary revolution in productivity:

- MATLAB from Mathworks [37] and Mathematica from Wolfram Research [64] offer languages and development environments with a focus on programming at higher levels of abstraction, but at considerable compromise in scaling and performance.
- Auto-parallelizing compilers, such as those supported on OpenSolaris, successfully automate some kinds of shared memory parallelism within some scale limitations.
- The PGAS family of languages automates some communication aspects of distributed memory programming when compared with MPI, but with very little additional abstraction.

Experience with all of these emerging technologies is promising in the areas of abstraction and automation, although they are not as widely used as one might expect. For example, the experiments mentioned in Section 3.7 involved rewriting a family of HPC standard benchmarks from MPI into a high-programmability style using tools that have been available for years. Code size dropped by an order of magnitude, and a currently available auto-parallelizing compiler for OpenSolaris produced execution times only 2x greater than the original in many cases, up to about 100 processors [35]. Adoption of such technologies has been slow for reasons discussed in Section 3.10.

Hero is designed to enable much more progress in these directions than has been possible on current and evolutionary systems. At its most basic level, Hero repairs fundamental computational abstractions that have been abandoned in the name of expensive manual tuning for performance. Those missing abstractions include:

- Single, global address space
- Virtualized, and thus interchangeable, hardware resources such as processors
- System-level ability to compute correctly through failure

Restoring each of these abstractions reduces the complexity of developing software, which in turn makes it possible to accelerate progress of automation in such crucial areas as parallelization and data layout. This encourages a new generation of highly abstract, possibly domain-dependent languages that can achieve necessary scaling and performance.

12.4 Fortress

Experiments in the design of such languages are underway. For example, Fortress provides a higher level of abstraction that more closely resembles scientific and mathematical expression, allowing scientists to program in a familiar notation and easily express inherent algorithm parallelism [2]. A powerful library mechanism addresses the automation of tedious and error-prone coding tasks such as array partitioning. This includes features that support concurrent execution such as parallel "for" loops and reduction operators such as summation (Σ), as well as atomic and transactional blocks. Early implementations of Fortress run on contemporary systems, but growing Fortress into its full potential involves extensive automation similar to that for which Hero is designed. Importantly, the Fortress effort is an open source project designed to draw on expertise and insights from a wide community of stakeholders.

Many Fortress language features are designed specifically with productivity in mind. For example:

- Fortran 90–style array operations reduce programming errors by reducing the amount of notation needed to express common patterns of computation on arrays (experiment reported in Section 3.7.2).
- Syntax designed to approximate traditional mathematical notation as closely as possible, with attention paid to how an integrated development environment (IDE) can further reformat code into traditional mathematical style, promises to make verifying scientific programs less error-prone (experiment reported in Section 3.7.1).
- Built-in numeric types designed to align with mathematics used in scientific problems include floating-point numbers of many sizes, imaginary and complex numbers, intervals of real numbers, signed and unsigned integers of fixed size, integers of unlimited size, and rational numbers with no size limits on numerators or denominators.
- Automatic dimensional analysis that statically detects programming errors missed by ordinary type checking.
- Extended aggregate types expressed using mathematical syntax and concepts, including arrays, vectors, matrices, tensors, sets, multisets, lists, maps, and hash tables.
- Powerful reduction operations over aggregates and over arbitrary parameterized expressions reduce programming errors.
- All operations, including control structures, fundamentally designed for parallelism.
- Transactional memory constructs promise to make programs more robust in the presence of failure than using locks.
- User-definable data distributions give programmers more control over data placement, including dynamic redistribution.
- Automatic storage management relieves programmers of the need to track and explicitly deallocate memory.
- Fortress code interoperates with other commonly used programming languages.

Furthermore, Fortress is fundamentally an extensible language. Extensions, implemented by the Fortress language mechanism, play two significant productivity-related roles. First, the library is where core functionality, including mediation between code and hardware resources, is implemented. At this level, many powerful HPC functions can be implemented, including syntax and type extensions, fine-grained control over parallelism, special purpose solvers, contract and invariant checking, automatic testing support, and many more. Second, extensibility invites the creation of libraries whose purpose is to create domain-specific programming languages with concepts, syntax, types, and solvers that are highly specialized for particular problem classes.

12.5 Portability: applications, tools, and skills

As the productivity team studies revealed, the lack of portability that represents such a significant perceived bottleneck in the HPC community is not just a matter of application and architectural features, but also includes tools and programming skills.

Hero is designed to address the familiar aspects of portability, as discussed in the previous section. High-programmability code is by definition abstract, eliminating confounding requirements that are not essential to the problems being solved: data decomposition, parallelization, and checkpointing. This makes the code highly portable only if other elements of the environment are also portable: tools and the skills acquired by programmers using those tools.

A good example of this phenomenon is an ongoing preference in the HPC community for FORTRAN 77 over Fortran 90/95. Project managers report that this choice is dictated by lack of confidence that high-quality Fortran 90/95 compilers will be available on all future platforms for 20 to 30 years, the anticipated lifetime of successful HPC applications.

This phenomenon extends to many other kinds of development tools, where valid concerns about future availability conspire to make software development tools be seen as risks to project success, rather than a pathway to productivity (Section 3.8). In fact, there is no path to a productivity revolution without tools, and in particular without portable tools that make people and skills portable, as well [62]. This observation is based on data from the productivity studies reported earlier and it applies to a wide variety of technologies that support HPC programming: languages, compilers, libraries, analyzers, and many more.

Part of Hero's design includes embracing cross-platform open source development tools in as many areas as possible. This means that HPC programming skills, once acquired, become as portable as high-programmability applications. In addition, a shared, standard tool suite is subject to ongoing research and improvement in performance, scalability, and correctness, rather than the ongoing "wheel reinvention" now experienced by the HPC programming community.

13 The interplay of design decisions

This section provides an example of how the design process described in Section 6 was used to ensure that the design described in Sections 7-12 satisfied Hero requirements. The requirement with the most pervasive influence on Hero's design was the need to support both new and existing HPC programming models. Although the ultimate architectural solution is described as a straightforward combination of new technologies and innovations, this solution was the product of a multiyear effort that included many false starts and design tradeoffs. It required a complete reconsideration of supercomputer design, with a focus on the three design tenets described in the introduction: focus on whole system properties, rethink system layers, and leverage new technologies.

The Hero design objective was to create a system that would satisfy the DARPA 10x productivity improvement challenge. Productivity requirements indicated that the best way to achieve this goal was to create the façade of a single system environment by using abstraction and automation. This would allow application developers, system programmers, and operations personnel to use familiar tools and processes to program and operate a petascale system almost as easily as if it were an SMP machine. Within such an environment:

- Highly productive programming models could eventually replace MPI.
- The system could achieve a significant fraction of its rated maximum performance without extensive tuning.
- The level of effort required to develop and maintain applications could drop by an order of magnitude.

Unfortunately, the most obvious ways to create this single-machine façade ran straight into a wall of technological limitations. This created the dilemma described in Section 13.1—finding an alternative that was technologically feasible while retaining the important productivity advantages of a single-system environment. Fortunately, new technologies were becoming available in the Hero time frame that helped solve this issue. These technologies and additional design constructs developed during the HCPS Phase II program are described in Section 13.2. Finally, Section 13.3 describes how new technologies were integrated into a single-system view that satisfies DARPA productivity requirements.

13.1 The dilemma

The team's initial assumption was that supporting a highly productive HPC programming model would require a fully cache-coherent, SMP-style machine with a single OS instance running across the machine to provide a single-system image. Hardware engineers set about designing the coherence mechanism, while the performance modeling team worked on performance estimates. Cache coherence for hundreds of nodes, even with Hero's high-bandwidth and low-latency interconnect, proved a daunting challenge. Detailed engineering work revealed that full cache coherence across the entire machine would fail to meet several system constraints, most notably cost, time to market, and performance. In addition, the complexity of a single-system image on a petascale system was vexing, and there were serious doubts about its robustness.

Distributed shared memory (DSM) was explored as an alternate way to implement a petascale SMP-style machine. A review of the literature, combined with analyses of several possible schemes, showed that this solution would fail to achieve adequate scalability and performance. Scalability issues arose from the need to track entries for all pages on all nodes in the system. A petascale memory system would require mapping tables that consume a substantial fraction of each node's memory. Furthermore, bandwidth utilization and latency of page updates would create significant issues at extreme scale.

Caught between the productivity-based requirement for a new programming model and the inadequacy of prior shared-memory schemes, the team revisited the productivity data and analyses that informed this requirement. A more detailed analysis (Section 10.1) revealed that the actual requirement was weaker than full system-wide cache coherence. The productivity bottleneck could be broken by providing a combination of global addressability and the ability to make the machine fully coherent at well-defined points in the computation (see Table 11). This would permit a programming model extremely well aligned with OpenMP parallelism, the PGAS family of languages, and new high-programmability languages, without the cost or complexity of full hardware cache coherence or DSM's performance and scaling problems.

13.2 Technology enablers

Although the analyses showed that global cache coherence was not required, new hardware and software constructs were still needed to enable global addressability and global virtual memory capability (global shared memory), along with mechanisms to support the limited form of cache coherence required (managed coherence). This required many innovative hardware capabilities that became available in the Hero time frame and led to the development of many other hardware and software mechanisms. These innovations and their roles in supporting global shared memory with managed coherence include the following:

- Programmers are reluctant to use global shared memory on a large system, due to long cross-system memory latencies that cripple performance. The Hero *massive chip multithreading (CMT) microprocessor* (Section 8.1) keeps hardware resources busy even in the presence of stalls, high-latency memory operations, and other processing irregularities. This provides dramatic performance improvements for HPC applications with long cross-system memory latencies and makes global shared memory feasible.
- Productivity benefits of a globally shared memory architecture will be lost if remote access is so slow that it requires programmers to expend significant effort maximizing locality. The Hero *high-bandwidth, low-latency system interconnect* (Section 9) provides the fast remote access among compute nodes necessary to enable highly productive, global shared-memory programming.
- The *SIF ASIC* (Section 8.2) provides hardware support for a fully shared global address space, with extra features to improve the performance of legacy applications and virtualization support for robustness. It performs the memory controller's function for all physical memory addresses not resident on the node where the request originated. The SIF makes address translation for remote memory accesses transparent to the application (Section 10.2.1), which allows the application to use standard shared-memory semantics for remote accesses. In addition, SIF functionality helps hide the latency penalty of random access fetches across the system interconnect.

