The Landscape of Parallel Computing Research: A View from Berkeley

Krste Asanovic
Ras Bodik
Bryan Christopher Catanzaro
Joseph James Gebis
Parry Husbands
Kurt Keutzer
David A. Patterson
William Lester Plishker
John Shalf
Samuel Webb Williams
Katherine A. Yelick

Electrical Engineering and Computer Sciences
University of California at Berkeley

December 18, 2006

Acknowledgement

The Landscape of Parallel Computing Research:

A View from Berkeley

Krste Asanovíc, Rastislav Bodik, Bryan Catanzaro, Joseph Gebis,
Parry Husbands, Kurt Keutzer, David Patterson,
William Plishker, John Shalf, Samuel Williams, and Katherine Yelick

December 18, 2006

Abstract

The recent switch to parallel microprocessors is a milestone in the history of computing. Industry has laid out a roadmap for multicore designs that preserves the programming paradigm of the past via binary compatibility and cache coherence. Conventional wisdom is now to double the number of cores on a chip with each silicon generation.

A multidisciplinary group of Berkeley researchers met nearly two years to discuss this change. Our view is that this evolutionary approach to parallel hardware and software may work from 2 or 8 processor systems, but is likely to face diminishing returns as 16 and 32 processor systems are realized, just as returns fell with greater instruction-level parallelism.

We believe that much can be learned by examining the success of parallelism at the extremes of the computing spectrum, namely embedded computing and high performance computing. This led us to frame the parallel landscape with seven questions, and to recommend the following:

- The overarching goal should be to make it easy to write programs that execute efficiently on highly parallel computing systems
- The target should be 1000s of cores per chip, as these chips are built from processing elements that are the most efficient in MIPS (Million Instructions per Second) per watt, MIPS per area of silicon, and MIPS per development dollar.
- Instead of traditional benchmarks, use 13 "Dwarfs" to design and evaluate parallel programming models and architectures. (A dwarf is an algorithmic method that captures a pattern of computation and communication.)
- "Autotuners" should play a larger role than conventional compilers in translating parallel programs.
- To maximize programmer productivity, future programming models must be more human-centric than the conventional focus on hardware or applications.
- To be successful, programming models should be independent of the number of processors.
- To maximize application efficiency, programming models should support a wide range of data types and successful models of parallelism: task-level parallelism, word-level parallelism, and bit-level parallelism.
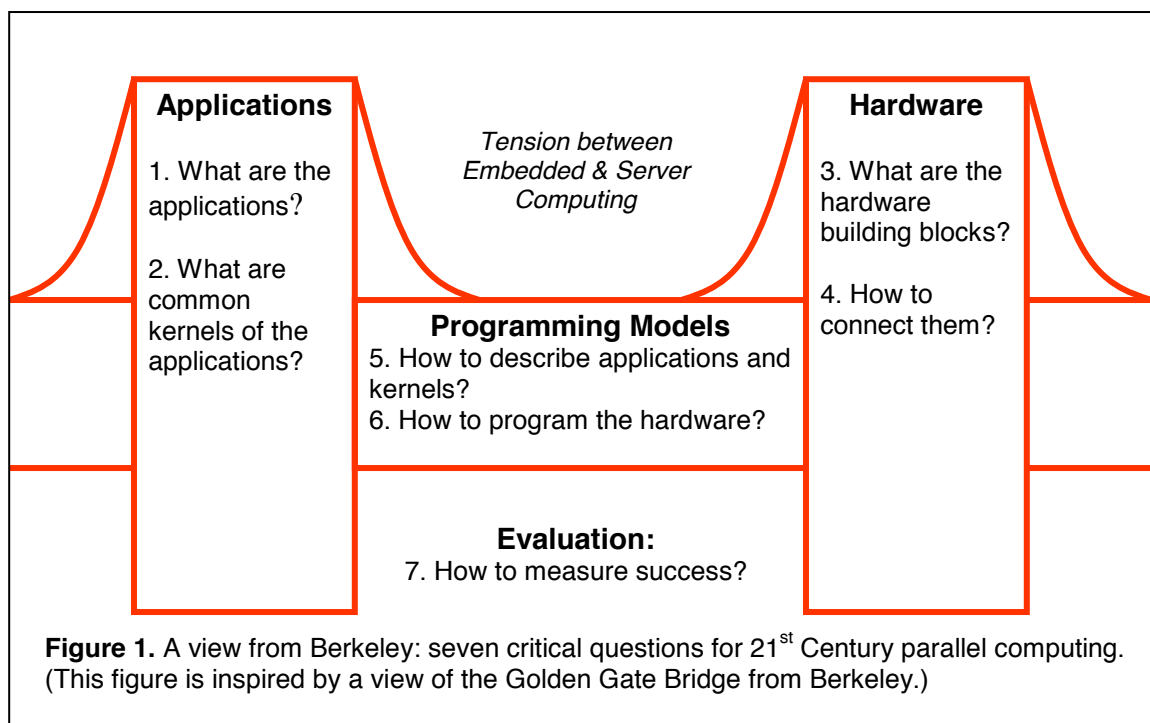
- Architects should not include features that significantly affect performance or energy if programmers cannot accurately measure their impact via performance counters and energy counters.
- Traditional operating systems will be deconstructed and operating system functionality will be orchestrated using libraries and virtual machines.
- To explore the design space rapidly, use system emulators based on Field Programmable Gate Arrays (FPGAs) that are highly scalable and low cost.

Since real world applications are naturally parallel and hardware is naturally parallel, what we need is a programming model, system software, and a supporting architecture that are naturally parallel. Researchers have the rare opportunity to re-invent these cornerstones of computing, provided they simplify the efficient programming of highly parallel systems.

1.0 Introduction

The computing industry changed course in 2005 when Intel followed the lead of IBM's Power 4 and Sun Microsystems' Niagara processor in announcing that its high performance microprocessors would henceforth rely on multiple processors or cores. The new industry buzzword "*multicore*" captures the plan of doubling the number of standard cores per die with every semiconductor process generation starting with a single processor. Multicore will obviously help multiprogrammed workloads, which contain a mix of independent sequential tasks, but how will individual tasks become faster? Switching from sequential to modestly parallel computing will make programming much more difficult without rewarding this greater effort with a dramatic improvement in power-performance. Hence, multicore is unlikely to be the ideal answer.

A diverse group of University of California at Berkeley researchers from many backgrounds—circuit design, computer architecture, massively parallel computing, computer-aided design, embedded hardware and software, programming languages, compilers, scientific programming, and numerical analysis—met between February 2005 and December 2006 to discuss parallelism from these many angles. We borrowed the good ideas regarding parallelism from different disciplines, and this report is the result. We concluded that sneaking up on the problem of parallelism via multicore solutions was likely to fail and we *desperately* need a new solution for parallel hardware and software.

Applications

1. What are the applications?

2. What are common kernels of the applications?

Tension between Embedded & Server Computing

Hardware

3. What are the hardware building blocks?

4. How to connect them?

Programming Models
5. How to describe applications and kernels?
6. How to program the hardware?

Evaluation:
7. How to measure success?

Figure 1. A view from Berkeley: seven critical questions for 21st Century parallel computing. (This figure is inspired by a view of the Golden Gate Bridge from Berkeley.)

Although compatibility with old binaries and C programs is valuable to industry, and some researchers are trying to help multicore product plans succeed, we have been thinking bolder thoughts. Our aim is to realize thousands of processors on a chip for new applications, and we welcome new programming models and new architectures if they

simplify the efficient programming of such highly parallel systems. Rather than multicore, we are focused on "*manycore*". Successful manycore architectures and supporting software technologies could reset microprocessor hardware and software roadmaps for the next 30 years.

Figure 1 shows the seven critical questions we used to frame the landscape of parallel computing research. We do not claim to have the answers in this report, but we do offer non-conventional and provocative perspectives on some questions and state seemingly obvious but sometimes-neglected perspectives on others.

Note that there is a tension between embedded and high performance computing, which surfaced in many of our discussions. We argue that these two ends of the computing spectrum have more in common looking forward than they did in the past. First, both are concerned with power, whether it is battery life for cell phones or the cost of electricity and cooling in a data center. Second, both are concerned with hardware utilization. Embedded systems are always sensitive to cost, but efficient use of hardware is also required when you spend $10M to $100M for high-end servers. Third, as the size of embedded software increases over time, the fraction of hand tuning must be limited and so the importance of software reuse must increase. Fourth, since both embedded and high-end servers now connect to networks, both need to prevent unwanted accesses and viruses. Thus, the need is increasing for some form of operating system for protection in embedded systems, as well as for resource sharing and scheduling.

Perhaps the biggest difference between the two targets is the traditional emphasis on real-time computing in embedded, where the computer and the program need to be just fast enough to meet the deadlines, and there is no benefit to running faster. Running faster is usually valuable in server computing. As server applications become more media-oriented, real time may become more important for server computing as well. This report borrows many ideas from both embedded and high performance computing.

The organization of the report follows the seven questions of Figure 1. Section 2 documents the reasons for the switch to parallel computing by providing a number of guiding principles. Section 3 reviews the left tower in Figure 1, which represents the new applications for parallelism. It describes the original "Seven Dwarfs", which we believe will be the computational kernels of many future applications. Section 4 reviews the right tower, which is hardware for parallelism, and we separate the discussion into the classical categories of processor, memory, and switch. Section 5 covers programming models and Section 6 covers systems software; they form the bridge that connects the two towers in Figure 1. Section 7 discusses measures of success and describes a new hardware vehicle for exploring parallel computing. We conclude with a summary of our perspectives. Given the breadth of topics we address in the report, we provide 134 references for readers interested in learning more.

In addition to this report, we also started a web site and blog to continue the conversation about the views expressed in this report. See view.eecs.berkeley.edu.
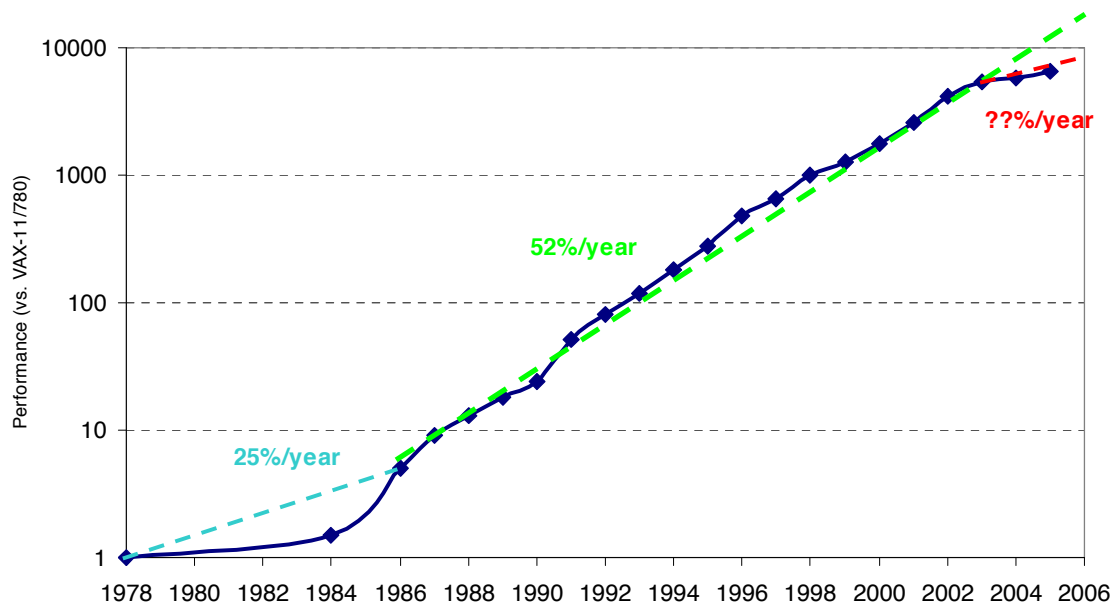
2.0 Motivation

The promise of parallelism has fascinated researchers for at least three decades. In the past, parallel computing efforts have shown promise and gathered investment, but in the end, uniprocessor computing always prevailed. Nevertheless, we argue general-purpose computing is taking an irreversible step toward parallel architectures. What's different this time? This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.

In the following, we capture a number of guiding principles that illustrate precisely how everything is changing in computing. Following the style of *Newsweek*, they are listed as pairs of outdated conventional wisdoms and their new replacements. We later refer to these pairs as CW #*n*.

1. *Old CW*: Power is free, but transistors are expensive.
- *New CW* is the "*Power wall*": Power is expensive, but transistors are "free". That is, we can put more transistors on a chip than we have the power to turn on.
2. *Old CW*: If you worry about power, the only concern is dynamic power.
- *New CW*: For desktops and servers, static power due to leakage can be 40% of total power. (See Section 4.1.)
3. *Old CW*: Monolithic uniprocessors in silicon are reliable internally, with errors occurring only at the pins.
- *New CW*: As chips drop below 65 nm feature sizes, they will have high soft and hard error rates. [Borkar 2005] [Mukherjee et al 2005]
4. *Old CW*: By building upon prior successes, we can continue to raise the level of abstraction and hence the size of hardware designs.
- *New CW*: Wire delay, noise, cross coupling (capacitive and inductive), manufacturing variability, reliability (see above), clock jitter, design validation, and so on conspire to stretch the development time and cost of large designs at 65 nm or smaller feature sizes. (See Section 4.1.)
5. *Old CW*: Researchers demonstrate new architecture ideas by building chips.
- *New CW*: The cost of masks at 65 nm feature size, the cost of Electronic Computer Aided Design software to design such chips, and the cost of design for GHz clock rates means researchers can no longer build believable prototypes. Thus, an alternative approach to evaluating architectures must be developed. (See Section 7.3.)
6. *Old CW*: Performance improvements yield both lower latency and higher bandwidth.
- *New CW*: Across many technologies, bandwidth improves by at least the square of the improvement in latency. [Patterson 2004]
7. *Old CW*: Multiply is slow, but load and store is fast.
- *New CW* is the "*Memory wall*" [Wulf and McKee 1995]: Load and store is slow, but multiply is fast. Modern microprocessors can take 200 clocks to access Dynamic Random Access Memory (DRAM), but even floating-point multiplies may take only four clock cycles.

8. *Old CW*: We can reveal more instruction-level parallelism (ILP) via compilers and architecture innovation. Examples from the past include branch prediction, out-of-order execution, speculation, and Very Long Instruction Word systems.

- *New CW* is the "*ILP wall*": There are diminishing returns on finding more ILP. [Hennessy and Patterson 2007]

9. *Old CW*: Uniprocessor performance doubles every 18 months.

- *New CW* is *Power Wall + Memory Wall + ILP Wall = Brick Wall*. Figure 2 plots processor performance for almost 30 years. In 2006, performance is a factor of three below the traditional doubling every 18 months that we enjoyed between 1986 and 2002. The doubling of uniprocessor performance may now take 5 years.

10. *Old CW*: Don't bother parallelizing your application, as you can just wait a little while and run it on a much faster sequential computer.

- *New CW*: It will be a very long wait for a faster sequential computer (see above).

11. *Old CW*: Increasing clock frequency is the primary method of improving processor performance.

- *New CW*: Increasing parallelism is the primary method of improving processor performance. (See Section 4.1.)

12. *Old CW*: Less than linear scaling for a multiprocessor application is failure.

- *New CW*: Given the switch to parallel computing, any speedup via parallelism is a success.

Figure 2. Processor performance improvement between 1978 and 2006 using integer SPEC [SPEC 2006] programs. RISCs helped inspire performance to improve by 52% per year between 1986 and 2002, which was much faster than the VAX minicomputer improved between 1978 and 1986. Since 2002, performance has improved less than 20% per year. By 2006, processors will be a factor of three slower than if progress had continued at 52% per year. This figure is Figure 1.1 in [Hennessy and Patterson 2007].

Although the CW pairs above paint a negative picture about the state of hardware, there are compensating positives as well. First, Moore's Law continues, so we will soon be able to put thousands of simple processors on a single, economical chip (see Section

4.1.2). For example, Cisco is shipping a product with 188 Reduced Instruction Set Computer (RISC) cores on a single chip in a 130nm process [Eatherton 2005]. Second, communication between these processors within a chip can have very low latency and very high bandwidth. These monolithic manycore microprocessors represent a very different design point from traditional multichip multiprocessors, and so provide promise for the development of new architectures and programming models. Third, the open source software movement means that the software stack can evolve much more quickly than in the past. As an example, note the widespread use of Ruby on Rails. Version 1.0 appeared in just December 2005.

