SANDIA REPORT

SAND2009-6229
Unlimited Release
Printed September 2009

Building More Powerful Less Expensive Supercomputers Using Processing-In-Memory (PIM) LDRD Final Report

Richard C. Murphy

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

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

Sandia National Laboratories

SAND2009-6229
Unlimited Release
Printed September 2009

Building More Powerful Less Expensive Supercomputers Using Processing-In-Memory (PIM) LDRD Final Report

Richard C. Murphy
Scalable Computer Architectures Department, 1422
Sandia National Laboratories
P.O. Box 5800, MS-1319
Albuquerque, NM 87185-1319

Abstract

This report summarizes the activities of the Processing-In-Memory (PIM) LDRD project from FY07-FY09.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Introduction

This report details the accomplishments of the "Building More Powerful Less Expensive Supercomputers Using Processing-In-Memory (PIM)" LDRD ("PIM LDRD", number 105809) for FY07-FY09.

Latency dominates all levels of supercomputer design. Within a node, increasing memory latency, relative to processor cycle time, limits CPU performance. Between nodes, the same increase in relative latency impacts scalability. Processing-In-Memory (PIM) is an architecture that directly addresses this problem using enhanced chip fabrication technology and machine organization. PIMs combine high-speed logic and dense, low-latency, high-bandwidth DRAM, and lightweight threads that tolerate latency by performing useful work during memory transactions. This work examines the potential of PIM-based architectures to support mission critical Sandia applications and an emerging class of more data intensive informatics applications.

This work has resulted in a stronger architecture/implementation collaboration between 1400 and 1700. Additionally, key technology components have impacted vendor roadmaps, and we are in the process of pursuing these new collaborations. This work has the potential to impact future supercomputer design and construction, reducing power and increasing performance.

This final report is organized as follow: this summary chapter discusses the impact of the project (Section 1), provides an enumeration of publications and other public discussion of the work (Section 1), and concludes with a discussion of future work and impact from the project (Section 1). The appendix contains reprints of the refereed publications resulting from this work.

Impact

This LDRD has helped to establish a processor and memory architecture research focus within the lab, facilitated collaboration between 1400 and 1700, and served as an intellectual basis for a large-scale proposal effort to DARPA targeted at the end of the Fall of 2009. A summary of impact is:

- A collaboration with Micron was started to examine alternatives for implementing PIM-based computers, with the focus now being on 3D integration of DRAM and logic. Sandia's Microelectronics center has deep experience in 3DI, and it has allowed us to begin post-LDRD collaborating with industry on both near- and long-term supercomputers.
- We have simulated a set of real application and application kernels, particularly those from the Mantevo LDRD project and demonstrated that those applications are memory bound in performance.
- We have identified processor inefficiencies related to the large data sets in HPC applications and the fact that commodity processors are not designed to address them.
- We have quantitatively demonstrated that memory latency, not memory bandwidth limits performance.
- We have developed a multithreaded programming model, qthreads, that increased on-node application concurrency. Given that increased concurrency lowers effective latency, this is a key method for creating future memory latency tolerant systems.
- We have enhanced the SST simulator to include improved memory models and provide better PIM processor simulations. This can be leveraged by academic and industry collaborators in the future.
- We have published six refereed publications and given three invited talks related to the LDRD, and expect future results to provide impact to two additional papers in the Fall.

Publications, Presentations, and Press

This section enumerates public presentation of our work.

Refereed Publications

- Wheeler, Kyle, Douglas Thain, and Richard Murphy, Portable Performance from Workstation to Supercomputer: Distributing Data Structures with Qthreads, Proceedings of the First Workshop on Programming Models for Emerging Architectures, 2009.

- Barrett, Brian W., and Jonathan W. Berry, Richard C. Murphy, and Kyle B. Wheeler, Implementing a Portable Multi-threaded Graph Library: the MTGL on Qthreads, International Parallel and Distributed Processing Symposium 2009 (IPDPS09), Rome, Italy, Pages 1–8, May 2009.
- Murphy, Richard C., DOE's Institute for Advanced Architecture and Algorithms: an Application-Driven Approach, Journal of Physics Conference Series 180(2009), 012044.
- Wheeler, Kyle, Richard C. Murphy, and Douglas Thain, Qthreads: An API for Programming with Millions of Lightweight Threads in the Proceedings of the Workshop on Multithreaded Architectures and Applications, Miami, FL, 2008.
- Murphy, Richard C. On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance in the IEEE International Symposium on Workload Characterization 2007 (IISWC07), Boston, MA, September 27-29, 2007.
- Murphy, Richard C. and Peter M. Kogge, On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications, IEEE Transactions on Computers 56(7): 937-945, July 2007.

Invited Talks

- DOE's Institute for Advanced Architecture and Algorithms: an Application-Driven Approach, SciDAC 2009, June 14-18, San Diego, CA.
- Data Movement Dominates: An Application-Centric View of High Performance Interconnects, HSD 2009 20th Annual Workshop on Interconnects within High Speed Digital Systems, May 3-6, 2009, Santa Fe, NM.
- Can We Continue to Build Supercomputers Out of Processors Optimized for Laptops?, 13th Workshop on Distributed Supercomputing (SOS 13), March 9-12, 2009, Hilton Head, SC.

Coverage in the Press

- Moore, Samuel K., "Multicore Is Bad News for Supercomputers", IEEE Spectrum, November 2008.

Workshops

This LDRD also contributed to the ideas behind the CSRI Workshop Memory Opportunities for High Performance Computing (MOHPC), January 9-10, 2008.

Future Work

We plan to continue to pursue three key points of impact from this work:

1. Work with Micron on advanced memory technologies for near-term (2014) and Exascale (2017) supercomputers will continue, and expand to include both DRAM and nonvolatile memory. This work will continue through the DOE Institute for Advanced Architecture and Algorithms (IAA), and the Sandia/Los Alamos Alliance for Computing at Extreme Scale (ACES). We believe impact can be shown in our Trinity supercomputer in the 2014 time-frame. This work has also generated interest from Other Government Agency (OGA) activities.

2. The qthreads programming model will be expanded and serve as the basis for an **experimental** on-node programming model for future machines. We expect qthreads will have to be expanded to include additional communication and synchronization primitives, particularly transactional memory, but that real applications could be targeted to a qthreads environment for evaluation purposes in FY10 and FY11.

3. The X-caliber strawman architecture, generated from this LDRD, will serve as the basis for a Sandia-led proposal to the DARPA UHPC program. We expect a BAA to be issued in the Fall of 2009, and have formed a multi-institutional team including industry and academia.

Appendix A

Reproduction of Referred Papers

The following pages reproduce the refereed publications enumerated in Section 1.

On The Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications

Richard C. Murphy†, *Member, IEEE,* Peter M. Kogge, *Fellow, IEEE*

***Abstract*— This paper compares the SPEC Integer and Floating Point suites to a set of real-world applications for high performance computing at Sandia National Laboratories. These applications focus on the high-end scientific and engineering domains, however the techniques presented in this paper are applicable to any application domain. The applications are compared in terms of three memory properties: first, *temporal locality* (or reuse over time); second, *spatial locality* (or the use of data "near" data that has already been accessed); and third, *data intensiveness* (or the number of unique bytes the application accesses). The results show that real world applications exhibit significantly less spatial locality, often exhibit less temporal locality, and have much larger data sets than the SPEC benchmark suite. They further quantitatively demonstrates the memory properties of real supercomputing applications.**

***Index Terms*— B.8.2 Performance Analysis and Design Aids, C.1.0 General, C.4.c Measurement techniques, C.4.g Measurement, evaluation, modeling, simulation of multiple-processor systems**

I. Introduction

The selection of benchmarks relevant to the supercomputing community is challenging at best. In particular, there is a discrepancy between the workloads that are most extensively studied by the computer architecture community, and the codes relevant to high performance computing. This paper examines these differences *quantitatively* in terms of the memory characteristics of a set of a real applications from the high-end science and engineering domain as they compare to the SPEC CPU2000 benchmark suite, and more general High Performance Computing (HPC) benchmarks. The purpose of this paper is two-fold: first to demonstrate what general memory characteristics the computer architecture community should look for when identifying benchmarks relevant to HPC (and how they differ from SPEC); and second, to quantitatively explain application's memory characteristics to the HPC community, which often relies on intuition when discussing memory locality. Finally, although most studies discuss temporal and spatial locality when referring to memory performance, this work introduces a new measure *data intensiveness* that serves as the biggest differentiator in application properties between real applications and benchmarks. These techniques can be applied to any architecture-independent comparison of any benchmark or application suite, and this study could be repeated for other markets of interest. The three key characteristics of the application are:

1) Temporal Locality: the reuse over time of a data item from memory;
2) Spatial Locality: the use of data items in memory near other items that have already been used; and,
3) Data Intensiveness: the amount of unique data the application accesses.

Quantifying each of these three measurements is extremely difficult (see Section IV and Section II). Beyond quantitatively defining the measurements, there are two fundamental problems: first, choosing the applications to measure; and second, performing the measurement in an architecture-independent fashion that allows general conclusions to be drawn about the application rather than specific observations of the applications performance on one particular architecture. This paper addresses the former problem by using a suite of real codes that consume significant compute time at Sandia National Laboratories; and it addresses the latter by defining the metrics to be orthogonal to each other, and measuring them in an architecture independent fashion. Consequently, these results (and the techniques used to generate them) are applicable for comparing any set of benchmarks or applications memory properties without regard to how those properties perform on any particular architectural implementation.

The remainder of this paper is organized as follows: Section II examines the extensive related work in measuring spatial and temporal locality, as well as application's working sets; Section III describes the Sandia integer and floating point applications, as well as the SPEC suite; Section IV quantitatively defines the measures of temporal locality, spatial locality, and data intensiveness; Section V compares the application's properties; Section VI presents the results; and Section VII ends with the conclusions.

II. Related Work

Beyond the somewhat intuitive definitions of spatial and temporal locality provided in computer architecture text books [14], [27], there have been numerous attempts to quantitatively define spatial and temporal locality [37]. Early research in computer architecture [7] examined *working sets*, or the data actively being used by a program, in the context of paging. That work focused on efficiently capturing the working set in limited core memory, and has been an active area of research [9], [31], [35].

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

More recent work is oriented towards addressing the memory wall. It examines the spatial and temporal locality properties of cache accesses, and represent modern hierarchical memory structures [6], [10], [32]. Compiler writers have also extensively examined the locality properties of applications in the context of code generation and optimization [11], [12].

In addition to definitions that are continuously refined, the methodology for modeling and measuring the working set has evolved continuously. Early analytical models were validated by small experiments [7], [9], [31], [35], while modern techniques have focused on the use of trace-based analysis and full system simulation [18], [21], [32].

Because of its preeminence in computer architecture benchmarks, the SPEC suite has been extensively analyzed [12], [18], [19], [21], [33], as have other relevant workloads such as database and OLTP applications [3], [6], [20].

Finally, the construction of specialized benchmarks such as the HPC Challenge RandomAccess benchmark [8] or the STREAM benchmark [22] is specifically to address memory performance. Real world applications on a number of platforms have been studied extensively [26], [36].

III. Applications and Benchmarks

This section describes a set of floating point and integer applications from Sandia National Laboratories, as well as the SPEC Floating Point and Integer benchmark suites to which they will be compared. The HPC Challenge RandomAccess benchmark, which measures random memory accesses in GUPS (Giga-Updates Per Second), and the STREAM benchmark, which measures effective bandwidth, are used as comparison points to show very basic memory characteristics. Finally, LINPACK, the standard supercomputing benchmark used to generate the Top500 list is included for comparison. In the case of MPI codes, the user portion of MPI calls is included in the trace.

A. Floating Point Benchmarks

Real scientific applications tend to be significantly different from common processor benchmarks, such as the SPEC suite. Their datasets are larger, the applications themselves are more complex, and they are designed to run on large-scale machines. The following benchmarks were selected to represent critical problems in supercomputing seen by the largest scale deployments in the United States. The input sets were all chosen to be representative of real problems, or, when they are benchmark problems, are the typical performance evaluation benchmarks used during new system deployment. Two of the codes are benchmarks, sPPM (see Section III-A.5), which is part of the ASCI 7x benchmark suite (that set requirements for the ASCI Purple supercomputer), and Cube3 which is used as a simple driver for the Trilinos linear algebra package. The sPPM code is a slightly simplified version of a real-world problem, and, in the case of Trilinos, linear algebra is so fundamental to many areas of scientific computing that studying core kernels is significantly important.

All of the codes are written for MPPs using the MPI programming model, but, for the purposes of this study, were traced as a single node run of the application. Even without the use of MPI the codes are structured to be MPI-scalable. Other benchmarking (both performance register and trace-based) has shown that the local memory access patterns for a single node of the application and serial runs are substantially the same.

1) LAMMPS: LAMMPS represents a classic molecular dynamics simulation designed to represent systems at the atomic or molecular level [28], [29]. The program is used to simulate proteins in solution, liquid crystals, polymers, zeolites, and simple Lenard-Jones systems. The version under study is written in C++, and two significant inputs were chosen for analysis:

- *Lenard Jones Mixture*: This input simulated a 2048 atom system consisting of three different types;
- *Chain*: simulates 32000 atoms and 31680 bonds.

LAMMPS consists of approximately 30,000 lines of code.

2) CTH: CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed over the last three decades at Sandia National Laboratories [16]. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials. CTH supports multiple types of meshes:

- Three-dimensional rectangular meshes;
- two-dimensional rectangular and cylindrical meshes; and
- one-dimensional rectilinear, cylindrical, and spherical meshes.

It uses second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results. CTH is used extensively within the Department of Energy laboratory complexes for studying armor/anti-armor interactions, warhead design, high explosive initiation physics and weapons safety issues. It consists of approximately 500,000 lines of Fortran and C.

CTH has two modes of operation: with or without adaptive mesh refinement (AMR)[1]. Adaptive mesh refinement changes the application properties significantly and is useful for only certain types of input problems. One AMR problem and two non-AMR problems were chosen for analysis.

Three input sets were examined:

- **2-Gas:** The input set uses an $80\times80\times80$ mesh to simulate two gases intersecting on a 45 degree plane. This is the most "benchmark-like" (e.g., simple) input set, and is included to better understand how representative it is of real problems.
- **Explosively Formed Projectile (EFP):** The simulation represents a simple Explosively Formed Projectile (EFP) that was designed by Sandia National Laboratories staff. The original design was a combined experimental and modeling activity where design changes were evaluated computationally before hardware was fabricated for testing. The design features a concave copper liner that is formed into an effective fragment by the focusing of shock waves from the detonation of the high explosive.

[1]AMR typically uses graph partitioning as part of the refinement, two algorithms for which are part of the integer benchmarks under study. One of the most interesting results of including a code like CTH in a "benchmark suite" is its complexity.

The measured fragment size, shape, and velocity is accurately (within 5%) modeled by CTH.

- **CuSt AMR:** This input problem simulates a 4.52 km/s impact of a 4 mm copper ball on a steel plate at a 90 degree angle. Adaptive mesh refinement is used in this problem.

3) Cube3: Cube3 is meant to be a generic linear solver and drives the Trilinos [15] frameworks for parallel linear and eigensolvers. Cube3 mimics a finite element analysis problem by creating a beam of hexagonal elements, then assembling and solving a linear system. The problem can be varied by width, depth, and degrees of freedom (e.g., temperature, pressure, velocity, or whatever physical modeling the problem is meant to represent). The physical problem is three dimensional. The number of equations in the linear system is equal to the number of nodes in the mesh multiplied by the degrees of freedom at each node. There are two variants based on how the sparse matrices are stored:

- **CRS:** a 55x55 sparse compressed row system; and
- **VBR:** a 32x16 variable block row system.

These systems were chosen to represent a large system of equations.

4) MPSalsa: MPSalsa performs high resolution 3D simulations of reacting flow problems [34]. These problems require both fluid flow and chemical kinetics modeling.

5) sPPM: The sPPM [5] benchmark is part of the ASCI Purple benchmark suite as well as the 7$\times$ application list for ASCI Red Storm. It solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code. The hydrodynamics algorithm requires three separate sweeps through the mesh per time step. Each sweep requires approximately 680 flops to update the state variables for each cell. The sPPM code contains over 4000 lines of mixed Fortran 77 and C routines. The problem solved by sPPM involves a simple, but strong (about Mach 5) shock propagating through a gas with a density discontinuity.

B. Integer Benchmarks

While floating point applications represent the classic supercomputing workload, problems in discrete mathematics, particularly graph theory, are becoming increasingly prevalent. Perhaps most significant of these are fundamental graph theory algorithms. These routines are important in the fields of proteomics, genomics, data mining, pattern matching and computational geometry (particularly as applied to medicine). Furthermore, their performance emphasizes the critical need to address the von Neumann bottleneck in a novel way. The data structures in question are very large, sparse, and referenced *indirectly* (e.g., through pointers) rather than as regular arrays. Despite their vital importance, these applications are significantly underrepresented in computer architecture research, and there is currently little joint work between architects and graph algorithms developers.

In general, the integer codes are more "benchmark" problems (in the sense that they use non-production input sets), heavily weighted towards graph theory codes, than are the floating point benchmarks.

1) Graph Partitioning: There are two large-scale graph partitioning heuristics included here: Chaco [13] and Metis [17]. Graph partitioning is used extensively in automation for VLSI circuit design, static and dynamic load balancing on parallel machines, and numerous other applications. The input set in this work consists of a 143,437 vertex and 409,593 edge graph to be partitioned into 1,024 balanced parts (with minimum edge cut between partitions).

2) Depth First Search (DFS): DFS implements a Depth First Search on a graph with 2,097,152 vertices and 25,690,112 edges. DFS is used extensively in higher-level algorithms, including identifying connected components, tree and cycle detection, solving the two-coloring problem, finding Articulation Vertices (e.g., the vertex in a connected graph that, when deleted, will cause the graph to become a disconnected graph), and topological sorting.

3) Shortest Path: Shortest Path computes the shortest path on a graph of 1,048,576 vertices and 7,864,320 edges, and incorporates a breadth first search. Extensive applications exist in real world path planning and networking and communications.

4) Isomorphism: The graph isomorphism problem determines whether or not two graphs have the same shape or structure. Two graphs are isomorphic if there exists a one-to-one mapping between vertices and edges in the graph (independent of how those vertices and edges are labeled). The problem under study confirms that two graphs of 250,000 vertices and 10 million edges are isomorphic. There are numerous applications in finding similarity (particularly, subgraph isomorphism) and relationships between two differently labeled graphs.

5) BLAST: The Basic Local Alignment Search Tool (BLAST) [1] is the most heavily used method for quickly searching nucleotide and protein databases in biology. The algorithm attempts to find both local and global alignment of DNA nucleotides, as well identifying regions of similarity embedded in two proteins. BLAST is implemented as a dynamic programming algorithm.

The input sequence chosen was obtained by training a hidden Markov model on approximately 15 examples of piggyBac transposons from various organisms. This model was used to search the newly assembled aedes aegypti genome (a mosquito). The best result from this search was the sequence used in the blast search. The target sequence obtained was blasted against the entire aedes aegypti sequence to identify other genes that could be piggyBac transposons, and to double check that the subsequence is actually a transposon.

6) zChaff: The zChaff program implements the Chaff heuristic [23] for finding solutions to the Boolean satisfiability problem. A formula in propositional logic is *satisfiable* if there exists an assignment of truth values to each of its variables that will make the formula true. Satisfiability is critical in circuit validation, software validation, theorem proving, model analysis and verification, and path planning. The zChaff input comes from circuit verification and consists of 1,534 Boolean variables, 132,295 clauses with five instances, that are all satisfiable.

TABLE I
SPEC CPU2000 INTEGER SUITE

Benchmark	Lang.	Description
164.gzip	C	Data Compression
175.vpr	C	FPGA Placement and Routing
176.gcc	C	GNU C Compiler
181.mcf	C	Combinatorial Optimization
186.crafty	C	Chess
197.parser	C	Word Processing
252.eon	C++	Visualization
253.perlbmk	C	PERL
254.gap	C	Group Theory
255.vortex	C	Object Oriented Database
256.bzip2	C	Data compression
300.twolf	C	VLSI Placement and Routing

TABLE II
SPEC FLOATING POINT SUITE

Benchmark	Lang	Description
168.wupwise	F77	Quantum Chromodynamics
171.swim	F77	Shallow Water modeling
172.mgrid	F77	Multi-grid Solver
173.applu	F77	Parabolic PDEs
177.mesa	C	3d Graphics
178.galgel	F90	Comp. Fluid Dynamics
179.art	C	Adaptive Resonance Theory
183.equake	C	Seismic Wave Propagation
187.facerec	F90	Face Recognition
188.ammp	C	Computational Chemistry
189.lucas	F90	Primary Number Testing
191.fma3d	F90	Finite Element Crash Simulation
200.sixtrack	F77	High Energy Physics Accelerator
301.apsi	F77	Pollutant Distribution

C. SPEC

The SPEC CPU2000 suite is by far the most currently studied benchmark suite for processor performance [4]. This work uses both the SPEC-Integer and SPEC-FP components of the suite, as summarized in Tables I and II respectively, as its baseline comparison for benchmark evaluation.

1) SPEC Integer Benchmarks: The SPEC Integer Suite, summarized in Table I, is by far the most studied half of the SPEC suite. It is meant to generally represent workstation class problems. Compiling (176.gcc), compression (164.gzip and 256.bzip2), and systems administration tasks (253.perlbmk) have many input sets in the suite. These tasks tend to be somewhat streaming on average (the perl benchmarks, in particular, perform a lot of line-by-line processing of data files). The more scientific and engineering oriented benchmarks (175.vpr, 181.mcf, 252.eon, 254.gap, 255.vortex, and 300.twolf) are somewhat more comparable to the Sandia integer benchmark suite. However selectively choosing benchmarks from SPEC produces generally less accurate comparisons than using the entire suite (although it would lessen the computational requirements for analysis significantly).

It should be noted that the SPEC suite is specifically designed to emphasize computational rather then memory performance. Indeed, other benchmark suites, such as the STREAM benchmark or RandomAccess focus much more extensively on memory performance. However, given the nature of the memory wall, what is important is a mix of the two. SPEC, in this work, represents the baseline only because it is, architecturally, the most studied benchmark suite. Indeed, a benchmark such as RandomAccess would undoubtedly overemphasize the memory performance at the expense of computation, as compared to the real-world codes in the Sandia suite.

2) SPEC Floating Point Benchmarks: The SPEC Floating Point suite is summarized in Table II, and primarily represents scientific applications. At first glance, these applications would appear very similar to the Sandia Floating Point suite; however the scale of the applications (in terms of execution time, code complexity, and input set size) differs significantly.

D. RandomAccess

The RandomAccess benchmarks is part of the HPC Challenge suite [8] and measures the performance of the memory system by updating random entries in a very large table that is unlikely to be cached. This benchmark is specifically designed to exhibit very low spatial and temporal locality, and a very large data set (as the table update involves very little computation). It represents the most extreme of memory intensive codes, and is used as a comparison point to the benchmarks and real applications in this work.