- The typical Hero system application consumes the resources of many nodes, so its process virtual address space spans many nodes. The innovative *global system virtual address space* (Section 10.2) resides between physical memory and the multi-node process virtual address space. By unifying all process data spread across nodes, it acts as a single federated view of global memory for the multi-node process, so that shared-memory programming semantics can be used.
- The *SuperZone* software mechanism (Section 11.2) is a key technology for producing a system-wide application environment without the complexity of either fully cache-coherent SMP hardware or a single-system image across the entire machine. A SuperZone is a multi-node set of computational resources that acts as a container for a multi-node application. It comprises a set of Unification Zones: containers that reside only in a single node but have built-in multi-node semantics. Multi-node support is provided through the interposition of these multi-node semantics on system calls, kernel upcalls, and signals. Operations that require the participation of more than one node invoke interzone communication mechanisms in the SuperZone.
- A *coherence fence* (Section 10.3) is a synchronization mechanism used to enforce managed coherence: memory coherence at designated points in the application. The coherence fence is a special form of memory barrier that makes sure all stores from all threads must complete and become visible to all threads before any thread exits the fence. A simple implementation flushes all dirty cache entries and invalidates all cached data before any thread leaves the fence. To implement the coherence fence, Hero supports remote cache-flush and cache-invalidate operations. The application is fully cache coherent when it exits a coherence fence.

13.3 A single-system view

Hero's hardware and software mechanisms, notably the ones described in the previous section, collaborate to create a single-system view for application software. This view supports the Hero execution model (shared global memory with managed coherence) and provides the architectural support necessary for a highly productive programming environment. Figure 34 shows how the Hero system architecture supports a single-system view using the following layers:

- The *interconnect layer* provides fast access to remote memory.
- The *node layer* provides transparent address translation for remote memory using SIF and processing resources that hide cross-system memory latencies.
- The *fault tolerant hypervisor (FTH) layer* supports the creation of Unification Zones (UZs) and SuperZones by providing robust access to hardware resources.
- The Hero Solaris *OS layer* provides UZs that interpose on standard system calls to provide multi-node semantics for operations such as memory allocation and mapping. Note that there can be multiple OS instances in the OS layer (two are shown in Figure 34) because a guest OS could be running an application in a Hero node, although it would not participate in a SuperZone.
- In the *zone layer*, the SuperZone, stitched together from individual UZs on each node, provides a virtual execution environment (or container) for program execution. The

application sees only the SuperZone, even though it is actually running as local threads on many nodes.

The SuperZone mechanism, using the SIF to provide global addressability, presents a single view of the system to applications, while enabling independent operating systems on individual nodes. UZs also provide mechanisms for sharing information such as file handles across the SuperZone without requiring each node to map files to the same ID. Interposition allows the global ID to be mapped to the appropriate local ID on each node. This gives applications transparent access to the system's full memory and I/O resources without the complexity and overhead of a single OS managing the entire system. Thus, Hero satisfies programming model productivity requirements *without* full system cache coherence and *without* a single-system image.

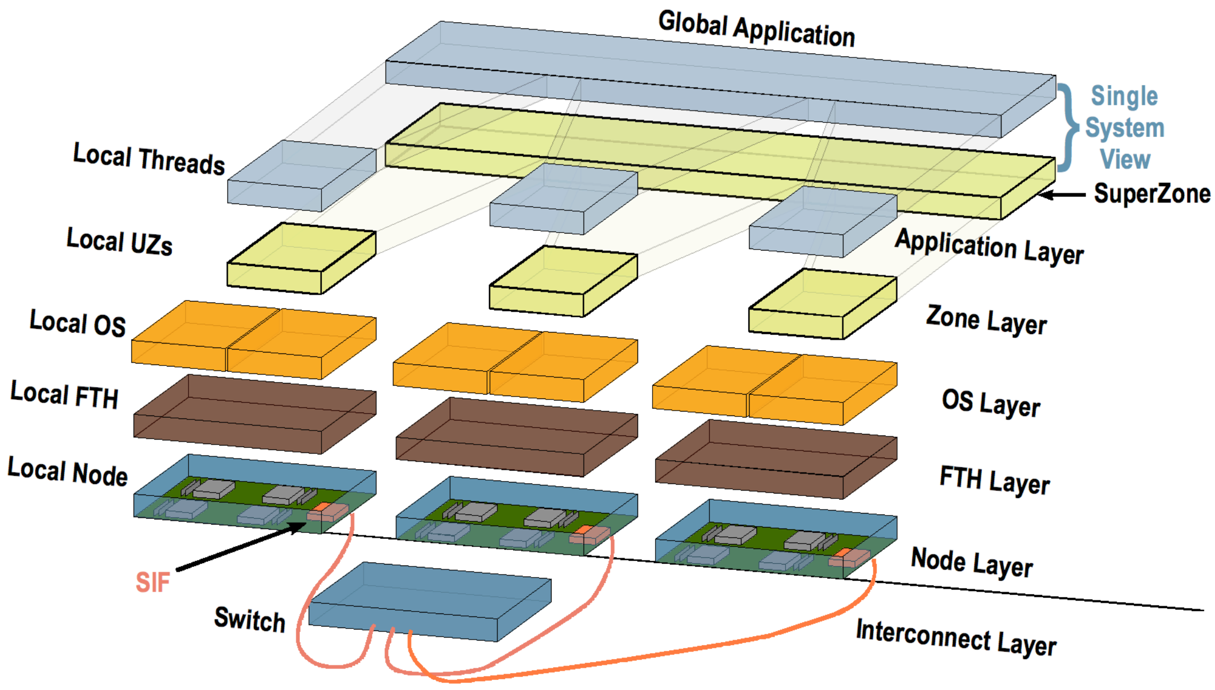


Figure 34: Application single-system view

Figure 34 illustrates the complex interdependence of Hero architectural features necessary to satisfy DARPA's 10x productivity goal. Such a design could not have been created without an integrated design team that focused on whole-system properties and completely redesigned system layers in order to exploit new technologies.

14 Conclusions

Recognizing that increasing hardware performance is no longer sufficient to drive the productivity gains needed by the HPC community, the DARPA HPCS Phase II challenge was to develop petascale computing systems that deliver at least 10x more productivity. Sun's response to that challenge was to study the entire nature of supercomputer productivity, identify productivity bottlenecks, and use the results of these studies to guide the design of a revolutionary supercomputer that could achieve the desired productivity improvement.

Although the HPCS community has traditionally regarded productivity as either a hardware or programming issue, Sun's analyses revealed that real productivity is a system-wide problem. It is not a property of isolated aspects of hardware or software processes, but of the way these work together over time, meaning that productivity must be considered as a whole-system property. Considering productivity as a system-wide problem led to systematic analyses of supercomputer productivity in its full context: this includes people, organizations, goals, practices, and skills in addition to processors, disks, memory, and software.

Sun's analyses identified a wide range of productivity bottlenecks in system hardware and software, system administration, software development practices, and runtime environments. Viewed as a systems problem, however, two overarching issues stand out: *expertise* and *effort*. The expertise bottleneck reflects the fact that developing software for current HPC systems requires expertise in several distinct, complex disciplines and that such expertise cannot be acquired without long experience. The effort bottleneck reflects the fact that development, parallelization, verification, validation, porting, and maintenance of HPC applications are now largely manual tasks for which the development platform, execution, and administrative environments provide little effective assistance.

These analysis results illustrate why conventional approaches (such as increasing hardware performance) have failed to solve the productivity problem. This is summarized in the Hero system strategic goal: to provide a system that significantly reduces the development effort and level of expertise required to achieve a given level of machine utilization. Though simple in its expression, this goal implies paradigm shifts in the capabilities provided by supercomputers and in the skills and practices required to develop and deploy HPC applications. This goal demands a productivity-driven, top-to-bottom reevaluation of supercomputer hardware and software design.

To implement this strategy, Sun created an interdisciplinary, highly collaborative design process. The productivity team, of which the authors were members, combined expertise in computer hardware, system software, software engineering, computational science, and cultural anthropology. Following Sun's iterative System Exploration Model, design teams worked in collaboration with one another and with the productivity team to make critical design tradeoffs. Hero's design drew heavily on emerging technologies, such as those in the areas of chip interconnect and resource virtualization, to make possible an innovative memory architecture for high-productivity programming.

The Hero design represents a systematic application of two strategic design principles: *abstraction* and *automation*. At the user level, this implies providing languages, tools, and a runtime environment that abstract from machine details and allow scientists to write and maintain programs in terms of problem domains (for example, fluid mechanics). At the system and software layers, this implies tools for automating routine tasks such as memory management, parallelization, and job control. At the hardware layer, this implies new approaches to memory management, localization, and reliability.

Gordon Bell once said: “The fastest, most reliable and least expensive components of a computer system are those that aren’t there.” It could also be said that the fastest, most reliable, and least expensive code is that which doesn’t need to be written. Sun’s approach was to eliminate programming tasks that scientist-programmers may be insufficiently prepared to perform and that distract from their most important goal: advancing science.