3.0 Applications and Dwarfs

The left tower of Figure 1 is applications. In addition to traditional desktop, server, scientific, and embedded applications, the importance of consumer products is increasing.

We decided to mine the parallelism experience of the high-performance computing community to see if there are lessons we can learn for a broader view of parallel computing. The hypothesis is *not* that traditional scientific computing is the future of parallel computing; it is that the body of knowledge created in building programs that run well on massively parallel computers may prove useful in parallelizing future applications. Furthermore, many of the authors from other areas, such as embedded computing, were surprised at how well future applications in their domain mapped closely to problems in scientific computing.

The conventional way to guide and evaluate architecture innovation is to study a benchmark suite based on existing programs, such as EEMBC (Embedded Microprocessor Benchmark Consortium) or SPEC (Standard Performance Evaluation Corporation) or SPLASH (Stanford Parallel Applications for Shared Memory) [EEMBC 2006] [SPEC 2006] [Singh et al 1992] [Woo et al 1992]. One of the biggest obstacles to innovation in parallel computing is that it is currently unclear how to express a parallel computation best. Hence, it seems unwise to let a set of existing source code drive an investigation into parallel computing. There is a need to find a higher level of abstraction for reasoning about parallel application requirements.

Our goal is to delineate application requirements in a manner that is not overly specific to individual applications or the optimizations used for certain hardware platforms, so that we can draw broader conclusions about hardware requirements. Our approach, described below, is to define a number of "dwarfs", which each capture a pattern of computation and communication common to a class of important applications.

3.1 Seven Dwarfs

We were inspired by the work of Phil Colella, who identified seven numerical methods that he believed will be important for science and engineering for at least the next decade [Colella 2004]. Figure 3 introduces the Seven Dwarfs, which constitute classes where membership in a class is defined by similarity in computation and data movement. The dwarfs are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of applications. Programs that are members of a particular class can

be implemented differently and the underlying numerical methods may change over time, but the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future.

Some evidence for the existence of this particular set of "equivalence classes" can be found in the numerical libraries that have been built around these equivalence classes: for example, FFTW for spectral methods [Frigo and Johnson 1998], LAPACK/ScaLAPACK for dense linear algebra [Blackford et al 1996], and OSKI for sparse linear algebra [Vuduc et al 2006]. We list these in Figure 3, together with the computer architectures that have been purpose-built for particular dwarfs: for example, GRAPE for N-body methods [Tokyo 2006], vector architectures for linear algebra [Russell 1976], and FFT accelerators [Zarlink 2006]. Figure 3 also shows the inter-processor communication patterns exhibited by members of a dwarf when running on a parallel machine [Vetter and McCracken 2001] [Vetter and Yoo 2002] [Vetter and Meuller 2002] [Kamil et al 2005]. The communication pattern is closely related to the memory access pattern that takes place locally on each processor.
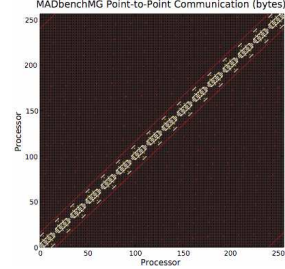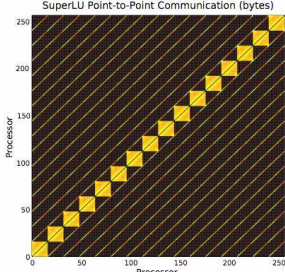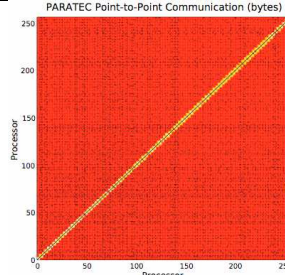
3.2 Finding More Dwarfs

The dwarfs present a method for capturing the common requirements of classes of applications while being reasonably divorced from individual implementations. Although the nomenclature of the dwarfs comes from Phil Colella's discussion of scientific computing applications, we were interested in applying dwarfs to a broader array of computational methods. This led us naturally to the following questions:

- How well do the Seven Dwarfs of high performance computing capture computation and communication patterns for a broader range of applications?
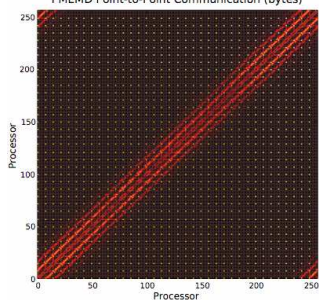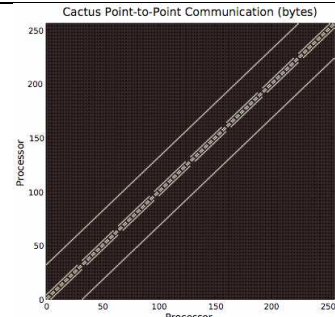- What dwarfs need to be added to cover the missing important areas beyond high performance computing?

If we find that an expanded set of dwarfs is broadly applicable, we can use them to guide innovation and evaluation of new prototypes. As long as the final list contains no more than two- or three-dozen dwarfs, architects and programming model designers can use them to measure success. For comparison, SPEC2006 has 29 benchmarks and EEMBC has 41. Ideally, we would like good performance across the set of dwarfs to indicate that new manycore architectures and programming models will perform well on applications of the future.

Dwarfs are specified at a high level of abstraction that can group related but quite different computational methods. Over time, a single dwarf can expand to cover such a disparate variety of methods that it should be viewed as multiple distinct dwarfs. As long as we do not end up with too many dwarfs, it seems wiser to err on the side of embracing new dwarfs. For example, unstructured grids could be interpreted as a sparse matrix problem, but this would both limit the problem to a single level of indirection and disregard too much additional information about the problem.

Dwarf	Description	Communication Pattern (Figure axes show processors 1 to 256, with black meaning no communication)	NAS Benchmark / Example HW
1. Dense Linear Algebra (e.g., BLAS [Blackford et al 2002], ScaLAPACK [Blackford et al 1996], or MATLAB [MathWorks 2006])	Data are dense matrices or vectors. (BLAS Level 1 = vector-vector; Level 2 = matrix-vector; and Level 3 = matrix-matrix.) Generally, such applications use unit-stride memory accesses to read data from rows, and strided accesses to read data from columns.	The communication pattern of MadBench, which makes heavy use of ScaLAPACK for parallel dense linear algebra, is typical of a much broader class of numerical algorithms	Block Triadiagonal Matrix, Lower Upper Symmetric Gauss-Seidel / Vector computers, Array computers
2. Sparse Linear Algebra (e.g., SpMV, OSKI [OSKI 2006], or SuperLU [Demmel et al 1999])	Data sets include many zero values. Data is usually stored in compressed matrices to reduce the storage and bandwidth requirements to access all of the nonzero values. One example is block compressed sparse row (BCSR). Because of the compressed formats, data is generally accessed with indexed loads and stores.	SuperLU (communication pattern pictured above) uses the BCSR method for implementing sparse LU factorization.	Conjugate Gradient / Vector computers with gather/scatter
3. Spectral Methods (e.g., FFT [Cooley and Tukey 1965])	Data are in the frequency domain, as opposed to time or spatial domains. Typically, spectral methods use multiple butterfly stages, which combine multiply-add operations and a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others.	PARATEC: The 3D FFT requires an all-to-all communication to implement a 3D transpose, which requires communication between every link. The diagonal stripe describes BLAS-3 dominated linear-algebra step required for orthogonalization.	Fourier Transform / DSPs, Zalink PDSP [Zarlink 2006]

Dwarf	Description	Communication Pattern (Figure axes show processors 1 to 256, with black meaning no communication)	NAS Benchmark / Example HW
4. N-Body Methods (e.g., Barnes-Hut [Barnes and Hut 1986], Fast Multipole Method [Greengard and Rokhlin 1987])	Depends on interactions between many discrete points. Variations include particle-particle methods, where every point depends on all others, leading to an $O(N^2)$ calculation, and hierarchical particle methods, which combine forces or potentials from multiple points to reduce the computational complexity to $O(N \log N)$ or $O(N)$.	PMEMD Point-to-Point Communication (bytes) ![chart] PMEMD's communication pattern is that of a particle mesh Ewald calculation.	(no benchmark) / GRAPE [Tokyo 2006], MD-GRAPE [IBM 2006]
5. Structured Grids (e.g., Cactus [Goodale et al 2003] or Lattice-Boltzmann Magneto-hydrodynamics [LBMHD 2005])	Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality. Updates may be in place or between 2 versions of the grid. The grid may be subdivided into finer grids in areas of interest ("Adaptive Mesh Refinement"); and the transition between granularities may happen dynamically.	Cactus Point-to-Point Communication (bytes) ![chart] Communication pattern for Cactus, a PDE solver using 7-point stencil on 3D block-structured grids.	Multi-Grid, Scalar Penta-diagonal / QCDOC [Edinburg 2006], BlueGeneL
6. Unstructured Grids (e.g., ABAQUS [ABAQUS 2006] or FIDAP [FLUENT 2006])	An irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighboring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighboring points, and then loading values from those neighboring points.		Unstructured Adaptive / Vector computers with gather/scatter, Tera Multi Threaded Architecture [Berry et al 2006]
7. Monte Carlo (e.g., Quantum Monte Carlo [Aspuru-Guzik et al 2005])	Calculations depend on statistical results of repeated random trials. Considered embarrassingly parallel.	Communication is typically not dominant in Monte Carlo methods.	Embarrassingly Parallel / NSF Teragrid

Figure 3. Seven Dwarfs, their descriptions, corresponding NAS benchmarks, and example computers.

To investigate the general applicability of the Seven Dwarfs, we compared the list against other collections of benchmarks: EEMBC from embedded computing and from SPEC2006 for desktop and server computing. These collections were independent of our study, so they act as validation for whether our small set of computational kernels are good targets for applications of the future. We will describe the final list in detail in Section 3.5, but from our examination of the 41 EEMBC kernels and the 26 SPEC2006 programs, we found four more dwarfs to add to the list:

- o **Combinational Logic** generally involves performing simple operations on very large amounts of data often exploiting bit-level parallelism. For example, computing Cyclic Redundancy Codes (CRC) is critical to ensure integrity and RSA encryption for data security.
- o **Graph Traversal** applications must traverse a number of objects and examine characteristics of those objects such as would be used for search. It typically involves indirect table lookups and little computation.
- o **Graphical Models** applications involve graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and Hidden Markov Models.
- o **Finite State Machines** represent an interconnected set of states, such as would be used for parsing. Some state machines can decompose into multiple simultaneously active state machines that can act in parallel.

To go beyond to EEMBC and SPEC, we examined three increasingly important application domains to see if we should increase the number of dwarfs: machine learning, database software, and computer graphics and games.

3.2.1 Machine Learning

One of the most promising areas for the future of computing is the use of statistical machine learning to make sense from the vast amounts of data now available due to faster computers, larger disks, and the use of the Internet to connect them all together.

Michael Jordan and Dan Klein, our local experts in machine learning, found two dwarfs that should be added to support machine learning:

- o **Dynamic programming** is an algorithmic technique that computes solutions by solving simpler overlapping subproblems. It is particularly applicable for optimization problems where the optimal result for a problem is built up from the optimal result for the subproblems.
- o **Backtrack and Branch-and-Bound**: These involve solving various search and global optimization problems for intractably large spaces. Some implicit method is required in order to rule out regions of the search space that contain no interesting solutions. Branch and bound algorithms work by the divide and conquer principle: the search space is subdivided into smaller subregions ("branching"), and bounds are found on all the solutions contained in each subregion under consideration.

Many other well-known machine-learning algorithms fit into the existing dwarfs:

- o *Support Vector Machines* [Cristianini and Shawe-Taylor 2000]: Dense linear algebra.
- o *Principal Component Analysis* [Duda and Hart 1973]: Dense or sparse linear algebra, depending on the details of implementation.
- o *Decision Trees* [Poole et al 1998]: Graph traversal.
- o *Hashing*: Combinational logic.

3.2.2 Database Software

Jim Gray of Microsoft Research believes sort is at the heart of modern databases. He sponsors an annual competition to see who can come up with the fastest sorter assuming the data is on the disk at the beginning and end. You win MinuteSort by sorting the most data in a minute, organized as 100-byte records. The 2006 winner sorted 400 million records (40 GB) on a 32-way shared memory multiprocessor using 1.6 GHz Itanium 2 processors with 128 GB of main memory and 128 disks. Its uses a commercial sorting package called Nsort, which does sorts either the records directly or pointers to records. [Nyberg et al 2004] The sorting algorithm is sample sort. While it will be important to have efficient interfaces between I/O and main memory to sort large files fast, sorting does not add to our list of dwarfs.

Another important function of modern databases is hashing. Unlike a typical hash, a database hash will compute over a lot of data, perhaps half of main memory. Once again, these computation and communication patterns do not expand the dwarfs.

Joe Hellerstein, our local expert in databases, said the future of databases was large data collections typically found on the Internet. A key primitive to explore such collections is MapReduce, developed and widely used at Google. [Dean and Ghemawat 2004] The first phase maps a user supplied function onto thousands of computers, processing key/value pairs to generate a set of intermediate key/value pairs, The second phase reduces the returned values from all those thousands of instances into a single result by merging all intermediate values associated with the same intermediate key. Note that these two phases are highly parallel yet simple to understand. Borrowing the name from a similar function in Lisp, they call this primitive "MapReduce".

MapReduce is a more general version of the pattern we had previously called "Monte Carlo": the essence is a single function that executes in parallel on independent data sets, with outputs that are eventually combined to form a single or small number of results. In order to reflect this broader scope, we changed the name of the dwarf to "MapReduce".

A second thrust for the future of databases was in genetics, exemplified by the widely popular BLAST (Basic Local Alignment Search Tool) code. [Altschul et al 1990] BLAST is a heuristic method used to find areas of DNA/protein sequences that are similar from a database. There are three main steps:

1. Compile a list of high-scoring words from the sequence
2. Scan database for hits from this list
3. Extend the hits to optimize the match

Although clearly important, BLAST did not extend our list of dwarfs.

3.2.3 Computer Graphics and Games

While the race to improve realism has pushed graphics processing unit (GPU) performance up into the Teraflops range, graphical realism is not isolated to drawing polygons and textures on the screen. Rather, modeling of the physical processes that govern the behavior of these graphical objects requires many of the same computational models used for large-scale scientific simulations. The same is true for many tasks in computer vision and media processing, which form the core of the "applications of the future" driving the technology roadmaps of hardware vendors.

Employing on-chip parallelism to accelerate computer graphics is considered a solved problem for all practical purposes via GPUs. The principle burden for the host processor at this point centers on modeling the physical properties of the graphical elements that comprise the game or the user interface. Realistic physics requires computational modeling of physical processes that are essentially the same as those required for scientific computing applications. The computational methods employed are very much like those that motivate the seven original dwarfs.

For instance, modeling of liquids and liquid behavior used for special effects in movies are typically done using particle methods such as Smooth Particle Hydrodynamics (SPH) [Monaghan 1982]. The rendering of the physical model is still done in OpenGL using GPUs or software renderers, but the underlying model of the flowing shape of the liquid requires the particle-based fluid model. There are several other examples where the desire to model physical properties in game and graphics map onto the other dwarfs:

- o Reverse kinematics requires a combination of sparse matrix computations and graph traversal methods.
- o Spring models, used to model any rigid object that deforms in response to pressure or impact such as bouncing balls or Jell-O, use either sparse matrix or finite-element models.
- o Collision detection is a graph traversal operation as are the Octrees and Kd trees employed for depth sorting and hidden surface removal.
- o Response to collisions is typically implemented as a finite-state machine.

Hence, the surprising conclusion is that games and graphics did not extend the drawfs beyond the 13 identified above.

One encouraging lesson to learn from the GPUs and graphics software is that the APIs do not directly expose the programmer to concurrency. OpenGL, for instance, allows the programmer to describe a set of "vertex shader" operations in Cg (a specialized language for describing such operations) that are applied to every polygon in the scene without having to consider how many hardware fragment processors or vertex processors are available in the hardware implementation of the GPU.

3.2.4 Summarizing the Next Six Dwarfs

Figure 4 shows six more dwarfs that were added because of the studies in the prior section. Note that we consider the algorithms independent of the data sizes and types (see Section 5.3).

