ExaScale Software Study: Software Challenges in Extreme Scale Systems

Saman Amarasinghe
Dan Campbell
William Carlson
Andrew Chien
William Dally
Elmootazbellah Elnohazy
Mary Hall
Robert Harrison
William Harrod
Kerry Hill
Jon Hiller
Sherman Karp
Charles Koelbel
David Koester
Peter Kogge
John Levesque
Daniel Reed
Vivek Sarkar, Editor & Study Lead
Robert Schreiber
Mark Richards
Al Scarpelli
John Shalf
Allan Snavely
Thomas Sterling

September 14, 2009

This work was sponsored by DARPA IPTO in the ExaScale Computing Study with Dr. William Harrod as Program Manager; AFRL contract number **FA8650-07-C-7724**. This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings

NOTICE

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED.

Exascale Software Study: Software Challenges in Extreme Scale Systems

DISCLAIMER

The material in this document reflects the thoughts and opinions of the participants only, and not those of any of the universities, corporations, or other institutions to which they are affiliated.

The views, opinions, and/or findings contained in this article are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

FOREWORD

This document reflects the thoughts of a group of highly talented individuals from universities, industry, and government research labs on the software challenges that will need to be addressed for the Extreme Scale systems that are anticipated in the 2015 – 2020 time-frame. It was drawn from a study conducted over a series of seven meetings held from June 2008 to February 2009. The goal of the study was to examine the state of the art, identify key challenges, and outline elements of a technical approach that can address the challenges without prescribing specific solutions. The report was assembled from input provided by study participants and guests over this short period of time. As such, all inconsistencies reflect either misunderstandings by the editor or areas where there were difference of opinion among members of the team. There was, however, unanimous agreement about the key challenges that surfaced from the study, and the criticality of addressing the software challenges in conjunction with hardware challenges when developing future Extreme Scale systems.

I am honored to have been part of this study, and wish to thank the study members and guests for their dedication to the field of parallel software and systems, and for all their hard work in contributing to the study.

Vivek Sarkar, Editor and Study Lead
Rice University
September 14, 2009.

Contents

Chapter 1

Executive Summary

This report presents the findings and recommendations of the **Exascale Software Study** conducted from June 2008 to February 2009. A characterization of Extreme Scale systems can be found in the recent report on "Technology Challenges in Achieving Exascale Systems" [62]. This characterization identifies three distinct classes of systems:

- **Data-center-sized Exascale systems**, capable of delivering 1 ExaFlops or 1 ExaOps[1], which is 1,000× the capability of currently emerging Petascale data-center-sized systems.
- **Departmental-sized Petascale systems** that allow the capabilities of a Petascale system to be shrunk in size and power to fit within a few racks, allowing widespread deployment.
- **Embedded Terascale systems** that reduce Terascale capability to a few chips and a few ten's of watts, thereby enabling deployment in a range of embedded environments.

Since the first system class listed above achieves Exascale performance, the terms *Exascale* and *Extreme Scale* are often interchangeably in the community and in this report. However, whenever possible, we prefer to use *Extreme Scale* to refer to systems across all three classes and *Exascale* specifically for the largest data-center sized system class.

The focus of this study is on *software challenges for Extreme Scale systems*. The scope of software considered spans the spectrum of operating systems; runtimes for scheduling, memory management, communication, performance monitoring, power management, and resiliency; computational libraries; compilers; programming languages; and application frameworks. Though there are significant differences in the software requirements for the three classes of Extreme Scale systems, all of them share some critical challenges: they will be built using *massive multi-core processors* with 100's of cores per chip, their performance will be *driven by parallelism and constrained by energy*, and they will be subject to *frequent faults and failures*. Thus, the three key challenges for Extreme Scale software are *Concurrency*, *Energy Efficiency* and *Resiliency*. This study addresses the Concurrency and Energy Efficiency challenges, whereas the third challenge is addressed by a companion study on Exascale Resiliency. Development of Extreme Scale algorithms and applications as well as development of Extreme Scale hardware are outside of the scope of this study. However, identification of opportunities for software-hardware co-design, as well interfaces between applications and system software and between system software and hardware, are very much in scope.

The concurrency challenge is manifest in the need for software to expose at least 1000× more concurrency in applications for Extreme Scale systems, relative to current systems. It is further

[1]Following common usage, "ops" refers to operations per second in this report unless otherwise specified.

exacerbated by the projected memory-computation imbalances in Extreme Scale systems, with Bytes/Ops ratios that may drop to values as low as 10^{-2} where Bytes and Ops represent the main memory and computation capacities of the system respectively. These ratios will result in 100× reductions in memory per core relative to Petascale systems, with accompanying reductions in memory bandwidth per core. Thus, a significant fraction of software concurrency in Extreme Scale systems must come from exploiting more parallelism within the computation performed on a single datum *i.e.,* from strong scaling or from the "new-era" weak scaling discussed in Chapter 4. Strong scaling often involves more frequent communication and synchronization than weak scaling, which in turn contributes to the energy efficiency challenge since data movement and synchronization are major contributors to energy costs in Extreme Scale systems. Another major obstacle to achieving a large degree of concurrency arises from the serialization bottlenecks in current system software approaches to communication and synchronization. A new software stack can reduce these overheads by orders of magnitude, especially with software-hardware co-design, thereby making it possible to achieve the parallel efficiency needed for Extreme Scale systems.

The energy efficiency challenge is critical also because all three classes of Extreme Scale systems will be expected to deliver their 1000× improvements in computation capability while essentially remaining within the power budgets of current systems. An aggressive hardware design for data-center-sized systems will need at least 60MW of power to achieve an Exa-op level of performance, under highly idealized zero-overhead assumptions for software [62]. When current software overheads are taken into account, it is clear that the Exascale capability cannot be achieved without a significant redesign of the system software stack.

As discussed in this report, current software approaches will be inadequate in enabling future Grand Challenge applications on Extreme Scale systems. Instead, the potential for a 1000× increase in computation capability offered by each class of Extreme Scale system will only be achievable through radical re-design of the underlying execution model and system software and hardware. Current execution models and system designs won't work at Extreme Scale because of their sequential foundations and their inherent energy inefficiencies. In addition, any attempt to use current execution models at Extreme Scale will result in prohibitively large costs in programmability. Recent trends in High Productivity Computing Systems (HPCS) have demonstrated reductions in the human effort required to develop high-productivity software for current Petascale systems, but they do not address the requirements of Extreme Scale architectures such as energy-constrained many-core parallelism and heterogeneous processors. Also, while there is some overlap between system software requirements for Extreme Scale and those for large scale commercial data centers, there are also significant differences. Commercial system software for Cloud Computing is primarily focused on optimizing throughput capacity of independent jobs, whereas system software for Extreme Scale must be capable of delivering a 1000× increase in parallelism to a single job.

This report recommends a technical approach for developing new software stacks for future Extreme Scale systems that includes new execution models and metrics (Chapter 3), with a focus on new implementations of grand challenge applications (Chapter 4) either developed from scratch or scaled up from existing Petascale applications. An Extreme Scale software stack must enable parallelism and locality to be expressed at the finest granularities possible so as to support *forward scalability* (Chapter 5), low overhead management of parallelism and locality (Chapter 6), and integration with future tools (Chapter 7). Finally, the new software stacks must tightly integrated with new Extreme Scale hardware via a rich set of hardware API interfaces in support of software-hardware co-design (Chapter 8). A 12-page abbreviated version of this report is available in [120].

Chapter 2

Exascale Hardware Characterization

The objectives of the prior exascale systems study [62]. were to understand the course of mainstream computing technology, and determine whether or not it would allow a 1,000X increase in the computational capabilities of computing systems by the 2015 time frame. As mentioned earlier, the focus was on the technology to address three classes of systems: data center, departmental, and embedded.

This chapter summarizes the hardware configurations from which the challenges were identified, and upon which much of the software discussion in this report is premised. As a summary, Table 2.1 lists the characteristics of the "aggressive strawman" which drove many of the prior reports conclusions. While the prior study focused on Flops as a primary measure of computation, most of the issues discussed in this chapter are also applicable to benchmarks that are not floating-point intensive.

2.1 Strawmen

To understand these challenges, we not only surveyed the technology space, but also did an extrapolation from three different baselines of potential approaches for achieving the data center scaled system. These "strawmen" included:

- An evolutionary approach, termed a "heavyweight strawman" that assumed machines that used commodity high performance microprocessors on relatively large, high heat dissipating, circuit boards, with separate routing and memory chips. A few dozen such cards would make up a computing "rack." Examples of such systems today include the Red Storm machine and its commercial XT derivatives from Cray, Inc.

- A second evolutionary approach, termed a "lightweight strawman" that assumed customized, lower power, microprocessors that permitted integration of a complete memory plus processing node on a small circuit card that could be stacked by the thousands into bigger systems. The IBM Blue Gene series is typical of this class today.

- A "clean sheet of paper" approach, termed the "aggressive strawman," that still assumed silicon for its base technology, but one where the transistor parameters could be adjusted for maximum delivered performance per unit of energy consumed.

Table 2.2 lists some of the salient characteristics of both the two strawmen based on today's machine architectures, and the aggressive strawman system as it might exist in 2015. In this table,

	Exascale System Class				
Characteristic	Exaflops Data Center	20 MW Data Center	Department	Embedded A	Embedded B
Top-Level Attributes					
Peak Flops (PF)	9.97E+02	303	1.71E+00	4.45E-03	1.08E-03
Cache Storage (GB)	3.72E+04	11,297	6.38E+01	1.66E-01	4.03E-02
DRAM Storage (PB)	3.58E+00	1	6.14E-03	1.60E-05	1.60E-05
Disk Storage (PB)	3.58E+03	1,087	6.14E+00	0	0
Total Power (KW)	6.77E+04	20,079	116.06	0.290	0.153
Normalized Attributes					
GFlops/watt	14.73	14.73	14.73	15.37	7.07
Bytes/Flop	3.59E-03	3.59E-03	3.59E-03	3.59E-03	1.48E-02
Disk Bytes/DRAM Bytes	1.00E+03	1.00E+03	1.00E+03	0	0
Total Concurrency (Ops/ Cycle)	6.64E+08	2.02E+08	1.14E+06	2968	720
Off-chip Memory Bandwidth (B/sec per flops)	0.0025	0.0025	0.0025	0.0025	0.01
Off-chip Network Bandwidth (B/sec per flops)	0.0008	0.0008	0.0008	0.0008	0.0032
Component Count					
Cores	1.66E+08	50,432,256	2.85E+05	742	180
Microprocessor Chips	223,872	67,968	384	1	1
Router Chips	223,872	67,968	384	0	0
DRAM Chips	3,581,952	1,087,488	6,144	16	16
Total Chips	4,029,696	1,223,424	6,912	17	17
Total Disk Drives	298,496	90,624	512	0	0
Total Nodes	223,872	67,968	384	1	1
Total Groups	18,656	5,664	32	0	0
Total racks	583	177	1	0	0
Connections					
Chip Signal Contacts	8.45E+08	2.57E+08	1.45E+06	2,752	2,752
Board connections	1.86E+08	5.65E+07	3.19E+05	0	0
Inter-rack Channels	2.35E+06	7.14E+05	8,064	0	0

Table 2.1: Exascale class system characteristics derived from aggressive design.

	Heavyweight Strawman based on XT4 (today)	Lightweight Strawman based on BG/P (today)	Aggressive Strawman (in 2015 technology)
Cores per socket	2	4	742
Sockets per node	4	1	12
Nodes per rack	24	1024	32
Racks per system	120	16	583
Cores per system	23,040	65,536	166,113,024
Sockets per system	11,520	16,384	223,872
Nodes per system	2,880	16,384	18,656
Flops per cycle per core	2	4	4
Clock (GHz)	2.6	0.85	1.55
GFlops per second per core	5.2	3.4	6.2
GFlops per second per socket	10.4	13.6	4,600
GFlops per second per node	41.6	13.6	55,205
GFlops per second per rack	998.4	13,926.4	1,766,554
GFlops per second per system	119,808.0	222,822.4	1,029,900,749
GBytes per core	2	0.5	0.0216
GBytes per socket	4	2	16
GBytes per node	16	2	192
GBytes per rack	384	2,048	6,144
Gbytes per system	46,080	32,768	3,581,952
Bytes per flop per second	0.3846	0.1471	0.0035

Table 2.2: Characteristics of hardware baselines.

a "core" is a processor capable of independent execution of a program thread, a "socket" is a single microprocessor chip, a "node" is a combination of processing, memory, and routing capable of complete program execution, and a "rack" is a refrigerator-sized enclosure that contains some number of nodes.

2.2 The Aggressive Strawman Architecture

The aggressive strawman was an attempt to see how far we could push if we started with a clean sheet of paper, especially to maximize gigaflops per watt. Figure 2.1 summarizes the resulting architecture of such a system. We assume a 2013 technology node of 32 nm silicon as a baseline for the projection, but with an alternative set of process parameters that permits much more aggressive voltage scaling than would be possible with "logic as usual" in the same time frame.

Figure 2.1 starts with a Floating Point Unit (**FPU**) along with its register files and amortized instruction memory. Four FPUs along with instruction fetch and decode logic and an L1 data memory forms a **core**. We combine 742 such cores on a 4.5Tflops, 150W active power (215W total) **processor chip**. This chip along with 16 1GB DRAMs forms a **Node** with 16GB of memory capacity. Together this combination by itself corresponds to an aggressive embedded exascale system. Using alternative silicon process parameters allows us to project about a factor of three in energy savings per flop over the conventional process.

The final three groupings correspond to the three levels of bigger system interconnection. 12

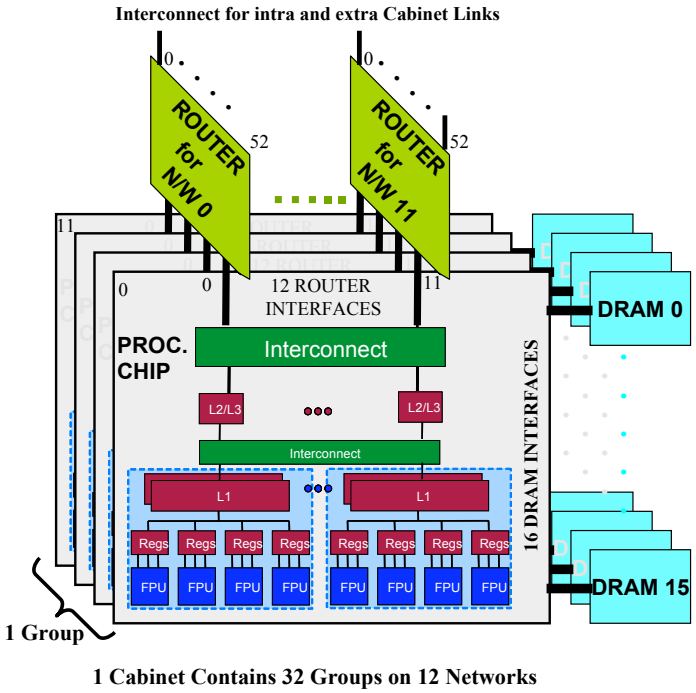

Figure 2.1: Aggressive strawman architecture.

nodes plus 12 separate routing chips form a **group**, 32 groups are packaged in a **rack**, and 583 **racks** are required to achieve a peak of 1 exaflops. We also assume 16 disk drives per group for secondary storage (192 TB in 2015), or an equivalent 512 drives (6.1 PB) per rack.

One rack by itself corresponds to over a petaflop, and thus corresponds to a departmental exascale system.

2.2.1 Adaptively Balanced Node

One of the concerns with the original strawman baseline of column C of Table 2.2 was that the bandwidth from the 742-core chip to the local memory was on the order of 0.0025 bytes per flop, and 0.00076 bytes per flop to off node memory. This is orders of magnitude less that what is typically considered desirable, but all that could be supported if one wanted in some sense to spend about equal power across processing, memory access, and interconnect.

To possibly overcome this on an application-by-application basis, the prior report [62] also proposed in Section 7.3.7 an *adaptive node design i.e.,* a design that would support the maximum dissipative power to be spent in **either** processing, memory, or interconnect by themselves, and then allow some power-adaptive mechanisms to select what mix of all three does best for a particular application or application phase, and without exceeding the power dissipation limits of the chips.

The resulting design point would hold 1060 cores which when all at full speed would consume the total chip power, and provide 6.4 Tflops per chip, with no memory or network bandwidth. Alternatively, this adaptive design also widened the memory interfaces to provide an order of magnitude more bandwidth, if all one wanted to do was access memory. For applications which need some of both processing and memory bandwidth, intermediate points could be chosen that are balanced from a performance perspective, and utilize the available power capabilities to best efficiency.

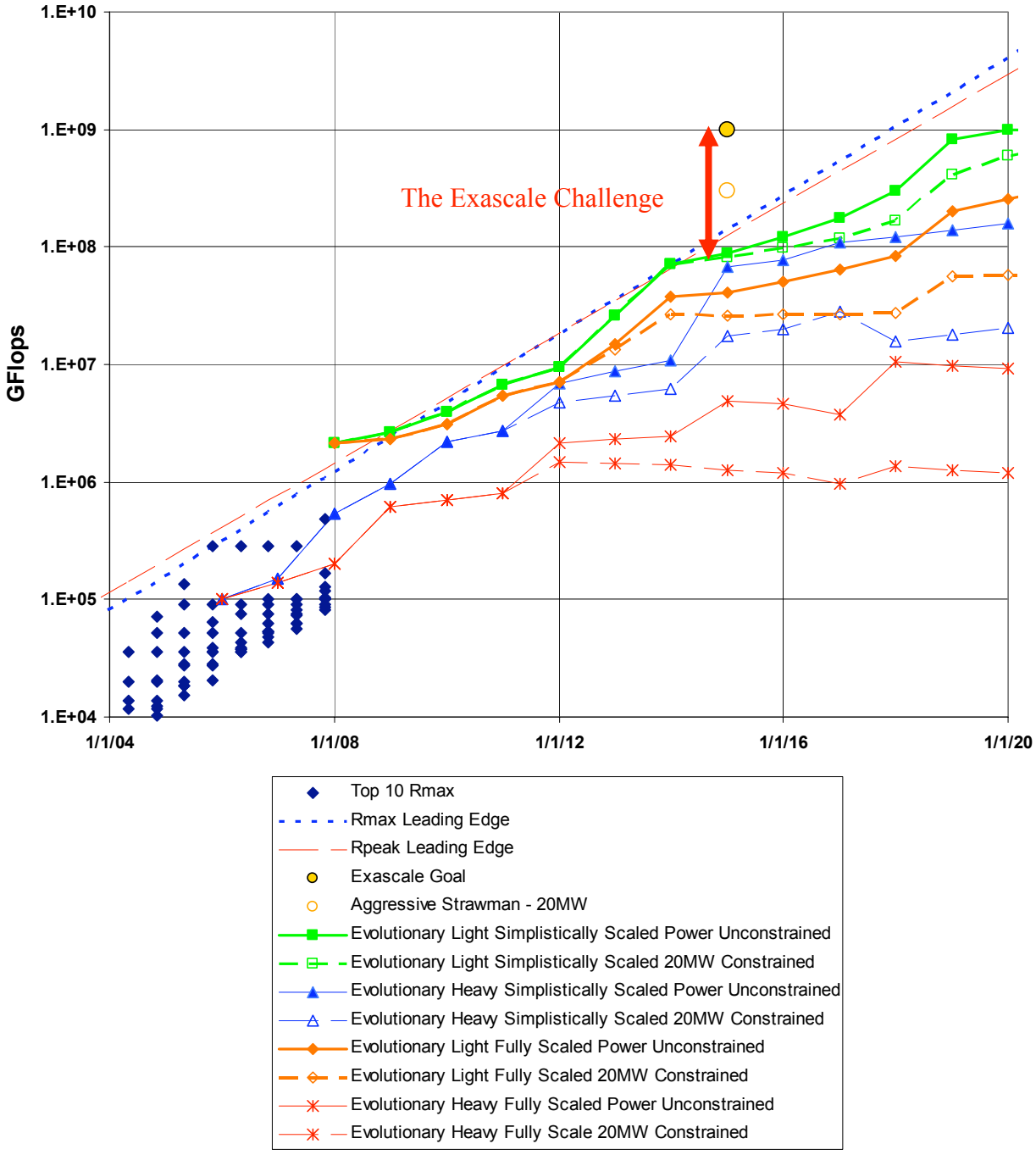

Figure 2.2: Exascale goals — Linpack.

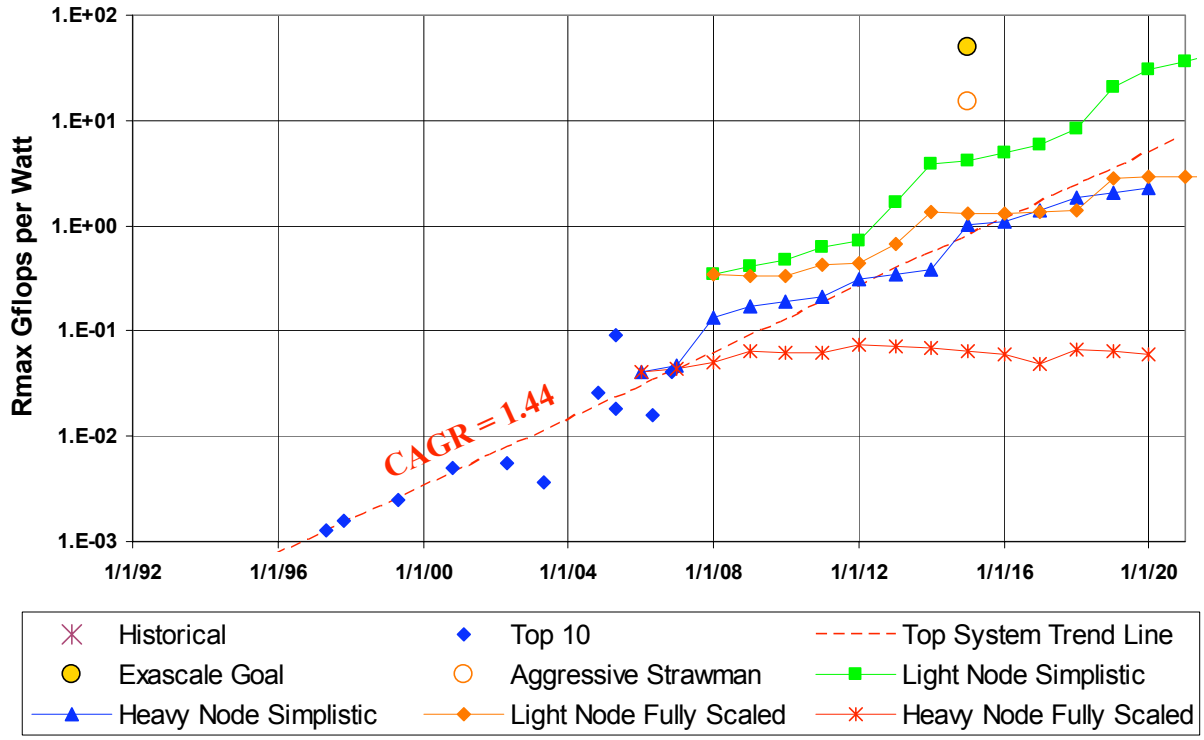

Figure 2.3: The power challenge for an Exaflops Linpack.

2.3 The Summary Extrapolations

2.3.1 Power

Figure 2.2 is a copy of Figure 8.1 from the original report, and summarizes the results of extrapolating the performance of both of the two "today" baselines from Table 2.2 through the expected evolution of technology, assuming nothing out of the ordinary happens *i.e.*, no "clean sheet" opportunities.

There are four lines associated with each of the conventional strawmen: one that assumed no maximum power constraint other than some maximum number of racks at some maximum feasible power per rack, one where the maximum power is limited to 20 MW (the original target power), one labeled "simplistically scaled" where the energy per bit moved or accessed from memory scales down with technology, and one labeled "fully scaled" where such energies do not change. The third is highly optimistic; the fourth pessimistic. We expect reality to lie somewhere in between these two curves.

There is also a point in 2015 at 1 exaflops representing the study's goal, and a point below it representing the aggressive strawman if power was limited to 20 MW.

The results indicate the severity of the performance and power problem. None of the design points studied reached the exaflops level of performance for 20 MW. Further, only two design points reach the desired exaflops performance — the aggressive design in 2015 at a cost of 67 MW, and an optimistic (unrealistic) extrapolation of the lightweight node strawman around 2020. Figure 2.3 (a copy of Figure 8.3 from the report) presents the same extrapolations with "Gflops per watt" as the y-axis. In the 2015 timeframe everything other than the aggressive design is off by one to three orders of magnitude, and the aggressive design itself is still off by a factor of 3.

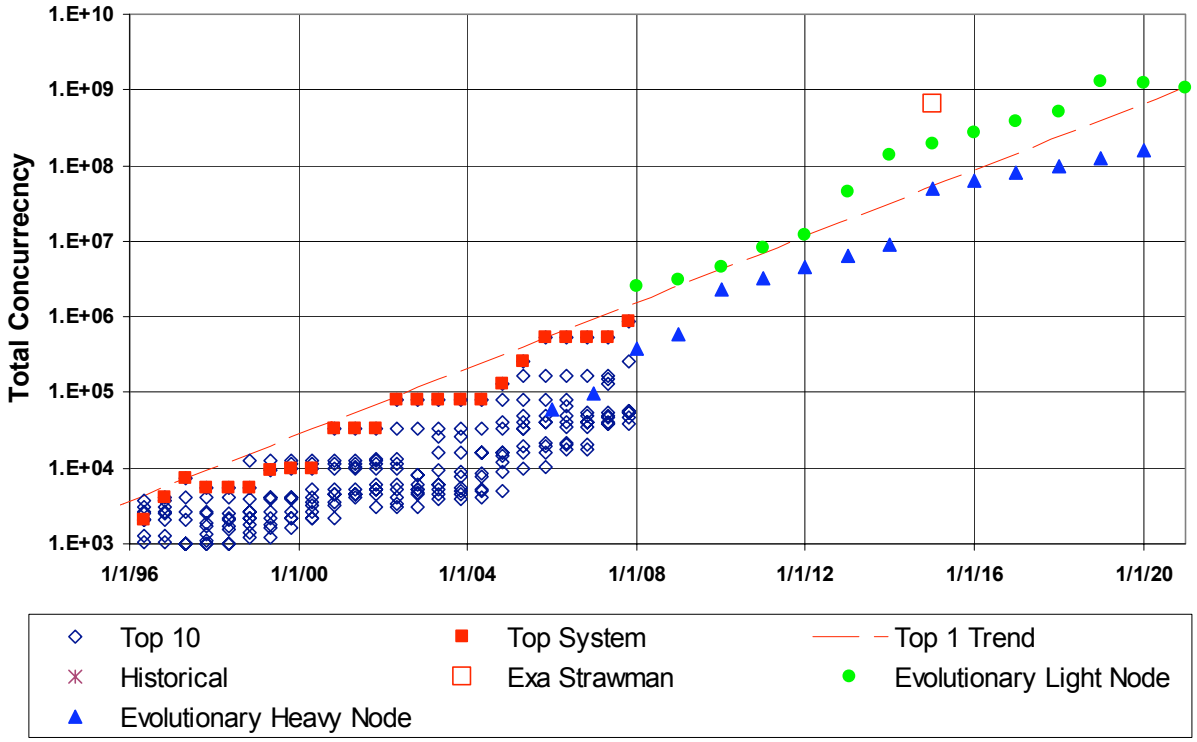

Figure 2.4: The overall concurrency challenge.

2.3.2 Concurrency

Figure 2.4 documents another relevant part of the study — an estimate of the concurrency seen in the various strawmen. Here, concurrency is defined as the number of operations (flops in this case) that must be started in each and every cycle by a program over the entire duration of an application in order to achieve exaflops performance. Again, the numbers are enormous, reaching a billion for the only two points that reach an exaflops. This is at least three orders of magnitude larger than the degree of concurrency exhibited by any machine today.

2.3.3 Memory and Bandwidth

A rule of thumb is that a supercomputer needs about a byte of memory capacity and a byte per second of memory bandwidth for each flop executed per second. Today's top machines are in the 0.1 to 0.3 range for both. As is listed in Table 2.2, the memory capacity of the aggressive design is considerably smaller, as is the bandwidth to memory as discussed in Section 2.2.1.

There are no obvious ways of fixing either gap with current technology and within any sort of acceptable power budget. Later in the report, we discuss techniques (such as "new-era" weak scaling in Chapter 4) that can tolerate this new imbalance between memory and computation.

Chapter 3

Extreme Scale Software Execution Models and Metrics

3.1 Execution Models

Before discussing the potential challenges of extreme scale computing, and potential solutions to those challenges, it is appropriate to define the environment in which programs that perform such computation will find themselves. This chapter introduces the concept of an "execution model" as the basis on which extreme computation must be specified, along with several examples. It then defines several general metrics by which extreme scale systems are likely to be measured and evaluated. Later chapters will elaborate on these metrics.

3.1.1 Models of Computation

A model of computation is a paradigm for organizing and carrying out computation across all levels of the computer system stack from programming models and languages through compilers and runtime systems to operating systems and system and micro architectures. It provides the conceptual scaffolding for deriving each of these system elements in the context of and consistent with all of the others. This paradigm suggests a decision chain to which each layer contributes that ultimately determines when, where, and how every operation of a computation is performed. An execution model is not a programming language although it may strongly influence the underlying programming model semantics of which the language is a representation. It is not a computer architecture although it establishes the needs for low level mechanisms the architectures must support and provides the governing principles that guide the structures and actions of a computer architecture in the performance of a computation. And, it is not a virtual machine isolating the abstractions above it from the implementation details below because it cross-cuts all layers from programming language to architecture influencing all aspects of the operation of all system layers in concert.

Models of computation serve as the continuum medium of existence and evolution for the subdomain of computing systems, perhaps most predominant in high-performance computing. The evolution of high-performance computing over the previous six decades has been marked by dramatic phase changes delineating sometimes overlapping major epochs in how we reason about high performance computations. Driven by advances in underlying enabling technologies and the opportunities and challenges they present to the design and implementation of HPC systems, execution models have been changed to enable their most effective application by exploiting those opportuni-

Execution Model	Device trend	Architecture trend	System Software trend
Von Neumann	SSI devices	Scalar instrs.	Scalar compilers
Vector Parallelism	MSI devices	Vector instrs.	Vectorizing Compilers
Shared-Mem. Parallelism	VLSI microprocessors	Cache coherence	Multithreaded OS and runtim
CSP with Bulk-Sync. Parallelism	VLSI microprocessors	Interconnects	Message-passing libraries (MP

Table 3.1: Examples of Past Successful Execution Models

ties while simultaneously addressing their challenges. During each epoch, incremental technology advances have been incorporated with incremental modifications to architecture, operating systems, and compilers largely keeping the programming model essentially invariant along with the foundation model of computation. However, past a critical threshold, continued optimizations become untenable and a metamorphosis catalyzes the emergence and adoption of a new model of computation inaugurating new opportunities, design points, and performance as well as other operational properties.

Examples of models of computation over the history of HPC include the Aiken Harvard model, the von Neumann model, the vector model, the SIMD array model, the dataflow model, the systolic model, the communicating sequential processes model (CSP), and the multithreaded shared memory model. Not all execution models (*e.g.,* dataflow) proved commercially successful at the programmer-visible level (although models like dataflow did migrate into the CPU's microarchitecture in the form of "out of order" execution). Others, such as SIMD, have impacted in multiple forms again and again. Successful execution models have primitives that are well matched with device, architecture, and software technology trends, as illustrated in Table 3.1.

The most recent MPI epoch is remarkable in its longevity and continuity over more than two orders of magnitude in multiple dimensions of enabling technologies and metrics (flops, bytes capacity, bps data transfer). This is perhaps most readily apparent by the concerns about continued support of legacy MPI code base; a concern never previously realized for any other current parallel programming methodology. But that very duration has stretched the gap to the breaking point. The current model of computation, CSP with Bulk-Synchronous Parallelism, is no longer capable of supporting the most effective exploitation of current and future generations of implementation technologies, addressing the many new challenges they impose (such as massive concurrency with asynchrony), or facilitating the scale of computation to the ExaFlops performance regime that is required within a decade's time. Past methods of extending delivered performance are no longer tenable. Power constraints, now perhaps the most dominant challenge, are inhibiting continued growth of clock rates, once a major source of performance improvement. Multicore components have almost universally replaced single processor devices imposing an entirely new level of user-exposed parallelism while aggravating chip I/O, cache behavior, and memory bandwidth problems as well as programming methodologies. Additional departures from conventionality include new instruction set architectures, new hardware structures, and accelerators such as GPGPUs, game machines (*e.g.,* IBM Cell SPE), and attached array processors (*e.g.,* ClearSpeed). All of these breaches of CSP conventionality are symptoms of an impending phase transition in HPC. And as has always happened before in such cases, it is the change in the foundation model of computation that will fully establish the new direction for the next epoch.

Then what exactly is a model of computation if it is not the manifestations of architecture, system software, and programming models that visibly reflect it? One perspective is that an execution model is a set of governing principles that guide the form and function of computation. This includes the nature of the state that initiates, evolves, and is finalized throughout the compu-

tation to its conclusion, the atomic operations that may be performed on elements or compound structures of such elements, the semantics of flow control and concurrency of operation, the means of coordination, cooperation, and synchronization, the name spaces and their interrelationships of the data and possible the control elements as well, the innate reflection for control of hierarchy, encapsulation, modularity, means of manifesting work-flow, distribution, and asynchrony of tasks. This laundry list of pieces in ensemble constitutes a paradigm and the different models of computation employed over the decade may be distinguished by these various attributes. More generally, a model of computation answers the broad question of how we structure and name data and instructions and how we interrelate the two. Within the realm of parallel models of computation, this generality extends to how we harness, distribute, and control concurrency of action.

A subtle question that drives an HPC phase change by transitioning between models of computation is how to assess when a new model constitutes a superior paradigm with respect to a previous one being replaced? A partial answer comes from consideration of the set of underlying factors that determine efficiency of parallel execution. These are starvation, overhead, latency, and contention. Starvation is the factor requiring sufficient algorithm concurrency to drive all parallel physical elements ($>$ 100 million cores, $>$ 10 billion-way program parallelism) and resource allocation (dynamic load balancing) to ensure that all cores in a multicore system have useful work to do. Overhead is the factor requiring mechanisms providing critical path services to exhibit minimum response time essential for efficient exploitation of parallelism for ultra scalability. Lightweight synchronization, distributed global address translation, process instantiation and termination, thread scheduling and context switching, load balancing, and communication exemplify overhead mechanisms of importance. Latency is the efficiency factor requiring delays to critical resources due to latency of information transfer and service request to be mitigated through minimization and hiding (overlapping) such as to memory or remote computing/data resources. Contention is the factor requiring that delays to critical resources due to blocking from simultaneous service requests of shared resources be minimized and mitigated with, for example, sufficient system area network bandwidth, memory bandwidth and chip I/O, and ALU throughput with respect to demand. Within the context of these factors the effectiveness of an execution model may be assessed. But the model most also leverage the technologies it is intended to exploit. Thus it is this interplay of semantics and physics that must be coordinated.

The CSP model has been so effective because it provides this close match between the strengths of the technologies and the semantics of the model. Specifically the CSP process maps cleanly to the physical processor. The parallelism to be exposed has to match the number of processors in a system. At scales of a few hundred to a few thousand this worked well, especially for weak scaling where the data size grew proportionally with the scale of the system. The static mapping eliminated much of the overhead. Latency was mitigated by maintaining much more intra-process execution to message passing (but it also requires that there be enough memory per processor). The vector model worked because it allowed faster technologies through pipelining. The SIMD array model was best when technology density permitted many more modest processor and memory nodes to work together but when clock rates were slow enough that the instruction broadcast time was not prohibitive. Dataflow didn't work well as conceived in the 1980s in part because it was memory and communication intensive with synchronization overheads exceeding the work of the fine grain operations.

3.1.2 Desiderata for an Extreme Scale Execution Model

What then are the requirements of a model of computation that will match the future capabilities and opportunities of extreme scale systems while addressing their limitations? Considering the

critical factors mentioned earlier, several attributes can be established. Concurrency must exceed a billion-way parallelism. This suggests that the overhead for managing such parallelism has to be lower both to expose finer grain parallelism (overhead lower than the useful work per independent action). A new model of computation has to change the synchronization semantics, removing global barriers, and exposing new levels of algorithmic parallelism. It has to provide dynamic adaptive resource and task scheduling to respond to unpredictable ordering of execution such as varying latencies, contention, and priorities. Such a model has to maintain some level of global name space and manage the address translation with low overhead. The model has to permit a wide variation in implementations from architecture up to programming languages.

Most importantly, a model of computation for future parallel system implementation in the pan-exaflops performance domain must enable rational and quantifiable co-design of the separate but interrelated layers of the system stack to guide the development of the semantics and structure of each.

Extreme Scale systems are characterized by new device technologies with significant power constraints and by computer architectures that will build on manycore processors with $O(10^3)$ cores per socket. A successful Extreme Scale execution model must make explicit the key performance factors in future Extreme Scale systems which include concurrency, locality, energy, and overhead. To that end, we outline the following desiderata for an Extreme Scale execution model:

1. Asynchronous lightweight tasks and communications
    - Motivation: need asynchrony to hide latency and handle variability among cores
    - Challenge: scheduling with bounded resources (adapt across eager vs. lazy scheduling for starvation vs. contention modes)
    - Related topics: control vs data-driven initiation/termination of tasks
2. Explicit locality model
    - Motivation: locality is key to efficient parallelism
    - Challenge: portable and hierarchical abstractions of locality
    - Related topics: co-locating distributed tasks and distributed data c.f., Sequoia and X10 execution models
3. Scalable Coordination and Synchronization
    - Mutual exclusion (transactions)
    - Producer-consumer (streams)
    - Collective synchronization (barriers, phasers)
    - Other collective operations (reductions)
    - Point-to-point synchronization (semaphores)
4. Abstract performance model
    - Parallelism
    - Locality
    - Energy

3.2 Metrics

In this section, we summarize metrics that can be used to evaluate the effectiveness of new system software technologies for Extreme Scale systems. Though there has been significant amount of past work on metrics for application characteristics (Chapter 4) and for hardware characteristics [62], less attention has been paid to metrics for the system software that bridges the two. This is unfortunate because it is widely recognized that limitations in system software can cause algorithms that are highly scalable in theory to fall short by one or more orders of magnitude in realizing their full potential on parallel hardware. Examples of system software characteristics that contribute to this attenuation include OS scalability limitations, lack of locality/affinity management, lack of adaptation and self-awareness, compiler optimization limitations, thread/task creation overhead, synchronization overhead, and other overheads in runtimes for scheduling, memory management, communication, performance monitoring, power management, and resilience.

As we study metrics for system software, we observe that system software for Extreme Scale systems must strive to balance conflicting goals in its management of *concurrency*, *locality*, and *energy*. The conflicts among some of these goals are well understood from past research. For example, there is a natural tension between concurrency and locality when spreading computations across multiple cores can improve concurrency but degrade locality. Likewise, frequency and voltage scaling for energy optimization may degrade concurrency and time-to-completion if the frequency scaling resulted in the slowdown of a task on the critical path. What we desire is a single combined metric or figure of merit that can capture all these trade-offs. To that end, our proposed single combined metric is to build on the idea of *energy-delay product* [138] as follows.

The cost of an execution of application A on an Extreme Scale platform with system software S and hardware H can be expressed as

$$C(A, S, H) = \text{TotalEnergy}(A, S, H) \times \text{TotalElapsedTime}(A, S, H).$$

This C-A-S-H metric can be used as the starting point for a number of detailed evaluations. For example, to evaluate the cost improvement delivered by a new system software stack, S_{new}, relative to an existing system software stack, S_{old}, we can compute the ratio, $C(A, S_{old}, H)/C(A, S_{new}, H)$, for some number of (A, H) pairs. The choice of applications, A, can be driven by a small number (say 3 to 5) of grand challenge applications that are representative of workloads of importance to mission partners, and the choice of hardware platforms, H, can be simulated by analytical models such as the strawmen Exascale systems introduced in [62] or by extrapolating from current Petascale systems.

The *TotalElapsedTime*(A, S, H) and *TotalEnergy*(A, S, H) sub-metrics can in turn be used for more detailed point-to-point comparisons. For example, the *TotalElapsedTime* metric can be used to compare the scalability of two different system software stacks, as discussed in Appendix B. Also, since vertical locality (or the lack thereof) has been identified as a significant contributor to energy costs, the *TotalEnergy* can be used to compare the locality management capabilities of two different system software stacks. Microbenchmarks akin to HPCC can also be developed to evaluate performance per unit energy (*e.g.,* operations/Joule and bytes-transferred/Joule), which is correlated with the reciprocal of the C-A-S-H metric. As yet another example, real-time applications that work with a fixed value for *TotalElapsedTime* (deadline) can use the C-A-S-H metric to compare the energy efficiency of two software stacks that satisfy the same deadline.

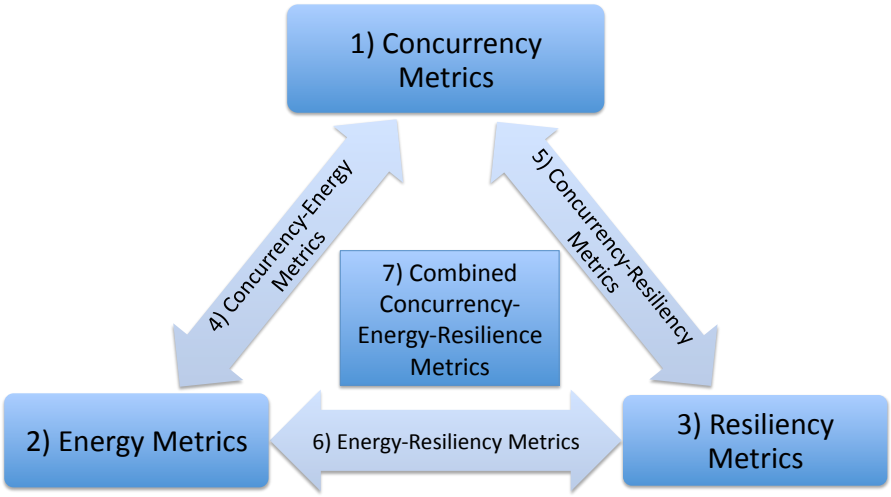

Figure 3.1: Structure of Metrics for Concurrency, Energy, and Resiliency

Chapter 4

Challenges in Developing Applications for Extreme Scale Systems

This chapter addresses the challenges facing the development of applications for Extreme Scale computing systems. These challenges lie both in understanding how much system resources are needed to support applications as their requirements change in size and complexity, and in the converse, namely how efficient applications may be as a function of how many resources are made available to them. In particular, Section 4.1 summarizes the discussions to date within the community about applications requirements and system size. Section 4.2 then discusses in more detail what "application scaling" means. The succeeding sections then consider different classes of applications. Section 4.9 concludes by summarizing application "sweet spots" in terms of resources for extreme scale as we currently understand them.

4.1 Application Overview

Application scaling addresses how applications may port to new systems in two respects: how data sets and problem sizes may grow and fit in new machines with more resources, and/or how existing applications may adapt when "bigger" systems become available for their execution. Discussions on how and why such scaling may occur have been topics of considerable debate within the community over the last few years. In particular, during 2007, there were numerous meetings held to investigate future computing applications and hardware/software requirements as they might exist at the exascale level. These meetings included:

- Three DOE Exascale Townhall Meetings [2]
  - Lawrence Berkeley National Laboratory (LBL) (April 2007) [3]
  - Oak Ridge National Laboratory (ORNL) (May 2007) [4]
  - Argonne National Laboratory (ANL) (May/June 2007) [5]
- Council on Competitiveness meeting on Exascale Applications [87]
- Frontiers of Extreme Computing 2007/Zettaflops workshop (October 2007) [6]

At the DOE-sponsored meetings, there were extensive discussions of those applications that could and should scale to exascale. At the Council on Competitiveness meeting, commercial HPC users discussed applications that they believed could benefit from extreme scaling.

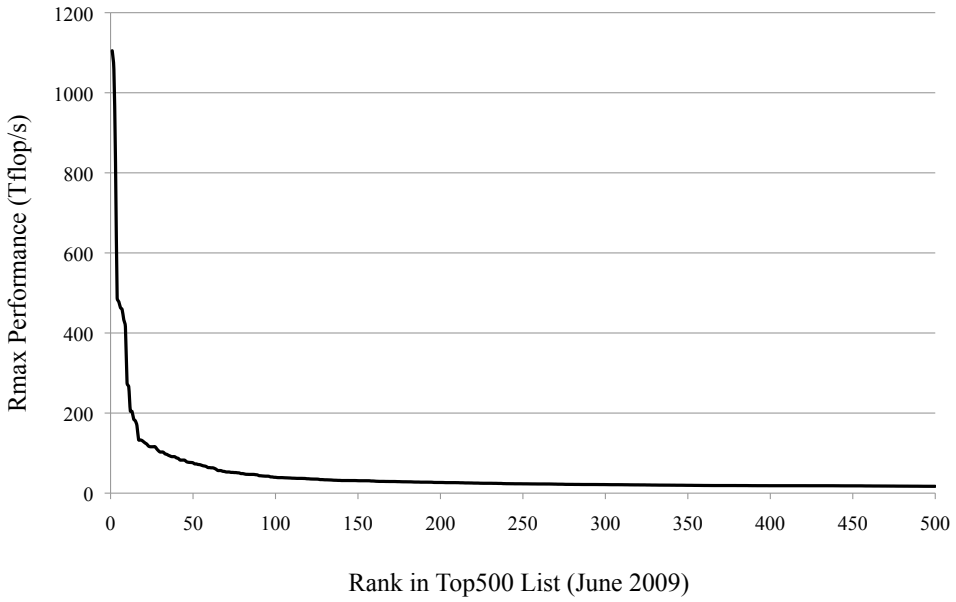

Figure 4.1: Rank-ordered distribution of Rmax in Tflop/s for systems in the June 2009 Top500 list [1]. 2 of 500 systems have Rmax > 1000 Tflop/s (1 Pflop/s). 468 of 500 systems have Rmax < 100 TFlop/s. 421 of 500 systems have Rmax < 50 TFlop/s.

Not all existing applications will scale to terascale, petascale, or on to exascale given current application/architecture characteristics. The reasons for limited scaling are potentially many, including (but not limited to):

- Parallelism
  - Terascale — $O(10^5)$ threads
  - Petascale — $O(10^8)$ threads
  - Exascale — $O(10^{11})$ threads
- Locality
  - Vertical — temporal locality (*i.e.,* reuse). One exploits vertical locality by moving data up and down a node's memory hierarchy.
  - Horizontal — occurs as a result of node-level data decomposition. One exploits horizontal locality through domain decomposition — by putting a portion of the application data on each node of the machine.
- Bottlenecks
  - Memory bandwidth
  - Bisection bandwidth
  - I/O bandwidth

Of those applications that operate at sustained terascale today, only a small fraction of those applications will be successful at reaching petascale. Hopefully, lessons learned when moving applications to petascale, will permit a reasonable fraction of petascale applications to scale out to exascale levels.

Insight into the current state of application scaling can be found by looking at the June 2009 Top500 list [1]. Figure 4.1 shows the rank-ordered distribution of the Rmax peak performance in Tflop/s for the systems in this list. While the Los Alamos National Laboratory's Roadrunner and Oak Ridge National Laboratory's Jaguar systems have broken the petascale performance barrier on high-performance LINPACK, more than 93% of the top 500 systems have Rmax below 100 TFlop/s. Clearly there are not substantial numbers of HPC applications currently running at near-petascale levels, which underscores the challenge for application enablement at the exascale level. As we examine future exascale application footprints, we will also need to analyze existing applications and develop models for application scaling to develop projections for applications running at petascale and exascale. Eventually, these models will be validated by petascale application development work being performed throughout DOE to provide applications for the LANL RoadRunner [7], and work being performed by the University of Illinois Urbana-Champaign (UIUC) to study scaling as they prepare to receive the National Science Foundation (NSF) Tier 1 Blue Waters sustained Pflop/s system [8].

4.2 Application Scaling

Application scaling remains a major challenge in the utilization of high-end parallelism. Of applications that operate at sustained Terascale performance today, only a small fraction is expected to be successful at reaching Petascale and an even smaller fraction at Exascale. Further, even for applications that reach Petascale performance today, the nature of scaling necessary to obtain Petascale performance on an Extreme Scale departmental system will raise new challenges for rewriting the application to address the concurrency and locality requirements of such systems. In this paper, we argue that the existing software "stack" is a major contributor to these scalability limitations, and that the approaches discussed in the following sections could have a significant impact in removing obstacles to scaling.

There are three primary ways to scale applications:

- Strong scaling — apply more resources to the same problem size to get faster results.
- Weak scaling — apply more resources to larger problem sizes to do more within a tractable time period.
- Temporal scaling — run an application longer.