15 Acknowledgements

The authors are grateful to all members of Sun's HPCS productivity team, as well as HPCS program colleagues who contributed to this effort. Particular thanks to collaborators in the case studies: Eugene Loh, Douglass Post, Richard Kendall and Walter Tichy, and to Tom Nash and Philip Johnson, whose work has deepened our understanding of the productivity problem.

This material is based upon work supported by DARPA under Contract No.NBCH3039002.

16 References

[1] S. Ahalt and K. Kelley, "Blue-Collar Computing: HPC for the Rest of Us," *ClusterWorld* **2**(11), November 2004. See also <http://www.osc.edu/bluecollarcomputing/>.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr., and S. Tobin-Hochstadt, *The Fortress Language Specification Version 1.0*, Sun Microsystems, Inc., 2008. <http://research.sun.com/projects/plrg/fortress.pdf>

[3] M. Ardis, N. Daley, D. Hoffman, H. Siy, and D. Weiss, "Software product lines: a case study," *Software Practice and Experience* **30**(7) June 2000, pp. 825-84.

[4] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems," *CTWatch Quarterly*, **2**(4B), November 2006 B. <http://www.ctwatch.org/quarterly/articles/2006/11/designing-scalable-synthetic-compact-applications-for-benchmarking-high-productivity-computing-systems>

[5] B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[6] F. Brooks, *The Mythical Man Month (Anniversary Edition)*, Addison-Wesley, 1995.

[7] J. Carver, R. Kendall, S. Squires, and D. Post, "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies." In *Proceedings of the 29th international Conference on Software Engineering* (May 20–26, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington DC, pp. 550-559.

[8] B. Chamberlain, D. Callahan, and H. Zima. "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, August 2007, **21**(3): 291-312.

[9] B. Chamberlain, S. J. Deitz, and L. Snyder, "A comparative study of the NAS MG benchmark across parallel languages and architectures," *Proceedings of the ACM Conference on Supercomputing*, 2000.

[10] A. Cockburn, *Agile Software Development*, Addison-Wesley, 2001.

[11] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004.

[12] Defense Advanced Research Project Agency (DARPA) Information Processing Technology Office, High Productivity Computing Systems (HPCS) Program. <http://www.darpa.mil/ipto/programs/hpcs/>

[13] E. Dijkstra, "The structure of the 'THE' multiprogramming system." *Communications of the ACM* **11**(5) May 1968, pp. 341-346.

[14] Robert Drost, R. D. Hopkins, Ron Ho, Ivan Sutherland, "Proximity communication," *IEEE Journal of Solid-State Circuits*, **39**(9), pp. 1529-1535, September 2004.

[15] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, and L. Votta, "Measuring High Performance Computing Productivity," *International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity* (ed. Kepner), vol. 18, no. 4, Winter 2004, pp. 459-473.

[16] E. Goldratt, *Theory of Constraints*, North River Press, 1999.

[17] J. Gosling B. Joy, G. Steele, and G. Bracha, *Java Language Specification, Third Edition*, Addison Wesley, 2005.

[18] S. Graham, and M. Snir, "The NRC Report on the Future of Supercomputing," *CTWatch Quarterly*, **1**(1), February 2005. <http://www.ctwatch.org/quarterly/articles/2005/02/nrc-report/>

[19] Hilfinger, P. N., Bonachea, D., Gay, D., Graham, S., Liblit, B., Pike, G., and Yelick, K. 2001 *Titanium Language Reference Manual*. Technical Report. UMI Order Number: CSD-01-1163, University of California at Berkeley.

[20] C. Holland, DoD Research and Development Agenda for High Productivity Computing Systems (White Paper), Pentagon, Washington DC, June 2001.

[21] D. Hopkins, A. Chow, R. Bosnyak, B. Coates, J. Ebergen, S. Fairbanks, J. Gainsley, R. Ho, J. Lexau, F. Liu, T. Ono, J. Schauer, I. Sutherland, and R. Drost, "Circuit techniques to enable 430 Gb/s/mm/mm proximity communication," *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2007, pp. 368-369.

[22] IBM, "X10: The New Concurrent Programming Language for Multicore and Petascale Computing." <http://x10-lang.org/>

[23] P.M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita, "Improving software development management through software project telemetry," *IEEE Software*, **22**(4), July–Aug. 2005, pp. 76-85.

[24] P. Johnson and M. Paulding. "Understanding HPCS development through automated process and product measurement with Hackystat," *Proceedings of the Second Workshop on Productivity and Performance in High-End Computing*, San Francisco, California, 2005.

[25] R. Kendall, J. Carver, A. Mark, D. Post, S. Squires, and D. Shaffer, *Case Study of the Hawk Code Project*, Los Alamos National Laboratory Report LAUR-05-9011, 2005.

[26] R. Kendall, A. Mark, D. Post, S. Squires, and C. Halverson, *Case Study of the Condor Code Project*, Los Alamos National Laboratory Report LA-UR-05-9291, 2005.

[27] R. Kendall, Douglass Post, Susan Squires, and Jeff Carver, *Case Study of the Eagle Code Project*, Los Alamos National Laboratory, Report LA-UR-06-1092, 2006.

[28] R. Kendall, Douglass Post, and Andrew Mark, *Case Study of the NENE Code Project*, CMU/SEI-2006-TN-044, Software Engineering Technical Note, January 2007.

[29] R. Kendall, Jeff Carver, David Fisher Dale Henderson, Andrew Mark, Douglass Post, Cliff Rhodes and Susan Squires, "Development of a Weather Forecasting Code: A Case Study," *IEEE Software* **25**(4), July–August 2008, pp. 59-65.

[30] J. Kepner, "HPC Productivity: an Overarching View," *International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity* (ed. Kepner), **18**(4), Winter 2004, pp. 393-397.

[31] J. Kepner, "HPC Productivity Model Synthesis," *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity* **18**(4), November 2004.

[32] G. K. Konstadinidis, et al., "Architecture and Physical Implementation of a Third Generation 65 nm, 16 core, 32 Thread Chip-Multithreading SPARC Processor," *IEEE Journal of Solid-State Circuits*, **44**(1), January 2009.

[33] A. S. Leon, et al., "A Power-Efficient High Throughput 32-Thread SPARC Processor," *IEEE Journal of Solid-State Circuits*, **42**(1), January 2007.

[34] C. Leiserson, "Fat-trees: Universal networks for hardware efficient supercomputing," *IEEE Transactions on Computers*, **34**(10), October 1985.

[35] E. Loh, M. Van De Vanter and L. Votta, "Can Software Engineering Solve the HPCS Problem?" *Second International Workshop on SE HPC Sys Applications,* May 2005.

[36] 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*, March 2005. <http://icl.cs.utk.edu/hpcc/pubs>

[37] The Mathworks, MATLAB. <http://www.mathworks.com/>

[38] MPI Forum, *MPI: A Message-Passing Interface Standard, version 1.1*, June 1995. <http://www.mpi-forum.org/docs/docs.html>

[39] MPI Forum, *MPI: A Message-Passing Interface Standard, version 2.1*, July 2008. <http://www.mpi-forum.org/docs/docs.html>

[40] Declan Murphy, Thomas Nash, Lawrence Votta, Jr., and Jeremy Kepner, "A System-wide Productivity Figure of Merit," *CTWatch Quarterly*, **2**(4B), November 2006 B. <http://www.ctwatch.org/quarterly/articles/2006/11/a-system-wide-productivity-figure-of-merit/>

[41] NASA, The *NAS Parallel Benchmarks (NPB),* NASA Advanced Supercomputing Division. <http://www.nas.nasa.gov/Resources/Software/npb.html>

[42] Numrich, R. W. and Reid, J. 1998. Co-array Fortran for parallel programming, *SIGPLAN Fortran Forum*, **17**(2) August 1998, pp. 1-31.

[43] OpenJDK, "The HotSpot Group." <http://openjdk.java.net/groups/hotspot/>

[44] OpenMP <http://www.openmp.org/>

[45] OpenSolaris <http://www.opensolaris.org>

[46] C. Pancake, *A Collaborative Effort in Parallel Tool Design*. Technical Report. UMI Order Number: 94-80-14, Oregon State University, 1994. Also appeared in *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, J. Dongarra and B. Tourancheau eds., SIAM, 1994, pp. 112-119.

[47] C. Pancake, *Establishing Standards for HPC System Software and Tools.* Technical Report. UMI Order Number: 97-60-04, Oregon State University, 1997.

[48] D. Post, R. Kendall, and E. Whitney, "Case Study of the Falcon Code Project," *Proceedings Second International Workshop on Software Engineering for High Performance Computing System Applications*, St. Louis, Missouri, May 15, 2005.

[49] D. Post and L. Votta, "Computational Science Demands a New Paradigm", *Physics Today,* **58**(1), 2005, pp. 35-41.

[50] D. Post and R. Kendall, "Large-Scale Computational Scientific and Engineering Project Development and Production Workflows," *USE OF HIGH PERFORMANCE COMPUTING IN METEOROLOGY, Proceedings of the Twelfth ECMWF Workshop*, Reading, U.K. October 30 – November 3, 2006, edited by George Mozdzynski, World Scientific Publishing Co. October, 2007, p. 284. ISBN 978-981-277-588-7. An abbreviated version appeared as *CTWatch Quarterly*, **2**(4B), November 2006 B. <http://www.ctwatch.org/quarterly/articles/2006/11/large-scale-computational-scientific-and-engineering-project-development-and-production-workflows/>

[51] Reflex Photonics, "Light on board OE-ASIC." http://www.reflexphotonics.com/light-on-board-oe-asic.htm

[52] V. Sarkar, C. Williams, and K. Ebcioglu, "Application Development Productivity Challenges for High-End Computing", *First Workshop on Productivity and Performance in High-End Computing*, Madrid, Spain, 2004.