Dwarf	Description
8. Combinational Logic (e.g., encryption)	Functions that are implemented with logical functions and stored state.
9. Graph traversal (e.g., Quicksort)	Visits many nodes in a graph by following successive edges. These applications typically involve many levels of indirection, and a relatively small amount of computation.
10. Dynamic Programming	Computes a solution by solving simpler overlapping subproblems. Particularly useful in optimization problems with a large set of feasible solutions.
11. Backtrack and Branch+Bound	Finds an optimal solution by recursively dividing the feasible region into subdomains, and then pruning subproblems that are suboptimal.
12. Construct Graphical Models	Constructs graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and Hidden Markov Models.
13. Finite State Machine	A system whose behavior is defined by states, transitions defined by inputs and the current state, and events associated with transitions or states.

Figure 4. Extensions to the original Seven Dwarfs.

Although 12 of the 13 Dwarfs possess some form of parallelism, finite state machines (FSMs) look to be a challenge, which is why we made them the last dwarf. Perhaps FSMs will prove to be *embarrassingly sequential* just as MapReduce is embarrassingly parallel. If it is still important and does not yield to innovation in parallelism, that will be disappointing, but perhaps the right long-term solution is to change the algorithmic approach. In the era of multicore and manycore. Popular algorithms from the sequential computing era may fade in popularity. For example, if Huffman decoding proves to be embarrassingly sequential, perhaps we should use a different compression algorithm that is amenable to parallelism.

In any case, the point of the 13 Dwarfs is *not* to identify the low hanging fruit that are highly parallel. The point is to identify the kernels that are the core computation and communication for important applications in the upcoming decade, independent of the amount of parallelism. To develop programming systems and architectures that will run applications of the future as efficiently as possible, we must learn the limitations as well as the opportunities. We note, however, that inefficiency on embarrassingly parallel code could be just as plausible a reason for the failure of a future architecture as weakness on embarrassingly sequential code.

More dwarfs may need to be added to the list. Nevertheless, we were surprised that we only needed to add six dwarfs to cover such a broad range of important applications.

3.3 Composition of Dwarfs

Any significant application, such as an MPEG4 (Moving Picture Experts Group) decoder or an IP (Internet Protocol) forwarder, will contain multiple dwarfs that each consume a significant percentage of the application's computation. Hence, the performance of a large application will depend not only on each dwarf's performance, but also on how dwarfs are composed together on the platform.

The collection of dwarfs comprising an application can be distributed on a multiprocessor platform in two different ways:

1. *Temporally distributed* or time-shared on a common set of processors, or
2. *Spatially distributed* or space-shared, with each dwarf uniquely occupying one or more processors.

The selection of temporal or spatial distribution will in part depend on the structure of communication between dwarfs. For example, some applications are structured as a number of serial phases, where each phase is a dwarf that must complete before we start the next. In this case, it would be natural to use time multiplexing to allocate the whole set of processors to each phase. Other applications can be structured as a network of communicating dwarfs running concurrently, in which case it would be natural to distribute the dwarfs spatially across the available processors.

The two forms of distribution can be applied hierarchically. For example, a dwarf may be implemented as a pipeline, where the computation for an input is divided into stages with each stage running on its own spatial division of the processors. Each stage is time multiplexed across successive inputs, but processing for a single input flows through the spatial distribution of pipeline stages.

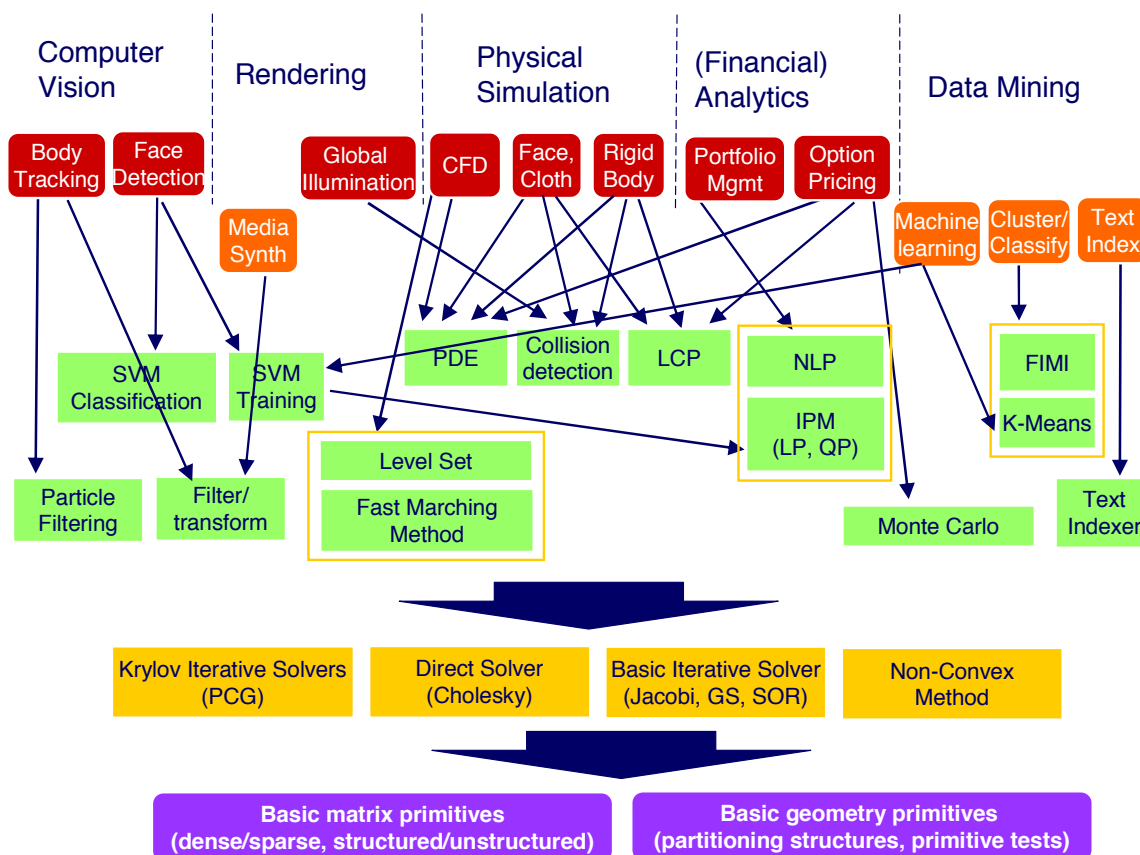Two software issues arise when considering the composition of dwarfs:

1. The choice of composition model—how the dwarfs are put together to form a complete application. The scientific software community has recently begun the move to component models [Bernholdt et al 2002]. In these models, however, individual modules are not very tightly coupled and this may affect the efficiency of the final application.
2. Data structure translation. Various algorithms may have their own preferred data structures, such as recursive data layouts for dense matrices. This may be at odds with the efficiency of composition, as working sets may have to be translated before use by other dwarfs.

These issues are pieces of a larger puzzle. What are effective ways to describe composable parallel-code libraries? Can we write a library such that it encodes knowledge about its ideal mapping when composed with others in a complete parallel application? What if the ideal mapping is heavily dependent on input data that cannot be known at compile time?

3.4 Intel Study

Intel believes that the increase in demand for computing will come from processing the massive amounts of information available in the "Era of Tera". [Dubey 2005] Intel classifies the computation into three categories: Recognition, Mining, and Synthesis, abbreviated as RMS. *Recognition* is a form of machine learning, where computers examine data and construct mathematical models of that data. Once the computers construct the models, *Mining* searches the web to find instances of that model. *Synthesis* refers to the creation of new models, such as in graphics. Hence, RMS is related to our examination of machine learning, databases, and graphics in Sections 3.2.1 to 3.3.3.

The common computing theme of RMS is "multimodal recognition and synthesis over large and complex data sets" [Dubey 2005]. <mark>Intel believes RMS will find important applications in medicine, investment, business, gaming, and in the home.</mark> Intel's efforts in Figure 5 show that Berkeley is not alone in trying to organize the new frontier of computation to underlying computation kernels in order to guide architectural research.

Figure 5. Intel's RMS and how it maps down to functions that are more primitive. Of the five categories at the top of the figure, Computer Vision is classified as Recognition, Data Mining is Mining, and Rendering, Physical Simulation, and Financial Analytics are Synthesis. [Chen 2006]

3.5 Dwarfs Summary

Figure 6 summarizes our investigation and shows the presence of the 13 Dwarfs in a diverse set of application benchmarks including EEMBC, SPEC2006, machine learning, graphics/games, database software, and Intel's RMS. As mentioned above, several of the programs use multiple dwarfs, and so they are listed in multiple categories. We do not believe that our list of dwarfs is yet complete, and we anticipate the addition of more dwarfs in the future. At the same time we are surprised at what a diverse set of important applications is supported by a modest number of dwarfs.

Dwarf	Embedded Computing	General Purpose Computing	Machine Learning	Graphics / Games	Databases	Intel RMS
1. Dense Linear Algebra (e.g., BLAS or MATLAB)	*EEMBC Automotive*: iDCT, FIR, IIR, Matrix Arith; *EEMBC Consumer*: JPEG, RGB to CMYK, RGB to YIQ; *EEMBC Digital Entertainment*: RSA MP3 Decode, MPEG-2 Decode, MPEG-2 Encode, MPEG-4 Decode; MPEG-4 Encode; *EEMBC Networking*: IP Packet; *EEMBC Office Automation*: Image Rotation; *EEMBC Telecom*: Convolution Encode; *EEMBC Java:* PNG	*SPEC Integer*: Quantum computer simulation (libquantum), video compression (h264avc) *SPEC Fl. Pl.*: Hidden Markov models (sphinx3)	Support vector machines, princpal component analysis, independent component analysis		Database hash accesses large contiguous sections of memory	Body Tracking, media synthesis linear programming, K-means, support vector machines, quadratic programming, *PDE*: Face, *PDE*: Cloth*
2. Sparse Linear Algebra (e.g., SpMV, OSKI, or SuperLU)	*EEMBC Automotive*: Basic Int + FP, Bit Manip, CAN Remote Data, Table Lookup, Tooth to Spark; *EEMBC Telecom*: Bit Allocation; *EEMBC Java:* PNG	*SPEC Fl. Pt.*: Fluid dynamics (bwaves), quantum chemistry (gamess; tonto), linear program solver (soplex)	Support vector machines, principal component analysis, independent component analysis	Reverse kinematics; Spring models		Support vector machines, quadratic programming, *PDE*: Face, *PDE*: Cloth* *PDE:* Computational fluid dynamics
3. Spectral Methods (e.g., FFT)	*EEMBC Automotive*: FFT, iFFT, iDCT; *EEMBC Consumer*: JPEG; *EEMBC Entertainment*: MP3 Decode		Spectral clustering	Texture maps		*PDE:* Computational fluid dynamics *PDE:* Cloth
4. N-Body Methods (e.g., Barnes-Hut, Fast Multipole Method)		*SPEC Fl. Pt.*: Molecular dynamics (gromacs, 32-bit; namd, 64-bit)				
5. Structured Grids (e.g., Cactus or Lattice-Boltzmann Magneto-	*EEMBC Automotive*: FIR, IIR; *EEMBC Consumer*: HP Gray-Scale; *EEMBC Consumer*: JPEG; *EEMBC Digital Entertainment*: MP3 Decode, MPEG-2 Decode, MPEG-2 Encode, MPEG-4 Decode; MPEG-4 Encode; *EEMBC Office Automation*:	*SPEC Fl. Pt.*: Quantum chromodynamics (milc),magneto hydrodynamics (zeusmp), general relativity (cactusADM), fluid dynamics (leslie3d-AMR; lbm), finite element methods (dealII-AMR; calculix), Maxwell's E&M		Smoothing; interpolation		

Dwarf	Embedded Computing	General Purpose Computing	Machine Learning	Graphics / Games	Databases	Intel RMS
hydrodynamics)	Dithering; *EEMBC Telecom*: Autocorrelation	eqns solver (GemsFDTD), quantum crystallography (tonto), weather modeling (wrf2-AMR)				
6. Unstructured Grids (e.g., ABAQUS or FIDAP)			Belief propagation			Global illumination
7. MapReduce (e.g., Monte Carlo)		*SPEC Fl. Pt.*: Ray tracer (povray)	Expectation maximization		MapReduce	
8. Combinational Logic	*EEMBC Digital Entertainment*: AES, DES ; *EEMBC Networking*: IP Packet, IP NAT, Route Lookup; *EEMBC Office Automation*: Image Rotation; *EEMBC Telecom*: Convolution Encode		Hashing		Hashing	
9. Graph Traversal	*EEMBC Automotive*: Pointer Chasing, Tooth to Spark; *EEMBC Networking*: IP NAT, OSPF, Route Lookup; *EEMBC Office Automation*: Text Processing; *EEMBC Java:* Chess, XML Parsing		Bayesian networks, decision trees	Reverse kinematics, collision detection, depth sorting, hidden surface removal	Transitive closure	Natural language processing
10. Dynamic Programming	*EEMBC Telecom*: Viterbi Decode	*SPEC Integer:* Go (gobmk)	Forward-backward, inside-outside, variable elimination, value iteration		Query optimization	
11. Back-track and Branch +Bound		*SPEC Integer:* Chess (sjeng), network simplex algorithm (mcf), 2D path finding library (astar)	Kernel regression, constraint satisfaction, satisficability			
12. Graphical Models	*EEMBC Telecom*: Viterbi Decode	*SPEC Integer:* Hidden Markov models (hmmer)	Hidden Markov models			

Dwarf	Embedded Computing	General Purpose Computing	Machine Learning	Graphics / Games	Databases	Intel RMS
13. Finite State Machine	*EEMBC Automotive*: Angle To Time, Cache "Buster", CAN Remote Data, PWM, Road Speed, Tooth to Spark; *EEMBC Consumer*: JPEG; *EEMBC Digital Entertainment*: Huffman Decode, MP3 Decode, MPEG-2 Decode, MPEG-2 Encode, MPEG-4 Decode; MPEG-4 Encode; *EEMBC Networking*: QoS, TCP; *EEMBC Office Automation*: Text Processing; *EEMBC Telecom*: Bit Allocation; *EEMBC Java:* PNG	*SPEC Integer*: Text processing (perlbench), compression (bzip2), compiler (gcc), video compression (h264avc), network discrete event simulation (omnetpp), XML transformation (xalancbmk)		Response to collisions		

Figure 6. Mapping of EEMBC, SPEC2006, Machine Learning, Graphcs/Games, Data Base, and Intel's RMS to the 13 Dwarfs. *Note that SVM, QP, PDE:Face, and PDE:Cloth may use either dense or sparse matrices, depending on the application.

4.0 Hardware

Now that we have given our views of applications and dwarfs for parallel computing in the left tower of Figure 1, we are ready for examination of the right tower: hardware. Section 2 above describes the constraints of present and future semiconductor processes, but they also present many opportunities.

We organize our observations on hardware around the three components first used to describe computers more than 30 years ago: processor, memory, and switch [Bell and Newell 1970].

4.1 Processors: Small is Beautiful

In the development of many modern technologies, such as steel manufacturing, we can observe that there were prolonged periods during which bigger equated to better. These periods of development are easy to identify: The demonstration of one *tour de force* of engineering is only superseded by an even greater one. Due to diminishing economies of scale or other economic factors, the development of these technologies inevitably hit an inflection point that forever changed the course of development. We believe that the development of general-purpose microprocessors is hitting just such an inflection point.

New Conventional Wisdom #4 in Section 2 states that the size of module that we can successfully design and fabricate is shrinking. New Conventional Wisdoms #1 and #2 in Section 2 state that power is proving to be the dominant constraint for present and future generations of processing elements. To support these assertions we note that several of the next generation processors, such as the Tejas Pentium 4 processor from Intel, were canceled or redefined due to power consumption issues [Wolfe 2004]. Even representatives from Intel, a company generally associated with the "higher clock-speed is better" position, warned that traditional approaches to maximizing performance through maximizing clock speed have been pushed to their limit [Borkar 1999] [Gelsinger 2001]. In this section, we look past the inflection point to ask: What processor is the best building block with which to build future multiprocessor systems?