E. STREAM

The STREAM benchmark [22] is used to measure sustainable bandwidth on a platform and does so via four simple operations performed non-contiguously on three large arrays:

- **Copy:** $a(i) = b(i)$
- **Scale:** $a(i) = q * b(i)$
- **Sum:** $a(i) = b(i) + c(i)$
- **Triad:** $a(i) = b(i) + q * c(i)$

To measure the performance of main memory, the STREAM rule is that the data size is scaled to four times the size of the platform's L2 cache. Because this work is focused on architecture independent numbers, the each array size was scaled to 32MB, which is reasonably large for a workstation.

IV. METHODOLOGY AND METRICS

This work evaluates the temporal and spatial locality characteristics of applications separately. This section describes the methodology used in this work and formally defines the temporal locality, spatial locality, and data intensiveness measures.

A. Methodology

The applications in this were each traced using the Amber instruction trace generator [2] for the PowerPC. Trace files containing 4 billion sequential instructions were generated by identifying and capturing each instruction executed in critical sections of the program. The starting point for each trace was chosen using a combination of performance register profiling of the memory system, code reading, and, in the case of SPEC, accumulated knowledge of good sampling points. The advice of application experts was also used for the Sandia codes. The traces typically represent multiple executions of the main loop (multiple time steps for the Sandia floating point benchmarks). These traces have been used extensively in other work, and are well understood [25].

B. Temporal Locality

The application's *temporal working set* describes its *temporal locality*. As in prior work [25], a temporal working set of size N is modeled as an N byte, fully associative, true least recently used cache with native machine word sized blocks. The hit rate of that cache is used to describe the effectiveness of the fixed-size temporal working set at capturing the application's data set. The same work found that the greatest differentiation between conventional and supercomputer applications occurred in the 32KB-64KB level one cache sized region of the temporal working set. The temporal locality in this work is given by a temporal working set of size 64 KB. The temporal working set is measured over a long-studied 4 billion instruction trace from the core of each application. The number of instructions is held constant for each application. This puts the much shorter running SPEC benchmark suite on comparable footing to the longer running supercomputing applications.

It should be noted that there is significant latitude in the choice of temporal working set size. The choice of a level one cache sized working set is given for two reasons: first, it has been demonstrated to offer the greatest differentiation of applications between the floating point benchmarks in this suite and SPEC FP; and second, while there is no direct map to a conventionally constructed L1 cache, the L1 hit rate strongly impacts performance. There are two other compelling choices for temporal working set size:

1) **Level 2 Cache Sized:** in the 1-8 MB region. Arguably, the hit rate of the cache closest to memory most impacts performance (given very long memory latencies).
2) **Infinite:** describes the temporal hit rate required to capture the application's total data set size.

Given N memory accesses, H of which hit the cache described above, the temporal locality is given by: $\frac{H}{N}$.

C. Spatial Locality

Measuring the spatial locality of an application may be the most challenging aspect of this work. Significant prior work has examined it as the application's stride of memory access. The critical measurement is how quickly the application consumes all the data presented to it in a cache block. Thus, given a cache block size, and a fixed interval of instructions, the spatial locality can be described as the ratio of data the application actually uses (through a load or store) to the cache line size. This work uses an instruction interval of $1,000$ instructions, and a cache block size of 64-bytes. For this work, every $1,000$ instruction window in the application's 4 billion instruction trace is examined for unique loads and stores. Those loads and stores are then clustered into 64-byte blocks, and the ratio of used to unused data in the block is computed. The block size is chosen as a typical conventional cache system's block size. There is much more latitude in the instruction window size. It must be large enough to allow for meaningful computation, while being small enough to report differentiation in the application's spatial locality. For example, a window size of the number of instructions in the application should report that virtually all cache lines are 100% used. The $1,000$ instruction window was chosen based on prior experience with the applications [24].

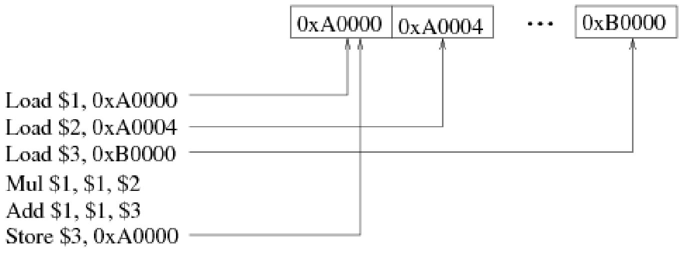


Fig. 1. An example of temporal locality, spatial locality, and data intensiveness.

Given U_{1000} unique bytes accessed in an average interval of 1,000 instructions that are clustered into L 64-byte cache lines, the spatial locality is given by $\frac{U}{64L}$.

D. Data Intensiveness

One critical yet often overlooked metric of an application's memory performance is its *data intensiveness*, or the total amount of unique data that the application accesses (regardless of ordering) over a fixed interval of instructions. Over the course of the entire application, this would be the application's memory footprint. This is not fully captured by the measurements given above, and it is nearly impossible to determine from a cache miss rate. This differs from the application's *memory footprint* because it only includes program data that is accessed via a load or store (where the memory footprint would also include program text). Because a cache represents a single instantiation used to capture an application's working set, a high miss rate could be more indicative of the application accessing a relatively small amount of memory in a temporal order that is poorly suited to the cache's parameters, or that the application exhibits very low spatial locality. It is not necessarily indicative of the application accessing a large data set, which is critical to supercomputing application performance. This work presents the data intensiveness as the total number of unique bytes that the application's trace accessed over its 4 billion instruction interval.

This is directly measured by counting the total number of unique bytes accessed over the given interval of 4 billion instructions. This is the same as the unique bytes measure given above, except it is measured over a larger interval (U_{4B}).

E. An Example

Figure 1 shows an example instruction sequence. Assuming that this is the entire sequence under analysis, each of the metrics given above is computed as follows:

Temporal Locality: is the hit rate of a fully associative cache. The first 3 loads in the sequence (of $0xA0000$, $0xA0004$, and $0xB0000$) miss the cache. The final store (to $0xA0000$) hits the item in the cache that was loaded 3 memory references prior. Thus, the temporal locality is:

$$\frac{1\ hit}{4\ memory\ references} = 0.25$$

Spatial Locality: is the ratio of used to unused bytes in a 64-byte cache line. Assuming that each load requests is 32-bits, there are two unique lines requested, $0xA0000$ (to $0xA0040$), and $0xB0000$ (to $0xB0040$). Two 32-bit words are consumed from $0xA0000$, and 1 32-bit word from $0xB0000$. The spatial locality is calculated as:

$$\frac{12\ consumed\ bytes}{128\ requested\ bytes} = 0.09375$$

Data Intensiveness: is the total number of unique bytes consumed by the stream. In this case, 3 unique 32-bit words are requested, for a total of 12 bytes.

V. Initial Observations of Program Characteristics

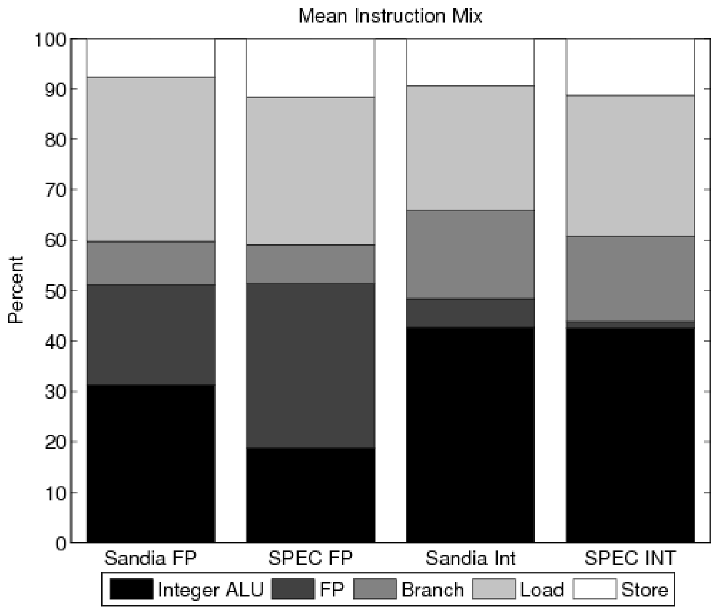


Fig. 2. Benchmark Suite Mean Instruction Mix

Figure 2 shows the instruction mix breakdown for the benchmark suites. Of particular importance is that the Sandia Floating Point applications perform significantly more **integer** operations than their SPEC Floating Point counterparts, in excess of 1.66 times the number of integer operations, in fact. This is largely due to the complexity of the Sandia applications (with many configuration operations requiring integer tests, table look ups requiring integer index calculations, etc.) as well as their typically more complicated memory addressing patterns [30]. This is largely due to the complexity of the algorithm, and the fact that significantly more indirection is used in memory address calculations. Additionally, in the case of the floating point applications, although the Sandia applications perform only about 1.5% more total memory references than their SPEC-FP counterparts, the Sandia codes perform 11% more loads, and only about $\frac{2}{3}$ the number of stores, indicating that the results produced require more memory inputs to produce fewer memory outputs. The configuration complexity can also be seen in that the Sandia codes perform about 11% more branches than their SPEC counterparts.

In terms of the integer applications, the Sandia codes perform about 12.8% fewer memory references over the same number of instructions, however those references are significantly harder to capture in a cache. The biggest difference is that the Sandia Integer codes perform 4.23 times the number of floating point operations as their SPEC Integer counterparts. This is explained by the fact that three of the Sandia Integer benchmarks perform somewhat significant floating point computations.

TABLE III
Sandia Integer Applications with Significant Floating Point Computation

Application	Percent Floating Point Instructions
Chaco	15.84%
DFS	14.74%
Isomorphism	13.41%

Table III summarizes the three Sandia Integer Suite applications with significant floating point work: Chaco, DFS, and Isomorphism. Their floating point ratios are quite below the median for SPEC FP (28.69%), but above the Sandia Floating Point median (10.67%). They are in the integer category because their primary computation is an integer graph manipulation, whereas CTH is in the floating point category even though runs have a lower floating point percentage (a mean over its three input runs of 6.83%), but the floating point work is the primary computation. For example, Chaco is a multilevel partitioner and uses spectral partitioning in its base case, which requires the computation of an eigenvector (a floating point operation). However, graph partitioning is fundamentally a combinatorial algorithm, and consequently in the integer category. In the case of CTH, which is a floating point application with a large number of integer operations, it is a shock physics code. The flops fundamentally represent the "real work", and the integer operations can be accounted for by the complexity of the algorithms, and the large number of table look-ups employed by CTH to find configuration parameters. In either case, the SPEC FP suite is significantly more floating point intensive.

VI. Results

The experimental results given by the metrics from Section IV are presented below. Each graph depicts the temporal locality on the X-axis, and the spatial locality on the Y-axis. The area of each circle on the graph depicts each application's relative data intensiveness (or the total amount of unique data consumed over the instruction stream).

Figure 3 provides the summary results for each suite of applications, and the RandomAccess memory benchmark.

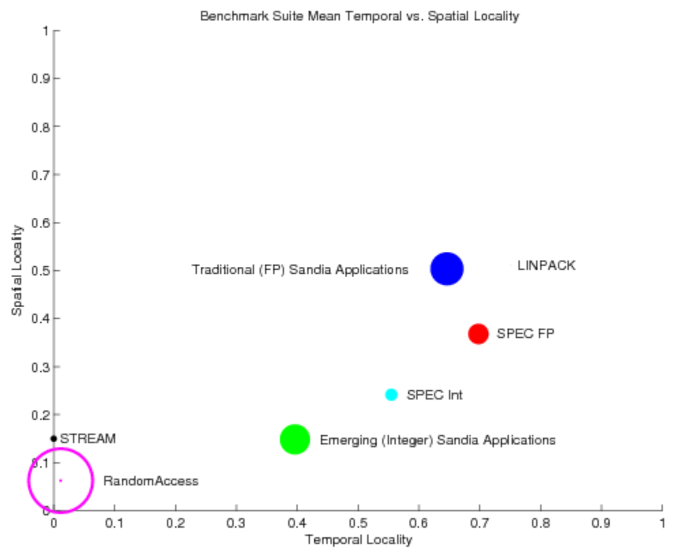


Fig. 3. Mean Temporal vs. Spatial Locality and Data Intensiveness for each benchmark suite.

The Sandia Floating Point suite exhibits approximately 36% greater spatial locality and nearly 7% less temporal locality than its SPEC-FP counterpart. The nearness in temporal locality, and increased spatial locality is somewhat surprising when taken out of context. One would typically expect scientific applications to be less well structured. The critical data intensiveness measure proves the most enlightening. The Sandia FP suite accesses over 2.6 **times** the amount of data as SPEC FP. The data intensiveness is the most important differentiator between the suites. A larger data set size would reflect significantly worse performance in any real cache implementation. Without the additional measure, the applications would appear more comparable. It should be noted that the increased spatial locality seen in the Sandia Floating Point applications is likely because those applications use the MPI programming model, which generally groups data to be operated upon into a buffer for transmission over the network (increasing the spatial locality).

The Sandia integer suite is significantly farther from the SPEC integer suite in all dimensions. It exhibits close to 30% less temporal locality, nearly 40% less spatial locality, and has a unique data set over 5.9 times the size of the SPEC integer suite.

The LINPACK benchmark shows the highest spatial and temporal locality of any benchmark, and by far the smallest data intensiveness (the dot is hardly visible on the graph). It is over 3,000 times smaller than any of the real world Sandia applications. It exhibits 17% less temporal locality and roughly the same spatial locality than the Sandia FP suite. The Sandia Integer suite has half the temporal locality and less than one third the spatial locality.

The STREAM benchmark showed over 100 times less temporal locality than RandomAccess, and 2.4 times the spatial locality. However, critically, the data intensiveness for streams is $1/95$th that of RandomAccess. The Sandia Integer Suite is only 1% *less* spatially local than STREAM, indicating that most of the bandwidth used to fetch a cache line is wasted.

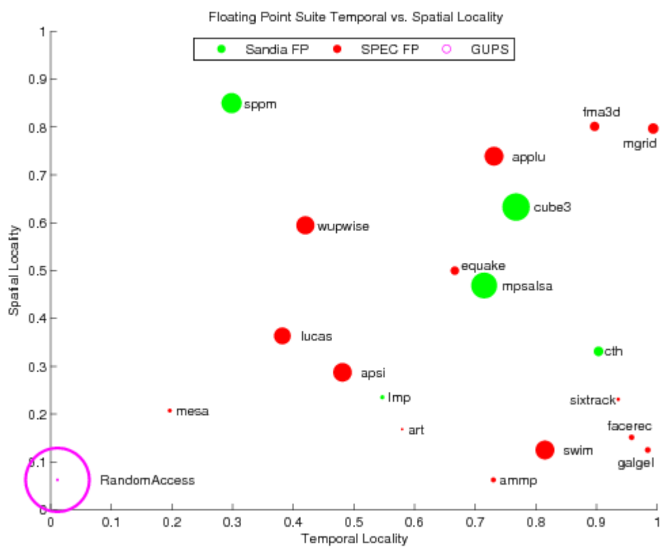
While it is expected that RandomAccess exhibits very low spatial and temporal locality, given its truly random memory access pattern, its data set is $3.7\times$ the size of the Sandia FP suite, $4.5\times$ the size of the Sandia Integer suite, and $9.7\times$ and $26.5\times$ the SPEC floating point and integer suites respectively.

Figure 4(a) shows each individual floating point application in the Sandia and SPEC suites. On the basis of spatial and temporal locality measurements alone, the the 177.mesa SPEC FP benchmark would appear to dominate all others in the suite. However, it has the second smallest unique data set size in the entire SPEC suite. In fact, the Sandia FP applications average **over** 9 times the data intensiveness of 177.mesa. There are numerous very small data set applications in SPEC FP, including 177.mesa, 178.galgel, 179.art, 187.facerec, 188.ammp, and 200.sixtrack. In fact, virtually all of the applications from SPEC FP that are "close" to a Sandia application in terms of spatial and temporal locality exhibit a much smaller unique data set. The mpsalsa application from the Sandia suite and 183.equake are good examples. While they are quite near on the graph, mpsalsa has almost 17 times the unique data set of equake. 183.equake is also very near the mean spatial and temporal locality point for the entire Sandia FP suite, except that the Sandia applications average more than 15 times 183.equake's data set size.

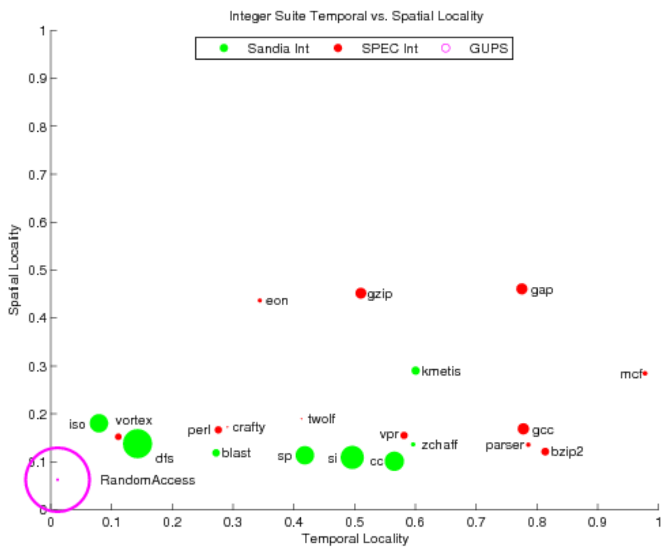
Unfortunately, it would be extremely difficult to identify a SPEC FP application that is "representative" of the Sandia codes (either individually, or on average). Often papers choose a subset of a given benchmark suite's applications when presenting the results. Choosing the five applications in SPEC FP with the largest data intensiveness (168.wupwise, 171.swim, 173.applu, 189.lucas, 301.apsi), and 183.equake (because of its closeness to the average and to mpsalsa) yields a suite that averages 90% of the Sandia suite's temporal locality, 86% of its temporal locality, 75% of it's data intensiveness. While somewhat far from "representative", particularly in terms of data intensiveness, this subset is more representative of the real applications than the whole.

Several interesting Sandia applications are shown on the graph. The CTH application exhibits the most temporal locality, but relatively low spatial locality, and a relatively small data set size. The LAMMPS (lmp) molecular dynamics code is known to be compute intensive, but it exhibits a relatively small memory footprint, and shows good memory performance. The temporal and spatial locality measures are quite low. SPPM exhibits very high spatial locality, very low temporal locality, and a moderate data set size.

Figure 4(b) depicts the Sandia and SPEC Integer benchmark suites. These applications are strikingly more different than the floating point suite. All of the applications exhibit relatively low spatial locality, although the majority of Sandia applications exhibit significantly less spatial locality than their SPEC counterparts. The DFS code in the Sandia suite is the most "RandomAccess-like", with 255.vortex in the SPEC suite being the closest counter part in terms of spatial and temporal locality. 255.vortex's temporal and spatial locality are within 25% and 15% of DFS' respectively. However, once again, DFS's data set size is over 19 times that of 255.vortex's.



(a)



(b)

Fig. 4. (a) Integer and (b) Floating Point Applications Temporal vs. Spatial Locality and Data Intensiveness.

300.twolf actually comes closest in terms of spatial and temporal locality to representing the “average” Sandia Integer application, however, the average Sandia code has nearly 140 times the data set size.

VII. Conclusions

This work has measured the temporal and spatial locality, and the relative data intensiveness of a set of real world Sandia applications, and compared them to the SPEC Integer and Floating Point suites, as well as the RandomAccess memory benchmark. While the SPEC floating point suite exhibits greater temporal locality and less spatial locality than the Sandia floating point suite, it averages significantly less data intensiveness. This is crucial because the number of unique items consumed by the application can affect the performance of hierarchical memory systems more than the average efficiency with which those items are stored in the hierarchy.

The integer suites showed even greater divergence in all three dimensions (temporal locality, spatial locality, and data intensiveness). Many of the key integer benchmarks, which represent applications of emerging importance, are close to RandomAccess in their behavior.

This work has further quantitatively demonstrated the difference between a set of real applications (both current and emerging) relevant to the high performance computing community, and the most studied set of benchmarks in computer architecture. The real integer codes are uniformly harder on the memory system than the SPEC integer suite. In the case of floating point codes, the Sandia applications exhibit a significantly larger data intensiveness, and lower temporal locality. Because of the dominance of the memory system in achieving performance, this indicates that architects should focus on codes with significantly larger data set sizes.

The emerging applications characterized by the Sandia Integer suite are the most challenging applications (next to the RandomAccess benchmark). Because of their importance and their demands on the memory system, they represent a core group of applications that require significant attention.

Finally, beyond a specific study of one application domain, this work presents an architecture-independent methodology for quantifying the difference in memory properties between any two applications (or suites of applications). This study can be repeated for other problem domains of interest (the desktop, multimedia, business, etc.).

Acknowledgments

The authors would like to thank Arun Rodrigues and Keith Underwood at Sandia National Laboratories for their valuable comments.

The application analysis and data gathering discussed here used tools that were funded in part by DARPA through Cray, Inc. under the HPCS Phase 1 program.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] Apple Architecture Performance Groups. *Computer Hardware Understanding Development Tools 2.0 Reference Guide for MacOS X*. Apple Computer Inc, July 2002.

[3] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 3–14. IEEE Computer Society, 1998.

[4] Daniel Citron, John Hennessy, David Patterson, and Gurindar Sohi. The use and abuse of SPEC: An ISCA panel. *IEEE Mico*, 23(4):73–77, July-August 2003.

[5] P. Colella and P.R. Woodward. The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations. *Journal of Computational Physics*, 54:174–201, 1984.

[6] Z. Cvetanovic and D. Bhandarkar. Characterization of alpha AXP performance using TP and SPEC workloads. In *Proceedings of the 21ST annual international symposium on Computer architecture*, pages 60–70. IEEE Computer Society Press, 1994.

[7] Peter J. Denning. The working set model for program behavior. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 15.1–15.12. ACM Press, 1967.

[8] J. Dongarra and P. Luszczek. Introduction to the HPCChallenge Benchmark Suite. Technical Report ICL-UT-05-01, 2005.

[9] Domenico Ferrari. A generative model of working set dynamics. In *Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 52–57. ACM Press, 1981.

[10] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache performance of the SPEC benchmark suite. Technical Report CS-TR-1991-1049, 1991.

[11] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *International Conference on Supercomputing*, pages 317–324, 1997.

[12] Swathi Tanjore Gurumani and Aleksandar Milenkovic. Execution characteristics of SPEC CPU2000 benchmarks: Intel C++ vs. Microsoft VC++. In *Proceedings of the 42nd annual Southeast regional conference*, pages 261–266. ACM Press, 2004.

[13] B. Hendrickson and R. Leland. The chaco user's guide — version 2.0. Technical Report SAND94-2692, 1994.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, 2002.

[15] Michael Heroux, Roscoe Bartlett, Vicki Howle, Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, 2003.

[16] E. Hertel, J. Bell, M. Elrick, A. Farnsworth, G. Kerley, J. McGlaun, S. Petney, S. Silling, P. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings of the 19th International Symposium on Shock Waves*, pages 377–382, July 1993.

[17] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[18] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the International Sympoisum on Computer Architecture*, pages 15–26, 1998.