[53] J. Segal, "Some Problems of Professional End User Developers", *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. VLHCC. IEEE Computer Society, Washington, DC, September 23-27, 2007, pp. 111-118.

[54] S. Squires, M. Van De Vanter, and L. Votta, "Software Productivity Research In High Performance Computing," *CTWatch Quarterly*, **2**(4A), November 2006 A. <http://www.ctwatch.org/quarterly/articles/2006/11/software-productivity-research-in-high-performance-computing/>

[55] S. Squires, M. Van De Vanter, and L. Votta, "Yes, There Is an 'Expertise Gap' In HPC Applications Development," *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing* (PPHEC'06), Austin, Texas, February 12, 2006.

[56] S. Squires, W. Tichy and L. Votta. "What Do Programmers of Parallel Machines Need? A Survey." *Proceedings of the Second Workshop on Productivity and Performance in High-End Computing,* San Francisco, California, 2005.

[57] T. Sze, M. Giere, B. Guenin, N. Nettleton, D. Popovic, J. Shi, S. Bezuk, R. Ho, R. Drost, and D. Douglas, "Proximity Communication Flip-Chip Package with Micron Chip-to-chip Alignment Tolerances," *Electronic Components and Technology Conference (ECTC)*, San Diego, California, May 26–29, 2009.

[58] M. Tremblay and S. Chaudhry, "A Third-Generation 65 nm 16-core 32-thread plus 32-scout-threads SPARC Processor," *IEEE ISSCC Dig. Tech. Papers*, February 2008, pp. 82-83.

[59] Transactional Memory, Sun Microsystems. http://research.sun.com/spotlight/2007/2007-08-13_transactional_memory.html

[60] R. Trotter and J. Schensul, "Methods in Applied Anthropology," *Handbook of Methods in Cultural Anthropology*, H. Russell Bernard (ed.), Walnut Creek, California, Altamara Press, 1999.

[61] UPC Consortium, *UPC Language Specifications, v1.2*, Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005.

[62] M. Van De Vanter, D. Post and M. Zosel, "HPC Needs a Tool Strategy," *Second International Workshop on SE HPC Systems Applications,* May 2005.

[63] L. Wang, K. Pattabiraman, Z. Kalbarczyk, R. Iyer, L. Votta, C. Vick, "Modeling Coordinated Checkpointing for Large-Scale Supercomputers," *International Conference on Dependable Systems*, June 2005.

[64] Wolfram Research, Mathematica <http://www.wolfram.com/>.

[65] A. Wood, S. Nathan, T. Tsai, C. Vick, L. Votta, "Multi-Tier Checkpointing for Peta-Scale Systems," *International Conference on Dependable Systems*, June 2005.

[66] R. Yu, *Case Study Research: Design and Methods*, SAGE Publications, 2002.

[67] X. Zheng, J. Lexau, D. Rolston, J. Cunningham, I. Shubin, R. Ho, A. Krishnamoorthy, "BGA Package Integration of Electrical, Optical, and Capacitive Interconnects," *Electronic Components and Technology Conference (ECTC)*, San Diego, California, May 26–29, 2009.

17 NAS BT code modification experiment

An experienced HPC programmer was asked to add new functionality to one of the benchmarks (NAS BT I/O) rewritten in the experiment described in 3.7.2, as might be done during the maintenance phase of an application's life cycle. The task was to add checkpoint-style I/O in several different programming models:

- *High-programmability style with Fortran 90*: Exploiting simplifications made during the previous experiment, the important state is contained in a single array for which high-level array operations are available.
- *Serial FORTRAN 77*: Standard FORTRAN I/O for reading and writing array contents.
- *MPI – simple*: Parallel I/O with naïve use of the MPI I/O API.
- *MPI – optimized*: Parallel I/O optimized use of the MPI/IO API, with collective I/O operations.

The task required supporting four operations: setup, write, read, and close. Table 12 (repeated from Section 3.7.2) shows the amount of code required for the task, using each of the four programming models. The total lines of code (LOC) required for each model give a rough measure of the code's complexity and the expected lifetime maintenance cost for this segment.

Table 12: NAS BT I/O code modification - lines of code

Programming Model	Setup	Write	Read	Close	Total LOC
High-programmability w/F90	1	1	1	1	4
Serial FORTRAN 77	7	19	20	1	47
MPI - simple	25	22	23	1	71
MPI - optimized	144	12	13	1	170

The code for each of these styles follows.

17.1 High-programmability style with Fortran 90

Setup

```
open(20,file="btio.dat",status="unknown",form="unformatted")
```

Write

```
write(20) u   ! write entire array, including boundary cells
```

Read

```
read(20) u    ! read entire array, including boundary cells
```

Close

```
close(20)
```

17.2 Serial FORTRAN 77

Setup

```
      if (node.eq.root) record_length = 40/fortran_rec_sz
      call mpi_bcast(record_length, 1, MPI_INTEGER,
     >                root, comm_setup, ierr)

      open (unit=99, file=filenm,
     $      form='unformatted', access='direct',
     $      recl=record_length)
```

Write

```
 do cio=1,ncells
     do kio=0, cell_size(3,cio)-1
         do jio=0, cell_size(2,cio)-1
             iseek=(cell_low(1,cio) +
$                   PROBLEM_SIZE*((cell_low(2,cio)+jio) +
$                   PROBLEM_SIZE*((cell_low(3,cio)+kio) +
$                   PROBLEM_SIZE*idump)))

             do ix=0,cell_size(1,cio)-1
                 write(99, rec=iseek+ix+1)
$                      u(1,ix, jio,kio,cio),
$                      u(2,ix, jio,kio,cio),
$                      u(3,ix, jio,kio,cio),
$                      u(4,ix, jio,kio,cio),
$                      u(5,ix, jio,kio,cio)
             enddo
         enddo
     enddo
 enddo
```

Read

```
   do cio=1,ncells
     do kio=0, cell_size(3,cio)-1
         do jio=0, cell_size(2,cio)-1
             iseek=(cell_low(1,cio) +
$                   PROBLEM_SIZE*((cell_low(2,cio)+jio) +
$                   PROBLEM_SIZE*((cell_low(3,cio)+kio) +
$                   PROBLEM_SIZE*ii)))


             do ix=0,cell_size(1,cio)-1
                 read(99, rec=iseek+ix+1)
$                      u(1,ix, jio,kio,cio),
$                      u(2,ix, jio,kio,cio),
$                      u(3,ix, jio,kio,cio),
$                      u(4,ix, jio,kio,cio),
$                      u(5,ix, jio,kio,cio)
             enddo
         enddo
     enddo
   enddo
```

Close

```
close(unit=99)
```

17.3 MPI - simple

Setup

```
      integer ierr

      iseek=0

      if (node .eq. root) then
          call MPI_File_delete(filenm, MPI_INFO_NULL, ierr)
      endif

      call MPI_Barrier(comm_solve, ierr)

      call MPI_File_open(comm_solve,
     $          filenm,
     $          MPI_MODE_WRONLY + MPI_MODE_CREATE,
     $          MPI_INFO_NULL,
     $          fp,
     $          ierr)

      call MPI_File_set_view(fp,
     $          iseek, MPI_DOUBLE_PRECISION, MPI_DOUBLE_PRECISION,
     $          'native', MPI_INFO_NULL, ierr)

      if (ierr .ne. MPI_SUCCESS) then
          print *, 'Error opening file'
          stop
      endif
```

Write

```
 do cio=1,ncells
     do kio=0, cell_size(3,cio)-1
         do jio=0, cell_size(2,cio)-1
             iseek=5*(cell_low(1,cio) +
$                   PROBLEM_SIZE*((cell_low(2,cio)+jio) +
$                   PROBLEM_SIZE*((cell_low(3,cio)+kio) +
$                   PROBLEM_SIZE*idump)))

             count=5*cell_size(1,cio)

             call MPI_File_write_at(fp, iseek,
$                  u(1,0,jio,kio,cio),
$                  count, MPI_DOUBLE_PRECISION,
$                  mstatus, ierr)

             if (ierr .ne. MPI_SUCCESS) then
                 print *, 'Error writing to file'
                 stop
             endif
         enddo
     enddo
 enddo
```

Read

```
   do cio=1,ncells
     do kio=0, cell_size(3,cio)-1
         do jio=0, cell_size(2,cio)-1
             iseek=5*(cell_low(1,cio) +
$                   PROBLEM_SIZE*((cell_low(2,cio)+jio) +
$                   PROBLEM_SIZE*((cell_low(3,cio)+kio) +
$                   PROBLEM_SIZE*ii)))

             count=5*cell_size(1,cio)

             call MPI_File_read_at(fp, iseek,
$                  u(1,0,jio,kio,cio),
$                  count, MPI_DOUBLE_PRECISION,
$                  mstatus, ierr)

             if (ierr .ne. MPI_SUCCESS) then
                 print *, 'Error reading back file'
                 call MPI_File_close(fp, ierr)
                 stop
             endif
         enddo
     enddo
   enddo
```