There are numerous advantages to building future microprocessors systems out of smaller processor building blocks:
- Parallelism is an energy-efficient way to achieve performance [Chandrakasan et al 1992].
- Many small cores give the highest performance per unit area for parallel codes.
- A larger number of smaller processing elements allows a finer-grained ability to perform dynamic voltage scaling and power down.
- A small processing element is an economical element that is easy to shut down in the face of catastrophic defects and easier to reconfigure in the face of large parametric variation. The Cisco Metro chip [Eatherton 2005] adds four redundant processors to each die, and Sun sells 4-processor, 6-processor, or 8-processor versions of Niagara based on the yield of a single 8-processor design. Graphics processors are also reported to be using redundant processors in this way, as is the

IBM Cell microprocessor for which only 7 out of 8 synergistic processors are used in the Sony Playstation 3 game machine.
- A small processing element with a simple architecture is easier to design and functionally verify. In particular, it is more amenable to formal verification techniques than complex architectures with out-of-order execution.
- Smaller hardware modules are individually more power efficient and their performance and power characteristics are easier to predict within existing electronic design-automation design flows [Sylvester and Keutzer 1998] [Sylvester and Keutzer 2001] [Sylvester et al 1999].

While the above arguments indicate that we should look to smaller processor architectures for our basic building block, they do not indicate precisely what circuit size or processor architecture will serve us the best. We argued that we must move away from a simplistic "bigger is better" approach; however, that does not immediately imply that "smallest is best".

4.1.1 What processing element is optimum?

Determining the optimum processing element will entail the solution, or at least approximating the solution, of a multivariable optimization problem that is dependent on the application, environment for deployment, workload, constraints of the target market, and fabrication technology. It is clear, however, that the tradeoff between performance and power will be of central importance across the entire spectrum of system applications for current and future multiprocessor systems.

It is important to distinguish between energy (Joules) and power (Joules/second or Watts), which is the rate of consuming energy. Energy per task is usually a metric to be minimized in a design, whereas peak power consumption is usually treated as a design constraint. The energy used by a computation affects the battery life of a mobile device, and the cost of powering a server farm. Peak power determines the cost of packaging and cooling the processor, and these costs rise as a steep step-function of the amount of power to be dissipated. Chip temperature must be limited to avoid excessive leakage power. High chip temperature may also lead to a reduced lifetime due to electromigration and other high temperature reliability issues. Reasonable upper limits for peak power consumption may be 150W for air-cooled server and desktop chips, 40W for a laptop, and 2W for low cost/low power embedded applications.

Different applications will present different tradeoffs between performance and energy consumption. For example, many real-time tasks (e.g., viewing a DVD movie on a laptop) have a fixed performance requirement for which we seek the lowest energy implementation. Desktop processors usually seek the highest performance under a maximum power constraint. Note that the design with the lowest energy per operation might not give the highest performance under a power constraint, if the design cannot complete tasks fast enough to exhaust the available power budget.

If all tasks were highly parallelizable and silicon area was free, we would favor cores with the lowest energy per instruction (SPEC/Watt). However, we also require good

performance on less parallel codes, and high throughput per-unit-area to reduce die costs. The challenge is to increase performance without adversely increasing energy per operation or silicon area.

The effect of microarchitecture on energy and delay was studied in [Gonzalez and Horowitz 1996]. Using energy-delay product ($SPEC^2/W$) as a metric, the authors determined that simple pipelining is significantly beneficial to delay while increasing energy only moderately. In contrast, superscalar features adversely affected the energy-delay product. The power overhead needed for additional hardware did not outweigh the performance benefits. Instruction-level parallelism is limited, so microarchitectures attempting to gain performance from techniques such as wide issue and speculative execution achieved modest increases in performance at the cost of significant area and energy overhead.

The optimal number of pipeline stages in a microarchitecture has been investigated by a number of researchers [Hrishikesh et al 2002] [Srinivasan et al 2002] [Harstein and Puzak 2003] [Heo and Asanovic 2004]. These results are summarized and reviewed in [Chinnery 2006]. Note that to date uniprocessor benchmarks, such as SPEC, have been the most common benchmarks for measuring computational and energy efficiency. We believe that future benchmark sets must evolve to reflect a more representative mix of applications, including parallel codes based on the 13 dwarfs, to avoid over-optimization for single-thread performance. As the results mentioned above have dependencies on process technology, logic family, benchmark set, and workload it is hard to generalize the results for our purposes. However, a review of this literature together with an analysis of empirical data on existing architectures gathered by Horowitz [Horowitz 2006], Paulin [Paulin 2006], and our own investigations [Chong and Catanzaro 2006] indicates that shallower pipelines with in-order execution have proven to be the most area and energy efficient. Given these physical and microarchitectural considerations, we believe the efficient building blocks of future architectures are likely to be simple, modestly pipelined (5-9 stages) processors, floating point units, vector, and SIMD processing elements. Note that these constraints fly in the face of the conventional wisdom of simplifying parallel programming by using the largest processors available.

4.1.2 Will we really fit 1000s of cores on one economical chip

This significant reduction in the size and complexity of the basic processor building block of the future means that many more cores can be economically implemented on a single die; furthermore, this number can double with each generation of silicon. For example, the "manycore" progression might well be 128, 256, 512, ... cores instead of the current "multicore" plan of 2, 4, 8, ... cores over the same semiconductor process generations.

There is strong empirical evidence that we can achieve 1000 cores on a die when 30nm technology is available. (As Intel has taped out a 45-nm technology chip, 30 nm is not so distant in the future.) Cisco today embeds in its routers a network processor with 188 cores implemented in 130 nm technology. [Eatherton 2005] This chip is 18mm by 18mm,

dissipates 35W at a 250MHz clock rate, and produces an aggregate 50 billion instructions per second. The individual processor cores are 5-stage Tensilica processors with very small caches, and the size of each core is 0.5 mm². About a third of the die is DRAM and special purpose functions. Simply following scaling from Moore's Law would arrive at 752 processors in 45nm and 1504 in 30nm. Unfortunately, power may not scale down with size, but we have ample room before we push the 150W limit of desktop or server applications.

4.1.3 Does one size fit all?

We would like to consider briefly the question as to whether multiprocessors of the future will be built as collections of identical processors or assembled from diverse heterogeneous processing elements. Existing embedded multiprocessors, such as the Intel IXP network processing family, keep at least one general-purpose processor on the die to support various housekeeping functions and to provide the hardware base for more general (e.g. Linux) operating system support. Similarly, the IBM Cell has one general-purpose processor and eight tailored processing elements. Keeping a larger processor on chip may help accelerate "inherently sequential" code segments or workloads with fewer threads [Kumar et al 2003].

As Amdahl observed 40 years ago, the less parallel portion of a program can limit performance on a parallel computer [Amdahl 1967]. Hence, one reason to have different "sized" processors in a manycore architecture is to improve parallel speedup by reducing the time it takes to run the less parallel code. For example, assume 10% of the time a program gets no speed up on a 100-processor computer. Suppose to run the sequential code twice as fast, a single processor would need 10 times as many resources as a simple core runs due to bigger power budget, larger caches, a bigger multiplier, and so on. Could it be worthwhile? Using Amdahl's Law [Hennessy and Patterson 2007], the comparative speedups of a homogeneous 100 simple processor design and a heterogeneous 91-processor design relative to a single simple processor are:

SpeedupHomogeneous = $1 / (0.1 - 0.9/100) = 9.2$ times faster
SpeedupHeterogeneous = $1 / (0.1/2 - 0.9/90) = 16.7$ times faster

In this example, even if a single larger processor needed 10 times as many resources to run twice as fast, it would be much more valuable than 10 smaller processors it replaces.

In addition to helping with Amdahl's Law, heterogeneous processor solutions can show significant advantages in power, delay, and area. Processor instruction-set configurability [Killian et al 2001] is one approach to realizing the benefits of processor heterogeneity while minimizing the costs of software development and silicon implementation, but this requires custom fabrication of each new design to realize the performance benefit, and this is only economically justifiable for large markets.

Implementing customized soft-processors in pre-defined reconfigurable logic is another way to realize heterogeneity in a homogenous implementation fabric; however, current area (40X), power (10X), and delay (3X) overheads [Kuon and Rose 2006] appear to make this approach prohibitively expensive for general-purpose processing. A promising

approach that supports processor heterogeneity is to add a reconfigurable coprocessor as a separate chip [Hauser and Wawrzynek 1997] [Arnold 2005]. This obviates the need for new custom silicon. Current data is insufficient to determine whether such approaches can provide energy-efficient solutions.

On the other hand, a single replicated processing element has many advantages; in particular, it offers ease of silicon implementation and a regular software environment. Managing heterogeneity in an environment with thousands of threads may make a difficult problem impossible.

Will the possible power and area advantages of heterogeneous multicores win out versus the flexibility and software advantages of homogeneous multicores? Alternatively, will the processor of the future be like a transistor: a single building block that can be woven into arbitrarily complex circuits? Alternatively, will a processor be more like a NAND gate in a standard-cell library: one instance of a family of hundreds of closely related but unique devices? In this section, we do not claim to have resolved these questions. Rather our point is that resolution of these questions is certain to require significant research and experimentation, and the need for this research is more imminent than industry's multicore multiprocessor roadmap would otherwise indicate.

4.2 Memory Unbound

The DRAM industry has dramatically lowered the price per gigabyte over the decades, to $100 per gigabyte today from $10,000,000 per gigabyte in 1980 [Hennessy and Patterson 2007]. Alas, as mentioned in CW #8 in Section 2, the number of processor cycles to access main memory has grown dramatically as well, from a few processor cycles in 1980 to hundreds today. Moreover, the memory wall is *the* major obstacle to good performance for almost half dwarfs (see Figure 9 in Section 8). Thomas Sterling expressed this concern in his provocative question to panelists at the SC06 conference: "will multicore ultimately be asphyxiated by the memory wall?" [Sterling 2006]

The good news is that if we look inside a DRAM chip, we see many independent, wide memory blocks. [Patterson et al 1997] For example, a 512 Mbit DRAM is composed of hundreds of banks, each thousands of bits wide. Clearly, there is potentially tremendous bandwidth inside a DRAM chip waiting to be tapped, and the memory latency inside a DRAM chip is obviously much better than from separate chips across an interconnect.

In creating a new hardware foundation for parallel computing hardware, we should not limit innovation by assuming main memory must be in separate DRAM chips connected by standard interfaces. New packaging techniques, such as 3D stacking, might allow vastly increased bandwidth and reduced latency and power between processors and DRAM. Although we cannot avoid global communication in the general case with thousands of processors and hundreds of DRAM banks, some important classes of computation have almost entirely local memory accesses and hence can benefit from innovative memory designs.

Another reason to innovate in memory is that increasingly, the cost of hardware is shifting from processing to memory. The old Amdahl rule of thumb was that a balanced computer system needs about 1 MB of main memory capacity per MIPS of processor performance [Hennessy and Patterson 2007].

Whereas DRAM capacity kept pace with Moore's Law by quadrupling capacity every three years between 1980 and 1992, it slowed to doubling every two years between 1996 and 2002. Today we still use the 512 Mbit DRAM that was introduced in 2002.

Manycore designs will unleash a much higher number of MIPS in a single chip. Given the current slow increase in memory capacity, this MIPS explosion suggests a much larger fraction of total system silicon in the future will be dedicated to memory.

4.3 Interconnection networks

At the level of the physical hardware interconnect, multicores have initially employed buses or crossbar switches between the cores and cache banks, but such solutions are not scalable to 1000s of cores. We need on-chip topologies that scale close to linearly with system size to prevent the complexity of the interconnect from dominating cost of manycore systems. Scalable on-chip communication networks will borrow ideas from larger-scale packet-switched networks [Dally and Towles 2001]. Already chip implementations such as the IBM Cell employ multiple ring networks to interconnect the nine processors on the chip and use software-managed memory to communicate between the cores rather than conventional cache-coherency protocols.

Although there has been research into statistical traffic models to help refine the design of Networks-on-Chip [Soteriou et al 2006], we believe the 13 Dwarfs can provide even more insight into communication topology and resource requirements for a broad-array of applications. Based on studies of the communication requirements of existing massively concurrent scientific applications that cover the full range of dwarfs [Vetter and McCracken 2001] [Vetter and Yoo 2002] [Vetter and Meuller 2002] [Kamil et al 2005], we make the following observations about the communication requirements in order to develop a more efficient and custom-tailored solution:

- The collective communication requirements are strongly differentiated from point-to-point requirements. Collective communication, requiring global communication, tended to involve very small messages that are primarily latency bound. As the number of cores increases, the importance of these fine-grained, smaller-than-cache-line-sized, collective synchronization constructs will likely increase. Since latency is likely to improve much more slowly than bandwidth (see CW #6 in Section 2), the separation of concerns suggests adding a separate latency-oriented network dedicated to the collectives. They already appeared in prior MPPs. [Hillis and Tucker 1993] [Scott 1996] As a recent example at large scale, the IBM BlueGene/L has a "Tree" network for collectives in addition to a higher-bandwidth "Torus" interconnect for point-to-point messages. Such an approach may be beneficial for chip interconnect implementations that employ 1000s of cores.
- The sizes of most point-to-point messages are typically large enough that they remain strongly bandwidth-bound, even for on-chip interconnects. Therefore, each point-to-

point message would prefer a dedicated point-to-point pathway through the interconnect to minimize the chance of contention within the network fabric. So while the communication topology does not require a non-blocking crossbar, the on-chip network should have high total bandwidth and support careful mapping of message flows onto the on-chip interconnect topology.

- These studies observed that most point-to-point communications were stable and sparse, and primarily bandwidth bound. With the exception of the 3D FFT (see Figure 2), the point-to-point messaging requirements tend utilize only a fraction of the available communication paths through a fully connected network switch fabric such as a crossbar or fat-tree. For on-chip interconnects, a non-blocking crossbar will likely be grossly over-designed for most application requirements and would otherwise be a waste of silicon given the resource requirements scale as the square of the number of interconnected processor cores. Applications that do not exhibit the communication patterns of the "spectral" dwarf, a lower-degree interconnect topology for on-chip interconnects may prove more space and power efficient.

- Although the communication patterns are observed to be sparse, they are not necessarily isomorphic to a low-degree, fixed-topology interconnect such as a torus, mesh, or hypercube. Therefore, assigning a dedicated path to each point-to-point message transfer is not solved trivially by any given fixed-degree interconnect topology. To this end, one would either want to carefully place jobs so that they match the static topology of the interconnect fabric *or* employ an interconnect fabric that can be reconfigured to conform to the application's communication topology.

The communication patterns observed thus far are closely related to the underlying communication/computation patterns. Given just 13 dwarfs, the interconnect may need to target a relatively limited set of communication patterns. It also suggests that the programming model provide higher-level abstractions for describing those patterns.

For the bandwidth bound communication pathways, we desire an approach to minimizing the surface area occupied by the switch while conforming to the requirements of the application's communication topology. The direct approach to optimizing the interconnect topology to the application requirements is to augment the packet switches using circuit switches to reconfigure the wiring topology between the switches to meet the application communication requirements while maintaining the multiplexing/demultiplexing capability afforded by the packet switches. The inverse approach to this problem relies on software to manage task mapping and task migration to adapt to lower degree static interconnect topologies. The circuit switched approach offers a faster way to reconfigure the interconnect topology, which may prove important for applications that have rapidly changing/adaptive communication requirements. In both cases, runtime performance monitoring systems (see Section 4.6), compile-time instrumentation of codes to infer communication topology requirements, or auto-tuners (see Section 6.1) will play an important role inferring an optimal interconnect topology and communication schedule.

One can use less complex circuit switches to provision dedicated wires that enable the interconnect to adapt to communication pattern of the application at runtime. A hybrid

design that combined packed switches with an optical circuit switch was proposed as a possible solution to the problem at a macro scale. [Kamil et al 2005] [Shalf et al 2005]. However, at a micro-scale, hybrid switch designs that incorporate electrical circuit switches to adapt the communication topology may be able to meet all of the needs of future parallel applications. A hybrid circuit-switched approach can result in much simpler and area-efficient on-chip interconnects for manycore processors by eliminating unused circuit paths and switching capacity through custom runtime reconfiguration of the interconnect topology.