We will examine the impacts of strong and weak scaling below, and defer more formal definitions to Appendix B. Temporal scaling does require additional resources, but those are spread over time. Temporal scaling does not provide additional work within a timestep. Thus there is no increase in the flop/s rate.

4.2.1 Strong Scaling

Strong scaling refers to the concept of applying more resources to the same problem size to get results faster. Unfortunately, few applications are amenable to strong scaling, so we cannot rely on strong scaling to move applications from petascale to exascale. As an application is strongly

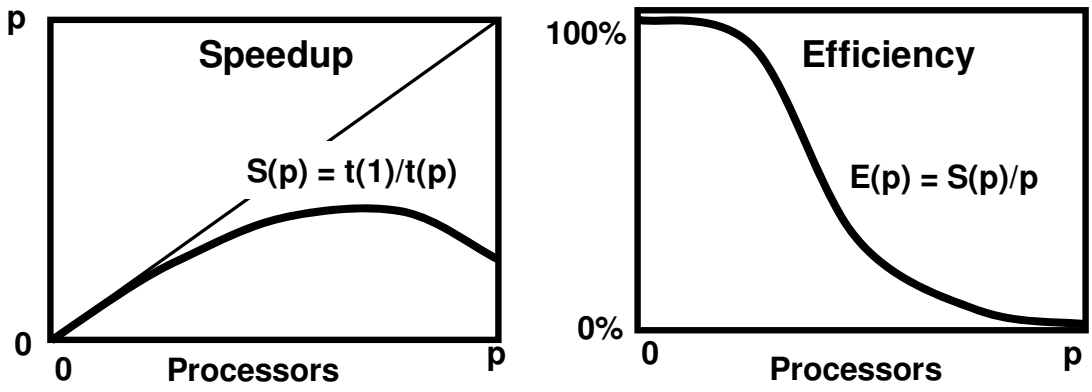

Figure 4.2: Conceptual speedup and efficiency curves for strong scaling

scaled, the work at a node/processor/core decreases and the relative overhead increases. Speedup may initially equal the number of processors, but eventually the amount of overhead causes the slope of the speedup curve to flatten. At this point, adding processors does not cause the application to run faster. Eventually, it is possible that overhead grows so rapidly that adding processors actually causes the time to solution to flatten or increase and speedup to flatten or decrease. An application that demonstrated reasonable scaling over three orders of magnitude increase in the number of processors is the first principles molecular dynamics "Qbox" code that won the 2006 ACM Gordon Bell Prize for "peak performance" with over 200 Tflop/s sustained performance (56% efficiency) on the LLNL BlueGene/L [9]. More recent winners of the 2008 ACM Gordon Bell Prize further underscored the importance of algorithmic innovations that attain very high levels of spatial and temporal locality.

4.2.2 Weak Scaling

Weak scaling refers to the concept of adding work as an application is run on more processors. By adding work, it is possible to ensure that overhead does not destroy performance. Traditionally, weak scaling has been accomplished by adding work due to spatial scaling. However, there are additional sources of scaling — discussed below and referred to in this report as "new-era" weak scaling — arising from new application trends in which additional work is done per datum *e.g.,* multi-scale, multi-physics, interaction analysis, and data mining. Weak scaling permits the user to look at larger or more complicated problems and use the additional processors to solve larger problems, obtain better resolution, or learn more about the phenomenon being examined.

Traditional weak scaling occurs in classical mechanics simulations, where either (1) larger problems are examined or (2) the grid size and time-step interval are reduced. Solving larger problems (*e.g.,* modeling the airflow around an entire airplane versus modeling the airflow over a section of the wing) results in a situation when memory scales nearly proportionally with work. In contrast, when the grid size is reduced (refined) in a 3-D mechanics simulation, the time step also needs to be reduced thereby increasing the amount of work relative to the amount of memory. When scaling in three dimensions, a 3/4 power rule applies to the amount of memory required whereas a 2/3 power rule applies when scaling in only two dimensions. Thus, for these applications, the required increase in memory size for a 1000× increase in work is 180× and 100× for 3-D and 2-D applications respectively.

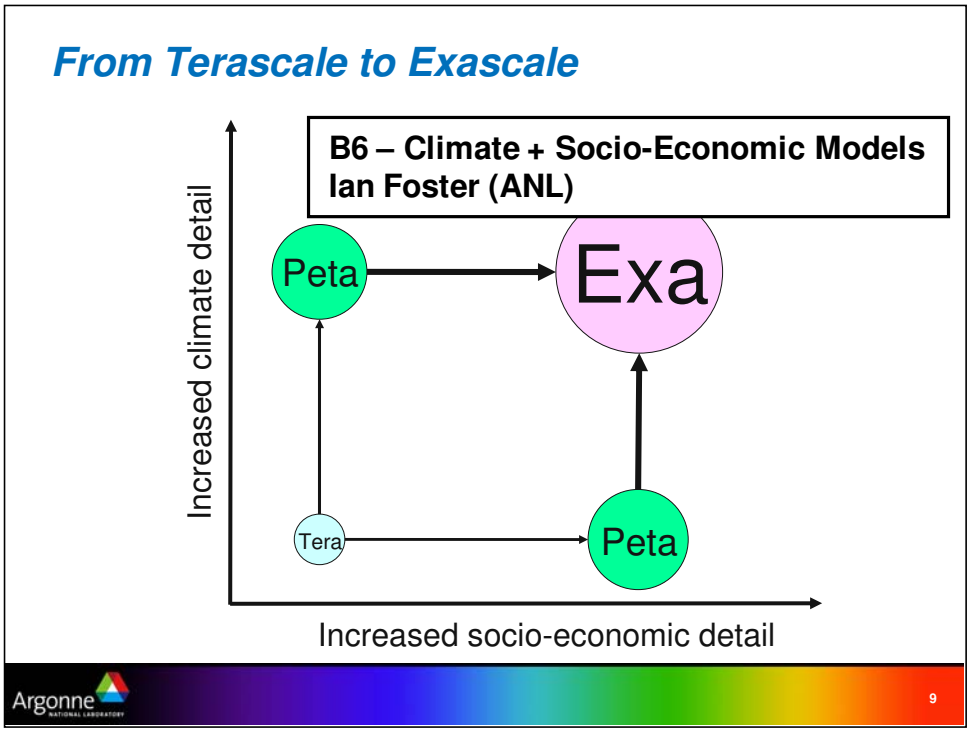

Figure 4.3: New-era weak scaling example

4.2.3 New Era Weak Scaling

"New-era" weak scaling typically adds extra work through one or more of the following:

- Multi-scale
- Multi-physics (multi-models)
- New models
- Interactions
- Mitigation analysis
- Data mining
- Data-derived models

This list is a product of aggregating materials presented at the DOE Exascale Townhall meetings. Figure 4.3 presents a slide indicating a conceptual scaling of the combination of climate and socio-economic models. In this figure, the climate and socio-economic models would be scaled out with increased detail then the analysis of the interactions would further boost the scale. Compared to traditional weak scaling, it can be more difficult to predict application footprints for new-era weak scaling.

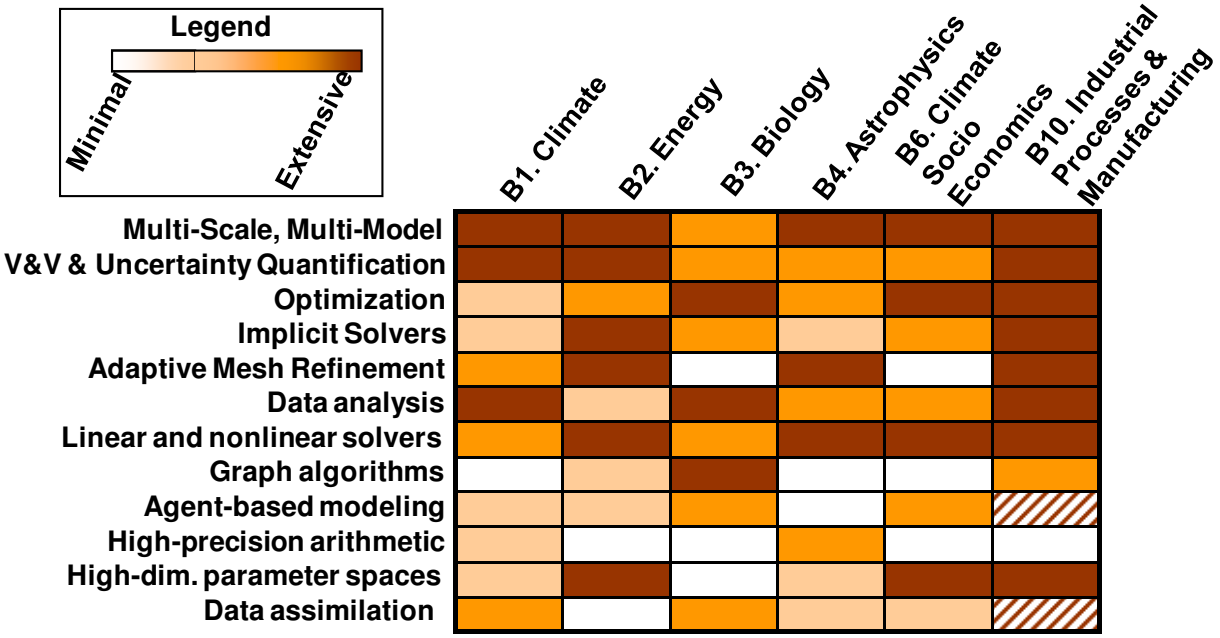

Figure 4.4: Application characteristics from Group B7 Mathematics and Algorithms, David Bailey, et.al.

4.2.4 Exascale Application Scaling

Nothing implies that we should expect a single silver bullet to enable applications to scale to exascale. We cannot expect extensive strong scaling in applications as the parallel computing "laws" of work versus overhead will still hold. "Ensemble" calculations cannot turn capability problems into capacity problems. Temporal scaling will not provide greater instantaneous amounts of "work". Anticipated new-era weak scaling may drive demanding footprints requiring more memory than the 180x implied by the $\frac{3}{4}$ power law. In addition to the need for extensive memory, there may be reduced data locality in the data-mining/data-intensive portions of the applications that examine domain interactions. These application characteristics may be far from the locality, reuse, and regular communications of high-performance LINPACK. On the other hand, anticipated new-era weak scaling may reduce memory demand when $O(n)$ scratch data is used with $O(n^m)$ data-mining work with m integrated modules. Some hardware characteristics may provide relief for other hardware deficiencies *e.g.,* it may be possible to trade memory footprint for disk bandwidth for application checkpointing. As stated above, exascale applications may benefit from current petascale application research to increase the amount of parallelism.

At the DOE Exascale Townhall meetings, David Bailey (LBL) presented a table of application characteristics for six application classes. This figure is represented in figure 4.4. These application classes were different "tracts" at the meetings. In this figure, the intensity of the block shading identifies the anticipated impact of the listed application characteristics. In two cases, this figure was modified to add information obtained from an Exascale Application discussion session run by the Council on Competitiveness. In those two cases the hatched blocks show that these application areas were modified from minimal impact to extensive impact.

4.3 Emerging Extreme Scale Applications

Candidate applications for extreme scale include:

- traditional HPC applications scaled up from petascale,
- traditional or new applications that are to run at petascale on departmental class systems,
- coupling together of multiple petascale applications to form an exascale application,
- emerging data mining applications, and
- real-time departmental applications.

Within these categories we wish to consider the following kinds of scaling:

- **Strong scaling:** Those that could solve the same problem as today but 1000× faster. For example, this category may include real-time CFD to stabilize a physically morphable airplane wing (effectively recalculating an AVUS calculation that now takes hours every few seconds).
- **Weak scaling:** Those that would solve the same problem as today but at a 1000× larger scale. This is normally called "weak scaling". In this category for example might be global weather models at sub 1km resolution (effectively solving a WRF calculation at the granularity used today to forecast hurricanes but at the scale of the entire earth's atmosphere).
- **Increased time steps:** Those that would solve the same problem as today but with 1000× more time steps. In this category for example might be local weather models at climatic timescales (effectively solving a WRF calculation at the size today used for medium-term weather forecasting but projecting out for centuries).
- **Increased resolution:** Those that would solve the same problem as today but at 1000× more resolution (or increased physics and chemistry in every cell). In this category might be for example global ocean and tide models that include micro-features such as wave refraction and coastal turbulent mixing. This may include coupling.
- **New approaches:** Those that solve entirely new problems than those solved today. These may include emerging applications in biology and social networks data mining.

Clearly Exascale applications could have aspects of any combination of the above; for example one could perform a calculation at 100x resolution and 10x more timesteps, for a total of 1000× more computation than is possible today. In the following sections we identify some specific exemplars of each and draw some general conclusions.

4.4 "Traditional" HPC Applications at Exascale

A good example of a scale-up of a traditional HPC application is next-generation hurricane modeling and prediction that requires ultra-high-resolution of gradients across the eyewall boundaries (at 1 km or less), and representation of the turbulent mixing process correctly (at 10 m or less). To do this over a local region, as for example a hurricane eye-wall, requires a petascale calculation. The requirements are for 100 kilometer square outer-most domain at 10 meter horizontal grid spacing and 150 vertical levels, 15-billion cell inner-most 10 meter nested domain, with a model time step

of 60 milliseconds. The calculation takes (at 100,000 tasks) 100 MB per task of data not counting buffers, executable size, OS overhead, etc. A petascale run generates $24 * 1.8$ terabyte datasets = 43.2 terabytes per simulation day assuming hourly output of 30 three-dimensional fields. Assuming an integration rate of 18 machine hours per simulated day at a sustained petaflop, the average sustained output bandwidth required is about 700 MB/second.

Now if one were to envision exascale extensions, scaling climate time frames is a "capacity" problem. A 1000× faster machine with no more main memory would allow one to model a very limited local region at the rate of 3 simulated years per day. However, an entire hemisphere of earth at "hurricane eyewall" resolution that might for example capture "butterfly effects" is a capability problem requiring a sustained exaflop and 10,000 Gbytes = 10 petabytes of main memory to model the earth at 10m resolution at the rate of 1 simulated day per day.

4.5 Coupled Models

In addition to traditional HPC applications, there is an opportunity to use an exascale facility to enable adding additional information to a model, for example adding ecological information, such as forest growth, to models of weather and climate. A grand challenge in geoscience is the addition of clouds to ecological modeling, especially since clouds and cloud-formation processes interact (in both directions) with the ecosystem.

Ecological models attached to a high-resolution model of climate change that already have petascale applicability may provide a good starting point. Ecologists have several embarrassingly parallel applications that can be "easily" scaled to a petascale system, including:

- stochastic processes[1] that need replication
- parameter sensitivity analysis
- heuristic optimization

Such problems are extremely common in ecological applications, as we elaborate here.

1. Stochasticity: Simulation-based ecological models often incorporate demographic stochasticity (random birth/death/movement, etc), environmental stochasticity (random components of climate forcing, resource availability, etc), and/or genetic stochasticity (random mating, mutation, etc). Outcomes are thus stochastic as well, and ecologists wish to ask questions like, "What is the simulated probability that the population size will fall below X within 100 years?" The simulation model must therefore be independently repeated (usually 100s-1000s of times) to generate a distribution of outcomes.

2. Parameter sensitivity (or more generally, model sensitivity): The "true" parameters of ecological models are rarely known, and in fact there are often disagreements about the form of the equations governing those processes. Consequently, ecologists frequently want to characterize the sensitivity of outcomes to input parameter values and model assumptions. This also requires repeated simulation.

3. Optimization: There are (at least) two distinct types of optimization questions that ecologists commonly ask. The first involves fitting parameters to observed data. In all but the most trivial models, it is impossible to use analytical or even simple approximating techniques to

[1] A stochastic process is one that has both predictable and random components in its formal description.

identify maximum likelihood estimates of parameters. Increasingly, ecologists are turning to stochastic optimization techniques such as simulated annealing, or the use of various implementations of Markov Chain Monte Carlo to simulate posterior probability distributions in a Bayesian framework. Secondly, applied ecological models often implement heuristic optimization algorithms as decision tools (*e.g.,* identifying the optimal spatial configuration of a land reserve system, given some cost criterion). As with parameter estimation, the simpler algorithms used in the past have been shown to be deficient in complex settings, but more reliable methods require many repeated simulations and long run-times. There is a tremendous need for HPC solutions that can deliver results sufficiently quickly even for models involving many parameters, fine-scale spatial and temporal resolution, and stochastic processes.

Putting this all together, it is clear that the compute time can be overwhelming when coupling one or more of the above procedures with even a moderately complex ecological simulation model. Specifically, some model examples include predicting evolution of a collection of interacting species, spatial spread of a disease, or the dynamics of a specific ecosystem. Taking the last example, imagine a regional-scale ecosystem model, the core of which is deployed as a small-scale HPC application (*e.g.,* a single simulation that takes days to complete on a cluster with dozens of nodes). Indeed, the ATLSS group[2] based at UT/ORNL has spent ∼10 years developing and refining a model that integrates a variety of complex and interacting sub-models to simulate key Geoscience and environmental components of the Florida Everglades; one sub-model has already been parallelized to run on 60+ nodes. Even if a researcher demands just several hundred stochastic replications in such a simulation, performed for each of 100 possible configurations of a proposed reserve system, there would be significant benefit from hierarchically organized parallelization, to enable a 100k-processor system run (imagine a multi-hundred simultaneous, distributed instantiation of the ecosystem simulation, which itself might be a 64-node data-parallel application). Whether the envisioned exascale system even provides the right architecture for this application could be debated, but the point is that it does not require significant effort to scale up moderately sized ecological models to result in large computational needs, resulting in the ability to address relevant and interesting problems.

Data integration would be critical to success. A candidate calculation would involve evolutionary correlations of networks and functions (phenotypes). To the extent that ecologists are able to refine mechanistic mathematical models in a way that is increasingly faithful to reality, one could easily conceive of petascale computing demands for simulating an entire ecosystem from its underlying biological and physical components. However, it is worth pointing out that the tradition in ecology is to simplify and scale back models — indeed, to err on the side of oversimplification; "realistic" models have long been mistrusted in favor of either highly abstract mechanistic (theoretical) models and/or simple phenomenological (statistical) models. In part this is for good reason: ecologists do not yet fully rely on their own more detailed mechanistic models (there being a lack of the ecological equivalents of physical laws, testing approximating models via experimentation and observation is difficult, and each real system seems to have its own unique features). This could in fact partly be a historical artifact: few ecologists are even aware of the computational possibilities now afforded by HPC systems. In a sense, one might argue that developments in this area are limited due to apparent belief in computational obstacles that no longer exist. Opportunities for making forays into developing complex ecological simulations and to enhance model output with observed data has the potential to lead to refinement and progress in this area.

[2] http://www.atlss.org

Class	Application	Benefits from	References
Data Mining	*De novo* genome assembly from high-throughput sequencers	Large shared memory	[49, 63, 79, 150]
	Clustering and correlation analysis of galaxies from cosmological simulations	Large shared memory	[40, 58, 97]
	Interaction network analysis	Fast I/O	[40]

Table 4.1: Example data-intensive HPC applications

4.6 Exascale Data Intensive and Data Mining Applications

In talking about exascale we should also consider improving the performance of data-intensive applications, not just floating-point intensive applications. By talking to users, examining their applications, and participating in community application studies [66, 129, 130], we have identified data-intensive HPC applications spanning a broad range of science and engineering disciplines that will benefit from fast I/O and large, fast shared memory packed onto a modest number of nodes, most notably data mining and predictive science applications that analyze large model data.

In a typical *data mining application*, one may start with a large amount of raw data on disk [134]. In the initial phase of analysis, these raw data are read into memory and indexed; the resulting database is then written back to disk. In subsequent steps, the indexed data are further analyzed based upon queries, and the database will also need to be reorganized and re-indexed from time to time.

As a general rule, data miners are less concerned about raw performance and place higher value on productivity, as measured by ease of programming and time to solution. Moreover, some data-mining applications have complex data structures that make parallelization difficult [149]. Taken together, this means that (for example) a large shared memory architecture and matching shared-memory programming model will be more attractive and productive than a message-passing approach for the emerging community of data miners. I/O speed is also important for accessing data sets so large that they do not fit entirely into memory.

A typical *predictive science* application may start from (perhaps modest) amounts of input data representing initial conditions but then generate large intermediate results that may be further analyzed in memory, or the intermediate data may simply be written to disk for later data-intensive post-processing. The former approach benefits from large memory; the latter needs fast I/O to disk. Predictive scientists also face challenges in scaling their applications due to the increasing parallelism required for petascale and beyond [149]; they benefit from large memory per processor as this mitigates the scaling difficulties, allowing them to solve their problems with fewer processors.

4.6.1 Data-Intensive Balance and the Latency Gap

The growth-rate of data is exponential in many application domains. For example, gene sequence data is increasing at a rate almost faster than it can be stored, much less analyzed [40], and the same is true of astronomical data [51]. As we forecast the characteristics of data-intensive applications in the future, we find that today's supercomputers are, for the most part, not particularly well-balanced for their needs. Creating a balanced data-intensive system requires acknowledging and addressing an architectural shortcoming of today's HPC systems.

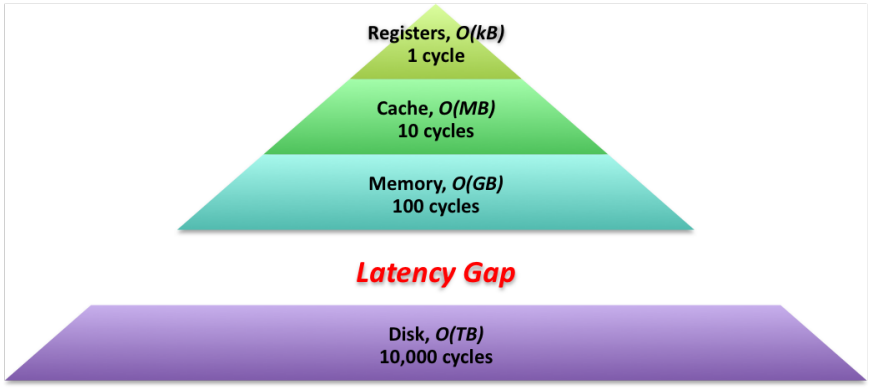

Figure 4.5: Typical memory hierarchy for today's HPC systems. Each level shows the order of capacity and typical access latency. Note the two-orders-of-magnitude gap between main memory and disk, which is expected to be bridged by new storage technologies.

4.6.1.1 Tipping the Balance to Data

When the amount of data that an application must analyze exceeds the capacity of main memory, the application may have to use many processors to obtain the amount of memory required. However, unless the calculation is highly parallel, it will incur substantial communications overhead. Also, reserving a lot of processors just for their memory is not very cost or energy effective.

Thus, a machine designed for data-intensive computing should have a large shared memory accessible to a much more modest number of processors than is common in today's systems. With a few exceptions, supercomputers in the current HPC portfolio have only 1 or 2 GB of main memory per core, while each core is capable of $\approx$ 10 Gflop/s. This means data-intensive calculations such as those in Table 4.1, when run on these machines, are limited, with respect to performance and capability, by the available memory rather than by the available flops. Within the nodes this translates to a need for the highest possible memory capacity and bandwidth and, for data sets that exceed the capacity of main memory, the fastest possible I/O to the next level of storage.

4.6.1.2 Hardware to Bridge the Latency Gap

While each level of the memory hierarchy in today's typical HPC systems increases in capacity, the costs of each increase are latencies that increase and bandwidths that decrease by orders of magnitude at each level (Figure 4.5). However, today's systems have a *latency gap* beyond main memory — the time to access disk is orders of magnitude greater than the access time to local DRAM memory. It is almost as though today's machines have a missing level of memory hierarchy that should read and write an order of magnitude faster than disk, a gap that is being filled by new storage technologies such as flash memory and phase change memory (PCM).

Since some data sets are becoming so large they may exceed the combined DRAM of even large parallel supercomputers, a data-intensive computer should, if possible, have a level of memory hierarchy between DRAM and spinning disk. By filling this missing level, a data-intensive exascale architecture could reap two possible benefits: to make main memory bigger or to make disk faster.

4.6.1.3 Software to Bridge the Latency Gap

Data-intensive calculations that arise outside of traditional HPC, such as data-mining applications, may not have even been parallelized yet [129], and achieving efficient data decomposition for them may be hard or impossible. For example, the fact that any gene may match any other gene or the fact that any node on the Internet may talk to any other makes it hard to partition the analysis involved in genomics or social networking, as in classical domain decomposition. As a result, these applications usually need a global address space. While technologies exist to extend the address space beyond physical memory (*e.g.,* PGAS languages), performance suffers when accessing memory outside of the current node. A data-intensive system designed to share memory across nodes should therefore have software that helps to alleviate the latency of inter-node memory accesses and, ideally, should help applications avoid such accesses by improving their data locality automatically.

Several consensus points emerged in the above studies as to the current and projected needs of data-intensive computing going towards exascale:

- Serial and/or modestly parallel data-intensive application developers expect their memory requirements to increase significantly. The reason is simple: data volume is exploding while methods to mine these data are not becoming massively parallel at the same rate; in other words, generating data is relatively easy, while inventing new parallel algorithms is hard. Therefore, shared-memory requirements of > 1 TB are expected to be the norm in many serial and modestly parallel data-intensive domains circa 2017 [130].

- Many data-intensive applications in genomics [129] and other domains have significant I/O requirements and are serial or only modestly parallel [129]. A typical data-intensive calculation today may scale poorly beyond only 128 or 256 processors [129]. This situation is not expected to improve dramatically with current trends.

- Additionally, data-intensive I/O requirements do not involve just reading or writing large chunks of data; they may also have significant amounts of small random I/O access. Therefore, simply combining disks in large RAID arrays to increase I/O bandwidth is not a panacea. Only a technology shift that reduces latency (increases random I/O access speed) can improve overall performance when there is significant random I/O.

- We estimate that, taken as a whole, data-intensive applications that need large amounts of memory, that exhibit poor scalability beyond 256 processors, or that are I/O-bound make up about 10% of the current HPC workloads. This low percentage may, in part, be due to self-elimination – there are science domains that cannot use the big flops machines effectively [129, 130]. In the future, particularly if machines that meet these needs are deployed, we expect this percentage to grow significantly.

4.6.2 Sample Data Intensive Applications

We now discuss some sample data-intensive applications, and consider their future memory and data growth.

4.6.2.1 Analyzing Interaction Networks

Interaction networks, or graphs, occur in many disciplines. These describe the relationships among objects in terms of how they are linked to each other. Often these graphs are constructed in terms of social networks [40]. Such networks (or graphs) have applicability in, for example, epidemiology

[110,122], phylogenetics [89], systems biology [44] and population biology [122] as well as in studies that combine information from these fields [40].

All these areas share a common theme — they combine information from multiple databases in order to answer questions about how the interactions lead to phenomena such as spread of disease. Key parameters that define their computational requirements include the latency to do a database lookup, and the total amount of fast storage available to the databases.

As an example, recent investigations combine social network databases with medical records and genomic profiles to explore questions such as the existence of genetic resistance to AIDS [48]. The analysis may proceed by identifying associates of diagnosed AIDS patients and analyzing their phenotype (*i.e.,* whether or not they have the disease) for correlation with genotypes such as the HLA genotype.

4.6.2.2 De novo genome assembly

Genome sequences are produced in a process known as fragment assembly: millions of tiny fragments of a genome are generated, read, and pieced together computationally. This is analogous to shredding several copies of a book, reading each fragment, and reconstructing the book.

New sequencing machines generate fragments (called reads) in orders of magnitude less time and cost than first used to sequence the human genome. Codes such as EULER-SR [49], Velvet [150], and Edena [79] automate the assembly. Each of these codes scans a file containing all reads, constructs a graph in memory that encodes all information from the reads, and then modifies the graph to produce the final genome sequence.

Key parameters that define the computational requirements for assembly are the length of the genome and the coverage, the latter being the average number of times a nucleotide is covered by a read. Interesting problems require large amounts of memory. Because of the complicated graph, none of the codes have been parallelized, which makes large shared memory very attractive.

A recent run of Velvet to assemble a plant genome with 120 Mbp (million base pairs) required 90 GB of memory [63]. Assembling the entire 3-Gbp human genome would require roughly 25x as much memory for the same coverage.

4.6.2.3 Data mining applications

Data mining is the process of analyzing large amounts of raw data and extracting useful information. The role of data mining in science and engineering will grow as the amount of data produced by experiments and observations continues to increase [134]. We anticipate applications across a continuum from simple database queries, in which samples of data are extracted for inspection, to sophisticated computations such as cluster analyses, which could run for many days. Although data need not be stored in a database, we expect the amount of database usage to increase rapidly, because of the need to efficiently organize the vast amounts of data [137]. Scientific databases in astronomy and Earth science already are terabytes in size and continue to grow. For example, the Sloan Digital Sky Survey has a 6-TB catalog archive [137], and the GEON LiDAR spatial data and indices also total about 6 TB [83]. These databases are currently stored on disk arrays and access is limited by disk read rates [137].

Future data mining investigations will involve the analysis and comparison of data from diverse sources. Metcalfe's law [68] predicts that the possibility of new discoveries grows quadratically with the number of federated databases [71]. For example, one could imagine studying the health of a community using data such as population demographics, input from various environmental sensors, and social science observations such as crime statistics. Another example is using the

ever-increasing myriad of biological data available to assist in the process of drug design [59]. By combining the information in databases on topics such as genome sequences, protein structure and function, and the biomedical literature, a researcher can generate a test hypothesis more easily because they have all the information relevant to their problem readily at hand.

Numerous data mining applications will also use large shared memory, not just fast file access. For instance, in astrophysics, many projects hinge upon assigning importance to over-dense regions in a set of points. Examples include identifying collapsed halos in a cosmological simulation, determining whether two galaxies have merged, finding clusters of galaxies in a survey, or locating dwarf galaxies in star counts. HOP [40] is a density-based clustering method, with poor scalability, that determines the location of such regions and mines the output of cosmological simulation codes such as Enzo [98]. For maximum performance such a calculation requires rapid access to the many-TB-sized simulation output files. The best approach is to park the output files in main memory.

4.6.2.4 Predictive science applications that are data-intensive

The geosciences include many applications that solve *inverse problems*, *i.e.,* problems in which measured data are supplemented with computer models to reconstruct 3D fields (often time-dependent as well) over a domain of interest.

Oceanographers in the ECCO consortium (Estimating the Circulation and Climate of the Oceans) are generating databases of ocean state as functions of space and time throughout the world's oceans. Such *ocean state estimation* supplements sparsely measured state data with a sophisticated ocean general circulation model (GCM), embodied in MITgcm. This code solves a nonlinear minimization problem by iteratively sweeping through the forward and adjoint GCM equations. These computations generate a large amount of intermediate data during the forward sweep that need to be reused during the backward, adjoint sweep.

A frequent problem for weather and climate modelers is to obtain initial conditions for subsequent simulations using atmospheric GCMs. This *data assimilation* problem is analogous to the ocean state estimation problem and is amenable to similar adjoint solution approaches [78, 146], which again benefit from the availability of additional memory.

Geophysicists within the Southern California Earthquake Center are using *full 3D seismic tomography* [146] to obtain a 3D elastic structure model of the Earth's crust under Southern California. Another class of predictive science applications exhibits only modest scalability. Well known examples are quantum chemistry packages, such as Gaussian [10] and GAMESS [11], and structural engineering packages, such as ABAQUS [12]. The equations solved by these packages, such as the matrix diagonalization step in the Hartree-Fock method from quantum chemistry, exhibit fundamental obstacles to efficient parallelization. Adding more and more processors simply does not improve performance.

4.7 Real-time Departmental Extreme Scale Applications

Extreme scale computers provide the opportunity to read in and react to an enormous amount of data in realtime. As an exemplar, consider the ability to analyze in real-time 1 million sources, such as field sensors that can produce $>$ 1 GB/s of visual data each (*e.g.,* the sources could be hi-res. cameras with 1920x1080 64 bit pixel resolution). A machine would have to be able to read in at the rate of 1 petabyte per second assuming there was no further computation involved. Thus the ability to do petascale computing at the departmental level could enable surveillance in real time to embedded sensors on a wide scale. A detailed discussion of real-time and other specialized requirements in embedded software can be found in Appendix A.1.

4.8 Framework Technology

Enabling scientific applications on extreme-scale HPC systems is a multi-disciplinary, multi-institutional, multi-national efforts. Application code complexity is growing to a point that it is increasingly difficult to make forward progress without high-level organizing constructs. Advanced parallel languages are necessary to make programming of complex systems tractable, but they are not sufficient. Frameworks provide higher-level organizing constructs for teams of programmers that clearly segregate the roles and responsibilities of application team members along the lines of their expertise as shown in Figure 4.6. Languages must work together with frameworks for a complete solution.

Frameworks confer the following benefits:

- Separate roles and responsibilities of expert programmers from that of the domain experts/-scientist/users (productivity layer vs. performance layer).
- Define a social contract between the expert programmers and the domain scientists.
- Enforce and facilitate software engineering discipline.
- Encapsulate complex parallel implementation details in the framework so that they can be hidden from scientists and end-users.
- Allow scientists/users to code nominally serial plug-ins that are invoked by a parallel schedule.
- Support modular composition of multi-physics applications using components supplied from different developers and vendors
- Restrict code rewrites to the driver level as the hardware industry moves towards multi-core architectures with massive parallelism.
- Reduce software development costs.

The definitions and some examples are taken from the High Performance Computing (HPC) Application Software Consortium (ASC) white paper on frameworks from March 17, 2008 [72]. The state of evolution of application software can be measured in terms of level of component interoperability, given in Figure 4.7. The level of interoperability is defined by the degree to which components must conform to a set of rules set by the framework in order to achieve interoperability. We refer to three levels of interoperability:

- *Minimal Component Interoperability:* A majority of the existing commercial solvers based on legacy codes can be described as having minimal component interoperability. Individual physics models are handled by separate solvers. Therefore, the physics domains are completely un-coupled; static analysis might be performed across physics domains using file translators or common interchange file formats, also referred to as workflow coupling. The only requirement that the framework imposes on the individual solvers is that they share the same file format.
- *Shallow Component Interoperability:* Several of the leading simulation software vendors have released simulation suites that are beginning to exhibit shallow component interoperability. At this level, physics models are loosely coupled at some time step or discrete event. Each solver maintains its own internal state representation of its respective domain. Common data is exchanged using wrappers to some interchange interface over a network service. Therefore, the development guidelines imposed by the framework stop at the interface to the component.

Developer Roles	Domain Expertise	CS/Coding Expertise	Hardware Expertise
***Application**: Assemble solver modules to solve science problems. (eg. combine hydro +GR+elliptic solver w/MPI driver for Neutron Star simulation)*	Expert	Intermediate	Novice
***Solver**: Write solver modules to implement algorithms. Solvers use driver layer to implement "idiom for parallelism". (e.g. an elliptic solver or hydrodynamics solver)*	Intermediate	Expert	Intermediate
***Driver**: Write low-level data allocation/ placement, communication and scheduling to implement "idiom for parallelism" for a given "dwarf". (e.g. GasNET or Cactus PUGH)*	Novice	Intermediate	Expert

Figure 4.6: Frameworks clearly separate roles and responsibilities for a large teams of programmers — enabling computer science experts, numerical algorithms experts, and domain scientists to collaborate together productively.

The framework developer need only provide a standards-based interface that is external to the solver to achieve interoperability. In the commercial market, shallow component interoperability is usually limited within a single vendor's offerings, although some open interchange standards are beginning to emerge based on web services.

- *Deep Component Interoperability:* A few leading HPC laboratories have developed physics component frameworks where the solvers share a common service infrastructure for communications and data management. Physics models can be tightly coupled at this level of interoperability. In this case, the component developer must also heed rules regarding the internal organization of the component in order to achieve interoperability with the framework. This approach hides the complexity of the underlying hardware platform and offers higher-level abstractions for managing parallelism, thereby providing opportunities for improved platform portability and parallel system library optimization by the hardware vendors themselves.

4.8.1 NASTRAN OMD-SA case study

The Open Multi-Discipline Simulation Architecture (OMD-SA) framework [131] is an extensible and flexible Service Oriented Architecture (SOA) for scalable multidisciplinary engineering analysis that has been designed by MSC Software. It supports efficient data transfer for modular multi-physics simulations on HPC systems. Whereas older generation codes used data files to exchange model data between various solvers in a multi-physics application, the OMD-SA architecture enables direct transfers between the components as well as a composition system for combining solver components into a single application. As the simulation data can easily be in the gigabytes or even terabytes, the transfer of data across service layers must be optimized. Specifically, the framework must avoid unneeded copying or transfer of data if it is not absolutely necessary. Services in this framework can be either in a local application space or in a remote application space. Local services should cost

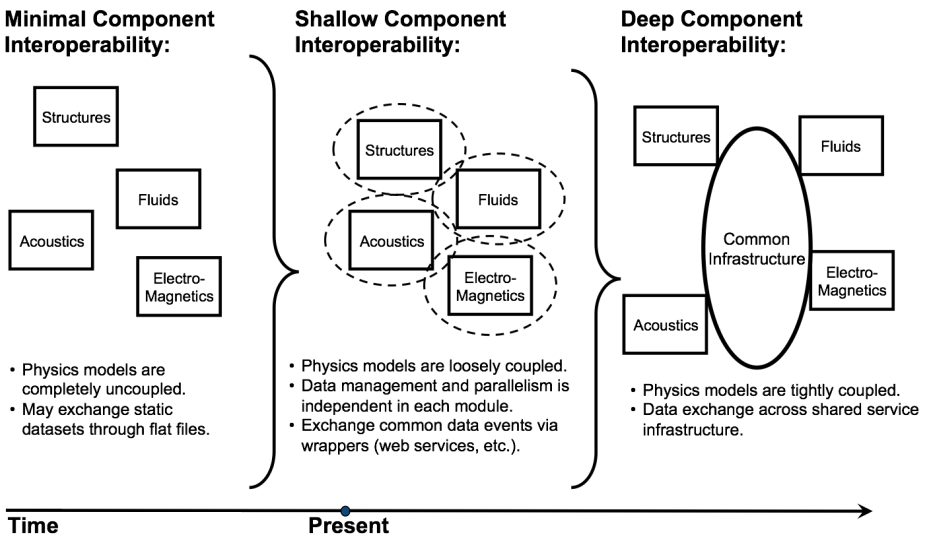

Figure 4.7: Evolution of Software Framework Integration *(From High Performance Computing (HPC) Application Software Consortium (ASC) Summit, March 17, 2008).*

no more than a simple function call so as to ensure that the framework imposes negligible overhead on the overall performance. The framework supports language interoperability so that existing optimized code written in FORTRAN, C, or C++ can be used to implement services within the framework to support this capability. Remote services leverage standard and emerging network protocols to maximize performance. A single service can be used both locally and remotely, and it is up to the framework to determine the usage and the appropriate optimizations relevant to each case. Services must be highly tuned internally for efficient processing, using multi-threading, caching, efficient sharing of memory across services where possible, etc.

There are 4 main elements to the OMD-SA architecture:

- *Component Framework* The component framework is an open SOA model where the services are available on-demand. The component framework is comprised of multiple layers. The services are connected through the Simulation Bus and a common data model that assures scalability and effective application of the services to a simulation application.

- *Simulation Clients* The services are exposed to the various players in the simulation process through different clients, both rich and thin, that address the specific user needs.

- *External Services* External services are available to OMD-SA through standard open plug-in technology. Legacy applications, 3rd party applications, as well as in-house developed applications can be exposed as services to OMD-SA applications.

- *Enterprise Service Bus* The Enterprise Service Bus can be either an existing ESB within an enterprise or a third-party ESB to which OMD-SA will interface. This allows for the use of external enterprise data and processes within a simulation process, *i.e.,* using geometry from PDM within simulation. This allows for the use of simulation services and processes within external enterprise applications.

OMD-SA uses emerging standards for interface definitions and Internet protocols, including OMG-IDL (ISO standard 14750) and WDSL for service description and interface definition, UDDI

for service discovery, and SOAP for service invocation/interaction. OMD-SA is an open platform for customers and partners to address extended or proprietary applications through the customizable service APIs, the SOA, and the programmable user interfaces.

OMD-SA is an example of a shallow component interoperability framework in terms of the interfaces it presents to developers. Integrating a component into this framework does not require any changes to the internal data model employed by the solver. Each component is able to share data with other physics solvers using the standards-based APIs to write data to the simulation bus in an operation that looks like writing a file to disk (as would be the case for older multi-physics simulations), but can reside in memory for local data exchanges between simulation clients. All parallelism remains internal to each component, which enables the solvers to be incorporated with little or no changes to their internal data model or data structures. While the shallow interoperability model simplifies coupling of components, the approach does not support any form of abstraction or modularity for the implementation of the parallelism.

4.8.2 Cactus case study

The Cactus Framework [121] is an open source, modular, portable programming environment for collaborative HPC computing. Cactus consists of both a programming model with a set of application-oriented APIs for parallel operations, management of grid variables, parameters etc, as well as a set of modular swappable tools implementing parallel drivers, coordinates, boundary conditions, elliptic solvers, interpolators, reduction operations, and efficient I/O. Although Cactus originated in the numerical relativity community where the largest HPC resources were required to model black holes and neutron stars, Cactus is now a general programming environment with application communities in computational fluid dynamics, coastal modeling, reservoir engineering, quantum gravity and others.

Cactus consists of four main elements:

- The Cactus Flesh, written in ANSI C, acts as the coordinating glue between modules that enables composition of the modules into full applications. Although the architecture is different, the Flesh plays the same role as the Enterprise Service Bus for the OMD-SA framework. The Flesh is independent of all modules, includes a rule based scheduler, parameter file parser, build system, and at run time holds information about the grid variables, parameters, methods in the modules and acts as a service library for modules.

- Cactus modules are termed Thorns and can be written in Fortran 77 or 90, C or C++. Each thorn is a separate library providing a standardized interface to some functionality. The thorns are similar in nature to the Simulation Clients in OMD-SA, but Cactus further externalizes the implementation of parallelism for the thorns, enabling different architecture-specific implementations of parallelism to be plugged in. Thorns providing the same interface are interchangeable and can be directly swapped. Each thorn contains four configuration files that specify the interface between the thorn and the Flesh or other thorns (variables, parameters, methods, scheduling and configuration details). These configuration files have a well-defined language and can thus be used as the basis for interoperability with other component based frameworks.

- Drivers are a specific class of Cactus Thorns that implement the model for parallelism. Each solver thorn is written to an abstract model for parallelism, but the Driver supplies the concrete implementation for the parallelism. For example, the PUGH (Parallel UniGrid Hierarchy) driver implements MPI parallelism, whereas the ShMUGH (Shared Memory UniGrid

Hierarchy) driver provides a shared memory/threaded implementation for the parallelism. The application can use different drivers without requiring any changes to the physics thorns. However, the thorns must be written specifically to the guidelines of the Cactus framework. The modular drivers for implementing parallelism are both the principle advantage of the deeply integrated framework model, but also the most daunting part due to the need to conform to framework coding requirements to take advantage of this capability.

- Cactus modules or thorns are grouped into Toolkits. Cactus is distributed with the Cactus Computational Toolkit that consists of a collection of thorns providing parallel drivers, boundary conditions, scalable I/O etc to support applications using multi-dimensional finite differencing. Community toolkits are provided or are under development by different application areas such as Numerical Relativity and Computational Fluid Dynamics.

The modular design of Cactus with swappable thorns provides several important features:

- Third-party libraries and packages can be used by applications through the abstract Cactus interfaces, decreasing application reliance on any particular package and making it possible to switch to new capabilities as they are available. For example, instead of using the Uni-Grid parallel driver PUGH distributed with Cactus, applications can use a variety of other independent adaptive mesh refinement drivers such as Carpet, PARAMESH, SAMRAI.

- New I/O methods can be added as thorns, and are then available to applications as a parameter file choice.

- Cactus currently supports a variety of output formats including HDF5, NetCDF, ASCII, JPEG, FlexIO, and provides architecture independent checkpoint and recovery along with interfaces for parameter steering and remote visualization.

Cactus has already been shown to scale to large processor numbers (4,000 to 33,000 cores) for different applications, and has active user and developer communities, along with funding from a range of agencies to both improve the infrastructure and build new application areas.

Whereas the shallow component interoperability framework enables modular composition of solver components into a multi-physics application, providing a scalable and modular model for parallelism requires deeper modifications to the code base. Deep component interoperability frameworks such as Cactus and Sierra (discussed in the next section) present an approach where the abstract model for parallel computation is external to each of the components. This requires a larger initial investment in code, but offers additional performance and scalability benefits down the road as systems move towards a massive parallelism on multicore systems.

4.8.3 Sierra case study

Sierra is a software framework [57], which is used for multi-physics computational mechanics simulations primarily targeting finite element and finite volume methods for solid mechanics, heat transfer, fluid dynamics with reacting chemistry, and multi-physics permutations of these mechanics. Sierra is designed around an in-core data model for supporting parallel, adaptive multi-physics on unstructured grids, with an emphasis of simultaneously handling parallelism, dynamic mesh modification, and multiple mesh solutions and transfer operations. Sierra also provides common services and interfaces for linear solver libraries, dynamic load balancing, file input parsing, and mesh file I/O. It was designed to unify and leverage a common base of computer science and data

capabilities across a wide range of applications, and facilitate research, development and deployment of multi-physics capabilities, while managing the complexities of parallel distributed mesh data.

Through its solvers class capability and external interfaces, Sierra provides plug-in capability of a range of solver libraries for different mechanics. Plug-ins play the same role as the thorns in Cactus nomenclature and the Simulation Clients in OMD-SA. At the coupled physics level, Sierra provides a procedural language to support operator splitting methods to couple mechanics, including the ability to iterate to convergence and to sub-cycle physics modules relative to one another. The procedural language, called SolutionControl, allows a user to specify how the coupled mechanics for the various Sierra Regions are executed in sequence, how variables are mapped between the computational domains of each region, and how solution convergence is controlled at the coupling level before moving the simulation forward in time. SolutionControl is the basis for composing solver components into composite multi-physics applications, much as the OMD-SA scripting environment and Cactus Flesh is used to support module composition in those respective frameworks. Sierra also supports limited tighter coupling through forming full Jacobians for multi-physics within a single Sierra Mechanics Region.

Sierra's support for parallelism is pervasive, and is designed to limit the amount of work and complexity associated with parallel data structures for the mechanics developer, so that they can focus on the physics-relevant aspects of their solver module. Like Cactus, the implementation of the parallelism is externalized from each of the solver modules, so that the implementation of the parallelism need not be replicated for each module that comprises the framework. Supporting this capability requires the solvers to adopt some common data structures and conform to framework coding requirements, which is the hallmark of a deeply integrated framework.

4.8.4 Comparison across Frameworks

Examining both shallow and deeply integrated frameworks for modeling and simulation on parallel computing platforms, some common themes have emerged. Physics solvers in these frameworks are implemented as modular software components so they can support flexible reconfiguration for different multi-physics problems. The coupling of physics modules follows loosely coupled at some time step or discrete event as opposed to tight coupling. The framework provides a flexible composition environment that matches the requirements of the application domain. In addition to these common features, deep component interoperability frameworks also partition the implementation of parallelism into separate components, in other words abstracting the implementation of parallelism, to reduce programming errors and support performance optimization and portability across diverse hardware platforms. The key distinction between shallow and deep component interoperability frameworks is that shallow framework components manage their own parallelism and data structures and exchange data using external interfaces, whereas a deep framework components externalize the parallelism and data structures so that they can be optimized and ported independently from the solver component implementations. Our belief is that deep interoperability will increase in importance for extreme scale systems.

Frameworks also provide a base set of services and build tools that simplify the customization of existing software components, and building and integration of new components within the framework. Examples of such services are I/O services, memory management services, error handling services, etc. As existing software modules are to be imported into a framework, their outer layer (a main program calling the subroutines) is peeled off and rewritten as declarations to the framework, which describe the high-level dataflow between the components. The framework manages the coarse-grain dataflow of an application, which is required for efficient parallelization. However,

fine-grain dataflow within subroutines remains under control of the individual components and thus remains highly efficient.

The attraction of shallow integrated frameworks is that they minimize the amount of code rewriting internal to each of the solver components. Each component interacts through a common SOA interface that preserves the opaqueness of the internal architecture of the component. However, such an architecture makes it difficult to impose constraints on the data layouts employed within each module, and therefore can lead to inefficient coupling between components due to the extra layer of data copying that must be employed between components with incompatible data layouts. It also limits the ability of a third party to innovate the implementation of parallelism for the components without getting inside of each module and rewriting the solver implementation. However, the shallow framework component model is well tested in enterprise applications and would require the least amount of effort for ISVs to cooperate. These shallow integration framework architectures consist of a few (tens) of components, each operating on large amounts of data for a significant amount of time. Overheads due to staging, invocation, load distribution etc. are amortized over the run time of the components' activity. One crucial advantage of shallow frameworks is that they arise naturally from preexisting, independent, large software packages as the need for coupling arises. The deeply integrated applications require that solvers agree upon an external data representation for the model data that is exchanged between solvers. This architecture also manages the parallelism external to the solvers. The framework then defines the optimal data layout that is common to all of the components, so as to minimize the amount of data re-copying required to couple components together. In addition, the deeply integrated approach enables the implementation of parallelism to be separated from the solver components, so that innovations in parallelization methods (particularly for multicore processors) can be exploited by the solvers without requiring them to be rewritten. However, the price of such a deep level of integration is that existing solver components must all be rewritten to conform to the frameworks restrictions. This requires a more significant initial investment and a deeper level of cooperation among ISVs, but can lead to a platform that is more scalable to future trends in concurrency.