Close

```
call MPI_File_close(fp, ierr)
```

17.4 MPI - optimized

Setup

```
      integer ierr
      integer combined_ftype
      integer mstatus(MPI_STATUS_SIZE)
      integer sizes(4), starts(4), subsizes(4)
      integer cell_btype(maxcells), cell_ftype(maxcells)
      integer cell_blength(maxcells)
      integer info
      character*20 cb_nodes, cb_size
      integer c
      integer cell_disp(maxcells)

       call mpi_bcast(collbuf_nodes, 1, MPI_INTEGER,
     >                root, comm_setup, ierr)

       call mpi_bcast(collbuf_size, 1, MPI_INTEGER,
     >                root, comm_setup, ierr)

       if (collbuf_nodes .eq. 0) then
          info = MPI_INFO_NULL
       else
          write (cb_nodes,*) collbuf_nodes
          write (cb_size,*) collbuf_size
          call MPI_Info_create(info, ierr)
          call MPI_Info_set(info, 'cb_nodes', cb_nodes, ierr)
          call MPI_Info_set(info, 'cb_buffer_size', cb_size, ierr)
          call MPI_Info_set(info, 'collective_buffering', 'true', ierr)
       endif

       call MPI_Type_contiguous(5, MPI_DOUBLE_PRECISION,
     $                          element, ierr)
       call MPI_Type_commit(element, ierr)
       call MPI_Type_extent(element, eltext, ierr)

       do  c = 1, ncells
```

```
c
c Outer array dimensions ar same for every cell
c
         sizes(1) = IMAX+4
          sizes(2) = JMAX+4
          sizes(3) = KMAX+4
c
c 4th dimension is cell number, total of maxcells cells
c
         sizes(4) = maxcells
c
c Internal dimensions of cells can differ slightly between cells
c
          subsizes(1) = cell_size(1, c)
          subsizes(2) = cell_size(2, c)
          subsizes(3) = cell_size(3, c)
c
c Cell is 4th dimension, 1 cell per cell type to handle varying
c cell sub-array sizes
c
         subsizes(4) = 1

c
c type constructors use 0-based start addresses
c
          starts(1) = 2
          starts(2) = 2
          starts(3) = 2
         starts(4) = c-1

c
c Create buftype for a cell
c
          call MPI_Type_create_subarray(4, sizes, subsizes,
     $          starts, MPI_ORDER_FORTRAN, element,
     $          cell_btype(c), ierr)
c
c block length and displacement for joining cells -
c 1 cell buftype per block, cell buftypes have own displacment
c generated from cell number (4th array dimension)
c
         cell_blength(c) = 1
          cell_disp(c) = 0

       enddo
c
c Create combined buftype for all cells
c
       call MPI_Type_struct(ncells, cell_blength, cell_disp,
     $            cell_btype, combined_btype, ierr)
       call MPI_Type_commit(combined_btype, ierr)

       do  c = 1, ncells
c
c Entire array size
c
          sizes(1) = PROBLEM_SIZE
          sizes(2) = PROBLEM_SIZE
          sizes(3) = PROBLEM_SIZE

c
c Size of c'th cell
c
```

```
           subsizes(1) = cell_size(1, c)
           subsizes(2) = cell_size(2, c)
           subsizes(3) = cell_size(3, c)

c
c Starting point in full array of c'th cell
c
           starts(1) = cell_low(1,c)
           starts(2) = cell_low(2,c)
           starts(3) = cell_low(3,c)

           call MPI_Type_create_subarray(3, sizes, subsizes,
     $          starts, MPI_ORDER_FORTRAN,
     $          element, cell_ftype(c), ierr)
           cell_blength(c) = 1
           cell_disp(c) = 0
       enddo

       call MPI_Type_struct(ncells, cell_blength, cell_disp,
     $            cell_ftype, combined_ftype, ierr)
       call MPI_Type_commit(combined_ftype, ierr)

       iseek=0
       if (node .eq. root) then
          call MPI_File_delete(filenm, MPI_INFO_NULL, ierr)
       endif


      call MPI_Barrier(comm_solve, ierr)

       call MPI_File_open(comm_solve,
     $          filenm,
     $          MPI_MODE_WRONLY+MPI_MODE_CREATE,
     $          MPI_INFO_NULL, fp, ierr)

       if (ierr .ne. MPI_SUCCESS) then
             print *, 'Error opening file'
             stop
       endif

      call MPI_File_set_view(fp, iseek, element,
     $       combined_ftype, 'native', info, ierr)

       if (ierr .ne. MPI_SUCCESS) then
             print *, 'Error setting file view'
             stop
       endif
```

Write

```
 integer mstatus(MPI_STATUS_SIZE)
 integer ierr

 call MPI_File_write_at_all(fp, iseek, u,
$           1, combined_btype, mstatus, ierr)
 if (ierr .ne. MPI_SUCCESS) then
     print *, 'Error writing to file'
     stop
 endif

 call MPI_Type_size(combined_btype, iosize, ierr)
 iseek = iseek + iosize/eltext
```

Read

```
  integer mstatus(MPI_STATUS_SIZE)
  integer ierr

  call MPI_File_read_at_all(fp, iseek, u,
$           1, combined_btype, mstatus, ierr)
  if (ierr .ne. MPI_SUCCESS) then
     print *, 'Error reading back file'
     call MPI_File_close(fp, ierr)
     stop
  endif

  call MPI_Type_size(combined_btype, iosize, ierr)
  iseek = iseek + iosize/eltext
```

Close

```
call MPI_File_close(fp, ierr)
```

18 Glossary

One lesson of the interdisciplinary research that characterized Sun's participation in the HPCS program is that there is no single common vocabulary among participants, an issue that is exacerbated by a focus on very advanced technologies. The following list describes common terms used in the research and design leading to Hero.

Active messages: asynchronous mechanism in Hero that delivers a (executable) message to a remote node for execution on a thread in the receiving node

Administrative Environment (AE): software layer at the boundary of the Hero system software and the Hero development environment; it provides tools that help manage the system and tools that help improve application execution

Applications: computer programs that are most directly related to the objectives in using a computer—for example, simulating fluids or molecules—in contrast to other computer software such as middleware or system software; referred to in the HPC community as codes

Application jitter: performance degradation that results when system software disrupts even a few threads or processes in an application with tight synchronization

Application-specific integrated circuit (ASIC): complex integrated circuit customized for a particular use

BrandZ: in OpenSolaris, extends the OpenSolaris Zones infrastructure to create Branded Zones; zones that contain non-native operating environments [45]

Cache coherency: consistency of data when its value is no longer stored in a single location (main memory), but is replicated and cached throughout a large system for performance reasons

Cache-coherent shared memory (CCSM): commonly understood set of properties provided by most SMP machines and relied upon for correct execution of programs using a form of threaded concurrency

Capability mode: in HPC, using a large computing resource for few large jobs, each of which demands the large resource; cf. capacity mode

Capacity mode: in HPC, using a large computing resource for many small jobs, none of which demands such a large resource for itself; cf. capability mode

Checkpointing: storing an entity's state (for example, thread, application or system) for use in error recovery or other resource remapping

Chip multiprocessor (CMP): placing multiple microcores on a single die

Chip multithreading (CMT): combining CMP microprocessors with multithreading

Co-Array Fortran: an emerging PGAS language based on FORTRAN [42], see partitioned global address space <http://www.co-array.org/>

Code, codes: what the HPC community calls programs, cf. Applications

Coherence fence: a special barrier mechanism that enforces full coherence at designated points in a program

Content-addressable memories (CAM): special type of computer memory designed to search its entire memory in a single operation

Cultural Anthropology: systematic study of culture, a complex phenomenon that includes interrelated economic systems, political systems, social organizations, and belief systems; applied in this research to understanding the context in which supercomputers are used

Defense Advanced Research Projects Agency (DARPA) <http://www.darpa.gov/>

Development environment: collection of software tools, libraries, and other structure to support software development

Distributed shared memory (DSM): system implementation in which each node of a cluster has access to a large shared memory, in addition to each node's limited, nonshared private memory

dbx: standard Sun Studio debugger, used for multithreaded debugging within a Hero node or SuperZone

Execution model: hardware architectural support for a programming model

Exploratory programming: in software engineering, describes a kind of software development where requirements are not given *a priori*, but whose development is part of the development's goal (observed in this report to describe scientific programming characteristic of the mission partners)

Fault Management Architecture: OpenSolaris technology that aggregates fault information and performs prediction, diagnosis, and healing [45]

Fault tolerant hypervisor: in OpenSolaris, the lowest software layer, supports the creation of Hero Unification Zones and SuperZones by providing robust access to hardware resources; see OpenSolaris, Unification Zones, SuperZones

Floating point operations per second (FLOPS), also FLOP: measure of computer performance

FORTRAN 77: serial, universally supported version of Fortran

Fortran 90, Fortran 95: modernized versions of Fortran

Fortress: Sun's new programming language, provides a higher level of abstraction more closely resembling scientific and mathematical expression [2] <http://projectfortress.sun.com/Projects/Community>

Global addressability: a program's ability to read or write any memory location in a large, physically distributed computer system with the same simple instructions, whether or not that location is local to the instruction's issuance

Hackystat: tool that collects data from multiple streams on a server and can produce many different kinds of activity reports and analyses spanning many time frames [23] <http://code.google.com/p/hackystat/>

Hero: Sun's revolutionary petascale supercomputer, designed to meet DARPA's HPCS requirements

Hero Solaris: the Hero operating system, based on OpenSolaris and enhanced with features to support increased robustness and a limited set of multi-node semantics for applications, including SuperZones and Unification zones; see OpenSolaris, SuperZones, Unification zones

High Performance Computing (HPC)

High Productivity Computing Systems (HPCS): DARPA program [12]

IDE: integrated development environment

I/O: input/output; often referring to network or storage devices

Interconnect: a mechanism for passing electrical signals among computer chips

Jitter: see application jitter

Life cycle: refers to the steps in creating and maintaining a product or application

LOC: lines of code; approximate measure of the code's complexity

Luxtera: Sun partner during HPCS Phase II; see silicon photonics <http://www.luxtera.com/>

MATLAB: programming language characterized by extensive use of higher level (more abstract) library functions that are specialized for the kind of scientific and numerical programming common in HPC applications [37] <http://www.mathworks.com/>