[19] Dennis C. Lee, Patrick Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on windows NT. In *International Symposium on Computer Architecture*, pages 27–38, 1998.

[20] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *International Symposium on Computer Architecture*, pages 39–50, 1998.

[21] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the Sixth International Conference on Architectural support for Programming Languages and Operating Systems*, pages 145–156. ACM Press, 1994.

[22] McCalpin, John D. *Stream: Sustainable memory bandwidth in high performance computers*, 1997.

[23] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference*, June 2001.

[24] Richard C. Murphy. *Traveling Threads: A New Multithreaded Execution Model*. Ph.D. Dissertation, University of Notre Dame, May 2006.

[25] Richard C. Murphy, Arun Rodrigues, Peter Kogge, and Keith Underwood. The Implications of Working Set Analysis on Supercomputing Memory Hierarchy Design. In *Proceedings of the 2005 International Conference on Supercomputing*, pages 332–340, June 20-22, 2005.

[26] Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, and Stephane Ethier. Scientific Computations on Modern Parallel Vector Systems. In *Proceedings of Supercomputing*, page 10, 2004.

[27] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, 2ed.* Morgan Kaufmann Publishers, 1997.

[28] Steven J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[29] Steven J. Plimpton, R. Pollock, and M. Stevens. Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[30] Arun Rodrigues, Richard Murphy, Peter Kogge, and Keith Underwood. Characterizing a New Class of Threads in Scientific Applications for High End Supercomputers. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing*, pages 164–174.

[31] Juan Rodriguez-Rosell. Empirical working set behavior. *Communications of the ACM*, 16(9):556–560, 1973.

[32] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 14–26. ACM Press, 1993.

[33] Rafael Saavedra and Alan Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–84, 1996.

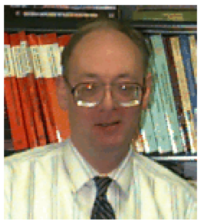
[34] J. Shadid, A. Salinger, R. Schmidt, T. Smith, S. Hutchinson, G. Hennigan, K. Devine, and H. Moffat. MPSalsa: A Finite Element Computer Program for Reacting Flow Problems. Technical Report SAND98-2864, 1998.

[35] D. R. Slutz and I. L. Traiger. A note on the calculation of average working set size. *Communications of the ACM*, 17(10):563–565, 1974.

[36] Jeffrey S. Vetter and Andy Yoo. An Empirical Performance Evaluation of Scalable Scientific Applications. In *Proceedings of Supercomputing*, pages 1–18, 2002.

[37] Jonathan Weinberg, Michael McCracken, Alan Snavely, and Erich Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005*, page 50, November, 2005.

Richard C. Murphy is a computer architect in the scalable systems group at Sandia National Laboratories. He received his Ph.D. in computer engineering at the University of Notre Dame. His research interests include computer architecture, with a focus on memory systems and Processing-In-Memory, VLSI, and massively parallel architectures, programming languages, and runtime systems. He spent 2000 to 2002 at Sun Microsystems focusing on hardware resource management and dynamic configuration. He is a member of the IEEE.



Peter M. Kogge was with IBM, Federal Systems Division, from 1968 until 1994, and was appointed an IEEE Fellow in 1990, and an IBM Fellow in 1993. In 1977 he was a Visiting Professor in the ECE Dept. at the University of Massachusetts, Amherst. From 1977 through 1994, he was also an Adjunct Professor in the Computer Science Dept. of the State University of New York at Binghamton. In August, 1994 he joined the University of Notre Dame as first holder of the endowed McCourtney Chair in Computer Science and Engineering (CSE). Starting in the summer of 1997, he has been a Distinguished Visiting Scientist at the Center for Integrated Space Microsystems at JPL. He is also the Research Thrust Leader for Architecture in Notre Dame's Center for Nano-Science and Technology. For the 2000-2001 academic year he was the Interim Schubmehl-Prein Chairman of the CSE Dept. at Notre Dame. Starting in August, 2001 he is the Associate Dean for Research, College of Engineering. Starting in the fall of 2003, he also is a Concurrent Professor of Electrical Engineering. He is a fellow of the IEEE.

On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance

Richard Murphy
Sandia National Laboratories[*]
PO Box 5800, MS-1110
Albuquerque, NM 87185-1110
rcmurph@sandia.gov

Abstract

Since the first vector supercomputers in the mid-1970's, the largest scale applications have traditionally been floating point oriented numerical codes, which can be broadly characterized as the simulation of physics on a computer. Supercomputer architectures have evolved to meet the needs of those applications. Specifically, the computational work of the application tends to be floating point oriented, and the decomposition of the problem two or three dimensional. Today, an emerging class of critical applications may change those assumptions: they are combinatorial in nature, integer oriented, and irregular. The performance of both classes of applications is dominated by the performance of the memory system. This paper compares the memory performance sensitivity of both traditional and emerging HPC applications, and shows that the new codes are significantly more sensitive to memory latency and bandwidth than their traditional counterparts. Additionally, these codes exhibit lower base-line performance, which only exacerbates the problem. As a result, the construction of future supercomputer architectures to support these applications will most likely be different from those used to support traditional codes. Quantitatively understanding the difference between the two workloads will form the basis for future design choices.

1. Introduction and Motivation

Supercomputing is in the midst of large technological, architectural, and application changes that greatly impact the way designers and programmers think about the system. Technologically, the constraints of power and the speed of light dictate that multicore architectures will form the basis for the commodity processors that constitute the heart of massively parallel processing (MPP) supercomputers. This impacts the architecture in three ways:

1. Power constraints dictate that clock rates will not improve appreciably as they have in the past.
2. The combination of power, the constraint of the speed of light, and the architectural limits of instruction level parallelism dictate that the trend in scalar processors towards higher performing individual cores will not hold.
3. Even though die area is increasing, cost is still dictated by packaging, and the number of pins available for external communication will likely not grow as quickly as the number of cores.

These technological and architectural trends are no less significant than the those which dictated the transition from vector-based supercomputers to commodity MPP architectures in the early 1990's. In fact, given the maturity of vector architectures, such a change was likely inevitable. One critical architectural trend still holds across any implementation: the challenge posed by the memory wall. In today's supercomputers, the memory wall manifests itself as a dramatic difference between the increase in processor clock rate and the rate of a memory access. In multicore machines – even with flat clock rates – the memory wall manifests itself due to the increasing number of cores compared to available *channels* (or independent access paths) to memory.

Unlike prior large-scale technological and architectural changes, today's architects must also contend with a shift in the application base. Historically, supercomputing has been dominated by the simulation of physics on a computer, which itself can be thought of as fundamentally structured in nature. In a three dimensional universe, problem decomposition can be performed in three dimensions (by dividing the simulated area into cubes). The types of supercom-

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

puter architectures that have been adopted often reflect this structure. For example, 3d mesh topologies reflect the kind of “nearest neighbors” communication pattern common in physics codes. Furthermore, the “work” performed in these applications is fundamentally floating point: the computation of temperature, pressure, volume, etc. Many emerging applications, however, are different. They are combinatorial in nature, fundamentally unstructured, and often consist of integer computations.

This paper examines the memory performance of a suite of **real world** applications from both the traditional and emerging problem domains. It examines the impact of memory latency and bandwidth on the applications. The results demonstrate that both sets of applications are fundamentally dominated by memory latency, but that the emerging applications both begin with a lower baseline performance, and are more sensitive to memory than their traditional counterparts. This represents a significant challenge to supercomputer architects, and quantitatively establishes how emerging applications differ from their traditional counterparts.

The remainder of this paper is organized as follows: Section 2 discusses the related work. Section 3 examines the applications under study. Section 4 specifies the methodology and metrics used for evaluation. Section 5 presents the results Conclusions are given in Section 6.

2. Related Work

Characterizing performance is a richly studied area of computer architecture. The SPEC suite has been extensively characterized[10, 8, 12, 14, 26], as have other workloads such as OLTP[13, 4, 2]. These codes are generally chosen to represent “typical” machine workloads for a class of applications.

In the area of supercomputing, specialized benchmarks have been constructed to test specific areas of machine performance. The HPC Challenge RandomAccess benchmark[6] and the STREAM benchmark[15] have been specifically constructed to measure the performance of the memory system. Although this information is useful to architects, it is difficult to directly map back to application performance.

The applications in this work have been studied extensively[19, 18, 20]. The floating point suite is similar in structure to other real world supercomputer applications that have been previously examine[21, 28].

Numerous definitions of spatial and temporal locality have been proffered to characterize the memory performance of applications, both canonical [22, 9] and experimental[19, 29, 5, 7, 27, 25]. This study examines the performance impact of the memory system on applications.

The memory wall is also an extremely well studied problem in computer architecture[30, 16]. It has been argued that this is the dominant problem in computer architecture[11]. Indeed, the results of this work further support this conclusion by affirming that key supercomputer applications are memory latency dominated in performance, which is the classic definition of the memory wall.

3. Applications

This study examines two classes of important applications from Sandia National Laboratories: a traditional set of primarily floating point codes, and an emerging class of primarily integer codes. These codes have been discussed extensively previously, and are significantly different from traditional suites such as SPEC CPU[20, 19]. Their description and basic instruction mix follow.

3.1. Traditional Floating Point Codes

Each of the floating point applications are production MPI codes designed to run at very large scale. Broadly speaking they are scientific and engineering applications which represent physical simulations. They are:

- **ALEGRA** is a finite element shock physics code capable of modeling near- and far-field responses to explosions, impacts, and energy depositions.
- **CTH** is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories over the last 30 years. CTH models multi-phase, elastic viscoplastic, porous and explosive materials with multiple mesh refinement methods.
- **Cube3** Cube3 is a generic linear solver that drives the Trilinos framework for parallel linear and eigensolvers. It mimics a finite element analysis problem by creating hexagonal elements, then assembling and solving a linear system. The width, depth, and degrees of freedom (e.g., temperature, pressure, velocity, etc.) can be varied.
- **ITS** performs Monte Carlo simulations of linear time-independent coupled electron/photon radiation transport.
- **MPSalsa** is a high resolution 3d simulation of reacting flow. The simulation requires both fluid flow and chemical kinetics modeling.
- **Xyce** is a parallel circuit simulation system capable of modeling very large circuits at multiple layers of abstraction (device, analog, digital, and mixed-signal). It includes both SPICE-like models and radiation models.

3.2. Emerging Integer Applications

Unlike their traditional counterparts, the emerging applications studied in this work are integer-oriented, generally problems from Discrete Math, typically unstructured, and written for a variety of programming models. They are at the core of important problems in proteomics, genomics, data mining, pattern matching, and computational geometry. Although many are classic algorithms problems (DFS, for example), the implementations are typically "newer" than their traditional counterparts (that is, there are no three decade old codes among these applications).

- **DFS** implements a depth-first search on a large graph and forms the basis for several high-level algorithms including connected components, tree and cycle detection, solving the two-coloring problem, finding Articulation Vertices, and topological sorting.
- **Connected Components** breaks a graph into components. Two vertices are in a connected component if and only if there is a path between them.
- **Subgraph Isomorphism** determines whether or not a subgraph exists within a larger graph.
- **Full Graph Isomorphism** determines whether or not two graphs have the same shape or structure.
- **Shortest Path** computes the shortest path between two vertices using a breadth first search. Real world applications include path planning and networking and communication.
- **Graph Partitioning** is used extensively in VLSI circuit design and adaptive mesh refinement. The problem divides a graph in to k partitions while minimizing the *cut* between the partitions (or the total weight of the edges that cross from one partition to another).
- **BLAST** is the Basic Local Alignment Search Tool and is the most heavily used method for quickly search nucleotide and protein databases in biology.
- **zChaff** is a heuristic for solving the Boolean Satisfiability Problem. In propositional logic, a formula is *satisfiable* if there exists an assignment of truth values to each of its variables that make the formula true.

3.3. Instruction Mix for Each Suite

Figure 1 shows the instruction mix for the integer and floating point suites. Although the "work" for the floating point suite is primarily floating point, real applications perform significantly less floating point than do typical benchmark suites. For example, the SPEC CPU 2000 suite averages 32% floating point, while real Sandia codes average

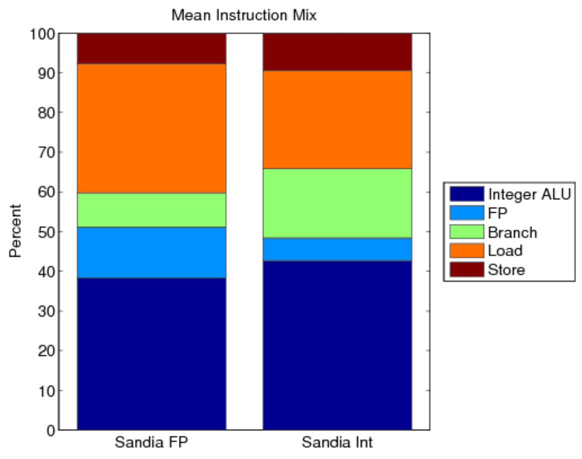


Figure 1. Sandia Integer and Floating Point Suite Instruction Mix

only about 12% floating point[19]. Other key differences between the integer and floating point codes are apparent. The integer codes perform 15% fewer memory references (although those references are much more likely to miss the cache, as Section 5 will demonstrate). Furthermore, the integer codes perform twice as many branches as their floating point counterparts. This is unsurprising since scientific codes tend to have larger basic blocks due to complex formula calculations[23].

The combination of an increased cache miss rate and an increased number of branches makes the integer codes challenging for modern superscalar processors.

4. Methodology and Metrics

This section discusses the application traces used in this work, the metrics used for evaluation, and the simulation environment. The simulation methodology is similar to that employed in prior studies of this application base, although the metrics are entirely new.

4.1. Application Traces

Each of the applications used in this work was analyzed using traces of 4 billion sequential instructions produced by the Amber[1] instruction trace generator for the PowerPC. These traces have been used in several prior studies[19, 18, 20, 23, 17, 24] and are well understood. The traces typically represent multiple executions of the main loop of the program and were originally generated with the input from applications experts and platform profiling tools. In the case of the MPI programs, traces were of single node execution.

Table 1. Processor Configuration

Parameter	**Val**
Issue Width	8
Commit Width	4
RUU Size	64
L1 Instruction/Data Cache	64k
	2-way Set Associative
	64-byte block
	Least Recently Used
L1 Cache Latency	3 cycles
L2 Unified Cache	1 MB
	16-way Set Associative
	64-byte block
	Least Recently Used
L2 Cache Latency	20 cycles
Integer ALUs	3
Integer Multiplier/Dividers	1
FP ALUs	2
FP Multiplier/Divider	1
Clock Rate	2.5 GHz

4.2. Metrics

This work defines bandwidth and latency as follows:

- **Latency:** the time between when the processor requests a memory value and when the first byte of that request arrives.
- **Bandwidth:** the transfer speed of the second and all subsequent bytes of a memory request.

The simulations in this work vary the memory bandwidth and latency according to this definition for each run.

4.3. Simulation Environment

The traces were used as inputs to Sandia's Structural Simulation Toolkit (SST). The SST is a parallel machine simulator (for both shared and distributed memory architectures) that uses an enhanced version of SimpleScalar's `sim-outorder` processor simulator[3] as the baseline processor. SST has significantly enhanced cache and memory models, and has been used to simulate several supercomputer architectures.

The memory simulated in this work is a DDR-like interface which performs transfers in 16-byte blocks.

Table 1 summarizes the conventional superscalar processor configuration used in this study. The memory latency and bandwidth were varied and the committed Instructions Per Cycle (IPC) measured.

The memory latencies examined were: 15ns, 30ns, 60ns, 120ns, and 240ns.

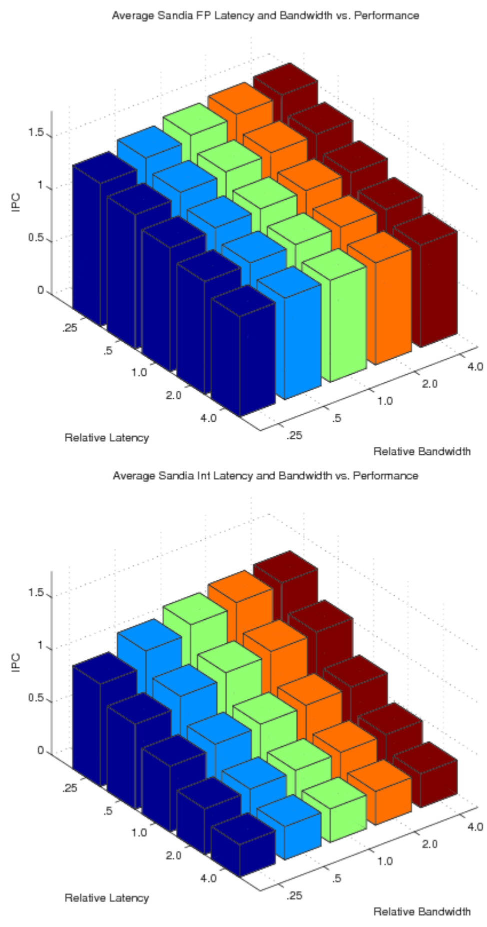


Figure 2. Average Latency and Bandwidth Effects for the Sandia Floating Point and Integer Suites

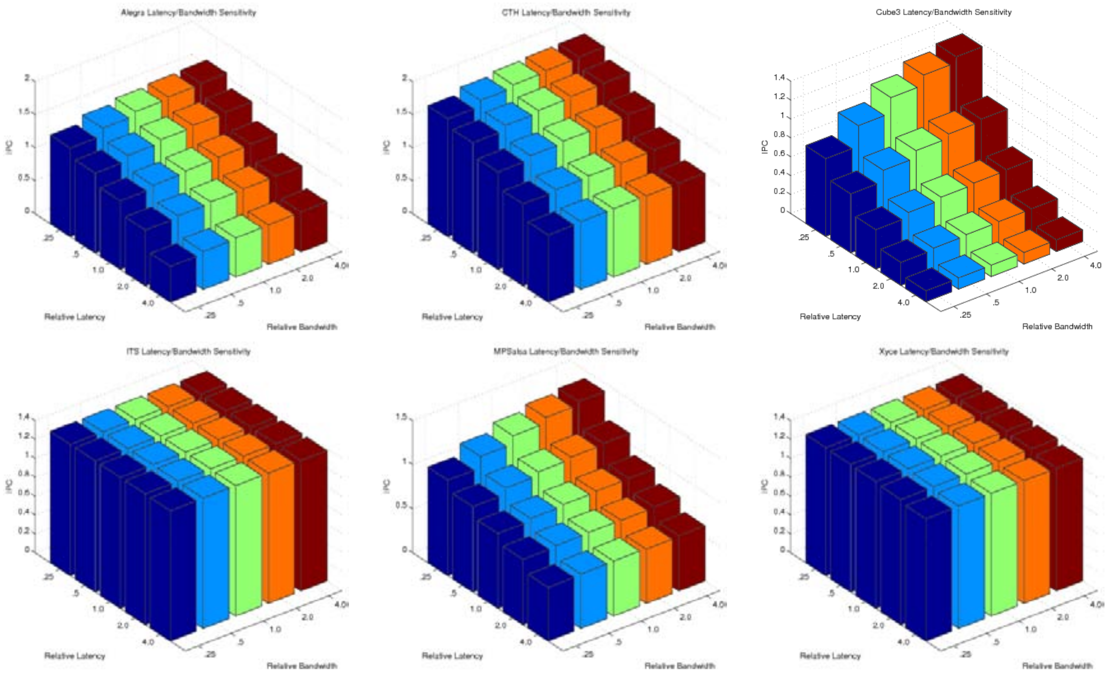
The bandwidths in this experiment were: 2.5 GB/sec, 5 GB/sec, 10 GB/sec, 20 GB/sec, and 40 GB/sec.

The baseline memory latency and bandwidth numbers in this study look somewhat more aggressive than a modern Opteron (60ns latency and 10 GB/sec of bandwidth). However, several points around that baseline were examined, and can be used for comparison given the reader's assumptions about what is appropriate. Those points were generated by halving and doubling each configuration parameter (latency and bandwidth) twice.

5. Results

Figure 2 shows the average effect of varying latency and bandwidth (60ns of memory latency and 10 GB/sec of memory bandwidth). The center point for each graph (relative latency and bandwidth of 1.0) is this baseline, and each bar represents the IPC achieved at that latency/bandwidth point.

Floating Point Applications



Integer Applications

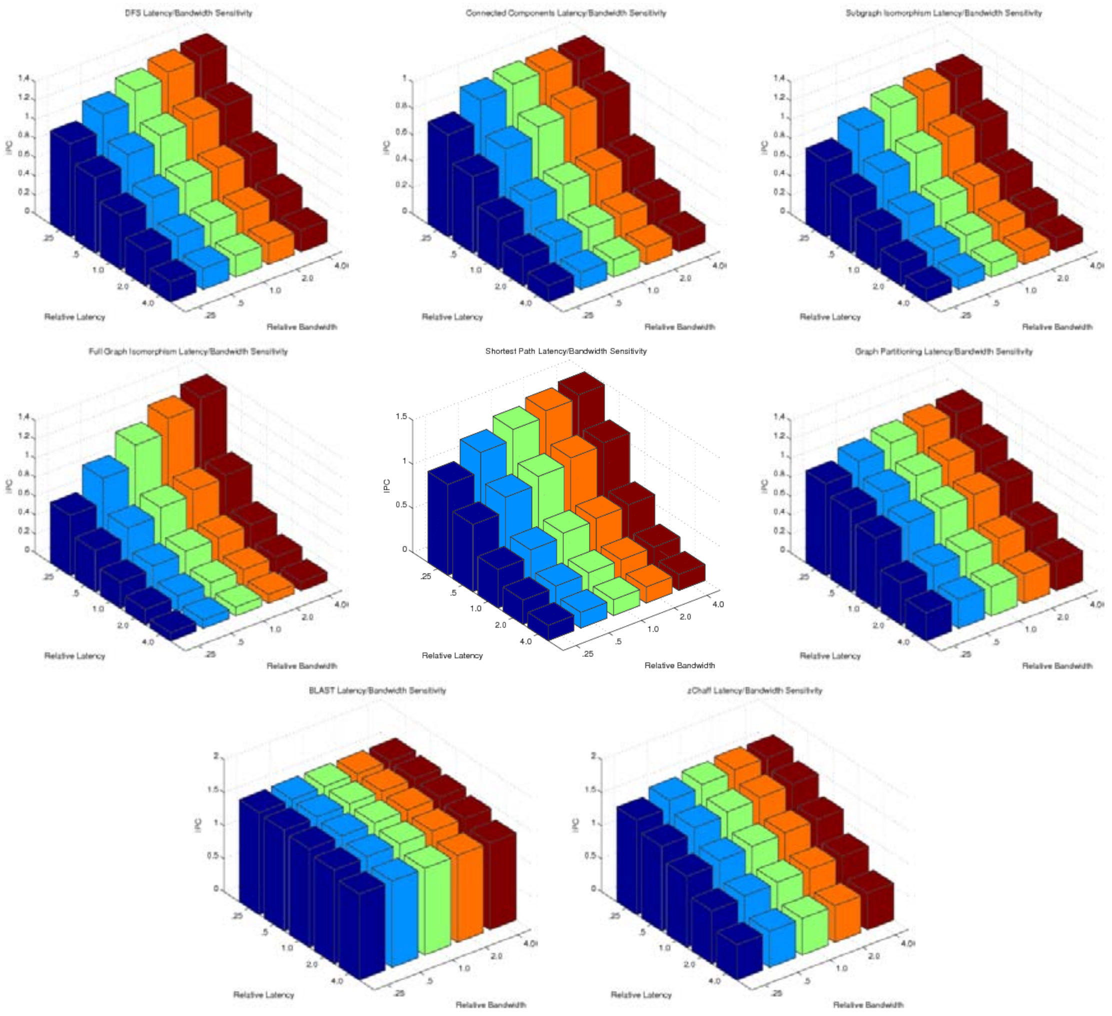


Figure 3. Complete Results for the Integer and Floating Point Suites

Figure 3 depicts the latency/bandwidth results for each application in both the integer and floating point suites.