Deep component interoperability framework architectures consist of many (hundreds) of smaller components, each invoked many times in parallel, operating only on small subsets of the overall data set, supervised by a framework driver layer. Efficiency is guaranteed by the driver layer's control over the data layout, which enables it to orchestrate calculations and relocate data as required. Examples of deeply interoperable framework architectures are Cactus, SIERRA, Chombo, and UPIC. The crucial advantage of deep component interoperability frameworks is the close yet efficient interaction since parallelization is handled by the driver layer, which allows for more accurate multi-physics simulation.

In summary, a tightly integrated framework architecture consists of several key components:

- A backbone which orchestrates the overall simulation, touching only metadata
- A driver layer providing the "heavy lifting", handling memory management and idiom for parallelism
- Parallel data-layout (domain-decomposition) management components that define or modify the data structures on which the simulation operates (structured, unstructured, AMR)
- Solver components, operating on driver-defined subsets of the data
- Statistics/introspection components, collecting metadata and providing feedback (provenance, performance, progress monitoring)

4.8.5 Domain-Specific Application Frameworks and Libraries

In the following we briefly examine three applications from chemistry that have all departed from norm of generic MPI+OpenMP parallelization and have instead pursued innovative solutions to their computational problems. One enabling characteristic in common is that all three are the result of close and long-term collaborations between application experts and computer scientists.

4.8.5.1 NWChem case study

NWChem was the first quantum chemistry code developed specifically for massively parallel computers and has been funded since circa 1992 by the DOE as part of the Environmental Molecular Sciences Laboratory at Pacific Northwest National Laboratory. Other institutions in Japan, Europe and the USA (*e.g.,* ORNL) contribute to the project and the software is in use at nearly all supercomputer sites worldwide. The genesis of the project was the recognition by Thom Dunning that lack of scalable software precluded chemistry, and in particular environmental molecular science, from exploiting the massively parallel computers emerging in the early 1990's. From the outset, the project adopted a multidisciplinary approach. Molecular electronic structure uses a wide variety of methods to solve a single chemical problem, and the associated computations have complex data structures that can be thought of as block-sparse multi-dimensional matrices. The nature of the sparsity and computation requires careful load-balancing to achieve scalability and the size of the data structures requires distribution to enable computations larger than those possible on a single computer. These challenges led to the development and adoption of the Global Array (GA) library [13] that provides one-sided access to arbitrary blocks of logically-shared but physically distributed matrices. The library has since been extended to other data structures to facilitate its use in many other disciplines. It remains the only portable distributed-shared memory programming library in use since the 1990's.

The combination of one-sided access and data structures/abstractions chosen for the domain revolutionized the writing of scalable applications in chemistry; all scalable chemistry codes either use GA or a derivative of it. Most of the algorithms implemented with GA are inherently more scalable than their MPI counterparts since the one-sided access eliminates unnecessary synchronization, is often closer than message passing to the actual hardware capabilities, and greatly facilitates full dynamic load balancing since computation and data can be easily relocated. GA makes aggressive use of overlapped, asynchronous communication, and also handles the complex translation of addresses from the application (multi-dimension matrix patches) to the hardware (non-contiguous blocks in partitioned linear address spaces). The only components of NWChem still written in message passing are those that for reasons of performance or correctness benefit from the (weak) synchronizations implied by exchange of messages, such as tight coordination of data motion in a parallel FFT or the dependent graph of tasks in a classical matrix factorization.

Going forward, the three main downsides of GA are lack of language support, the fact that its memory model is tied to a specific machine model, and the fact that it only emphasizes two levels of the memory hierarchy (three if disk arrays are considered). The lack of language support is a large source of errors and increased complexity since the user is responsible for tiling accesses to global data structures, managing local arrays, and correctly invoking GA interfaces. GA was consciously designed as a memory model so as to keep its implementation efficient on available computers, but this has now become a major limitation. Within GA, all one can do is move data to/from the computation and this turns out to be insufficient for efficient management/use of more complex data structures. For instance, a distributed sparse tree or hash table would be very inefficient and complex to manage with GA. What is needed is the ability to move computation to data. Finally,

while GA does a good job at managing the coarse grain data motion and parallelism it does little to help with increasing concurrency within SMP nodes. In particular, by forcing the programmer to chose which data to distribute and which to replicate what was once a strength of GA has now locked its existing applications into a specific choice of granularity for their parallelism.

4.8.5.2 NAMD case study

NAMD is a parallel molecular dynamics code designed for high-performance simulation of large bio-molecular systems and was recipient of a 2002 Gordon Bell Award. It is based upon the `Charm++` parallel programming environment and represents a long-standing collaboration between distinguished UIUC researchers Sanjay Kale (Computer Science) and Klaus Schulten (Molecular Biochemistry). `Charm++` is now the only widely used parallel programming model that emphasizes virtualizing all aspects of the computation, with an intelligent runtime taking full responsibility for scheduling and data management. Parallel programs are composed in terms of messaging between objects (chares) addressed in name spaces (chare arrays) without reference to the underlying hardware. This virtualization and separation of responsibilities encourages the expression of the intrinsic parallelism in the application as advocated in Chapter 5. This is exemplified by NAMD being the first chemistry application to execute efficiently on tens of thousand of processors on the IBM BG/L primarily because it was the only application that had expressed sufficient parallelism. Due to its scalability, and the support of Kale's and Schulten's research groups, NAMD is widely used with an increasing amount of science functionality accruing around it.

The main limitation to date of the `Charm++` environment might also be interpreted as a defect of the rest of the world. `Charm++`'s complete virtualization means that its applications are largely incompatible with existing MPI applications unless those applications are also imported into `Charm++` using the AMPI library. There are also performance issues associated with fine grain virtualization and object models that can be alleviated with more powerful language, compiler, and runtime technologies.

4.8.5.3 TCE case study

The Tensor Contraction Engine (TCE) arose out of an NSF ITR project and a DOE SciDAC project, and is the application of compiler optimization and source-to-source translation technology to craft a domain specific language for many-body theories in chemistry and physics. The underlying equations of these theories are all expressed as contractions of many-dimensional arrays or tensors. There may be many thousands of such terms in any one problem but their regularity means that they can be translated into efficient massively parallel code that respects the boundedness of each level of the memory hierarchy and minimizes overall runtime with effective trade-off of increased computation for reduced memory consumption. The approach has been overwhelming successful and now NWChem contains about 1M lines of human-generated code and over 2M lines of machine -generated code from TCE. The resulting scientific capabilities would have taken many man-decades of effort; instead, new theories/models can be tested in a day on a full-scale system. In combination with the OCE (operator contraction engine) that turns Feynman-like diagrams into tensor expressions, the TCE represents perhaps the first end-to-end production quality example of a solution to the semantic gap between applications and hardware.

Clearly, domain specific languages will be an integral part of future computational science and we note that several of the HPCS languages had at their core the idea of being extensible and readily specialized to new fields. However, translating the narrow success of the TCE into broad relevance remains a challenge. For instance, how can application scientists make effective use of

the optimization and compilation tools of computer science without having a computer scientist at their side? What elements are in common between languages tailored to chemistry or material science or linguistics or forestry, and how do we ensure that such programs can inter-operate when composing multi-physics applications?

4.9 Footprints

A "footprint" is the trace of resource usage left by an application as it executes on a system. Understanding such footprints is thus important to understanding whether or when new Extreme Scale systems are in fact capable of supporting the kinds of Extreme Scale systems discusses elsewhere in this chapter. In the remainder of this section, we will examine several different types of footprints.

4.9.1 Application Footprints — System Memory

The quantities of required system memory are highly dependent on specific applications and the associated problem size, architecture balance, research in progress, and anticipated future research. Research includes new-era weak scaling concepts such as multi-scale, multi-physics, new models, interactions, mitigation analysis, data mining, data-derived models, as well as new mathematics and algorithms. Our recommendations are as follows:

- For petascale systems, we recommend O(100TB) to O(1PB) of system memory. These numbers are derived from the following:
  - For all applications, the quantity of system memory should not be less than the bytes/flop/s ratio of the current IBM BlueGene/L (0.083) or BlueGene/P (0.144) systems.
  - Many traditional applications may require a bytes/flop/s ratio similar to Red Storm, Jaguar, and ASCI Purple (0.3-0.5).
  - Applications or data sets may exist that require O(1PB). This is consistent with the bytes/flop/s ratio of 1.0 from Amdahl's Memory Law (1.0).
- For exascale systems, we recommend O(10PB) to O(1EB) of system memory. These numbers are derived from the following:
  - It may not be possible to perform an exascale application "existence proof" with just O(1PB) of system memory.
  - For many exascale "hero" applications, O(10PB) to O(100PB) system memory may be required.
  - For special applications with massive in-core databases, O(1EB) system memory may be required.

4.9.2 Application Footprints — Storage Capacity

The quantities of required scratch storage are highly dependent on the application and problem size. The amount of scratch storage has traditionally been driven by the need for checkpoint/restart to provide application resiliency — thus it is correlated with the system memory footprint. The need to access additional data in the future may substantially increase the scratch storage requirements due to new-era weak scaling applications. New-era weak scaling may also require that additional data be stored for post-processing data mining. We recommend that scratch storage capacity grow

at a rate faster than system memory. For many applications, scratch storage capacity greater than 10-100x system memory may be required.

- For petascale systems, we recommend O(1PB) to O(100PB) of scratch storage capacity.
- For exascale systems, we recommend O(100PB) to O(100EB) of scratch storage capacity.

The quantities of required archival storage are highly dependent on the application and problem size. The same application requirements that drive scratch storage will drive the need to access additional data from archival storage in the future. We recommend that archival storage capacity grow at a rate faster than system memory or scratch storage. For many applications, massive archival storage capacity greater than 100x system memory may be required.

- For petascale systems, we recommend greater than O(100PB) of archival storage capacity.
- For exascale systems, we recommend greater than O(100EB) of archival storage capacity.

4.9.3 Application Footprints — Node/System Memory Bandwidth and Latency

"Local" memory bandwidth and latency requirements will be driven by the fact that new applications will have reduced vertical locality on a node because the trend in new applications are to employ:

- New mathematics and algorithms that trade regular data access for improved convergence.
- Model-directed adaptive mesh refinement (AMR) for improved computational accuracy where required.
- Data-derived models, data-mining, and interaction studies in multi-physics models.

It will be a challenge for hardware designers to provide adequate bandwidth and sufficiently low memory latency to keep processors "busy". There must be adequate memory bandwidth to feed instructions to the processors/cores on a node (given the high latencies). Key techniques include attacking the traditional "Memory Wall" and employing latency tolerance techniques with massive multi-threading.

"Global" memory bisection bandwidth and latency requirements will be driven by the fact that new applications will have reduced "horizontal" locality for many of the same reasons that there will be reduced vertical locality in new applications. Bisection bandwidth requirements will be highly dependent on application characteristics such as:

- Scientific and Engineering Codes including solvers for Partial Differential Equations (PDEs) and 3-D meshes
  - Structured Grids — nearest neighbor communications
  - Unstructured Grids — indirect addressing and random communications
  - Adaptive Mesh Refinement — move extensive amounts of data around machine
- Multi-scale, multi-model, etc. — less likely to be able to map data/processes to (nearest-neighbor) locations to minimize communications
- Models may trade global data access for better convergence in new mathematics and algorithms

- Application reliability could have a significant impact on bisection bandwidth in the future

Global memory latency will be driven by these same application characteristics. Given the physical size of exascale computers, system diameter and "speed-of-light" issues will set the lower-bound on latency. Hardware designers must provide sufficiently low latency or adequate bandwidth and latency tolerance in keeping with Little's Law [14, 15].

After analyzing these application characteristics we recommend that latency be as low as possible and the bandwidth be as follows:

- For petascale systems, we recommend bisection bandwidths — O(50TB/s) to O(1PB/s). These numbers are derived from the following:
  - For all applications, bisection bandwidth should be no less than current 3-D topologies scaled to a petaflop/s performance rates.
    - Sample XT4 configuration (40 x 32 x 24) @ 318Tflop/s ← 19.4TB/s
    - Scaled to (80 x 64 x 48) @ 2.4Pflop/s ← 80.0TB/s
- For many applications: bisection bandwidth may need to approach the bandwidth specified in the DARPA HPCS program (500.0TB/s-3.2 PB/s)
  - For exascale systems, we recommend bisection bandwidths — O(10PB/s) to O(1EB/s). These numbers are derived from the following:
  - While it may be possible to perform an exascale "existence proof" at O(1PB/s), real applications will require substantially greater interprocessor communications capability.
  - For nearly all exascale applications, bisection bandwidth should scale with the quantity of memory — O(10PB/s) to O(1EB/s).

4.9.4 Summary — Design Sweet Spots

In summary, we do not expect applications to strongly scale three or more orders of magnitude efficiently to provide faster application run times with similar quantities of system memory. Some applications will be able to scale to petascale and on to exascale using traditional weak scaling. We anticipate that many applications to be run at exascale will employ new-era weak scaling and employ one or more of the following: Multi-scale, Multi-physics (multi-models), New models, Interactions, Mitigation analysis, Data mining, and Data-derived models.

In figure 4.8, we present a summary chart of the petascale and exascale application derived "footprints". On this chart, we have proposed design "sweet spots" that will address the capabilities of a majority of new exascale applications. These footprints and "sweet-spots" are in agreement with the 2007 Exascale Hardware study [62].

4.10 Two Illustrative Graph Scenarios

An examination of two graph algorithms illustrates some of the issues with exascale software. An N-body code (*e.g.,* GROMACS) in which neighbor lists of vertices are traversed to compute interactions represents one data point where the locality is easily managed. A shortest-path algorithm, on the other hand is more challenging because of fine-grain mutability and lower arithmetic intensity.

	Petascale		Exascale	
	Range	"Sweet Spot"	Range	"Sweet Spot"
Memory Footprint				
System Memory	O(100TB) to O(1PB)	500 TB	O(10PB) to O(1EB)	100 PB
Scratch Storage	O(1PB) to O(100PB)	10 PB	O(100PB) to O(100EB)	2 EB
Archival Storage	Greater than O(100PB)	100 PB	Greater than O(100EB)	100 EB
Communications Footprint				
Local Memory Bandwidth and Latency	Expect low spatial locality			
Global Memory "Bisection" Bandwidth	O(50TB/s) to O(1PB/s)	1 PB/s	O(10PB/s) to O(1EB/s)	200 PB/s
Global Memory Latency	Expect limited locality			
Storage Bandwidth	Storage bandwidth will need to grow at a faster rate than system peak performance or system memory growth			

Figure 4.8: Petascale and Exascale Application Design "Sweet Spots"

4.10.1 N-Body

N-body codes are used to study the interactions of systems ranging from proteins to galaxies. N-body codes are often structured as graph algorithms where each particle maintains a neighbor list of other particles within an interaction radius. Each time step, every particle computes an interaction with every neighbor, and this interaction is used to update the state of the particle at the end of the time step. Every few timesteps (typically 10) the neighbor lists are recomputed often using a cell structure that reflects the three-dimensional nature of the problem.

In very rough terms the code is:

```
for each timestep {
        if(neighborListsStale()) {
                forall particle in particles {
                        particle.neighbors = computeNeighbors(particle) ;
                }
        }
        forall particle in particles {
                forall neighbor in particle.neighbors {
                        computeInteraction(particle, neighbor) ;
                }
        }
        forall particle in particles {
                updateState(particle) ;
        }
}
```

This is a gross oversimplification, but captures many important issues outlined below. In particular, the analysis assumes that the particles in a cell interact only with a thin boundary layer of particles

in adjacent cells and that the partition of force computations follows the partition of the particles.

Parallelization: The bulk of the computation has parallelism of PN where P is the number of particles and N is the number of neighbors per particle. The update step has parallelism of P but accounts for a much smaller fraction of the total computation. For a large problem, there can be 10^8 particles each with 10^3 neighbors, so the amount of parallelism may be 10^{11}.

Note that this parallelism is needed both to take advantage of multiple cores and to hide latency to higher levels of the storage hierarchy.

Synchronization: The required synchronization is implied by the program data flow. Each time step must use the updated state from the last time step. There is no need for two barriers per time step although many implementations would over-synchronize the application in this manner. A looser synchronization would give improved performance.

Locality: The amount of locality depends on the underlying graph. If the graph is generated by connecting all particles within an interaction radius in a 3D space, it will have $X^{(2/3)}$ connections out of a partition of X particles. A partition of X particles can be gathered into a local memory along with a halo of neighbor particles and operated on entirely locally. XN operations are performed for each X+halo(X) particles fetched. The size of a partition X is driven by the amount of storage available at a level of the storage hierarchy. This partitioning is done recursively down the hierarchy. Achieving this locality depends on generating a good (min-cut) partition of the set of particles P. Because the graph changes over time, this partition also changes over time.

A related issue is the computational intensity of the program — how much work is done in the routine "computeInteraction"? For molecular dynamics codes, the interaction is complex — 100s of operations — giving a high arithmetic to bandwidth ratio.

Load Balance: Load balancing can be accomplished by distributing partitions to levels of the hierarchy — nodes, multi-core chips, regions on these chips, and cores. This distribution can by dynamic — and needs to be at least partly dynamic as the partitions will change as the graph is modified. As opposed to fine-grain work stealing, load balancing needs to be done at a coarser grain — that of partitions — to avoid destroying locality.

In summary, the N-body problem has lots of parallelism and easily exposed locality. It is straightforward to load-balance and synchronize the application provided the parallelism and locality can be easily expressed.

4.10.2 Shortest Path

Consider a single-point shortest path problem in a graph. Starting at a single point, for each vertex in an active set, we visit all neighbors, possibly updating their distance, and if updated we add the neighbor to the active set. In rough form the code looks like:

```
activeSet = Singleton(sourceNode) ;
while(notEmpty(activeSet) {
        for vertex in activeSet {
                for neighbor in vertex.neighbors {
                        test = Update(vertex, neighbor) ;
                        if(test) Insert(neighbor, activeSet) ;
                }
        }
}
```

Again, this is an oversimplification. For example, the active set should be managed as a priority queue, but it captures some interesting behavior.

Parallelism: The amount of parallelism depends on the shape of the graph. It starts with a single thread at the source node and increases as the active set increases. Depending on how many times a node is revisited, the average parallelism is roughly V/D where V is the number of vertices and D is the diameter of the graph. For a graph of 10^9 nodes with diameter 100, the average parallelism would be about 10^7.

Synchronization: The calls to the routine "Update" need to be atomic. Multiple vertices may have the same neighbor, and one update of that neighbor must be completed before the next is started. The difficulty of this atomic action depends greatly on implementation — specifically where the update is performed. If the update is done on the core that is co-located with neighbor, then this can be done relatively inexpensively. The atomic section of code can be run out of local memory with no long latency accesses or message round trips.

On the other hand, if one attempts to run Update on an arbitrary node, the synchronization can become prohibitively expensive. This requires acquiring a lock (or the equivalent), fetching the current value of neighbor, writing back the updated value, and then releasing the lock. This requires at least six messages (three round trips), and the number can easily be several times this number if a cache coherence protocol is involved. The amount of time a neighbor remains "locked" is a critical parameter here as it affects the amount of usable parallelism.

Locality: Locality depends on the nature of the graph and the contents of the active set at a given point in time. However, it is likely to be low. The active set represents a slice of the graph at a given distance from the source and hence is not likely to be highly interconnected. The re-use is most likely about the average degree of a vertex — since each vertex gets updated by each of its neighbors. To get this amount of locality, one needs to partition the active set so that vertices that share neighbors are in the same partition (which may not be easy).

The locality problem is made worse by the fact that the update function is likely very simple — a few arithmetic operations — making the arithmetic to bandwidth ratio low. Despite the low locality, we can make data transfers efficient by doing block transfers (gathers) of partitions of the active set and the neighbors of that partition. Neighbors that are shared by partitions of the active set are placed in only one partition.

Load Balance: As above, this can be load balanced by distributing partitions of the active set to levels of the storage hierarchy. The load balancing needs to be done at the level of partitions to avoid destroying what little locality there is.

Chapter 5

Challenges in Expressing Parallelism and Locality in Extreme Scale Software

The focus of this chapter is on the challenges in expressing parallelism and locality that are encountered by application-level programmers across the three classes of Extreme Scale systems. The task of managing the parallelism and locality is relegated to the system software discussed in Chapter 6. It is likely that some heroic programmers, particularly for the data-center and embedded configurations, will wish to program directly at the system level using the interfaces in Chapter 6. However, for the remainder, it will be critical to address the challenges outlined in this section to enable them to use the capabilities of Extreme Scale systems. The capabilities described in this chapter are intended to address the needs of Applications (Chapter 4), serve as a portable interface to Runtime Management of Locality and Parallelism (Chapter 6), and provide information to Extreme Scale Tools (Chapter 7).

5.1 Application Programming for Extreme Scale Require Fundamental Breakthroughs

As outlined in Chapter 4, if applications are to be able to tap the power of future extreme-scale systems, they must exploit parallelism at multiple scales, at fine granularity, and across a wide variety of irregular program structures, data structures, program inputs, and in widely varying dynamic resource environments. In short, there are fundamental challenges which amount to a *major crisis* in the underpinnings of software development for all high performance computing systems. Simply put, the existing programming approaches we have relied on to get to 100,000 fold parallelism in nascent petaflop computing systems of today will not take us to extreme scale. They are too rigid and labor-intensive, while also failing to expose sufficient parallelism and locality to enable scalability and portability to future extreme scale computing systems.

Simultaneously expressing all available application parallelism and locality is a significantly different task from writing traditional sequential programs. Sequential programs can be thought of as a single expression of operation order and implied data movement (and thereby realized locality), and traditional program optimizers and parallelizers required "proof" of effect equivalence to make limited changes around that sequential order for performance [37], parallelism [85, 90, 118, 145] or locality [85, 119, 144]. With expression of only a single path, the limits of analysis meant

that the computation expressions were over-constrained. While those programming systems could increase parallelism and performance moderately, they are incapable of taking sequential application programs and automatically creating the needed parallelism and locality for efficient extreme scale parallel execution. Expressing all available application parallelism and locality requires a much richer expression — of the lesser constraints required to realize the computation, and the rich structure of locality over which the underlying program implementation system can play to achieve low power and high parallelism — both being synonymous with performance in extreme scale computing.

We identify several critical challenges to enable applications to exploit parallelism at multiple scales, at fine granularity, across a wide variety of irregular program structures, data structures, program inputs, and in widely varying dynamic resource environments:

- Billion-fold *parallelism* is required to tap the performance of extreme scale machines; achieving this requires flexible exploitation of regular and irregular parallelism across a range of scales from coarse to fine-grained.
- *Locality* is a critical requirement both to reduce energy per unit computation, and reduce latency (which in turn reduces the energy and complexity costs of managing many concurrent outstanding memory operations). This is a critical requirement (not an optional one) because the power requirements of extreme scale systems are directly tied to achievable performance [62].
- A third critical requirement is for the programming system to provide a simple *execution model* for the programmer to think about. As the scale and complexity of software to meet a variety of mission and commercial needs continues to expand in complexity, a simple execution model will reduce the application programming complexity required to achieve the goals of exposing all parallelism and locality. While the execution model may be defined at a high level of abstraction, the underlying programming system should enable programmers to provide non-binding guidance (hints or "default" control) when needed, thereby providing a level of *performance transparency*.

While we do not expect any one means of balancing these factors to be appropriate for all extreme scale applications, we believe that significant progress is required (and possible) in all three. Attacking all three simultaneously is the grand challenge of extreme scale programming. These challenges are demanding, but they must be addressed if extreme scale systems are to achieve their performance potential.

5.2 Portable Expression of Massive Parallelism

The requirement of pervasive extreme-scale parallelism outlined earlier demands that all of the intrinsic parallelism be exposed at all levels in the application. This is a marked contrast to current practice where programmers repeatedly rewrite applications to expose incrementally more parallelism for the next generation of hardware. Instead, our goal should be to express all opportunities for parallelism, leaving the choice of what to exploit to the layers of the software stack responsible for managing parallelism and locality (Chapter 6). While this is likely to be a more demanding task for current programmers who have been trained in sequential programming, it is expected that expression of parallelism and locality will be simplified in future programming models that *break sequential habits of thought* [132]. In this section, we briefly summarize some of the key points made in [132] and related work.

First, a major focus in writing efficient sequential code is to *minimize the total number of operations*. In contrast, efficient parallel code needs to focus on maximizing parallelism *i.e., minimizing the number of operations on the critical path*. With modern memory hierarchies, both sequential and parallel code must also focus on *improved locality*, but parallel code offers more opportunities than sequential code to (say) perform redundant operations to reduce communication. As mentioned earlier, locality optimization will have a first-order impact on energy reduction for future Extreme Scale systems. Second, good sequential algorithms attempt to minimize space usage and often include clever tricks to reuse storage; however, parallel algorithms need to use extra space to permit temporal decoupling and to achieve larger scales of parallelism. Finally, sequential idioms often stress linear problem decomposition through sequential iteration and linear induction. On the other hand, good parallel code usually requires multi-way problem decomposition and multi-way aggregation of results. A simple example is the difference between specifying a summation as a sequential iteration vs. a Fortran 90 SUM intrinsic for arrays.

A fundamental issue in the portable expression of parallelism is the need for data structures that lend themselves naturally to *data-parallel* operations. Fortunately, the *array* or *vector* data structure, which is a cornerstone of traditional HPC applications, is very well suited to data parallelism as evidenced by programming languages such as APL [82], Fortran 90 [102], NESL [42] and Ct [67]. These languages are able to express both flat data parallelism on vectors (element-wise operations, reductions, constrained permutations) and nested data parallelism on sparse or indexed vectors. *Streams* represent another data structure that is well suited for parallelism, as exemplified in data flow languages and programming models such as Sisal [100], Synchronous Data Flow [93], Brook [45] and StreamIt [69]. However, graph and other pointer-based data structures necessary for new Extreme Scale applications pose additional challenges for expression of parallelism and locality. The notion of *abstract collections* in modern object-oriented languages can help bring some of the benefits of data parallelism from arrays and streams to pointer-based data structures. In addition, *asynchronous dynamic parallelism*, as embodied in languages such as Cilk [43], Chapel [53], Fortress [38], and X10 [50], is necessary for operating on irregular data structures. Compared to current approaches, a key challenge for Extreme Scale is the ability to express this parallelism at the finest granularity possible, while delegating to the implementation the choice of what parallelism to exploit in a locality-sensitive manner.

In summary, a program that is organized according to sequential thinking and linear problem decomposition principles will be very hard to parallelize, whether by manual or automatic means. On the other hand, a program organized according to parallel problem decomposition principles should be easily run either in parallel or sequentially, according to available resources. At first, the costs and overheads for the "intrinsically parallel" approach may be daunting, but we have no choice than to overcome these challenges so as to enable software to use future Extreme Scale systems. Along with advancing foundational technologies for intrinsic parallelism and locality, we will need to also advance the pedagogy and curricula for software development. We will need to teach new strategies for problem decomposition ranging from data structure design to algorithmic organization that do not incorporate any inherent sequentiality. Approaches to multi-way problem decomposition may make the process of combining general sub-solutions harder than the sequential case, but this is our only hope for program portability in the future.

5.3 Portable Expression of Locality

The term "locality" refers to the logical or physical proximity of data to the units that perform computation on them. As mentioned earlier, locality optimization will have a first-order impact on

energy reduction for future Extreme Scale systems. Also, good sequential algorithms attempt to minimize space usage and often include clever tricks to reuse storage; however, parallel algorithms often require extra space to enable the temporal decoupling necessary for larger scales of parallelism. In Chapters 3 and 4, we also discussed the need for extra storage in weakly scaled parallel algorithms.

Programs must capture the opportunities for reuse and locality independent of machine structure or management policy. The compiler and runtime/OS can then decide how to best exploit the locality that has been exposed — fitting it to the structure of a target machine. Locality must include both *horizontal* (between processing elements) and *vertical* (between levels of the hierarchy) types.

5.3.1 A Simple Example

Consider the following simplified pseudocode fragment from a fluid dynamics application:

```
for t in timesteps {
  forall cell in cells {
    forall neighbor in cell.neighbors {
      compute_flux(cell, neighbor) ; // uses old value of pressure
    }
  }
  forall cell in cells {
    compute_pressure(cell) ; // uses both x-flux and y-flux
  }
}
```

Here `cells` is a possibly irregular collection of cells that is large enough so that it fits only in the aggregate main memory of a large machine. The computation iterates over the collection twice. On the first pass, it computes the fluxes through each face of a cell as a function of its pressure and that of its neighbors. On the second pass, it computes the pressure of each cell as a function of these fluxes. Each cell structure holds a collection of pointers to the cell's neighbors, the fluxes through each face of the cell, and the pressure within the cell.

The locality of this program is expressed by the neighbor relationship among cells, which also captures the dependences in the computation. The neighbor graph is data dependent, and may be time varying. Ideally we would like to exploit this locality by capturing both the reuse of shared neighbors and the producer-consumer locality between flux and pressure. Both of these forms of locality can be exploited horizontally and vertically.

While the neighbor relationship captures the locality of this program, it is easier to exploit this locality if it is expressed by partitioning the cell collection into sub-collections of arbitrary size in a manner that minimizes the number of external neighbor links from each sub-collection. Consider the following pseudocode:

```
partition(N, cells, parts) {
forall part in parts {
  forall cell in part {
    forall neighbor in cell.neighbors {
      compute_flux(cell, neighbor) ; // uses old value of pressure
    }
  }
}
```

```
forall part in parts {
  forall cell in part {
    compute_pressure(cell) ; // uses both x-flux and y-flux
  }
}
```

Here we assume an application-specific function decomposes the cell collection into a partition called `parts`. Each `part` can then be mapped to a given *node* to exploit horizontal locality or to a level of the memory hierarchy to exploit vertical locality. Multiple levels of vertical locality can be expressed in this manner by recursively partitioning a collection. In a style motivated by Sequoia [64] and by Hierarchical Place Trees [147], we can write:

```
void compute_cells(cells, level) {
  if(is_leaf(level)) {
    forall cell in cells {
      // do the computation
    }
  } else {
    partition(N[level], cells, parts) ;
    forall part in parts {
      compute_cells(part, level-1) ;
    }
  }
}
```

To capture producer-consumer locality, we need to fuse the two forall nests as shown below:

```
partition(N, cells, parts) {
forall part in parts {
  forall cell in part {
    forall neighbor in cell.neighbors {
      compute_flux(cell, neighbor) ; // uses old value of pressure
    }
    compute_new_pressure(cell) ; // but don't update old pressure yet
  }
}
make_new_current() ; // now make new pressure visible
```

The programmer here is expressing the producer/consumer locality between fluxes and pressures by bringing the two computations closer together. Now the fluxes are used (consumed) as soon as they are generated (produced). Hence the flux values can be captured in the smallest level of the storage hierarchy and never have to be written out to main memory. This transformation also improves reuse as it avoids having to read the cell back in to compute its pressure. To avoid creating a read-after-write (RAW) hazard, however, the update of pressure needs to be synchronized so that all flux computations in the current timestep see the old pressure value. While in theory such transformations could be automatically discovered, it is much easier if this locality is explicitly expressed, rather than leaving it to be discovered.

5.3.2 Parameterized Decomposition

As shown in the example above, the key to expressing locality in a portable manner is a parameterized decomposition of a dataset. The `partition` function above expresses what the programming system needs to know about the collection to exploit both horizontal and vertical locality. Specifying the partition function is a key aspect of expressing locality.

The partition function is application specific. For dense matrices, partitioning is just a block decomposition. For graphs, the structure of the problem may be exploited to generate good partitions inexpensively. For example, in a 3D code, the physical partitioning of space into contiguous 3D regions may be used to generate good partitions.

The temporal nature of the partition is also application specific. For some codes the partition is data independent. For others it is data dependent but static *i.e.,* the partition must be computed when the data set is loaded, but then remains constant for the remainder of the program execution. In other cases, the partition is time varying — the partition changes (perhaps incrementally) each timestep.

Expressing locality also requires that the dependence graph of the application be easily determined from the source code. In the example above, the dependence information is easy to determine — cell fluxes depend on cell and neighbor pressures and cell pressure depends on cell fluxes. For applications with significant indirection, such dataflow can be harder to determine making it more challenging for a compiler and run-time system to discover locality. Programs must capture the opportunities for re-use independent of machine structure or management policy, let the runtime/OS exploit this as it can.

5.4 Portable Expression of Synchronization with Dynamic Parallelism

Writing programs using today's state-of-the-art synchronization primitives is akin to using assembly language for programming. All the burden of performance, scalability and correctness falls squarely on the shoulders of the programmer with minimal support from the programming languages, development tools, runtime systems or hardware. In order to make parallel programs robust, portable, and scalable and reduce the burden on the programmers, many innovations are required in synchronization and communication. As an example, the *phasers* construct [124, 125] extends X10's clocks [50] so as to integrate collective and point-to-point synchronization with fine-grained dynamic parallelism, while providing a *next* statement that guarantees that all synchronizations will be performed in a deadlock-free manner. Each fine-grained task has the option of registering with a phaser in *signal-only/wait-only* mode for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. Support for dynamic parallelism dictates that it should be possible for new tasks to be dynamically added and dropped from phaser registrations, which creates a potential challenge to avoid race conditions between synchronization operations and registration add/drop requests. The fine-grain synchronization that accompanies fine-grain parallelism also presents the challenge of *phaser contraction* [126] to reduce the synchronization overhead when reducing the actual parallelism that is exploited on a given system.

One of the biggest challenges with synchronization for a programmer is the difficulty in avoiding deadlock and data races, both of which can appear non-deterministically in current programming models. Of the two, data race avoidance is more challenging than deadlock avoidance, since deadlock freedom can be enforced by well-defined programming practices and the use of deadlock-free programming constructs such as transactions and phasers. Removing non-determinism from

the programming model (as in declarative and functional programming approaches) can greatly simplify the testing and debugging of parallel programs, but the key challenge there is to ensure that the resulting model is sufficiently expressive for Extreme Scale software while still being efficient enough for execution on Extreme Scale hardware.

5.5 Support for Composable and Scalable Parallel Programs with Algorithmic Choice

Sequential idioms often stress linear problem decomposition through sequential iteration and linear induction. On the other hand, good parallel code usually requires multi-way problem decomposition and multi-way aggregation of results. A simple example is the difference between specifying a summation as a sequential iteration vs. a Fortran 90 SUM intrinsic for arrays. Extreme scale software will tax our ability to build programs — due to the complexity of application logic, sophisticated irregular algorithms, and program structure required. To manage these complexities, it must be possible to tap the full collection of higher level programming tools available to modern software engineering. In particular, in extreme scale programming systems, the expression of parallelism, concurrency and synchronization, and locality must interact gracefully with modern software engineering tools. The importance of composition was discussed earlier in Section 4.8. Ideally, concurrency, synchronization, and locality would be expressed and managed at the level of programmer meaningful abstract entities — objects or data abstractions — not individual bytes or words or values. (Of course, the system level interfaces outlined in Chapter 6 may well operate on lower-level machine-specific primitives and datatypes.)

To reduce the development cost of extreme scale software, it is imperative to have programs that can seamlessly scale across embedded, departmental and data center-sized extreme scale systems. However, it is often impossible to obtain a one-size-fits-all solution for high performance algorithms that will work effectively in both ends of the scale. Often the best algorithm for an architecture is tightly coupled to differences in parallelism, communication, and available system resources. Different algorithms are effective at different sets of architectural parameters. Current compiler and programming languages are unable to handle algorithmic choice. Thus, it is important to have programming languages and compilers that also support algorithmic choice. These systems should let the programmers express different algorithms to solve a problem and have the compiler and runtime system automatically identify the best algorithm for the given deployment.

5.6 Managing Heterogeneity in a Portable Manner

With the advent and increasing popularity of hybrid architectures, programming systems face the challenge of how to efficiently exploit multiple levels of parallelism, often coupled with different memory systems, instruction sets, or even numerics. In current-day systems, such as the LANL Roadrunner system [86], there may be as many as three distinct types of processors with distinct memory, messaging, and performance characteristics. These elements of heterogeneity are managed explicitly by the application programmers — through the use of coroutine-style models (one for each type of heterogeneity), explicit message passing for data movement, and distinct address spaces. Other examples of heterogeneous systems might include instruction set and performance heterogeneity or simply differences in memory structure such as cache coherent shared-memory, partitioned global address space, or shared-nothing. Unfortunately, if such characteristics of hardware heterogeneity are explicitly addressed by the programmer, not only is the programming effort

increased, it is likely that the software will not be functionally portable, much less performance portable to other systems.

Extreme Scale systems of the future may have both *designed heterogeneity* (in dimensions such as architecture, organization, instruction set that are exhibited today in hybrid systems), as well as *intrinsic heterogeneity* that arises from manufacturing variability, configuration, or aging differences. It is critical the software built for such large-scale parallel systems address the heterogeneity of the system in a fashion that supports portability of the applications. That is, it should be possible to move applications from one machine to another — with different heterogeneous characteristics — without significant change at the application source code level. This imposes major challenges in expression of parallelism, locality, and computation so as to both enable the compiler and runtime to deliver performance on one Extreme Scale system, but also in a form portable and flexible enough that it can enable the compiler and runtime to deliver performance on other heterogeneous Extreme Scale systems. This is a daunting challenge, but is in our view a fundamental requirement for a technology landscape that supports Extreme Scale computing.

Chapter 6

Challenges in Managing Parallelism and Locality in Extreme Scale Software

Earlier in this report, we summarized the hardware characteristics of future Extreme Scale systems (Chapter 2) as well as the challenges involved in developing applications (Chapter 4) and expressing parallelism and locality (Chapter 5) for such systems. In this chapter, we focus on the challenges and implications in *software management of parallelism and locality* for Extreme Scale *i.e.,* in bridging between high-level application frameworks and programming models and the realities of Extreme Scale hardware. Current software for high-end data-center, departmental and embedded systems build on a *classical software stack* which primarily consists of operating systems, parallel runtimes, static compilers, and libraries. Different embodiments of the classical software stack have been used in the past for data-center-class capability systems, departmental systems, and embedded systems with thinner and more restricted stacks used at the two extreme ends, and a richer software stack used for departmental systems in the middle. However, as described in the following sections, the general structure of the classical software stack has remained largely unchanged for decades, and will be highly mismatched to the requirements of all three classes of future Extreme Scale systems.

6.1 Operating System Challenges

6.1.1 Introduction

Extreme Scale processors containing hundreds or even thousands of cores will challenge current operating system (OS) practices. Many of the fundamental assumptions that underlie current OS technology are based on design assumptions that are no longer valid for a Extreme Scale processor containing thousands of cores. In the context of Exascale system requirements, as machines grow in scale and complexity, techniques to make the most effective use of network, memory, processor, and energy resources are becoming increasingly important. In its role as gate-keeper to all these resources, the OS becomes a major obstacle in allowing the application to view the hardware in accordance with the Extreme Scale Execution Model outlined in Section 3.1. A baseline challenge for the exascale software stack is: how to reduce OS overheads without compromising the need to protect hardware state from errant or malicious software.

Execution models that support more asynchrony will be necessary to hide *latency*. Such execution models will also require more carefully coordinated scheduling to balance resource utilization

and minimize work *starvation* or resource *contention*. These execution models will also require extraordinarily *low-overhead*, fine-grained messaging. However, the attributes required by the execution model are nearly impossible to achieve when the OS intervenes for every operation that touches its privileged domain *e.g.,* for exclusive and privileged control of scheduling policy, for exclusive ownership of resource management policies, and for inter-processor communication operations.

6.1.2 What is wrong with current Operating Systems?

Over time, operating systems have evolved into multifaceted and hugely complex software implementations that have accreted a broad range of capabilities. We refer to the challenge of breaking the OS apart based on separation of concerns as "deconstructing the OS". Below are a subset of leading issues that motivate the need to reexamine the underlying assumptions that are encoded in current OS implementations. These issues are discussed from the viewpoint of all three classes of Extreme Class systems, not just the data center class for which specialized solutions have been developed in past work on developing lightweight kernels for supercomputers.

6.1.2.1 Time Sharing

Time-sharing OS's are built around the assumption that the CPU is a precious resource that must be shared. This is no longer true for a CMP (Chip level Multi-Processor) containing thousands of cores and constrained memory bandwidth.

- Old Conventional Wisdom (CW): When CPUs are considered the most precious resource, time-multiplexing is performed to share access. However, when hundreds of CPUs are available, it no longer makes sense to suffer the overheads incurred by context switching. Indeed, the cost of a context switch is not merely the time spent preserving registers because the associated cache pollution (and other shared resources) can substantially reduce CPU throughput. Context switching makes inefficient use of the new precious resources, which are energy, on-chip memory and off-chip bandwidth.
- New CW: If cores are cheap, then allocation of cores should be spatially partitioned for function rather than offering time slices of a single resource. This spatial partitioning is analogous to Logical Partitioning (LPAR) from 1970's era mainframe terminology.
- Research examples: MITOSYS (MIT/Berkeley), K42.

6.1.2.2 Inter-Processor Communication

All device interfaces are at the OS's privilege level in a typical system of today. Therefore, any access to virtualized device interfaces is mediated by the OS in order to protect the hardware interface. However, the overhead of the privilege change and additional data buffer copies required for OS mediation has given way to a wide variety of "OS bypass" and "user space messaging" implementations such as VIA, PORTALS, and DRI. However, these approaches expose the hardware to irrecoverable state corruption by errant software. Whereas current OS works in-band to protect the hardware state, it would be more efficient to employ extra cores or some other supervisory hardware to monitor application-to-device transactions out-of-band to check for state corruption without adding additional overhead to the communication stream.

- Old CW: invoke OS for *any* interprocessor communication or scheduling to protect hardware state from corruption.

- New CW: direct HW access allowed by application, but hypervisor monitors for state corruption out-of-band from the transactions (perhaps using a dedicated subset of cores as observers).
- Research examples: Singularity, MVIA/MVAPICH. that is isolated to OS bypass for MPI)

6.1.2.3 Interrupts and Asynchronous I/O

Background task handling for asynchronous operations are currently handled by threads and signals in modern OS's and application designs. In particular, devices must interrupt the CPU in order to invoke the device driver software to service their requests.

- Old CW: Interrupts and threads (a side-effect of time-multiplexing access to a core) are used to implement asynchronous operations or system calls in user codes. Even more fundamentally, device handling is implemented by interrupting the CPU for the time-critical top-half of the device driver and then scheduling the bottom half of the driver to run on the next available time-slice of the CPU. The interrupt subjects processes to non-deterministic delays that can result in load-imbalances for parallel applications.
- New CW: If CPUs are abundant, side-cores can be dedicated to asynchronous I/O and device handling. For that matter, cores can be used to implement programmable DMA and asynchronous I/O instead of using dedicated hardware components. Using a core or background thread to implement DMA eliminates the need to write to the device control interface for a dedicated DMA engine, which require costly msync() operations by the requesting CPU.

  In this way, finer-grained spatial deconstruction can be achieved with legacy device drivers wrapped and isolated on separate cores, interrupts delivered automatically to free cores, and traditional facilities (*e.g.,* file systems) handled as servers on separate cores. In addition, resource allocation and Quality of Service (QoS) guarantees for network and memory bandwidth and fair access to I/O modules can be obtained by using hardware mechanisms for QoS as well as software-based policy implementations. The complexity added, however, is in duplicating management information of virtual memory and other resources.

6.1.2.4 Device Drivers and Virtualization

OS's play an important role in virtualizing finite hardware device interfaces — making it appear to each application that it is the exclusive owner of a replicated copy of the device interface. In the parallel context, current OS's assume a workload of uncoordinated processes that stochastically vie for control of finite/virtualized devices.

- Old CW: OS's currently implement a greedy allocation policy where the first process acquires a lock to gain exclusive access to a device (or OS/driver interface) for each I/O transaction. This approach is sensible for stochastic access to the virtualized resource, but very bad for highly-synchronous parallel algorithms. Resource and lock contention hurts performance by flooding the inter-processor communication network with redundant/spinning lock acquisition requests. Locks also serialize otherwise parallel processes when they attempt to access the same resource (such as the network interface), and subjects them to nondeterministic delays.
- New CW: A new OS will need to use a QoS management for symmetric device access by a large number of entities, or hand over coordinated scheduling of device access to the application. The development of novel mechanisms for coordinating parallel access to finite number of virtualized device interfaces is essential.

- Examples of Research: NOOKs, K42.

6.1.2.5 Fault Isolation

Another important role of operating systems on more robust systems is fault isolation. This is particularly difficult for large SMP systems as errors can propagate rapidly through the system and are extremely difficult to track down. With growing node concurrency, and increasing likelihood of soft errors, the ability to rapidly identify and isolate errors will be essential. The total lack of a fault isolation strategy in modern OS's is arguably one of their weakest points moving forward.

- Old Conventional Wisdom (CW): CPU failure on an SMP node will result in a "kernel panic" that takes down the system. Such events will happen with increasing frequency in future silicon.
- New CW: CPU failure should result in isolation of the partition containing the failure. It would be better yet if the hardware supported integrated rollback mechanisms to support partition Restart. There is existing work in the Singularity OS to consider how transactions between the application and device interfaces can be rolled-back to a known state to support restart of partitions containing the application.
- Research Examples: VMM containers, Singularity.

6.1.3 Parallelism Scalability Challenges in Operating Systems

its role as the gate-keeper to shared resources, operating systems have traditionally been a major bottleneck in achieving scalability on SMP's. This is especially true for the open source Linux operating system, which has historically lagged behind commercial Unix OS's such as AIX and Solaris in scalability but has now become the dominant OS of choice for high-end systems. Significant attention has been devoted by the Linux community over multiple years to bridge the scalability gap with commercial OS's, starting with efforts such as improvements to the Linux scheduler in 2001 [88]. More recent examples of scalability efforts explored and undertaken by the Linux community include large-page support, NUMA support [95], and the Read-Copy Update (RCU) API [101]. While these Linux enhancements have resulted in improvements for commercial workloads with independent requests and flow-level parallelism [140] on small-scale SMP's, the scalability requirements for even a single socket of an Extreme Scale system will be two orders of magnitude higher than what can be supported by Linux today. It is clear that this gap cannot be bridged by business-as-usual efforts; in fact, future scalability improvements in Linux are expected to be harder rather than easier to achieve, as evidenced by the RCU experience [101] and the complexities uncovered by ongoing efforts to reduce the scope of the Linux Big Kernel Lock (BKL) *e.g.,* see [114].

High-end systems typically use specialized operating systems for compute and I/O nodes, and standard operating systems for service, front-end, and file-server nodes. In the case of Blue Gene/L [105], the specialized OS operates at the level of a *processing set* (pset) which consists of one I/O node and a collection of compute nodes. The design of the Compute Node Kernel (CNK) was simplified by placing a number of restrictions on the application *e.g.,* single thread per processor, absence of virtual paging, and support for only 68 system calls in Linux. While this design approach was necessary to support the schedule and system requirements of the early Blue Gene/L systems, it will not be practical for Extreme Scale systems with a thousand cores per chip or for the dynamic, asynchronous, and irregular parallelism structures expected in future grand challenge applications for Extreme Scale systems (Chapter 4).

The scalability challenges in operating systems of course extends to parallel file systems as well. In recent years, a significant amount of research on parallel file systems has been reported, including Lustre [99], GPFS [70], PVFS [96], pNFS [80], PanFS [107] and others [41,52,108,112,127,143,151]. Different APIs to interface with files such as MPI-IO [106], HDF5 [77] and NetCDF [115] have gained popularity as an alternative to the basic POSIX API. Object-Based Storage [133] provides a different way to organize data and metadata on the storage medium than file or block methods. Much of the increased functionality of these parallel file systems comes at the cost of increased complexity and overhead of the file system software. Some recent work seeks to reduce overhead of the file system and its load on the file servers. The Light Weight File System (LWFS) [111], a current project at Sandia National Laboratory, is a parallel file system that implements only essential functionality without any additional functionality that degrades performance and scalability. Implementations of additional features are moved into libraries and the application itself, allowing the application to be optimized to a right-weight solution.