Message passing interface (MPI): application programmer interface for distributed memory programming that uses message-oriented communication between computational nodes together with protocol and semantic specifications for how its features must behave (the predominant HPC parallel programming model today) [38] [39] <http://www.mpi-forum.org/docs/docs.html>

Microcore: core set of computing resources such as a floating point pipeline; a chip multiprocessor (CMP) usually contains several microcores that may share some resources such as caches

Mission partners: organizations that plan to use the HPCS-produced supercomputers, including the U.S. Departments of Energy and Defense

Multi-node: multiple nodes in a single system, zone or domain

Multiprocessor: multiple processors on a single chip (CMP)

Multithreading: ability to execute multiple software threads on a single microcore, sharing the microcore computing resources

NAS: NASA Advanced Supercomputing division

NAS BT: NAS parallel benchmark using a block-tridiagonal solver to simulate fluid flow

NAS BT I/O: NAS parallel benchmark based on NAS BT but adding an I/O component

NAS CG: NAS parallel benchmark using a conjugate-gradient method to solve a sparse linear system

NAS MG: NAS parallel benchmark testing the performance of a multigrid solver

NAS parallel benchmarks (NPB): set of programs designed by NASA's Advanced Supercomputing division to help evaluate performance of parallel supercomputers <http://www.nas.nasa.gov/Resources/Software/npb.html>

Nonuniform memory access (NUMA): computer memory design where memory access time depends on the memory's location relative to the accessing processor

OpenMP: existing and ubiquitous HPC shared-memory parallel programming model based on Fortran and C/C++ language, using explicit directives to specify concurrency [44] <http://www.openmp.org/>

OpenSolaris: open source version of the Solaris Operating System [45] <http://www.opensolaris.org>

Parallel programming: developing software that will, at execution time, be characterized by many computational processes taking place concurrently

Partitioned global address space (PGAS): family of emerging languages for HPC programming (Co-Array Fortran [42], Titanium [19] and UPC [61]), characterized by explicit expression of distributed-memory parallelism as per-node private memory, combined with globally shared memory

Petascale: adjective that loosely refers to the ability to perform a quadrillion (10^{15}) operations per second

Port, porting: migration of an application from one computing platform to another, often requiring substantial modification in the HPC community

Productivity (DARPA HPCS program goals): ability to develop and deploy high-performance supercomputer applications at acceptable time and cost [30]

Productivity (economics): value divided by cost of goods or services produced

Productivity (supercomputer): holistic metric of a computer's usefulness, taking into account its real value to end users and all the costs associated with acquiring, maintaining, and using it

Programming model: programming language together with its execution, memory, and parallelism semantics as exposed to the programmer; cf. execution model

Proximity communication: Sun's technology that uses capacitive coupling (extreme proximity without direct physical interconnection) between pairs of neighboring chips to enable very low-power, low-latency, high-bandwidth communication [14] [21]

Reliability, availability, serviceability (**RAS**)

Remote load/store: operations to load data from or store data to remote memory locations in a large, physically distributed computer system

Safepoints: locations in a program where it is safe for the program to stop, as determined by global data flow and control-flow analysis

Scalability interface (SIF): special-purpose ASIC in Hero that provides hardware support for a fully shared global address space

Scale: in HPC, refers to the degree of parallelism achieved by an application, expressed as the number of processors that can be used effectively; as a verb, to modify an application to achieve a higher degree of parallelism

Silicon photonics: technology that performs transmission, amplification, detection, modulation, and switching of light on silicon; WDM silicon photonics switches multiple wavelengths (colors) of light on silicon [51] [67]; see Luxtera

Single-system image: provides applications with a single view of all system services, which usually means that there is a single OS image running across the entire (multi-node) system; cf. single-system view

Single-system view: provides applications with a single view of all computation resources—memory, files, and I/O, but does not support all system services; cf. single-system image

Space sharing: running multiple jobs concurrently on a large computer, each one on a physically disjoint portion of the system; cf. time sharing

SuperZone: a software construct in Hero Solaris: stitched together from individual Unification Zones (UZs) to provide a virtual execution environment (or container) for program execution

Switch: hardware component of a chip interconnect that routes signals from source to destination

Switching fabric: topological arrangement of a set of switches to interconnect all compute and I/O nodes

Symmetric multiprocessing (SMP): multiprocessor computer architecture where two or more identical processors (or microcores) connect to a single shared main memory; usually implies cache coherence

System Exploration Model (SEM): Sun's HPCS Phase II process for concurrent, highly collaborative design

Time sharing: running multiple jobs on a large computer by having them take turns accessing computational resources; cf. space sharing

Titanium: an emerging PGAS language, based on the Java programming language [19]; see partitioned global address space

Total store ordering (TSO): guarantee that all load/store operations from a given processor (or microcore) appear in memory in the same order they were issued by the processor

Transactional memory: a program's ability to perform a set of instructions as a single atomic unit [59]

Unification Zone (UZ): Hero Solaris software construct in a Hero node that interposes on standard system calls to provide multi-node semantics for operations such as memory allocation and mapping

UPC: an emerging PGAS language, based on the C programming language [61]; see partitioned global address space <http://upc.gwu.edu/>

Utilization: also machine utilization; a proxy metric for productivity commonly used in the HPC community, defined as a percentage of potentially available floating point operations consumed by an application during execution

Validation: checking that an application achieves its intended objective, for example, correctly simulating some physical phenomenon

Verification: checking that a computer component (for example, a software program) behaves according to its design, correctly implementing a prescribed algorithm

Wavelength division multiplexing (WDM), see silicon photonics

Workflow: description of possible tasks and their sequences in the use of a computer system

19 Sun HPCS Phase II publications

19.1 Productivity

Stuart Faulk, John Gustafson, Philip M. Johnson, Adam Porter, Walter F. Tichy, Lawrence G. Votta, "Measuring HPC Productivity," *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity*, J. Kepner (editor), **18**(4), Winter 2004 (November).

John Gustafson, "Purpose-Based Benchmarks," *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity*, J. Kepner (editor), **18**(4), Winter 2004 (November).

Douglass Post, Lawrence G. Votta, "Computational Science Requires a New Paradigm," *Physics Today* **58**(1), pp. 35-41, January 2005.

Susan Squires, Walter F. Tichy, Lawrence G. Votta, "What Do Programmers of Parallel Machines Need? A Survey," *Second Workshop on Productivity and Performance in High-End Computing* (P-PHEC), San Francisco, California, February 13, 2005.

Philip M. Johnson and Michael G. Paulding, "Understanding HPC Development through Automated Process and Product Measurement with Hackystat," *Second Workshop on Productivity and Performance in High-End Computing* (P-PHEC), San Francisco, California, February 13, 2005.

Eugene Loh, Michael L. Van De Vanter, and Lawrence G. Votta, "Can Software Engineering Solve the HPCS Problem?" *Proceedings Second International Workshop on Software Engineering for High Performance Computing System Applications*, St. Louis, Missouri, May 15, 2005.

Michael L. Van De Vanter, Douglass Post, Mary Zosel, "HPC Needs a Tool Strategy," *Proceedings Second International Workshop on Software Engineering for High Performance Computing System Applications*, St. Louis, Missouri, May 15, 2005.

Richard Kendall, Jeff Carver, Andrew Mark, Douglass Post, Susan Squires, Dolores Shaffer, *Case Study of the Hawk Code Project*, Los Alamos National Laboratory, Report LA-UR-05-9011, 2005.

Richard Kendall, Douglass Post, Susan Squires, Christine Halverson, *Case Study of the Condor Code Project*, Los Alamos National Laboratory Report LA-UR-05-9291, 2005.

Susan Squires, Michael L. Van De Vanter, and Lawrence G. Votta, "Yes, There Is an 'Expertise Gap' in HPC Application Development," *Third Workshop on Productivity and Performance in High-End Computing* (P-PHEC), Austin, Texas, February 12, 2006.

Russell Brown, Ilya Sharapov, "Parallelization of a Molecular Modeling Application: Programmability Comparison Between OpenMP and MPI," *Third Workshop on Productivity and Performance in High-End Computing* (P-PHEC), Austin, Texas, February 12, 2006.

Richard Kendall, Douglass Post, Susan Squires, and Jeff Carver, *Case Study of the Eagle Code Project*, Los Alamos National Laboratory, Report LA-UR-06-1092, 2006.

Susan Squires, Michael L. Van De Vanter, and Lawrence G. Votta, “Software Productivity Research In High Performance Computing," *CTWatch Quarterly*, **2**(4A), November 2006 A. <http://www.ctwatch.org/quarterly/articles/2006/11/software-productivity-research-in-high-performance-computing/>

Declan Murphy, Thomas Nash, Lawrence Votta, Jr., and Jeremy Kepner, “A System-wide Productivity Figure of Merit,” *CTWatch Quarterly*, **2**(4B), November 2006 B. <http://www.ctwatch.org/quarterly/articles/2006/11/a-system-wide-productivity-figure-of-merit/>

Jeffrey C. Carver, Richard P. Kendall, Susan Squires, Douglass E. Post, “Software Development Environments for Scientific and Engineering Software: A Series of Case Studies,” *Proceedings of the 29th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, pp. 440-559, May 20–26, 2007.

Richard Kendall, Jeffrey C. Carver, David Fisher, Dale Henderson, Andrew Mark, Douglass Post, Clifford E. Rhoades Jr., Susan Squires, "Development of a Weather Forecasting Code: A Case Study," *IEEE Software* **25**(4), July–August 2008, pp. 59-65.