On average, in the floating point case, halving the available bandwidth results in an average drop in performance of 1.24%. In the case of the integer suite, the average drop in performance is 3.59%. By contrast, doubling the memory latency leads to a respective drop in performance of 11% and 32% respectively. Thus, all of the codes are more latency than bandwidth dominated. This is a critical point for two reasons:

1. Memory bandwidth is the typical unit of memory performance measure discussed by supercomputer architects. This may be because the construction of MPP-based supercomputers is dominated by the construction of a high performance network interface, and MPP system architects therefore tend to think in more networking oriented terms. It may also be because bandwidth is an easier system characteristic to affect than latency. Regardless, the more critical unit of measurement is quite conclusively latency.

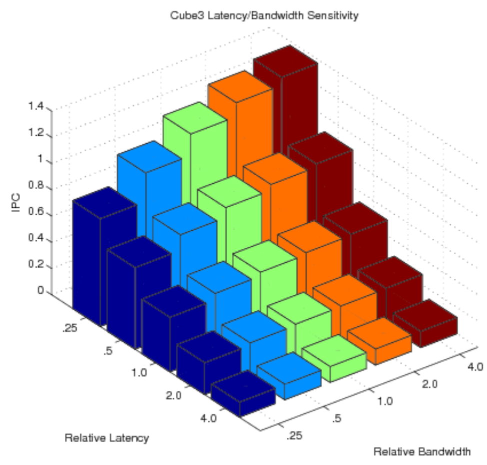
2. As discussed in Section 1, one of the key concerns for supercomputer architects is the transition to multicore machines. If today's instruction streams in a unicore processor were memory bandwidth bound, and available bandwidth did not grow with the number of cores, performance would suffer. However, these results indicate that there is potentially headroom in bandwidth.

Additionally, the structure of these applications generally makes it very difficult for the processor to compute memory addresses quickly enough to keep the memory bus busy. This tends to make bandwidth less important than latency.

The integer codes are clearly more sensitive to memory performance than their traditional floating point counterparts (doubling the latency or halving the bandwidth has $2.9\times$ the impact on the integer suite as compared to the floating point suite).

It is also critical to note that the baseline performance of each suite is significantly different. The floating point suite has a baseline IPC of 1.22, while the integer suite has a baseline performance of 0.70. Thus, the integer suite baselines with 43% less performance than the floating point suite, and is significantly more sensitive to latency and bandwidth variations after that.

Figure 4 shows the most affected applications for the floating point and integer suites. Cube3 from the floating point suite and Shortest Path from the integer suite look remarkably similar. Cube3 has a baseline performance only 10% lower than Shortest Path, and both applications experience a 55% drop in performance if the memory latency is doubled. Cube3 experiences a 6% drop in performance if the bandwidth is halved while Shortest Path experiences a 7% drop in performance. This is not surprising since sparse graphs can be represented as sparse matrices, similar to those that are fundamental to linear algebra problems. The sparse matrix representation is not used in this particular instance of the graph problem, however the fundamental nature of the data structures used are similarly sparse.



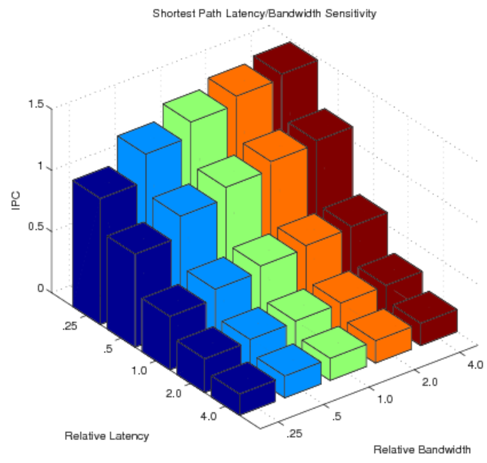


Figure 4. The Most Sensitive Applications From Each Suite

The least affected applications from each suite (Xyce from the floating point suite and BLAST from the integer suite) show nearly flat latency/bandwidth curves. Doubling the memory latency shows less than a 1.25% performance degradation for Xyce and less than a 5% performance degradation for BLAST. Similarly, halving the bandwidth yields less than half a percent performance degradation. These applications are known to generally be more compute intensive, and this result confirms that prior knowledge. These are the only applications with relatively flat curves.

Cube3 and Alegra are the most sensitive to latency or bandwidth changes of the floating point suite, while connected components and shortest path dominate the integer suite.

To better quantify each suite's sensitivity to bandwidth and latency effects, the *sensitivity* must be defined. Intuitively, the sensitivity can be thought of as the slope of either the latency or bandwidth lines. As a first-order approximation, these curves can be thought of as linear (they are very nearly so). For a given latency or bandwidth, the slope of the resulting 2-dimensional slice of Figure 2 (either latency vs. IPC or bandwidth vs. IPC) can be calculated. For example, given the baseline latency of 60ns, the bandwidth sensitivity (or slope of the bandwidth line) can be calculated as follows:

$$\frac{P_{60ns,2.5GB/sec} - P_{60ns,40GB/sec}}{15ns - 240ns} \tag{1}$$

Where $P_{latency,bandwidth}$ represents the performance at a given latency and bandwidth. Similarly, given the baseline bandwidth of $10GB/sec$, the latency sensitivity can be computed by:

$$\frac{P_{15ns,10GB/sec} - P_{240ns,10GB/sec}}{2.5GB/sec - 40GB/sec} \tag{2}$$

The sensitivity to bandwidth is a positive number because an increase in bandwidth leads to an increase in performance, while the sensitivity to latency is a negative number because an increase in latency leads to a decrease in performance.

Figure 5(a) shows the average sensitivity to latency and bandwidth for both suites, while (b) includes each individual application for the floating point suite, and (c) includes the same information for the integer suite. As discussed earlier, the integer suite is clearly more sensitive to both bandwidth and latency than the floating point suite. The integer suite is $42 - 60\%$ more sensitive to latency than the floating point suite. While it begins 66% more sensitive to bandwidth than the floating point suite, at very high memory latencies the integer suite is nearly 30% less sensitive to bandwidth than the floating point suite.

6. Conclusions and Future Work

This paper has examined the impact of memory latency and bandwidth on a set of traditional floating-point oriented and emerging integer oriented supercomputer applications. The results demonstrate that both application suites are more dominated by latency than bandwidth. It has further shown that the emerging integer applications are $2.9\times$ more sensitive to a halving of the memory bandwidth or doubling of the memory latency than their traditional counterparts, which is critical to understanding the nature of these emerging codes.

This result has two critical impacts to the field of supercomputer architecture: first, it demonstrates that there is some degree of "bandwidth headroom" in the construction of multicore supercomputers; and second, it quantitatively shows that emerging applications are much more memory sensitive than traditional scientific computing codes. This provides further evidence in support of the long-held belief that the lessons of scientific computing have little applicability to these applications, particularly as they relate to data decomposition.

Because the transition to multicore MPPs will impact the memory system in future machines, understanding the on-node memory performance of real applications is critical. The technology determines the number of available independent channels into memory (which affects aggregate latency by constraining the number of simultaneous memory accesses the memory system can sustain), and the speed and width of those channels (which affects bandwidth). Both are likely to be more constrained than the number of cores that can placed on a die. Choosing the right balance for supercomputer applications will depend on characterizing the requirements of the workloads of those machines. This study does so, and shows that this is an even more significant problem for emerging codes than for classical supercomputing applications.

Finally, this study identifies four critical applications (two from each suite) that demonstrate the most sensitivity to latency and bandwidth. Cube3 and Alegra from the floating point suite and Connected Components and Shortest Path from the integer suite. As a whole, problems in graph theory are shown to be particularly challenging to the memory system.

Future work will extend this study to examine the impact of moving to multicore architectures with simpler cores.

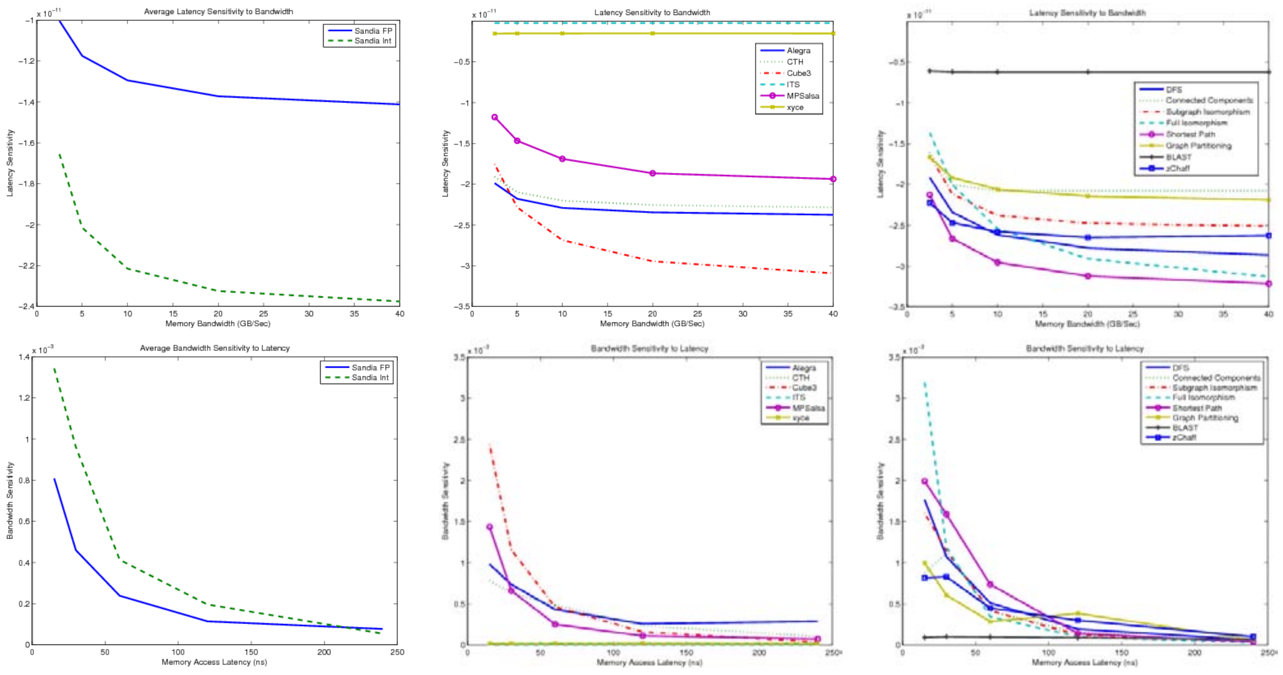
References

[1] Apple Architecture Performance Groups. *Computer Hardware Understanding Development Tools 2.0 Reference Guide for MacOS X.* Apple Computer Inc, July 2002.

[2] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 3–14. IEEE Computer Society, 1998.

[3] Doug Burger and Todd Austin. *The SimpleScalar Tool Set, Version 2.0.* SimpleScalar LLC.

[4] Z. Cvetanovic and D. Bhandarkar. Characterization of alpha AXP performance using TP and SPEC workloads. In *Proceedings of the 21ST annual international symposium on Computer architecture*, pages 60–70. IEEE Computer Society Press, 1994.



(a) Suite Mean **(b) Floating Point** **(c) Integer**

Figure 5. Latency and Bandwidth Sensitivity. It should be noted that bandwidth sensitivity is positive because increasing bandwidth increases performance while latency sensitivity is negative because increasing latency decreases performance. Part (a) of the figure depicts the average for each of the benchmark suites, while (b) shows the floating point applications and (c) the integer.

[5] Peter J. Denning. The working set model for program behavior. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 15.1–15.12. ACM Press, 1967.

[6] J. Dongarra and P. Luszczek. Introduction to the HPC-Challenge Benchmark Suite. Technical Report ICL-UT-05-01, 2005.

[7] Domenico Ferrari. A generative model of working set dynamics. In *Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 52–57. ACM Press, 1981.

[8] Swathi Tanjore Gurumani and Aleksandar Milenkovic. Execution characteristics of SPEC CPU2000 benchmarks: Intel C++ vs. Microsoft VC++. In *Proceedings of the 42nd annual Southeast regional conference*, pages 261–266. ACM Press, 2004.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, 2002.

[10] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the International Sympoisum on Computer Architecture*, pages 15–26, 1998.

[11] Peter M. Kogge, Jay B. Brockman, and Vincent Freeh. Processing-In-Memory Based Systems: Performance Evaluation Considerations. In *Workshop on Performance Analysis and its Impact on Design held in conjunction with ISCA, Barcelona, Spain*, June 27-28, 1998.

[12] Dennis C. Lee, Patrick Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on windows NT. In *International Symposium on Computer Architecture*, pages 27–38, 1998.

[13] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In

International Symposium on Computer Architecture, pages 39–50, 1998.

[14] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the Sixth International Conference on Architectural support for Programming Languages and Operating Systems*, pages 145–156. ACM Press, 1994.

[15] McCalpin, John D. *Stream: Sustainable memory bandwidth in high performance computers*, 1997.

[16] Sally A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, page 162, New York, NY, USA, 2004. ACM Press.

[17] Richard C. Murphy. *Traveling Threads: A New Multithreaded Execution Model*. Ph.D. Dissertation, University of Notre Dame, May 2006.

[18] Richard C. Murphy, Jonathan Berry, William McLendon, Bruce Hendrickson, Douglas Gregor, and Andrew Lumsdaine. DFS: A Simple to Write Yet Difficult to Execute Benchmark. In *IEEE International Symposium on Workload Characterization 2006 (IISWC06)*, October 25-27, 2006.

[19] Richard C. Murphy and Peter M. Kogge. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and its Implications. *To Appear In IEEE Transactions on Computers*.

[20] Richard C. Murphy, Arun Rodrigues, Peter Kogge, and Keith Underwood. The implications of working set analysis on supercomputing memory hierarchy design. In *The 2005 International Conference on Supercomputing*, June 20-22, 2005.

[21] Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, and Stephane Ethier. Scientific Computations on Modern Parallel Vector Systems. In *Proceedings of Supercomputing*, page 10, 2004.

[22] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, 2ed.* Morgan Kaufmann Publishers, 1997.

[23] Arun Rodrigues, Richard Murphy, Peter Kogge, and Keith Underwood. Characterizing a New Class of Threads in Scientific Applications for High End Supercomputers. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing*, pages 164–174, June 26-July 1 2004.

[24] Arun F. Rodrigues. *Programming Future Architectures: Dusty Decks, Memory Walls, and the Speed of Light*. Ph.D. Dissertation, University of Notre Dame, 2006.

[25] Juan Rodriguez-Rosell. Empirical working set behavior. *Communications of the ACM*, 16(9):556–560, 1973.

[26] Rafael Saavedra and Alan Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–84, 1996.

[27] D. R. Slutz and I. L. Traiger. A note on the calculation of average working set size. *Communications of the ACM*, 17(10):563–565, 1974.

[28] Jeffrey S. Vetter and Andy Yoo. An Empirical Performance Evaluation of Scalable Scientific Applications. In *Proceedings of Supercomputing*, pages 1–18, 2002.

[29] Jonathan Weinberg, Michael McCracken, Alan Snavely, and Erich Strohmaier. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Supercomputing 2005*, page 50, November, 2005.

[30] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

Qthreads: An API for Programming with Millions of Lightweight Threads

Kyle B. Wheeler
University of Notre Dame
South Bend, Indiana, USA
kwheeler@cse.nd.edu

Richard C. Murphy
Sandia National Laboratories*
Albuquerque, New Mexico, USA
rcmurphy@sandia.gov

Douglas Thain
University of Notre Dame
South Bend, Indiana, USA
dthain@cse.nd.edu

Abstract

Large scale hardware-supported multithreading, an attractive means of increasing computational power, benefits significantly from low per-thread costs. Hardware support for lightweight threads is a developing area of research. Each architecture with such support provides a unique interface, hindering development for them and comparisons between them. A portable abstraction that provides basic lightweight thread control and synchronization primitives is needed. Such an abstraction would assist in exploring both the architectural needs of large scale threading and the semantic power of existing languages. Managing thread resources is a problem that must be addressed if massive parallelism is to be popularized. The qthread abstraction enables development of large-scale multithreading applications on commodity architectures. This paper introduces the qthread API and its Unix implementation, discusses resource management, and presents performance results from the HPCCG benchmark.

1. Introduction

Lightweight threading primitives, crucial to large scale multithreading, are typically either platform dependent or compiler-dependent. Generic programmer-visible multithreading interfaces, such as pthreads, were designed for "reasonable" numbers of threads—less than one hundred or so. In large-scale multithreading situations, the features and guarantees provided by these interfaces prevent them from scaling to "unreasonable" numbers of threads (a million or more), necessary for multithreaded teraflop-scale problems.

Parallel execution has largely existed two different worlds: the world of the very large, where programmers explicitly create parallel threads of execution, and the world of the very small, where processors extract parallelism from serial instruction streams. Recent hardware architectural research has investigated lightweight threading and programmer-defined large scale shared-memory parallelism. The lightweight threading concept allows exposure of greater potential parallelism, increasing performance via greater hardware parallelism. The Cray XMT [7], with the Threadstorm CPU architecture, avoids memory dependency stalls by switching among 128 concurrent threads. XMT systems support between over 8000 processors. To maximize throughput, the programmer must provide at least 128 threads per processor, or over 1,024,000 threads.

Taking advantage of large-scale parallel systems with current parallel programming APIs requires significant computational and memory overhead. For example, standard POSIX threads must be able to receive signals, which either requires an OS representation of every thread or requires user-level signal multiplexering [14]. Threads in a large-scale multithreading context often need only a few bytes of stack (if any) and do not require the ability to receive signals. Some architectures, such as the Processor-in-Memory (PIM) designs [5, 18, 23], suggest threads that are merely a few instructions included in the thread's context.

While hardware-based lightweight threading constructs are important developments, the methods for exposing such parallelism to the programmer are platform-specific and typically rely either on custom compilers [3, 4, 7, 10, 11, 26], entirely new languages [1, 6, 8, 13], or have architectural limitations that cannot scale to millions of threads [14, 22]. This makes useful comparisons between architectures difficult. With a standard way of expressing parallelism that can be used with existing compilers, comparing cross-platform algorithms becomes convenient. For example, the MPI standard allows a programmer to create a parallel application that is portable to any system providing an MPI library, and different systems can be compared with the same code on each system. Development and study of large-scale multithreaded applications is limited because of the platform-specific nature of the available interfaces. Having a portable large-scale multithreading interface al-

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

lows application development on commodity hardware that can exploit the resources available on large-scale systems.

Lightweight threading requires a lightweight synchronization model [9]. The model used by the Cray XMT and PIM designs, pioneered by the Denelcor HEP [15], uses full/empty bits (FEBs). This technique marks each word in memory with a "full" or "empty" state, allows programs to wait for either state, and makes the state change atomically with the word's contents. This technique can be implemented directly in hardware, as it is in the XMT. Alternatives include ADA-like protected records [24] and fork-sync [4], which lack a clear hardware analog.

This paper discusses programming models' impact on efficient multithreading and the resource management necessary for those models. It introduces the qthread lightweight threading API and its Unix implementation. The API is designed to be a lightweight threading standard for current and future architectures. The Unix implementation is a proof of concept that provides a basis for developing applications for large scale multithreaded architectures.

2. Recursive threaded programming models and resource management

Managing large numbers of threads requires managing per-thread resources, even if those requirements are low. This management can affect whether multithreaded applications run to completion and whether they execute faster than an equivalent serial implementation. The worst consequence of poor management is deadlock: if more threads are needed than resources are available, and reclaiming thread resources depends on spawning more threads, the system cannot make forward progress.

2.1. Parallel programming models

An illustrative example of the effect the programming model on resource management is the trivial problem of summing integers in an array. A serial solution is trivial: start at the beginning, and tally each sequential number until the end of the array. There are at least three parallel execution models that could compute the sum. They can be referred to as the recursive tree model, the equal distribution model, and the lagging-loop model.

A recursive tree solution to summing the numbers in an array is simple to program: divide the array in half, and spawn two threads to sum up both halves. Each thread does the same until its array has only one value, whereupon the thread returns that value. Thread that spawned threads wait for their children to return and return the sum of their values. This technique is parallel, but uses a large amount of state. At any point, most of the threads are not doing useful work. While convenient, this is a wasteful technique.

The equal distribution solution is also simple: divide the array equally among all of the available processors and spawn a thread for each. Each thread must sum its segment serially and return the result. The parent thread sums the return values. This technique is efficient because it matches the needed parallelism to the available parallelism, and the processors do minimal communication. However, equal distribution is not particularly tolerant of other load imbalances: execution is as slow as the slowest thread.

The lagging-loop model relies upon arbitrary workload divisions. It breaks the array into small chunks and spawns a thread for each chunk. Each thread sums its chunk and then waits for the preceding thread (if any) to return an answer before combining the sum and returning its own total. Eventually the parent thread will do the same with the last chunk. This model is more efficient than the tree model, and the number of threads depends on the chunk size. The increased number of threads makes it more tolerant of load imbalances, but has more overhead.

2.2. Handling resource exhaustion

These methods differ in the way resources must be managed to guarantee forward progress. Whenever new thread is requested, one of four things can be done:

1. Execute the function inline.
2. Create a new thread.
3. Create an record that will become a thread later.
4. Block until sufficient resources are available.

In a large enough parallel program, eventually the resources will run out. Requests for new threads must either block until resources become available or must fail and let the program handle the problem.

Blocking to wait for resources to become available affects each parallel model differently. The lagging loop method works well with blocking requests, because the spawned threads don't rely on spawning more threads. When these threads complete, their resources may be reused, and deadlock is easily avoided. The equal distribution method has a similar advantage. However, because it avoids using more than the minimum number of threads, it does not cope as well with load imbalances.

The recursive tree method gathers a lot of state quickly and slowly releases it, making the method particularly susceptible to resource exhaustion deadlock, where all running threads are blocked spawning more threads. In order to guarantee forward progress, resources must be reserved when threads spawn and threads must execute serially when reservation fails. The minimum state that must be reserved is the amount necessary get to the bottom of the recursive tree serially. Thus, if there are only enough resources for a single depth-first exploration of the tree, recursion may only occur serially. If there are enough resources for two serial explorations of the tree, the tree may be divided into two segments to be explored in parallel, and so forth. Once resource reservation fails, only a serial traversal of the recursive tree may be performed. Thus, blocking for resources is a poor behavior for a recursive tree as forward progress cannot be assured.