6.2 Runtime Challenges

Runtime support for parallel programming requires key innovations in lightweight mechanisms for communication and memory hierarchy management, and user-controllable policies for managing the system resources. Expected contributions to this area of research include:

- Lightweight runtime mechanisms to exploit the novel features of interconnection networks, including topology queries, atomic operations, remote procedure invocation, fast one-sided transfer notification used in synchronization.
- Extensions of the execution models to handle fast and slow memory associated with a single thread, and demonstration of that model on a single-chip system with software-managed local memory that replaces or augments the traditional hardware-managed cache hierarchy.
- Runtime support to virtualize the set of processors through the use of multi-threading and dynamic task migration. Programming model extensions that allow for such virtualization when needed, without enforcing it for all applications.
- Runtime support for memory system virtualization, including object caching and migration. As with processor resources, the programming model will be extended to permit runtime-managed data placement in addition to the user-managed placement already available.
- Support for multiple runtime systems for different execution models and soft real-time applications.

6.2.1 Task Scheduling and Locality Runtime Challenges

Past task scheduling runtime systems have typically been optimized for dynamic parallelism that is oblivious of locality (*e.g.,* Cilk, OpenMP, Intel Thread Building Blocks) or for locality in the absence of dynamic parallelism (*e.g.,* MPI, UPC, CAF). As discussed in Chapter 5, a desirable characteristic for future programming models is that they express large amounts of concurrency with locality control so as to be "forward scalable" to future generations of parallel hardware. However, efficient locality-sensitive scheduling of $O(10^{11})$ lightweight tasks is a major research challenge.

6.2.2 Communication Runtime Challenges

High performance computing (HPC) systems implementing *fully-connected networks* (FCNs) such as fat-trees and crossbars have proven popular due to their excellent bisection bandwidth and ease of application mapping for arbitrary communication topologies. However, as extreme scale systems move towards millions (and even billions) of processors, FCNs quickly become infeasibly expensive. These trends have renewed interest in networks with a lower topological degree, such as mesh and torus interconnects (like those used in the IBM BlueGene and Cray XT series), whose costs rise linearly with system scale. Future systems will also need to take into account wiring complexity as they consider alternative low-degree interconnect topologies.

Indeed, the number of systems using lower degree interconnects such as the BG/L and Cray Torus interconnects has increased from 6 systems in the November 2004 list to 28 systems in the more recent Top500 list of June 2008. However, it is unclear what portion of scientific computations have communication patterns that can be efficiently embedded onto these types of networks without advanced runtime support to more efficiently map the communication graph onto the underlying hardware interconnection topology.

Figure 6.1 shows the communication patterns of a broad-variety of applications at a modest parallelism. Even at 256p concurrency, the analysis highlights the application's irregular communication patterns, evincing the limitations of 3D mesh interconnects [123]. At the same time, most communication patterns are sparse, revealing that the large bandwidth of a FCN is not necessary and have good locality, showing that an intelligent task-to-processor assignment can significantly decrease the load on the network. It is clear that either the interconnect will need to adapt to the diverse communication requirements of the applications, or there needs to be a system to implement static task placement or runtime migration of tasks to better map the communication topology onto the underlying topology of the hardware. The runtime system will play a crucial role in either case.

Current programming practice presumes an entirely flat model for communication locality, where every processor is equidistant to its peers. Although topological hints exist in MPI, they are rarely, if ever used. Most PGAS programming models only express two-levels of locality — local and remote. HPCS languages such as Chapel and X10 attempt to mitigate this by allowing the programmer to express locality using "locales" and "places". However, optimized mapping of places onto hardware elements by the runtime system is still a major open problem.

There must be substantial changes in software practice to better expose communication requirements. If explicit task placement is left to the application developers, then performance portability may be brittle. This further supports the idea that the runtime system will need to play a more important role in task placement and interconnect configuration in future systems.

6.2.3 Synchronization Runtime Challenges

Applications that need to exploit high levels of hardware parallelism are usually expressed in terms of deep software parallelism and perform a significant amount of synchronization operations at various levels of the system hierarchy. Fast synchronization primitives might define several areas of research.

- Intra-node synchronization and notification: Synchronization primitives are usually implemented using either signals/interrupts or polling. Polling is the faster technique since it does not require any context switches but it makes implementations that have to deal with unexpected events cumbersome. One generic question that needs to be addressed in any implementation is the servicing of notification events. In some cases, these notification events initiate complicated execution paths that consume processor resources. As an example consider the

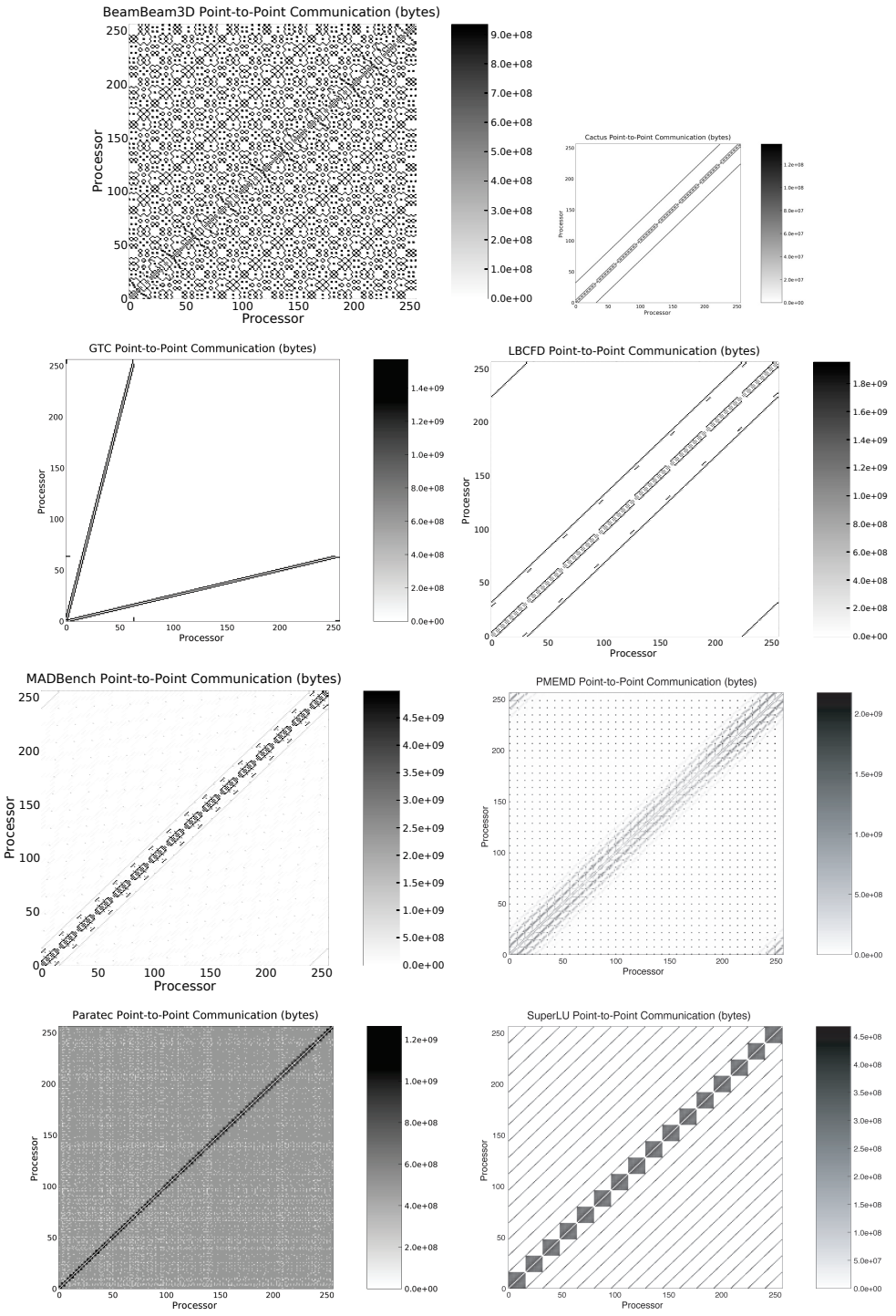

Figure 6.1: Topological connectivity of a broad range of scientific computing applications, showing volume of communication at P=256.

asynchronous remote execution functionality proposed in the DARPA HPCS languages. In the ideal case, execution of unexpected events is dispatched to the "execution unit" that already has possession of the required resources. Examples are requests for data that already resides in a processors cache or requests for execution of services that a processor has just served. One research direction will be interrupt dispatch based on resource requirements.

- Remote synchronization: The networking layer for advanced execution models supports several primitives for remote synchronization. Depending on the hardware target, these primitives are implemented using either active messages [142] or specific system features (such as offload). We envision a highly concurrent and asynchronous execution environment where events are generated and dispatched to various system components with various resource allocations and privileges. In this setting, asynchronous events are ideally dispatched to the entity that "expects" them and also has the available resources. The necessity for a distinction between resource ownership and availability to serve an event is illustrated by networking layer implementations of data packing/unpacking communication primitives using AMs. Whenever an active message requesting data packing arrives at a node, it should ideally be served at the processor thats most likely to have the data in caches. However, if that processor is busy, the request should be served by a different core.

Several requirements become apparent from the previous discussion. Events have to express their resource requirements, execution entities (threads) have to be paired with resource usage (*e.g.,* memory footprint) and schedulers/dispatchers have to be able to access this information. Meeting these requirements translates into research into performance instrumentation and techniques to extract and describe application behavior.

6.2.3.1 Thread and Resource Virtualization

Current threading packages, such as Pthreads, offer very limited control over scheduling decisions. Ideally, information about resource utilization is associated with threads and schedulers take this information into account. Furthermore, experience indicates that cooperative threading rather than preemption is able to increase performance in a HPC environment. This unfolds into several research directions:

- Develop mechanisms to extract resource usage (memory, functional units) for parallel applications, using both dynamic (instrumentation and runtime monitoring) and static (program analysis) approaches.
- Develop mechanism to extract the current state of execution: precise point in the execution thread and expectation of near future action.
- Develop scheduling mechanisms and policies based on resource usage, priority and data dependence information.
- Develop mechanisms and policies to avoid deadlock in a cooperative scheduling environment.
- Develop frameworks to expose this functionality to applications/libraries.
- Develop mechanisms to ensure progress and attentiveness in the presence of asynchronous remote execution and Active Messages based implementations.

6.3 Compiler Challenges

The crucial role of compilers at the extreme scale is to map from language constructs that express a very high-level decomposition of an application to highly power-efficient and memory-efficient architecture-specific code and runtime layer calls. As has been proven historically, completely automatic compiler optimization from high level code will not meet the performance requirements at the extreme scale; in the extreme scale regime, we will also encounter memory and power constraints that programmers and tools could previously ignore. Further, compiler-based approaches, such as, for example, compilers for PGAS languages, have generally focused on regular, static parallelism. As the applications for extreme scale platforms expand to encompass irregular, unstructured and dynamic algorithms, so must the compiler technologies that support these challenging application domains. An article reporting on a recent NSF-sponsored workshop on the future of compiler research listed the following 6 research challenges, in addition to other guidelines on enhancing research and enriching education [75].

Compiler research challenges in optimization include:

- Make parallel programming mainstream;
- Write compilers capable of self improvement (*i.e.,* auto-tuning); and
- Develop performance models to support optimizations for parallel code.

Compiler research challenges in correctness include:

- Enable development of software as reliable as an airplane;
- Enable system software that is secure at all levels; and
- Verify the entire software stack.

Compilers at the extreme scale must collaborate closely with the application programmer to derive an architecture-independent algorithm description that can be mapped to high-quality code; further, the compiler must incorporate lightweight mechanisms that interface with the runtime layer and architecture to dynamically map this code for a specific execution context to be both high performing and power efficient.

6.3.1 Customized and Dynamic Code Generation

Generally speaking, the role of code generation must fundamentally change in response to the need for agile code mappings that respond dynamically to execution context, including input data set properties, machine load and power constraints. Rather than a single statically-generated implementation of a computation, the compiler must represent a space of possible implementations that are generated either a priori, by predicting relevant features of execution contexts, or dynamically, in response to execution context.

Some of the critical decisions made in the code generation process must be exposed to both programmers and the runtime layer, and the set of alternative implementations must be well-defined and systematic. Such requirements will make it possible to explain the code generation process to the application programmer, and to mechanically evaluate these alternatives, either off-line or dynamically during execution. An essential feature of code generation is that the mechanisms, either dynamic code generation, partial code generation that is instantiated at run time, or run-time selection of statically-generated code, be efficient in both execution time and memory requirements.

6.3.2 Extracting Useful Parallelism from Ideal Parallelism

At the language level, as discussed in the previous chapter, we want the application programmer to be able to express an abundance of parallelism, providing the underlying system architecture with sufficient degrees of freedom to derive an efficient mapping of the parallelism. We need to strike a delicate balance between runtime overhead for managing parallelism and the risk of idle resources or load imbalance from lower overhead static thread management.

Historically, compiler-managed parallelism is mostly static, and possibly explicitly declared by the programmer. In the extreme scale regime, the compiler should make some decisions on static management of parallelism for efficiency's sake, and defer other decisions for the runtime layer. Therefore, the virtualization of some but not all parallelism is desirable, and even for runtime decisions, the compiler should optimize for efficiency of dynamically-mapped code. The programming model can assist with how to decide what parallelism to bind statically, and what to defer to runtime, and the complexity of runtime decisions. For example, a hierarchical expression of parallelism would provide the compiler with useful guidance on how to do this mapping [116, 117].

6.3.3 Optimizations for Vertical and Horizontal Locality

A wealth of prior research on compiler optimizations to manage locality for uni-processors will provide an important foundation for both vertical and narrowly-defined horizontal locality within a chip. Dramatic reductions in memory per core and increased competition for off-chip memory bandwidth will make such optimizations truly essential, but the approaches taken must be modified to support parallel threads and in particular sharing of data across threads. Where caches are shared across cores, fine-grain scheduling of parallelism to exploit locality in aggregate caches will be needed. Further, on-chip non-uniform access time to caches (*i.e.,* NUCA architectures) may require careful placement of data even in caches.

Across processor chips and across the storage and processor hierarchy, careful data layout in memory will be required to optimize performance and manage power. While PGAS and HPCS languages offer some support for expressing data layout, as discussed in the previous chapter, new programming model constructs are needed to express hierarchy and manage locality in light of dynamic parallelism. Further, user-defined data layouts and libraries, working in conjunction with compiler-generated code, will be required for more irregular applications. The role of the compiler is to map high-level data layouts into levels of the memory hierarchy.

6.3.4 Synchronization and Communication Optimizations

Extreme Scale environments will demand advanced compiler analyses and optimizations that expand on current communication optimizations. Such optimizations must extend the scope of compilers to handle (or even specify) new runtime mechanisms for synchronization, caching, and other critical dynamic decisions based on user-space scheduling, memory management or communication.

6.3.5 Energy Optimizations

Compiler optimizations to reduce energy consumption have been a topic of research interest for over a decade, but have mostly been deployed only in embedded environments. In conventional architectures and high-end systems alike, energy is largely managed directly by hardware and low-level system software. It is still the case that most compiler optimization research has focused on minimizing execution time, without regard to power. For extreme scale architectures, the compiler's optimization objective function must consider both performance and power. Fortunately,

some optimizations can be good for both objectives, such as increasing processor utilization or improving data locality. However, improving performance and managing power might be at odds with each other in parallelization since a faster time to solution may use resources less efficiently in computations that do not exhibit perfect strong scaling. Therefore, situations arise in which the compiler must balance performance and power to obtain a solution that meets both sets of objectives, suggesting the need for incorporating estimates of power consumption and power budgets into the optimization process.

6.3.6 Support for Resilience

As discussed in [75], future requirements call for continuing the trend of the increasing role of compilers in detecting errors automatically or semi-automatically. Research must continue to be pursued in static and dynamic analyzes, in conjunction with language constructs, to detect programmer errors. Further, the compiler will be responsible for appropriate code generation mechanisms to detect hardware and system software errors and respond to faults.

6.3.7 Global Auto-tuning and Dynamic Optimizations

Compilers will play an important role in auto-tuning and dynamic optimizations, systematically applying the set of optimizations described in this section to empirically evaluate the best-performing solution. Auto-tuning could be used for managing both performance and power, for computation kernels and whole applications, and in both off-line and on-line settings. Because auto-tuning as a technique requires a number of technologies that are beyond the capabilities of a compiler such as techniques for navigating prohibitively large search spaces, library, run-time and application-level optimizations, further discussion of auto-tuning is deferred until Chapter 7.

Dynamic optimization strategies must be efficient enough to employ at run time, or must be used in conjunction with partial code generation and dynamic instantiation or a priori code generation and dynamic code selection. The overhead of dynamic optimization means that it must be applied at the appropriate computation granularity. If high speedup is feasible for a repeated computation, then the execution can also overlap optimization with execution of a previous time step and execute the newly optimized code when it becomes available.

6.4 Library Challenges

One of the many tenets of computer science is an aspiration to develop techniques to manage and reduce software complexity. Often the practical limitations on complex computer systems are human comprehension — not the physical computing and I/O capacity. This is especially true of high-end computing (HEC), with a heightened emphasis as we scale to petascale and on to exascale systems. Earlier in Section 4.8, we discussed the role of application frameworks in reducing software complexity. In this section, we discuss the role of *software libraries*.

Libraries have historically been an important means to managing the complexity of system software. In the HPC community, libraries can provide both productivity and performance. The domain-specific libraries discussed earlier in Section 4.8.5 make use of support libraries to provide optimized performance across a wide range of architectures – possibly ranging from workstations to Top10 systems. The relationships between domain-specific and support libraries are depicted in Figure 6.2. Note that support libraries may be architecture dependent with (1) numerical libraries optimized for specific microprocessors, (2) communications libraries optimized for specific system interprocessor communications characteristics, and (3) data management libraries and I/O

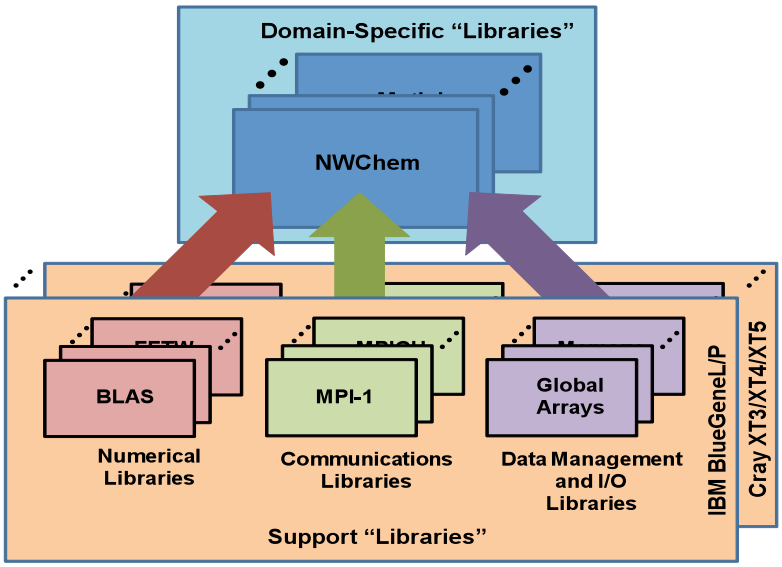

Figure 6.2: The Relationships between Domain-specific and Support Libraries

libraries optimized for specific shared memory and I/O system characteristics. It is important to note that support libraries — numerical, communications, and data management — can be used by any program and need not exclusively be used in a hierarchy below domain-specific libraries and frameworks.

6.4.1 Numerical Libraries

Numerical libraries have been available to support application developers for over 40 years. The International Mathematics and Statistics Library (IMSL) may be one of the oldest general numerical libraries, currently supporting a comprehensive set of 1000+ algorithms [16]. The Basic Linear Algebra Subprograms (BLAS) was first published in 1979 and is used to develop other numerical libraries such as LAPACK [17] [18] [19]. An important feature of numerical libraries is that they offer both portability and performance. While open source software is available to compile and link for any architecture, the highest performance most often comes from highly optimized libraries for specific processor/system architectures. To minimize the effort to reach high performance, an open source auto-tuning implementation of BLAS APIs was developed, notably ATLAS (the Automatically Tuned Linear Algebra Software) [20].

The algorithms in numerical libraries have evolved as a function of architecture over time. This is illustrated in figure 6.3. LINPACK in the 1970's was based on level-1 vector-vector operations, which were well matched with the vector architectures of the time. LAPACK in the 1980's needed to deal with caches, data movement, and locality and used data-reuse friendly methods that employed level-3 BLAS routines. With the advent of massively parallel processing with distributed memory in the 1990's, LAPACK evolved into ScaLAPACK based on PBLAS message passing. The most recent incarnation of numerical libraries must be able to work on new many/multi-core architectures where thread-level parallelism is as important as distributed memory parallelism.

The Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) project aims to address the critical and highly disruptive situation that has been a result of the introduction of many/multi-core processor architectures. Like the BLAS, LINPACK, LAPACK, and ScaLAPACK,

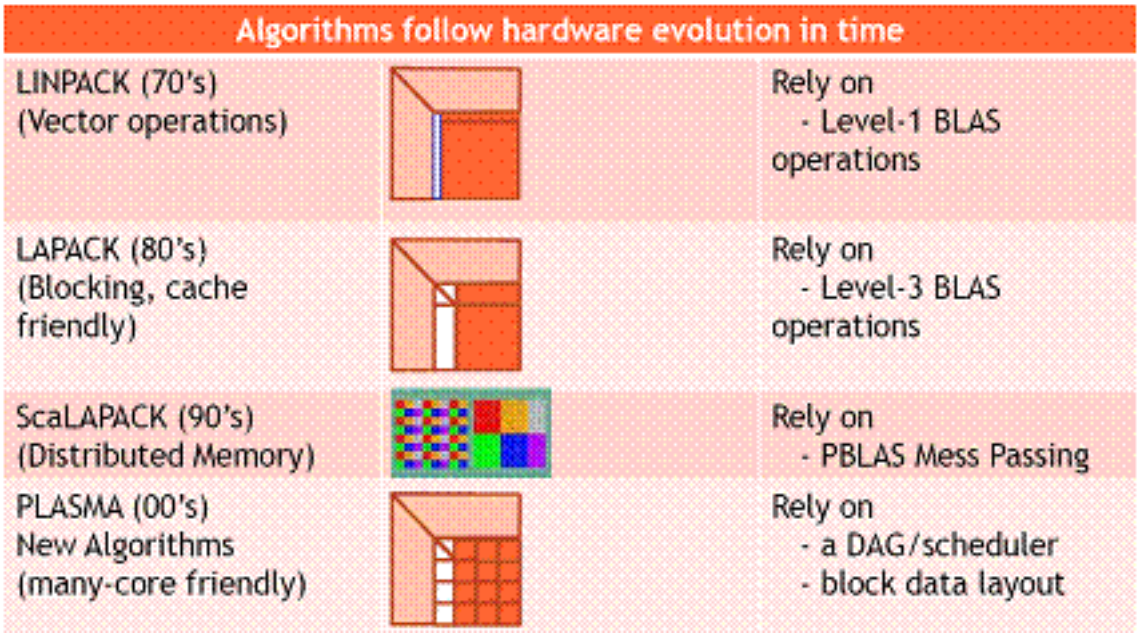

Figure 6.3: Figure Evolution of Optimized Algorithms in Numerical Libraries [55]

PLASMA's ultimate goal is to create software frameworks that enable programmers to simplify the process of developing applications that can achieve both high performance and portability across a range of new architectures. The new technologies employed in PLASMA are based on asynchronous, out of order scheduling of operations as the basis for the definition of a scalable yet highly efficient software framework for linear algebra applications [21]. Other open source numerical libraries include Fast Fourier transforms (FFTs) such as FFTW [22], FFTE [23], and Vector Signal Image Processing Libraries such as VSIPL/VSIPL++ [24].

Nearly all open source numerical libraries have been designed to deal with issues relating to portability and performance. Many numerical libraries offer simple portability with reference implementations that can be compiled and linked with larger applications. Many of the numerical libraries offer both portability and performance. Some of the aforementioned libraries have automatic tuning features *e.g.,* FFTW. Others have been optimized for particular architectures *e.g.,* FFTC, which has been optimized for the Cell BE processor architecture. Others rely on either hierarchically employing optimized vendor libraries or having vendors develop optimized implementations *e.g.,* VSIPL/VSIPL++.

While the HPC community makes extensive use of open source numerical libraries, there are optimized numerical libraries available provided by vendors *e.g.,* Intel and IBM both provide optimized numerical libraries for processors they have developed. Intel provides the Math Kernel Library v10.x (multi-threaded and thread-safe) [25]. IBM provides the Engineering Scientific Subroutine library (ESSL) (thread-safe) and the Parallel ESSL (based on MPI for distributed memory applications) [26]. These libraries are often available free for non-commercial use.

6.4.2 Communication Libraries

Communications libraries come in two basic varieties — general purpose and domain-specific — with the possibility that domain-specific communications libraries have been built upon general purpose

communications libraries. We traditionally think of "libraries" as using language subroutine or function calls, but here we permit the more general idea of an application programming interface (API). MPI (and its variants) is based on the subroutine/function model [27] In the past, there were application specific communications toolkits *e.g.,* TCGMSG that was provided with the Global Arrays support software that is part of NWChem [28]. Later versions were built upon MPI (*i.e.,* TCGMSG-MPI) but maintained the domain-specific flavor in the programming interface [29].

6.4.3 Data Management and I/O Libraries

The third support library category describes those libraries developed to support data management and I/O. There are general purpose libraries *e.g.,* MPI-I/O, which combines the portability and "look and feel" of MPI with performance and file interoperability [30]. Additionally, there are domain-specific libraries to simplify data management and I/O. For example, NWChem makes use of the following data management and I/O libraries:

- Global Arrays provides an efficient and portable "shared-memory" programming interface for distributed-memory computers [13].
- Dynamic Memory Allocator provides a dynamic memory allocator for use by C, Fortran, or mixed-language applications [31].
- Aggregate Remote Memory Copy (ARMCI) library provides a portable remote memory access (RMA) operations (one-sided communication) optimized for contiguous and noncontiguous (strided, scatter/gather, I/O vector) data transfers [32].
- ChemIO provides a standard I/O API that meets chemistry requirements while being portable and being highly efficient [33].

Chapter 7

Challenges in Supporting Extreme Scale Tools

Extreme Scale tools encompass the portion of the software environment that support human clients or other parts of the software stack. These tools must interact with producers and consumers of information in two directions, both to provide guidance and query for additional information.

- *Explaining application and system behavior:* Tools to collect and present information to the user or other parts of the software stack to help assist in the process of changing the application to improve its behavior.
- *Exploiting information beyond application code:* Tools to collect and utilize information from the user or other parts of the software stack to automatically improve the application.

We group all such tools collectively into *development environments*, which support the mapping of application code to specific architectures. Several broad categories of development environment capability include (1) *performance and power optimization tools*, which exploit architectural features and eliminate bottlenecks; (2) *correctness tools*, which pinpoint and eliminate errors and vulnerabilities; (3) *analysis of computation*, which provide analysis of application behavior, preferably through interactive visualization; (4) *application completion tools* to manage the low-level details of mapping an application to an architectural platform; and, (5) *compilers*, which provide support for all of the above, in addition to translation from the programming model. While more details on these classes of tools are presented in Appendix A.2, this chapter examines the technological challenges facing tool developers.

7.1 History of Tools and Development Environments

For well over twenty years, researchers and tool developers in academia and industry have been struggling to develop technologies and tools for tuning performance and debugging parallel systems and applications, with only partial success. The reasons for these struggles are manifold, but can be traced to a combination of technology, economics and human psychology.

Technologically, finding and re-mediating performance or correctness problems is difficult, given the complexity and scale of today's parallel systems. Not only do tens to hundreds of thousands of hardware components (processors, cache and memory systems, communication interfaces and interconnection networks, and storage systems) interact in oft-unexpected ways, their behavior is mediated by complex, multilevel system and application software hierarchies. Subtle interactions

among the components at any level, or even multiple levels, can adversely affect observed application performance. Each new, large-scale system exposes examples of such unexpected problems, from the deleterious, system-wide effects of a TLB replacement algorithm, to operating systems jitter induced by system daemons, to adaptive runtime systems that conflict with system policies. If history is any guide, such challenges will only grow with future extreme scale systems.

Despite, or perhaps given this complexity, software performance and debugging tools are rarely a priority during development for HPC system designers or vendors. Perhaps paradoxically, the shift to commodity-based HPC systems has exacerbated this difficulty, for software performance and debugging tools are one of the few aspects that are not readily extensible from the sequential domain. One can construct large-scale systems using entirely commodity components - processors, memory and storage systems, interconnect, operating system, libraries and compilers, yet the challenges of tuning and debugging one hundred thousand concurrent threads differ markedly from PC-based debugging and tuning.

Moreover, experience has shown that there is no economic market to stimulate development of parallel software tools. There is no thriving ISV market for HPC debuggers or performance tools, nor is development of such tools a priority for the large system vendors. The selection criteria for HPC system procurement overwhelmingly focus on other aspects — peak and sustained performance, reliability and power consumption — but less on human productivity and total time to solution. Government contracts mandate basic tools as part of most, if not all procurements, but this has done little to advance either the technical or commercial state of the art.

Finally, unlike other elements of the HPC software stack, software performance and debugging tools must combine both technology and usability. However arcane the compiler and code development interface, no current user would consider writing assembly code. In contrast, performance and debugging tools must be capable of capturing and presenting data on possible performance or correctness problems, and they must do so in ways that users find intuitive and helpful, else the users will eschew these tools in favor of more rudimentary alternatives such as manual instrumentation.

Given the technological complexity of tools, the realization that market forces seem unlikely to solve current problems, and the usability challenges inherent in tool development, several workshops and reports have recommended that we change our current research and development model. Simply put, our current tool approaches are not working and have not worked for the past twenty years. The remainder of this chapter examines how tool research can be integrated with research on extreme scale systems to navigate the complexities of $O(10^{11})$-way parallelism.

7.2 Overview of Extreme Scale Development Environment Challenges

Many of the most critical challenges facing tools and development environments are variations on challenges that have plagued the HPC community for 20 years, as discussed in Section 7.1. Progress in this area demands a very different approach to developing and deploying tools. Beyond existing challenges, an extreme scale regime will dramatically increase the complexity of developing, debugging, modifying, porting and validating future applications. Hundreds of thousands of threads executing in an increasingly dynamic environment is beyond the intellectual scope of even the most veteran HPC application developers. The application programmer must of necessity let go of total control of performance and focus on using more productive ways of expressing and mapping their applications. Further, managing power consumption will become a new priority for application programmers. This growth in complexity across a number of dimensions makes the use of tools and development environments *even more essential* in managing the workload of HPC developers

and broadening the pool of talent for developing applications at this scale. We now enumerate the essential properties of extreme scale tools and development environments, and then discuss specific details of how these properties impact the technological direction of the extreme scale software stack.

Performability. One immediate consequence of the dramatic growth in system scale is the increased importance of system reliability and the need for alternate approaches to system resilience. Today's HPC systems contain more addressable nodes than the entire Internet did just a few years ago, yet our modus operandi continues to presume systemic reliability. With millions to tens of millions of hardware components, component failures will be frequent events, yet they should not trigger system failure. This suggests that future extreme scale systems must adroitly support a continuum of operating modes, ranging from complete health (all components operating correctly) to substantially degraded operation (many failures). From the tools perspective, this notion of performability (*i.e.,* integrated performance and reliability) means combining multiple techniques for fault tolerance (*e.g.,* checkpointing, redundant computation, restart-retry) with real-time monitoring to detect aperiodic component failures.

Scalability. When debugging and tuning a parallel application on a small-scale SMP, one has the luxury to capture detailed data on the fine-grained interactions among threads and hardware components. At the exascale, such detailed examination is neither productive nor even possible. The volume of performance and debugging data and the more worrisome perturbations induced by its capture become unmanageable. This suggests that extreme scale performance analysis and debugging tools must combine dynamic instrumentation techniques (*i.e.,* those that can be enabled and disabled rapidly and on demand) with methods that exploit large scale as an advantage. Stratified population sampling, adaptive compression, temporal logic and classification mechanisms can be used to capture data and reason about behavioral equivalence classes. Only with such approaches can instrumentation infrastructure and overhead grow sublinearly with system size.

Abstraction. The usability counterpoint to scalability is abstraction — presenting system and application behavior in terms relevant to the system operator or application developer. In turn, this has implications for program transformations and compile-time and run-time optimizations, as crucial information must be preserved across transformation boundaries, else there will be inadequate data to relate measured behavior to application specifications. Moreover, if we are to broaden the base of HPC application developers, we must move to presentation metaphors that are deeply tied to the application model, not the hardware. Although extreme scale software developers, as with any apex system, will be willing to accept the need to understand some level of system detail, this will be unacceptable to users of departmental extreme scale systems. Knowledge of multicore chip structure and interconnects, transient bit errors, memory hierarchies and interconnect topologies should not be required to develop portable, reliable, efficient extreme scale applications. This is especially true when a level of hardware-software virtualization will likely be required to hide ongoing component failures.

Adaptation and Autotuning. As noted, the volume and complexity of performance and debugging data are likely to overwhelm even the most determined application developer, even when abstracted and presented in terms relevant to the application programming model. This suggests that manual optimization should be complemented by dynamic adaptation and autotuning. Such introspective runtimes would combine real-time measurement, via targeted sensors, decision procedures for intelligent policy configuration and selection and actuators for decision implementation. Implementation challenges include hysteresis (*i.e.,* the lag between change and its manifestation), oscillation (*i.e.,* avoiding repeated policy or configuration changes) and multilevel adaptation, where compiler-synthesized adaptation (*e.g.,* multiversion code generation), runtime library adaptation (*e.g.,* scheduling or code dispatching) and operating system management (*e.g.,* virtualization and

dynamic provisioning) interact appropriately and intelligently.

Multilevel Integration As noted earlier, one of the many challenges of complex HPC systems is the behavioral interdependence across hardware and software levels. As we raise the level of programming abstraction to increase human productivity and to hide the idiosyncrasies of specific hardware implementations, the semantic gap between user specifications and run-time behavior will continue to widen. Data parallel languages, functional specifications, domain-specific toolkits and libraries all hide system details, and their implementations depend on multilevel translation and mapping. Understanding the performance and assuring the reliability and correctness of applications written using these tools will only be possible if the hardware and software tool chain contains information sharing specifications and interfaces that can relate measured data to application specifications.

Availability and Portability. When tools are provided by vendors of specific platforms, application developers must either focus on using only a set of platforms sold by that vendor, or use different tools as they port from one platform to another. Neither of these strategies is reasonable, and undoubtedly has contributed to the general lack of adoption of tools as part of the application developer's workflow. An alternative and more desirable solution is the existence of robust tools that are available on every HPC platform, preferably open-source tools that can be adopted in academic environments and taught to the next generation of application developers. The motivation to develop and deploy such tools must be carefully examined, with appropriate incentives to guarantee a long-term evolution and maintenance of portable, robust and low-cost tool and development environments. Fundamentally, the path to robust, efficient and effective tools and programming environments demands that such software must be an integral part of an overall system design, and not as an afterthought when the system has already been developed.

7.3 Enabling Technologies for Exascale Tools

Given the previously-described requirements, as part of this study we have identified three key enabling technologies that will be essential for tools in exascale systems. The first of these is the ability to collect and analyze enormous volumes of data to improve an application's execution. The second considers the computation side of data collection and analysis: how to enlist additional computation to improve the execution of the main application. Finally, we discuss *autotuning*, a principled methodology for using additional computation to test out alternative mappings of a computation to find the best implementation.

7.3.1 Scalable Data Collection and Analysis

Data Collection. While almost every microprocessor provides performance counters to examine execution data, these counters fall short of the requirements for exascale tools. Fundamentally, hardware performance counters collect low-level information that is not directly meaningful to application programmers. Vendors find these counters useful in understanding performance bottlenecks on existing workloads, and improving upon functionality in future generations of devices. Deriving meaningful application performance data from such low-level counters requires the programmer or tool developer to interpret the meaning of the counters, multiplex the counters of interest since only a small number of events can be counted, modify the application to access the counters, and then interpret the results and decide how to modify the application. Further, there are gaps in what the counters are collecting, such as, for example, communication events from the interconnect.

Software layers can also inhibit data collection. The goal of exposing performance counter information in an architecture-independent way, while desirable, is somewhat elusive. Portability issues are a frequent challenge when counters on different platforms are not counting the same thing. Compilers also introduce a number of challenges in mapping collected data back to code structures in the application code. Following lowering from high-level constructs and optimization, the compiled code may look very different from the original application. Only with proper preservation of the original structure during the compilation process is it possible to do this mapping [135, 136]. Runtime layers and operating systems face similar challenges in mapping dynamically collected data back to application constructs. A further requirement for dynamically collecting data in software is its overhead, since a subset of data collection may need to occur during production executions. The importance of efficient data collection in exascale systems begs the question of whether the design of hardware counters, compilers, operating systems and development environments might be very different if the designers start with the goal of explaining application behavior.

Autonomous Data Analysis and Mining. The system must autonomously and efficiently collect data about what the program and system are doing, and perform on-the fly analysis of the result at scale. Due to the complexity of exascale application behavior, the system must find anomalies that are unanticipated, and therefore must collect data on routine and production runs, and not just when looking for a specific problem. Thus, efficient modes of collection and analysis become a high priority, as compared to detailed traces and post mortem analysis.

On-the-fly analysis algorithms must be developed that rapidly reason about behavior and compare it to expectations, perhaps with system support to maintain efficiency. What are the baselines or ground truth to which an execution can be compared? As an example, in monitoring SPMD programs, analysis could look for threads with the most deviant behavior. Other baselines could include user-specified expectations, models of expected behavior or results of prior execution. In support of automatic validation of applications, scalable techniques for off-line detection of common errors must be developed.

Implicit in this analysis is a set of important decisions about what information to collect, analyze and discard or retain. The retained data must be organized in a performance database that provides meaningful information to subsequent executions or phases in the current execution, but also provides efficient access for on-the-fly analysis. Further, it should be noted that while there may be some differences, much of the underlying support for identifying performance, power or correctness anomalies will be common.

7.3.2 Companion Computations

While some applications may achieve high utilization on Extreme Scale systems and remain compute-bound for their entire duration, many applications will likely be limited by other aspects of the system such as memory bandwidth for at least some part of their execution. For these applications, the unused hardware resources can be used to assist in improving critical application characteristics such as programmability and resiliency. The notion of utilizing otherwise wasted resources to improve execution is not a new idea, and has been proposed to exploit unused issue slots for ILP, in multithreaded architectures, and now for multiple cores. An available surplus of resources suggests mechanisms to support additional threads or processes not directly contributing to computation, which we call *companion computations*. A companion computation provides information about execution of the main process that can be used to analyze or improve performance, identify the health of a node to improve reliability, examine tolerances to ensure accuracy, increase throughput and interface with developer or other tools. Companion computations can be both sensors, detecting problems during execution, and actuators, modifying execution to improve its behavior.

The notion would be to allow the user/tool to start up a companion computation at the same time that the application is started. The companion computation would not only have its own processors and memory, it could read and write into the memory of the executing application. The companion computation would also have access to the interconnect and be able to communicate with other companion computations associated with the same application.

The operating system should permit and assist in co-location of application and companion threads. In addition to the reading and writing of the application's data, the companion computation must be able to multiplex hardware counters that are important to the determination of the bottlenecks in the application. Monitoring message traffic is also extremely important.

On the process side we have the ability of the companion process to capture important information about the execution of the application and at the user/tool end we have the interface to allow the user to direct the function of the companion process.

In designing mechanisms and roles for companion computations, we must adhere to the premise that companion computations should do no harm. If the purpose of companion computations is to gain useful system throughput, it is problematic if they are competing for resources with the main computation.

7.3.3 Autotuning

Autotuning is broadly used to describe application code, libraries, and compiler-generated tools that, either in an off-line or dynamic way, evaluate a set of alternative implementations. This evaluation usually involves executing code under representative execution contexts, to select the most appropriate for a specific hardware platform, input data set, and execution environment. As with companion computations, off-line autotuning is feasible in today's powerful systems, and on-line autotuning will become feasible in light of the vast resources available in exascale systems.

The popularity of autotuning arose in response to the complexity of modeling or predicting the impact of code changes on performance, particularly given the subtle interactions between hardware features. This complexity will grow in the exascale regime, but so will the availability of hardware resources to be used in the autotuning process. At exascale, on-line parallel search of alternative implementations becomes more feasible.

While auto-tuners have largely focused on improving performance, the general approach is well-suited for other optimization criteria, such as reducing power consumption, increasing throughput, or limiting the load on some overused resource such as interconnect. Support for auto-tuners might include appropriate hardware performance counters, parallel search algorithms and heuristics to prune the search space of potential implementations, programming model features to express both the space of possible implementations (as discussed in Section 5.5) and ways to prune this space, and compiler technology to map the set of implementations to executable code segments.

7.4 Scenarios for Interaction with Tools

This section describes scenarios for how tools may interact with the exascale software stack to manage the complexity of application development, execution and understanding/modifying tasks. In each scenario, we present a motivation for why new approaches will be needed in an exascale regime, and connect the scenarios to the six requirements in Section 7.2 and the three enabling technologies in Section 7.3.

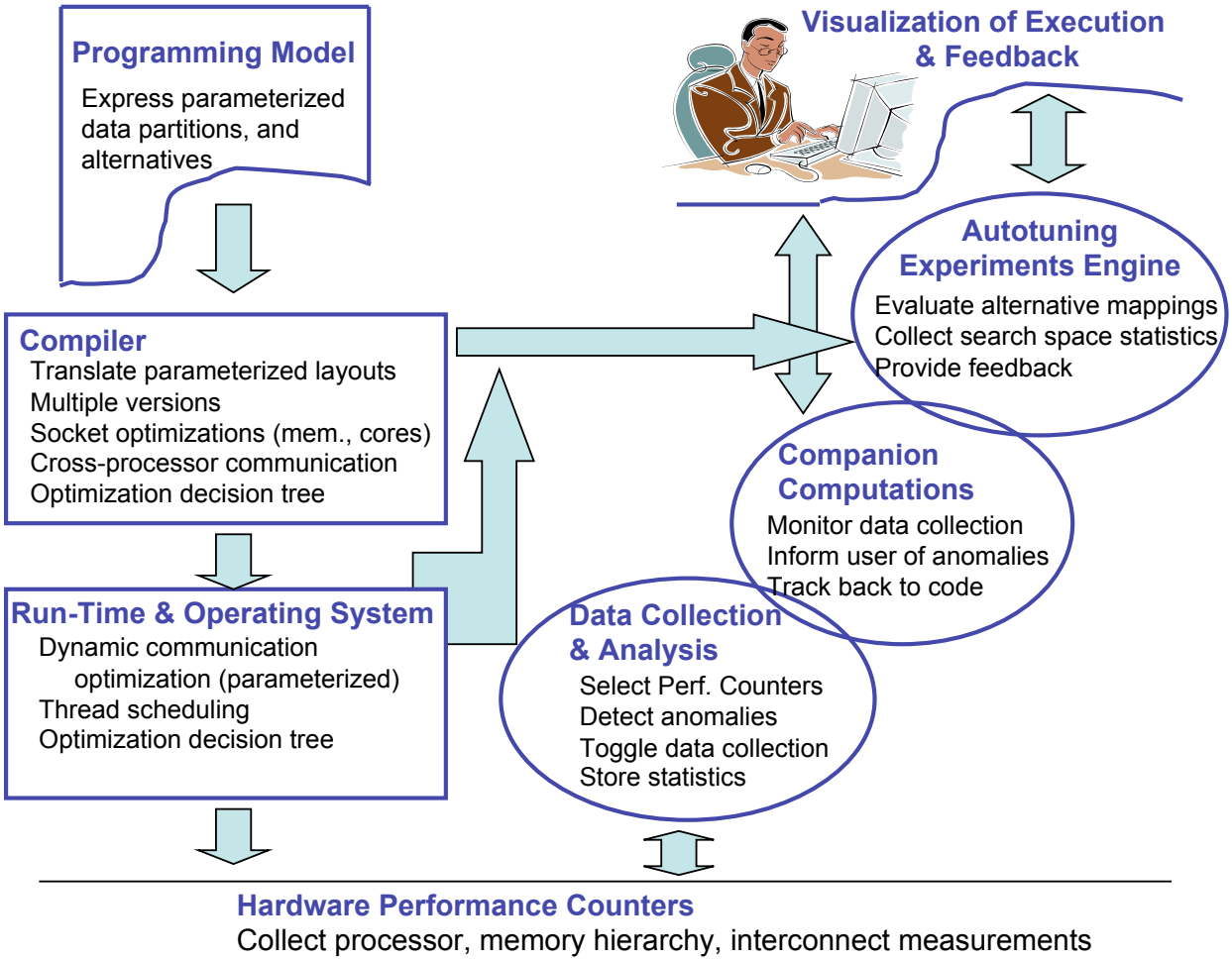

Figure 7.1: Illustration of information flow for Analyzing Data Partition

7.4.1 Scenario 1: Clarify Performance Behavior for Developer

Abstraction involves distilling relevant behavior so that it is meaningful to the developer and helps them implement changes that improve execution of their application. This requirement demands that the system pinpoint causes of problems and map them back to application constructs, and also provide the developer with suggestions of changes that have a direct and observable impact on execution behavior. Therefore, the previously-described multi-level integration must support, from initial design, mechanisms for such changes, and measurement capability to provide feedback on the effectiveness of such mechanisms.

To understand what the developer needs to know we look back to the early days of vector processors, when the execution behavior was easy to discover from feedback from the compiler and profiling tools. The developer simply had to go to the routine in the application where the time was being spent and then determine (with compiler assistance) if the code vectorized, and if not, why.

With the advent of large-scale and distributed-memory architectures, the reasons why the application is not performing well are more obscure and more difficult to remediate. When the developer specifies a potential decomposition approach (parameterized by problem and machine size) for their application data, this single choice has a significant impact upon the performance of the application. Once the decomposition is specified, a number of other performance features of the implementation are set as well, such as load balancing and number of messages. Additional optimizations on the message-passing and overlap with computation can further improve performance, including message consolidation and asynchronous pre-posting of receives. Once an application is in production use, the modification of the decomposition is typically very difficult and the optimization of the implementation of the message passing is the only way to improve the performance. Virtualization approaches as in CHARM++ [84] can ameliorate this problem by allowing the programmer to specify an "over-decomposition" and using the runtime system to perform dynamic load balancing.

Using a high-level language, for example High-Performance FORTRAN (HPF), the decomposition is easy to change; however, the optimization of the implementation is left to the compiler. Among the many lessons to be drawn from the HPF experiences of the 1990s is the need for invertible performance mappings that can relate the measured behavior of executing code to application programming idioms. HPF compilers generated message passing code (typically MPI) from data parallel FORTRAN, annotated with data distribution directives. Presenting message passing metrics, derived from instrumenting the generated code, to an HPF application developer was of no value, as the application developer had no way to either understand the performance problem or to change it. What was needed was a mapping from measured communication patterns to recommendations about changes to data distribution directives or code fragments.

The average application developer who uses the high level language would have little or no knowledge of the implications of parallelizing their application. Where there was a simple, "Did it Vectorize?" question on the vector machines, on the large scale parallel systems, the questions are significantly more difficult to ask. To address this challenge, the compiler/runtime system must be very precise and accurate as to the major reasons for the lack of scaling. As the level of abstraction rises, the difficulty of relating certain performance abnormalities to the actual code becomes much more difficult. The importance of solving this problem is at the root of the issue of using new languages that reduce the difficulty of using explicit message passing.

Thus, in Figure 7.1, we illustrate how the exascale software stack might collaborate to find an appropriate data layout for a computation. We rely on abstraction and information sharing, which have been among the most successful of our software ideas. Programming models and languages, libraries and runtime systems, operating systems and hardware, all define abstractions and

information sharing interfaces. However, these interfaces are predominantly lossy and communicate information in one direction — from above to below. To implement run-time adaptation and performability at exascale, we must increase the volume and types of bi-directional information sharing. For our performance optimization scenario, compilers must retain code transformation data to allow performance tools to relate measured data to application source code in ways that suggest how application code might be changed to increase performance or reliability.

Let us assume the programmer expresses in the programming model a parameterized data partition, with actual sizing of dimensions of the layout left to be bound empirically. The programming model may also allow specification of multiple organizations and partitions, *e.g.,* for a sparse graph, that can be compared. The compiler translates each partitioning strategy into parameterized code, possibly generating multiple alternatives that can be selected at run time. The compiler generates cross-processor communication, hierarchical parallelization across cores, and a set of optimizations to exploit the memory hierarchy. For each set of optimizations the compiler performs, it may have unbound parameters that can be set empirically. It may also have a set of decisions that it would prefer to defer until run time. Similarly, the run-time system will dynamically perform thread scheduling and optimize communication, and the run-time too may parameterize these optimizations or have a set of decisions it would

We can think of the result of compilation as a set of alternative mappings, rather than just one, and execution as a comparison of multiple different execution strategies. This suggests a complex evaluation process to arrive at the most appropriate mapping. An autotuning experiments engine evaluates the alternative mappings, providing feedback to the application programmer as to what mappings were most successful. To support this evaluation process, the hardware must provide access to performance counters to describe (among others): (1) processing within a core and within a socket; (2) memory hierarchy behavior; and, (3) from the on-chip and cross-socket interconnects. Companion computations derive important performance metrics and look for abnormal indicators that could indicate poor performance. On-line collection and analysis of data steers the search for the most appropriate mapping, with a tiny subset of results added to the performance database. For example, if a companion computation for data analysis observed load imbalance, it could initiate a tracing of messages coming into and going out of the process, combined with workload characteristics, to understand how the data layout is contributing to this performance problem. With such real-time access to the application, a companion computation could provide a sophisticated interactive visualization of the execution, providing the user multi-level access to performance across a set of experiments for different mappings. This interactive interface would be designed to correlate the application details to the location in the program responsible for the performance details. Through automatic evaluation of a range of implementations and online comparison across these implementations, such an approach would allow the programmer a view of how the layout specification impacts performance.