4.4 Communication Primitives

Initially, applications are likely to treat multicore and manycore chips simply as conventional symmetric multiprocessors (SMPs). However, chip-scale multiprocessors (CMPs) offer unique capabilities that are fundamentally different from SMPs, and which present significant new opportunities:

- The inter-core bandwidth on a CMP can be many times greater than is typical for an SMP, to the point where it should cease to be a performance bottleneck.
- Inter-core latencies are far less than are typical for an SMP system (by at least an order of magnitude).
- CMPs could offer new lightweight coherency and synchronization primitives that only operate between cores on the same chip. The semantics of these fences are very different from what we are used to on SMPs, and will operate with much lower latency.

If we simply treat multicore chips as traditional SMPs—or worse yet, by porting MPI applications (see Figure 7 in Section 5)—then we may miss some very interesting opportunities for new architectures and algorithm designs that can exploit these new features.

4.4.1 Coherency

Conventional SMPs use cache-coherence protocols to provide communication between cores, and mutual exclusion locks built on top of the coherency scheme to provide synchronization. It is well known that standard coherence protocols are inefficient for certain data communication patterns (e.g., producer-consumer traffic), but these inefficiencies will be magnified by the increased core count and the vast increase in potential core bandwidth and reduced latency of CMPs. More flexible or even reconfigurable data coherency schemes will be needed to leverage the improved bandwidth and reduced latency. An example might be large, on-chip, caches that can flexibly adapt between private or shared configurations. In addition, real-time embedded applications prefer more direct control over the memory hierarchy, and so could benefit from on-chip storage configured as software-managed scratchpad memory.

4.4.2 Synchronization Using Locks

Inter-processor synchronization is perhaps the area where there is the most potential for dramatic improvement in both performance and programmability. There are two categories of processor synchronization: mutual exclusion and producer-consumer. For mutual exclusion, only one of a number of contending concurrent activities at a time

should be allowed to update some shared mutable state, but typically, the order does not matter. For producer-consumer synchronization, a consumer must wait until the producer has generated a required value. Conventional systems implement both types of synchronization using locks. (*Barriers*, which synchronize many consumers with many producers, are also typically built using locks on conventional SMPs).

These locking schemes are notoriously difficult to program, as the programmer has to remember to associate a lock with every critical data structure and to access only these locks using a deadlock-proof locking scheme. Locking schemes are inherently non-composable and thus cannot form the basis of a general parallel programming model. Worse, these locking schemes are implemented using spin waits, which cause excessive coherence traffic and waste processor power. Although spin waits can be avoided by using interrupts, the hardware inter-processor interrupt and context switch overhead of current operating systems makes this impractical in most cases.

4.4.3 Synchronization Using Transactional Memory

A possible solution for mutual exclusion synchronization is to use transactional memory [Herlihy and Moss 1993]. Multiple processors speculatively update shared memory inside a transaction, and will only commit all updates if the transaction completes successfully without conflicts from other processors. Otherwise, updates are undone and execution is rolled back to the start of the transaction. The transactional model enables non-blocking mutual exclusion synchronization (no stalls on mutex locks or barriers) [Rajwar and Goodman 2002]. Transactional memory simplifies mutual exclusion because programmers do not need to allocate and use explicit lock variables or worry about deadlock.

The Transactional Coherence & Consistency (TCC) scheme [Kozyrakis and Olukotun 2005] proposes to apply transactions globally to replace conventional cache-coherence protocols, and to support producer-consumer synchronization through speculative rollback when consumers arrive before producers.

Transactional memory is a promising but still active research area. Current software-only schemes have high execution time overheads, while hardware-only schemes either lack facilities required for general language support or require very complex hardware. Some form of hybrid hardware-software scheme is likely to emerge, though more practical experience with the use of transactional memory is required before even the functional requirements for such a scheme are well understood.

4.4.4 Synchronization Using Full-Empty Bits in Memory

Reducing the overhead of producer-consumer synchronization would allow finer-grained parallelization, thereby increasing the exploitable parallelism in an application. Earlier proposals have included full-empty bits on memory words, and these techniques could be worth revisiting in the manycore era [Alverson et al 1990] [Alverson et al 1999]. Full-empty bits have proven instrumental for enabling efficient massively parallel graph algorithms (corresponding to the "graph following" dwarf) that are essential for emerging bioinformatics, database, and information processing applications [Bader and Madduri

2006]. In particular, recent work by Jon Berry et al. [Berry et al 2006] has demonstrated that graph processing algorithms executing on a modest 4 processor MTA, which offers hardware support for full-empty bits, can outperform the fastest system on the 2006 Top500 list – the 64k processor BG/L system.

4.4.5 Synchronization Using Message Passing

Shared memory is a very powerful mechanism, supporting flexible and anonymous communication, and single-chip CMP implementations reduce many of the overheads associated with shared memory in multi-chip SMPs. Nevertheless, message passing might have a place between cores in a manycore CMP, as messages combine both data transfer and synchronization in a form that is particularly suited to producer-consumer communications.

4.5 Dependability

CW #3 in Section 2 states that the next generation of microprocessors will face higher soft and hard error rates. Redundancy in space or in time is the way to make dependable systems from undependable components. Since redundancy in space implies higher hardware costs and higher power, we must use redundancy judiciously in manycore designs. The obvious suggestion is to use single error correcting, double error detecting (SEC/DED) encoding for any memory that has the only copy of data, and use parity protection on any memory that just has a copy of data that can be retrieved from elsewhere. Servers that have violated those guidelines have suffered dependability problems [Hennessy and Patterson 2007].

For example, if the L1 data cache uses write through to an L2 cache with write back, then the L1 data cache needs only parity while the L2 cache needs SEC/DED. The cost for SEC/DED is a function of the logarithm of the word width, with 8 bits of SEC/DED for 64 bits of data being a popular size. Parity needs just one bit per word. Hence, the cost in energy and hardware is modest.

Mainframes are the gold standard of dependable hardware design, and among the techniques they use is repeated retransmission to try to overcome soft errors. For example, they would retry a transmission 10 times before giving up and declaring to the operating system that it uncovered an error. While it might be expensive to include such a mechanism on every bus, there are a few places where it might be economical and effective. For example, we expect a common design framework for manycore will be globally asynchronous but locally synchronous per module, with unidirectional links and queues connecting together these larger function blocks. It would be relatively easy to include a parity checking and limited retransmission scheme into such framework.

It may also be possible to fold in dependability enhancements into mechanisms included to enhance performance or to simplify programming. For example, Transactional Memory above (Section 4.4.3) simplifies parallel programming by rolling back all memory events to the beginning of a transaction in the event of mis-speculation about parallelism. Such a rollback scheme could be co-opted into helping with soft errors.

Virtual Machines can also help systems resilient to failures by running different programs in different virtual machines (see Section 6.2). Virtual machines can move applications from a failing processor to a working processor in a manycore chip before the hardware stops. Virtual machines can help cope with software failures as well due to the strong isolation they provide, making an application crash much less likely to affect others.

In addition to these seemingly obvious points, there are open questions for dependability in the manycore era:
- What is the right granularity to check for errors? Whole processors, or even down to registers?
- What is the proper response to an error? Retry, or decline to use the faulty component in the future?
- How serious are errors? Do we need redundant threads to have confidence in the results, or is a modest amount of hardware redundancy sufficient?

4.6 Performance and Energy Counters

Performance counters were originally created to help computer architects evaluate their designs. Since their value was primarily introspective, they had the lowest priority during development. Given this perspective and priority, it is not surprising that measurement of important performance events were often inaccurate or missing: why delay the product for bugs in performance counters that are only useful to the product's architects?

The combination of Moore's Law and the Memory Wall led architects to design increasingly complicated mechanisms to try to deliver performance via instruction level parallelism and caching. Since the goal was to run standard programs faster without change, architects were not aware of the increasing importance of performance counters to compiler writers and programmers in understanding how to make their programs run faster. Hence, the historically cavalier attitude towards performance counters became a liability for delivering performance even on sequential processors.

The switch to parallel programming, where the compiler and the programmer are explicitly responsible for performance, means that performance counters must become first-class citizens. In addition to monitoring traditional sequential processor performance features, new counters must help with the challenge of efficient parallel programming.

Section 7.2 below lists efficiency metrics to evaluate parallel programs, which suggests performance counters to help manycore architectures succeed:
- To minimize remote accesses, identify and count the number of remote accesses and amount of data moved in addition to local accesses and local bytes transferred.
- To balance load, identify and measure idle time vs. active time per processor.
- To reduce synchronization overhead, identify and measure time spent in synchronization per processor.

As power and energy are increasingly important, they need to be measured as well. Circuit designers can create Joule counters for the significant modules from an energy and power perspective. On a desktop computer, the leading energy consumers are

processors, main memory, caches, the memory controller, the network controller, and the network interface card.

Given Joules and time, we can calculate Watts. Unfortunately, measuring time is getting more complicated. Processors traditionally counted processor clock cycles, since the clock rate was fixed. To save energy and power, some processors have adjustable threshold voltages and clock frequencies. Thus, to measure time accurately, we now need a "picosecond counter" in addition to a clock cycle counter.

While performance and energy counters are vital to the success of parallel processing, the good news is that they are relatively easy to include. Our main point is to raise their priority: do not include features that significantly affect performance or energy if programmers cannot accurately measure their impact.

5.0 Programming Models

Figure 1 shows that a *programming model* is a bridge between a system developer's natural model of an application and an implementation of that application on available hardware. A programming model must allow the programmer to balance the competing goals of *productivity* and *implementation efficiency*. Implementation efficiency is always an important goal when parallelizing an application, as programs with limited performance needs can always be run sequentially. We believe that the keys to achieving this balance are two conflicting goals:

- *Opacity* abstracts the underlying architecture. Abstraction obviates the need for the programmer to learn the architecture's intricate details and increases programmer productivity.
- *Visibility* makes the key elements of the underlying hardware visible to the programmer. It allows the programmer to realize the performance constraints of an application by exploring design parameters such as thread boundaries, data locality, and the implementation of elements of the application.

While maximizing the raw performance/power of future multicores is important, the real key to their success is the programmer's ability to harvest that performance.

Figure 7 shows the current lack of agreement on the opacity/visibility tradeoff. It lists 10 examples of programming models for five critical parallel tasks that go from requiring the programmer to make explicit decisions for all tasks for efficiency to models that make all the decisions for the programmer for productivity. In between these extremes, the programmer does some tasks and leaves the rest to the system.

The struggle is delivering performance while raising the level of abstraction. Going too low may achieve performance, but at the cost of exacerbating the software productivity problem, which is already a major hurdle for the information technology industry. Going too high can reduce productivity as well, for the programmer is then forced to waste time trying to overcome the abstraction to achieve performance.

In the following sections, we present some recommendations for designers of programming systems for parallel machines. Instead of the conventional focus just on

hardware, applications, or mathematical formalisms, create and evaluate programming models inspired more by results from psychology (Section 5.1). A few seemingly obvious but often neglected characteristics for a successful parallel model that raise the level of abstraction without hurting efficiency are making programs independent of the number of processors (Section 5.2), supporting a rich set of data types (Section 5.3), and supporting styles of parallelism that have been proven successful in the past (Sections 5.4).

Model	*Domain*	*Task Identification*	*Task Mapping*	*Data Distribution*	*Commun- ication Mapping*	*Synchro- nization*
Real-Time Workshop [MathWorks 2004]	DSP	Explicit	Explicit	Explicit	Explicit	Explicit
TejaNP [Teja 2003]	Network	Explicit	Explicit	Explicit	Explicit	Explicit
YAPI [Brunel et al 2000]	DSP	Explicit	Explicit	Explicit	Explicit	Implicit
MPI [Snir et al 1998]	HPC	Explicit	Explicit	Explicit	Implicit	Implicit
Pthreads [Pthreads 2004]	General	Explicit	Explicit	Implicit	Implicit	Explicit
StreamIt [Gordon et al 2002]	DSP	Explicit	Implicit	Explicit	Implicit	Implicit
MapReduce [Dean and Ghemawat 2004]	Large Data sets	Explicit	Implicit	Implicit	Implicit	Explicit
Click to network processors [Plishker et al 2004]	Network	Implicit	Implicit	Implicit	Implicit	Explicit
OpenMP [OpenMP 2006]	HPC	Implicit (directives, some explicit)	Implicit	Implicit	Implicit	Implicit (directives, some explicit)
HPF [Koelbel et al 1993]	HPC	Implicit	Implicit	Implicit (directives)	Implicit	Implicit

Figure 7. Comparison of 10 current parallel programming models for 5 critical tasks, sorted from most explicit to most implicit. High-performance computing applications [Pancake and Bergmark 1990] and embedded applications [Shah et al 2004a] suggest these tasks must be addressed one way or the other by a programming model: 1) Dividing the application into parallel tasks; 2) Mapping computational tasks to processing elements; 3) Distribution of data to memory elements; 4) mapping of communication to the inter-connection network; and 5) Inter-task synchronization.

5.1 Programming model efforts inspired by psychological research

Developing programming models that productively enable development of highly efficient implementations of parallel applications is the biggest challenge facing the deployment of future manycore systems. Hence, research in programming models is a high priority. In our view, programming model development in the past has been hardware-centric, application-centric, or formalism-centric.Hardware-centric programming models are typically developed by the hardware-manufacturers themselves in an attempt to maximize the efficiency of the hardware they produce. For example, the C-variant known as IXP-C [Intel 2004], together with library elements known as microblocks, was developed for the Intel IXP family of network processors [Adiletta et al 2002]. Such environments typically do not offer the desired productivity improvements or support for the broader parallel programming process–architecting, debugging, and so on – involved in the development of a parallel application.

Application-centric programming models, such as Matlab [MathWorks 2006], are typically focused on easing the development of related application domains. These models also don't support the broader parallel programming process nor do they offer support for fine-tuning implementations to realize efficiency constraints.

Formalism-centric programming models, such as Actors [Hewitt et al 1973], try to reduce the chance of programmer making mistakes by having clean semantics and offer the chance to remove bugs by verifying correctness of portions of the code.

All three goals are obviously important: efficiency, productivity, and correctness. It is striking, however, that research from psychology has had almost no impact, despite the obvious fact that the success of these models will be strongly affected by the human beings who use them. Testing methods derived from the psychology research community have been used to great effect for HCI, but are sorely lacking in language design and software engineering. For example, there is a rich theory investigating the causes of human errors, which is well known in the human-computer interface community, but apparently it has not penetrated the programming model and language design community. [Kantowitz and Sorkin 1983] [Reason 1990] There have been some initial attempts to identify the systematic barriers to collaboration between the Software Engineering (SE) and HCI community and propose necessary changes to the CS curriculum to bring these fields in line, but there has been no substantial progress to date on these proposals. [Seffah 2003] [Pyla et al 2004] We believe that integrating research on human psychology and problem solving into the broad problem of designing, programming, debugging, and maintaining complex parallel systems will be critical to developing broadly successful parallel programming models and environments.

Transactional memory is an example of a programming model that helps prevent human errors. Programmers have a difficult time determining when to synchronize in parallel code, and often get it wrong. An advantage of transactional memory is that the system will ensure correctness, even when programmers make incorrect assumptions about the safety of parallelizing a piece of code. The payoff of transactional memory is not

primarily efficiency, formalism, or even productivity; it is that programs can work properly even when programmers err or overly aggressive auto-parallelizing compilers make mistakes.

Not only do we ignore insights about human cognition in the design of our programming models, we do not follow their experimental method to resolve controversies about how people use them. That method is human-subject experiments, which is so widespread that most campuses have committees that must be consulted before you can perform such experiments. Subjecting our assumptions about the process or programming to formal testing often yields unexpected results that challenge our intuition. [Mattson 1999]

A small example is a study comparing programming using shared memory vs. message passing. These alternatives have been the subject of hot debates for decades, and there is no consensus on which is better and when. A recent paper compared efficiency and productivity of small programs written both ways for small parallel processors by novice programmers. [Hochstein et al 2005] While this is not the final word on the debate, it does indicate a path to try to resolve important programming issues. Fortunately, there are a growing number of examples of groups that have embraced user studies to evaluate the productivity of computer languages. [Kuo et al 2005] [Solar-Lezama et al 2005] [Ebcioglu et al 2006]

We believe that future successful programming models must be more human-centric. They will be tailored to the human process of productively architecting and efficiently implementing, debugging, and maintaining complex parallel applications on equally complex manycore hardware. Furthermore, we believe we must use human subject experiments to resolve open issues for us to make progress in discovering how to make it genuinely easy to program manycore systems efficiently.