19.2 Hardware

19.2.1 Proximity communication

Robert Drost, Craig Forrest, Bruce Guenin, Ron Ho, Ashok V. Krishnamoorthy, Danny Cohen, John E. Cunningham, Bernard Tourancheau, Arthur Zingher, Alex Chow, Gary Lauterbach, and Ivan Sutherland. “Challenges in Building a Flat-Bandwidth Memory Hierarchy for a Large-Scale Computer with Proximity Communication,” *Proceedings of the 13th Symposium on High Performance interconnects*, pp. 13-22, August 17–19, 2005, HOTI, IEEE Computer Society, Washington, DC.

Robert Drost, R. D. Hopkins, Ron Ho, Ivan Sutherland, "Proximity communication," *IEEE Journal of Solid-State Circuits*, **39**(9), pp. 1529-1535, September 2004.

Robert Drost, Ron Ho, R. D. Hopkins, Ivan Sutherland, "Electronic alignment for proximity communication," *Digest of Technical Papers. ISSCC. 2004 IEEE International Solid-State Circuits Conference*, pp. 144-518 Vol.1, pp. 15-19, February 2004.

Robert Drost, R. D. Hopkins, Ivan Sutherland, “Proximity Communications,” *IEEE Custom Integrated Circuits Conference*, pp. 469-472, September 2003.

Ron Ho, Jonathan Gainsley, Robert Drost, “Long Wires and Asynchronous Control,” *Proceedings 10th International Symposium on Asynchronous Circuits and Systems*, pp. 240-249, April 19—23, 2004.

Jon Lexau, Xuezhe Zheng, Jonathan Bergey, Ashok V. Krishnamoorthy, Ron Ho, Robert Drost, John E. Cunningham, “CMOS Integration of Capacitive, Optical, and Electrical Interconnects," *Proceedings International Interconnect Technology Conference* (IITC), pp. 78-80, June 4–6, 2007.

Ron Ho, Tarik Ono, Frankie Liu, R. D. Hopkins, Alex Chow, Justin Schauer, Robert Drost, "High-Speed and Low-Energy Capacitively-Driven On-Chip Wires," *IEEE International Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers*, pp. 412-612, February 11–15, 2007.

Ron Ho, Tarik Ono, Frankie Liu, R. D. Hopkins, Alex Chow, Justin Schauer, Robert Drost, "High-Speed and Low-Energy Capacitively-Driven Wires," *IEEE Journal of Solid State Circuits*, **43**(1), pp. 52-60, January 2008.

R. D. Hopkins, Alex Chow, Robert Bosnyak, Bill Coates, Jo Ebergen, Scott Fairbanks, Jonathan Gainsley, Ron Ho, Jon Lexau, Frankie Liu, Tarik Ono, Justin Schauer, Ivan Sutherland, Robert Drost, "Circuit Techniques To Enable 430 Gb/s/mm^2 Proximity Communication," *Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 368-9, February 2007.

Jo Ebergen, Alex Chow, Bill Coates, Justin Schauer, R. D. Hopkins, "An asynchronous high-throughput control circuit for proximity communication," *12th IEEE International Symposium on Asynchronous Circuits and Systems, 2006.* pp. 9-33, March 13–15, 2006.

Alex Chow, R. D. Hopkins, Ron Ho, Robert Drost, "Measuring 6D Chip Alignment in Multi-Chip Packages," *2007 IEEE Sensors,* pp.1307-1310, October 28–31, 2007.

19.2.2 Optical interconnect

John E. Cunningham, Daniel Beckman, Xuezhe Zheng and Ashok V. Krishnamoorthy, "Scaling VCSEL performance for 100Terabits/s Systems," (invited paper) *Proceedings of the SPIE*, Volume 6124, pp. 204-214, 2006.

John E. Cunningham, David K. McElfresh, Leon D. Lopez, Dan Vacar, and Ashok V. Krishnamoorthy, "Scaling Vertical-Cavity Surface-Emitting Laser Reliability for Petascale Systems," *Applied Optics* **45**(25), pp. 6342-48, September 2006.

John E. Cunningham, Ashok V. Krishnamoorthy and Maxim Abashin, Kazuhiro Ikeda, Chia-Ho Tsaiand, Yeshaiahu Fainman, Uriel Levy, "Design, fabrication and characterization of subwavelength based slab lens in Silicon," *Optical Society of America Topical Meeting on Nanophotonics*, Connecticut, April 2006.

John E. Cunningham and Ashok V. Krishnamoorthy, "Silicon photonics for High For Productivity Computing Systems," Tech Report, October 16, 2005.

Ashok V. Krishnamoorthy and John E. Cunningham, "Optical interconnects for high-productivity computing systems," Tech Report, October 14, 2005 and presentation at OSA (invited talk) Frontiers in Optics Annual Meeting, Tucson, Arizona, October 2005.

Jon Lexau, Ron Ho, Robert Drost, Ashok V. Krishnamoorthy, John E. Cunningham, Jonathan Bergey, and Barmak Mansoorian, "LoPI (the Chip)," Sun Tech Report, August 26, 2006.

John E. Cunningham, D. Beckman, Xuezhe Zheng, Dawei Huang, Theresa Sze, Ashok V. Krishnamoorthy, "PAM-4 Signaling over VCSELs with 0.13µm CMOS," *Optics Express* **14**(25), pp. 12028-38, December 2006.

John E. Cunningham, D. Beckman, David K. McElfresh, Craig Forrest*, Danny Cohen*, and Ashok V. Krishnamoorthy, "Scaling VCSEL Reliability Up to 250Terabits/s of System Bandwidth," in *Adaptive Optics: Analysis and Methods/Computational Optical Sensing and Imaging/Information Photonics/Signal Recovery and Synthesis Topical Meetings on CD-ROM*, Technical Digest, Optical Society of America, 2005.

Uriel Levy, Yeshayahu Fainman, Ashok V. Krishnamoorthy and John E. Cunningham, “Novel Slab Lens Based on Artificial Graded Index Medium,” in *Adaptive Optics: Analysis and Methods/Computational Optical Sensing and Imaging/Information Photonics/Signal Recovery and Synthesis Topical Meetings on CD-ROM*, Technical Digest, Optical Society of America, 2005.

Ashok V. Krishnamoorthy, D. Huang, Theresa Sze, Robert Drost, Ron Ho, H. Davidson, and Rick Lytel, “Challenges and potentials for multiterabit-per-second optical transceivers,” *Biophotonics/Optical Interconnects and VLSI Photonics/WBM Microcavities, 2004 Digest of the LEOS Summer Topical Meetings*, pp. 28-30, June 2004.

D. Huang, Theresa Sze, Ashok V. Krishnamoorthy, D. Beckman, S. Fazelpour, H. Davidson, J. Cooley, and Rick Lytel, "The chip-multithreading architecture and parallel optical interconnects," *Biophotonics/Optical Interconnects and VLSI Photonics/WBM Microcavities, 2004 Digest of the LEOS Summer Topical Meetings*, pp. 28-30, June 2004.

Ashok V. Krishnamoorthy and Rick Lytel, “Interconnects for Large Scale Computing Systems,” (invited talk) IEEE IHSDS Workshop, Santa Fe, New Mexico, May 2004.

Xuezhe Zheng, Jon Lexau, Jonathan Bergey, John E. Cunningham, Ron Ho, Robert Drost, Ashok V. Krishnamoorthy, “Optical Transceiver Chips based on Co-Integration of Capacitively Coupled Proximity Interconnects and VCSELs,” *IEEE Photonics Technology Letters*, **19**(7), pp. 453-455, April 2007.

Uriel Levy, Maxim Abashin, Kazuhiro Ikeda, Ashok V. Krishnamoorthy, John E. Cunningham, and Yeshaiahu Fainman, “Inhomogenous Dielectric Metamaterials with Space-Variant Polarizability,” *Physical Review Letters*. **98**(15), June 2007.

19.2.3 Packaging

Xuezhe Zheng, Jon Lexau, David Rolston, John E. Cunningham, Ivan Shubin, Ron Ho, Ashok V. Krishnamoorthy, “BGA Package Integration of Electrical, Optical, and Capacitive Interconnects”, *Electronic Components and Technology Conference (ECTC)*, San Diego, California, pp. 26-29, May 2009.

T. Sze, M. Giere, B. Guenin, N. Nettleton, D. Popovic, J. Shi, S. Bezuk, R. Ho, R. Drost, and D. Douglas, “Proximity Communication Flip-Chip Package with Micron Chip-to-chip Alignment Tolerances”, *Electronic Components and Technology Conference (ECTC)*, San Diego, California, pp. 26-29, May 2009.

19.3 Software

19.3.1 Runtime executive

Christopher A. Vick, Michael Paleczny, “The Application of Virtual Machine Technology to Peta-Scale Systems,” (position paper) *Language Runtimes 2004 Workshop*.

L. Wang, Karthik Pattabiraman, Z. Kalbarczyk, Ravishankar K. Iyer, Lawrence G. Votta, Christopher A. Vick, Alan Wood, “Modeling coordinated checkpointing for large-scale supercomputers,” *Proceedings International Conference on Dependable Systems and Networks, 2005. DSN 2005*, pp. 812-821, June 28-July 1, 2005.

Alan Wood, Swami Nathan, Tim Tsai, Christopher A. Vick, Lawrence G. Votta, and Anoop Vetteth, “Multi-Tier Checkpointing for Peta-Scale Systems,” in *Supplemental Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, pp. 112-121, June 28-July 1, 2005, Yokohama, Japan.

19.3.2 Programming languages (Fortress)

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., Sam Tobin-Hochstadt, *The Fortress Language Specification Version 1.0*, Sun Microsystems, 2008. <http://research.sun.com/projects/plrg/fortress.pdf>

Joseph J. Hallett, *Semantics and Type Soundness Proof of a Core Fragment of Fortress with Hidden Type Variables*, Sun Microsystems, March 2007. <http://research.sun.com/projects/plrg/Publications/core-fortress.pdf>

Joseph J. Hallett, Sukyoung Ryu, *Formal Semantics for MCJ*, Sun Microsystems, June 2004. <http://research.sun.com/projects/plrg/Publications/mcj-semantics.pdf>

Sam Tobin-Hochstadt, Eric Allen, "A Core Calculus of Metaclasses," *Twelfth International Workshop on Foundations of Object-Oriented Languages*, Long Beach, California, 2005.