7.4.2 Scenario 2: Online Failure Detection and Response

Historically, we have treated performance and reliability as distinct objectives, with equally distinct approaches to design and optimization. For scientific and technical applications, checkpointing is the standard mechanism, where the nodes of an MPI application collect and write data to secondary storage, typically every few hours. These checkpoints can be used to restart the computation at a later time. Implicit in this approach is the assumption that failures can be detected reliably. Further, our performance tuning approaches rely on instrumentation and (predominantly) post-mortem analysis of the resulting performance data to identify and correct performance bottlenecks.

At exascale, we can expect failures to be more common, suggesting that we must bridge the

chasm separating performance and reliability approaches, addressing performability as a single, runtime, rather than post-mortem concept. This implies the need for real-time measurement of reliability indicators (*i.e.,* transient memory and data transmission errors, behavioral differentials across cores) and performance metrics (computation, networking and storage). By monitoring the temperature of a node, a key insight into the health of the node can be tracked. For example, monitoring the number of retries in the transfer of a message can give an indication of the health of the interconnect. This analysis could be combined with dynamic adaptation, including adaptively selecting checkpointing frequencies, multi-version code execution for verification, automated task retry and autotuning that adapts computation to changing resources. Equally importantly, this will require greater information transparency across software and hardware levels. Coupled with programming model support to describe responses to failure and information from previous executions, compilers should generate multiple code versions that reflect expected failure modes (*e.g.,* restartable models, configurable checkpoints, etc).

7.4.3 Scenario 3: Power Management

Exascale raises power management issues at two levels, both for individual nodes and at the system level. At the node level, this includes managing processor power states, enabling and disabling cores, memory and storage power management, and network interface states. To date, power management has focused largely on processors and voltage/frequency adjustment. However, rather than managing individual cores, managing collections of cores and their chip-stacked memory will become increasingly important, and complicated, as memory will be a major fraction of node power consumption. At the system level, it includes partition management and scheduling and interaction with cooling infrastructure, something not normally considered in high-performance computing software.

Given the wide range of power management scales, it is critical that clear interfaces be defined for information sharing and coordination across hardware, system software, runtime systems and applications. No single level of the hardware/software stack contains all the data needed for power management and optimization. Perhaps equally importantly, these power management issues are deeply intertwined with performance optimization and reliability management (performability), particularly when systems include heterogeneous cores.

To enable intelligent, adaptive decision making, exascale tool systems must include real-time measurement of performance metrics, reliability indicators and power consumption. In turn, decision runtime procedures must be designed to balance the oft-conflicting goals of high performance, bounded power consumption and reliable execution. Similarly, software tools should present not only performance data but power consumption profiles as well. Optimizing for performance alone can push systems beyond their thermal limits and increase the probability of component failure due to thermal stress. Conversely, optimizing for power consumption alone can unduly constrain application performance.

7.4.4 Scenario 4: Debugging

When a user encounters a problem when running an application across a large number of processors, they need to investigate the cause of the problem, where does it manifest itself? Is it a logic error or a system error? If it is a logic error, is it in the serial code, parallel code, or communication code? The application programmer has several options available to them:

1. Run the application again and see if the problem is repeatable

2. Run the application on fewer processors and see if the problem exists

3. Try to schedule interactive time on the same large number of processors to use a debugger to isolate the problem.

Since an exascale system will have billions of threads of execution, no one expects today's debuggers to scale to that level. Therefore, how can the aforementioned enabling technologies be employed to assist the programmer in identifying the problem? One of the most difficult problems is the amount of time a code compiled to be debugged takes to run. The use of the companion process has the potential to significantly reduce that time. With new research, we anticipate that the companion process should be able to use compiler and runtime information to perform the necessary mapping between optimized and unoptimized execution states without slowing down the application threads.

If the program runs the application again and it runs correctly, that does not assure that the application/software is correct. We could actually have the worst kind of error that is typical of a race condition where we have a successful execution one time and an unsuccessful application the next time. A companion process could assist the application developer in identifying the problem in several ways. For example, the companion process could make memory watch points significantly more efficient. By monitoring the executing program's memory, the companion could identify the initial point when a race condition occurs. Then it could halt the application and allow the user to step back to identify which operation actually caused the error.

Another use of a companion process would be to execute an earlier correct version of the program in parallel with the current version to be able to identify the point in time when the two versions differ. This type of analysis is particularly valuable when the programmer is restructuring a currently running application to perform more effectively. Development of a new age of debuggers which have access to a companion process would open up a wide range of additional capabilities to quickly identify error, even when running large scale applications.

A companion process could improve the performance of differential debuggers. With such a companion process, the application could be dynamically checkpointed when the user identified an abnormality. The user could then examine the contents of the application's memory and quickly determine the cause of the problem. Plotting the contents of an array in real time would allow the user to visually identify where abnormal computation is taking place, perhaps at a boundary that is not properly being handled in the application.

7.5 Summary

This chapter has highlighted three key areas of innovation in the software stack needed to develop useable tools that truly enhance programmer productivity in the complexities of an exascale regime. These three areas of innovation are: (1) scalable data collection and analysis; (2) companion computations; and, (3) auto-tuning. Using four scenarios highlighting how tools could enhance programmer productivity, we examine how these areas would facilitate interaction with the application developer and the rest of the software stack. While a lengthy discussion of tool research is beyond the scope of this report, we describe in more detail the set of tools currently used for petascale or envisioned for exascale in Appendix A.2.

Chapter 8

Technical Approach

In this chapter, we outline key elements of a technical approach for Extreme Scale software. The aim of this chapter is to provide examples that are indicative of the kind of software technologies that will be needed to address the Concurrency and Energy Efficiency challenges of Extreme Scale systems, without prescribing specific solutions. It is expected that the community will create shared open source components through efforts such as the International Exascale Software Project [56] that can be leveraged in building Extreme Scale software stacks. Section 8.1 highlights the importance of *software-hardware interfaces* in an Extreme Scale system. Section 8.2 identifies opportunities for addressing Concurrency and Energy Efficiency challenges through *software-hardware co-design*. Section 8.3 discusses the importance of *deconstructed operating systems* in the future OS roadmap for Extreme Scale systems. Section 8.4 presents a vision for Extreme Scale system software based on the notions of a *global OS* and *self-aware computing* [36]. Finally, Section 8.5 describes an example execution model and technical approach, in an effort to encourage the community to think of breakthrough approaches for building Extreme Scale software.

8.1 Software-Hardware Interfaces in an Extreme Scale System

Figure 8.1 shows a notional structure for software and hardware interfaces in an Extreme Scale system. The main motivation for these interfaces is that we expect optimization of Concurrency and Energy Efficiency to be best achieved by graceful cooperation among software and hardware layers. The separation between software and hardware layers is specified as a "Hardware API" — we refer to this interface as an "API" to emphasize the fact that hardware interfaces should not look different from software interfaces from the application viewpoint. As shown in Figure 8.1, there can be multiple levels of information flow within and across software and hardware layers. Some examples of this information flow are as follows:

Programming language supplying data access pattern information to compiler: Programming languages do not often provide the means to express many facts that the programmer may know about their application, such as data access patterns. For example, a programmer may write a library routine for generality such that it can handle both sparse and dense array arguments. However, if the routine is often used for dense linear algebra, a hint like `#pragma expect stride 1` could help the compiler generate more efficient code in the common case. These sorts of pragmas were commonly supported in languages and compilers for old vector machines, but are rarely found on modern cluster platforms. If the programmer does not know what data access patterns the code engenders (perhaps the programmer responsible for code tuning/maintenance did not write the code originally), they should be able to use a tool like

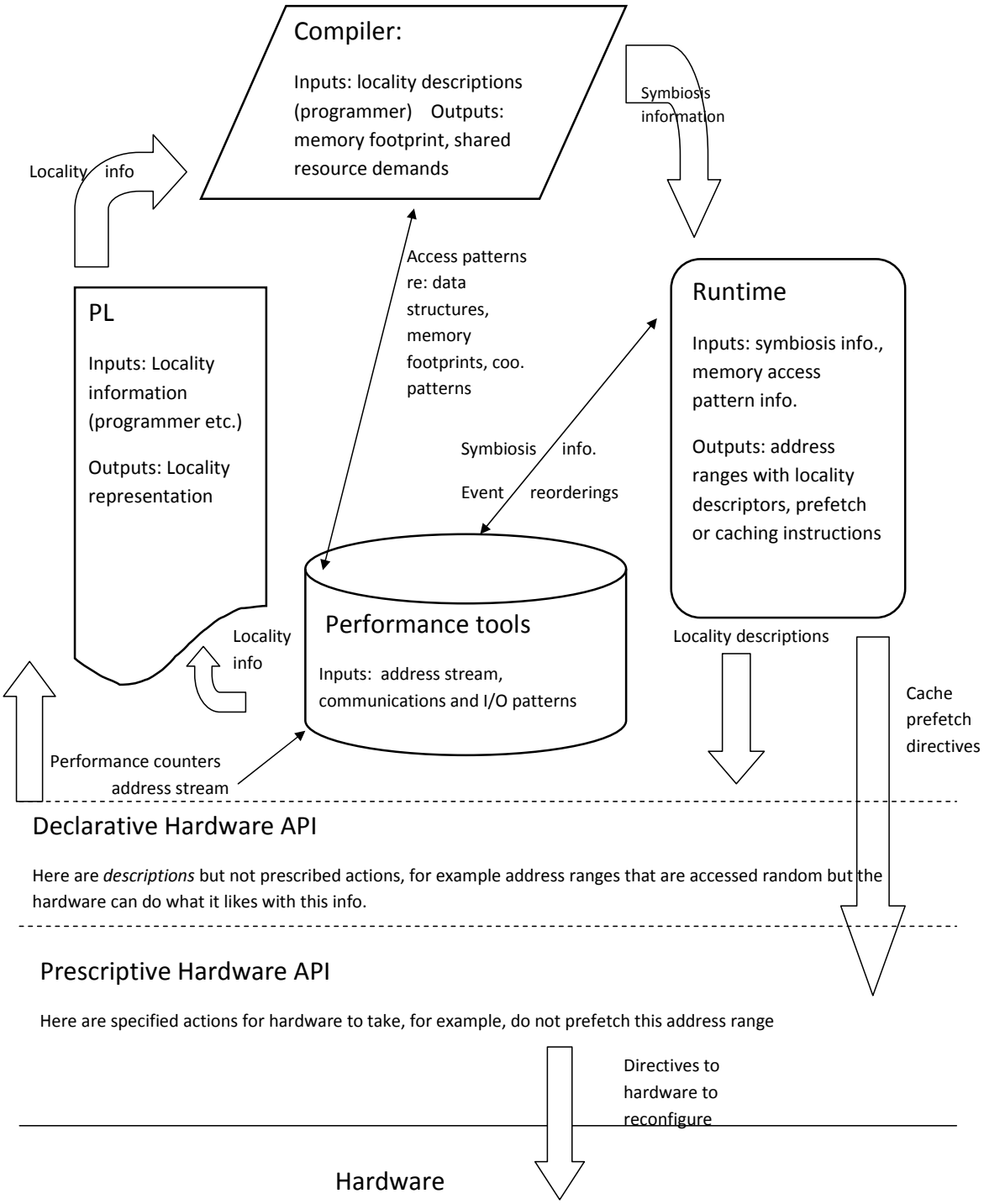

Figure 8.1: Software and Hardware Interfaces in an Extreme Scale System (Notional)

a memory access profiler to discover the pattern, and then insert a pragma to capture what has been learned. This scenario involves a feed-back loop among the programmer, compiler, and memory tracer. It should be noted that trace-based recompilation is available today but memory access pattern information for the most part is not available today (a missing, but somewhat trivial to supply hardware API feature).

Compiler providing scheduling information to runtime system: Compilers invest a lot of effort to discover the control flow, instruction mixes, and in many cases the data access patterns and memory footprints of an application, then throw this information away after code generation. In the meantime if the runtime knew the computational demands of the programs being handed to it, it could reason about resource conflicts, and (say) explore symbiotic scheduling that avoids conflicts on shared resources. For example, by binding a non-memory-intensive and a memory-intensive thread together, rather than two memory intensive threads together on the same shared memory node, cache thrashing and memory bandwidth competition could be reduced.

Runtime system controlling data-motion hardware at a fine level of granularity: Armed with information from the user and compiler about what data access patterns to expect, the runtime should be able to ask the hardware to cache certain data structures, not cache certain others, and also should be able to control prefetching and other data motion operations. Based on this information conveyed in a hardware API, a smart memory controller should then be able to avoid moving long cache lines, big memory pages, and substantial disk blocks when only one element of them is likely to be needed.

Performance tools: Memory address stream information (including communications and disk I/O) can be used by performance tools to carry out very simple data analyses (*e.g.,* identifying data structure access patterns) or highly sophisticated analyses (*e.g.,* identifying possible task reorderings that preserve semantics while improving locality) and reflect this back to the appropriate level. In Figure 8.1, the performance tools sit in the middle, observe, and inform, all the other levels.

Declarative Hardware API: This high-level API describes properties of data structures or computations that hardware may choose to take advantage of, for example that a data structure is accessed randomly, or that a computation is memory intensive. The API is also used to support queries on the choices made by the hardware that go far beyond today's hardware performance monitor interfaces *e.g.,* querying the cache management policies used by the hardware for a given address range, based on the declarative information received by the software.

Further examples of the high-level declarative API include:

- Performance profiling requests that include identification of events to be counted and sampled, and interface for software to collect information on performance events such as AMD's proposed interface for Lightweight Profiling.
- Resilience information that includes identification of threads with lower resilience requirements *e.g.,* for which software can perform error detection and recovery so hardware does not have to.

Prescriptive Hardware API The declarative hardware API establishes information flow between software and hardware, does not enable the software to directly control the hardware's

actions. In contrast, the lower level prescriptive hardware API, describes actions that the hardware should or should not take. For example, the hardware can be told to not cache data in a certain address range, or that it should voltage-scale the processor when running a particular region of code. The lower level API should allow hardware to reflect up hardware performance counter information including access to physical or virtual address (something not the case today) without high overhead; this feature alone would greatly expand the capability of performance tools to carry out analysis that in turn can feedback to all layers of software, and back to hardware via API, to improve parallel and energy efficiency. Further elements of the low level API could include:

- Memory hierarchy configuration parameters *e.g.,*
  - Cache sizes, line lengths, degree of associativity
  - Register file sizes, data widths
- Memory access patterns
  - Address ranges that should bypass cache
  - Address ranges that require hardware coherence
  - Address ranges for which coherence will be managed by software
  - Address ranges with values that are guaranteed to be read-only (immutable) for certain application phases
- Network bandwidth partitioning for different forms of data movement and communication
  - PGAS
  - RDMA
  - Message passing
  - Stream processing
  - Other network reconfigurability parameters including topology and packet size

  Power management
  - Frequency scaling (smaller time constant, but constrained by voltage)
  - Voltage scaling (larger time constant)
  - Issue width management
  - speculation control
  - core power cycling temperature-based power policies

As different software stack components are developed for Extreme Scale systems, it will be desirable to reuse components across the three classes of systems. To that end, Table 8.1 summarizes the key similarities and differences among software stack components across all three extreme scale system classes.

8.2 Opportunities for Software-Hardware Co-Design

We believe that software-hardware co-design will be a critical necessity for Extreme Scale systems, in addition to the interfaces outlined in the previous section. This form of co-design has been essential for *vector parallelism* [85] in current and past systems, and is also being explored for scalable approaches to mutual exclusion using *transactional memory* [92]. In this section, we discuss a few additional examples of software runtime capabilities that will be necessary for future Extreme Scale systems, and examine how they can be made more effective with software-hardware co-design.

Software Stack Components	**Data Center**	**Departmental**	**Embedded**
Operating Systems	Capability: restricted high-end applications	Feature-rich: broader mix of applications, ISV software	Deterministic, Real time
File Systems	Scalability: Large number of files	Enterprise file system	Flat, simple, limited
I/O	Some apps are massively I/O bound	Standard network I/O, richer set of I/O drivers	I/O bound
Resilience Runtime	Intelligent checkpointing	Could approach Enterprise mission-critical	Fault tolerant apps, Replication for mission-critical
Task Scheduling Runtime	Featherweight asynchronous tasks	Lightweight asynchronous tasks	Deterministic, Real time
Memory Management Runtime	Typically, single job at a time, limited # large address spaces	More sharing, larger # address spaces	Single job
Intra-System Communication Runtime	Massive # nodes, special communication hardware, Special controls for data movement, Latency on critical path	Commodity	Small number of modes, application-specific data movement
Power Management Runtime	Adaptive, large-scale power management	OS-driven	Application-driven (platform-limited)
Profiling & Monitoring Runtime	Scalable, Online analysis/aggregation, anomaly detection	Should be accessible to non-experts	Lightweight, predictable, fixed time intervals
On-chip communication Runtime	Locality-aware dynamic data movement and load balancing	Locality-aware dynamic data movement and load balancing	Support for dynamic, systolic, and statically scheduled communications
Static & Dynamic Compilers	Support for new task & locality constructs, use of spare cores, dynamic optimization	Support for legacy codes and their migration	System-level optimization, more heterogeneity
Programming models	Expression of new task & locality constructs, ultra-fine-grain parallelism	Expression of new task & locality constructs, very fine-grain parallelism, Should be accessible to non-experts	Expression of new task & locality constructs, real-time constraints

Table 8.1: Key Software Stack Component Similarities and Differences for Extreme Scale System Classes

8.2.1 Scheduling dynamic parallelism with fine-grained tasks

As discussed in Chapter 5, it is important to ensure that the intrinsic parallelism in a program can be expressed at the finest level possible *e.g.,* at the statement or expression level, and that the compiler and runtime system can then exploit the subset of parallelism that is useful for a given target machine. There have been multiple proposals for expressing fine-grained parallelism *e.g.,* statement-level *spawn* [43] or *async* [50] operations, expression-level *future* [76] operations, and operator-level data flow graphs [54, 128]. These operations for fine-grained parallelism are in stark contrast with the *bulk-synchronous parallel model* [139]. While profile-directed compile-time partitioning can be used to optimize the granularity of fine-grained tasks in certain cases [116, 117], in general the runtime system also needs to participate in the partitioning so as to best adapt to unpredictable execution times. A classic approach to runtime partitioning is *lazy task creation* [104], which has been extended into *work-stealing runtimes* for fine-grained tasks [65, 73]. A work-stealing runtime system creates a fixed number of worker threads, with one local double-ended queue (deque) per worker. Each worker repeatedly picks up work from a deque of lightweight tasks using scheduling policies that are designed to achieve good load balance while bounding the size of the deques. This approach has been shown to yield scalability that is orders-of-magnitude superior to the scalability achieved if each task were to be created as a thread at the OS level.

However, there are still significant overheads that remain in a software-only approach, that will likely prevent it from being usable at Extreme Scale. These overheads involve locking operations, and in the case of nonblocking algorithms involve spin loops on shared memory locations with their accompanying cache consistency overheads. As mentioned earlier, these overheads are especially important because they occur on critical paths in parallel programs. *Hardware support* for shared queue data structures can result in orders-of-magnitude reductions in scheduling overheads and scalability bottlenecks, while still retaining the flexibility of task scheduling policies in software. As mentioned in Section 8.2.4, hardware support for shared queues can have other uses as well in Extreme Scale systems.

Another source of overhead in task scheduling lies in the operations that need to be performed on the fast path to save local variables, so as to ensure that the task can be resumed on a separate worker from the one that it started on (if needed). A software-only approach introduces word-at-a-time store instructions to save the local variables, and some of these stores are often redundant. In contrast, hardware support for saving and restoring local variables (as in calling conventions) can help reduce this overhead that occurs on the fast path.

8.2.2 Distribution and co-location of tasks and data

Another candidate for software-hardware co-design pertains to distribution and co-location of tasks and data, which is one mechanism that can be used in support of locality optimization. As observed throughout this report, it will be critical to optimize vertical locality so as to satisfy the energy constraints of Extreme Scale systems. Runtime systems for programming languages such as UPC [60] and Co-Array Fortran [109] that are based on a *Partitioned Global Address Space* (PGAS) model include the notion of virtual *home location* for each shared datum. The more recent HPCS languages extend this notion of home locations to computational tasks, as in Chapel's *locales* [53] and X10's *places* [50], so as to enable tasks to be shipped to data, data to be shipped to tasks, or any meet-in-the-middle combination thereof. The translation from global to local addresses is a major source of overhead in a software-only approach to implementing such languages, along with the communications that accompany non-local accesses. Thus, it becomes important for a compiler for such languages to perform redundancy elimination on address computations, to coalesce

contiguous accesses into a single communication operation, and to overlap communication with computation [148]. However, many of these optimizations can still remain in a software-only approach after compiler optimization. Opportunities for software-hardware co-design include the use of translation buffers to accelerate virtual-to-physical address translations, and DMA-like hardware support to reduce the processor overhead of data transfers.

8.2.3 Collective and point-to-point synchronization with dynamic parallelism

As discussed in Section 5.4, the fine-grained parallelism intrinsic to a program may need to be accompanied by fine-grained collective and point-to-point synchronization among dynamically varying sets of fine-grained tasks. These synchronization structures may be irregular, and tasks are permitted to dynamically join or leave these structures as in the *phasers* construct [125]. Further, it is usually desirable to augment the synchronization structures with communication for reductions [124], collectives, and systolic computations. As discussed in Section 8.2.1, synchronization structures always represent good candidates for software-hardware co-design since a software-only approaches for synchronization suffer from unnecessary cache consistency and serialization bottlenecks. Hardware support (*e.g.,* in the form of counting semaphores) can be used to reduce the overhead of inter-core synchronization, and extensions in the form of register-level inter-core communication (*e.g.,* as in the Raw project [94]) can reduce the overhead of communication. Further, the use of a single master task to perform a reduction in software can be a scalability bottleneck, and a software-only approach to creating combining trees incurs high setup and tear-down overhead. Instead, hardware support for combining synchronization and reductions will greatly reduce the overhead of collective and point-to-point synchronization with dynamic parallelism.

8.2.4 Producer-consumer parallelism

Another common idiom in fine-grained parallel programs is that of producer-consumer parallelism. In this model, a single-writer task serves as the producer of a datum for multiple readers. To accomplish this, the writer task typically stores its result in a designated location, and the reader tasks block when they request the result (if the result is not ready). In the case of *futures* [76], the execution of the writer task may (optionally) be deferred till the datum's value is requested by one of the readers. Once again, we observe that a software-only approach suffers cache consistency and serialization bottlenecks, and hardware support can be used to reduce these bottlenecks. A classical example of hardware support in this area is the *full-empty bit*, but there may be many other variations. Also, in many cases, the location on which the producers and consumers wish to synchronize may be designated by a tag rather than an address. Hardware support to accelerate the translation of tags to addresses can be very useful. Intel's Concurrent Collections (CnC) [46, 47] is an example of a high-level programming model that relies heavily on producer-consumer parallelism and that would benefit greatly from any hardware support.

8.3 Deconstructed Operating Systems

Operating systems must be refactored (deconstructed) to offer more flexible resource management and runtime support for parallel execution models with the focus on exposing system resource usage policies to the various level of the programming stack. The overall goal of a deconstructed OS would be to allow the application to compose the best resource usage policies for its particular needs and to adapt to system scale and load. Policy control should be hierarchical, with different levels of abstraction depending on their consumer. For example, a future communication scheduling

mechanism could expose to the libraries/compilers explicit control over message sizes and ordering, while exposing to the application level/programmer only abstract policies like "long routes first". Adaptation can take form of Quality of Service mechanisms or migration for communication locality.

Previous experience with compiler and runtime optimizations for PGAS languages indicates that lightweight control over OS mechanisms is not sufficient for good performance and additional control is required over the policies that guide the management of these mechanisms. Looking beyond PGAS languages, even more control of resource management will be necessary to support the kind of novel execution strategies required for Exascale applications. Current OS designs favor generic policies, *e.g.,* preemptive thread scheduling or Least Recently Used page replacement, which have been selected as being the least common denominator for commercial workloads. In contrast, the execution models required for structured parallel algorithms in scientific computing applications are less diverse, usually require cooperation among computing entities and exhibit a relatively ordered execution imposed by data/task dependence.

8.3.1 Role of Hypervisors in a Deconstructed OS

The resources (such as memory, bandwidth, and power) in an extreme scale system are expected to be highly constrained. Exascale hardware technology is also envisioned to be highly memory constrained. Therefore, rather than a full OS model, there may be substantial benefit from an "exokernel" model [61] where applications are bundled with only the necessary OS functions linked in to the application that run in their own virtual machines (VM) container, relying on the hypervisor or VM container for protection, resource sharing, and management of Quality of Service (QoS) guarantees. We will refer to these protection mechanisms as an "application container" because there are a number of technologies including hypervisors, VMMs, and runtime environments such as Singularity that can implement this kind of isolation in a spatially partitioned CMP. The primary roles of the application container are to manage partitioning of hardware resources, including physical processors, physical memory, and memory and interconnect bandwidths, while the runtime layer above the application container will have complete control over scheduling and virtualization, if any. For example, in an SPMD execution layer used in UPC, there is no need for processor virtualization, while for dynamic threading used in the DARPA HPCS languages, a lightweight user-space thread scheduler that can be directly controlled by the application or runtime would be beneficial.

One approach to both operating systems and runtimes for parallel execution is to deconstruct conventional functionality into primitive mechanisms that software can compose to meet application needs. A traditional OS is designed to multiplex a large number of sequential jobs onto a small number of processors, with virtual machine monitors (VMMs) adding another layer of virtualization. An alternative approach is to explore the usage of a very thin hypervisor layer that exports spatial hardware partitions to application-level software. These virtual machines allow each parallel application to use custom processor schedulers without fighting fixed policies in OS/VMM stacks. The hypervisor supports hierarchical partitioning, with mechanisms to allow parent partitions to swap child partitions to memory, and partitions can communicate either through protected shared memory or messaging. Traditional OS functions are provided as unprivileged libraries or in separate partitions.

For example, device drivers run in separate partitions for robustness, and to isolate parallel program performance from I/O service events. An alternative "deconstructed" architecture would enable partitioning not only of cores and on-chip/off-chip memory but also of the communication bandwidth among these components, with QoS guarantees for each partition. The resulting performance predictability improves parallel program performance, simplifies code (auto)tuning and

dynamic load balancing, and supports real-time applications.

Hypervisors, VMM-based application containers, and various code-rewriting systems offer a thin protection layer that can be used to support desired capabilities for massively parallel CMP-based systems summarized in the following subsections.

8.3.1.1 Minimalism, Modularity, Mediation

A thin protection layer is needed on a CMP to prevent hardware state corruption, but otherwise offer bare-metal access to the underlying hardware wherever possible. Many system facilities will be linked at user level as a set of optional systems libraries. Hardware protection mechanisms will allow direct, user-level access to facilities such as networking and I/O through the lightweight protected messaging layer. Parallel applications will be given bare metal partitions of processors that are dedicated to the given application for sufficiently long periods to provide performance predictability. The primary roles of the protection layer are to manage partitioning of hardware resources, including physical processors, physical memory, and memory and interconnect bandwidths, while the runtime layer above the protection layer will have complete control over scheduling and virtualization, if any. There are many lessons to be learned from K42 [39], the MIT Exokernel [61], and embedded operating systems such as VxWorks [34] regarding approaches to efficient and modular system services.

The thin protection layer (possibly a hypervisor or VMM-based application container) will play a role in mediating concurrent access to devices to ensure fair sharing of resources. Although software functions can be virtualized through replication, hardware devices are finite and access to them by multiple hardware components must be managed. This will include Quality Of Service guarantees for access to certain rationed resources such as memory or network bandwidth.

8.3.1.2 Isolation

Groups of processors can be combined into protected partitions. Boundaries will be enforced through hardware mechanisms restricting, for example, sharing of memory across partitions. Messaging between partitions can be restricted based on a flexible, tag-based access control mechanism. OS functionality such as device drivers and file systems will be spatially distributed rather than time-multiplexed; we refer to this as spatial partitioning. This approach works in synergy with the sidecore techniques [91], which allow an application to vector OS or driver functions to execute on free cores rather than forcing a context-switch on the core running the application.

8.3.1.3 Safe User-Level Messaging

Messages will be used to cross protection domains rather than more traditional trap-based mechanisms. Through hardware mechanism and/or static analysis, applications will have direct, user-level access to network interfaces and DMA. Further, fast exception handling and hardware dispatch of active message handlers will permit low overhead communication without polling. Most traditional system-calls will translate into messages to remote cores (other system-calls will be to linked system libraries) [91].

8.3.2 Related Work

Operating system research is an infrastructure-intensive process with a long initial development time. Traditional operating systems have a monolithic design with little or no control over the internals exposed to the application or user level. Novel programming models need to demonstrate

clear performance and productivity advantages over the established paradigms in order to have a chance for widespread adoption. Their efficiency can be greatly improved when having access to fine application level control over functionality provided by the system software stack (OS). The same fine-level of control is beneficial to established execution models of existing parallel runtime environments.

There are several DOE sponsored ongoing research projects related to operating systems design. The ZeptoOS [35] project distinguishes between service node and compute node kernels and provides tool for OS instrumentation and understanding of the interaction between the OS and the application layer. The Right-Weight Kernels [103] project focuses on a judicious selection of OS level services in order to diminish OS interference. The K42 research project focuses in parallelizing the OS itself and providing abstractions for overall system adaptation at scale. A common characteristic of these projects is the focus on improving overall system behavior and reducing OS interference [113] and putting more resources under the application and user space control while maintaining fault isolation and security. They mostly explore the opportunities offered by lightweight kernels and focus on kernel level mechanisms for resource control: CPU, memory, and network.

8.4 Global OS and Self-Aware Computing

8.4.1 Services for Adaptation at Large System Scale

Optimization and execution decisions should be made based both on the instantaneous system state and global knowledge about the application behavior. For example, data transfer mechanisms should be instantiated based on current load and the logical communication topology of the application. This unfolds into the following research directions:

- Develop automated frameworks for exploration of system characteristics to understand scaling behavior. Scaling behavior is determined by a combination of the hardware characteristics of the system and the way the application uses the hardware. The result is a highly dimensional parameter space that is not fully explored by current benchmarking techniques. Furthermore, due to restricted access to large-scale systems, the benchmark process today provides in some instances statistically unsound data.

- Develop policies and mechanisms for adaptation. These will include:

    1. Automatic mechanisms to enforce flow control and avoid congestion intra and inter node, exposing and using QoS primitives in the application
    2. Intra-node mechanisms for communication scheduling such as scheduling of communication operations based on layout, *e.g.,* long-range communication has priority over short range communication or coalescing of communication operations that target the same node.
    3. Selection of the best communication primitives based on load and application requirements, *e.g.,* selection between active message or pipelining based implementations for applications that process a large number of disjoint remote memory regions.

- Develop mechanisms to expose application behavior to the adaptation mechanisms. This may include exploration of two different alternatives:

    1. offline instrumentation and feedback

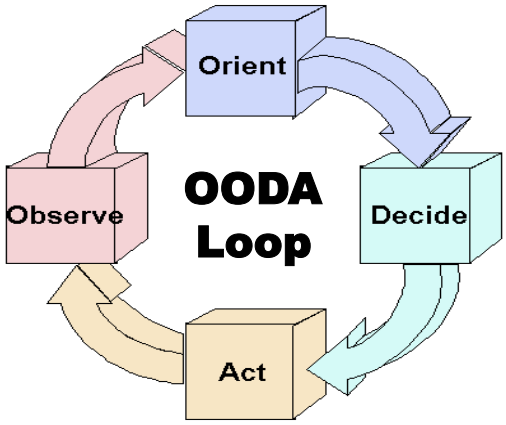

Figure 8.2: OODA Loop

2. continuous monitoring and feedback loop.

- Explore formal methods to describe the communication requirements and behavior of an application and annotated execution models.

8.4.2 Sensors and Actuators

Sensors will include *machine profile sensors* that can determine properties of the target machine on which the application runs including capacities and bandwidths of the memory hierarchy, to allow the application to adapt to the hardware configuration on which it runs. Further sensors may determine dynamically the level of contention form the hardware/runtime and allow an application to deploy an algorithm that is more efficient when bandwidths to shared resources drop.

Actuators from the application side will include *application signatures* to represent the resource requirements of the application to the other levels of the stack and the hardware. For example an application's memory footprint will be an actuator that can cause the runtime to allocate sufficient memory; memory access patterns may cause the runtime or compiler to optimize prefetching policies for the specific application. Information about symbiosis (the impact of the application upon shared resources) may be used by the O/S to choose co-scheduling partners for the application.

8.4.3 Self-Aware Systems

Current operating systems have pre-programmed behaviors that are based on guesses about resource availability. As a result, they are ill-suited to complex multicore systems and result in sub-optimal performance in the face of changing conditions. In contrast, it will be desirable for the OS to behave like a *self-aware system* that "learns" to address a particular problem by building self-performance models, responding to user goals, and adapting to changing goals, resources, models, operating conditions, and even to failures.

Figure 8.2 illustrates the basic Observe-Orient-Decide-Act (OODA) loop of a self-aware system. Thus, a self-aware system is

- Introspective — it observes itself, reflects on its behavior, and learns.

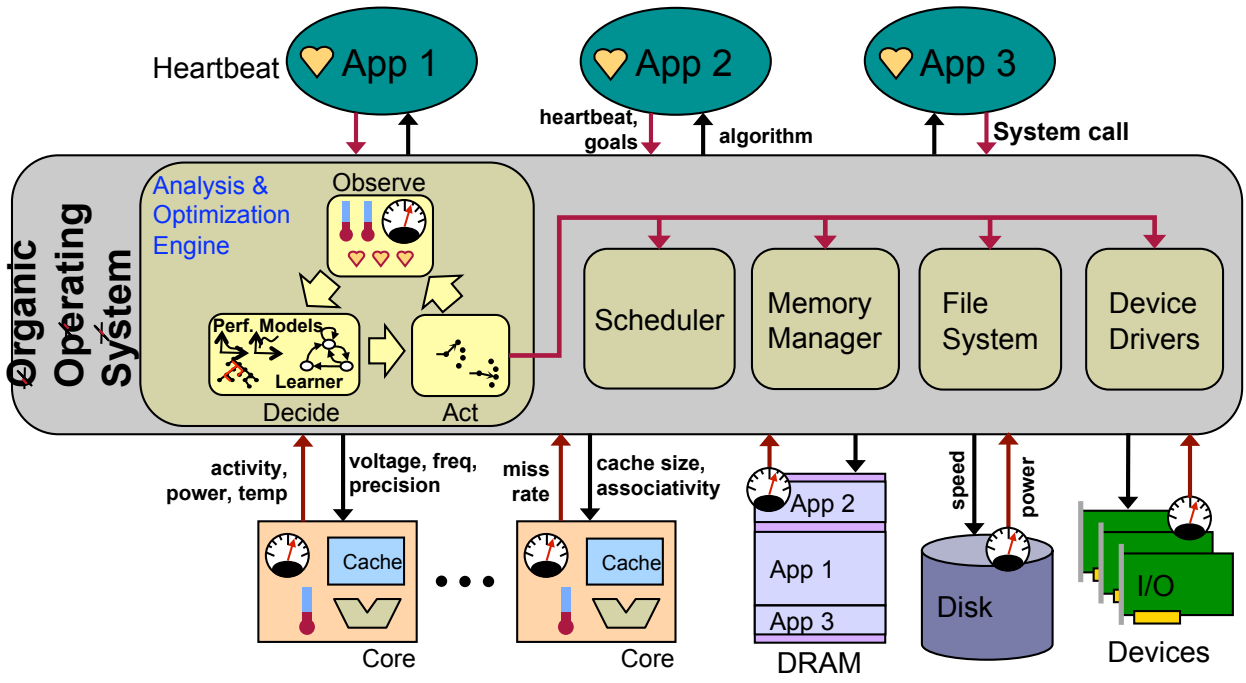

Figure 8.3: Organic Operating System (Notional)

- Goal-oriented — ideally, the system's client only specifies the goal, and it is the system's job to figure out how to get there
- Adaptive — the system analyzes the observations, computing the delta between the goal and observed state, and takes actions to optimize its behavior
- Self-healing — the system continues to function through faults and degrades gracefully
- Approximate — the system does not expend any more effort than necessary to meet goals

As an example of a self-aware OS, consider the notional Organic Operating System (OOS) shown in Figure 8.3. The Observe and Orient steps are accomplished via a number of new *sensor* interfaces that will need to be added to all software and hardware components of an Extreme Scale system (yet another example of software-hardware co-design). These observations include processor characteristics such as performance, energy, miss rates, queue lengths, resource utilizations as well as physical characteristics such as temperature. The Decide and Act steps are enabled by a new *actuator* interfaces that enable the system to control application behavior at a number of levels such as number of allocated, cache configurations, scheduler policies, clock frequencies, and numerical precision desired. Thus, a separation of concerns is achieved between the application and the OOS where the application communicates goals and options to the OOS, and the OOS uses component performance models to decide how best to meet goals under given system constraints.

8.5 Silver: An Example Execution Model and Technical Approach for Extreme Scale Systems

This section offers an example of one possible Exascale software stack, referred to as "Silver", to serve as an exemplar for the general class of transformative software strategies to address the broad challenges of extreme scale computing. The design of Silver is based on the premise that any extreme scale software stack must address four critical obstacles to scalability: starvation or insufficient program parallelism, overhead or critical path management work, latency to main memory and across system, and contention for concurrent service requests to shared resources. Each of these critical factors is influenced and impacted by design choices made at every layer of the computing stack. Furthermore, the design of any stack layer interrelates with all of the others, but in particular with adjacent layers. Additional performance and quality of service factors to be addressed include reliability, availability, programmability, and cost. The future generation extreme scale system software stack may take on any one of many forms and understanding and evolving these will require a vision that recognizes these needs.

8.5.1 Silver Execution Model

The unifying set of principles for Silver is a model of computation that is a synthesis of semantic constructs, policies, and mechanisms comprising the logical organization and operation of a parallel computation to be performed. The Silver model strives to

1. provide an abstraction of parallel computation that exposes and exploits a high degree of algorithm concurrency, particularly that available from dynamic directed graph structure-based applications,

2. enable intrinsic latency hiding through automatic overlap of computation and communication through message-driven work-queue multithreaded execution,

3. minimize impact of synchronization and other overheads for efficient scalable execution through lightweight object-oriented semantics,

4. support dynamic global address space scheduling for adaptive resource management, and

5. unify heterogeneous structure computing for diversity of processing modalities and exploitation of accelerator micro-architectures.

The Silver model supports a work-queue model. Threads are created locally by other threads or remotely by message-driven mechanisms (parcels) permitting work to be moved to the data. Threads are organized within the contexts of parallel processes, each spanning potentially many local domains (localities). Synchronization among threads is achieved through local control objects (LCOs), providing a plethora of mechanisms from simple mutex to more complex dataflow and future constructs. The message-driven model serves logical destinations or physical destinations, accelerators, or output devices. Percolation supports pre-staging of data and executables at remote resources to hide latency to these separate computing elements by overlapping communications with computing and to avoid their overhead costs, needed to take advantage of heterogeneous processors and precious resources.

8.5.2 Silver Stack

Silver addresses the challenges of the design and operation of a system stack for a new generation of ultra scalable computer systems capable of extreme scale performance with technologies nearing nanoscale feature size and the end of Moore's Law. The overall system stack as structured by Silver is deceptively similar to more conventional systems and comprises seven layers:

1. Driver applications — a focus on the emerging class of applications incorporating very large sparse, irregular, and time varying data structures including science and informatics will drive co-design across the system stack with a set of selected problems employed for experimental pursuits.

2. Programming models, methods, and tools — provides at least three distinct but interrelated user-level programming interfaces that serve as protocols for program requirements to system resources and control mechanisms. These variations include a library of service calls, mapping to this of conventional (legacy models), and an advanced low-level language optimized around the needs of the execution model and the directed graph based applications.

3. Compiler strategies and design — combines compile time analysis, adaptive learning, just-in-time modules, interface to advanced runtime mechanisms to control application parallelism, data management and distribution, and physical resource allocation.

4. Runtime system software — provides dynamic application execution scheduling, synchronization, and name space management under the control of the compiler and utilizing OS supplied resources.

5. Operating system - manages the hardware resources, by providing control of user processes, virtual memory allocation and name spaces, provides essential services to programs and runtime system, and recovery response in the presence of hardware and software faults.

6. Architecture - comprises both system level and micro-architecture utilizing the underlying enabling technologies to provide most effective and scalable computing for the essential modalities exhibited by the target applications by means of the software and programming layers.

7. Enabling technologies — defines a technology roadmap that will establish the bounding conditions, opportunities, and requirements that have to be satisfied to realize efficient, scalable computation.

For this Exascale Software Study the first and last layers of the System Stack provide boundary conditions and the penultimate layer, architecture, is a flexible supporting medium of computation that will be influenced by as well as influences the other layers of the software stack. We elaborate on items 2–6 below.

8.5.2.1 Programming Methods

An important part of the programming layer of the stack is to ensure that existing applications (so-called legacy applications) which embody a great deal of knowledge and development can be adapted to the Silver platform. This means adapting codes that use the Message Passing Interface (MPI). There are several approaches to this. First, it needs to be demonstrated that MPI programs can run well on this system. The MPI Forum has begun developing the next generation of the MPI specification. This layer of the stack must reflect the extensions to MPI that will define MPI-3.

Silver-Threads will be a language for the Silver programming model. It is expected that this effort will result in a new language, not bound by the semantic decisions of any existing languages. This programming interface will be largely defined by what it chooses to expose to, and what it chooses to hide from, the programmer. Low level constructs, such as synchronization through local control objects and messaging via parcels, will be hidden by high-level descriptions of tasks. The programmer should not be concerned with the exact locations of resources: all resources will appear local, with the address translation and parcel system handling migration of flow control between localities. Silver-Threads should take advantage of the advanced namespace management in the Silver execution model, including the DGAS component and the promotion of processes and threads to first class objects. The latter is particularly interesting, as it will allow the program to "reason about its execution." Another facet of Silver-Threads programming language will be the ability to specify optional heuristics, for providing "hints" to the runtime system regarding expectations for load balancing and resource management.

8.5.2.2 Compilation Methods and Tools

Substantial amount of research in parallelization has focused on "regular" programs which manipulate dense matrices. Unfortunately, exploiting parallelism in "irregular" programs — such as those that operate on lists, trees and graphs — is much harder. The amount of parallelism in dynamic graph-based computations depends on the input data and is not known until execution. The Silver compiler approach to this problem is to develop a suite of dynamic locality-aware partitioning techniques that allow hiding of global system latency. These techniques exploit significant computation-communication overlap. For dynamic graph-based computations, effective dynamic mapping and remapping of data will be crucial to realizing high-levels of performance. The Silver approach is to first develop mapping and remapping techniques, which will be augmented with strategies for deciding when it is effective to remap data. These solutions will be integrated seamlessly with the message-driven threaded-execution model that is at the heart of Silver Execution Model. Another task for the compiler is to extract high levels of performance from the underlying heterogeneous architecture containing such diverse themes as streaming and PIM micro-architectures. We will develop program partitioning strategies aimed at this.

8.5.2.3 Runtime Systems

The Silver runtime system is designed to be a modular, feature-complete, and performance-oriented representation of the Silver execution model on conventional (Linux based) architectures, offering an alternative to conventional computation models, such as MPI. It will also be targeted to the advanced Silver OS described below. The Silver model is intrinsically latency hiding, delivering an abundance of parallelism in within a hierarchical distributed global shared name-space environment. This allows the Silver runtime to provide a multi-threaded, message-driven, split-phase transaction, non-cache coherent distributed shared memory programming model using futures based synchronization. It is a second-generation runtime system for possibly heterogeneous platforms designed for applications handling very large dynamic distributed graphs. It can be implemented using C++ which allows combining well designed modularity with efficient runtime performance. This enables

- software optimizations at component level,
- policy based parameterization and configuration at compile time and runtime, and
- high portability to new hardware and systems architectures. The Silver runtime will support advanced dynamic application frameworks such as Charm++.

8.5.2.4 Operating System

The Silver operating system strategy is a two-prong approach with 1) a limited-scale extended conventional path based on emerging community-wide multicore Linux solutions, and 2) a second transformative approach to ultra scalable operating system design based on synergistic integration of lightweight kernel modules; the latter called "Silver-OS" or "SOS". The first permits early integration of the Silver runtime system on conventional platforms to support the directed graph oriented advanced programming models and applications for near term use and experimentation. The second will empower ultra scalability to hundreds of millions of cores of diverse structure and operational modalities for extreme scale sustained performance implemented with evolving semiconductor and optical technologies culminating with nano-scale devices. Integration of incremental enhancements of Linux will be achieved by the runtime and compiler layers.

Silver-OS is a new strategy to provide global unification of system-wide services such as memory management, thread management, and global communications for systems comprising billion-way scale parallelism. The innovative concept to be developed and applied is that of synergistic protocols between like services on separate compute nodes. This synergistic control model will enable the synthesis of the physically disparate elements of like functionality to be logically integrated into single global functions that may be dynamically managed and globally optimized in their allocation and operation. The overriding transformative benefit from this unique strategy is its scalability with fixed design complexity. Thus global unification is accomplished and is logically provided to the user through local interfaces among global functional services rather than global interfaces among local functional services. This revolutionizes the system support for user programs and parallel programming language design for scalable systems to potentially hundreds of millions of cores required for Exaflops scale system operation. Distributed global functionality to be realized through synergistic lightweight kernel agents include:

- distributed global address space allocation and address translation,
- parallel process multiple locality assignment and environment context management,
- parallel thread instantiation, suspension, and context switching,
- active message (parcels) creation, routing, buffering, and acquisition,
- I/O including interactive channels and file system interface, and
- process isolation and protection.

The Silver OS will use these lightweight distributed interoperative agents to support a POSIX equivalent API with lower overhead and greater scalability as well as provide the more dynamic functionality required by future generation programming models (*e.g.,* Silver-Threads) and applications.

8.5.2.5 Silver Architecture

Silver is a conceptual system (hardware and software stack) devised to support parallel computation guided by the governing principles of the Silver execution model of computation to effectively exploit future trends of enabling technologies. The Silver architecture that is hypothesized as a starting point for exploration is based on the Silver execution model and is envisioned as a heterogeneous structure of two classes of micro-architectures optimized for two modalities, or operating points, respectively. Like conventional architectures, temporal locality is a dominant dimension.

But while conventional architectures assume good temporal locality and commit the majority of its resources to cache hierarchies, the Silver architecture recognizes that computations exhibit disparate operational modalities. Examples of high temporal locality compute elements include the Stanford Merimac, the UT-Austin Trips, and the Cray X1 PVP. A low/no temporal locality class of architecture elements considered as part of the Silver stack architecture layer is an advanced PIM architecture to accelerate data intensive computation, such as dynamic directed graph processing, exhibiting low temporal locality resulting in little data reuse and poor cache behavior. The Silver-PIM is a lightweight multi-threaded core that exploits the low latency and high bandwidth achieved through direct access to the wide row buffer of memory banks. Silver-PIM incorporates distributed global address space (DGAS) virtual to physical address translation, and message-driven thread instantiation with efficient compound atomic operations on structs for efficient local control object synchronization overhead. It supports the message-driven work-queue method for Silver execution Model and messages for system-wide intrinsic latency hiding.

Chapter 9

Conclusions

There are several reasons for paying attention to software in the development of Extreme Scale systems. First, the extreme scale systems that are projected for the 2015 – 2020 timeframe are dramatically different from today's Petascale systems and will require correspondingly fundamental changes in the execution model and structure of system software (both of which have remained relatively stagnant during the last two decades). Second, while there has been significant innovation at the hardware and system level for today's Petascale systems, previous approaches have not paid much attention to the co-design of multiple levels in the system software stack (OS, runtime, compiler, libraries, application frameworks) that is needed for extreme scale systems. Third, while certain execution models such as Map-Reduce in cloud computing and CUDA in GPGPU data parallelism have demonstrated large degrees of concurrency, they haven't demonstrated the ability to deliver 1000× increase in parallelism to a single job with the energy efficiency and strong scaling fraction necessary for Extreme Scale systems.