5.2 Models must be independent of the number of processors

MPI, the current dominant programming model for parallel scientific programming, forces coders to be aware of the exact mapping of computational tasks to processors. This style has been recognized for years to increase the cognitive load on programmers, and has persisted primarily because it is expressive and delivers the best performance. [Snir et al 1998] [Gursoy and Kale 2004]

Because we anticipate a massive increase in exploitable concurrency, we believe that this model will break down in the near future, as programmers have to explicitly deal with decomposing data, mapping tasks, and performing synchronization over thousands of processing elements.

Recent efforts in programming languages have focused on this problem and their offerings have provided models where the number of processors is not exposed [Deitz 2005] [Allen et al 2006] [Callahan et al 2004] [Charles et al 2005]. While attractive, these models have the opposite problem—delivering performance. In many cases, hints can be provided to co-locate data and computation in particular memory domains. In addition,

because the program is not over-specified, the system has quite a bit of freedom in mapping and scheduling that in theory can be used to optimize performance. Delivering on this promise is, however, still an open research question.

5.3 Models must support a rich set of data sizes and types

Although the algorithms were often the same in embedded and server benchmarks in Section 3, the data types were not. SPEC relies on single- and double-precision floating point and large integer data, while EEMBC uses integer and fixed-point data that varies from 1 to 32 bits. [EEMBC 2006] [SPEC 2006] Note that most programming languages only support the subset of data types found originally in the IBM 360 announced 40 years ago: 8-bit characters, 16- and 32-bit integers, and 32- and 64-bit floating-point numbers.

This leads to the relatively obvious observation. If the parallel research agenda inspires new languages and compilers, they should allow programmers to specify at least the following sizes (and types):
- 1 bit (Boolean)
- 8 bits (Integer, ASCII)
- 16 bits (Integer, DSP fixed point, Unicode)
- 32 bits (Integer, Single-precision floating point, Unicode)
- 64 bits (Integer, Double-precision floating point
- 128 bits (Integer, Quad-Precision floating point
- Large integer (>128 bits) (Crypto)

Mixed precision floating-point arithmetic—separate precisions for input, internal computations, and output—has already begun to appear for BLAS routines [Demmel et al 2002]. A similar and perhaps more flexible structure will be required so that all methods can exploit it. While support for all of these types can mainly be provided entirely in software, we do not rule out additional hardware to assist efficient implementations of very wide data types.

In addition to the more "primitive" data types described above, programming environments should also provide for distributed data types. These are naturally tightly coupled to the styles of parallelism that are expressed, and so influence the entire design. The languages proposed in the DARPA High Productivity Language Systems program are currently attempting to address this issue, with a major concern being support for user-specified distributions.

5.4 Models must support of proven styles of parallelism

Programming languages, compilers, and architectures have often placed their bets on one style of parallel programming, usually forcing programmers to express all parallelism in that style. Now that we have a few decades of such experiments, we think that the conclusion is clear: some styles of parallelism have proven successful for some applications, and no style has proven best for all.

Rather than placing all the eggs in one basket, we think programming models and architectures should support a variety of styles so that programmers can use the superior choice when the opportunity occurs. We believe that list includes at least the following:

1. *Independent task parallelism* is an easy-to-use, orthogonal style of parallelism that should be supported in any new architecture. As a counterexample, older vector computers could not take advantage of task-level parallelism despite having many parallel functional units. Indeed, this was one of the key arguments used against vector computers in the switch to massively parallel processors.

2. *Word-level parallelism* is a clean, natural match to some dwarfs, such as sparse and dense linear algebra and unstructured grids. Examples of successful support include array operations in programming languages, vectorizing compilers, and vector architectures. Vector compilers would give hints at compile time about why a loop did not vectorize, and non-computer scientists could then vectorize the code because they understood the model of parallelism. It has been many years since that could be said about a new parallel language, compiler, and architecture.

3. *Bit-level parallelism* may be exploited within a processor more efficiently in power, area, and time than between processors. For example, the Secure Hash Algorithm (SHA) for cryptography has significant parallelism, but in a form that requires very low latency communication between operations on small fields.

In addition to the styles of parallelism, we also have the issue of the memory model. Because parallel systems usually contain memory distributed throughout the machine, the question arises of the programmer's view of this memory. Systems providing the illusion of a uniform shared address space have been very popular with programmers. However, scaling these to large systems remains a challenge. *Memory consistency* issues (relating to the visibility and ordering of local and remote memory operations) also arise when multiple processors can update the same locations, each likely having a cache. Explicitly partitioned systems (such as MPI) sidestep many of these issues, but programmers must deal with the low-level details of performing remote updates themselves.

6.0 Systems Software

In addition to programming models, compilers and operating systems help span the gap between applications and hardware towers of Figure 1. In our view, both of these vital programs have grown so large over the decades that it is hard to do the innovation that may need as we switch to parallelism. Hence, instead of completely re-engineering compilers for parallelism, we recommend relying more on autotuners that search to yield efficient parallel code (Section 6.1). Instead of relying on the conventional large, monolithic operating systems, we recommend relying more on virtual machines and system libraries to include only those functions needed by the application (Section 6.2)

6.1 Autotuners vs. Traditional Compilers

Regardless of the programming model, performance of future parallel applications will crucially depend on the quality of the generated code, traditionally the responsibility of the compiler. For example, it may need to select a suitable implementation of synchronization constructs or optimize communication statements. Additionally, the compiler must generate good sequential code; a task complicated by complex

microarchitectures and memory hierarchies. The compiler selects which optimizations to perform, chooses parameters for these optimizations, and selects from among alternative implementations of a library kernel. The resulting space of optimization alternatives is large. Such compilers will start from parallelism indicated in the program implicitly or explicitly, and attempt to increase its amount or modify its granularity—a problem that can be simplified, but not sidestepped, by a good programming model.

6.1.1 The Difficulty of Enhancing Modern Compilers

Unfortunately, it is difficult to add new optimizations to compilers, presumably needed in the transition from instruction-level parallelism to task- and data-level parallelism. As a modern compiler contains millions of lines of code and new optimizations often require fundamental changes to its internal data structures, the large engineering investment is difficult to justify, as compatibility with language standards and functional correctness of generated code are usually much higher priorities than output code quality. Moreover, exotic automatic optimization passes are difficult to verify against all possible inputs versus the few test cases required to publish a paper in a research conference. Consequently, users have become accustomed to turning off sophisticated optimizations, as they are known to trigger more than their fair share of compiler bugs.
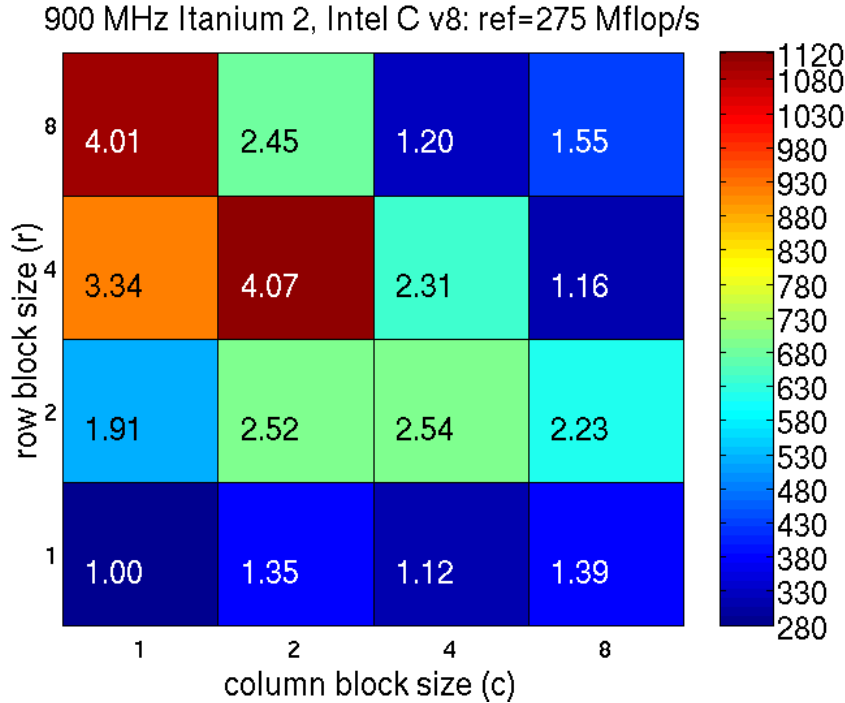
Due to the limitations of existing compilers, peak performance may still require handcrafting the program in languages like C, FORTRAN, or even assembly code. Indeed, most scalable parallel codes have all data layout, data movement, and processor synchronization manually orchestrated by the programmer. Such low-level coding is labor intensive, and usually not portable to different hardware platforms or even to later implementations of the same instruction set architecture.

6.1.2 The Promise of Search-Based Autotuners

Our vision is that relying on search embedded in various forms of software synthesis can solve these problems. Synthesizing efficient programs through search has been used in several areas of code generation, and has had several notable successes. [Massalin 1987] [Granlund et al 2006] [Warren 2006].

In recent years, "Autotuners" [Bilmes et al 1997] [Frigo and Johnson 1998] [Frigo and Johnson 2005] [Granlund et al 2006] [Im et al 2005] [Whaley and Dongarra 1998] gained popularity as an effective approach to producing high-quality portable scientific code. Autotuners optimize a set of library kernels by generating many variants of a given kernel and benchmarking each variant by running on the target platform. The search process effectively tries many or all optimization switches and hence may take hours to complete on the target platform. Search needs to be performed only once, however, when the library is installed. The resulting code is often several times faster than naive implementations, and a single autotuner can be used to generate high-quality code for a wide variety of machines. In many cases, the autotuned code is faster than vendor libraries that were specifically hand-tuned for the target machine! This surprising result is partly explained by the way the autotuner tirelessly tries many unusual variants of a particular routine, often finding non-intuitive loop unrolling or register blocking factors that lead to better performance.

For example, Figure 8 shows how performance varies by a factor of four with blocking options on Itanium 2. The lesson from autotuning is that by searching many possible combinations of optimization parameters, we can sidestep the problem of creating an effective heuristic for optimization policy.

Figure 8. Sparse matrix performance on Itanium 2 for a finite element problem using block compressed sparse row (BCSR) format [Im et al 2005]. Performance (color-coded, relative to the 1x1 baseline) is shown for all block sizes that divide 8x8—16 implementations in all. These implementations fully unroll the innermost loop and use scalar replacement for the source and destination vectors. You might reasonably expect performance to increase relatively smoothly as r and c increase, but this is clearly not the case. Platform: 900 MHz Itanium-2, 3.6 Gflop/s peak speed. Intel v8.0 compiler.

The popularity of autotuners could lead to changes in benchmarks. Conventional benchmarks such as SPEC are distributed as source code that must be compiled and run unaltered. This code often contains manual optimizations favoring a particular target computer, such as a particular cache blocking. Autotuned code, however, would allow a benchmark to find the best approach for each target automatically.

6.1.3 Extending Autotuners to Parallelism

We believe that autotuning can help with the compilation of parallel code as well. Parallel architectures, however, introduce many new optimization parameters, and so far, no successful autotuners for parallel codes exist. For any given problem, there may be several parallel algorithms, each with alternative parallel data layouts. The optimal choice may depend not only on the processor architecture but also on the parallelism of the computer, as well as the network bandwidth and latency. Consequently, in a parallel setting, the search space can be much larger than that for a sequential kernel.

To reduce the search space, it may be possible to decouple the search for good data layout and communication patterns from the search for a good compute kernel, especially with the judicial use of performance models. The network and memory performance may be characterized relatively quickly using test patterns, and then plugged into performance models for the network to derive suitable code loops for the search over compute kernels [Vadhiyar et al 2000].

6.2 Deconstructing operating system support

Although programming models live above the operating system layer, the efficiency of that layer can strongly affect the efficiency of the programs that rely upon it. Just as processors have crossed an inflection point of the benefits of growing larger, we believe operating systems have as well. Going forward, we believe that operating systems must be deconstructed, with virtual machines enabling end applications to select only the portion of the OS capabilities that are needed rather be forced to accept a gargantuan soft stack. Just as hardware is moving away from a single monolithic processor, operating systems may be moving away from a single monolithic program. We lay out those arguments in this section.

6.2.1 Increasing Need of Protection in Embedded Computing

One place where there is the greatest tension between the embedded and server communities in the past is operating systems. Embedded systems have historically had very minimal application-specific run-time systems, with tight control over real-time scheduling, but with little support for protection and virtualization. This reflects the desire to reduce processor cost, memory footprint, and power consumption, and the assumption that software will be custom written for a particular embedded system by the manufacturer. Traditional server operating systems have millions of lines of code, and provide a very rich set of features. Protection and virtualization are essential to support large software systems built using a range of third-party code written to industry-standard APIs, and communicating over the unsecured global Internet.

We believe these two worlds are colliding and merging, as embedded systems increase in functionality. For example, cell phones and game machines now support multi-gigabyte file systems and complex Web browsers. In particular, cell phone manufacturers who have previously resisted the installation of third-party software due to reliability concerns, now realize that a standard API must be provided to allow user extensibility, and this will require much more sophisticated and stable operating systems and the hardware support these require.

Since embedded computers are increasingly connected to networks, we think they will be increasingly vulnerable to viruses and other attacks. Indeed, the first personal computer operating systems dropped protection since developers thought a PC had only a single user, which worked OK until we connected PCs to the Internet. Imagine how much better our lives would be if security had been a PC OS priority before they joined the Internet.

6.2.2 Virtual Machines to the Rescue

Traditional OSes are too large and brittle to support radical innovation but contain millions of lines of valuable legacy code essential to application functionality. The resurgence of interest in *virtual machines* (VMs) is evidence that operating systems have reached their own technology inflexion point. VM technology allows a complete operating system with running applications to be treated as a software component, manipulated by a *virtual machine monitor* (VMM) or hypervisor. The VMM inserts a thin software layer between a guest OS and the hardware to give the guest OS the illusion that it is running on its own copy of the real hardware. This approach allows a very small, very low overhead VMM to provide innovative protection and resource sharing without having to run or modify multimillion-line OSes.

Virtual machines appear to be the future of server operating systems. For example, AMD, Intel, and Sun have all modified their instruction set architectures to support virtual machines. VMs have become popular in server computing for a few reasons: [Hennessy and Patterson 2007]
- To provide a greater degree of protection against viruses and attacks;
- To cope with software failures by isolating a program inside a single VM so as not to damage other programs; and
- To cope with hardware failures by migrating a virtual machine from one computer to another without stopping the programs

VMMs provide an elegant solution to the failure of conventional OSes to provide such features. VMMs are also a great match to manycore systems, in that space sharing will be increasingly important when running multiple applications on 1000s of processors.

What is the cost of a VMM? The overhead of running an OS on a VMM is generally a function of the instruction set architecture. We believe manycore architectures for embedded and server should support virtualization, as the hardware costs are trivial. By designing an instruction set architecture to be virtualizable, the software overhead can be very low. Indeed, an important architectural goal would be to provide the support that helps prevent the VMM from growing over time.

6.2.3 Deconstructing Operating Systems

Rosenblum argues that the future of server operating system could essentially be libraries where only the functions needed are linked into the application, on top of a thin VMM layer providing protection and sharing of hardware resources. [Rosenblum 2006] This vision is similar to embedded OSes today. For example, VxWorks lets the user choose which features of the OS will be included in this embedded application. [Wind River 2006] Hence, we see operating systems having more in common for embedded and server computing.

While this vision is compelling, it is not binding. An application can run either a very thin or a very thick OS on top of the VMM, or even multiple OSes simultaneously to accommodate different task needs. For example, a real-time code and a best effort code running on different cores, or a minimal data-plane OS on multiple high-density cores and a complex control-plane OS on a large general-purpose core.

7.0 Metrics for Success

Having covered the six questions from the full bridge in Figure 1, we need to decide how to best invent and evaluate answers to those questions. In the following, we focus on maximizing two metrics–programmer productivity and final implementation efficiency–and to provide a vehicle to help researchers innovate more quickly.

7.1 Maximizing programmer productivity

Having thousands of processing elements on a single chip presents a major programming challenge to application designers. The adoption of the current generation of on-chip multiprocessors has been slow due to the difficulty of getting applications correctly and productively implemented on these devices. For example, the trade press speaking of current on-chip multiprocessors targeted for network applications says [Weinberg 2004]:

> "... network processors with powerful and complex packet-engine sets have proven to be notoriously difficult to program."