Such an algorithm is only possible when the maximum depth of the recursive tree is known. If the depth is unknown, then sufficient resources for a serial execution cannot be reserved. *Any* resources reserved for a parallel execution could prevent the serial recursive tree from completing.

It is worth noting that a threading library can only be responsible for the resources necessary for basic thread state. Additional state required during recursion has the potential to cause deadlock and must be managed similarly.

3. Application programming interface

The qthread API provides several key features:

- Large scale lightweight multithreading support
- Access to or emulation of lightweight synchronization
- Basic thread-resource management
- Source-level compatibility between platforms
- A library-based API, forgoing custom compilers

The qthread API maximizes portability to architectures supporting lightweight threads and synchronization primitives by providing a stable interface to the programmer. Because architectures and operating systems supporting lightweight threading are difficult to obtain, initial analysis of the API's performance and usability studies commodity architectures such as Itanium and PowerPC processors.

The qthread API consists of three components: the core lightweight thread command set, a set of commands for resource-limit-aware threads ("futures"), and an interface for basic threaded loops. Qthreads have a restricted stack size, and provide a locking scheme based on the full/empty bit concept. The API provides alternate threads, called "futures", which are created as resources are available.

One of the likely features of machines supporting large scale multithreading is non-uniform memory access (NUMA). To take advantage of NUMA systems, they must be described to the library, in the form of "shepherds," which define memory locality.

3.1. Basic thread control

The API is an anonymous threading interface. Threads, once created, cannot be controlled by other threads. However, they can provide FEB-protected return values so that a thread can easily wait for another. FEBs do not require polling, which is discouraged as the library does not guarantee preemptive scheduling.

Threads are assigned to one of several "shepherds" at creation. A shepherd is a grouping construct. The number of shepherds is defined when the library is initialized. In an environment supporting traveling threads, shepherds allow threads to identify their location. Shepherds may correspond to nodes in the system, memory regions, or protection domains. In the Unix implementation, a shepherd is managed by at least one pthread which executes qthreads. It is worth noting that this hierarchical thread structure, particular to the Unix implementation (not inherent to the API), is not new but rather useful for mapping threads to mobility domains. A similar strategy was used by the Cray X-MP [30], as well as Cilk [4] and other threading models.

Only two functions are required for creating threads: qthread_init (shep), which initializes the library with shep shepherds; and qthread_fork(func,arg,ret), which creates a thread to perform the equivalent of $*$ret = func(arg). The API also provides mutex-style and FEB-style locking functions. Using synchronization external to the qthread library is not encouraged, as that prevents the library from making scheduling decisions.

The mutex operations are qthread_lock(addr) and qthread_unlock(addr). The FEB semantics are more complex, with functions to manipulate the FEB state in a non-blocking way (qthread_empty(addr) and qthread_fill (addr)), as well as blocking reads and blocking writes. The blocking read functions wait for a given address to be full and then copy the contents of that address elsewhere. One (qthread_readFF()) will leave the address marked full, the other (qthread_readFE()) will then mark the address empty. There are also two write actions. Both will fill the address being written, but one (qthread_writeEF()) will wait for the address to be empty first, while the other (qthread_writeF()) won't. Using the two synchronization techniques on the same addresses at the same time produces undefined behavior, as they may be implemented using the same underlying mechanism.

3.2. Futures

Though the API has no built-in limits on the number of threads, thread creation may fail due to memory limits or other system-specific limits. "Futures" are threads that allow the programmer to set limits on the number of futures that may exist. The library tracks the futures that exist, and stalls attempts to create too many. Once a future exits, a future waiting to be created is spawned and its parent thread is unblocked. The futures API has its own initialization function (future_init (limit)) to specify the maximum number of futures per shepherd, and a way to create a future (future_fork (func,arg, ret)) that behaves like qthread_fork ().

3.3. Threaded loops and utility functions

The qthread API includes several threaded loop interfaces, built on the core threading components. Both C++-based templated loops and C-based loops are provided. Several utility functions are also included as examples. These utility functions are relatively simple, such as summing all numbers in an array, finding the maximum value, or sorting an array.

There are two parallel loop behaviors: one spawns a separate thread for each iteration of the loop, and the other uses an equal distribution technique. The functions that provide one thread per iteration are qt_loop() and qt_loop_future(), using either qthreads or futures, respectively. The functions that use equal distribution are qt_loop_balance() and qt_loop_balance_future(). A variant of these, qt_loopaccum_balance(), allows iterations to return a value that is collected ("accumulated").

The qt_loop() functions take arguments start, stop, stride, func, and argptr. They behave like this loop:

```
unsigned int i;
for (i = start; i < stop; i += stride) {
    func(NULL, argptr);
}
```

The qt_loop_balance() functions, since they distribute the iteration space, require a function that takes its iteration space as an argument. Thus, while it behaves similar to qt_loop(), it requires that its func argument point to a function structured like this:

```
void func(qthread_t *me, const size_t startat,
          const size_t stopat, void *arg) {
    for (size_t i = startat; i < stopat; i++)
        /* do work */
}
```

The qt_loopaccum_balance() functions require an accumulation function so that return values can be gathered. The function behaves similar to the following loop:

```
unsigned int i;
for (i = start; i < stop; i++) {
    func(NULL, argptr, tmp);
    accumulate(retval, tmp);
}
```

Similar to the qt_loop_balance() function, it uses the equal distribution technique. The func function must store its return value in tmp, which is then given to the accumulate function to gather and store in retval.

4. Performance

The design of the qthread API is based around two primary goals: efficiency in handling large numbers of threads and portability to large-scale multithreaded architectures. The implementation of the API discussed in this section is the Unix implementation, which is for POSIX-compatible Unix-like systems running on traditional CPUs, such as PowerPC, x86, and IA-64 architectures. In this environment, the qthread library relies on pthreads to allow multiple threads to run in parallel. Lightweight threads are created as a processor context and a small (4k) stack. These lightweight threads are executed by the pthreads. Context-switching between qthreads is performed as necessary rather than on an interrupt basis. For performance, memory is pooled in shepherd-specific structures, allowing shepherds to operate independently.

Without hardware support, FEB locks are emulated via a central hash table. This table is a bottleneck that would not exist on a system with hardware lightweight synchronization support. However, the FEB semantics still allow applications to exploit asynchrony even when using a centralized implementation of those semantics.

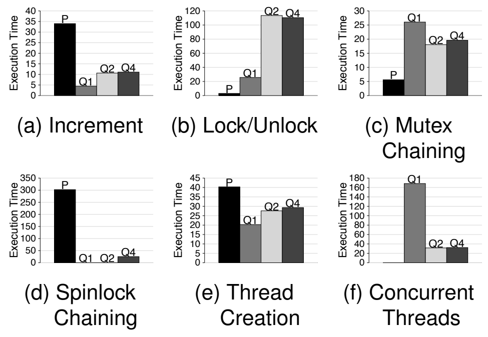
4.1. Benchmarks

To demonstrate qthread's advantages, six microbenchmarks were designed and tested using both pthreads and qthreads. The algorithms of both implementations are identical, with the exception that one uses qthreads as the basic unit of threading and the other uses pthreads. The benchmarks are as follows:

1. Ten threads atomically increment a shared counter one million times each
2. 1,000 threads lock and unlock a shared mutex ten thousand times each
3. Ten threads lock and unlock 1 million mutexes
4. Ten threads spinlock and unlock ten mutexes 100 times
5. Create and execute 1 million threads in blocks of 200 with at most 400 concurrently executing threads
6. Create and execute 1 million concurrent threads

Figure 1 illustrates the difference between using qthreads and pthreads on a 1.3Ghz dual-processor PowerPC G5 with 2GB of RAM. Figure 2 illustrates the same on a 48-node 1.5Ghz Itanium Altix with 64GB of RAM. Both systems used the Native Posix Thread Library Linux Pthread implementation. The bars in each chart in Figure 1 are, from left to right, the pthread implementation, the qthread implementation with a single shepherd, with two shepherds, and with four shepherds. The bars in each chart in Figure 2 are, from left to right, the pthread implementation, the qthread implementation with a single shepherd, with 16 shepherds, with 48 shepherds, and with 128 shepherds.

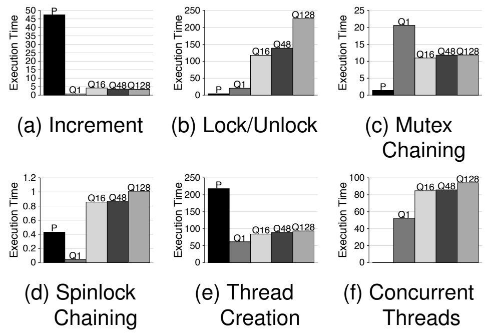
In Figures 1(a) and 2(a), using pthreads is outperformed by qthreads because qthreads uses a hardware-based atomic increment while pthreads is forced to rely on a mutex. Because of contention, additional shepherds do not improve the qthread performance but rather decrease it slightly. Since the qthread locking implementation is built with pthread mutexes, it cannot compete with raw pthread mutexes for speed, as illustrated in Figures 1(b), 2(b), 1(c),



(a) Increment (b) Lock/Unlock (c) Mutex Chaining

(d) Spinlock Chaining (e) Thread Creation (f) Concurrent Threads

Figure 1: Microbenchmarks on a dual PPC



(a) Increment (b) Lock/Unlock (c) Mutex Chaining

(d) Spinlock Chaining (e) Thread Creation (f) Concurrent Threads

Figure 2: Microbenchmarks on a 48-node Altix

and 2(c). This is a detail that would likely not be true on a system that had hardware support for FEBs, and would be significantly improved with a better centralized data structure, such as a lock-free hash table. Because of the qthread library's simple scheduler, it outperforms pthreads when using spinlocks and a low number of shepherds, as illustrated in Figure 1(d). The impact of the scheduler is demonstrated by larger numbers of shepherds (Figure 2(d)).

The pthread library was incapable of more than several hundred concurrent threads—requesting too many threads deadlocked the kernel (Figures 1(f) and 2(f)). A benchmark was designed that worked within pthreads' limitations by allowing a maximum of 400 concurrent threads. Threads are spawned in blocks of 200, and after each block, threads are joined until there are only 200 outstanding before spawning a new block of 200 threads. In this benchmark, Figures 1(e) and 2(e), pthreads performs more closely qthreads—on the PowerPC system, it is only a factor of two more expensive.

5. Application development

Development of software that realistically takes advantage of lightweight threading is important to research, but difficult to achieve due to the lack of lightweight threading interfaces. To evaluate the performance potential of the API and how difficult it is to integrate into existing code, two representative applications were considered. First, a parallel quicksort algorithm was analyzed and modified to fit the qthread model. Secondly, a small parallel benchmark was modified to use qthreads.

5.1. Quicksort

Portability of an API does not free the programmer completely from taking the hardware into consideration when designing an algorithm. There are features of alternative threading environments that the qthread API does not emulate, such as the hashed memory design found in the Cray MTA-2. Memory addresses in the MTA-2 are distributed throughout the machine at word boundaries. When dividing work amongst several threads on the MTA-2, the boundaries of the work regions can be fine-grained without significant loss of performance. Conventional processors, on the other hand, assume that memory within page boundaries are all contiguous. Thus, conventional cache designs reward programs that allow an entire page to reside in a single processor's cache, and limit the degree to which tasks can be divided among multiple processors without paying a heavy cache coherency penalty.

An example wherein the granularity of data distribution can be crucial to performance is a parallel quicksort algorithm. In any quicksort algorithm, there are two phases: first the array is partitioned into two segments around a "pivot" point, and then both segments are sorted independently. Sorting the segments independently is relatively easy, but partitioning the array in parallel is more complex. On the MTA-2, elements of the array to be partitioned can be divided up among each thread without regard to the location of the elements. On conventional processors, however, that behavior is very likely to result in multiple processors transferring the same cache-line or memory page between processors. Constantly sending the same memory back and forth between processors prevents the parallel algorithm from exploiting the capabilities of multiple processors.

The qthread library includes an implementation of the quicksort algorithm that avoids contention problems by ensuring that work chunks are always at least the size of a page. This avoids cache-line competition between processors while still exploiting the parallel computational power of all available processors on sufficiently large arrays. Figure 3 illustrates the scalability of the qthread-based quicksort implementation, and compares its performance to the libc `qsort()` function. This benchmark sorts an array of one billion double-precision floating point numbers on a 48-node SGI Altix SMP with 1.5Ghz Itanium processors.

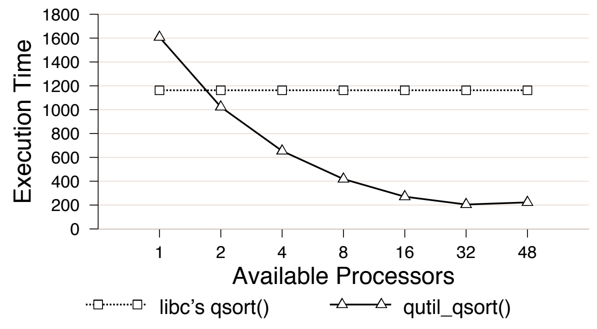


Figure 3: qutil_qsort() and libc's qsort()

5.2. High performance computing conjugate gradient benchmark

The qthread API makes parallelizing ordinary serial code simple. As a demonstration of its capabilities, the HPCCG benchmark from the Mantevo project [20] was parallelized with the Qloop interface of the qthread library. The HPCCG program is a conjugate gradient benchmarking code for a 3-D chimney domain, largely based on code in the Trilinos[21] solver package. The code relies largely upon tight loops where every iteration of the loop is essentially independent of every other iteration. With simple modifications to the code structure, the serial implementation of HPCCG was transformed into multithreaded code. As illustrated in Figure 4, the parallelization is able to scale well. Results are presented using strong scaling with a uniform 75x75x1024 domain on a 48-node SGI Altix SMP. The SGI MPI results are presented to 48 processes, or one process per CPU, as further results would over-subscribe the processors, which generally underperforms with SGI MPI.

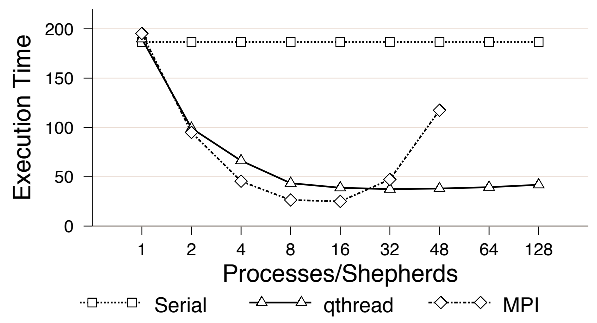


Figure 4: HPCCG on a 48-Node SGI Altix SMP

One of the features of the HPCCG benchmark is that it comes with an optimized MPI implementation. The MPI implementation, using SGI's MPI library, is entirely different from the qthread implementation and does not use shepherds. The qthread and MPI implementations scale approximately equally well up to about sixteen nodes. Beyond sixteen nodes however, MPI begins to behave very badly. At the same time, the qthread implementation's execution time does not change significantly.

Upon analysis of the MPI code, the poor performance of the MPI implementation is caused by MPI_Allreduce() in one of the main functions of the code. While this takes almost 18.9% of execution time with eight MPI processes, it takes 84.1% of the execution time with 48 MPI processes. While it is tempting to simply blame the problem on a bad implementation of MPI_Allreduce(), it is probably more valid to examine the difference between the qthread and MPI implementations. The qthread implementation performs the same computation as the MPI_Allreduce(), but rather than require all nodes to come to the same point before the reduction can be computed and distributed, the computation is performed as the component data becomes available from the threads returning, the computational threads can exit, and other threads scheduled on the shepherds can proceed. The qthread implementation exposes the asynchronous nature of the whole benchmark, while the MPI implementation does not. This asynchrony is revealed even though the Unix implementation of the qthread library relies upon centralized synchronization, and would likely provide further improvement on a real massively parallel architecture.

6. Related work

Lightweight threading models generally fit one of two descriptions: they either require a special compiler or they aren't sufficiently designed for large-scale threading (or both). For example, Python stackless threads [28] provide extremely lightweight threads. Putting aside issues of usability, which is a significant issue with stackless threads, the interface allows for no method of applying data parallelism to the stackless threads: a thread may be scheduled on any processor. Many other threading models, from nanothreads [26] to OpenMP [11], lack a sufficient means of allowing the programmer to specify locality. This becomes a significant issue as machines get larger and memory access becomes non-uniform [31]. Languages such as Chapel [8] and X10 [6], or modifications to existing languages such as UPC [13] and Cilk [4], that require special compilers are interesting and allow for better parallel semantic expressiveness than approaches based in adding library calls to existing languages. However, such models not only break compatibility with existing large codebases but also do not provide for strong comparisons between architectures. Some threading models, such as Cilk, use a fork-and-join style of synchronization that, while semantically convenient, does not allow for as fine-grained control over communication between threads as the FEB-based model, which allows individual load and store instructions to be synchronized.

The drawbacks of heavyweight, kernel-supported threading such as pthreads are well-known [2], leading to the development of a plethora of user-level threading models. The GNU Portable Threads [14], for example, allow a programmer to use user-level threading on any system that supports the full C standard library. It uses a signal stack to allow the subthreads to receive signals, which limits its ability to scale. Coroutines [29] are another model that allow for virtual threading even in a serial-execution-only environment, by specifying alternative contexts that get used at specific times. Coroutines can be viewed as the most basic form of cooperative multitasking, though they can use more synchronization points than just context-switch barriers when run in an actual parallel context. One of the more powerful details of coroutines is that generally one routine specifies which routine gets processing time next, which is behavior that can also be obtained when using continuations [19, 27]. Continuations, in the most broad sense, are primarily a way of minimizing state during blocking operations. When using heavyweight threads, whenever are a thread does something that causes it to stop executing, its full context—local variables, a full set of processor registers, and the program counter—are saved so that when the thread becomes unblocked it may continue as if it had not blocked. A continuation allows the programmer to specify that when a thread blocks it exits, and that unblocking causes a new thread to be created with specific arguments, thus requiring the programmer to save any necessary state to memory while any unnecessary state can be disposed of. Protothreads [12, 17] and Python stackless threads [28], by contrast, assert that outside of CPU context there is no thread-specific state (i.e. "stack") at all. This makes them extremely lightweight but limits the flexibility (at most, only one of them can call a function), which has repercussions for ease-of-use. User-level threading models can be further enhanced with careful kernel modification [25] to enable convenient support of many of the features of heavyweight kernel threads, such as signals, advanced scheduling conventions, and even limited software interrupt handling.

The problem of resource exhaustion due to excessive parallelism was considered by Goldstein et. al. [16]. Their "lazy-threads" concept addresses the issue that most threading models conflate logical parallelism and actual parallelism. This semantics problem often requires that programmers tailor the expression of parallelism to the available parallelism, thereby forcing programmers to either require too much overhead in low-parallelism situations or forgo the full use of parallelism in high-parallelism situations.

The qthread API combines many of the advantages of other threading models. The API allows parallelism to be expressed independently of the parallelism used, much like Goldstein's lazy-thread approach. However, rather than require a customized compiler, the qthread API does this within a library that uses two different categories of threads: thread workers (shepherds) and stateful thread work units (qthreads). This technique, while convenient, has overhead that a compiler-based optional-inlining method would not: every qthread requires memory. This overhead can be limited arbitrarily through the use of futures, which is a powerful abstraction to express resource limitations without limiting the expressibility of inherent algorithmic parallelism.

7. Future work

Much work still remains in development of the qthread API. A demonstration of how well the API maps to the APIs of existing large scale architectures, such as the Cray MTA/XMT systems, is important to reinforce the claim of portability. Custom implementations for other architectures would be useful, if not crucial.

Along similar lines, development of additional benchmarks to demonstrate the potential of the qthread API and large-scale multithreading would be useful for studying the effect of large-scale multithreading on standard algorithms. The behavior and scalability of such benchmarks will provide guidance for the development of new large-scale multithreading architectures.

Thread migration is an important detail of large scale multithreading environments. The qthread API addresses this with the shepherd concept, but the details of mapping shepherds to real systems requires additional study. For example, shepherds may need to have limits enforced upon them, such as CPU-pinning, in some situations. The effect of such limitations on multithreaded application performance is unknown, and deserving of further study.

8. Conclusions

Large scale computation of the sort performed by common computational libraries can benefit significantly from low-cost threading, as demonstrated here. Lightweight threading with hardware support is a developing area of research that the qthread library assists in exploring while simultaneously providing a solid platform for lighter-weight threading on common operating systems. It provides basic lightweight thread control and synchronization primitives in a way that is portable to existing highly parallel architectures as well as to future and potential architectures. Because the API can provide scalable performance on existing platforms, it allows study and modeling of the behavior of large scale parallel scientific applications for the purposes of developing and refining such parallel architectures.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 1.0β edition, March 2007.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support fot the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[3] A. Begel, J. MacDonald, and M. Shilman. Picothreads: Lightweight threads in java.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[5] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge. A low cost, multithreaded processing-in-memory system. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 16–22, New York, NY, USA, 2004. ACM Press.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[7] Cray XMT platforrm. http://www.cray.com/products/xmt/index.html, October 2007.

[8] Cray Inc., Seattle, WA 98104. *Chapel Language Specification*, 0.750 edition.

[9] H.-E. Crusader. High-end computing needs radical programming change. *HPCWire*, 13(37), September 2004.

[10] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. Tama compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, 1993.

[11] L. Dagum and R. Menon. OpenMP: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[12] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[13] T. El-Ghazawi and L. Smith. Upc: unified parallel c. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27, New York, NY, USA, 2006. ACM.

[14] R. S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *ATEC'00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[15] M. C. Gilliland, B. J. Smith, and W. Calvert. Hep - a semaphore-synchronized multiprocessor with central control (heterogeneous element processor). In *Summer Computer Simulation Conference*, pages 57–62, Washington, D.C., July 1976.

[16] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[17] B. Gu, Y. Kim, J. Heo, and Y. Cho. Shared-stack cooperative threads. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied Computing*, pages 1181–1186, New York, NY, USA, 2007. ACM Press.

[18] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 57, New York, NY, USA, 1999. ACM Press.

[19] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 293–298, New York, NY, USA, 1984. ACM Press.

[20] M. Heroux. Mantevo. http://software.sandia.gov/mantevo/index.html, December 2007.

[21] M. Heroux, R. Bartlett, V. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, et al. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[22] Institute of Electrical and Electronics Engineers. *IEEE Std 1003.1-1990: Portable Operating Systems Interface (POSIX.1)*, 1990.

[23] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, 1996.

[24] C. D. Locke, T. J. Mesler, and D. R. Vogel. Replacing passive tasks with ada9x protected records. *Ada Letters*, XIII(2):91–96, 1993.

[25] B. D. marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operaing Systems Principles*, pages 110–121, New York, NY, USA, 1991. ACM Press.

[26] X. Martorell, J. Labarta, N. Navarro, and E. Ayguade. A library implementation of the nano-threads programming model. In *Euro-Par, Vol. II*, pages 644–649, 1996.