The starting point for this study was the characterization of Exascale systems in the prior hardware study on "Technology Challenges in Achieving Exascale Systems" [62], summarized in Chapter 2 of this report. In this study, we identified Concurrency, Energy Efficiency and Resiliency as fundamental challenges for Extreme Scale software, and focused on the first two. (The third challenge, Resiliency, is addressed by a companion study.) The Concurrency and Energy challenges are further exacerbated by the lower bytes/ops ratios and the dominant contribution of data movement to energy costs expected in Extreme Scale systems. We observed that the Concurrency and Energy Efficiency challenges have to be addressed at all levels of the software stack, and in conjunction with hardware interfaces and software-hardware co-design.

To better understand the software challenges for Extreme Scale systems, we first introduced a set of desiderata for an Extreme Scale execution model and defined metrics that can be used to compare different software stacks for Extreme Scale systems using *energy-delay* as the foundational metric (Chapter 3). We then studied the challenges and implications in developing applications for extreme scale computing by examining multiple application classes including traditional HPC applications, coupled models, data-intensive, data mining, and real-time applications (Chapter 4). From an application viewpoint, the Concurrency and Energy challenges boil down to the ability to express and manage parallelism and locality in the applications. This chapter concludes that applications can be enabled for exploiting extreme scale hardware by exploring a range of *strong scaling* and *new-era weak scaling* techniques, but only with suitable attention to efficient parallelism and locality.

Given this context, Chapter 5 summarized the challenges in *expressing* parallelism and locality in Extreme Scale software. One of them is the ability to expose all of the intrinsic parallelism and

locality in an application, so as to make the application *forward scalable*. Another is to ensure that this expression of parallelism and locality is *portable* across vertical and horizontal dimension. Additional challenges include *composability* of parallel programs, support for *algorithmic choice* across scale, and support for *heterogeneous hardware*.

The challenges in *managing* parallelism and locality are discussed next in Chapter 6. Since the Operating System provides the foundation for the software stack, a lot of attention was devoted to limitations in current OS structures in addressing Extreme Scale software requirements. OS-related challenges include *parallel scalability*, *spatial partitioning* of OS and application functionality, *direct hardware access* for inter-processor communication, *asynchronous* rather than interrupt-driven events, and *fault isolation*. There are additional challenges in *runtime systems* for scheduling, memory management, communication, performance monitoring, power management, and resiliency, all of which will be built atop future Extreme Scale operating systems. The chapter concludes with challenges in *compilers* and *libraries* for Extreme Scale systems.

Programming tools play an important role in making the development of parallel software more productive. Though tools were not directly in the scope of our study, we recognized from the start of the study that any software stack for Extreme Scale systems must be capable of supporting the tools that we envision will be shipped in that time frame. To that end, Chapter 7 identifies a number of challenges in supporting Extreme Scale tools including *performability*, *scalability*, *abstraction*, *adaptation and autotuning*, *multilevel integration*, and *availability and portability*. This chapter also lists some of the key technologies that will be necessary to address these challenges, as well as six different scenarios that can be used to evaluate the effectiveness of an Extreme Scale system in supporting tools.

Chapter 8 outlines key elements of a technical approach for Extreme Scale software. The aim of this chapter is to provide examples that are indicative of the kind of software technologies that will be needed to address the Concurrency and Energy Efficiency challenges of Extreme Scale systems, without prescribing specific solutions. Section 8.1 highlights the importance of *software-hardware interfaces* in an Extreme Scale system. Section 8.2 goes one step further and identifies opportunities for addressing Concurrency and Energy Efficiency challenges through *software-hardware co-design*. Section 8.3 discusses the importance of *deconstructed operating systems* in the future OS roadmap for Extreme Scale systems. Section 8.4 presents a vision for Extreme Scale system software based on the notions of a *global OS* and *self-aware computing*. Finally, Section 8.5 describes an example execution model and technical approach, in an effort to encourage the community to think of breakthrough approaches for building Extreme Scale software.

Appendix A

Additional Extreme Scale Software Ecosystem Requirements

This chapter summarizes requirements from other components of the software ecosystem that future Extreme Scale software stacks need to be aware of. However, the core technologies needed for these components are considered to be beyond the scope of this study. Many of these topics are being addressed by separate studies and research programs.

A.1 Real-time and Other Specialized Requirements in Embedded Software

Terascale embedded systems based on extreme scale technology will share some of the software stack elements required by departmental and data center systems (Table 8.1), but will also have additional specialized requirements not needed in the other systems. For instance, Figure A.1 (taken from [62]) illustrates the relative importance of technology gaps in different classes of extreme scale systems. This figure suggests that embedded extreme scale systems will need many of the same power and concurrency management features of the software stack as the larger systems, but may not require as many resources devoted to resiliency management. The difference in requirements at the embedded scale was not addressed during this study, however, some of the potential differences are summarized below. Additional studies are under way to identify and characterize requirements particular to the embedded scale in more detail.

The primary additional requirements for embedded systems are *real-time* constraints and restrictions on the deployable form factor. Embedded systems are frequently created for scenarios in which they operate on data that is a direct observation of an external system, obtained via sensors. Often, the external system under observation is an independent physical system that can not be slowed down or accelerated. For a given sensor time resolution, this results in a flow of observed data that is input to the embedded system at a fixed rate. The behavior of the observed system therefore imparts a "real-time" requirement on the embedded computing system. For some applications, it is possible to lower the rate of input to the embedded computing system, resulting in outputs that are of degraded value. Depending on the mission of the embedded computing system, this can impose throughput or latency requirements, or both. The embedded system must perform the desired calculations at a rate that consumes the input data at least as quickly as they are supplied. If the computing is implemented as a pipeline, then each stage of the pipeline must be capable of keeping up with its own data input rate. Some applications (*e.g.,* control, or interactive applications) have an additional latency requirement. For these, throughput high enough to keep

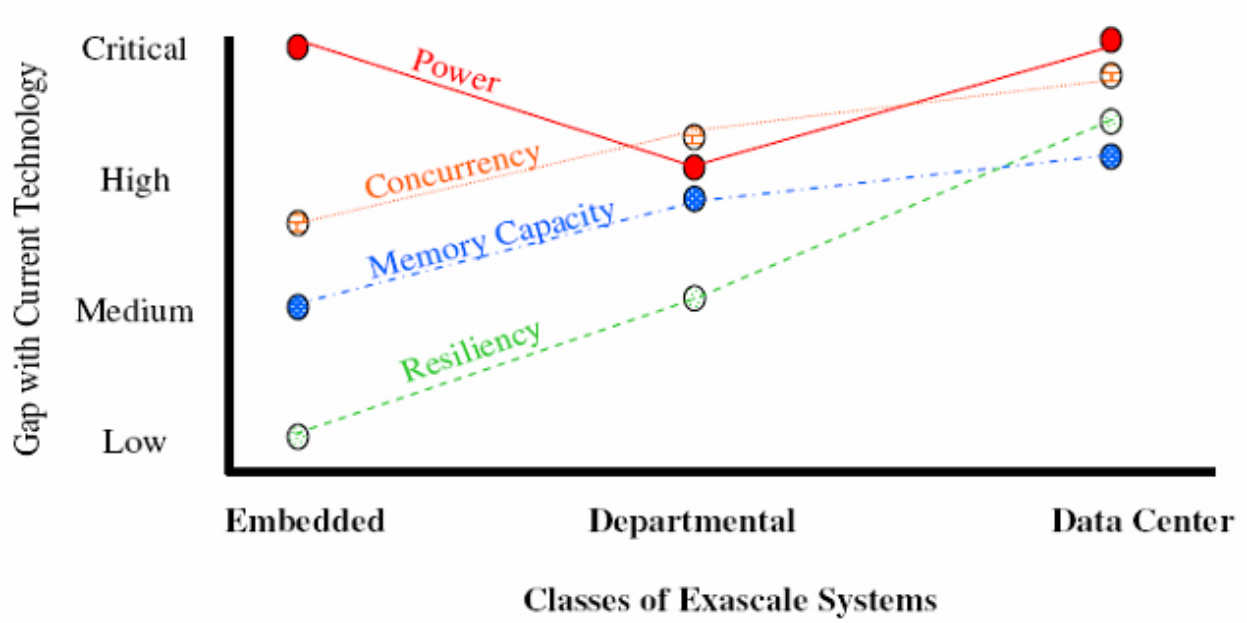

Figure A.1: Technology gaps in Exascale System Classes [62]

pace with the source data is not sufficient — the embedded computing system must also react to its input and produce a stimulus to the external system within some deadline in order to ensure proper behavior of the overall system in which it is embedded.

An important implication of real-time requirements is that, for a given functionality, the utility function for the embedded system versus throughput or latency becomes a step function. This is in contrast to the departmental and data center systems considered in this study. If the throughput is below the minimum required to meet performance goals, or the latency above the maximum, the system fails and has no utility. In many cases, if the throughput is higher than the minimum or the latency is below the maximum, no additional utility is created. In contrast, a non-real-time system still has utility if it falls somewhat short of performance goals, and generally increases in usefulness as its performance increases beyond the goals.

In addition to real-time requirements, embedded computing systems often need to restrict or minimize one or more aspects of the deployed form factor, such as volume, weight, and power consumption. For example, a computing system intended to fly on an unmanned aerial vehicle (UAV) would be constrained in all three aspects. The physical space available to the computing system may be limited, excess weight will reduce the range and endurance of the UAV, and excess power decreases the power available to other subsystems while increasing the heat that must be dissipated. As a result, the performance of embedded computing systems is frequently described not in Floating Point Operations per Second (FLOPS) but in FLOPS per Watt, FLOPS per cubic meter, or FLOPS per kilogram.

The step function utility of embedded computing systems' performance, along with the presence of form factor costs and constraints creates optimization and constraint spaces that differ from the departmental and data center systems considered in this study. For example, a computing system in a UAV that is able to exceed its time performance requirements is no more useful than one that strictly meets its requirements. For such a system, if it is possible to give up some excess computing speed in order to reduce power or space requirements, the utility of the system is improved by doing so. Holistic optimizations are necessary to maximize the overall utility of the embedded

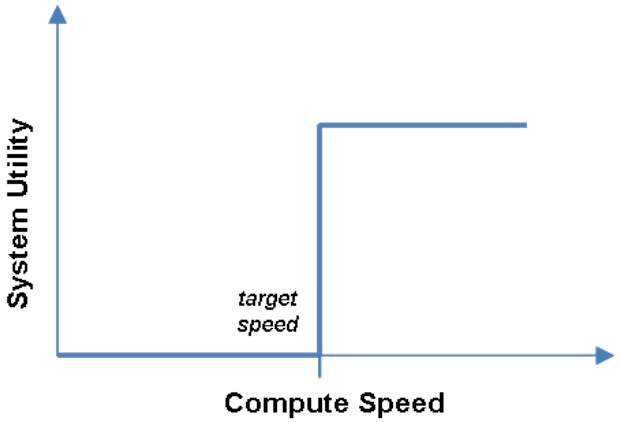

Figure A.2: Notional utility function of real-time system

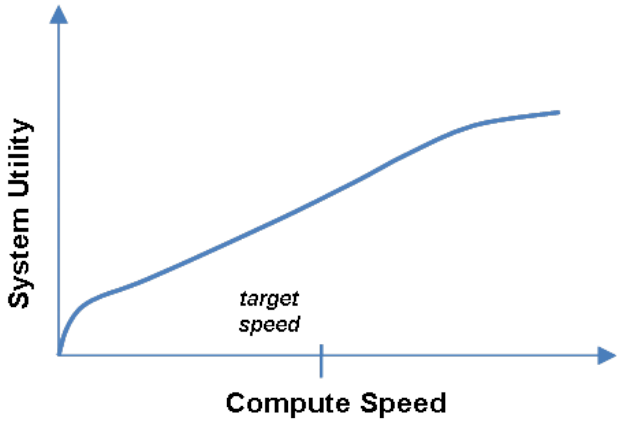

Figure A.3: Notional utility function of non-real time system

computing system. The software stack of current systems does not facilitate the automation of this optimization and constraint satisfaction. Point solutions are generated by human intervention, with coarse attempts at optimization that can, at best, locate local maxima for utility.

The ability to optimize for additional dimensions such as power, weight, and volume, as well the ability of the software stack to solve for specific constraints in each of the optimization dimensions is outside the scope of this study. Adding these capabilities to the proposed extreme scale software stack would require several additional technical improvements, including at least:

1. A formal, analyzable method for describing the constraints of elements and compositions of software, in multiple dimensions, including size, weight, power, and time.
2. A formal method to describe the utility of each dimension of the optimization space.
3. Ability at all layers of the software stack to optimize for size, weight, power, and time in arbitrary combination, as required, while simultaneously satisfying constraints in one or more of the dimensions.
4. A hardware/software co-design layer that allows the selection and configuration of computing elements to support a system.

Improvements in hardware and software for extreme scale are likely to provide direct benefit to the existing software stack for embedded systems. For example, the need for extreme scale hardware and software stacks to optimize for power as well as execution time will improve the utility of embedded computing systems that use them. Previous studies indicate that energy efficiency will be a dominating factor in the design of extreme scale computing platforms. Computing platforms designed to meet the energy efficiency needs of extreme scale systems will be attractive platforms for embedded computing systems. For such platforms, different operations may have widely different energy costs — for example, the cost of a memory fetch from the most distant intra-node memory space would expend much more memory than a floating point add of two registers. Given sufficient information about the energy cost of each operations, elements of the software stack can optimize for power as well as execution time. This ability will be necessary for extreme scale systems, and will likewise improve the utility of embedded systems.

A.2 Tools and Development Environments

A.2.1 Performance Tools

Given the complexity of Extreme Scale software, the role of performance tools must be expanded beyond showing the application programmer a set of measurements of what happened, towards synthesizing all the clues from a variety of sources and pointing the application programmer towards solutions. Extreme Scale performance tools must provide a visual presentation of what happened during program execution, and relate certain behaviors back to corresponding pieces of the code.

In support of this capability, data mining algorithms must be developed to detect anomalous behavior, and to the extent possible, this must be effective during at-scale production runs of extreme scale applications. An integration of capabilities across the software stack, as described in Chapter 7, is also needed to provide a set of "knobs" that the programmer can turn to adjust performance that can be guided by results from the performance tools.

As discussed in Chapter 7, autotuning technology should progress in support of a broader range of optimizations. Today's autotuning technology is largely focused on locality and instruction-level parallelism, with support for multi-core code generation really just getting started. This technology must expand beyond kernels to portions of full applications, hierarchical parallelism, management of data movement, communication and parallelism tradeoffs, and performance and power tradeoffs. Research towards these expanded goals will necessarily develop domain-specific pruning heuristics and scalable empirical search techniques that make it feasible to evaluate such an expanded search space.

A.2.2 Correctness Tools

We describe correctness tools that include debuggers, analysis to proactively improve reliability, formal verification and validation.

A.2.2.1 Debuggers

The most commonly used correctness tools are debuggers, which present an interface to the application programmer to view the execution state of their application as it is running. Debuggers for parallel architectures expand the capabilities of sequential debuggers by (1) pinpointing race conditions between threads on memory accesses; and, (2) identifying communication errors in message passing code.

Debugging at extreme scale is daunting. After decades of parallel computing research, standard practice still involves fairly primitive means of examining an application's execution state that do not scale beyond the standard 32 or fewer processors in most SMPs. With message passing codes, debugging focuses on examining communication streams and incorrect placement of barriers, but many bugs are timing sensitive and therefore intermittent. It is simply not realistic to debug each thread as a separate unit. Aggregating data for the application programmer across all threads often obscures information. Some middle ground is needed to group threads together, and to incorporate information from other correctness tools described below to focus attention on specific pieces of the computation.

New technology to increase productivity of developers during debugging of their code will rely on new sophisticated techniques and support from the software stack to make these techniques feasible at extreme scale. The remainder of the discussion on correctness tools focuses on ways to prevent or pinpoint errors through sophisticated static and dynamic analysis of the application's execution.

A.2.2.2 Tools to Increase Reliability

Over the past decade, a large body of research and several commercial tools seek to increase the reliability of software proactively rather than in response to observed errors. The bulk of these tools look for errors in sequential code, such as memory leaks, buffer overflow, array out-of-bound errors, and similar software defects that may lead to intermittent errors. For parallel codes, analysis tools are used to detect memory race conditions, and various communication and synchronization errors. These techniques rely on a combination of static analysis and run-time testing coupled with lightweight instrumentation of the source code.

As discussed in Chapter 7, at extreme scale such techniques will be needed, not just in debugging mode, but during production runs of the application to detect errors that only appear at scale.

A.2.2.3 Formal Verification

Formal methods are a broad collection of formal specification and verification techniques to provide a rigorous verification of correctness, as an alternative to ad hoc testing. They include: (1) Formal specification methods that can elucidate critical interactions between hybrid programs; and, (2) Dynamic verification methods that can instrument a hybrid MPI/OpenMP program, and consider all its relevant execution schedules. Formal approaches are superior to traditional testing based methods in many ways. They can help verification tools automatically check for commonly committed mistakes without requiring users to create custom-made test harnesses. Many codes break only when ported to new platforms where a hitherto un-attained process interleaving suddenly manifests. Formal techniques can help identify these 'relevant but elusive' interleavings based on action dependence information (*e.g.,* access to common locks). Together with program instrumentation and execution control, dynamic methods can help ensure that these interleavings are considered. Last but not least, they help erect pedagogical foundations necessary for training engineers and researchers with superior skills.

MPI program verification tools (*e.g.,* [141]) are currently used to verify the absence of deadlocks, resource leaks, and communication races, and in optimization, to detect and eliminate functionally irrelevant barriers. Going into the Extreme Scale computing arena, the importance of formal methods is bound to escalate significantly. Given the sheer scale of Extreme Scale computing system designs, there will be an increased use of different programming models all the way from core-to-core communication protocols to middleware that manages multi-problem integration. Handling a plethora of such models in a seamless way, and allowing programmers to pursue efficiency while still providing multiple safety nets, are open challenges, all needing the use of formal methods. The increased propensity for faults (both hardware and software) to propagate unchecked coupled with the infeasibility of taking global checkpoints to restart failed simulation runs all calls for a judicious combination of formal methods. To be effective for Extreme Scale computing systems, we will need a combination of many formal methods technologies. In addition to static and dynamic analysis techniques mentioned above, we will need to consider formal models of compilation, memory model semantics, the correctness of compiler optimizations all the way to determining how often and what to checkpoint, defining an exception handling semantics, and last but not least, how to design adaptive power-down protocols at the hardware and software levels.

A.2.2.4 Validation

As an alternative, or in conjunction with, the formal verification techniques described above, reliability challenges at extreme scale demand new techniques for validating the correctness of a computation. Frequent non-catastrophic failures, coupled with dynamic and possibly non-deterministic

behavior, code optimized by autotuning software, and the hazards of extreme-scale parallel execution increase the likelihood that two instances of the same computation will produce different output. Conceptually, one would want to compare output of a computation or sub-computation to that of some previous and trusted execution. Combined with checkpointing, upon detecting invalid output, the computation could roll back to the previous checkpoint.

Comparing output presents numerous challenges, which is why validation is something that is currently the sole responsibility of the application programmer. Even if the amount of data to be compared is small, floating point differences across different hardware or for different but similar code demand a specification by the programmer of error tolerances. By far the biggest challenge in validating software is the sheer volume of data to be compared. Collection, storage and comparison of large volumes of output is prohibitive. Rather than full-scale comparison, simple techniques like comparing whether an output value is within a particular range, or sampling a small but representative subset of the data are examples of ways that application programmers validate their software, hard-coded into the application itself. An interesting research question is whether the extreme scale software stack can provide general support for validation, both to simplify the code and increase the use of validation to achieve more reliable software.

A.2.3 Visualization and Knowledge Discovery

While visualization is usually considered as part of understanding the results of scientific applications, visualization also plays a unique role in developing new applications, and debugging the results. Techniques from scientific visualization, which are really aimed at drawing insight from large volumes of complex data, may be brought to bear on the significant data collection and analyses involved in managing performance, power and resilience in extreme scale architectures.

The set of research challenges in this area was the subject of a Department of Energy study group which produced a report entitled, "Visualization and Knowledge Discovery: Report from the DOE/ASCR Workshop on Visual Analysis and Data Exploration at Extreme Scale". The following key findings summarize new research areas from from Appendix A of that study:

- Mathematical Foundations: new algorithms in robust topological methods, high order tensor analysis, statistical analysis, feature detection and tracking, and uncertainty management and mitigation.
- Data Fusion: multi-modal data understanding, multi-field and multi-scale analysis and time-varying datasets.
- Exploiting Advanced Architectures and Systems: in situ processing, data access, distance visualization and end-to-end integration.
- Knowledge-Enabling Visualization and Analysis: Scientist-computer interface, collaboration, and quantitative metrics for parameter choices.

Further, related to the last item and as discussed in Scenario 6 from Chapter 7, in response to results from visualization, the application developer may wish to steer the computation in new directions.

A.2.4 Application and Execution Completion Tools

A.2.4.1 Workflow Systems

Scientific workflows capture the individual data transformations and analysis steps as well as the mechanisms to execute them. Each step in the workflow specifies a process or computation to be

executed (*e.g.,* a software program to be executed, a web service to be invoked). The steps are linked according to the data flow and dependencies among them. Workflows can capture complex analysis processes at various levels of abstraction, and also provide the provenance information necessary for scientific reproducibility, result publication, and result sharing among collaborators. By providing formalism and by supporting automation, workflows have the potential to accelerate and transform the scientific analysis process. Workflows have also become a tool capable of bringing sophisticated analysis to a broad range of users, enhancing scientific collaboration and education. Today workflows are being used in astronomy, bioinformatics, earthquake science, gravitational-wave physics, high-energy physics, and many others.

Extreme Scale workflow systems must be capable of scheduling a workflow hierarchy, formed by individual workflow tasks, entire workflows, ensembles of workflows that form an overall analysis, and workflow pools that represent computations that need to be performed at any given time. Scheduling must also cooperate with data management systems that manage data in the distributed environment at different levels of granularity from on-the-node storage associated with a node on a cluster (site), to site storage, to archival storage. Data management also needs to adhere to constraints of available disk space and policies that communities impose on managing the data life-cycle, exploring the interplay between computation and data management.

Scheduling at extreme scale cannot be statics. As failures in the execution environment occur, new failure recovery techniques need to be developed. Making sure that the computations progress in the face of failures requires sensitive monitoring systems, adaptive scheduling, computation migration, and on-demand data recovery. To minimize failures, new resource provisioning techniques need to be explored. Acquiring resources ahead of workflow execution can assure that processors are available to handle the computational tasks, can make enough disk space available to handle the data needed by and produced by computations, and reserving network bandwidth can help stage-in data when needed and stage-it out reliably and fast enough so that the storage and compute resources are available for the next set of tasks. Finally, more applications are moving into the on-demand, near-real-time performance requirements realm, thus all the computation and data management functions and capabilities need to perform efficiently and reliably.

A.2.4.2 Build systems

Build systems are used to compose source code, libraries and input data into executable applications. While building applications is relatively straightforward on commodity platforms, significant challenges arise at the extreme scale from a combination of experimental hardware, experimental software, portability issues, and simply lack of investment in tools at this scale. The following recommendations are a direct quote from the final report of a 2007 Department of Energy workshop entitled, "Workshop on Software Development Tools for Petascale Computing":

"The current state of tools for program configuration and construction is deplorable. Applications must be built for multiple systems, including perhaps one or more petascale machines. We found that too much complexity results from multiple compilers, operating systems, libraries (and their versions). Common option sets and command-line interfaces are missing. We are concerned that the lack of shared libraries and dynamic linking capabilities on petascale systems currently in development will contribute more difficulties. We recommend consideration of new tools (make is still broken), improved tools (*e.g.,* for managing linking order), and more attention to interoperability of program build tools."

A.2.5 Compilers

While it is obvious that compilers are needed to translate from the programming models discussed in Chapters 5 and 6, we also discuss compilers as tools, since they can and should take on additional roles in the extreme scale regime. All the distinct classification of tools mentioned above rely on compilers, coupled with the run-time environment, for program analysis, transformation, optimization and code generation, and providing an abstraction for the application programmer between the architecture and the application code.

Before looking at how compilers can support tools, we must talk about the need for a fundamental shift in how high-end application developers interact with compilers in support of all the new requirements at extreme scale. A lesson from the past two decades is that it is too ambitious to expect success from fully automatic techniques for mapping high-level application code to high-performance executables. Thus, for compiler technology to be effective, it is essential to engage the application programmer in providing domain knowledge about how tools should interact with their code. Having said this, it is still the case that application programmers have extremely limited means of interacting with compilers. Beyond compiler flags, and pragmas or programming model extensions, application programmers simply treat compilers as black boxes. To change the compiled code, they often must guess at alternative code, optimization flags, and pragmas, and empirically evaluate what produces the best result.

Compilers will continue in their role of optimizing code to exploit architectural features and improve performance. Locality and power optimizations will become a bigger focus of attention for extreme scale, as the potential for impact of such optimizations grows significantly. Given the expected dynamic and difficult-to-predict run-time behavior of extreme scale applications, optimization will increasingly become a dynamic process, and adaptive optimizations that respond to feedback from the run-time environment will grow in importance. Compilers must also provide a portion of the autotuning infrastructure described in Chapter 7 for important computational kernels, for automatically generating a set of alternative code variants and pruning the space of alternatives that are considered. Autotuning frameworks must be extended to consider portions of whole applications rather than computational kernels, consider the tradeoffs between parallelism, locality and power, and optimize more global constructs such as data organization and communication. Compilers must in some cases generate the code for the companion computations, described in Chapter 7, which are used not only to enhance performance but for many other purposes.

In terms of correctness tools, compilers should help application programmers develop correct programs through analysis to detect buffer overflow or memory violations. Compiler analysis is used in formal verification, and in generating validation code, possibly automatically. Reliability can be enhanced by supporting programming models that actually prevent certain kinds of errors. For example, languages such as Java enforce array bounds checking, type systems have been used to obtain high-level intent of the programmer, and functional languages prevent side effects across parallel threads. While such approaches are not currently in mainstream use in the HPC community, largely due to hysteresis and performance concerns, making such techniques efficient will demand new compiler support.

Compilers must support the types of interactions programmers have with the remainder of the development environment. In application completion, compilers may provide the interface for dynamic code selection and parameterized code. In general, compiler technology at extreme scale must provide mechanisms to interact with application programmers at a high-level of abstraction.

Appendix B

Definitions of Seriality, Speedup, and Scalability

Scalability has become a golden term to much of the HPC community: "this hardware is scalable; that algorithm is not scalable" but with little consistent formalism behind its use. This is particularly true when it comes to its application to system software. This discussion tries to shed some light on such a use of this term by relating it to other key terms of serialism, and speedup, and trying to develop some formal definitions. While there is nothing new in the following, we do try to relate it to exascale in a way that may help understand effects of and on software in achieving and maintaining performance.

B.1 Definitions

B.1.1 Parallelism and Concurrency

For this discussion we will distinguish between parallelism and concurrency in a rather precise way. *Parallelism* will refer to physical replication. Thus we can talk about the parallelism of function units such as floating point units (FPUs), cores, sockets (a.k.a. multi-core microprocessor chips), nodes, racks, etc.

Concurrency will refer to the overlap of operations as seen during the execution of a program, and will appear in at least three forms:

- *New concurrency*, C_{new}, will refer to the number of new operations that a program starts in whatever time units are appropriate, as in per cycle or per second. This may be the number of new instructions issued per cycle, or the number of floating point operations started per cycle. We assume that the number of operations started is equivalent to the number completed.

- *Active concurrency*, C_{active}, will refer to the total number of operations that are in some sort of execution at the same time. This reflects the pipelined nature of most function units where an operation is started in one cycle and is in computation for several more.

- *Total concurrency*, C_{total}, will refer to the total number of operations, primarily instructions, that are ready to issue but not necessarily in active computation in hardware. This would include, for example, register files in a multi-threaded core that hold currently ready to run threads that are simply waiting their turn to run on the hardware.

We note that the term "new concurrency" is most closely aligned with typical measures of performance, as in flops per cycle, or instructions per second. Also, unless otherwise specified, we will simply use the term *concurrency* to refer to new concurrency.

B.1.2 Work and Performance Metrics

For this discussion we define the *work* involved with solving a problem as the total number of basic operations (flops, instructions, etc) that need to be executed in its solution. While work and time to solution are clearly related, they are not necessarily proportional; depending on the processor's architecture and the selected problem size, different operations (especially memory references) may take different amounts of time at different points in the computation. Thus a *performance metric* will typically be stated in terms of work per unit of time, such as flops per second, and may be stated in several forms. A *peak performance metric* is the absolute maximum amount of work that can be done in unit time by any algorithm running on some processor, regardless of the algorithm or the implementation of the algorithm in terms of a real program. A *sustained performance metric* is one that is algorithm and problem size specific, and takes into account all the delays that may exist when a real code is run on the system. It is typically the case that sustained performance metrics for real systems running the same code may vary widely with the size of the problem. Problem sizes that are, for example, "cache-resident" will exhibit far higher sustained performance numbers than ones that exceed physical memory, and require time-consuming swaps from disk.

B.1.3 Scalability

The Webster's online dictionary defines *scalability* as either "capable of being scaled" or "capable of being easily expanded or upgraded on demand," and as the noun *scale* as "a graded series of tests or of performances used in rating individual intelligence or achievement". Likewise, as an adjective *to scale* is defined as "according to the proportions of an established scale of measurement". As a verb *scaling* means "to have a specified weight on scales".

B.2 Approximate Inter-relationships

Numerically, if the average latency of the hardware that executes a typical operation is L_{ave}, then to a first approximation:

$$C_{active} = L_{ave} * C_{new}.$$

Likewise, if the average thread can initiate instructions that correspond to on average W_{ave} of the above operations at a time, then to initiate C_{new} operations per time unit requires C_{new}/W_{ave} concurrent threads actively initiating instructions. If the longest latency for any such operation is L_{max} (typically a memory access in the hundreds or thousands of cycles), and the fraction of all operations that are such is f_{max}, then using Little's Law the number of such operations that are active at any one time is:

$$(C_{new} * f_{max}) * L_{max}$$

This means that the number of active operations of this type that must be managed by a single active thread is:

$$(C_{new} * f_{max}) * L_{max} \ / \ (C_{new}/W_{ave}) = W_{ave} * f_{max} * L_{max}$$

Managing a long latency operation such as a remote memory reference typically requires some hardware support in a processing core such as an entry in a Load/Store queue that must be dedicated to that operation for its duration. We assume that there are Q such entries in a core. Thus, once a thread has filled all such Q entries, no further such operations can be issued, and when one is found, the core must *block* the thread, regardless of whether or not the thread still has issuable instructions. If Q is much less than the above number of outstanding operations (which is usually the case) then the only way to maintain the C_{new} level of concurrency is to have other threads ready to run. This can be done either by adding more processing cores to the system (and assume each goes idle frequently) or to *multi-thread* each core. This core or thread multiplier is thus:

$$(W_{ave} * f_{max} / Q) * L_{max}$$

As a numerical example, if we assume that Q=4, W_{ave}=4, f_{max}=1%, and L_{max}=1000, then the above multiplier is 10.

We note that this is strictly a lower bound, since it assumes that a typical thread's program is in fact capable on average of finding at least Q independent operations that can be issued without a data dependency. We can thus bound C_{total} as:

$$C_{total} \geq (W_{ave} * f_{max} / Q) * L_{max} * C_{new}$$

B.3 Algorithmic Scalability

In high performance computing, scalability has had a long history of controversy as to its meaning and relevance [81]. We will try to be explicit by using time to solution as the basis for defining scalability, not as a general property of a system, but as relevant to only one application at a time. In this section we define $T_{A,X}(N, P)$ as the time to solution for some algorithm A when converted into a program and executed on some architecture X, where the size of the input in some units is N, and the number of "processors" allocated from X to the program execution is P. We call the combination of A and X as the *system configuration*.

For simplicity we will drop the subscripts A,X for the rest of this discussion. However it is critical to remember that all of our discussions are, in fact, relevant only to a particular system *and* a particular application.

To help make the following discussion more visual, we will develop a series of 3D plots of T(N,P) as in Figure B.1 where the horizontal *P axis* is P — the number of processors, the *N axis* extending back into the picture is N — the size of the problem instance, and the vertical *T axis* is execution time in units of T(1,1) (the time to solve a problem instance of minimal size 1 on a single processor).

Within such charts, a time of C is equal to an absolute execution time of C*T(1,1). To make the graphs easier to explore over large variations in any direction, we assume logarithmic axes; thus the origin represents unit execution time with N=1 and P=1. In particular, N=1 here represents the smallest problem size for which the time complexity curve is "relevant," and is up to the user to define.

We observe that a plot of T(N,1) represents the classical execution time curve for a single processor, and lies wholly on the NT-plane as pictured in Figure B.1. We call it the *sequential time complexity curve*. It also starts at the (1,1,1) origin. As problem size increases (larger N along the N axis), the execution time increases. The notional curve in the figure is for some algorithm with greater than linear time complexity on the assumed architecture.

In the following subsections we now define two classes of scaling, depending on whether or not N is allowed to change.

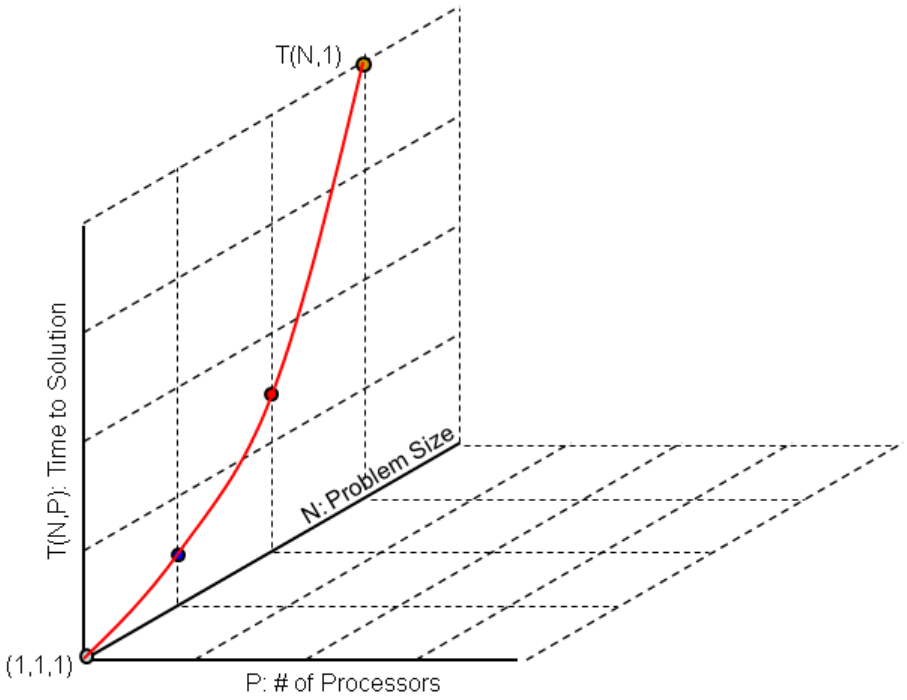

Figure B.1: Dimensions of Scalability

B.3.1 Strong Scaling

A system configuration A,X is said to exhibit *strong scaling* if, when we hold the problem size N constant, the time to solution decreases as P is allowed to increase, that is

$$T(N,P_1) > T(N,P_2) \text{ if } P_1 < P_2.$$

We say that the system exhibits *perfect strong scaling* if:

$$T(N,P) = T(N,1)/P \ \forall \ P > 0.$$

In this latter case, growing the number of processors employed by some factor decreases the execution time by the same factor. Under normal circumstances, this is as good as one can expect.

Figure B.2 extends the prior Figure B.1 to reflect this concept. The curve in the PN plane is the mirror image of that in the NT plane, and represents perfect strong scaling where just enough processors (namely P = T(N,1) processors) are engaged for a problem of size N to always keep the execution time at 1 unit — that of the simplest problem on one processor. The lines from the NT plane to the PN plane represent how the execution of problems of fixed size vary as the number of processors increase. These lines also assume perfect strong scaling — that is the execution time decreases as 1/P. Finally, the lines that are in planes parallel to the NT plane reflect how using a system with some fixed number of processors works as the size of the problem changes.[1]

[1]Such curves obviously make sense if continued "below" the (P=1,N=1) plane, as here the solution times are less than 1, *i.e.,* faster than a standard sized problem with only 1 processor.

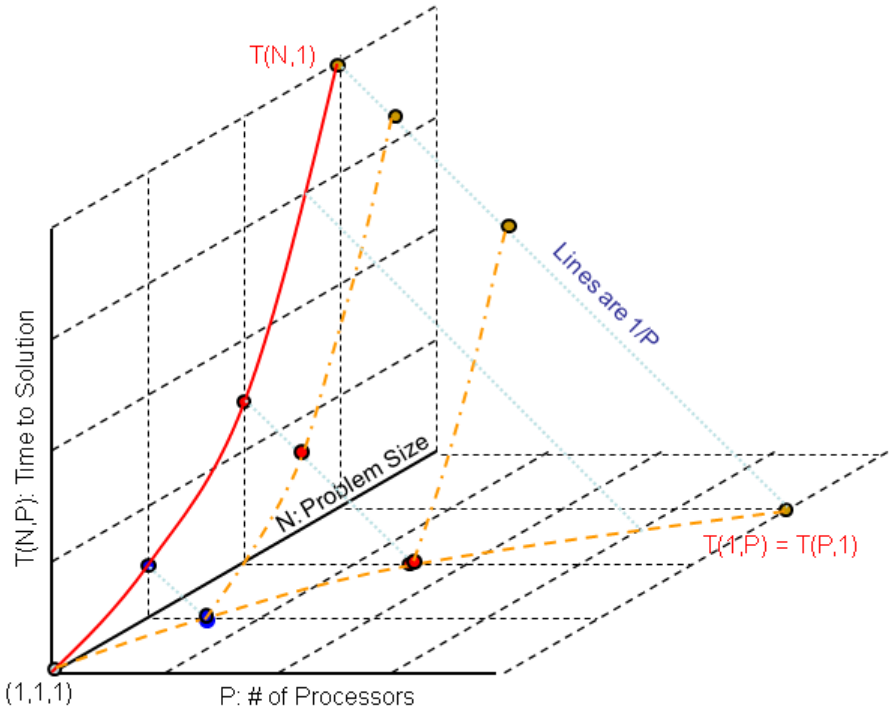

Figure B.2: Strong Scaling

The surface in Figure B.2 is important because it represents in a real sense, an upper bound for implementations of a particular algorithm. Any point on the surface represents a perfect strong speedup for a particular size of problem. As such, any point "behind" this surface is highly unlikely to be achievable (unless it demonstrates *superlinear speedup* – discussed later), without some significant variation in algorithm or architecture. Thus, it is far more likely that in the real world, strong speedups for the specified algorithm are likely to end up "in front of" this surface.

This can be bounded a bit more by adding a surface to Figure B.2 that represents "no speedup" as P increases. Figure B.3 introduces this surface as a projection perpendicular to the NT plane. Any point "in front of" this surface represents systems that "slow down" when adding additional processors (a rare but not unheard of situation). Now, any system configuration exhibiting strong scaling will correspond to a point between the two surfaces pictured.

The concept of work discussed previously also sheds some light on strong scaling. If we assume that the curve in the NT plane — the T(N,1) curve – is in fact proportional to the work needed (a possibly bad assumption when N is large and memory hierarchy effects come into play), and if P processors are capable of P times the peak or sustained performance of one processor, then for a problem size of N we need T(N,1) units of work. Now with P processors, for T(N,P) units of time we have T(N,P)*P units of work available (the "peak" performance potential). However, for perfect strong scaling where T(N,P) = T(N,1)/P we are using (T(N,1)/P)*P units of work — which is exactly all that is available. To put it another way, when perfect strong scaling occurs, the sustained performance of the processing system is independent of the size of the problem, Less than perfect scaling means that some of the processing capability is "wasted."

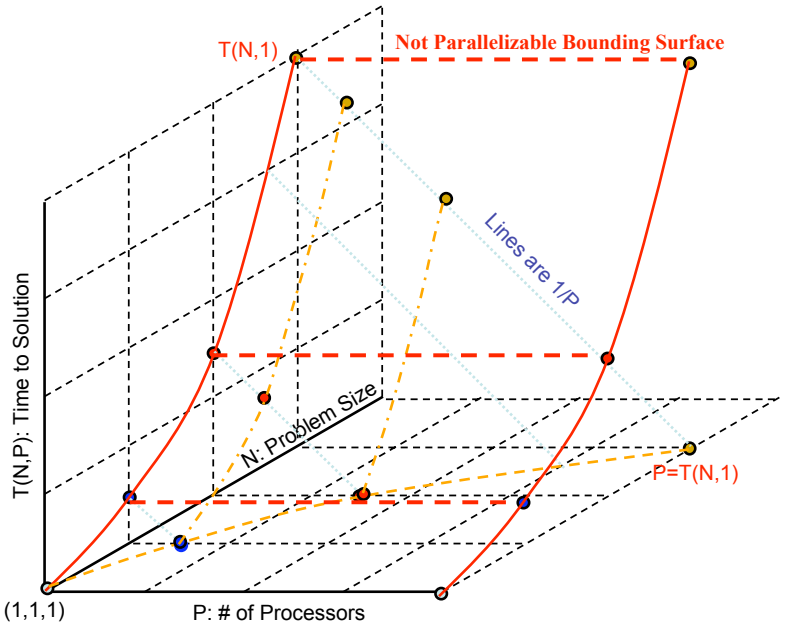

Figure B.3: Strong scaling with bounding surface

B.3.2 Weak Scaling

The other major common class of algorithm scaling is designated as *weak scaling*. While there does not appear to be a formal definition of weak scaling in the literature, the term commonly refers to the case when the amount of work performed by an application increases in proportion to the number of processors. As discussed in Section 4.2.2, the increase in work was traditionally achieved by spatial increase of the problem size but is more recently being achieved by more performing more computation per datum ("new-era" weak scaling).

The importance of weak scaling was stated in John Gustafson's paper on "Reevaluating Amdahl's Law" as follows [74]:

> "One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, the problem size scales with the number of processors. When given a more powerful processor, the problem generally expands to make use of the increased facilities. Users have control over such things as grid resolution, number of timesteps, difference operator complexity, and other parameters that are usually adjusted to allow the program to be run in some desired amount of time. Hence, it may be most realistic to assume that run time, not problem size, is constant."

In weak scaling, the problem size N that can be solved in constant time increases as the number of processors P increases, that is:

$$\text{For any } P_2 > P_1, \text{ there is some } N_2 > N_1 \text{ such that } T(N_2,P_2) = T(N_1,P_1).$$

It is important to realize that strong and weak scaling are *not* necessarily mutually exclusive. Each basically indicates how using additional processing can affect execution time in two different directions — strong in decreasing execution time of a fixed size problem, and weak in increasing the size of the problem without increasing time. It *is not* inconceivable to use additional processing in some way to *both* decrease execution time *and* increase problem size in some intermediate way.

B.4 Speedup

A term used with perhaps even more frequency than scalability is *speedup*. Two definitions from the web include:

- An acceleration, to go faster.
- A measure of how much faster a given program runs when executed in parallel on several processors as compared to serial execution on a single processor.

B.4.1 Defining Speedup

The latter definition leads to the typical definition of speedup as the ratio of the sequential execution time of an algorithm to its parallel execution time. In most cases, what is of real value is not the speedup for a particular algorithm at a particular level of parallelism, but an understanding of speedup as a function of parallelism P. Consequently in concert with our definition of time to solution we define $S_{A,X}(N,P)$ as the speedup for some algorithm A when converted into a program and executed on some architecture X, where the size of the input in some units is N, the number of processors allocated from X to the program execution is P, and the speedup is relative to the time for the same sized problem on a single processor. As before we will drop the subscripts.

One can raise an argument as to whether the numerator (the single processor case) should be assuming the same code as run on multiple processors, or for an optimal code written for exactly one processor (and thus avoiding all the potential overheads introduced to manage parallelism that is not used). For convention, we will assume the former, thus:

$$\text{S(N,P)} = \text{T(N,1)} \ / \ \text{T(N,P)}.$$

B.4.2 Classes of Speedup

As with scalability, there are several common classes of speedup functions. *Non-parallelizable code* is where the speedup is 1 (or less) for all P. *Linear speedup* occurs when $S(N,P) \approx KP$ for some constant K and for a range of P that is of interest. Perfect linear speedup occurs when K=1, that is S(N,P) = P for all P. While *perfect linear speedup* is the "holy grail", several other types of speedups are actually much more common. *Logarithmic speedup* up occurs when $S(N,P) \approx P/(\log_2(P))$. *Fixed overhead speedup* occurs when S(N,P) approaches some constant due to code that does not parallelize, as P approaches ∞. *Superlinear speedup* occurs when the reduction in execution time as P increases is better than the increase in processor count. Figure B.4 places all of these speedup classes in context.

B.4.3 Fixed Overhead Speedup — Amdahl's Law

Fixed overhead speedup typically occurs (or is believed to occur) when some fraction F of a program's execution is purely sequential (and thus cannot be sped up with parallelism), and the rest of the program's execution is sped up as the number of processors employed increases. If we assume perfect linear speedup for the rest of the program, then

$$T(N,P) = F * T(N,1) + (1-F) * T(N,1)/P$$

Expanding S(N,F) to include an extra argument F (for efficiency) yields a speedup of:

$$S(N,P,F) = T(N,1)/(F * T(N,1) + (1-F) * T(N,1)/P) = 1/(F + (1-F)/P) \rightarrow 1/F as P \rightarrow \infty$$

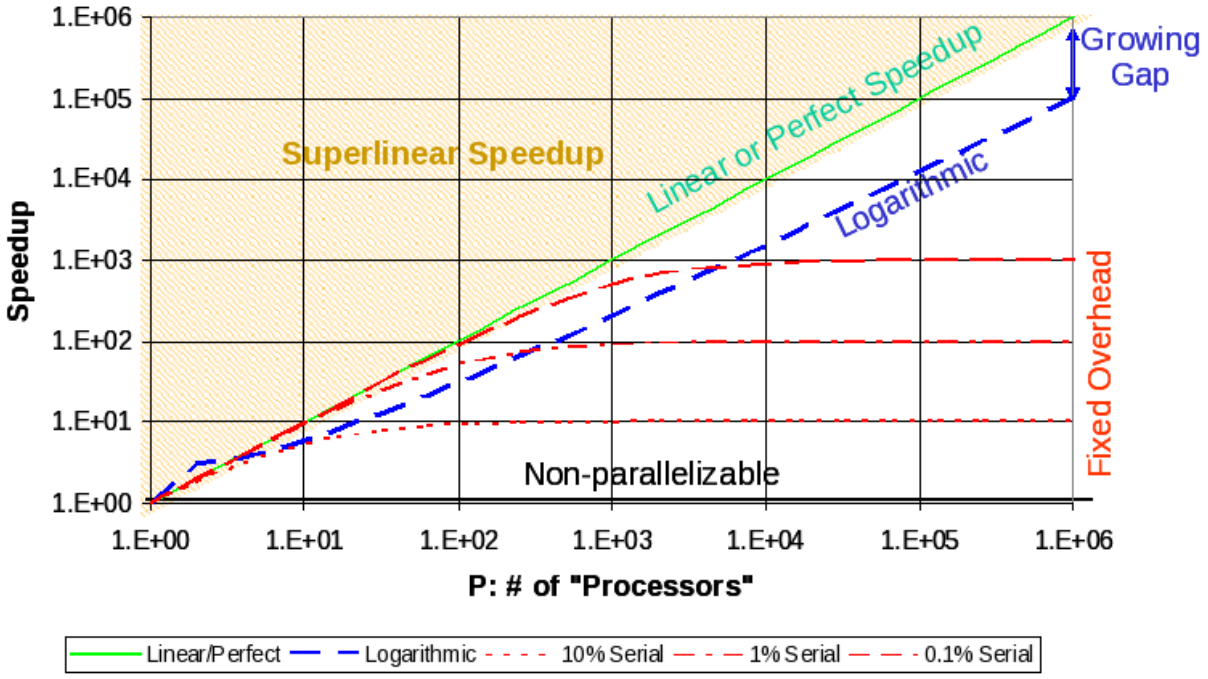

Figure B.4: Classes of speedup

We note the addition of the argument F to denote the percent of sequential time. As P goes towards infinity, this asymptotically approaches 1/F – the reciprocal of the fraction of sequential complexity that cannot be parallelized.

This is also known as *Amdahl's Law*. Figure B.5 diagrams this relationship for a variety of F values. As can be see, if expected parallelism is expected to grow into the billions, the only way to get any useful speedup is to have essentially zero serialization. In this report, we often refer to the "serial part" of an application for simplicity; in practice, rather than a containing a completely serial part and a perfectly parallelizable part, an application is more likely to exhibit different scales of parallelism in different regions of code.