Earlier on-chip multiprocessors such as the TI TMS320C80 failed altogether because application designers could not tap their performance productively. Thus, the ability to productively program these high-performance multiprocessors of the future is as at least as important as providing high-performance silicon implementations of these architectures.

Another area that deserves consideration is the addition of hardware structures that assist language productivity features. For example, supporting transactional memory entirely in software may be too slow to be useful, but can be made efficient with hardware support. Other examples of this include support for garbage collection, fine-grained synchronization (the Cray MTA), one-sided messaging, trace collection for debugging [Xu et al 2003], and performance and energy counters to aid program optimization (see Section 4.5).

Productivity is a multifaceted term that is difficult to quantify. However, case studies such as [Shah et al 2004b] and the work in the ongoing DARPA HPCS program [HPCS 2006] build our confidence that productivity is amenable to quantitative comparison. In addition, work in the psychology of programming can also inform our evaluation efforts.

7.2. Maximizing application performance

One implication of Figure 2 is that for 15 years application performance steadily increased simply by running applications on new generations of processors with minimal additional programmer effort. As processor performance growth has slowed, new ideas will be required to realize further application performance gains. Radical ideas are required to make manycore architectures a secure and robust base for productive software development since the existing literature only shows successes in narrow application domains such as Cisco's 188-processor Metro chip for networking applications [Eatherton 2005].

Moreover, since the power wall has forced us to concede the battle for maximum performance of individual processing elements, we must aim at winning the war for application efficiency through optimizing total system performance. This will require

extensive design space exploration. The general literature on design-space exploration is extensively reviewed in [Gries 2004] and the state-of-the art in commercial software support for embedded processor design-space exploration using CoWare or Tensilica toolsets is presented in [Gries and Keutzer 2005]. However, evaluating full applications requires more than astute processing element definition; the full system-architecture design space including memory and interconnect must be explored. Although these design space explorations focus on embedded processors, we believe that the processors of manycore systems will look more like embedded processors than current desktop processors (see Section 4.1.2.).

New efficiency metrics will make up the evaluation of the new parallel architecture. As in the sequential world, there are many "observables" from program execution that provide hints (such as cache misses) to the overall efficiency of a running program. In addition to serial performance issues, the evaluation of parallel systems architectures will focus on:

- *Minimizing remote accesses*. In the case where data is accessed by computational tasks that are spread over different processing elements, we need to optimize its placement so that communication is minimized.
- *Load balance*. The mapping of computational tasks to processing elements must be performed in such a way that the elements are idle (waiting for data or synchronization) as little as possible.
- *Granularity of data movement and synchronization*. Most modern networks perform best for large data transfers. In addition, the latency of synchronization is high and so it is advantageous to synchronize as little as possible.

Software design environments for embedded systems such as those described in [Rowen and Leibson 2005] lend greater support to making these types of system-level decisions. To make help programmers progress towards these goals, we recommend hardware counters that can measure these performance issues (see Section 4.6).

The conventional path for exploring new architectures for the last decade has been simulation. We are skeptical that software simulation alone will provide sufficient throughput for thorough evaluation of manycore systems architectures. Nor will per-project hardware prototypes that require long development cycles be sufficient. The development of these *ad hoc* prototypes will be far too slow to influence the decisions that industry will need to make regarding future manycore system architectures. We need a platform where feedback from software experiments on novel manycore architectures running real applications with representative workloads will lead to new system architectures within days, not years.

7.3 RAMP: Research Accelerator for Multiple Processors

The Research Accelerator for Multiple Processor (RAMP) project is an open-source effort of ten faculty at six institutions to create a computing platform that will enable rapid innovation in parallel software and architecture [Arvind et al 2005] [Wawrzynek et al 2006]. RAMP is inspired by:

1. The difficulty for researchers to build modern chips, as described in CW #5 in Section 2.

2. The rapid advance in field-programmable gate arrays (FPGAs), which are doubling in capacity every 18 months. FPGAs now have the capacity for millions of gates and millions of bits of memory, and they can be reconfigured as easily as modifying software.
3. Flexibility, large scale, and low cost trumps absolute performance for researchers, as long as performance is fast enough to do their experiments in a timely fashion. This perspective suggests the use of FPGAs for system emulation.
4. Smaller is better (see Section 4.1) means many of these hardware modules can fit *inside* an FPGA today, avoiding the much tougher mapping problems of the past when a single module had to span many FPGAs.
5. The availability of open-source modules, from Opencores.org, Open SPARC, and Power.org, which can be inserted into FPGAs with little effort [Opencores 2006] [OpenSPARC 2006] [Power.org 2006].

While the idea for RAMP is just 18 months old, the group has made rapid progress. It has financial support from NSF and several companies and it has working hardware based on an older generation of FPGA chips. Although RAMP will run, say, 20 times more slowly than real hardware, it will emulate many different speeds of components accurately to report correct performance as measured in the emulated clock rate.

The group plans to develop three versions of RAMP to demonstrate what can be done:
- *Cluster RAMP ("RAMP Blue")*: Led by the Berkeley contingent, this version will a large-scale example using MPI for high performance applications like the NAS parallel benchmarks [Van der Wijngaart 2002] or TCP/IP for Internet applications like search. An 8-board version will run the NAS benchmarks on 256 processors.
- *Transactional Memory RAMP ("RAMP Red")*: Led by the Stanford contingent, this version will implement cache coherency using the TCC version of transactional memory [Hammond et al 2004]. A single board system runs 100 times faster than the Transactional Memory simulator.
- *Cache-Coherent RAMP ("RAMP White")*: Led by the CMU and Texas contingents, this version will implement a ring-based coherency or snoop based coherency.

All will share the same "gateware"—processors, memory controllers, switches, and so on—as well as CAD tools, including co-simulation. [Chung et al 2006]

The goal is to make the "gateware" and software freely available on a web site, to redesign the boards to use the recently announced Virtex 5 FPGAs, and finally to find a manufacturer to sell them at low margin. The cost is estimated to be about $100 per processor and the power about 1 watt per processor, yielding a 1000 processor system that costs about $100,000, that consumes about one kilowatt, and that takes about one quarter of a standard rack of space.

The interactions between massively parallel programming models, real-time constraints, protection, and virtualization provide a rich ground for architecture and software systems research. The hope is that the advantages of large-scale multiprocessing, standard instruction sets and OSes, low cost, low power, and ease-of-change will make RAMP a

standard platform for parallel research for many types of researchers. If it creates a "watering hole effect" in bringing many disciplines together, it could lead to innovation that will more rapidly develop successful answers to the seven questions of Figure 1.

8.0 Conclusion

CWs # 1, 7, 8, and 9 in Section 2 say the triple whammy of the Power, Memory, and Instruction Level Parallelism Walls has forced microprocessor manufacturers to bet their futures on parallel microprocessors. This is no sure thing, as parallel software has an uneven track record.

From a research perspective, however, this is an exciting opportunity. Virtually any change can be justified—new programming languages, new instruction set architectures, new interconnection protocols, and so on—if it can deliver on the goal of making it easy to write programs that execute efficiently on manycore computing systems.

This opportunity inspired a group of us at Berkeley from many backgrounds to spend nearly two years discussing the issues, leading to the seven questions of Figure 1 and the following unconventional perspectives:
- *Regarding multicore versus manycore*: We believe that manycore is the future of computing. Furthermore, it is unwise to presume that multicore architectures and programming models suitable for 2 to 32 processors can incrementally evolve to serve manycore systems of 1000s of processors.
- *Regarding the application tower*: We believe a promising approach is to use 13 Dwarfs as stand-ins for future parallel applications since applications are rapidly changing and because we need to investigate parallel programming models as well as architectures.
- *Regarding the hardware tower*: We advise using simple processors, to innovate in memory as well as in processor design, to consider separate latency-oriented and bandwidth-oriented networks. Since the point-to-point communication patterns are very sparse, a hybrid interconnect design that uses circuit switches to tailor the interconnect topology to application requirements could be more area and power efficient than a full-crossbar and more computationally efficient than a static mesh topology. Traditional cache coherence is unlikely to be sufficient to coordinate the activities of 1000s of cores, so we recommend a richer hardware support for fine-grained synchronization and communication constructs. Finally, do not include features that significantly affect performance or energy if you do not provide counters that let programmers accurately measure their impact.
- *Regarding the programming models that bridge the two towers*: To improve productivity, programming models must be more human-centric and engage the full range of issues associated with developing a parallel application on manycore hardware. To maximize application efficiency as well as programmer productivity, programming models should be independent of the number of processors, they should allow programmers to use a richer set of data types and sizes, and they should support successful and well-known parallel models of parallelism: independent task, word-level, and bit-level parallelism.

- We also think that autotuners should take on a larger, or at least complementary, role to compilers in translating parallel programs. Further, we argue that traditional operating systems will be deconstructed and operating system functionality will be orchestrated using virtual machines.
- To provide an effective parallel computing roadmap quickly so that industry can safely place its bets, we encourage researchers to use autotuners and RAMP to explore this space rapidly and to measure success by how easy it is to program the 13 Dwarfs to run efficiently on manycore systems.
- While embedded and server computing have historically evolved along separate paths, in our view the manycore challenge brings them much closer together. By leveraging the good ideas from each path, we believe we will find better answers to the seven questions in Figure 1.

As a test case to see the usefulness of these observations, one of the authors was invited to a workshop that posed the question of what could you do if you had infinite memory bandwidth? We approached the problem using the dwarfs, asking which were computationally limited and which were limited by memory. Figure 9 below gives the results of our quick study, which was that memory latency was a bigger problem than memory bandwidth, and some dwarfs were not limited by memory bandwidth or latency. Whether our answer was correct or not, it was exciting to have a principled framework to rely upon to try to answer such open and difficult questions.

This report is intended to be the start of a conversation about these perspectives. There is an open, exciting, and urgent research agenda to flush out the concepts represented by the two towers and span of Figure 1. We invite you to participate in this important discussion by visiting view.eecs.berkeley.edu.

Dwarf	Performance Limit: Memory Bandwidth, Memory Latency, or Computation?
1. Dense Matrix	Computationally limited
2. Sparse Matrix	Currently 50% computation, 50% memory BW
3. Spectral (FFT)	Memory latency limited
4. N-Body	Computationally limited
5. Structured Grid	Currently more memory bandwidth limited
6. Unstructured Grid	Memory latency limited
7. MapReduce	Problem dependent
8. Combinational Logic	CRC problems BW; crypto problems computationally limited
9. Graph traversal	Memory latency limited
10. Dynamic Programming	Memory latency limited
11. Backtrack and Branch+Bound	?
12. Construct Graphical Models	?
13. Finite State Machine	Nothing helps!

Figure 9. Limits to performance of dwarfs, inspired by an suggestion by IBM that a packaging technology could offer virtually infinite memory bandwidth. While the memory wall limited performance for almost half the dwarfs, memory latency is a bigger problem than memory bandwidth

Acknowledgments

References

[ABAQUS 2006] ABAQUS finite element analysis home page. http://www.hks.com

[Adiletta et al 2002] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson, "The Next Generation of the Intel IXP Network Processors," *Intel Technology Journal*, vol. 6, no. 3, pp. 6–18, Aug. 15, 2002.

[Allen et al 2006] E. Allen, V. Luchango, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt, *The Fortress Language Specification*, 2006. Available at http://research.sun.com/projects/plrg/

[Altschul et al 1990] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, "Basic local alignment search tool," *Journal Of Molecular Biol*ogy, vol. 215, no. 3, 1990, pp. 403–410.

[Alverson et al 1990] R. Alverson, D. Cllahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," in *Proceedings of the 1990 ACM International Conference on Supercomputing (SC'90),* pp. 1–6, Jun. 1990.

[Alverson et al 1999] G.A. Alverson, C.D. Callahan, II, S.H. Kahan, B.D. Koblenz, A. Porterfield, B.J. Smith, "Synchronization Techniques in a Multithreaded Environment," US patent 6862635.

[Arnold 2005] J. Arnold, "S5: the architecture and development flow of a software configurable processor," in *Proceedings of the IEEE International Conference on Field-Programmable Technology*, Dec. 2005, pp. 121–128.

[Arvind et al 2005] Arvind, K. Asanovic, D. Chiou, J.C. Hoe, C. Kozyrakis, S. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek, "RAMP: Research Accelerator for Multiple Processors - A Community Vision for a Shared Experimental Parallel HW/SW Platform," U.C. Berkeley technical report, UCB/CSD-05-1412, 2005.

[Aspuru-Guzik et al 2005] A. Aspuru-Guzik, R. Salomon-Ferrer, B. Austin, R. Perusquia-Flores, M.A. Griffin, R.A. Oliva, D. Skinner, D. Domin, and W.A. Lester, Jr., "Zori 1.0: A Parallel Quantum Monte Carlo Electronic Package," *Journal of Computational Chemistry*, vol. 26, no. 8, Jun. 2005, pp. 856–862.

[Bader and Madduri 2006] D.A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in *Proceedings of the 35th International Conference on Parallel Processing (ICPP)*, Aug. 2006, pp. 523–530.

[Barnes and Hut 1986] J. Barnes and P. Hut, "A Hierarchical O(n log n) force calculation algorithm," *Nature*, vol. 324, 1986.

[Bell and Newell 1970] G. Bell and A. Newell, "The PMS and ISP descriptive systems for computer structures," in *Proceedings of the Spring Joint Computer Conference*, AFIPS Press, 1970, pp. 351–374.

[Bernholdt et al 2002] D.E. Bernholdt, W.R. Elsasif, J.A. Kohl, and T.G.W. Epperly, "A Component Architecture for High-Performance Computing," in *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*, Jun. 2002.

[Berry et al 2006] J.W. Berry, B.A. Hendrickson, S. Kahan, P. Konecny, "Graph Software Development and Performance on the MTA-2 and Eldorado," presented at the 48th Cray Users Group Meeting, Switzerland, May 2006.

[Bilmes et al 1997] J. Bilmes, K. Asanovic, C.W. Chin, J. Demmel, "Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology," in *Proceedings of the 11th International Conference on Supercomputing*, Vienna, Austria, Jul. 1997, pp. 340–347.

[Blackford et al 1996] L.S. Blackford, J. Choi, A. Cleary, A. Petitet, R.C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker, "ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, Nov. 1996.

[Blackford et al 2002] L.S. Blackford, J. Demmel, J. Dongarra, I. Du, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R.C. Whaley, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2 , Jun. 2002, pp. 135–151.

[Borkar 1999] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, Jul.–Aug. 1999, pp. 23–29.

[Borkar 2005] S. Borkar, "Designing Reliable Systems from Unrealiable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, Nov.–Dec. 2005, pp. 10–16.

[Brunel et al 2000] J.-Y. Brunel, K.A. Vissers, P. Lieverse, P. van der Wolf, W.M. Kruijtzer, W.J.M. Smiths, G. Essink, E.A. de Kock, "YAPI: Application Modeling for Signal Processing Systems," in *Proceedings of the 37th Conference on Design Automation (DAC '00)*, 2000, pp. 402–405.

[Callahan et al 2004] D. Callahan, B.L. Chamberlain, and H.P. Zima. "The Cascade High Productivity Language," in *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, IEEE Computer Society, Apr. 2004, pp. 52–60.

[Chandrakasan et al 1992] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, 1992, pp. 473–484.

[Charles et al 2005] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, Oct. 2005.

[Chen 2006] Y.K. Chen, Private Communication, Jun. 2006.

[Chinnery 2006] D. Chinnery, *Low Power Design Automation*, Ph.D. dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA, 2006.

[Chong and Catanzaro 2006] J. Chong and B. Catanzaro, Excel spreadsheet.

[Chung et al 2006] E.S. Chung, J.C. Hoe, and B. Falsafi, "ProtoFlex: Co-Simulation for Component-wise FPGA Emulator Development," in the *2nd Workshop on Architecture Research using FPGA Platforms (WARFP 2006)*, Feb. 2006.

[Colella 2004] P. Colella, "Defining Software Requirements for Scientific Computing," presentation, 2004.

[Cooley and Tukey 1965] J. Cooley and J. Tukey, "An algorithm for the machine computation of the complex Fourier series," *Mathematics of Computation*, vol. 19, 1965, pp. 297–301.

[Cristianini and Shawe-Taylor 2000] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines*, Cambridge University Press, Cambridge, 2000.