Eric Allen, Victor Luchangco, and Sam Tobin-Hochstadt, *Encapsulated Upgradable Components*, Sun Microsystems, April 2005. <http://research.sun.com/projects/plrg/Publications/components.pdf>

Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele Jr., "Object-Oriented Units of Measurement," *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, SIGPLAN Notices* **39**(10), pp. 384-403, October 2004.

Arvind and Jan-Willem Maessen, "Memory Model = Instruction Reordering + Store Atomicity," *Proceedings of the 33rd Annual international Symposium on Computer Architecture*, pp. 29-40, IEEE Computer Society, Washington, DC, June 17–21, 2006. .

Yossi Lev and Jan-Willem Maessen, "Towards a Safer Interaction with Transactional Memory by Tracking Object Locality," *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2004. <http://urresearch.rochester.edu/retrieve/4804/lev.pdf>

Jan-Willem Maessen, and Arvind, "Store Atomicity for Transactional Memory," *Multithreading in Hardware and Software: Formal Approaches, Design and Verification (TV06)*, Seattle, Washington. <http://www.cs.utah.edu/tv06/tv06proceedings.pdf>

Jan-Willem Maessen, and Arvind, "Store Atomicity for Transactional Memory," *Electron. Notes Theor. Comput. Sci.*, **174**(9), pp. 117-137, 2007.

19.3.3 Administration

David Vengerov, “A Reinforcement Learning Framework for Utility-Based Scheduling in Resource-Constrained Systems,” in *Future Generation Computer Systems* [to appear 2008]. Revised and extended version of the Sun Labs technical report TR-2005-141.

David Vengerov, “A Gradient-Based Reinforcement Learning Approach to Dynamic Pricing in Partially-Observable Environments,” *Future Generation Computer Systems*, **24**(7), pp. 687-693, July 2008.

David Vengerov, “A Reinforcement Learning Framework for Online Data Migration in Hierarchical Storage Systems,” *Journal of Supercomputing*, **43**(1), pp. 1-19, January 2008.

David Vengerov, “A Reinforcement Learning Approach to Dynamic Resource Allocation,” *Engineering Applications of Artificial Intelligence*. **20**(3), pp. 383-390, April 2007.

David Vengerov, Lykomidis Mastroleon, Declan Murphy and Nick Bambos, *Adaptive Data-Aware Utility-Based Scheduling in Resource-Constrained Systems*, Sun Labs Technical Report Number: TR-2007-164, April 23, 2007.

19.3.4 File system

Andrew Hastings and Alok Choudhary, “Exploiting Shared Memory to Improve Parallel I/O Performance,” in *Proceedings of the 13th European PVM/MPI Users Group Meeting*, Springer-Verlag, September 2006.

19.3.5 Visualization

http://www.tacc.utexas.edu/research/users/features/remotevis.php
Feature article about Maverick use for scientific research.

http://virtualgl.sourceforge.net/
VirtualGL open source site, offering documentation and downloads for VirtualGL and TurboVNC software.

http://chromium.sourceforge.net/
Chromium open source site, offering documentation and downloads.

19.3.6 Interval arithmetic

Eldon Hansen and G. William Walster, *Global Optimization Using Interval Analysis,* New York: Marcel Dekker, Inc., 2004.

Gregory R. Ruetsch, “An interval algorithm for multi-objective optimization.” *Structural and Multidisciplinary Optimization* **30**(1), pp. 27–37, 2005.

G. William Walster, Eldon Hansen, “Using pillow functions to efficiently compute crude range tests,” *Numerical Algorithms* **37**(1-4), pp. 401-415, December 2004.

Eldon Hansen and G. William Walster, “Solving Overdetermined Systems of Interval Linear Equations,” *Reliable Computing* **12**(3), pp. 239-243, June 2006. Also electronically published April 26, 2006. http://dx.doi.org/10.1007/s11155-006-7221-8

19.4 Performance analysis

Ilya Sharapov, Robert Kroeger, Guy Delamarter, Matthew Ramsay, Razvan Cherevesan. “A Case Study in Top-Down Performance Estimation for a Large-Scale Parallel Application,” In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 81-89, New York, New York, March 29–31, 2006. PPoPP ’06, ACM.

Russell A. Brown, David A. Case, “Second derivatives in generalized Born theory,” *Journal of Computational Chemistry* **27**(14), pp. 1662-1675, 2006.

Russell A. Brown, Ilya Sharapov, "High Scalability Parallelization of a Molecular Modeling Application: Performance and Productivity Comparison Between OpenMP and MPI Implementations," *International Journal of Parallel Programming* **35**(5), pp. 441-458, October 2007.

Lodewijk Bonebakker, Andrew Over, Ilya Sharapov, "Working Set Characterization of Applications with an Efficient LRU Algorithm," in *Formal Methods and Stochastic Models for Performance Evaluation*, Springer Lecture Notes in Computer Science **4054,** pp. 78-92, 2006.

Pramod Rustagi, Ilya Sharapov, "Projecting Performance of LS-DYNA Implicit for Large Multiprocessor Systems," *LS-DYNA International Users Conference*, 2006.

Razvan Cheveresan, Matthew Ramsay, Chris Feucht, Ilya Sharapov, "Characteristics of Workloads Used in High Performance and Technical," *21st ACM International Conference on Supercomputing (ICS '07)*, Seattle, Washington, June 2007.

Russell A. Brown, Ilya Sharapov, "Performance and Programmability Comparison Between OpenMP and MPI Implementations of a Molecular Modeling Application," in *OpenMP Shared Memory Parallel Programming*, Springer Lecture Notes in Computer Science **4315**, pp 349-360, 2008.

19.5 RAS architecture

Kenny C. Gross and Wendy Lu, "Early Detection of Signal and Process Anomalies in Enterprise Computing Systems," *Proc. of the IEEE International Conference on Machine Learning and Applications* (ICMLA), pp. 204-210, June 2002.

Timothy K. Tsai, Kalyan Vaidyanathan, Kenny C. Gross, "Low-Overhead Run-Time Memory Leak Detection and Recovery." *Proceedings of the 12th Pacific Rim international Symposium on Dependable Computing*, pp. 329-340, PRDC, IEEE Computer Society, Washington DC, December 18–20, 2006.

Long Wang, Karthik Pattabiraman, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Christopher A. Vick, Lawrence G. Votta, Alan Wood, "Modeling coordinated checkpointing for large-scale supercomputers," *Proceedings International Conference on Dependable Systems and Networks, 2005. DSN 2005,* pp. 812-821, June 28–July 1, 2005.

Alan Wood, Swami Nathan, Timothy K. Tsai, Christopher A. Vick, Lawrence G. Votta, Anoop Vetteth, "Multi-Tier Checkpointing for Peta-Scale Systems," *International Conference on Dependable Systems and Networking DSN2005*, Tokyo, Japan, June 28–July 1, 2005.

Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Long Wang, "Application fault tolerance with Armor middleware," *Internet Computing, IEEE* **9**(2), pp. 28-37, March–April 2005.

Alan Wood, Swami Nathan, "RAS by the Yard," *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '07,* pp. 606-611, June 25–28, 2007.

Qinghuai Gao, "Parallel Rendering Pipeline: Design, High Level Algorithms and Analysis," in Milestone 5 supplementary materials.

20 About the authors

Michael L. Van De Vanter was a member of Suns's HPCS productivity team, specializing in software development technologies and practices. His work at Sun Microsystems Laboratories focuses on highly productive developer tools: previously as Principal Investigator of the Jackpot Project, developing technologies that transferred into the NetBeans^{TM} IDE, and currently developing tools specialized for the development of a next generation Virtual Machine.

Alan Wood was Sun's HPCS RAS lead and fault tolerance architect. He is currently a Distinguished Engineer at Sun Microsystems Laboratories, doing research on switch architectures using proximity communications. His other research interests include energy-efficient reliability and energy utilization modeling. Previously, he created hardware and software reliability and availability models in several industries.

Christopher Vick was Sun's HPCS software architect and developed the Hero execution model. He is currently a Distinguished Engineer at Sun Microsystems Laboratories, investigating implementation of legacy ISAs such as SPARC and x64 by dynamic translation to a new, simpler ISA. His past work at Sun includes development of the HotSpot Java Virtual Machine and leading research in virtualization technologies.

Stuart Faulk was a consultant for Sun's HPCS productivity team. He is an Associate Research Professor in Computer and Information Science at the University of Oregon. His research interests include software productivity, software architecture, software product-lines, and assistive technology for the cognitively disabled.

Susan Squires was the anthropologist on Sun's HCPS productivity team tasked with tracking the workflow of HPC computational scientists, learning how HPC teams organizes the work, and investigating how organizational environments do or do not support the scientists and their team. She is now the senior social scientist at the Technology Research for Independent Living Centre, Trinity College Dublin.

Lawrence G. Votta, Jr. was the principal investigator, productivity lead, for Sun's HPCS program and a coauthor of Sun's HPCS phase II and phase III proposals. He is now working (through his consulting company, Brincos Inc.) on leveraging high performance computational engineering to manage the maintenance and evolution of complex systems. The Computational Research and Engineering Acquisition Tools and Environments program is in the third year of its ten year mission in the Modernization Office of the US Department of Defense.

Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025

Productive Petascale Computing: Requirements, Hardware, and Software

Van De Vanter, Wood, Vick, Faulk, Squires, Votta

SMLI TR-2009-183