[27] A. Meyer and J. G. Riecke. Continuations may be unreasonable. In *LFP '88: Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 63–71, New York, NY, USA, 1988. ACM Press.

[28] Stackless python. http://www.stackless.org, January 2008.

[29] S. E. Sevcik. An analysis of uses of coroutines. Master's thesis, 1976.

[30] F. Szelényi and W. E. Nagel. A comparison of parallel processing on Cray X-MP and IBM 3090 VF multiprocessors. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 271–282, New York, NY, USA, 1989. ACM.

[31] J. Tao, W. Karl, and M. Schulz. Memory access behavior analysis of NUMA-based shared memory programs, 2001.

DOE's Institute for Advanced Architecture and Algorithms: an Application-Driven Approach

Richard C. Murphy

Sandia National Laboratories, PO Box 5800, MS-1319, Albuquerque, NM 87185

E-mail: rcmurph@sandia.gov

Abstract. This paper describes an application driven methodology for understanding the impact of future architecture decisions on the end of the MPP era. Fundamental transistor device limitations combined with application performance characteristics have created the switch to multicore/multithreaded architectures. Designing large-scale supercomputers to match application demands is particularly challenging since performance characteristics are highly counter-intuitive. In fact, data movement more than FLOPS dominates. This work discusses some basic performance analysis for a set of DOE applications, the limits of CMOS technology, and the impact of both on future architectures.

1. Introduction

As the MPP era draws to a close, the energy inefficiency of modern supercomputer architectures is rapidly becoming the limiting factor for architectural scalability. The MPP era's total reliance on commodity processors that are optimized for less challenging application spaces has created energy inefficiencies throughout the system. Further, fundamental transistor device physics has forced scaling in the form of increased parallelism at the chip level, rather than the exponential increase in single thread performance that characterized the MPP era. Indeed, the transition to multicore architectures is driven entirely by the aging of the CMOS fabrication process. Without a well-defined successor capable of Moore's Law scaling, the transistor limits the potential of future supercomputers deployed in energy constrained environments. As a result, changes to computer architecture are required to enable newer, more highly-scalable applications. However, choosing the right architectural path is extremely difficult. Generally, computer architects do everything possible to hide the details of the hardware implementation from the programmer: cache sizes, branch prediction policies, out-of-order execution mechanisms, and thread scheduling at the hardware level are all examples of critical architectural mechanisms and structures that are typically observed only indirectly by the programmer.

More worrisome still, each of these structures that architects use to boost single thread performance and hide the depths and complexities of modern communication hierarchies (memory or interconnect) consumes energy that could otherwise be available for the computation if the programmer could exploit simpler, lighter-weight mechanisms. It has long been argued that these mechanisms should be exposed to the programmer[9, 5, 1, 4], but the transition to multithreaded/multicore architectures leaves little choice in an energy constrained environment.

This paper presents a synthesis of application analysis results derived from simulation, analysis tools, and real hardware. The remainder of the paper is structured as follows: Section

2 discusses some of the applications studied in this work; Section 3 examines the limits of CMOS transistors; Section 4 provides results to show that performance bottlenecks tend to be latency and concurrency dominated; and Section 5 describes the impact of these trends on future architectures.

2. Applications Discussed In This Work

The application studies referenced in this work are divided into two basic application classes: physics codes, which tend to be floating point oriented; and informatics codes, which tend to be more integer oriented. The full application analysis and simulation methodology has been previously described[7, 6, 8]. The results are summarized in this paper.

2.1. Physics Applications

Broadly speaking, the physics application suite analyzed by Sandia are large-scale MPI applications, and represent real world physical simulations. They are:

- **ALEGRA** is a finite element shock physics code capable of modeling near- and far-field responses to explosions, impacts, and energy depositions.
- **CTH** is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories over the last 30 years. CTH models multi-phase, elastic viscoplastic, porous and explosive materials with multiple mesh refinement methods.
- **Cube3** Cube3 is a generic linear solver that drives the Trilinos framework for parallel linear and eigensolvers. It mimics a finite element analysis problem by creating hexagonal elements, then assembling and solving a linear system. The width, depth, and degrees of freedom (e.g., temperature, pressure, velocity, etc.) can be varied.
- **ITS** performs Monte Carlo simulations of linear time-independent coupled electron/photon radiation transport.
- **MPSalsa** is a high resolution 3d simulation of reacting flow. The simulation requires both fluid flow and chemical kinetics modeling.
- **Xyce** is a parallel circuit simulation system capable of modeling very large circuits at multiple layers of abstraction (device, analog, digital, and mixed-signal). It includes both SPICE-like models and radiation models.

This suite of applications is somewhat more narrow than those of interest to the Office of Science, but future work in the DOE Institute for Advanced Architecture will apply the same fundamental analysis techniques to specific Office of Science applications.

2.2. Informatics Applications

Recently, new large-scale applications in the area of informatics have emerged as important and substantially different from physics applications that have been the core of supercomputing over the past three decades. These applications tend to be more integer-oriented, and represent problems in discrete math. They are typically less structured (in terms of memory access patterns), and are often techniques used to analyze the output of a physical simulation, or to examine real world data sets in an attempt to find patterns or form hypotheses. Additionally, because Informatics applications tend to be less structured in nature, they are written using a variety of programming models. The application set discussed are:

- **DFS** implements a depth-first search on a large graph and forms the basis for several high-level algorithms including connected components, tree and cycle detection, solving the two-coloring problem, finding Articulation Vertices, and topological sorting.

- **Connected Components** breaks a graph into components. Two vertices are in a connected component if and only if there is a path between them.
- **Subgraph Isomorphism** determines whether or not a subgraph exists within a larger graph.
- **Full Graph Isomorphism** determines whether or not two graphs have the same shape or structure.
- **Shortest Path** computes the shortest path between two vertices using a breadth first search. Real world applications include path planning and networking and communication.
- **Graph Partitioning** is used extensively in VLSI circuit design and adaptive mesh refinement. The problem divides a graph in to k partitions while minimizing the *cut* between the partitions (or the total weight of the edges that cross from one partition to another).
- **BLAST** is the Basic Local Alignment Search Tool and is the most heavily used method for quickly search nucleotide and protein databases in biology.
- **zChaff** is a heuristic for solving the Boolean Satisfiability Problem. In propositional logic, a formula is *satisfiable* if there exists an assignment of truth values to each of its variables that make the formula true.

These applications have the potential to significantly change the methodology used in computer-driven science, and fundamentally represent a new and disruptive use of high performance computing resources.

3. CMOS Transistor Limitations

The switch to multicore is fundamentally driven by a change in the CMOS transistor that forms the implementation technology for commodity processors. Power dissipation (manifesting itself as heat which must be removed from the device) can be broken into two components: Static Power Dissipation (P_{stat}), which is dissipated continuously while the circuit is powered and Dynamic Power Dissipation (P_{dyn}) which is dissipated during switching. Although increasingly important, static power dissipation cannot be addressed by the computer architect (other than by powering off parts of the machine not in use), and is primarily a function of the fabrication process. It is defined as:

$$P_{stat} + I_{leakage}V_{dd}$$

Where $I_{leakage}$ is the leakage current of the device, and V_{dd} is the voltage. It should be noted that while the computer architect cannot do much to control leakage current, the system architect and environment in which the machine is deployed can. At the junction, leakage is generally caused by thermally generated carriers, and *increases* exponentially as temperature rises. In fact, at 85°C, leakage current increases by a factor of 60 over room temperature values. These factor and increased failure rates must be taken into account in the “hotter running” compute environments that have become a topic of recent significant interest.

The majority of power dissipation is dynamic, and is defined as as:

$$P_{dyn} = C_L V_{dd}^2 f$$

Where C_L is the capacitance of the entire circuit, V_{dd} the voltage, and f the switching frequency (proportional to the clock frequency since not all devices will switch every cycle). This equation is critically important in describing the switch to multicore.

Moore’s Law as originally defined demands an exponential increase in the number of transistors, which increases C_L. Specifically, for a given area, the total capacitance is the sum of the capacitance of each device. Although capacitance per device decreases as the feature size decreases (decreasing the individual transistor's contribution to C_L), the number of devices is increasing exponentially. Similarly, until the multicore era the frequency increased substantially in each generation. To keep the total dynamic power dissipation essentially constant (so that chips could be cooled with relatively inexpensive heat syncs at roughly room temperature), the equation was balanced by *decreasing* V_{dd}, which as it approaches 1 volt is nearing a fundamental limit (in fact, the rate at which V_{dd} has decreased has already begun to flatten).

Although there are additional manufacturing benefits to moving to multicore architecture, including simplifying the verification of designs, this fundamental technology limit is the real driver. The devices are capable of higher clock rates, at higher densities, but engineering a system to cope with the power dissipation is nearly impossible. The era is characterized as an increasing bounty of transistors (Moore's original observation), but the inability to increase single thread performance (through faster clock rates). The inevitable result is more parallelism.

Of course the end of the CMOS scaling curve in the next decade or so will cause additional fundamental physical limitations, unless a solution can be found.

4. Performance Bottlenecks Tend to Be Counterintuitive

The following section discusses basic application properties, and performance bottlenecks in the network and memory system.

4.1. Real Floating Point Applications Do Less Floating Point Than Expected

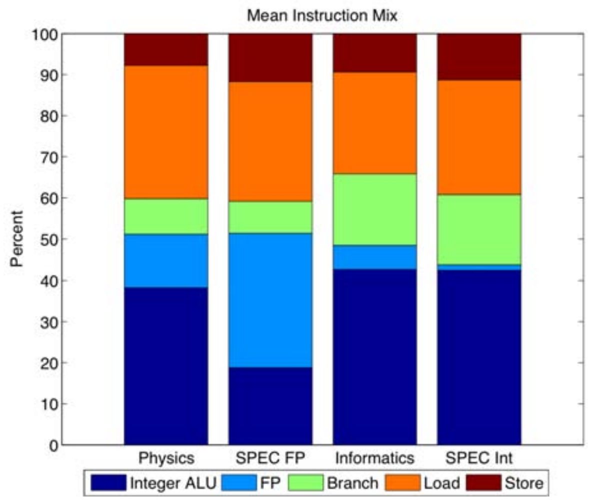


Figure 1. Instruction Mix

The instruction mix for the Physics and Informatics suites is depicted in Figure 1, and compared to the industry standard SPEC CPU 2000 suite. While SPEC FP averages approximately one floating point instruction in three, real Sandia Physics codes average only 12% (and is typically much less than 10%) because they exhibit more complex memory addressing patterns, which, in turn require more integer instructions to calculate the addresses. In fact, nearly 85% of integer instructions are calculating memory addresses, with the remaining 15% split between real integer data and branch (boolean) support.

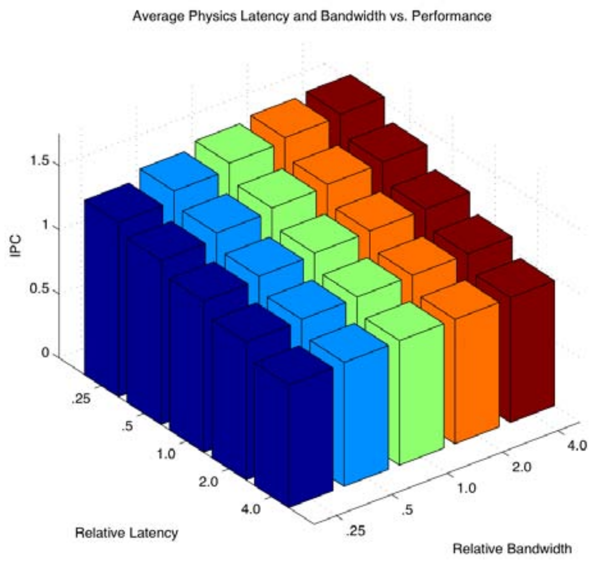
The Informatics applications are very different from the Physics codes as well, suggesting the possibility of two different supercomputer architecture classes in the future. They perform 15% fewer memory references, and those references are much more likely to miss the cache, which stresses the memory subsystem. Additionally, the Informatics applications perform nearly twice as many branches as their Physics counterparts, leading to relatively smaller basic blocks. (This result is unsurprising due to the complex floating point calculations typically performed by the Physics codes[9].) This combination makes the Informatics codes more difficult to scale on modern superscalar processors than their physics counterparts.

4.2. Latency and Concurrency Matter More than Bandwidth

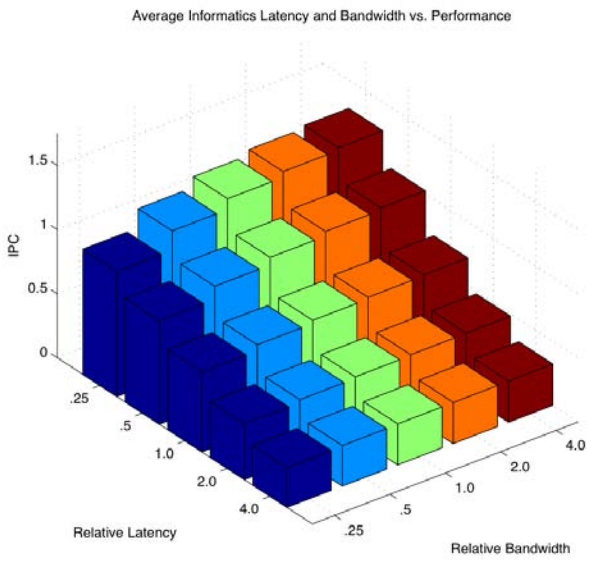
Data movement systems are typically rated by bandwidth, but most evidence points to applications being more sensitive to latency or concurrency. This can be simplistically described by Little's Law (from Economics) which says that:

$$throughput = \frac{concurrency}{latency}$$

Particularly in the case of memory systems, increasing the number of memory requests outstanding (currently limited by hardware and instruction sequences) or decreasing the latency required to receive a response has significantly more impact than changing the bandwidth.



(a)



(b)

Figure 2. Latency and Bandwidth Sensitivity for (a) Physics Applications and (b) Informatics Applications

Figure 2 shows the change in Instructions Per Cycle (IPC, the computer architect's typical measure of performance) as relative latency and bandwidth are altered (the center bar being the latency and bandwidth of a typical processor's memory system. The full results and methodology have been described previously[6]. The results demonstrate that both application suites are significantly more sensitive to latency than bandwidth. A decrease in bandwidth by half on either suites affects performance by less than 5%, while doubling the memory latency halves performance[1]

[1] It should be noted that this bandwidth measure is more strict than that presented for a typical memory part, which will also include a decrease in latency in subsequent generations.

These result combined are confirmed by observations from performance counters and other real system measurements going back to ASCI Red that demonstrates that real applications have difficult saturating the memory bus on a modern architecture because address generation (the integer instructions discussed in Section 4.1) is the bottleneck more than typical memory performance.

This is further demonstrated by typical out-of-order processors that are designed to mask the multi-cycle latency of a Level 1 cache **hit**, more than the latency of a memory access (due to limitations of both the incoming instruction stream and the implementation technology).

The performance of the two application suites is also significantly different. The Physics application achieve reasonable performance on the simulated out-of-order execution unit, achieving an IPC of 1.22 for the base case. The processor is capable of retiring 4 instructions per cycle, so the achieved IPC is approximately 30% of maximum. However, any IPC greater than one is quite good for a typical application suite. In contrast, the baseline case for the Informatics suite is only 0.70, indicating that the applications are significantly more memory intensive, causing the processor additional idle time. Key graph algorithms, including both isomorphism problems and connected components show IPCs significantly less than 0.5. Aside from being less than an eighth of achievable performance, the results indicate that significant power is being wasted on high clock rates for those applications.

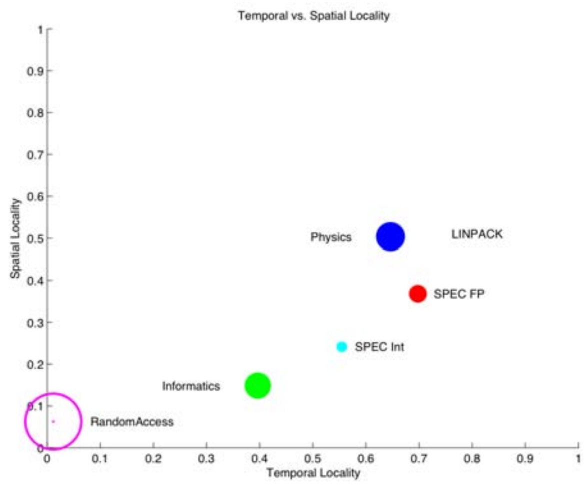


Figure 3. Application Memory Performance Properties

Figure 3 describes the fundamental memory performance properties of each benchmark suite, and has been fully described previously[7]. The spatial locality describes the probability that data near previously referenced data will be referenced. The temporal locality describes the probability that data already referenced will be reused. And the relative size of the dots represents the amount of unique data that each application suite consumes over a fixed interval of instructions. These fundamental application properties dictate the performance observed in Figure 2, with applications consuming more data, and doing so in a more random fashion (i.e., closer to the origin on the graph) showing similarly low performance in comparison to applications with smaller data sets consumed in a more structured pattern. These fundamental application properties determine the class of architecture required to provide high performance at low power more than any other issue.

4.3. Network Performance

These issues are much more difficult to measure for the network, but preliminary results and prior experience show the importance of network injection rate vs. raw bandwidth.

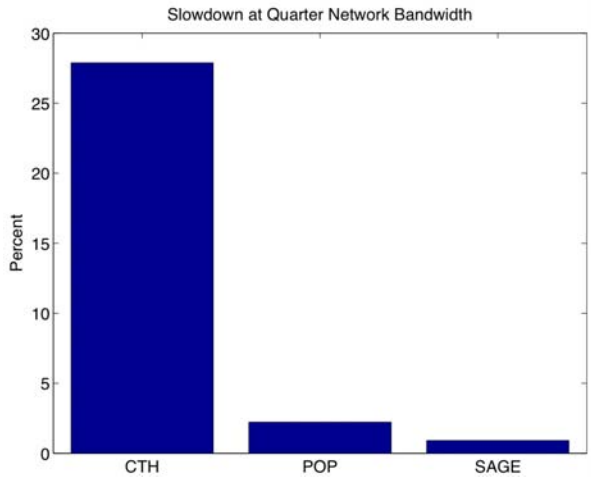


Figure 4. Measurement of Bandwidth Restricted Performance on Red Storm. This is a recast of data collected by Kurt Ferreira at Sandia, a subset of which is discussed in [3]

The trade-off between latency, bandwidth, and message rates are difficult to measure for modern networks, and the trade-offs appear to be highly application dependent. This state of affairs makes drawing firm conclusions on minimal network design difficult, which may lead to either poor application performance for a subset of applications or over-provisioned networks for some subset of applications.

Figure 4 shows the effects of running three applications on Sandia's Red Storm machine, with bandwidth degraded in the mesh by 75%, while not degrading small message latency or injection rate. The mesh is intentionally capable of approximately twice the bandwidth of the link between processor and NIC to offer more stable performance in the face of message contention, whether from non-3D communication patterns or poor application and process placement. The applications shown are CTH, SAGE, and POP. CTH and Sage were run at 2,048 nodes, and POP at 2,500. CTH and Sage were run with weak scaling data sets, and POP as a strong scaling problem.

POP is limited by a combination of message rate and latency, and it is therefore not surprising that reducing network bandwidth does not impact application performance. SAGE, like POP, appears to be immune to bandwidth changes, although it is unclear whether this is problem specific. CTH, on the other hand, shows a 27% performance degradation when mesh bandwidth is reduced to half of injection bandwidth, and is traditionally thought to be insensitive to latency and message rate. The difference between behavior of POP and CTH suggests a significant challenge for hardware design – a need to balance hardware cost, energy usage, and performance across a wide variety of applications.

In an energy optimized environment, one could envision a system that dynamically scales network bandwidth performance to meet both application and overall system demands (contention, I/O, etc.) when full topological bandwidth is not required for the current set of running applications.

5. Impact on Future Architectures: Multicore, Multithreading

Section 3 described the fundamental technology changes pushing multicore processors. The discussion of IPC in Section 4.2 shows that the processor is tremendously underutilized, especially for applications with very large, sparse data sets. Because the problem is fundamentally one of latency, there are only two basic solutions: avoiding the latency through faster devices, memory hierarchies, and caches; or tolerating the latency via mechanisms such as multiple outstanding loads, out-of-order execution, vector pipelines, or threads.

Avoiding latency via faster devices is fundamentally an economically and technologically driven process, and often provides a "one time" benefit rather than a sustained and continuous benefit over multiple generations. For example, cheap 3D integration of memory onto the processor would reduce wire lengths between the processor and memory from centimeters to microns, providing a one-time latency reduction.

Architectural techniques for avoiding latency such as caching also show diminishing returns. Caches place frequently used items closer to the processor, but become less effective proportional to cost as they increase in size. They are also large consumers of energy.

In terms of memory toleration, many of the classic techniques have reached their limits. Given the relatively aggressive out-of-order execution in modern processors and the observed IPCs from Figure 2, it is clear that there is not much room for improvement. In fact, most processors are moving towards simpler latency toleration mechanisms because the power dissipation for out-of-order execution is so high compared with the relatively small benefits. As discussed previously, most applications fail to fully utilize memory system bandwidth, meaning that the problem of generating more outstanding loads is not due to a hardware limitation, but fundamental application properties. For the Informatics suite in particular, the pointer chasing requires the results from one load to be available before another load can be generated, limiting architectural options for solving the problem.

The remaining option for power-efficient latency toleration is threads, and it can be seen in the successes of early throughput-oriented architectures such as the Sun Niagara. Threads are:

- Relatively inexpensive to implement in hardware, requiring additional memory structures to support larger register files, but not requiring complex logic as is required for out-of-order execution;
- Efficient in that switching to another thread while the currently executing thread waits on a long latency event still allows "useful" work to be performed during the idle time (in contrast with techniques such as speculative execution which simply create heat when the architecture predicts an erroneous path of execution);
- Low power, primarily because useful work is always being done and they replace more complex hardware structures;
- Capable of fully utilizing the memory system by exposing more address generation streams to the programmer (if the programmer can exploit them), which can be seen in the trend toward fewer memory controllers per core in modern processors; and,
- Potentially plentiful, in that for the desktop and server space for which the processors are optimized there typically exists sufficient parallelism (even at the task level) to allow for multithreaded processors not to starve, which is potentially a significant application problem for future supercomputers.

Consequently, the trend towards multicore/multithreaded architectures is inevitable, especially in highly energy constrained environments. The DARPA Exascale Report[2] describes dominance of data movement in future exascale architecture energy budgets.

Acknowledgments

I would like to thank Kurt Ferreira for sharing his data used in the networking discussion, as well as Arun Rodrigues, Brian Barrett, Scott Hemmert, and Jim Ang for many useful discussions over the course of preparing this paper.

References

[1] Jay B. Brockman, Peter M. Kogge, Vincent Freeh, Shannon K. Kuntz, and Thomas Sterling. Microservers: A new memory semantics for massively parallel computing. In *ICS*, 1999.

[2] Peter M. Kogge (editor). *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems.* University of Notre Dame, CSE Deptartment Technical Report TR-2008-13, September 28, 2008.