[Dally and Towles 2001] W.J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Proceedings of the 38th Conference on Design Automation (DAC '01)*, 2001, pp. 684–689.

[Dean and Ghemawat 2004] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implemention*, San Francisco, CA, Dec. 2004.

[Deitz 2005] S.J. Deitz, *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*, PhD thesis, University of Washington, Feb. 2005.

[Demmel et al 1999] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu, "A supernodal approach to sparse partial pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 3, pp. 720–755.

[Demmel et al 2002] J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, X. Li, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. Yoo, "Design, Implementation and Testing of Extended and Mixed Precision BLAS," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, Jun. 2002, pp. 152–205.

[Dubey 2005] P. Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *Technology@Intel Magazine*, Feb. 2005.

[Duda and Hart 1973] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*, New York: Wiley, 1973.

[Eatherton 2005] W. Eatherton, "The Push of Network Processing to the Top of the Pyramid," keynote address at *Symposium on Architectures for Networking and Communications Systems*, Oct. 26–28, 2005. Slides available at: http://www.cesr.ncsu.edu/ancs/slides/eathertonKeynote.pdf

[Ebcioglu et al 2006] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, J. Urbanic, "An Experiment in Measuring the Productivity of Three Parallel Programming Languages," in *Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (P-PHEC 2005)*, Feb. 2005.

[Edinburg 2006] University of Edinburg, "QCD-on-a-chip, (QCDOC)," http://www.pparc.ac.uk/roadmap/rmProject.aspx?q=82

[EEMBC 2006] Embedded Microprocessor Benchmark Consortium. http://www.eembc.org

[FLUENT 2006] FIDAP finite element for computational fluid dynamics analysis home page. http://www.fluent.com/software/fidap/index.htm

[Frigo and Johnson 1998] M. Frigo and S.G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '98)*, Seattle, WA, May 1998, vol. 3, pp. 1381–1384.

[Frigo and Johnson 2005] M. Frigo and S.G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, pp. 216–231.

[Gelsinger 2001] P.P. Gelsinger, "Microprocessors for the new millennium: Challenges, opportunities, and new frontiers," in *Proceedings of the International Solid State Circuits Conference (ISSCC)*, 2001, pp. 22–25.

[Gonzalez and Horowitz 1996] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 9, 1996, pp. 1277–1284.

[Goodale et al 2003] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf, "The cactus framework and toolkit: Design and applications," in *Vector and Parallel Processing (VECPAR'2002), 5th International Conference*, Springer, 2003.

[Gordon et al 2002] M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A.A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A Stream Compiler for Communication-Exposed Architectures," MIT Technology Memo TM-627, Cambridge, MA, Mar. 2002.

[Granlund et al 2006] T. Granlund et al. GNU Superoptimizer FTP site. ftp://prep.ai.mit.edu/pub/gnu/superopt

[Gries 2004] M. Gries, "Methods for Evaluating and Covering the Design Space during Early Design Development," *Integration, the VLSI Journal*, Elsevier, vol. 38, no. 2, Dec. 2004, pp. 131–183.

[Gries and Keutzer 2005] M. Gries and K. Keutzer (editors), *Building ASIPs: The MESCAL Methodology*, Springer, 2005.

[Gursoy and Kale 2004] A. Gursoy and L.V. Kale, "Performance and Modularity Benefits of Message-Driven Execution," *Journal of Parallel and Distributed Computing*, vol. 64, no. 4, Apr. 2004, pp. 461–480.

[Hammond et al 2004] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency (TCC)," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Jun. 2004.

[Harstein and Puzak 2003] A. Harstein and T. Puzak, "Optimum Power/Performance Pipeline Depth," in *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, Dec. 2003, pp. 117–126.

[Hauser and Wawrzynek 1997] J.R. Hauser and J. Wawrzynek, "GARP: A MIPS processor with a reconfigurable coprocessor," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1997, pp. 12–21.

[Hennessy and Patterson 2007] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Morgan Kauffman, San Francisco, 2007.

[Heo and Asanovic 2004] S. Heo and K. Asanovic, "Power-Optimal Pipelining in Deep Submicron Technology," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2004, pp. 218–223.

[Herlihy and Moss 1993] M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, 1993, pp. 289–300.

[Hewitt et al 1973] C. Hewitt, P. Bishop, and R. Stieger, "A Universal Modular Actor Formalism for Artificial Intelligence," in *Proceedings of the 1973 International Joint Conference on Artificial Intelligence*, 1973, pp. 235–246.

[Hillis and Tucker 1993] W.D. Hillis and L.W. Tucker, "The CM-5 Connection Machine: A Scalable Supercomputer," *Communications of the ACM*, vol. 36, no. 11, Nov. 1993, pp. 31–40.

[Hochstein et al 2005] L. Hochstein, J. Carver, F. Shull, S. Asgari, V.R. Basili, J.K. Hollingsworth, M. Zelkowitz. "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," *International Conference for High Performance Computing, Networking and Storage* (SC'05). Nov. 2005.

[Horowitz 2006] M. Horowitz, personal communication and Excel spreadsheet.

[Hrishikesh et al 2002] M.S. Hrishikesh, D. Burger, N.P. Jouppi, S.W. Keckler, K.I. Farkas, and P. Shivakumar, "The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*, May 2002, pp. 14–24.

[HPCS 2006] DARPA High Productivity Computer Systems home page. http://www.highproductivity.org/

[IBM 2006] IBM Research, "MD-GRAPE." http://www.research.ibm.com/grape/

[Im et al 2005] E.J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, Spr. 2004, pp. 135–158.

[Intel 2004] Intel Corporation, "Introduction to Auto-Partiontioning Programming Model," Literature number 254114-001, 2004.

[Kamil et al 2005] S.A. Kamil, J. Shalf, L. Oliker, and D. Skinner, "Understanding Ultra-Scale Application Communication Requirements," in *Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, Oct. 6–8, 2005, pp. 178–187. (LBNL-58059)

[Kantowitz and Sorkin 1983] B.H. Kantowitz and R.D. Sorkin, *Human Factors: Understanding People-System Relationships*, New York, NY, John Wiley & Sons, 1983.

[Killian et al 2001] E. Killian, C. Rowen, D. Maydan, and A. Wang, "Hardware/Software Instruction set Configurability for System-on-Chip Processors," in *Proceedings of the 38th Conference on Design Automation (DAC '01)*, 2001, pp. 184–188.

[Koelbel et al 1993] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel, *The High Performance Fortran Handbook*, The MIT Press, 1993. ISBN 0262610949.

[Kozyrakis and Olukotun 2005] C. Kozyrakis and K. Olukotun, "ATLAS: A Scalable Emulator for Transactional Parallel Systems," in *Workshop on Architecture Research using FPGA Platforms, 11th International Symposium on High-Performance Computer Architecture (HPCA-11 2005)*, San Francisco, CA, Feb. 13, 2005.

[Kumar et al 2003] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen, "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, Dec. 2003.

[Kuo et al 2005] K. Kuo, R.M. Rabbah, and S. Amarasinghe, "A Productive Programming Environment for Stream Computing," in *Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (P-PHEC 2005)*, Feb. 2005.

[Kuon and Rose 2006] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proceedings of the Internation Symposium on Field Programmable Gate Arrays (FPGA '06)*, Monterey, California, USA, ACM Press, New York, NY, Feb. 22–24, 2006, pp. 21–30.

[Massalin 1987] H. Massalin, "Superoptimizer: a look at the smallest program," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, CA, 1987, pp. 122–126.

[MathWorks 2004] The MathWorks, "Real-Time Workshop 6.1 Datasheet," 2004.

[MathWorks 2006] The MathWorks, MATLAB Function Reference, 2006.

[Mattson 1999] T. Mattson, "A Cognitive Model for Programming," U. Florida whitepaper, 1999.
Available at
http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/Psychology/CognitiveModel.htm

[Monaghan 1982] J.J. Monaghan, "Shock Simulation by the Particle Method SPH," *Journal of Computational Physics*, vol. 52, 1982, pp. 374–389.

[Mukherjee et al 2005] S.S. Mukherjee, J. Emer, and S.K. Reinhardt, "The Soft Error Problem: An Architectural Perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11 2005)*, Feb. 2005, pp. 243–247.

[Nyberg et al 2004] C. Nyberg, J. Gray, C. Koester, "A Minute with Nsort on a 32P NEC Windows Itanium2 Server", http://www.ordinal.com/NsortMinute.pdf, 2004.

[Opencores 2006] Opencores home page. http://www.opencores.org

[OpenMP 2006] OpenMP home page. http://www.openmp.org

[OpenSPARC 2006] OpenSPARC home page. http://opensparc.sunsource.net

[OSKI 2006] OSKI home page. http://bebop.cs.berkeley.edu/oski/about.html

[Pancake and Bergmark 1990] C.M. Pancake and D. Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Programmers?" *IEEE Computer*, vol. 23, no. 12, Dec. 1990, pp. 13–23.

[Patterson 2004] D. Patterson, "Latency Lags Bandwidth," *Communications of the ACM*, vol. 47, no. 10, Oct. 2004, pp. 71–75.

[Patterson et al 1997] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM: IRAM," *IEEE Micro*, vol. 17, no. 2, Mar.–Apr. 1993, pp. 34–44.

[Paulin 2006] P. Paulin, personal communication and Excel spreadsheet.

[Plishker et al 2004] W. Plishker, K. Ravindran, N. Shah, K. Keutzer, "Automated Task Allocation for Network Processors," in *Network System Design Conference Proceedings*, Oct. 2004, pp. 235–245.

[Poole et al 1998] D. Poole, A. Mackworth and R. Goebel, *Computational Intelligence: A Logical Approach*, Oxford University Press, New York, 1998.

[Power.org 2006] Power.org home page. http://www.power.org

[Pthreads 2004] IEEE Std 1003.1-2004, *The Open Group Base Specifications Issue 6*, section 2.9, IEEE and The Open Group, 2004.

[Pyla et al 2004] P.S. Pyla, M.A. Perez-Quinones, J.D. Arthur, H.R. Hartson, "What we should teach, but don't: Proposal for cross pollinated HCI-SE Curriculum," in *Proceedings of ASEE/IEEE Frontiers in Education Conference*, Oct. 2004, pp. S1H/17–S1H/22.

[Rajwar and Goodman 2002] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, ACM Press, New York, NY, USA, Oct. 2002, pp. 5–17.

[Reason 1990] J. Reason, *Human error*, New York, Cambridge University Press, 1990.

[Rosenblum 2006] M. Rosenblum, "The Impact of Virtualization on Computer Architecture and Operating Systems," Keynote Address, *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, San Jose, California, Oct. 23, 2006.

[Rowen and Leibson 2005] C. Rowen and S. Leibson, *Engineering the Complex SOC : Fast, Flexible Design with Configurable Processors*, Prentice Hall, 2nd edition, 2005.

[Scott 1996] S.L. Scott. "Synchronization and communication in the T3E multiprocessor." In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, MA, Oct. 1996.

[Seffah 2003] A. Seffah, "Learning the Ropes: Human-Centered Design Skills and Patterns for Software Engineers' Education," *Interactions*, vol. 10, 2003, pp. 36–45.

[Shah et al 2004a] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer, "NP-Click: A Productive Software Development Approach for Network Processors," *IEEE Micro*, vol. 24, no. 5, Sep. 2004, pp. 45–54.

[Shah et al 2004b] N. Shah, W. Plishker, and K. Keutzer, "Comparing Network Processor Programming Environments: A Case Study," *2004 Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, Feb. 2004.

[Shalf et al 2005] J. Shalf, S.A. Kamil, L. Oliker, and D. Skinner, "Analyzing Ultra-Scale Application Communication Requirements for a Reconfigurable Hybrid Interconnect," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*, Seattle, WA, Nov. 12–18, 2005. (LBNL-58052)

[Singh et al 1992] J.P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," in *Computer Architecture News*, Mar. 1992, vol. 20, no. 1, pages 5–44.

[Snir et al 1998] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference (Vol. 1)*. The MIT Press, 1998. ISBN 0262692155.

[Solar-Lezama et al 2005] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu, "Programming by Sketching for Bit-Streaming Programs," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, Jun. 2005, pp. 281–294.

[Soteriou et al 2006] V. Soteriou, H. Wang, L.-S. Peh, "A Statistical Traffic Model for On-Chip Interconnection Networks," in *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06)*, Sep. 2006, pp. 104–116.

[SPEC 2006] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/index.html

[Srinivasan et al 2002] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski, and P.G. Emma, "Optimizing pipelines for power and performance," in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, 2002, pp. 333–344.

[Sterling 2006] T. Sterling, "Multi-Core for HPC: Breakthrough or Breakdown?" Panel discussion at the *International Conference for High Performance Computing, Networking and Storage (SC'06)*, Nov. 2006..Slides available at http://www.cct.lsu.edu/~tron/SC06.html

[Sylvester et al 1999] D. Sylvester, W. Jiang, and K. Keutzer, "Berkeley Advanced Chip Performance Calculator," http://www.eecs.umich.edu/~dennis/bacpac/index.html

[Sylvester and Keutzer 1998] D. Sylvester and K. Keutzer, "Getting to the Bottom of Deep Submicron," in *Proceedings of the International Conference on Computer-Aided Design*, Nov. 1998, pp. 203–211.

[Sylvester and Keutzer 2001] D. Sylvester and K. Keutzer, "Microarchitectures for systems on a chip in small process geometries," *Proceedings of the IEEE*, Apr. 2001, pp. 467–489.

[Teja 2003] Teja Technologies, "Teja NP Datasheet," 2003.

[Teragrid 2006] NSF Teragrid home page. http://www.teragrid.org/

[Tokyo 2006] University of Tokyo, "GRAPE," http://grape.astron.s.u-tokyo.ac.jp

[Vadhiyar et al 2000] S. Vadhiyar, G. Fagg, and J. Dongarra, "Automatically Tuned Collective Communications," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Nov. 2000.

[Vahala et al 2005] G. Vahala, J. Yepez, L. Vahala, M. Soe, and J. Carter, "3D entropic lattice Boltzmann simulations of 3D Navier-Stokes turbulence," in *Proceedings of the 47th Annual Meeting of the APS Division of Plasma Phsyics*, 2005.

[Vetter and McCracken 2001] J.S. Vetter and M.O. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," in *Proceedings of the Eigth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPOPP)*, 2001, pp. 123–132.

[Vetter and Mueller 2002] J.S. Vetter and F. Mueller, "Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, 2002, pp. 272–281.

[Vetter and Yoo 2002] J.S. Vetter and A. Yoo, "An Empirical Performance Evaluation of Scalable Scientific Applications," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002.

[Vuduc et al 2006] R. Vuduc, J. Demmel, and K. Yelick, "OSKI: Optimized Sparse Kernel Interface," http://bebop.cs.berkeley.edu/oski/.

[Warren 2006] H. Warren, A Hacker's Assistant. http://www.hackersdelight.org

[Wawrzynek et al 2006] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J.C. Joe, D. Chiou, and K. Asanovic, "RAMP: A Research Accelerator for Multiple Processors," U.C. Berkeley technical report, 2006.

[Weinburg 2004] B. Weinberg, "Linux is on the NPU control plane," *EE Times*, Feb. 9, 2004.

[Whaley and Dongarra 1998] R.C. Whaley and J.J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998.

[Van der Wijngaart 2002] R.F. Van der Wijngaart, "NAS Parallel Benchmarks Version 2.4," NAS technical report, NAS-02-007, Oct. 2002.

[Wind River 2006] Wind River home page.
http://www.windriver.com/products/platforms/general_purpose/index.html

[Wolfe 2004] A. Wolfe, "Intel Clears Up Post-Tejas Confusion," VARBusiness, May 17, 2004.
http://www.varbusiness.com/sections/news/breakingnews.jhtml?articleId=18842588

[Woo et al 1995] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA '95)*, Santa Margherita Ligure, Italy, Jun. 1995, pp. 24–36.

[Wulf and McKee 1995] W.A. Wulf and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, no. 1, Mar. 1995, pp. 20–24.

[Xu et al 2003] M. Xu, R. Bodik, and M.D. Hill, "A 'Flight Data Recorder' for Enabling Full-System Multiprocessor Deterministic Replay," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, 2004.

[Zarlink 2006] Zarlink, "PDSP16515A Stand Alone FFT Processor,"
http://products.zarlink.com/product_profiles/PDSP16515A.htm.