B.4.4 Superlinear Speedup

It is also possible, but not frequent, to have *superlinear speedup*, where the growth in speedup exceeds the growth in processors. In such cases, doubling the number of processors, for example, more than doubles the effective speedup. In terms of slopes, a superlinear curve is one where the slope is greater than one (the slope for a perfect linear speedup).

There are several example situations where such an implausible speedup might actually occur:

- Separating different "loop iterations" to different processors may in fact eliminate loop overhead.
- Increasing the number of processors may in fact increase the total effective "size" of caches near to processing cores, meaning that less latency is foisted on data memory references, and thus each processor gets relatively speaking "faster" than it was for smaller configurations.

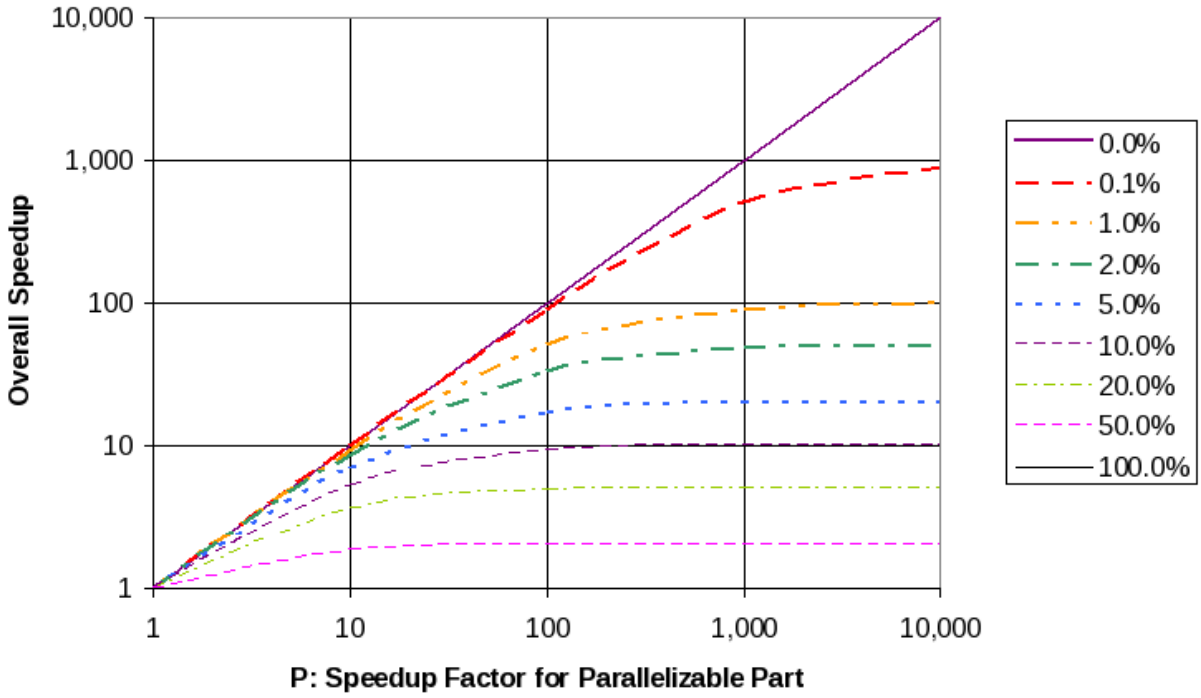

Figure B.5: Amdahl's Law

- Problems involving bounded search, such as alpha-beta search, may actually accelerate faster for larger number of processors because of an increase in breadth-first searching that reveals a tighter bound earlier than it might have in a sequential implementation, resulting in earlier cut-offs of searches across the board, and the reallocation of processors to those depth-first paths that still have promise of a better solution.

B.4.5 Speedup and Average Parallelism and Concurrency

The speedups discussed above, especially associated with Amdahl's Law, all have the implicit view that the P processors are either 100% busy during the parallel parts, or idle during the serial parts. An alternative view is that the actual degree of parallelism – how many processors are actually doing computation at any point in time, is very dynamic, and can change from moment to moment. The tree-like computations associated with parallel prefix algorithms are good examples of this — at one instant they are all busy, then only half, then $1/4^{th}$, and so on. At some time the pattern may repeat.

With this view, we may re-interpret our speedup term as an average "sustained parallelism" metric. A speedup of S(N,P) for some real number of processors P is equivalent to having S(N,P) processors and an algorithm that keeps them busy 100% of the time.

In addition, if we know the average pipeline depth of function units then the product with S(N,P) and the number of function units per processor then gives us a realistic measure of *average concurrency* — the average number of operations that are in some state of computation at any one time. This in turn is thus related to how many *independent* operations must be extracted from the program on each and every clock cycle.

B.5 Efficiency

The above discussion of average parallelism leads naturally into a discussion on *efficiency*. Typical definitions of this term relate it to the percentage of some resource that is effectively used during execution of a program. In common usage, a typical parallel computing efficiency is defined as given some set of function units, say floating point units, over the period of computation time, what percentage actually were used in the computation. For the Linpack benchmarks used as the basis for the TOP500 list, this efficiency is often stated as R_{max} over R_{peak}, where R_{peak} is the peak floating point rate the hardware of a system is capable of, with all other constraints ignored, and R_{max} is the rate that actually was used.

Different resources may have different efficiencies during the same program, and there may in fact be a rather complex relationship between efficiencies of different resources that are "packaged" in the same subsystem. For example, we may discuss the "efficiency" of using processors as a whole in a computation, and yet that efficiency may be different when we look at the use of FPUs within the processors, or of memory bandwidth between the CPU and their local memory chips.

B.5.1 Computing Efficiency

At the top level of most discussions, the key efficiency number deals with the use of processors, and there are several different ways of determining this:

- As the ratio of total operations or instructions actually used, over the total operations or instructions possible in the algorithm's running time if all processors were always busy doing useful work = "sustained" over "peak" in our prior definitions.

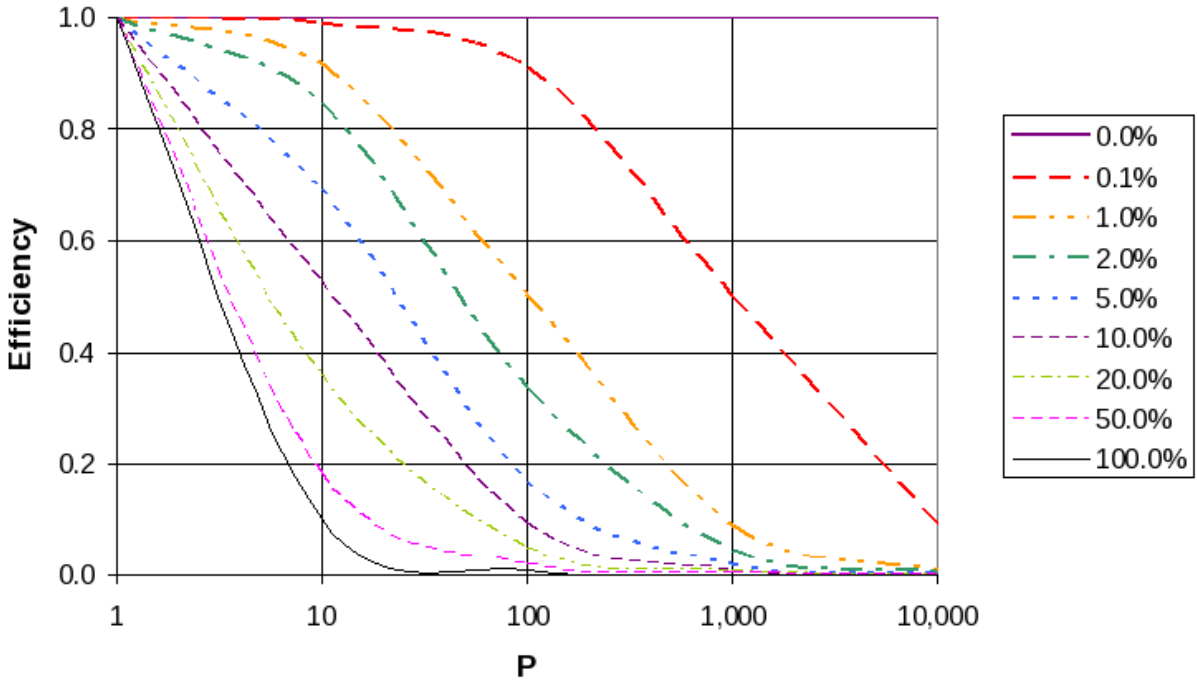

Figure B.6: Efficiency

- As the serial execution time T(N,1) over the parallel execution time T(N,P) times the number of processors P.
- As the average parallelism S(N,P) over the peak parallelism P.

B.5.2 Efficiency and Amdahl's Law

If Amdahl's Law is a reasonable view of how some particular program behaves on some parallel computer system, then we can use the last of the above approaches to compute E(N,P,F): the efficiency of use of processors for a problem of size N, processor count of P, and sequential percent F as follows:

$$E(N,P,F) = S(N,P,F)/P = (1/(F + (1-F)/P))/P = 1/(PF + 1 - F)$$

We note that this equation is independent of the problem size. Figure B.6 graphs this relationship for a variety of F and P values. Again as can be seen, it takes a vanishingly small F value to obtain decent efficiency when there are large numbers of processors.

With this measure a 100% efficiency occurs when all the processors are doing "useful work" all the time, where "useful work" is computation at the same rate as single processor. Also, the lowest possible efficiency is 1/P — the equivalent work of only one processor is being done.

An interesting side bar to the above equation is to ask at what point does the efficiency drop below some arbitrary threshold. Reordering the above equation assuming a desired efficiency E, we get:

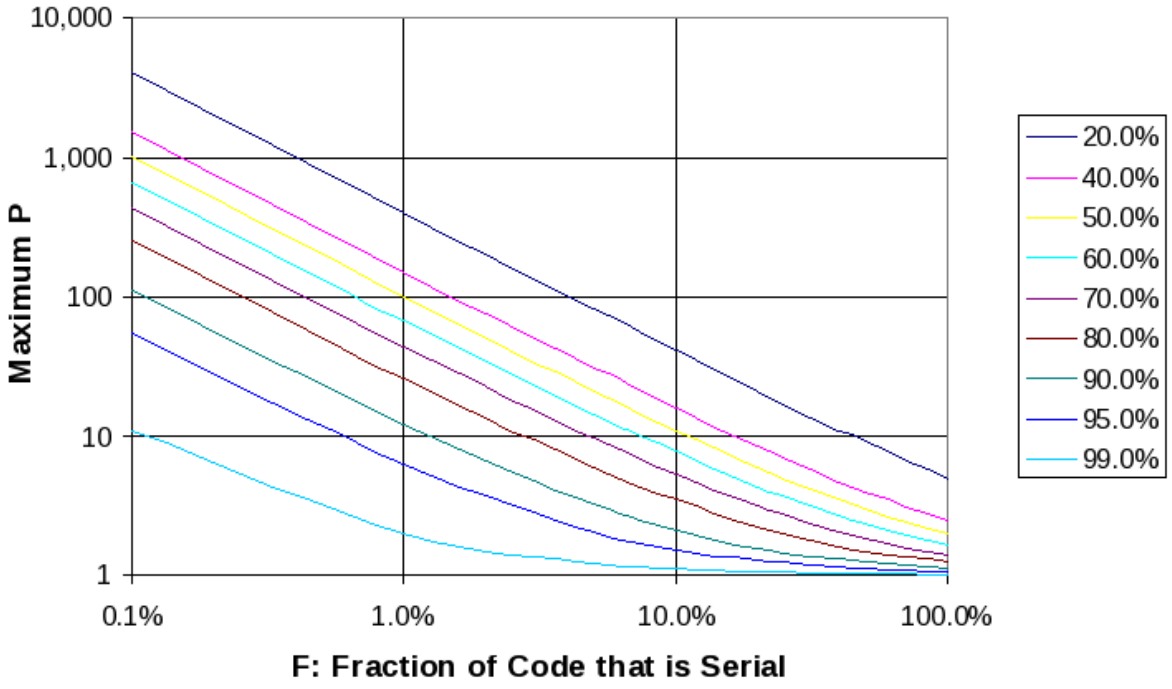

Figure B.7: Maximum usable parallelism for various efficiencies

$$P \leq (1 + EF - E)/(EF)$$

.

Consider, for example, a design goal of at least 50% efficiency. Using this in the above equation yields that to achieve at least 50% efficiency, we must constrain the number of processors as follows:

$$P \leq (1 + F)/F$$

.

Figure B.7 diagrams these bounds as a function of F for a spectrum of target efficiencies.

B.6 More Nuanced Views of Speedup

B.6.1 Gustafson's Law

Amdahl's Law assumes that for any sized problem some fixed percentage of the operations "are not parallelizable." While this may be true for some part of a problem, such as for an outermost loop overhead, it may not be true in general. In fact, it may be that as a percentage the serial overhead changes, and in particular decreases, as the size of the problem is allowed to grow. An example might be some fixed setup code at the beginning of a computation whose length is independent of problem size. Thus as the problem increases, the percent serial decreases, and parallelism is more effective.

This happens enough to be coined *Gustafson's Law*, which states that very often, if the problem size can be increased arbitrarily with sufficient parallelism then an arbitrarily large speedup can be obtained. To set this up as an equation, we assume as before that T(N,P) is the time spent by a parallel processor of parallelism P solving a problem of size N. We assume also that F(N,P) is the percentage of T(N,P) that runs as if it is on a single processor, while 1-F(N,P) is the percentage that runs at the equivalent of 100% efficiency on the P parallel processors. With this we get:

$$T(N,1) = T(N,P) * (F(N,P) + P * (1 - F(N,P)))$$

Converting this into a speedup:

$$S(N,P) = T(N,1)/T(N,P) = F(N,P) + P * (1 - F(N,P))$$

The closer F(N,P) is to zero, the smaller the P needed to reach some arbitrary speedup.

B.6.2 An Example: Fixed Overhead

As an example assume the total program execution consists of a + bN instructions, where a and b are constants, and N is related to the size of the problem We also assume that the "bN" part is perfectly scalable, that is with P processors it takes bN/P time. This corresponds to some algorithm of linear time complexity — a perfect weakly scalable problem. Now the parallel time is:

$$T(N,P) = a + bN/P$$

and the serial percentage is:

$$F(N,P) = a/((N,P) = a/(a + bN/P)$$

As N goes towards infinity, F(N,P) approaches zero.
Plugging this into the previous speedup equation yields:

$$S(N,P) = (a + bN)/(a + bN/P)$$

Figures B.8 and B.9 diagram a case where a = 100b, that is the non parallelizable code is 100 times the size of the parallelizable code. As can be seen, for this problem, any desired speedup can be reached by making N (and P) big enough.

B.6.3 The Karp-Flatt Metric

The process of determining the "serial" portion of a code in advance is usually hard. An alternative is, once a program is running, to fix the problem size at N, run the same code at different degrees of parallelism, and then measure the execution time T*(N,P) (the wall clock time). We now define the effective serial fraction or the Karp-Flatt metric as F*(N,P) as the fraction of time that would still be serial if the parallelism was at 100% efficiency for the rest of the time. We can then write an equation between the measured times T'(N,P) and T'(N,1) and the term F'(N,P) as follows:

$$T'(N,P) = T'(N,1) * (F'(N,P) + (1 - F'(N,P)) * T'(N,1)/P)$$

or solving for F*(N,P):

$$F*(N,P) = (P * T'(N,P) - T'(N,1))/(P * T'(N,1) - T'(N,1))$$

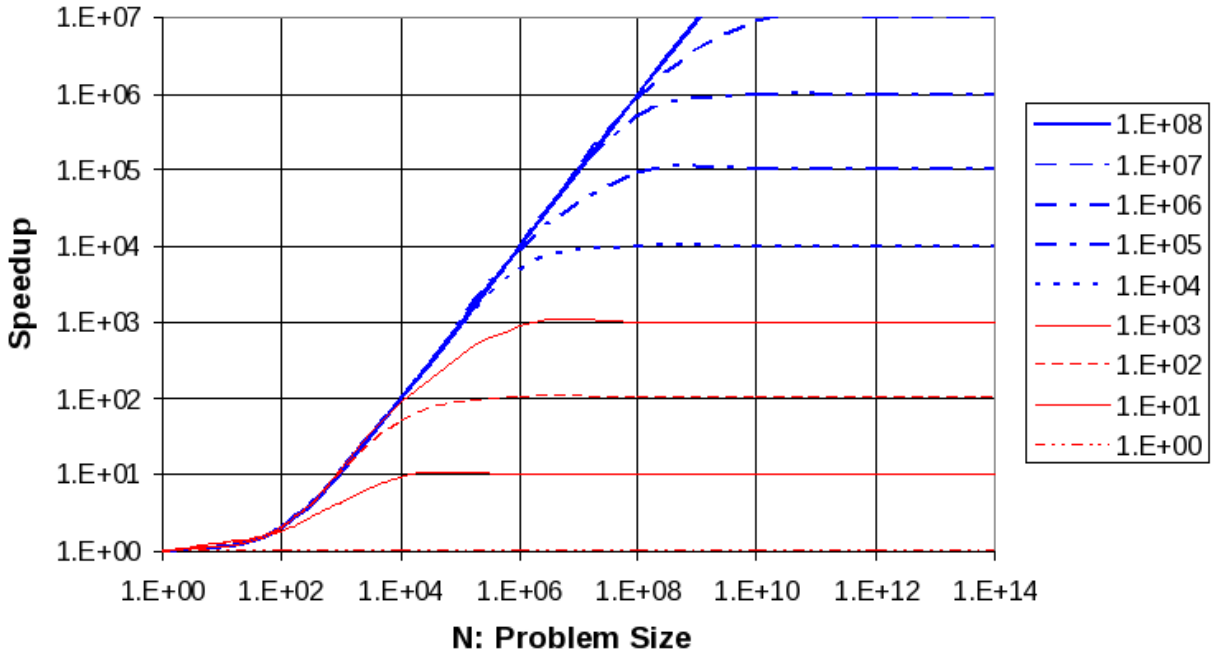

Figure B.8: Fixed overhead: Speedup as a function of N

As an example, Figure B.10 diagrams such calculations for the NAMD 2.5 code on an IBM Blue Gene computer[2]. Figure B.11 diagrams the same data on a log curve. Note that these figures show speedup as a function of the number of processors (P) rather than the problem size (N). The latter curve in particular hints at some interesting properties as declines are interspersed with flat areas. The declines seem to occur as we move up to a drawer full of Blue Gene nodes, and then after passing beyond a board to a full rack. It would be interesting to see if further declines occur when more racks are added.

Also included on Figure B.11 is a synthetic curve of the form:

$$-0.0045 * \log_2(N) + 0.00578$$

As can be seen, this is a rather decent approximation to the observed Karp-Flatt metric up to about 1000 processors, after which it appears the metric flattens at about a 0.13% serial fraction, which, using Amdahl's Law predicts a maximum speedup of about 770.

B.7 Memory and Bandwidth Scaling

Other key parameters of real interest to any scaling discussion are memory capacity and bandwidth; both of which, if inadequate, can constrain either the size of the problems that are solvable or the rate of their solution.

[2]S. Kumar, G. Almasi, and L. V. Kale, "Achieving Strong Scaling On Blue Gene/L: Case Study with NAMD," 2006

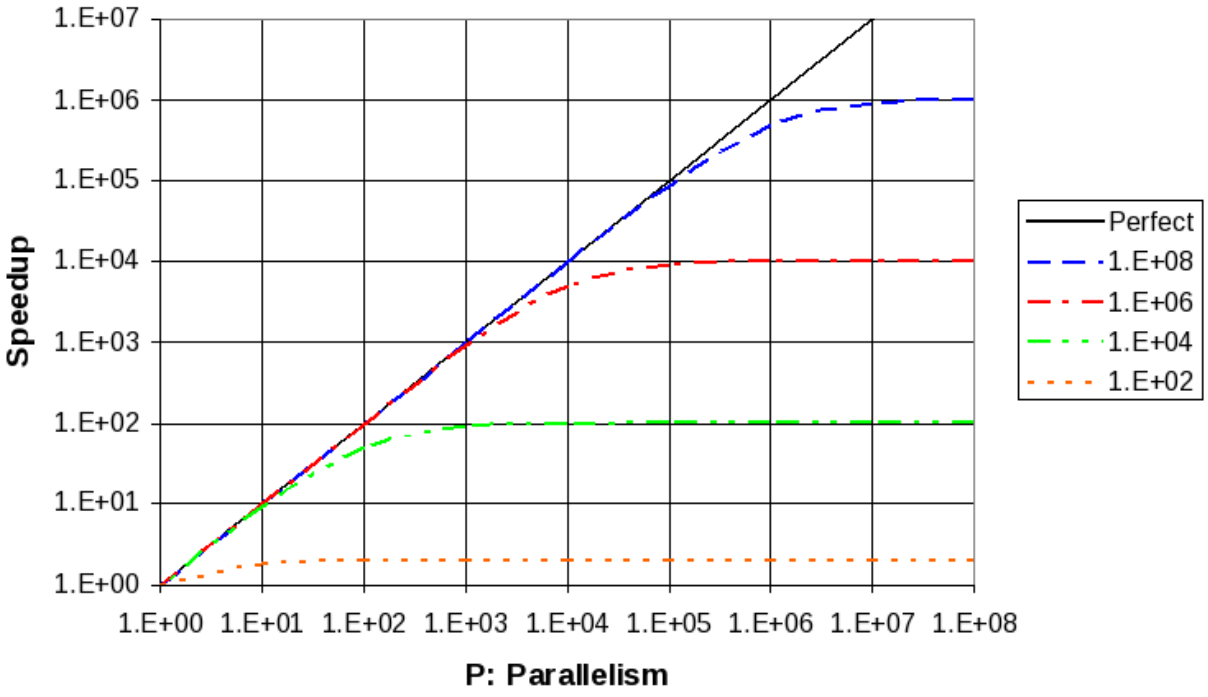

Figure B.9: Fixed overhead: Speedup as a function of P

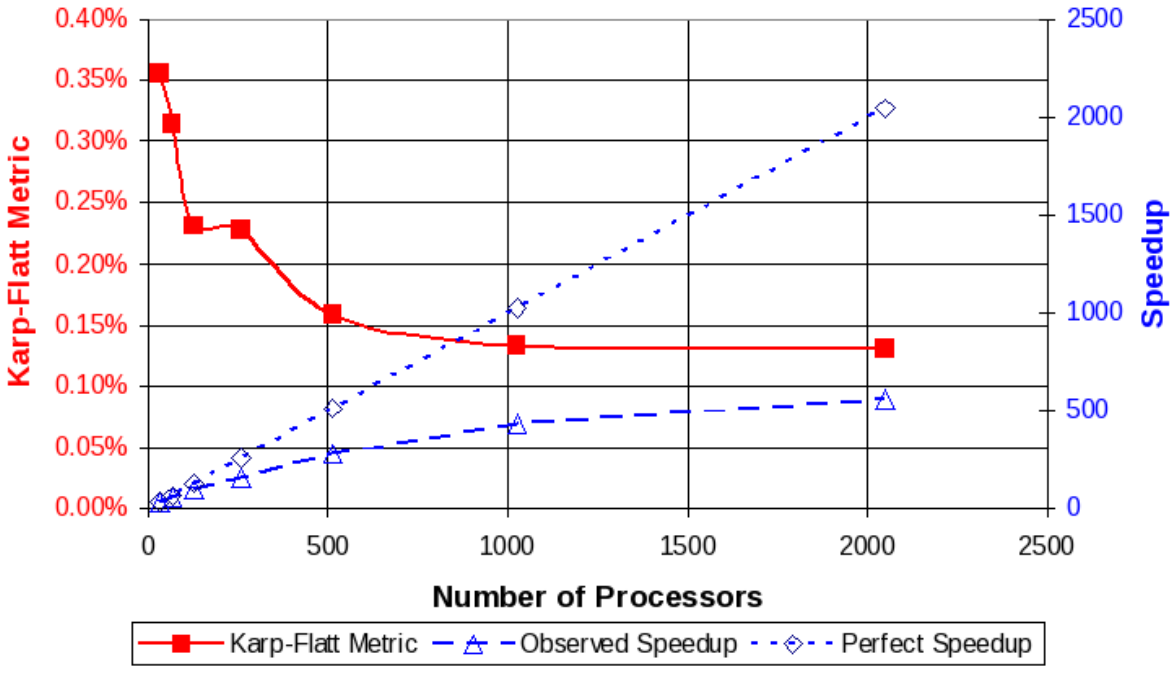

Figure B.10: Karp-Flatt metric for NAMD on a linear scale

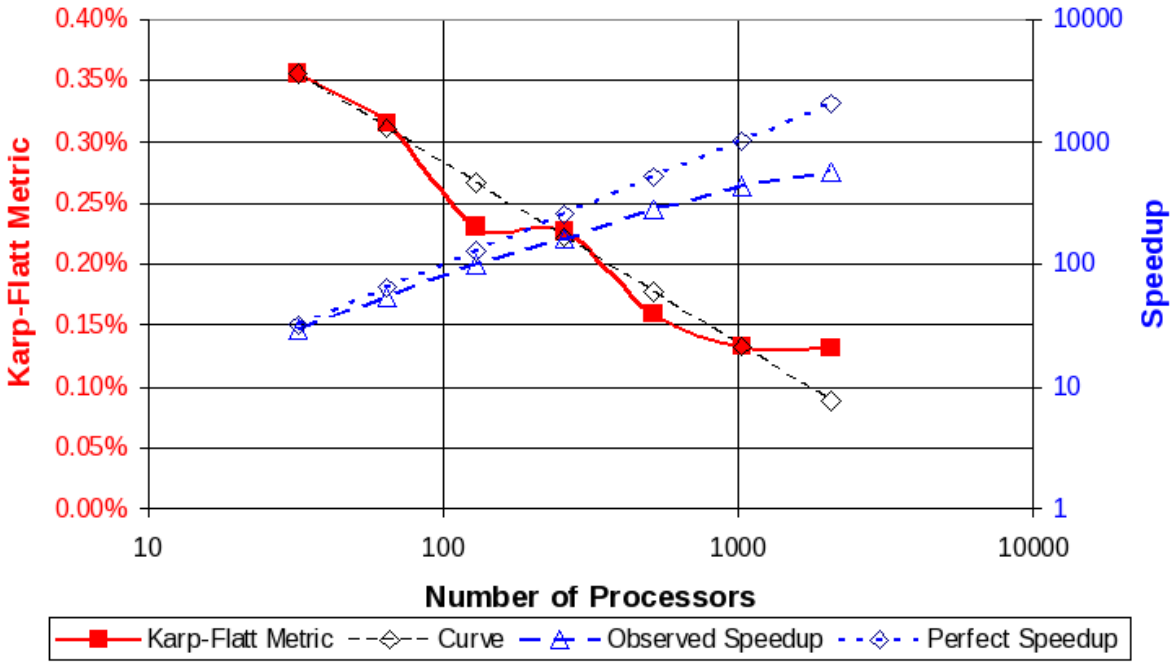

Figure B.11: Karp-Flatt metric for NAMD on a log scale

Because of the way systems are built, total memory capacity, especially for large systems, is directly proportional to the number of processors, since some fixed amount of memory is typically packaged with each processor or fixed set of processors. Available bandwidth, however, is a bit harder to extrapolate, since it is a function of both the individual nodes where the traffic is generated and of the interconnect that ties multiple nodes together. The latter, in particular, is often a function of the number of nodes and the topology of the interconnect.

On a notional level, we might expect at least the memory parameter to track the kind of scaling that the applications they support exhibit. For example, a configuration that exhibits strong scaling keeps the size of the problem constant as the number of processors increase. Thus in an ideal situation, the total required memory capacity is constant, and thus the capacity per processor might drop linearly with the number of processors.

Weak scaling assumes that the problem size increases as the number of processors increase. For perfect weak scaling, the relationship is linear; if problem size and required memory capacity is linear, then one might expect the memory capacity on a per node/processor basis to ideally stay constant as the number of processors increase.

The following subsections explore this in a bit more detail. Several candidate data structures are used as a basis for the study.

B.7.1 System Architecture and Baseline Assumptions

Understanding both of these parameters requires making some assumptions about the system architecture, For this section we assume a partitioned system where each "processor" in the prior sense has an equal-sized chunk of memory. Together we call this processor plus memory combination a

node.

We also assume mechanisms exist to allow any processor to gain access to data in other nodes. Whether this is a "pull" mechanism (as in access via load or store instructions), or "push" (as in message passing from one node to another) depends on the exact system architecture, but for now we simply assume that such access is possible, and that the cost of moving data across such node boundaries is relevant.

For memory capacity, the rule of thumb for decades has been to to look at the ratio of bytes of memory capacity to flops performed; for bandwidth it is similarly the ratio of bytes per second transiting a node to the flops performed.

The magic number for decades for both numbers has been 1 to 1, but many of the most modern systems, especially at the high end, have much lower numbers. Memory ratios today, for example, run from 0.3 (commodity microprocessor-based systems) to 0.15 bytes per flop (Blue Gene-like systems). Modern GPUs are even lower, with on the order of 1 GB for a few hundred gigaflops, for ratios of around 0.002 to 0.01. The strawman design of the Exascale technology report was in the same range as the GPUs at 0.0036 bytes per flop, regardless of system size. These ratios can be a cause for concern for a) applications with computational complexity that is linearly proportional to their working set size, and b) storage systems in which the next level of the storage hierarchy is too slow for out-of-core algorithms to be practical. However, as discussed in Chapter 4, there are many applications that are not limited by a). Also, recent trends in storage technologies such as Phase Change Memory (PCM) and Flash Memory suggest that b) may not be a hard limitation either.

Bandwidth numbers are similar, but even more difficult to pin down, primarily because of the exploding trend toward multi-core chips with complex memory hierarchies. In such cases on-chip inter-core bandwidths may vary widely, as area and power for processing can be traded off for interconnect routing. Off-chip, however, is different, primarily because the number of off-chip contacts has been relatively flat for years, and the bandwidth at which these contacts can be driven is often fixed by external parameters such as cable characteristics, and the amount of power that a designer wants to spend in driving them at higher rates. The strawman exascale design from the previous report employed a tapered design, with less bandwidth available as one moved up the hierarchy (page 180, Table 7.7). Here the bytes per second per flop per second varied from 0.25 when accessing a local cache to 0.02 when accessing the (limited) local memory, to 0.006 when communicating with other nodes. Even at these highly reduced ratios, the estimated power of the upper level interconnect was 27% of the total system power, meaning that increasing such numbers can only be done at a huge power penalty. The suggested "adaptive" design was an attempt to allow at least some flexibility in boosting these ratios in cases where the processing needs are reduced.

The reason for these lower ratios is cost — both in provisioning more memory chips and in power consumed by them. Both come from both the memory chips themselves and the associated I/Os (both contacts and driver/receivers), especially when additional interface chips must be added to glue in large numbers of memory chips. This trend will continue through Exascale systems, and thus it is crucial to understand the limitations.

B.7.2 Sample Application Patterns

Any real application will have its own unique access pattern to application instance data which will change as we change the mapping of such data to node memories. However, to get some insight we define here three simple application data sets, each of size "N," and their mappings onto P different nodes:

- *Random*: We assume here that major data structure for any run of the application is not correlatable with any placement policy, meaning that any reference made by a program in a node could be to any datum in any other node, with equal probability. Thus each node gets N/P elements of data. GUPS is an example of such an application.

- *Fully Partitioned*: In contrast to the random access mapping, here we assume that the data can be perfectly partitioned to different nodes, so that for the bulk of the application, there is no possibility of any one node accessing any other. At the end of the major processing, however, there may be some exchange of data among nodes, as in a parallel prefix computation. Again, each node gets N/P elements of data. Searching, max finding, and various forms of data clustering and data mining are possible examples of such problems.

- *Dense Multi-dimensional*: Here the data structure is a regular-shaped "cube" of dimension D, and width W of each face. Thus N = W^D, and each node receives a unique sub-cube of size $(W/p)^D$ where for simplicity we assume that $P = p^D$ for some p. Applications will want to access both local data within a node, and *ghost data* consisting of y "layers" of data in other nodes' sub-cubes that are logically "at the surface" of each node's sub-cube. Examples of such applications in one dimension might be sub-string search in a text string, in two dimensions of various matrix problems, and in three dimensions of many 3D physics models.

An additional factor in many problems with the latter multi-dimensional structure is that for weak scaling, if the growth in problem set is due to grid refinement (smaller distances between grid points), then the unit of time represented by one computation cycle also decreases, so that additional computational cycles are needed to get to the same overall period of simulated time.

B.7.3 Off-node Data

The actual amount of physical data that must be resident in each node is a function of both the data structure (as discussed above) and performance-driven considerations. In most real systems, "off-node" data is significantly further away than "on-node," and thus there may be significant reasons to keep copies of such data in local memory. There are at least two kinds of such copied data:

- *Replicated Common Data*: This is information which is identical from one node to another, as in program code and/or lookup tables. It is typically read in at the beginning of an application's execution, and remains unchanged for the duration. Thus for long running applications, while it may occupy memory space, it adds nothing in terms of inter-node bandwidth requirements.

- *Ghost Data*: This corresponds to the ghost surface data when the main data structure is multi-dimensional as described in Section B.7.2. Such data typically makes up a goodly chunk of inter-node communication, with either a push or a pull of the current values of all surfaces at the beginning of each iteration of the algorithm. A reasonable approximation for the size of such ghost data is; $2D * Y * (W/p)^{(D-1)}$, where Y is the number of layers.

B.7.4 Memory per Node

The minimum memory needed per node is the sum of the replicated and the non-replicated. For both the Random and Fully Partitioned data structures, there is no ghost data, so that the required memory per node should be constant for increasing P for weak scaling (problem size increases), or

decrease for increasing P for strong scaling. Whether or not the decrease in the latter case goes as 1/P depends on the size of any non-replicated data.

For the Multi-dimensional case, it is interesting to note that when we attempt strong scaling the percentage of data in ghost cells relative to local data grows as P increases. In fact they become equal when $p = W/(2DY)$ or $P = (W/(2DY))^D$. For the common case of 3 dimensions and one layer of ghost cells on each sub-cube surface, this corresponds to $P = N/198$, or when each node holds a cube of width 6. Further, when P=N each node holds the minimum of exactly one grid point of real data, and for D=3 and L=1 has 6 ghost points around it. If we define R as the ratio of the non-replicated data with ghost to just the non-replicated data, then if we keep N constant as we increase P:

$$\mathrm{R} = 1 + (2\mathrm{DY}/\mathrm{W})*\mathrm{P}^{1/D}$$

For P=N/198, R=2; for P=N, R=7. Again, replicated data will change these ratios

B.7.5 Off-Node

For applications with a Random data structure as defined in Section B.7.2, the bandwidth in and out of a node is a strong function only of the number of accesses made by each node into the global data, and only a weak function of the number of nodes (if accesses are truly random, then only 1/P of them are local, which for large P is near zero). Thus if the size of the total table and the total number of accesses to be made are held constant (*i.e.,* the problem scales strongly), then as P increases, each node holds smaller data and both generates and receives fewer accesses, but in a shorter period of time. With perfect strong scaling, the shorter time balances out the fewer references and the net bandwidth in and out of a node needs remain approximately constant.

If both the total table and the total number of accesses made grows with P (*i.e.,* the problem scales perfectly weakly), then as P increases, the number of accesses per unit time again remains approximately constant, and the bandwidth requirement is unchanged.

Applications with the Fully Partitioned model only access their local data sets until some final stage. Thus to a first approximation, their off-node bandwidth is zero regardless of how P or N changes.

Multi-dimensional applications have an entirely different off-node bandwidth scaling characteristic. In this case, the data transferred between nodes at each time step is twice the ghost data size (the ghost data from the other nodes must come in and each node must distribute part of its data to surrounding nodes as part of their ghost data). Thus, for strongly scaled configurations where the total data size is constant, and the processing is a linear function of the number of local grid points, then the off-node bandwidth is approximately twice the ghost size divided by the node's execution time, which is proportional to the number of local data points. This ratio is thus:

$$2*2\mathrm{D}*\mathrm{Y}*(\mathrm{W}/\mathrm{p})^{(D-1)} \;/\; ((\mathrm{W}/\mathrm{p})^D) = 4\mathrm{DYp}/\mathrm{W}$$

Since D and W are constant, the bandwidth thus actually *climbs* as the D'th root of P.

For weak scaling, the local data set size and the ghost data remains constant per node. Thus on a per computational cycle basis the bandwidth needed remains constant.

B.8 Relevance to Exascale

This section tries to draw some lessons from the above discussion in terms of the "data center" sized exascale system from the prior exascale report. We use as a baseline the clean sheet "aggressive" strawman of Section 7.3 of that report.

B.8.1 Scaling Implications

The aggressive strawman of the prior report had significantly different characteristics in many of the metrics than current generations of machines. To get some handle on what this means in terms of supporting applications, we will use the scaling relations developed above, and extrapolate from some existing systems, in particular from an XT4 and a Blue Gene/P system.

To start, we define P_{exa} as the number of cores/processors available in an exascale system (166 million in the aggressive strawman when scaled to the data center size system). Likewise P_{base} will be the equivalent number of cores in a baseline system (either the XT4 or the BG/P). Further, we assume that the extra processors in the exascale system will be used for both strong and weak scaling, and relate P_{exa} and P_{base} as follows:

$$P_{exa} = K_s * K_w * P_{base}$$

where K_s and K_w refer to how the additional processors in the exa system are used for *strong* and *weak* scaling of applications, respectively. We are free to select any combination of K_s and K_w as long as they obey the above relation and don't violate any other constraints.

In particular, if we assume we can port and then weakly scale an application that runs today on a baseline system by K_w, then this means that K_w*P_{base} groups of K_s processors on the exascale system will hold an instance whose size is K_w times that running on the baseline. Further, with perfect weak scaling, using only one core per group should result in an execution time on the exascale system for the K_w scaled data set that equals the time if we had used only the original data set and P_{base} cores. This can only work if the exascale system has sufficient aggregate memory, *i.e.,*

$$\text{Memory}_{exa} \geq K_w * \text{Memory}_{base}$$

As discussed earlier, for data structures such as the multi-dimensional, it may be that this $\geq$ relationship may require a factor of 2 or more.

Next, we assume that strong scaling will be used within each of the K_w*P_{base} groups to accelerate their solution to their part of the problem. There are K_s such processors available for each such acceleration. With perfect strong scaling, the total solution time would thus drop by a factor of K_s over using only one core per group in the weakly scaled version.

Finally, since the performance characteristics of the exascale system and the baseline cores are different, we need to throw in a correction factor K_{speed} to make for an apples-to-apples time to solution comparison. As an approximation of a best case conversion factor, we compute:

$$K_{speed} = (\text{Clock}_{exa}/\text{Clock}_{base}) * (\text{Flops-per-cycle}_{exa}/\text{Flops-per-cycle}_{base})$$

Thus the best an exascale system can do is to speed up by a factor of K_{speed}*K_s an application that is K_w times larger than that which will run on the baseline system, as long as:

$$K_w \leq \text{Memory}_{exa} \ / \ \text{Memory}_{base}$$
and
$$K_s * K_w \leq P_{exa} \ \ P_{base}$$

One additional caveat to the above is for those cases where the data structure being expanded is a multi-dimensional data set associated with a physics problem, then the number of computational cycles needed to simulate the same period of time grows roughly with the "width" of the overall data structure. For dimensionality D, a growth in size by K_w corresponds to a growth in width of $(K_w)^{1/D}$. the effective speedup of the enlarged application to cover the same amount of simulated time must be reduced to:

	XT4	BG/P
Maximum K_w	78	109
Maximum K_w*K_s	7210	2535
K_{speed}	1.19	1.82

Table B.1: Bounding parameters for scaling to aggressive exascale system.

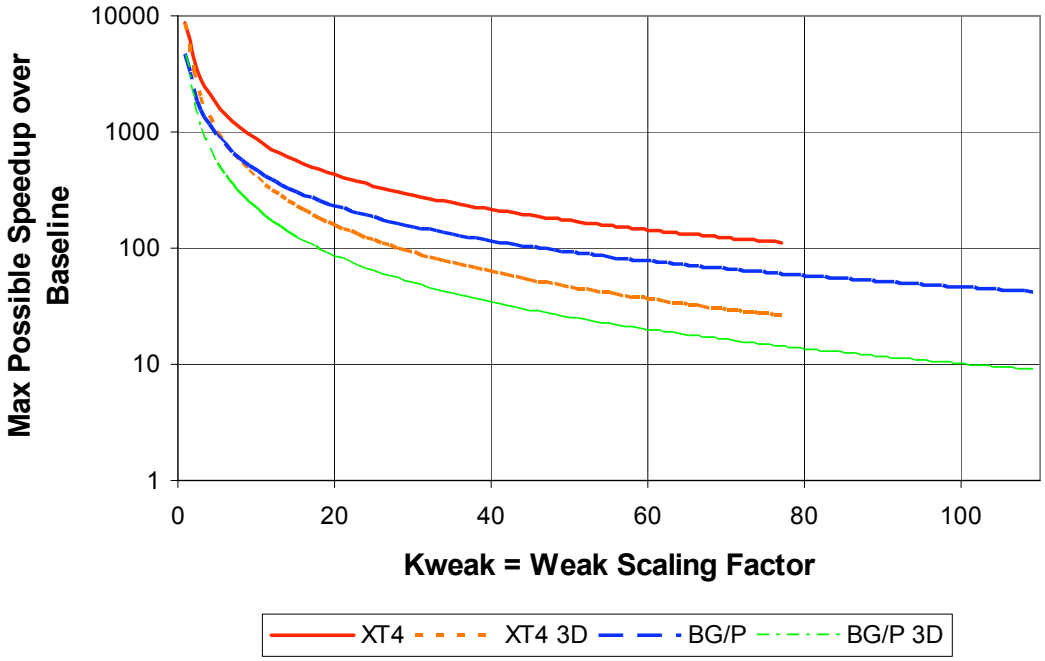

Figure B.12: Speedups as a function of weakly scaled problems.

$$K_{speed} * K_s \ / \ (K_w)^{1/D}$$

Table B.1 summarizes these constants for the two baseline systems. Figure B.12 then graphs the maximum possible speedup that an aggressive strawman exascale system might provide for applications that are scaled in size over what could be fit on a baseline system today. The lines labeled "3D" refer to cases where there are extra calculations needed to account for the refined grids.

B.8.2 Implications of Amdahl's Law

The historical records for the TOP500 indicate an average efficiency of between 70% to 80% for the top systems over time, where efficiency here is measured in terms of flop rate, and the benchmark is Linpack. For comparison, we can ask what is the maximum serial percentage, as seen by the code run in the 166 million cores that would still yield 70% efficiency. Using the above equations, this results in a ratio of about 2.6E-9, or only about one out of every 2.6 billion instructions issued can be issued by one core when all other cores are idle. This is perhaps a million times smaller than typical serial percentages for today's machines, and indicates a significant potential problem to be overcome.

Sync Time (in cycles)	Minimum Time between Syncs (in cycles)
400	933
4000	9,330
40,000	93,300

Table B.2: Minimum time between sync points to maintain 70% efficiency.

B.8.3 Synchronization Points

Another way to gain insight is to ask how often can a program running on all 166 million cores go through a global synchronization before we drop to 70% efficiency. Table B.2 lists, for several different ranges of time for all threads in all cores to synchronize, what is the minimum time between sync points during which all cores must be running at 100% efficiency to maintain 70% efficiency, assuming no jitter in the completion of code that leads to the synchronization. The 400 cycle case corresponds to an architecture where direct hardware support for synchronization is provided that can sync on chip in perhaps 10 cycles, and then requires perhaps 4 traversals of the entire system. The 40,000 cycle case corresponds to a more software-implemented case where the barrier is formed as a tree of smaller barriers between groups of processors. As can, be see, the resulting times between sync points in all cases are not unreasonable.

Appendix C

CUDA as an Example Execution Model

In the last few years, as graphics processors (*GPUs*) have become more general purpose, execution models have been introduced to allow them to participate in high end computationally-intensive applications. This section overviews one such model, namely that which emerged from a line of GPUs from NVIDIA but that has been ported to other hardware from other vendors, and has in fact become in the last few months the basis for a new language standard called OPENCL[1] adopted by most major computer vendors. NVIDIA uses the name *CUDA* for *Compute Unified Device Architecture* for both the overall execution model and the programming model that accompanies it. The result is a unique combination of SPMD-like, and SIMD-like, massively multi-threaded shared memory multi-processing, that may be particularly relevant for at least some of the types of architectures that may emerge from Extreme computing technologies.

C.0.3.1 General CUDA Execution Model

The basic compute model behind CUDA assumes that data is organized into multi-dimensional arrays, and that most processing involves applying same functions to different regions of these arrays. This application of a single function to a region of data is termed a *kernel*. The execution of such functions against the smallest subset of such data is called a *thread*, and multiple threads are in general assumed to be able to execute logically independently of each other.

The specification of the mapping of different threads to different data subsets is termed an *execution configuration*. Such configurations are themselves hierarchical, with the lowest level, termed a *thread block*, representing a set of threads that are guaranteed to run more or less in tandem on the same processor at around the same time. Each thread within a block has a unique identifier, which a typical thread program then uses to associate itself with a different subset of data. Thread blocks may be up to 512 threads in size.

Sets of such thread blocks, called *thread grids*, may also be defined within a configuration, and while each thread within a block must be executed concurrently on the same processor, different blocks may execute on different processors, in arbitrary order. These grids may be multi-dimensional, and again each such block within a grid has a unique identifier that can be used to associated the threads with different data subsets.

In general, kernel invocation is asynchronous, that is multiple kernels may be executing concurrently, unless they have been linked into *streams*, where the start of one kernel is predicated

[1] http://www.khronos.org/opencl/

upon completion of another. Multiple independent streams may be executing concurrently and interleaved in arbitrary fashions.

An explicit memory hierarchy exists, from a *global memory* that is visible to all threads, and where data is persistent throughout execution, to *local memory* (also called *shared memory*) that is allocated only for the execution period of a thread block and where only the threads started by the kernel's execution configuration may access, to *register files* that contain data that is private to individual threads. These is, however, little coherency or consistency support to memory access made by threads to these memories, especially global or shared.

C.0.3.2 Current CUDA Hardware Model

The current hardware model for NVIDIA processor chips that support CUDA assume a *host processor* of conventional design that is separate from the CUDA *device*. Both the host and the device have separate memory spaces, with only the host capable of accessing both. The device can access only its own memory. The host executes much of the overall application program control, including data transfer between host and device memories, and initiation of the highest level kernels into the device.

The device has an input queue into which kernel execution configuration requests are placed. The output of this queue schedules different thread blocks on different *Stream Multiprocessors* (SM) within the device. Each SM has within it multiple (today 8) separate *Stream Processors* (SP), each with their own register file of significant capacity (today 2048 words). Each SP is multi-threaded, meaning that it can hold the current state for dozens of separate threads, and use one of these state sets at a time to perform program computations. Each SM has a local memory (today 16384 bytes) that is accessible to all of its associated SPs. This memory can be accessed an order of magnitude faster than device global memory, and can be used as an explicitly managed cache, as well as a means of inter-SP communication. The local memory is divided into a number of banks (today 16) that allow concurrent access.

When the queue schedules a thread block, it breaks it up into fixed sized sets of threads, called *warps*, which are then executed on SMs. Today there are 32 threads in a warp. When given to an SM, the warp is distributed over the register files of the associated SPs (today 4 thread states from the warp are placed in each of the 8 SPs). The SM maintains a common program counter for all the threads in the warp, and fetches instructions for the warp one at a time. The same instruction is then given (over 4 cycles today) to all threads associated with the warp, and is then executed by each thread in the SP holding its register set.

Each thread maintains a record of whether or not it is to participate in some particular instruction so that even though the code is broadcast by the SM to all threads in a warp, each may choose individually to ignore or execute it. For example, on if-then-else blocks all threads in a warp evaluate the test condition, and then all the code associated with both the then and else paths is broadcast by the SM. Each SP decides which instructions it executes.

Loads and stores may all be executed by the SPs associated with a warp at the same time, but there is no requirement that the addresses generated by each SP be related to each other in any way. Separate hardware between the SPs and the multiple off-chip device memory ports takes the addresses generated by the SPs and *coalesces* them into patterns that maximizes the bandwidth with the memory. The address patterns that can be coalesced into concurrent memory accesses are relatively simple. *Broadcast* and adjacent accesses to global memory can generally be coalesced. Accesses to local memory can be coalesced as long as there are no accesses to multiple locations within a given bank (*i.e.,* bank conflicts).

Several warps are simultaneously assigned to each SM, which then execute in a time multiplexed

manner, allowing latency hiding of memory accesses. The GPU supports a maximum number of warps per multiprocessor (today 32) which may be reduced by insufficient number of threads available for execution, exhaustion of the SM local memory, or exhaustion of the SM register file.

This conditional execution and unpredictable execution time for individual instructions in a warp means that warps do not move in lockstep from a timing fashion. Thus each SM and its SPs support multiple warps simultaneously, and the SM will switch to issuing instructions for a different warp if not all threads associated with a prior warp have completed their instructions.