[3] Kurt Ferreira, Ron Brightwell, and Patrick Bridges. Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection. In *Supercomputing 2008 (SC08), Austin TX, November 2008.*

[4] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Supercomputing, Portland, OR*, November 1999.

[5] Shannon K. Kuntz, Richard C. Murphy, Michael T. Niemier, Jesus Izaguirre, and Peter M. Kogge. Petaflop Computing for Protein Folding. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, VA*, March 12-14, 2001.

[6] Richard C. Murphy. On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance. In *IEEE International Symposium on Workload Characterization 2007 (IISWC2007)*, September 27-29, 2007.

[7] Richard C. Murphy and Peter M. Kogge. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and its Implications. *To Appear In IEEE Transactions on Computers.*

[8] Richard C. Murphy, Arun Rodrigues, Peter Kogge, and Keith Underwood. The implications of working set analysis on supercomputing memory hierarchy design. In *The 2005 International Conference on Supercomputing*, June 20-22, 2005.

[9] Arun Rodrigues, Richard Murphy, Peter Kogge, and Keith Underwood. Characterizing a New Class of Threads in Scientific Applications for High End Supercomputers. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing*, pages 164–174, June 26-July 1 2004.

Implementing a Portable Multi-threaded Graph Library: the MTGL on Qthreads

Brian W. Barrett, Jonathan W. Berry, and Richard C. Murphy
Sandia National Laboratories
Albuquerque, NM
{bwbarre,jberry,rcmurph}@sandia.gov

Kyle B. Wheeler
University of Notre Dame and
Sandia National Laboraroties
Notre Dame, IN
kwheeler@cse.nd.edu

1. Introduction

Graph-based Informatics applications challenge traditional high-performance computing (HPC) environments due to their unstructured communications and poor load-balancing. As a result, such applications have typically been relegated to either poor efficiency or specialized platforms, such as the Cray MTA/XMT series. The multi-threaded nature of the Cray MTA architecture[1] presents an ideal platform for graph-based informatics applications. As commodity processors adopt features to enable greater levels of multi-threaded programming and higher memory densities, the ability to run these multi-threaded algorithms on less expensive, more available hardware becomes attractive.

The Cray MTA architecture provides both an auto-threading compiler and a number of architectural features to assist the programmer in developing multi-threaded applications. Unfortunately, commodity processors have increased the amount of concurrency available by adding an ever-growing number of processor cores on a single socket, but have not added the fine-grained synchronization available on the Cray MTA architecture. Further, while auto-threading compilers are being discussed, none provide the feature set of the Cray offerings.

Although massively multi-threaded architectures have shown tremendous potential for graph algorithms, development poses unique challenges. Algorithms typically use light-weight synchronization primitives (Full/Empty bits, discussed in Section 3.1) for synchronization. Parallelism is not expressed explicitly, but instead compiler hints and careful code construction allow the compiler to parallelize a given code. Unlike the Parallel Boost Graph Library (PBGL) [8], which runs on the developers laptop as well as the largest supercomputers, applications developed for the MTA architecture only run on the MTA architecture. Experiments with the programming paradigm require access to the platform, which is obviously a constrained resource.

In this paper, we explore the possibility of using the Qthreads user-level threading library to increase the portability of scalable multi-threaded algorithms. The Multi-Threaded Graph Library (MTGL) [2], which provides generic programming access to the XMT, is our testbed for this work. We show the use of important algorithms from the MTGL on on emerging commodity multi-core and multi-threaded platforms, with only minor changes to the code base. Although performance is not at the same level as the same algorithm on a Cray XMT, the performance motivates our technique as a workable solution for developing multi-threaded codes for a variety of architectures.

1. Throughout this paper, we will use the phrase MTA architecture to refer to Cray's multi-threaded architecture, including both the Cray MTA-2 and Cray XMT platforms.

2. Background

Recent work [9], [12] has described the challenges in HPC graph processing. These challenges are fundamentally related to locality (both spatial and temporal), and the lack thereof when graph algorithms are applied to highly unstructured datasets. The PBGL [8] attempts to meet these challenges through storage techniques that reduce communication. These techniques have been shown to work well in certain contexts, though they introduce other challenges such as memory scalability. Even when they achieve run-time scalability, the processor utilization on commodity CPUs is considerably lower than that found in the MTA architecture.

2.1. Cray XMT

The Cray XMT is the successor to the Cray MTA-2 highly multi-threaded architecture. Unlike the MTA-2, in which all memory was equidistant from any processor on the network, the XMT uses a more traditional model in which memory is closer to a single processor than all others. The Cray XMT utilizes similar processors to the MTA-2, including the ability to sustain 128 simultaneous hardware threads, but with an improved 500 MHz clock rate. Rather than the custom network found on the MTA-2, the XMT utilizes the SeaStar based network found on the Cray XT massively parallel processor distributed memory platform.

2.2. Multi-core Architectures

As processor vendors have begun offering quad-core processors, as well as more commodity multi-threaded processors such as the Sun Niagara processors, it has become possible to write multi-threaded applications on more traditional platforms. Given the high cost of even a small XMT platform, the ability of modern workstations to support a growing number of threads makes them attractive for algorithm development and experimentation.

The Sun Niagara platform opens even greater multi-threaded opportunities, supporting 8 threads per core and 8 cores per socket, for a total of 64 threads per socket. Current generation Niagara processors support single, dual, and quad socket installations. Unlike the Cray XMT, the Sun Niagara uses a more traditional memory system, including L1 and shared L2 cache structures, and an unhashed memory system. The machines are also capable of running unmodified UltraSPARC executables.

2.3. Multi-threaded Programming

Our approach is to take algorithm codes that have already been carefully designed to perform on the MTA architecture, and run them without altering the core algorithm on commodity multi-core machines by simulating the threading hardware. In contrast, codes written using frameworks specifically designed for multi-core commodity machines (e.g. SWARM [13]) won't run on the MTA architecture.

Standard multi-core software designs, such as Intel's Thread Building Blocks [10], OpenMP [6], and Cilk [3], target current multicore systems, and their architecture reflects this. For example, they lack a means of associating threads with a locale. This becomes a significant issue as machines get larger and memory access becomes more non-uniform.

Another important consideration is the granularity and overhead of synchronization. Existing large scale multi-threaded hardware, such as the XMT, implement full/empty bits. This provides for blocking synchronization in a locality-efficient way. Existing multi-threaded software systems tend to use lock-based techniques, such as mutexes and spinlocks, or require tight control over memory layout. These methods are logically equivalent, but are not as efficient to implement. FEB's are memory efficient when implemented in hardware, and thus allow tight memory structures that can be safely operated upon without requiring locking structures to be inserted into them.

3. Qthreads

The Qthread API [16] is a library-based API for accessing lightweight threading and synchronization primitives similar to those provided on the MTA architecture. The API was designed to support large-scale lightweight threading and synchronization in a cross-platform library that can be readily implemented on both conventional and massively parallel architectures. On architectures where there is no hardware support for the features it provides, or where native threads are heavyweight, these features are emulated. There are several existing threading models that support lightweight threading and lightweight synchronization, but none that sufficiently closely emulate the MTA architecture semantics.

Equivalents for basic thread control, FEB-based read and write functions, as well as basic threaded loops (analogs for many of the pragma-defined compiler loop optimizations available on the MTA architecture) are all provided by the API. Even though the operations that do not have hardware support, such as FEB-based operations, are emulated, they retain usefulness as a means of intra-thread communication.

The API establishes convenient management of the basic memory requirements of threads as they are created. When insufficient resources are available, either thread creation fails or it waits for the resources to become available, depending on how the API is used.

Relatively speaking, locality of reference is not an important consideration to the MTA architecture, as the address space is hashed and divided among all processors at word boundaries. This is an unusual environment, and locality is an important consideration in most other large parallel machines. To address this, the Qthread API provides a generalized notion of locality, called a "shepherd", which identifies the location of a thread. A machine may be described to the library as a set of shepherds, which can refer to memory boundaries, CPUs, nodes, or whatever is a useful division. Threads are assigned to specific shepherds when they are created.

3.1. Implementation of MTA Intrinsics

The MTA architecture has several features that are intrinsic to the architecture, which the Qthread library emulates. These features include full/empty bits (FEBs), fast atomic increments, and conditionally created threads.

On the MTA architecture, a full/empty bit (FEB) is an extra hardware flag associated with every word in memory, marking that word either full or empty. Qthreads uses a centralized collection data structure to achieve the same effect: if an address is present in the collection, it is considered "empty", and if not, it is considered "full". Thus, all memory addresses are considered full until they are operated upon by one of the commands that will alter the memory word's contents and full/empty status. The synchronization protecting each word is pushed into the centralized data structure. Not all of the semantics of the MTA architecture can be fully emulated, however. For example, on the MTA architecture, all writes to memory implicitly mark the corresponding memory words as full. However, when pieces of memory are being used for synchronization purposes, even implicit operations are done purposefully by the programmer, and replacing implicit writes with explicit calls is trivial.

The MTA architecture also provides a hardware atomic increment intrinsic. Atomic increment functions have often been considered useful, even on commodity architectures, and so hardware-based techniques for doing atomic increments are common. The Qthread API provides an atomic increment function that uses a hardware-based implementation on supported architectures, but which falls back to using emulated locks to achieve the same behavior on architectures without explicit hardware support in the library. This is an example of opportunistically using hardware features while providing a standardized interface; a key feature of the Qthread API.

3.2. Qthreads implementation of thread virtualization

Conditionally created threads are called "futures" in MTA architecture terminology, and are used to indicate that threads need not be created now, but merely whenever there are resources available for them. This can be crucial on the MTA, as each processor can handle at most 128 threads, and extremely parallel algorithms may generate significantly more. The Qthread API provides an analogous feature by providing alternate thread creation semantics that allow the programmer to specify the permissible number of threads that may exist concurrently, and which will stall thread creation until the number of threads is less than the number of permissible threads.

A key application of this is in loops. While a given loop may have a large number of entirely independent iterations, it is typically unwise to spawn all of the iterations as threads, because each thread has a context and eventually the machine will run out of memory to hold all the thread contexts. Limiting the number of concurrently extant threads limits the amount of overhead that will be used by the threads. In a loop, the option to stall the thread creation while the maximum number of threads still exist provides the ability to specify a threaded loop without the risk of using an excessive amount memory for thread contexts. The limit on the number of threads is a per-shepherd limit, which helps with load balancing.

4. The Multi-Threaded Graph Library

The Multi-Threaded Graph Library is a graph library designed in the spirit of the Boost Graph Library (BGL) and Parallel Boost Graph Library. The library utilizes the generic component features of the C++ language to allow flexibility in graph structures, without changes to a given algorithm. Unlike the distributed memory, message passing based PBGL, the MTGL was designed specifically for the shared-memory multi-threaded MTA architecture. The MTGL includes a number of common graph algorithms, including the breadth-first search, connected components, and PageRank algorithms discussed in this paper.

To facilitate writing new algorithms, the MTGL provides a small number of basic intrinsics upon which graph algorithms can be implemented. The intrinsics hide much of the complexity of multi-threaded race conditions and load-balancing from algorithm developers and users. Parallel Search (PSearch), a recursive parallel variant of depth-first search (which is not truly depth-first in order to achieve parallelism), combined with an extensive vertex and edge visitor interface, provides powerful parallelism for a number of algorithms.

MTA architecture-specific features used by the MTGL are either compiler hints specified via the `#pragma` mechanism or are encapsulated into a limited number of templated functions, which are easily re-implementable for a new architecture. An example is the `mt_readfe` call, which translates to `readfe` on the MTA architecture, a simple read for serial builds on commodity architectures, and `qthread_readfe` on commodity architectures using the Qthreads library.

The combination of an internal interface for explicit parallelism and the set of core intrinsics upon which much of the MTGL is based provides an ideal platform for extension to new platforms. While auto-threading compilers like those found on the MTA architecture are not available for other platforms, the small number of intrinsics can be hand-parallelized with a reasonable amount of effort.

5. Qthreads and the MTGL

Making the MTGL into a cross-platform library required overcoming significant development challenges. The MTA architecture programming environment has a large number of intrinsic semantics, and its cacheless hashed memory architecture has unusual performance characteristics. The MTA compiler also recognizes common programming patterns, such as reductions, and optimizes them transparently. For these reasons, the MTA developer is encouraged to develop "close to the compiler".

The size of stack necessary, for example, presents a challenge. Some MTGL routines are highly recursive, and the MTA transparently handles expanding the stack for each thread as-needed. The Qthread library, however, has a fixed stack size. Iterative solutions, combined with using larger stacks was required to address the issue.

Both the MTA architecture and commodity processors are susceptible to the problem of hot spotting, performance degradation due to repeated access to the same memory location. The MTA architecture suffers from both read and write hot spotting, due to constraints in traffic across the platform's network. Commodity processors, however, provide cache structures to improve performance and benefit from read hot spotting. Commodity architectures also have a larger granularity of memory sharing: a cache line, which can be as large as 64 bytes. Concurrent writes within a cache line create a hot spot, even if the writes affect independent addresses. The cache was a consideration for atomic operations as well, as they typically cause a cache flush to memory. Avoiding atomic operations where possible, such as in reductions, is important for performance.

6. Multi-platform Graph Algorithms

We consider three graph kernel algorithms: a search, a component finding algorithm, and an algebraic algorithm. There are myriad other graph algorithms, but we use these three as primitive representatives on which other algorithms can be built.

6.1. BFS

Breadth-first search (BFS) is, perhaps, the most fundamental of graph algorithms. Given a vertex v, find the neighbors of v, then the neighbors of those neighbors, etc. Furthermore BFS is well-suited for parallelization. Pseudocode for BFS from [5] is included in Figure 1.

```
BFS(G,s)
1  for each vertex $u \in V[G] - \{s\}$
2      do color[u] $\leftarrow$ WHITE
3         $d[u] \leftarrow$ inf
4  color[$s$] $\leftarrow$ GRAY
5  $d[s] \leftarrow 0$
6  $Q \leftarrow \emptyset$
7  while $Q \neq 0$
8      do $u \leftarrow$ DEQUEUE($Q$)
9         for each vertex $v \in$ Adj[$u$]
10            do if color[v] $\leftarrow$ WHITE
11                  then color[v] $\leftarrow$ GRAY
12                       $d[v] \leftarrow d[u] + 1$
13                       ENQUEUE($Q, v$)
14        color[$v$] $\leftarrow$ BLACK
```

Figure 1. The basic BFS algorithm

There are two inherent problems with using this basic algorithm in a multithreaded environment. The first is that a parallel version of the *for* loop beginning on Line 9 will make many synchronized writes to the *color* array. This is a problem on machines like the Niagara regardless of the data characteristics. It is also a problem on the XMT if there is a vertex v of high in-degree (since many vertices u would test v's color simultaneously, making it a hot spot).

The second problem is even more basic: the ENQUEUE operation of Line 13 typically involves incrementing a tail pointer. As all threads will increment this same location, it is an obvious hot spot.

We avoid these problems by chunking and sorting: suppose that the next BFS level contains k vertices, whose adjacency lists have combined length l. We divide the work of processing these adjacencies into $\lceil l/C \rceil$ chunks, each of size C (except for the last one). Then $\lceil l/C \rceil$ threads process the chunks individually, saving newly discovered vertices to local stores. Each thread can then increment the Q tail pointer only once, mitigating that hot spot. However, in order to handle the *color* hot spot, we do not write the local stores directly into the Q. Rather, we concatenate them into a buffer, sort that buffer with a thread-safe sorting routine (qsort in Qthreads, or a counting sort on the XMT), then have a single thread put the unique elements of this array into the Q. This thread does linear work in serial, but the "hot spot" is now used to advantage in cache-based multicore architectures.

A better BFS algorithm is known for the XMT. Although we currently do not have an implementation of this algorithm, it would be a straightforward exercise to incorporate it into the MTGL so that the same program could run efficiently on either type of platform.

6.2. Connected Components

A connected component of a graph G is a set S of vertices with the property that any pair of vertices $u, v \in S$ are connected by a path. Finding connected components is a prerequisite for dividing many graph problems into smaller parts. The canonical algorithm for finding connected components in parallel is the Shiloach-Vishkin algorithm (SV) [15], and the MTGL has an implementation of this algorithm that roughly follows [1].

Unfortunately, a key property of many real-world datasets will limit the performance of SV in practice. Specifically, it is known both theoretically [7] (for random graphs), and in practice (for interaction networks such, the World-Wide Web, and many social networks) that the majority of the vertices tend to be grouped into one "giant component" (GCC). Algorithms like SV work by assigning a representative to each vertex. Toward the end of these algorithms, all vertices in the GCC are pointing at the same representative, making it a severe hot spot.

We adopt a simple alternative to SV, which we call GCC-SV. It is overwhelmingly likely (though we do not not provide any formal analysis here) that the vertex of highest degree is in the GCC. Given this assumption, we BFS from that vertex using the method of Section 6.1 (or psearch on the XMT), then collect all *orphaned edges* that do not link vertices discovered during this search. Running SV on the subgraph induced by the orphaned edges we find the remaining components. This subproblem is likely to be small enough so that even if the largest component of the induced subgraph is a GCC of that graph (which is likely), the running time is dwarfed by that of the original BFS. If there is no GCC in the original graph, then the original SV would perform well.

6.3. PageRank

```
#pragma mta assert nodep
for (int i=0; i<n; i++) {
    double total=0.0;
    int begin = g[i];
    int end = g[i+1];
    for (int j=begin; j<end; j++) {
        int src = rev_end_points[j];
        double r = rinfo[src].rank;
        double incr = (r/rinfo[src].degree);
        total += incr;
    }
    rinfo[i].acc = total;
}
```

Figure 2. The MTGL code for PageRank's inner loop on the XMT

PageRank, the algorithm made famous by Google for ranking web pages [14], is a linear algebraic technique for modeling the propagation of *votes* through a directed graph, where each page contributes a fraction of its vote to each of its out-neighbors. Ranks continue propagating until convergence. A thorough mathematical explanation of PageRank is beyond the scope of this paper. However, at an abstract level PageRank is a sequence of matrix-vector multiplications, each followed by a normalization step. In graph terms, the most computationally expensive portion of the algorithm is simply traversing all of the adjacencies in the graph in order to accumulate votes.

Figure 2 shows the vote accumulation loops of PageRank used by the MTGL on the XMT. The structure of these loops enables the XMT compiler to merge them into one, and to remove the reduction of votes into the variable `total` from the final line of the inner loop. The result is excellent performance. We simulate this in a Qthread-enabled version of this code in the MTGL in order to achieve good scaling on multi-core machines.

6.4. R-MAT graphs

R-MAT [4] is a parameterized generator of graphs that can mimic real-world datasets. The term stands for "Recursive-MATrix," derived from the generation procedure, which is a simulation of repeated *Kronecker products* [11] of the adjacency matrix by itself. Intuitively, the R-MAT procedure can be thought of as repeatedly dropping marbles through a series of plastic trays. The topmost one typically is divided into 4 quadrants, the second one into 16, etc. The bottom tray is the adjacency matrix. At each level, a marble will pass through one of 4 holes with probability given by 4 input parameters; a, b, c, d. Multiple edges are not allowed, so if a marble ends up on top of another marble in the adjacency matrix, it is discarded and we try again.

Varying the parameters a, b, c, d determines much about the structure of the resulting graph. For example, using $a = 0.25, b = 0.25, c = 0.25, d = 0.25$ would generate an Erdös-Rényi random graph. Putting more weight on one of the quadrants tends to generate an inverse power-law degree distribution, which is found in many real datasets.

In our experiments we generate two different classes of R-MAT graphs:

- *nice* graphs have $a = 0.45, b = 0.15, c = 0.15, d = 0.25$. These graphs feature two natural communities at each of many levels of recursion (quadrants a and d). However, even in graphs a quarter of a billion edges, the maximum vertex degree is only roughly a thousand.
- *nasty* graphs have $a = 0.57, b = 0.19, c = 0.19, d = 0.05$. These feature a much steeper degree distribution, with a maxmimum degree of roughly 200,000 in our quarter-billion edge example. Load balancing would naturally be more challenging in this case.

Furthermore, we label our graphs with the exponent of the number of vertices and hold the average degree at a constant 16, since this is relatively close to (though an over-estimate of) the average degree of a page in the WWW. For example, graph "R-MAT 21 Nasty" has 2^{21} vertices, 2^{24} undirected edges, and R-MAT parameters as given above.

7. Multiplatform Experiments

We compare performance of the three graph kernel algorithms described in Section 6—breadth-first search, connected components, and PageRank—on three platforms capable of executing multiple threads simultaneously: the Cray XMT, the Sun Niagara T2, and a traditional multi-socket, multi-core platform.

The Cray XMT used in testing contains 64 500 MHz ThreadStorm processors, each capable of sustaining 128 simultaneous hardware threads and 500 GB of shared memory. The SeaStar based network is a 3-d torus in a $8x4x2$ configuration. The system was running version 6.2.1 of the XMT operating system.

A Sun SPARC Enterprise T5240 server, with two 1.2 GHz UltraSPARC T2 processors, each capable of sustaining 64 simultaneous hardware threads, was also used in testing. The system contains 128 GB of memory and was running Sun Solaris 10, 5/08 Release. The Sun CoolThreads version of GCC was used to compile all tests.

Finally, a quad-socket, quad-core Opteron system, clocked at 2.2 GHz, provides a traditional multi-core environment. The system provides 32 GB of memory and is running Red Hat EL 5.1. GCC 4.1.2 was used to compile all tests.