While there is no guaranteed memory consistency or operation ordering, the hardware does support instructions that allow a warp to suspend until all threads in all warps associated with a block have reached the same point.

C.0.3.3 CUDA Programming Model

For the most part, CUDA today is implemented as an extension of C, with the major differences coming in two areas. First there are *prefixes* that allow a programmer to specify whether a particular function is to be executed on the device or the host, and into which class of device memory a variable is to be allocated. Second is some additional annotation at a device function call that specifies the execution configuration of threads to be invoked: their number and structure both in grids and blocks.

Additionally, a suite of predefined variables allow a thread to determine which one it is within a block, and which block within a grid it is. Significant libraries also provide a wide range of additional capabilities.

C.0.3.4 Current Capabilities

As an example, a Tesla C1060 Computing Processor has the following characteristics[2]:

- 30 separate SMs on each chip, each with 8 SPs, for a total of 240 SPs.
- Each SP is capable of executing up to 3 single precision floating point operations per cycle, at up to 1.3GHz, for a peak of about 933 GFlops.
- An SM and its SPs may contain up to 8 separate thread blocks, consisting of up to 24 separate active warps, for a total of up to 768 active threads per SM.
- Up to 4 GB of GDDR3 memory may be attached, at an aggregate bandwidth of up to 102 GB/sec.

These numbers translate into a device that can support up to 23,040 concurrent threads running at an aggregate of up to about 1 Tflop single precision, with about 0.004 bytes per single precision flop of memory, and 0.1 bytes per second of bandwidth per single precision flop.

[2] http://www.nvidia.com/object/product_tesla_c1060_us.html

Appendix D

Extreme Scale Software Study Group Members

D.1 Committee Members

Academia & Industry	
Name	**Organization**
Vivek Sarkar, Chair	Rice University
Saman Amarasinghe	Massachusetts Institute of Technology
William Carlson	Institute for Defense Analyses
Andrew Chien	Intel
William Dally	Stanford University
Elmootazbellah Elnohazy	IBM
Mary Hall	University of Southern California Information Sciences Institute
Robert Harrison	Oak Ridge National Laboratory
Charles Koelbel	Rice University
David Koester	MITRE
Peter Kogge	University of Notre Dame
John Levesque	Cray
Daniel Reed	Microsoft
Robert Schreiber	Hewlett-Packard Laboratories
John Shalf	Lawrence Berkeley Laboratory
Allen Snavely	University of San Diego & San Diego Supercomputer Center
Thomas Sterling	Louisiana State University
Government and Support	
William Harrod, Organizer	DARPA
Daniel Campbell	Georgia Tech Research Institute
Kerry Hill	Air Force Research Laboratory
Jon Hiller	Science & Technology Associates
Sherman Karp	Consultant
Mark Richards	Georgia Institute of Technology
Al Scarpelli	Air Force Research Laboratory

D.2 Biographies

Saman P. Amarasinghe is an Associate Professor in the Department of Electrical Engineering and Computer Science at Massachusetts Institute of Technology and a member of the Computer Science and Artificial Intelligence Laboratory (CSAIL). Currently he leads the Commit compiler group and was the co-leader of the MIT Raw project. Under Saman's guidance, the Commit group developed the StreamIt language and compiler for the streaming domain, Superword Level Parallelism for multimedia extensions, DynamoRIO dynamic instrumentation system, Program Shepherding to protect programs against external attacks, and Convergent Scheduling and Meta Optimization that uses machine learning techniques to simplify the design and improve the quality of compiler optimization. His research interests are in discovering novel approaches to improve the performance of modern computer systems and make them more secure without unduly increasing the complexity faced by either the end users, application developers, compiler writers, or computer architects. He was also the founder of Determina Corporation, which productized Program Shepherding. Prof. Amarasinghe received his BS in Electrical Engineering and Computer Science from Cornell University in 1988, and his MSEE and Ph.D from Stanford University in 1990 and 1997, respectively.

Daniel P. Campbell is a Senior Research Engineer in the Sensors and Electromagnetic Applications Laboratory of the Georgia Tech Research Institute. Mr. Campbell's research focuses on application development infrastructure for high performance embedded computing, with an emphasis on inexpensive, commodity computing platforms. He is co-chair of the Vector Signal Image Processing Library (VSIPL) Forum, and has developed implementations of the VSIPL and VSIPL++ specifications that exploit various graphics processors for acceleration. Mr. Campbell has been involved in several programs that developed middleware and system abstractions for configurable multicore processors, including DARPA's Polymorphous Computing Architectures (PCA) program.

William W. Carlson is a member of the research staff at the IDA Center for Computing Sciences where, since 1990, his focus has been on applications and system tools for large-scale parallel and distributed computers. He also leads the UPC language effort, a consortium of industry and academic research institutions aiming to produce a unified approach to parallel C programming based on global address space methods. Dr. Carlson graduated from Worcester Polytechnic Institute in 1981 with a BS degree in Electrical Engineering. He then attended Purdue University, receiving the MSEE and Ph.D. degrees in Electrical Engineering in 1983 and 1988, respectively. From 1988 to 1990, Dr. Carlson was an Assistant Professor at the University of Wisconsin-Madison, where his work centered on performance evaluation of advanced computer architectures.

Andrew Chien is vice president of the Corporate Technology Group and director of Research for Intel Corporation. He previously served as the Science Applications International Corporation Endowed Chair Professor in the department of computer science and engineering, and the founding director of the Center for Networked Systems (CNS) at the University of California at San Diego. CNS is a university-industry alliance focused on developing technologies for robust, secure, and open networked systems. For more than 20 years, Chien has been a global leader in research and development of high-performance computing systems. His expertise includes networking, Grids, high performance clusters, distributed systems, computer architecture, high speed routing networks, compilers, and object oriented programming languages. He is a Fellow of the American Association for Advancement of Science (AAAS), Fellow of the Association for Computing Machinery

(ACM), Fellow of Institute of Electrical and Electronics Engineers (IEEE), and has published over 130 technical papers. He serves on the Board of Directors for the Computing Research Association (CRA), Advisory Board of the National Science Foundation's Computing and Information Science and Engineering (CISE) Directorate, and the Editorial Board of the Communications of the Association for Computing Machinery (CACM). From 1990 to 1998, Chien was a professor at the University of Illinois at Urbana-Champaign. During that time, he held joint appointments with both the National Center for Supercomputing Applications (NCSA) and the National Partnership for Advanced Computational Infrastructure (NPACI), working on large-scale clusters. In 1999 he co-founded Entropia, Inc., an enterprise desktop Grid company. Chien received his bachelor's in electrical engineering, master's and Ph.D. in computer science from the Massachusetts Institute of Technology.

William J. Dally is The Willard R. and Inez Kerr Bell Professor of Engineering and the Chairman of the Department of Computer Science at Stanford University. He is also co-founder, Chairman, and Chief Scientist of Stream Processors, Inc. Dr. Dally and his group have developed system architecture, network architecture, signaling, routing, and synchronization technology that can be found in most large parallel computers today. While at Bell Labs Bill contributed to the BELLMAC32 microprocessor and designed the MARS hardware accelerator. At Caltech he designed the MOSSIM Simulation Engine and the Torus Routing Chip which pioneered wormhole routing and virtual-channel flow control. While a Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology his group built the J-Machine and the M-Machine, experimental parallel computer systems that pioneered the separation of mechanisms from programming models and demonstrated very low overhead synchronization and communication mechanisms. At Stanford University his group has developed the Imagine processor, which introduced the concepts of stream processing and partitioned register organizations. Dr. Dally has worked with Cray Research and Intel to incorporate many of these innovations in commercial parallel computers, with Avici Systems to incorporate this technology into Internet routers, co-founded Velio Communications to commercialize high-speed signaling technology, and co-founded Stream Processors, Inc. to commercialize stream processor technology. He is a Fellow of the IEEE, a Fellow of the ACM, and a Fellow of the American Academy of Arts and Sciences. He has received numerous honors including the IEEE Seymour Cray Award and the ACM Maurice Wilkes award. He currently leads projects on computer architecture, network architecture, and programming systems. He has published over 200 papers in these areas, holds over 50 issued patents, and is an author of the textbooks, Digital Systems Engineering and Principles and Practices of Interconnection Networks.

Elmootazbellah N. (Mootaz) Elnozahy is a Senior Manager and Master Inventor in IBM's Research Division in Austin, Texas. He obtained a B.Sc. degree with Highest Honours in Electrical Engineering from Cairo University in 1984, and the M.S. and Ph.D. degrees in Computer Science from Rice University in 1990 and 1993, respectively. From 1993 until 1997, he was on the faculty at the School of Computer Science at Carnegie Mellon University, where he received a prestigious NSF CAREER award. In 1997, he moved to the IBM Austin Research Lab and started the Systems Software Department, which today includes over 25 researchers investigating high performance computing, low-power systems, and simulation tools. From 2005 to 2007, Mootaz joined the product division to accelerate the productization of his research project. Prior to joining IBM, he has worked on rollback-recovery, replication, and reliable distributed systems. While at IBM, he has worked on code and trace compression cc-NUMA systems, acceleration of the web site performance for the U.S. Census Bureau, blade-based servers, security of IP-based protocols, and performance tools.

He led the first two phases of the Productive, Easy-to-Use, Reliable Computing System (PERCS) project, which is IBM's effort under DARPA's HPCS initiative. Mootaz is also an Adjunct Associate Professor at the University of Texas at Austin, and has consulted with Bell Labs, Bellcore, NSF and the state of Texas. He has served on over 30 technical program committees in the areas of distributed operating systems and reliability. Mootaz's research interests include distributed systems, operating systems, computer architecture, and fault tolerance. He has published 31 refereed articles in these areas, and has been awarded 20 patents.

William Harrod joined DARPA's Information Processing Technology Office (IPTO) as a Program Manager in December of 2005. His area of interest is extreme computing, including a current focus on advanced computer architectures and system productivity, including self- monitoring and self-healing processing, Exascale computing systems, highly productive development environments and high performance, advanced compilers. He has over 20 years of algorithmic, application, and high performance processing computing experience in industry, academics and government. Prior to his DARPA employment, he was awarded a technical fellowship for the intelligence community while employed at Silicon Graphics Incorporated (SGI). Prior to this at SGI, he led technical teams developing specialized processors and advanced algorithms, and high performance software. Dr. Harrod holds a B.S. in Mathematics from Emory University, a M.S. and a Ph.D. in Mathematics from the University of Tennessee.

Mary Hall is an associate professor in the School of Computing at University of Utah. Her research focuses on compiler technology for exploiting performance-enhancing features for novel computer architectures. She received her PhD from Rice University in 1991. From 1996 to 2008, she was jointly a project leader at Information Sciences Institute and a research associate professor in the Department of Computer Science at University of Southern California. Prior to 1996, Prof. Hall held research positions at Caltech, Stanford and Rice University. Prof. Hall is currently leading the autotuning group in the DOE SciDAC Performance Engineering Research Institute. Previously, she was principal investigator on DIVA (Data-IntensiVe Architecture), a system architecture project that utilizes processing logic internal to memory chips as smart memory co-processors, and on DEFACTO (Design Environment for Adaptive Computing), an end-to-end design environment for FPGA-based computing environments. Prof. Hall has served on over 40 program committees in compilers and their interaction with architecture, parallel computing, and embedded and reconfigurable computing, including 2010 program chair for the ACM Principles and Practice of Parallel Programming and 2009 program chair of the Code Generation and Optimization Conference, and workshop co-chair for SC08, among others. She has authored over 70 papers in these areas. She is active in the ACM, serving as chair of the ACM History Committee, and previously on the ACM Health of Conferences Committee. She also participates in outreach programs to encourage the participation of women in computer science.

Robert Harrison Robert J. Harrison holds a joint appointment between Oak Ridge National Laboratory (ORNL) and the University of Tennessee, Knoxville. At the university, he is a professor in the chemistry department. At ORNL he is a corporate fellow and leader of the Computational Chemical Sciences Group in the Computer Science and Mathematics Division. He has many publications in peer-reviewed journals in the areas of theoretical and computational chemistry, and high-performance computing. His undergraduate (1981) and post-graduate (1984) degrees were obtained at Cambridge University, England. Subsequently, he worked as a postdoctoral research fellow at the Quantum Theory Project, Univ. Florida, and the Daresebury Laboratory, England,

before joining the staff of the theoretical chemistry group at Argonne National Laboratory in 1988. In 1992, he moved to the Environmental Molecular Sciences Laboratory of Pacific Northwest National Laboratory, conducting research in theoretical chemistry and leading the development of NWChem, a computational chemistry code for massively parallel computers. In August 2002, he started the joint faculty appointment with UT/ORNL. In addition to his DOE Scientific Discovery through Advanced Computing (SciDAC) research into efficient and accurate calculations on large systems, he has been pursuing applications in molecular electronics and chemistry at the nanoscale. In 1999, the NWChem team received an R&D Magazine R&D100 award, and, in 2002, he received the IEEE Computer Society Sydney Fernbach award.

Kerry L. Hill is a Senior Electronics Engineer with the Advanced Sensor Components Branch, Sensors Directorate, Air Force Research Laboratory, Wright-Patterson AFB OH. Ms. Hill has 27 years experience in advanced computing hardware and software technologies. Her current research interests include advanced digital processor architectures, real-time embedded computing, and reconfigurable computing. Ms. Hill worked computer resource acquisition technical management for both the F-117 and F-15 System Program Offices before joining the Air Force Research Laboratory in 1993. Ms. Hill has provided technical support to several DARPA programs including Adaptive Computing Systems, Power Aware Computing and Communications, Polymorphous Computing Architectures, and Architectures for Cognitive Information Processing.

Jon C. Hiller is a Senior Program Manager at Science and Technology Associates, Inc. Mr. Hiller has provided technical support to a number of DARPA programs, and specifically computing architecture research and development. This has included the Polymorphous Computing Architectures, Architectures for Cognitive Information Processing, Power Aware Computing and Communications, Data Intensive Systems, Adaptive Computing Systems, and Embedded High Performance Computing Programs. Previously in support of DARPA and the services, Mr. Hiller's activities included computer architecture, embedded processing application, autonomous vehicle, and weapons systems research and development. Prior to his support of DARPA, Mr. Hiller worked at General Electric's Military Electronic Systems Operation, Electronic Systems Division in the areas of GaAs and CMOS circuit design, computing architecture, and digital and analog design and at Honeywell's Electro-Optics Center in the areas of digital and analog design. Mr. Hiller has a BS from the University of Rochester and a MS from Syracuse University in Electrical Engineering.

Sherman Karp has been a consultant to the Defense Research Projects Agency (DARPA) for the past 21 years and has worked on a variety of projects including the High Productivity Computing System (HPCS) program. Before that he was the Chief Scientist for the Strategic Technology Office (STO) of DARPA. At DARPA he did pioneering work in Low Contrast (sub-visibility) Image enhancement and Multi-Spectral Processing. He also worked in the area of fault tolerant spaceborne processors. Before moving to DARPA, he worked at the Naval Ocean Systems Center where he conceived the concept for Blue-Green laser communications from satellite to submarine through clouds and water, and directed the initial proof of principle experiment and system design. He authored two seminal papers on this topic. For this work he was named the NOSC Scientist of the Year (1976), and was elected to the rank of Fellow in the IEEE. He is currently a Life Fellow. He has co-authored four books and two Special Issues of the IEEE. He was awarded the Secretary of Defense Medal for Meritorious Civilian Service, and is a Fellow of the Washington Academy of Science, where he won the Engineering Sciences Award. He was also a member of the Editorial Board of the IEEE Proceedings, the IEEE FCC Liaison Committee, the DC Area

IEEE Fellows Nomination Committee, the IEEE Communications Society Technical Committee on Communication Theory, on which he served as Chairman from 1979-1984, and was a member of the Fellows Nominating Committee. He is also a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi and the Cosmos Club.

David Koester received his Masters in Applied Statistics (MAS) from The Ohio State University in 1978 and his doctorate from Syracuse University in 1996 under the guidance of Dr. Geoffrey Fox and Dr. Sanjay Ranka. He joined the MITRE Corporation at the MITRE-Rome site in 1978 and continues to work from that office which is co-located with the Air Force Research Laboratory (AFRL) Site Rome, NY. Dr. Koester's present areas of interest in High End Computing (HEC) technologies include understanding the high-level mappings of applications to computing architectures and metrics to evaluate system performance and productivity.

Peter M. Kogge is currently the Associate Dean for research for the College of Engineering, the Ted McCourtney Chair in Computer Science and Engineering, and a Concurrent Professor of Electrical Engineering at the University of Notre Dame, Notre Dame, Indiana. From 1968 until 1994, he was with IBM's Federal Systems Division in Owego, NY, where he was appointed an IBM Fellow in 1993. In 1977 he was a Visiting Professor in the ECE Dept. at the University of Massachusetts, Amherst, MA, and from 1977 through 1994, he was also an Adjunct Professor of Computer Science at the State University of New York at Binghamton. He has been a Distinguished Visiting Scientist at the Center for Integrated Space Microsystems at JPL, and the Research Thrust Leader for Architecture in Notre Dame's Center for Nano Science and Technology. For the 2000-2001 academic year he was also the Interim Schubmehl-Prein Chairman of the CSE Dept. at Notre Dame. His research areas include advanced VLSI and nano technologies, non von Neumann models of programming and execution, parallel algorithms and applications, and their impact on massively parallel computer architecture. Since the late 1980s' this has focused on scalable single VLSI chip designs integrating both dense memory and logic into "Processing In Memory" (PIM) architectures, efficient execution models to support them, and scaling multiple chips to complete systems, for a range of real system applications, from highly scalable deep space exploration to trans-petaflops level supercomputing. This has included the world's first true multi-core chip, EXECUBE, that in the early 1990s integrated 4 Mbits of DRAM with over 100K gates of logic to support a complete 8 way binary hypercube parallel processor which could run in both SIMD and MIMD modes. Prior parallel machines included the IBM 3838 Array Processor which for a time was the fastest single precision floating point processor marketed by IBM, and the Space Shuttle Input/Output Processor which probably represents the first true parallel processor to fly in space, and one of the earliest examples of multi-threaded architectures. His Ph.D. thesis on the parallel solution of recurrence equations was one of the early works on what is now called parallel prefix, and applications of those results are still acknowledged as defining the fastest possible implementations of circuits such as adders with limited fan-in blocks (known as the Kogge-Stone adder). More recent work is investigating how PIM-like ideas may port into quantum cellular array (QCA) and other nanotechnology logic, where in- stead of "Processing-In-Memory" we have opportunities for "Processing-In-Wire" and similar paradigm shifts.

John Levesque is the Director of Cray's Supercomputing Center of Excellence based at Oak Ridge National Laboratory (ORNL). The group is tasked with assisting the DoE Office of Science Researchers in porting and scaling their applications to the Petascale systems based at ORNL. Mr. Levesque is in the Chief Technology Office of Cray Inc, responsible for Application awareness

throughout the company. Prior to joining Cray, he was the Director of the Advanced Computing Technology Center based in IBM Research. Mr. Levesque started his career in high performance computing 40 years ago as an application developer at Sandia Laboratory and Air Force Weapons Laboratory in Albuquerque, New Mexico. He joined R&D Associates in Los Angeles, CA in 1972 where he ran a DARPA project to monitor ILLIAC IV code development efforts. Until joining IBM in 1998, he ran software development groups at Pacific Sierra Research and Applied Parallel Research. These groups pioneered parallel programming tools using "whole program" analysis in the VAST and FORGE software. Mr. Levesque co-authored a book "Guidebook to Fortran on Supercomputers" in 1981 and is currently working on a similar book addressing effective programming for multi-core architectures. Mr. Levesque received his Master Degree in Mathematics at the University of New Mexico in 1972.

Daniel Reed is Microsoft's Scalable and Multicore Computing Strategist, responsible for re-envisioning the mega-data center of the future. Previously, he was the Chancellor's Eminent Professor at UNC Chapel Hill, as well as the Director of the Renaissance Computing Institute (RENCI) and the Chancellor's Senior Advisor for Strategy and Innovation for UNC Chapel Hill. Dr. Reed has served as a member of the U.S. President's Council of Advisors on Science and Technology (PCAST) and as a member of the President's Information Technology Advisory Committee (PITAC). He recently chaired a review of the U.S. networking and IT research portfolio, and he recently completed a term as chair of the board of directors of the Computing Research Association. He was previously Head of the Department of Computer Science at the University of Illinois at Urbana-Champaign (UIUC). He has also been Director of the National Center for Supercomputing Applications (NCSA) at UIUC, where he also led National Computational Science Alliance. He was also one of the principal investigators and chief architect for the NSF TeraGrid. He received his PhD in computer science in 1983 from Purdue University.

Mark A. Richards is a Principal Research Engineer and Adjunct Professor in the School of Electrical and Computer Engineering, Georgia Institute of Technology. From 1988 to 2001, Dr. Richards held a series of technical management positions at the Georgia Tech Research Institute, culminating as Chief of the Radar Systems Division of GTRI's Sensors and Electromagnetic Applications Laboratory. From 1993 to 1995, he served as a Program Manager for the Defense Advanced Research Projects Agency's (DARPA) Rapid Prototyping of Application Specific Signal Processors (RASSP) program, which developed new computer-aided design (CAD) tools, processor architectures, and design and manufacturing methodologies for embedded signal processors. Since the mid-1990s, he has been involved in a series of pro- grams in high performance embedded computing, including the efforts to develop the Vector, Signal, and Image Processing Library (VSIPL) and VSIPL++ specifications and the Stream Virtual Machine (SVM) middleware developed under DARPA's Polymorphous Computing Architectures (PCA) program. Dr. Richards is the author of the text Fundamentals of Radar Signal Processing (McGraw-Hill, 2005).

Vivek Sarkar (Chair) is the E.D. Butcher Professor of Computer Science at Rice University. He conducts research in programming languages, compiler optimizations and runtime systems for parallel and high performance computer systems, and currently leads the Habanero Multicore Software Research project at Rice University. Prior to joining Rice, he was Senior Manager of Programming Technologies at IBM Research. His responsibilities at IBM included leading IBM's research efforts in programming model, tools, and productivity in the PERCS project during 2002- 2007 as part of the DARPA High Productivity Computing System program. His past projects include the X10

programming language, the Jikes Research Virtual Machine for the Java language, the ASTI optimizer used in IBM's XL Fortran product compilers, the PTRAN automatic parallelization system, and profile-directed partitioning and scheduling of Sisal programs. Vivek became a member of the IBM Academy of Technology in 1995, the E.D. Butcher Professor of Computer Science at Rice University in 2007, and was inducted as an ACM Fellow in 2008. He holds a B.Tech. degree from the Indian Institute of Technology, Kanpur, an M.S. degree from University of Wisconsin-Madison, and a Ph.D. from Stanford University. In 1997, he was on sabbatical as a visiting associate professor at MIT, where he was a founding member of the MIT RAW multicore project.

Alfred J. Scarpelli is a Senior Electronics Engineer with the Advanced Sensor Components Branch, Sensors Directorate, Air Force Research Laboratory, Wright-Patterson AFB OH. His current research areas include advanced digital processor architectures, real-time embedded computing, and reconfigurable computing. Mr. Scarpelli has 33 years research experience in computer architectures and computer software. In the 1970's, he conducted benchmarking to support development of the MIL-STD-1750 instruction set architecture, and test and evaluation work for the DoD standard Ada language development. In the 1980's, he was involved with the DoD VHSIC program, and advanced digital signal processor development, a pre- cursor to the F-22 Common Integrated Processor. In the 1990's, his research focused on the DARPA Pilot's Associate, the development of an embedded, artificial intelligence processor powered by an Associative Memory CoProcessor, real-time embedded software schedulability techniques, VHDL compilation and simulation tools, and application partitioning tools for reconfigurable computing platforms. Since 1997, he has provided technical support to multiple DARPA programs such as Adaptive Computing Systems, Polymorphous Computing Architectures, Architectures for Cognitive Information Processing, Networked Embedded Systems Technology, Mission Specific Processing, and Foliage Penetration. He holds a B.S. degree in Computer Science from the University of Dayton (1979) and an M.S. degree in Computer Engineering from Wright State University (1987).

Rob Schreiber is a Distinguished Technologist in and Assistant Director of the Exascale Computing Lab at Hewlett Packard Laboratories. Dr. Schreiber received an AB in mathematics from Cornell in 1972 and a PhD in Computer Science from Yale in 1977. He is known for research in sequential and parallel algorithms for matrix computation and compiler optimization for parallel languages. He was a professor of Computer Science at Stanford and at RPI and was chief scientist of the Saxpy Computer company. He was a co-developer of the sparse matrix extension of Matlab, and a leading designer of the High Performance Fortran programming language. He was one of the developers of the NAS parallel benchmarks. At HP, Rob helped lead the PICO Project, which developed a system for embedded processor synthesis from high-level specifications. In 2007 he was named as a Distinguished Scientist by the Association for Computing Machinery.

John Shalf is with the National Energy Research Scientific Computing Center of the Lawrence Berkeley National Laboratory. His background is in electrical engineering. He spent time in graduate school at Virginia Tech working on a C-compiler for the SPLASH-2 FPGA-based computing system, and at Spatial Positioning Systems Inc. (now ArcSecond) he worked on embedded computer systems. John first got started in HPC at the National Center for Supercomputing Applications (NCSA) in 1994, where he provided software engineering support for a number of scientific applications groups. While working for the General Relativity Group at the Albert Einstein Institute in Potsdam Germany, he helped develop the first implementation of the Cactus Computational Toolkit, which is used for numerical solutions to Einstein's equations for General Relativity and

which enables modeling of black holes, neutron stars, and boson stars. John joined Berkeley Lab in 2000 where he co-authored he "View from Berkeley" report with David Patterson et. al. at UC Berkeley, which discussed the future research challenges of multicore computing. He currently leads the Science Driven System Architecture group at the National Energy Research Supercomputing Center and leads the *Green Flash* project at LBL to develop energy-efficient computer architectures.

Allan E. Snavely is an Adjunct Assistant Professor in the University of California at San Diego's Department of Computer Science and is founding director of the Performance Modeling and Characterization (PMaC) Laboratory at the San Diego Supercomputer Center. He is a noted expert in high performance computing (HPC). He has published more than 50 papers on this subject, has presented numerous invited talks including briefing U.S. congressional staff on the importance of the field to economic competitiveness, was a finalist for the Gordon Bell Prize 2007 in recognition for outstanding achievement in HPC applications, and is primary investigator (PI) on several federal research grants. Notably, he is PI of the Cyberinfrastructure Evaluation Center supported by National Science Foundation, and Co-PI in charge of the performance modeling thrust for PERI (the Performance Evaluation Research Institute), a Department of Energy SciDAC2 institute.

Thomas Sterling is the Arnaud and Edwards Professor of Computer Science at Louisiana State University and a member of the Faculty of the Center for Computation and Technology. Dr. Sterling is also a Faculty Associate at the Center for Computation and Technology at California Institute of Technology and a Distinguished Visiting Scientist at Oak Ridge National Laboratory. Sterling is an expert in the field of parallel computer system architecture and parallel programming methods. Dr. Sterling led the Beowulf Project that performed seminal pathfinding research establishing commodity cluster computing as a viable high performance computing approach. He led the Federally sponsored HTMT project that conducted the first Peta ops scale design point study that combined advanced technologies and parallel architecture exploration as part of the national peta ops initiative. His current research directions are the ParalleX execution model and processor in memory architecture for directed graph based applications. He is a winner of the Gordon Bell Prize, co-author of five books, and holds six patents.

Appendix E

Extreme Scale Software Study Meetings, Speakers, and Guests

Meeting # 1 (Kickoff Meeting)

June 17, 2008, Boston, MA
Host: Massachusetts Institute of Technology

Committee members present

Dan Campbell, Andrew Chien, Bill Dally, Mootaz Elnohazy, Mary Hall, Robert Harrison, Bill Harrod, Jon Hiller, Sherman Karp, David Koester, John Levesque, John Shalf, Vivek Sarkar, Allan Snavely

Visitors

- Anant Agarwal, Massachusetts Institute of Technology
- Guy Steele, Sun Microsystems

Presentations

- "Introduction" — Vivek Sarkar:
- "Project Overview" — Bill Harrod
- "Software Challenges and Approaches for 1000 Cores: the CSAIL Angstrom Project" — Anant Agarwal
- "Breaking Sequential Habits of Thought" — Guy Steele

Meeting # 2

July 9, 2008, Atlanta, GA
Host: Georgia Institute of Technology

Committee members present

Saman Amarasinghe, Bill Carlson, Dan Campbell, Andrew Chien, Bill Dally, Mootaz Elnohazy, Bill Harrod, Jon Hiller, Sherman Karp, Peter Kogge, John Levesque, Mark Richards, Vivek Sarkar, Allan Snavely

Visitors

- David Bader, Georgia Institute of Technology
- Guy Steele, Sun Microsystems

Presentations

- "Status Update" — Vivek Sarkar
- "Concurrency at Exascale" — Peter Kogge
- "Exascale Analytics in Biology, Social Networks, and Security" - David Bader

Meeting # 3

July 31, 2008, Argonne, IL
Host: Argonne National Laboratory

Committee members present

Saman Amarasinghe, Dan Campbell, Bill Dally, Thomas Dunning, Mootaz Elnohazy, Mary Hall, Robert Harrison, Bill Harrod, Jon Hiller, Sherman Karp, Charles Koelbel, John Levesque, Vivek Sarkar, Allan Snavely

Visitors

- Peter Beckman, Argonne National Laboratory
- William Gropp, University of Illinois at Urbana-Champaign
- Rusty Lusk, Argonne National Laboratory
- Arvind Mithal, Massachusetts Institute of Technology
- Rob Pennington, University of Illinois at Urbana-Champaign
- Rob Schreiber, Hewlett-Packard Laboratories
- Marc Snir, University of Illinois at Urbana-Champaign
- Guy Steele, Sun Microsystems

Presentations

- "Status Update" — Vivek Sarkar
- "Programming models for Petascale Computing, and beyond" — Marc Snir
- "Scalability Challenges in System Software" — Pete Beckman
- "Reliability in Large-Scale Systems (DARPA Exascale Computing Resiliency Study)" — Mootaz Elnozahy

Meeting # 4

August 12, 2008, Houston, TX
Host: Rice University

Committee members present

Saman Amarasinghe (via teleconference), Dan Campbell, Mootaz Elnohazy, Mary Hall, Bill Harrod, Jon Hiller, Sherman Karp, Charles Koelbel, David Koester, Peter Kogge, John Levesque, Mark Richards, Vivek Sarkar, Allan Snavely, Tom Sterling

Visitors

- Jack Dongarra, University of Tennessee at Knoxville
- John Mellor-Crummey, Rice University
- Krishna Palem, Rice University
- Rob Schreiber, Hewlett-Packard Laboratories

Presentations

- "Status Update" — Vivek Sarkar
- "Scheduling for Numerical Linear Algebra Library at Scale" — Jack Dongarra
- "Tool Challenges for Exascale Computing" — John Mellor-Crummey
- "Compiler Optimizations for Power Aware Computing" — Krishna Palem

Meeting # 5

September 16, 2008, Stanford, CA
Host: Stanford University

Committee members present

Saman Amarasinghe, Dan Campbell, Andrew Chien, Bill Dally, Mootaz Elnohazy, Mary Hall (via teleconference), Robert Harrison, Bill Harrod, Jon Hiller, Sherman Karp, Charles Koelbel (via teleconference), David Koester, Mark Richards, Vivek Sarkar (via teleconference), John Shalf, Allan Snavely

Visitors

- Ron Brightwell, Sandia National Laboratories
- G. R. Gao, University of Delaware
- Kathy Yelick, University of California, Berkeley

Presentations

- "Status Update" — Vivek Sarkar
- "Presentation" — Kathy Yelick
- "FAST-OS" — Ron Brightwell
- "Cyclops System Software" — G. R. Gao

Meeting # 6

November 11, 2008, Stanford, CA
Host: Stanford University

Committee members present

Dan Campbell, Bill Carlson (via teleconference), Andrew Chien, Bill Dally, Mootaz Elnohazy, Mary Hall, Bill Harrod, Jon Hiller, Sherman Karp, David Koester, Peter Kogge, Mark Richards, Vivek Sarkar, John Shalf, Allan Snavely, Tom Sterling

Visitors

- Anant Agarwal, Massachusetts Institute of Technology

Presentations

- "Status Update" — Vivek Sarkar
- "Self-Aware Computing Presentation" — Anant Agarwal

Meeting # 7

February 25, 2009, Stanford, CA
Host: Stanford University

Committee members present

Bill Dally, Mootaz Elnohazy, Mary Hall, Bill Harrod, Jon Hiller, Sherman Karp, Peter Kogge, Mark Richards, Vivek Sarkar, John Shalf, Allan Snavely, Tom Sterling

Visitors

- None

Presentations

- "Status Update" — Vivek Sarkar

Bibliography

[1] http://top500.org/.

[2] http://www.er.doe.gov/ASCR/ProgramDocuments/TownHall.pdf.

[3] http://hpcrd.lbl.gov/E3SGS/main.html.

[4] http://computing.ornl.gov/workshops/town_hall/.

[5] https://www.cels.anl.gov/events/workshops/townhall07/index.php.

[6] http://www.zettaflops.org/fec07/index.html.

[7] http://www.lanl.gov/roadrunner/.

[8] http://www.ncsa.uiuc.edu/BlueWaters/.

[9] http://portal.acm.org/ft_gateway.cfm?id=1188502&type=pdf&coll=ACM&dl=GUIDE&CFID=1529677&CFTOKEN=97129850.

[10] http://www.gaussian.com/.

[11] http://www.msg.ameslab.gov/GAMESS/.

[12] http://www.simulia.com/.

[13] http://www.emsl.pnl.gov/docs/global/ga.html.

[14] http://www.csm.ornl.gov/workshops/HPA/documents/1-arch/hpa-xmt.pdf.

[15] http://en.wikipedia.org/wiki/Little's_law.

[16] http://www.vni.com/products/imsl/fortran/overview.php.

[17] http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms.

[18] http://www.netlib.org/blas/.

[19] http://www.netlib.org/lapack/.

[20] http://math-atlas.sourceforge.net/.

[21] http://icl.cs.utk.edu/plasma/index.html.

[22] http://www.fftw.org/.

[23] http://www.ffte.jp/.

[24] http://www.vsipl.org/.

[25] http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

[26] http://www-03.ibm.com/systems/p/software/essl/index.html.

[27] http://www.mpi-forum.org/docs/.

[28] http://www.emsl.pnl.gov/docs/parsoft/tcgmsg/tcgmsg.html.

[29] http://www.emsl.pnl.gov/docs/parsoft/tcgmsg-mpi/.

[30] http://www.mhpcc.edu/training/workshop2/mpi_io/MAIN.html.

[31] http://www.emsl.pnl.gov/docs/parsoft/ma/MA.html.

[32] http://www.emsl.pnl.gov/docs/parsoft/armci/.

[33] http://www.emsl.pnl.gov/docs/parsoft/chemio/chemio.html.

[34] http://www.windriver.com.

[35] http://www-unix.mcs.anl.gov/zeptoos.

[36] Anant Agarwal and Bill Harrod. Organic computing, August 2006.

[37] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[38] E. Allan et al. The Fortress language specification version 0.618. Technical report, Sun Microsystems, April 2005.

[39] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an smmp os. *ACM Trans. Comput. Syst.*, 25(3):6, 2007.

[40] David A. Bader. *Petascale Computing: Algorithms and Applications*. Chapman & Hall/CRC, 2007.

[41] Pavan Balaji, Wu-chun Feng, Jeremy Archuleta, Heshan Lin, Rajkumar Kettimuthu, Rajeev Thakur, and Xiaosong Ma. Semantics-based distributed i/o for mpiblast. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 293–294, New York, NY, USA, 2008. ACM.

[42] Guy Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-92-103, Carnegie Mellon University, January 1992.

[43] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[44] Fred Brauer and Carlos Castillo-Chavez. *Mathematical Models in Population Biology and Epidemiology*, volume 40 of *Texts in Applied Mathematics*. Springer, 2001.

[45] Ian Buck. Brook Specification v0.2.

[46] Zoran Budimlić, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *CPC '09: 14th International Workshop on Compilers for Parallel Computers*. Springer, January 2009.

[47] Zoran Budimlic, Aparna M. Chandramowlishwaran, Kathleen Knobe, Geoff N. Lowney, Vivek Sarkar, and Leo Treggiari. Declarative aspects of memory management in the concurrent collections parallel programming model. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 47–58, New York, NY, USA, 2008. ACM.

[48] Mary Carrington and Stephen J. O'Brien. The influence of hla genotype on aids. *Annual Review of Medicine*, 54(1):535–551, 2003. PMID: 12525683.

[49] Mark J. Chaisson and Pavel A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Research*, 18(2):324–330, 2008.

[50] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[51] Francis S. Collins, Michael Morgan, and Aristides Patrinos. The human genome project: Lessons from large-scale biology. *Science*, 300(5617):286–290, 2003.

[52] Willem de Bruijn and Herbert Bos. Pipesfs: fast linux i/o in the unix tradition. *SIGOPS Oper. Syst. Rev.*, 42(5):55–63, 2008.

[53] Steven J. Deitz, David Callahan, Bradford L. Chamberlain, and Lawrence Snyder. Global-view abstractions for user-defined reductions and scans. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 40–47, New York, NY, USA, 2006. ACM.

[54] Jack B. Dennis. Data Flow Supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[55] Jack Dongarra. Manycore computing: The impact on numerical software for linear algebra libraries. `http://www.netlib.org/utk/people/JackDongarra/SLIDES/sc07-wksh-mcore-1107.pdf`.

[56] Jack Dongarra, Pete Beckman, Patrick Aerts, Frank Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Terry Moore, Rick Stevens, Anne Trefethen, and Mateo Valero. The international exascale software project: A call to cooperative action by the global high performance community. *The International Journal of High Performance Computing Application*, 23(4), November 2009.

[57] H. Carter Edwards. Software environment for developing complex multiphysics applications. 2002.

[58] D. J. Eisenstein and P. Hut. Hop: A new group-finding algorithm for n-body simulations. *Astrophysical Journal*, 498:137–142, May 1998.

[59] D. J. Eisenstein and P. Hut. Hop: A new group-finding algorithm for n-body simulations. *Astrophysical Journal*, 498:137–142, May 1998.

[60] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specification v1.1.1, October 2003.

[61] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. pages 251–266, 1995.

[62] Peter Kogge et al. Exascale computing study: Technology challenges in achieving exascale system.

[63] J. Fass. Personal communication, 2008.

[64] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[65] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[66] A. Funk, V. Basili, L. Hochstein, and J. Kepner. Analysis of parallel software development using the relative development time productivity metric. *CTWatch Quarterly*, 2(4).

[67] Anwar Ghuloum. Ct: channelling nesl and sisal in c++. In *CUFP '07: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming*, pages 1–3, New York, NY, USA, 2007. ACM.

[68] G. Gilder. *Telecosm: The World After Bandwidth Abundance*. Free Press, New York, NY, 2002.

[69] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[70] *GPFS V3.2.1 Concepts, Planning, and Installation Guide.* IBM Corporation., August 2008.

[71] Jim Gray and Alexander S. Szalay. Where the rubber meets the sky: Bridging the gap between databases and science, 2004.

[72] B. Graybill, G. Allen, K. Alvin, A Atashi, M. Drahzal, D. Fisher, M. Giles, B. Lucas, T. Mattson, H. Morgan, E. Schnetter, B. Schott, E. Seidel, J. Shalf, S. Shamsian, and S.S. Tong. Hpc application software consortium summit (hpcasc) - concept paper. `http://www.cct.lsu.edu/gallen/Reports/HPCASC_March2007.pdf`, March 25-26, 2008.

[73] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.

[74] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[75] Mary Hall, David Padua, and Keshav Pingali. Compiler research: the next 50 years. *Commun. ACM*, 52(2):60–67, 2009.

[76] Robert Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions of Programming Languages and Systems*, 7(4):501–538, October 1985.

[77] *HDF5 User's Guide, HDF5 Release 1.8.2.* HDF Group., November 2008.

[78] Patrick Heimbach, Chris Hill, and Ralf Giering. An efficient exact adjoint of the parallel mit general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.

[79] David Hernandez, Patrice Franois, Laurent Farinelli, Magne Ø steråas, and Jacques Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, 2008.

[80] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pnfs. In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 18–27, Washington, DC, USA, 2005. IEEE Computer Society.

[81] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.

[82] Kenneth Iverson. *A Programming Language*. John Wiley and Sons, New York, NY, 1962.

[83] Efrat Jaeger-Frank, Christopher Crosby, Ashraf Memon, Viswanath Nandigam, J. Arrowsmith, Jeffery Conner, Ilkay Altintas, and Chaitan Baru. chapter A Three Tier Architecture for LiDAR Interpolation and Analysis, pages 920–927. 2006.

[84] L Kale and S Krishnan. CHARM++: A Portable Concurrent Object-Oriented System based on C++. *ACM Sigplan Notices:Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 28(10):91–108, October 1993.

[85] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[86] Michael Kistler, John Gunnels, Daniel Brokenshire, and Brad Benton. Petascale computing with accelerators. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 241–250, New York, NY, USA, 2009. ACM.

[87] David Koester. Exascale study application footprints. `https://pastec.gtri.gatech.edu/ECSS/images/c/c5/Exascale_App_Footprints_07-1449.pdf`, October 2007.

[88] Mike Kravetz and Hubertus Franke. Implementation of a multi-queue scheduler for linux. `http://lse.sourceforge.net/scheduling/mq1.html`, 2001.

[89] Andres Kriete and Roland Eils. *Computational Systems Biology*. Academic Press, 2005.

[90] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. *Conference Record of 8th ACM Symposium on Principles of Programming Languages*, 1981.

[91] S. Kumar, H. Raj, K. Schwan, and I. Ganev. The sidecore approach to efficient virtualization in multicore systems. 2007.

[92] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[93] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[94] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, October 1998.

[95] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

[96] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 471, Washington, DC, USA, 1996. IEEE Computer Society.

[97] Ying Liu, Wei keng Liao, and Alok Choudhary. Design and evaluation of a parallel hop clustering algorithm for cosmological simulation. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 82.1. IEEE Computer Society, 2003.

[98] William Loging, Lee Harland, and Bryn Williams-Jones. High-throughput electronic biology: mining information for drug discovery. *Nature Reviews Drug Discovery*, 6(3):220–230.

[99] *Lustre 1.6 Operations Manual.* Sun Microsystems., Nov 21 2008.

[100] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language Reference Manual Version 1.2. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-146, Rev. 1.

[101] Paul E. McKenney and Jonathan Walpole. Introducing technology into the linux kernel: a case study. *SIGOPS Oper. Syst. Rev.*, 42(5):4–17, 2008.

[102] M Metcalf and J Reid. *Fortran 90 Explained*. Oxford Science Publications, Oxford, England, 1990.

[103] R. G. Minnich, M. J. Sottile, S.-E. Choi, E. Hendriks, and J. McKie. Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels. *SIGOPS Operating Systems Review*, 40, 2006.

[104] E Mohr, D Kranz, and JR. Halstead, R. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[105] José Moreira, Michael Brutman, José Castanos, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: the blue gene/l story. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 118, New York, NY, USA, 2006. ACM.

[106] *MPI: A Message-Passing Interface Standard, Version 2.1.* Message Passing Interface Forum., June 2008.

[107] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[108] Jarek Nieplocha, Holger Dachsel, and Ian Foster. Distant i/o: One-sided access to secondary storage on remote processors. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 148, Washington, DC, USA, 1998. IEEE Computer Society.

[109] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum Archive*, 17:1–31, August 1998.

[110] J. Michael Oakes and Jay S. Kaufman. *Methods in Social Epidemiology: Research Design and Methods.* Wiley Default, 2006.

[111] R.A. Oldfield, P. Widener, A.B. Maccabe, L. Ward, and T. Kordenbrock. Efficient data-movement for lightweight i/o. *Cluster Computing, IEEE International Conference on*, 0:1–9, 2006.

[112] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte. Giga+: scalable directories for shared file systems. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*, pages 26–29, New York, NY, USA, 2007. ACM.

[113] F. Petrini, D. Kerbyson, and S. Pakin. The case of missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. *In Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC03)*, 2003.

[114] Removing the big kernel lock. `http://kerneltrap.org/Linux/Removing_the_Big_Kernel_Lock`, May 2008. Viewed on September 27, 2008.

[115] Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, and Ed Hartnett. *The NetCDF Users Guide. NetCDF Version 4.0.1.* Unidata Program Center., March 2009.

[116] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

[117] Vivek Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development*, 35(5/6), 1991.

[118] Vivek Sarkar. The PTRAN Parallel Programming System. In B. Szymanski, editor, *Parallel Functional Programming Languages and Compilers*, ACM Press Frontier Series, pages 309–391. ACM Press, New York, 1991.

[119] Vivek Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM Journal of Research and Development*, 41(3), May 1997.

[120] Vivek Sarkar, William Harrod, and Allan E. Snavely. Software challenges in extreme scale systems. In *2009 Conference on Scientific Discovery through Advanced Computing Program (SciDAC)*, June 2009.

[121] E. Schnetter, C. D. Ott, G. Allen, P. Diener, T. Goodale, T. Radke, E. Seidel, and J. Shalf. Cactus framework: Black holes to gamma ray bursts. 2007.

[122] Charles Semple and M. A. Steel. *Phylogenetics*. Oxford University Press, 2003.

[123] John Shalf, Shoaib Kamil, Leonid Oliker, and David Skinner. Analyzing ultra-scale application communication requirements for a reconfigurable hybrid interconnect. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 17, Washington, DC, USA, 2005. IEEE Computer Society.

[124] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer III. Phaser accumulators: a new reduction construct for dynamic parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2009.

[125] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[126] Jun Shirako, Jisheng Zhao, V. Krishna Nandivada, and Vivek Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. ACM, June 2009.

[127] Stephen C. Simms, Gregory G. Pike, S. Teige, Bret Hammond, Yu Ma, Larry L. Simms, C. Westneat, and Douglas A. Balog. Empowering distributed workflow with the data capacitor: maximizing lustre performance across the wide area network. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 53–58, New York, NY, USA, 2007. ACM.

[128] S. Skedzielewski and J. Glauert. IF1 – An Intermediate Form for Applicative Languages. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-170.

[129] A. Snavely, R. Pennington, and R. Loft. Workshop report: Petascale computing in the biological sciences. `www.sdsc.edu/~allans`.

[130] A. Snavely, R. Pennington, and R. Loft. Workshop report: Petascale computing in the geosciences. `www.sdsc.edu/~allans`.

[131] MSC Software. Simenterprise extending simulation to the enterprise. 2007.

[132] Guy Steele. The Future Is Parallel: What's a Programmer to Do? Breaking Sequential Habits of Thought. One-hour talk presented at the 5 March 2009 meeting of the New England Programming Languages and Systems (NEPLS) Symposium at Mitre. `http://research.sun.com/projects/plrg/Publications/NEPLSMarch2009Steele.pdf`, 2009.

[133] *Draft OSD Standard. T10 Committee.* Storage Networking Industry Association (SNIA)., July 2004.

[134] Alexander Szalay and Jim Gray. 2020 computing: Science in an exponential world. *Nature*, 440(7083):413–414, Mar 2006.

[135] Nathan R. Tallent and John M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[136] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 441–452, New York, NY, USA, 2009. ACM.

[137] Ani R. Thakar. Lessons learned from the sdss catalog archive server. *Computing in Science and Engineering*, 10(6):65–71, 2008.

[138] Christophe Tretz. Cmos transistor sizing for minimization of energy-delay product. In *GLSVLSI '96: Proceedings of the 6th Great Lakes Symposium on VLSI*, page 168, Washington, DC, USA, 1996. IEEE Computer Society.

[139] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[140] Bryan Veal and Annie Foong. Performance scalability of a multi-core web server. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66, New York, NY, USA, 2007. ACM.

[141] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical mpi programs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 261–270, New York, NY, USA, 2009. ACM.

[142] Thorsten von Eicken, David E. Culler, Klaus Erik Schauser, and Seth Copen Goldstein. Retrospective: active messages: a mechanism for integrating computation and communication. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 83–84, New York, NY, USA, 1998. ACM.

[143] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Miller, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[144] Michael E. Wolf and Monica S. Lam. A Data Locality Optimization Algorithm. *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 30–44, June 1991.

[145] Michael E. Wolf and Monica S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[146] C. Wunsch and P. Heimbach. Practical global oceanic state estimation. *Physica D Nonlinear Phenomena*, 230:197–208, June 2007.

[147] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing, LCPC'09*, 2009.

[148] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.

[149] D.G. York. The sloan digital sky survey astronomical journal. `http://www.sdss.org/`, 2000.

[150] Daniel R. Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829, 2008.

[151] Shujia Zhou, Amidu Oloso, Megan Damon, and Tom Clune. Application controlled parallel asynchronous io. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 178, New York, NY, USA, 2006. ACM.