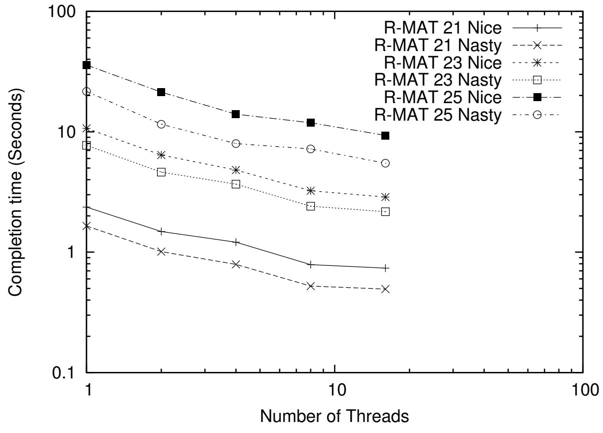


Figure 3. Opteron Breadth-First Search

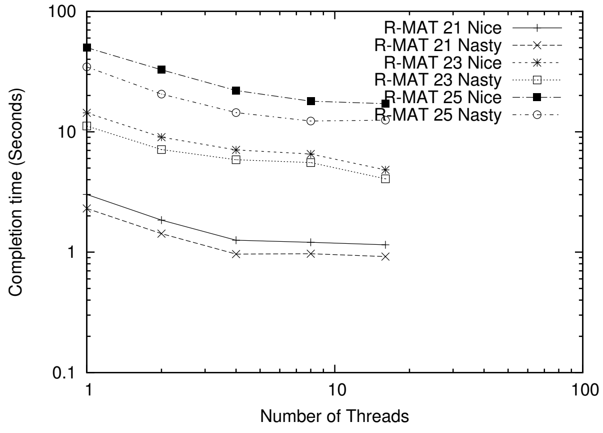


Figure 5. Opteron Connected Components GCC-SV

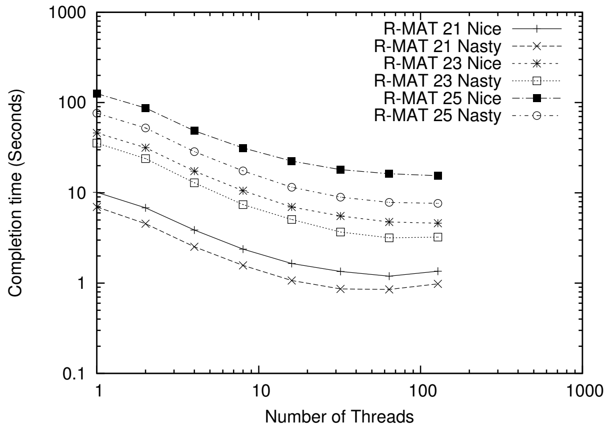


Figure 4. Niagara T2 Breadth-First Search

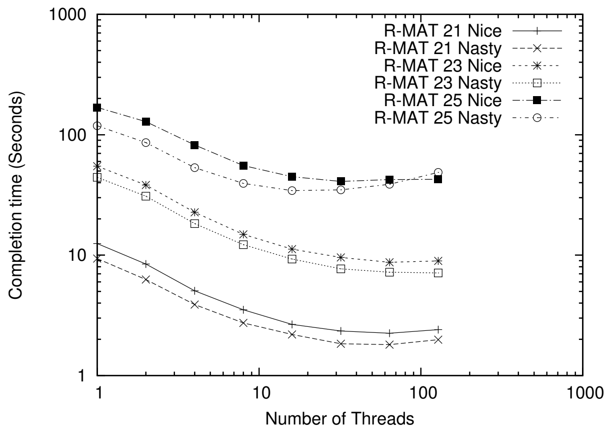


Figure 6. Niagara T2 Connected Components GCC-SV

7.1. Breadth-First Search

We find that our method of avoiding hot spots in BFS enables scaling beyond what would be achievable by a naive algorithm. At the time of this writing, our implementation runs on the XMT, but does not perform as well as native XMT BFS implementations have done in the past. However, our method does leverage the multi-core and Niagara platforms effectively. As implied before, MTGL programmers will run BFS by associating a visitor object with the kernel algorithm, then running the latter. Underlying differences in the kernel implementation, such as that likely in the XMT implementation of BFS, will be hidden from the programmer.

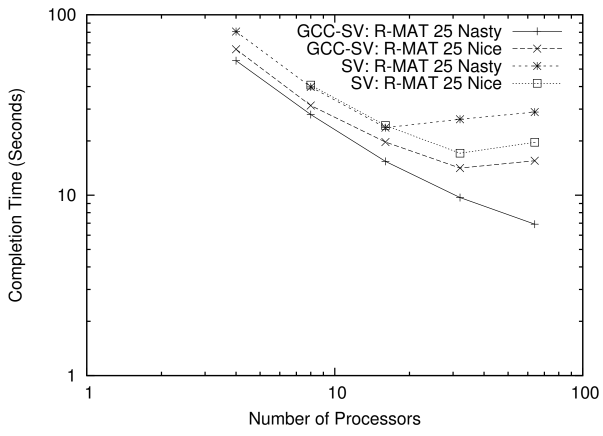


Figure 7. Cray XMT Connected Components - GCC-SV and SV

7.2. Connected Components

Our connected components codes demonstrate strong scaling on multi-core and Niagara, as the GCC-SV algorithm is dominated by a single run of BFS on the realistic datasets we address. Furthermore, we are able to demonstrate strong scaling on the XMT as well by replacing the BFS by the recursive psearch. Note the effect of data on algorithm

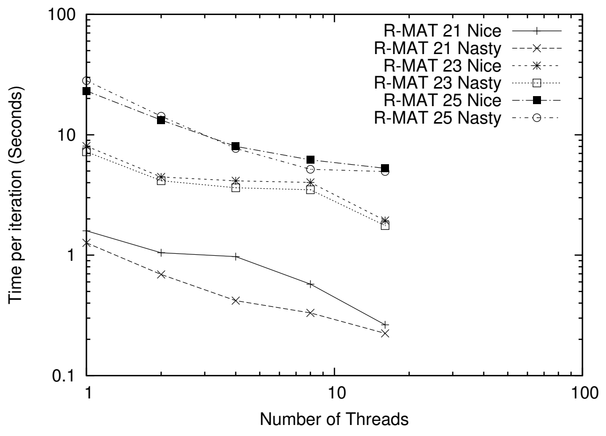


Figure 8. Opteron PageRank

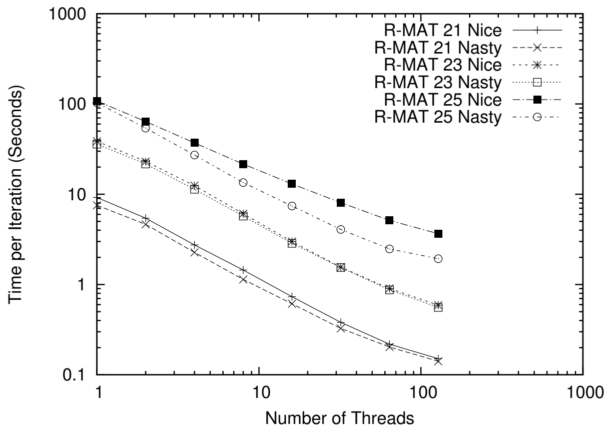


Figure 9. Niagara T2 PageRank

performance in Figure 7. Ironically, the "nasty" datasets are most friendly to the algorithm, as the vast majority of all vertices fall into the GCC in this case. As we consider the "nice" datasets, this GCC membership becomes less pathological (and less realistic). Therefore, the inherently hot spotting SV algorithm has more work to do once the GCC has been processed.

7.3. PageRank

As we saw in Figure 2, PageRank can be written to leverage the auto-parallelizing compiler of the XMT quite effectively. We cannot match the XMT's performance in emulation without work to reconstruct the compiler's optimization. However, a straightforward parallelization of the outer loop using qthreads still provides significant benefit, as we see in Figures 8 and 9.

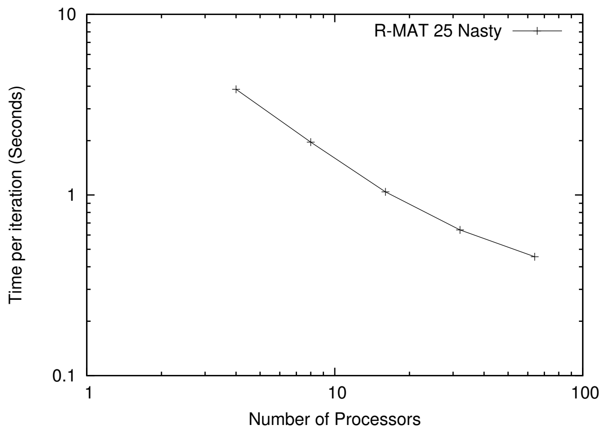


Figure 10. Cray XMT PageRank

8. Conclusions and future work

Developing multi-threaded graph algorithms, even when using the MTGL infrastructure, provides a number of challenges, including discovering appropriate levels of parallelism, preventing memory hot spotting, and eliminating accidental synchronization. In this paper, we have demonstrated that using the combination of Qthreads and MTGL with commodity processors enables the development and testing of algorithms without the expense and complexity of a Cray XMT. While achievable performance is lower for both the Opteron and Niagara platform, performance issues are similar.

While we believe it is possible to port Qthreads to the Cray XMT, this work is still on-going. Therefore, porting work still must be done to move algorithm implementations between commodity processors and the XMT. Although it is likely that the Qthreads-version of an algorithm will not be as optimized as a natively implemented version of the algorithm, such a performance impact may be an acceptable trade-off for ease of implementation.

9. Acknowledgments

The BFS used in this paper leverages a load balancing algorithm to divide adjacencies into chunks. Cynthia Phillips and Jon Berry developed an preliminary version of this algorithm in 2008.

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

References

[1] BADER, D., CONG, G., AND FEO, J. On the architectural requirements of efficient execution of graph algorithms. In

The 33rd International Conference on Parallel Processing (ICPP) (2005), pp. 547–556.

[2] BERRY, J. W., HENDRICKSON, B. A., KAHAN, S., AND KONECNY, P. Software and algorithms for graph queries on multithreaded architectures. In *Proceedings of the International Parallel & Distributed Processing Symposium* (2007), IEEE.

[3] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not. 30*, 8 (1995), 207–216.

[4] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-mat: A recursive model for graph mining. In *In SDM* (2004).

[5] CORMAN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. *Introduction to Algorithms*. MIT Press, 2001.

[6] DAGUM, L., AND MENON, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering 05*, 1 (1998), 46–55.

[7] ERDÖS, P., AND RÉNYI, A. On random graphs I. *Publicationes Mathematicae*, 6 (1959), 290–297.

[8] GREGOR, D., AND LUMSDAINE, A. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)* (July 2005).

[9] HENDRICKSON, B., AND BERRY, J. Graph analysis with high-performance computing. *Computers in Science and Engineering 10*, 2 (2008), 14–19.

[10] INTEL CORPORATION. *Intel®Thread Building Blocks*, 1.6 ed., 2007.

[11] LESKOVEC, J., AND FALOUTSOS, C. Scalable modeling of real graphs using kronecker multiplication. In *In Proceedings of the 24th International Conference on Machine Learning* (2007).

[12] LESKOVEC, J., LANG, K. J., DASGUPTA, A., AND MAHONEY, M. W. Statistical properties of community structure in large social and information networks. In *WWW* (2008), pp. 695–704.

[13] MINAR, N., BURKHART, R., LANGTON, C., AND ASKENAZI, M. The Swarm simulation system: A toolkit for building multi-agent simulations. Working Paper 96-06-042, Santa Fe Institute, 1996.

[14] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.

[15] SHILOACH, Y., AND VISHKIN, U. An o(n log n) parallel connectivity algorithm. *J. Algorithms 3*, 7 (1982), 57–67.

[16] WHEELER, K., MURPHY, R., AND THAIN, D. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium* (April 2008), MTAAP '08, IEEE Computer Society Press, pp. 1–8.

Portable Performance from Workstation to Supercomputer: Distributing Data Structures with Qthreads

Kyle B. Wheeler[1,2], Douglas Thain[2], and Richard C. Murphy[1]

[1]Sandia National Laboratories*
Albuquerque, New Mexico, USA
{kbwheel,rcmurph}@sandia.gov

[2]University of Notre Dame
Computer Science and Engineering
Notre Dame, Indiana, USA
{kwheeler,dthain}@cse.nd.edu

Abstract

Locality and data layout are critical to the performance of threaded parallel programs, but standard threading interfaces incorrectly presume that data is equally accessible from all threads. The qthread library's locality framework has been expanded to provide convenient CPU affinity. This locality support enables development of three locality aware distributed data structures: a memory pool, an array, and a queue. This paper presents benchmark results illustrating the performance characteristics of these data structures on an Altix ccNUMA SMP system, a highly multithreaded Niagara-based server, and a conventional SMP workstation. The combination of locality and threading interface with adaptive distributed data structures provides scalable performance on multiple parallel architectures.

1. Introduction

Data placement is one of the most critical challenges in multicore application performance. Increasingly, multiprocessor machines have non-uniform memory access (NUMA) latencies. Unfortunately, parallel machines have a wide variety of structural differences that impose a large performance penalty on non-optimal data layout [5, 22], so data locality must be exposed for performance.

Standard threading interfaces — such as pthreads [11], OpenMP [20], and Intel Threading Building Blocks (TBB) [12] — are designed for commodity multiprocessing and multicore systems. They provide powerful and convenient tools for developing software tailored to such systems. While these interfaces make it easy to create parallel tasks, they assume that all data is equally accessible from all threads. Because this assumption is generally false, these interfaces do not maintain data and thread locality or provide tools to discover and exploit system topology. Some operating systems provide mechanisms to discover machine topology and to specify thread and memory block locality. Such mechanisms are system-specific and generally non-portable.

Our solution to this problem is the qthread library [23], which integrates locality with the threading interface. The qthread library is a cross-platform threading library that provides lightweight threads, an integrated locality framework, and lightweight synchronization in a way that maps well to developing multithreaded architectures. It supports many different operating systems and hardware architectures and provides a consistent interface to their unique methods of discovering and specifying locality. We expanded its locality framework and added distributed data structures that adapt to NUMA environments. The new distributed data structures hide the complexity of data locality. This paper presents the design of a distributed memory pool, a distributed array, and two forms of distributed queue.

Benchmark results, gathered from several parallel architectures, demonstrate the scalability of these data structures. By comparing operations-per-second and effective bandwidth, the benchmarks show the importance of choosing appropriate system-specific design parameters, such as memory distribution pattern and segment size.

The remainder of this paper is organized as follows. A survey of related work is presented in Section 2. Section 3 summarizes the qthread library and the locality features it provides. Section 4 describes the three architectures used to benchmark the distributed data structures. The design of the data structures, along with benchmark results demonstrating their efficiency, is presented in Section 5. Section 6 suggests future research.

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

Thread Operations

qthread_init(*num_sheps*)
initialize the library

qthread_finalize()
clean up, shut down the library

qthread_fork(*func*, *arg*, *ret*)
spawn a thread

qthread_fork_to(*f*, *a*, *r*, *shep*)
spawn a thread to the specified shepherd

qthread_migrate_to(*shep*)
move the calling thread to the specified shepherd

qthread_distance(*src*, *dest*)
returns the distance between two shepherd IDs

Memory Pools

qpool_create(*item_size*)
create a pool of objects of the specified size

qpool_create_aligned(*i_s*, *align*)
similar to qpool_create(), but returns aligned objects

qpool_alloc(*pool*)
get an object from the pool

qpool_free(*pool*, *addr*)
return an object to the pool

qpool_destroy(*pool*)
deallocates all pool memory

Array Operations

qarray_create(*count*, *unit_size*)
allocate an array, distribute its memory

qarray_create_tight(*c*, *u_s*)
same as qarray_create(), but guarantees item size will not change

qarray_elem(*array*, *index*)
returns a pointer to the specified element in the array

qarray_iter_loop(*array*, *func*, *arg*)
iterate over the array elements in parallel

qarray_destroy(*array*)
deallocate the array

Queue Operations

{qd|qlf}queue_create()
allocate a queue

{qd|qlf}queue_enqueue(*q*, *elem*)
append an element to the the queue

{qd|qlf}queue_dequeue(*queue*)
get the head off the queue

{qd|qlf}queue_empty(*queue*)
check if the queue contains any elements

{qd|qlf}queue_destroy(*queue*)
deallocate a queue

A qlfqueue is a lock-free queue, and a qdqueue is a distributed queue.

Figure 1. Qthread API (abridged)

2. Related work

This research draws from three categories of prior work: lightweight threading models, data structures, and topology interfaces. Cilk [4] and OpenMP [20] are examples of convenient and lightweight threading models. However, they ignore locality, forcing the programmer to rely on the scheduler to keep each thread near the data it is manipulating. Unfortunately, the work-stealing scheduling algorithms these threading models use assume that all tasks can be performed equally well on any processor, which is an invalid assumption on NUMA machines with a high latency variability.

Concurrent vectors, hashes, and queues, such as provided by Intel's Threading Building Blocks [12] are good examples of data structures designed around parallel management operations rather than data operations, ignoring locality. Co-array Fortran [18] provides a parallel container — the co-array — that integrates locality with the execution model, but uses loader-defined static distribution. A distributed queue is a special case of a concurrent pool [15], which is designed for parallel efficiency. Our distributed queue uses locality information to combine Johnson's stochastic queue [13], the push-based queue by Arpaci-Dusseau et. al. [3], and a high-speed lock-free queue based on the work of Michael and Scott [16].

Counterintuitively, threading interfaces tend not to address memory topology. This is probably a result of the historical low variance in memory access latency [17]. Modern and future large-scale shared-memory machines have larger latency variances, and the impact is magnified by increasing CPU speeds. Topology interfaces tend to be operating-system specific. Linux systems largely rely on the libnuma [14] library to exploit system topology. This library presents computers as a set of numbered nodes, CPUs, and "physical" CPUs that overlap. The distance to a node's own memory is normalized to ten, and other distances are expressed on that scale. Without libnuma, Linux processes can define their CPU affinity using the sched_setaffinity () interface. Unfortunately, this interface changes across kernel versions. Software that must work reliably across multiple Linux systems must either detect the available interface variety or use the Portable Linux Processor Affinity (PLPA) library [19], which provides a stable interface. Solaris systems provide the liblgrp library [21], which presents a hierarchical description of topology. Each locality group (lgrp) contains a set of CPUs and/or additional lgrps and is associated with a block of memory. Recent versions of the library can report the latency between lgrps in machine-specific units. All of these interfaces enable thread CPU affinity and libnuma and liblgrp enable memory CPU affinity. Because these interfaces manipulate the operating system's scheduler and memory subsystem, they can only affect things that the operating system schedules, which are inherently heavyweight.

Programming languages such as Chapel [8], X10 [6], Fortress [2], and UPC [9], provide explicit locality as a function of the programming environment. These languages provide expressive semantics and can be effective in exploiting available hardware resources. Unfortunately, they are incompatible with existing software.

3. The qthread library

The qthread library [23] is a cross-platform lightweight threading library with an integrated locality framework and lightweight synchronization that bridges the gap between commodity systems and developing multithreaded architectures. A "qthread" is a nearly-anonymous thread with a small stack that lacks the expensive guarantees and features of heavyweight threads, such as per-thread process identifiers (PIDs), signal vectors, and the ability to be canceled. This lightweight nature supports large numbers of threads with fast context-switching, enabling high-performance fine-grained thread-level parallelism. The API of the qthread library is summarized in Figure 1. The locality framework has been expanded to support thread and data affinity in parallel systems.

Virtually all computer architectures provide atomic op-

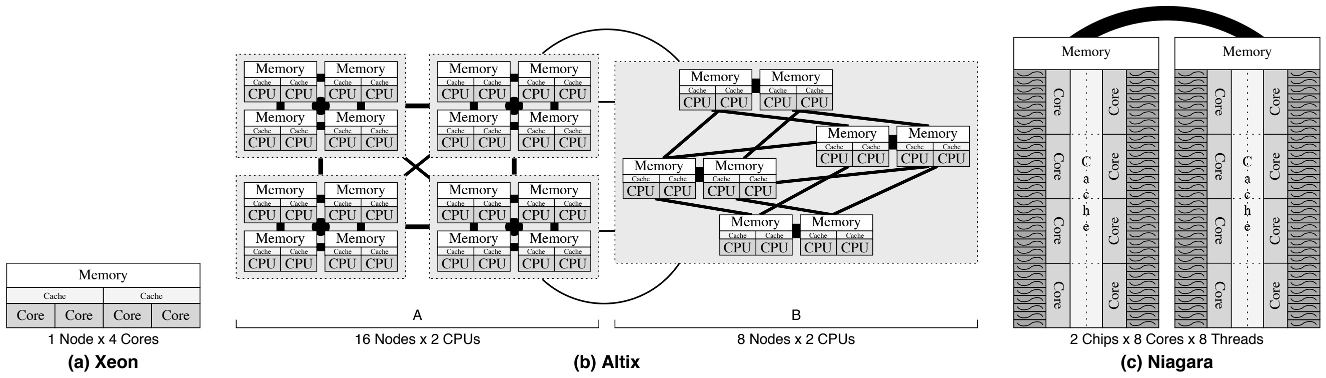


Figure 2. System Topologies

erations, yet they are not directly accessible in most programming interfaces; OpenMP is a notable exception. The qthread library provides an atomic increment via qthread_incr(), and atomic compare-and-swap with qthread_cas(). Data-based blocking synchronization is a powerful way to express task dependencies. The qthread library provides both full/empty bit (FEB) and mutex-like blocking operations. FEB operations are atomic read/writes that depend on per-word full/empty status. Some systems, such as the Cray MTA/XMT, provide hardware support for FEB operations, but usually they must be emulated.

The qthread library uses cooperative-multitasking: blocking operations trigger context switches. These context switches occur in user space via function calls and are thus cheaper than signal-based context switches. This approach allows threads to hide communication latencies and avoid context-switch overhead until unavailable data is needed.

Each qthread has a location, or "shepherd." Shepherds define immovable regions within the system. Qthreads can be spawned directly to a specific shepherd with qthread_fork_to(), and migrate between shepherds with qthread_migrate_to(). Explicit migration guarantees that threads cannot execute in unexpected places. Such control previously required the use of platform-specific libraries and heavyweight threads. The distance between shepherds may be determined using the qthread_distance() function.

4. Parallel architectures

Several different systems with dramatically different topologies are used to demonstrate using topology to inform runtime decisions: a dual-processor dual-core Intel Xeon 5150 workstation, a 48-processor SGI Altix 3700 ccNUMA SMP, and a dual-processor 16-core Sun Niagara 2 server. These machines were selected to demonstrate three different types of parallel systems: a common development workstation, a large ccNUMA system, and a massively multithreaded chip architecture. The topology of each of these machines is illustrated in Figure 2.

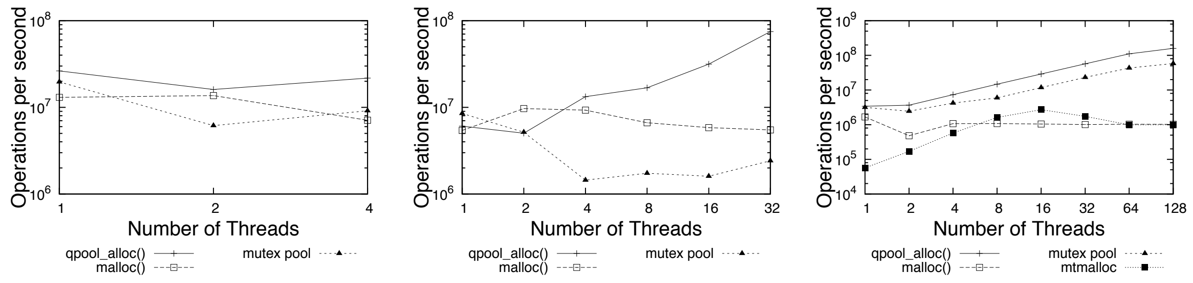
The Xeon system, Figure 2(a), is a dual-core dual-processor. Each processor has its own cache but shares the 1066Mhz front-side bus. This system runs Linux, providing the libnuma interface. Due to the shared bus, libnuma presents this system as a single node with four equidistant processors. This lack of detail is unfortunate, since cache-independence is an important aspect of the memory hierarchy. Without cache, the memory latency is uniform.

The Altix SMP, Figure 2(b), uses Intel Itanium 2 processors. The nodes are connected by dual 3.2 GB/s unidirectional links. Each node has two CPUs and a large block of RAM. The machine is divided into two components: one (A) with 16-nodes the other (B) with 8. Component A is arranged in four clusters of four nodes. Component B's nodes form a dual-plane fat-tree. The two components are asymmetrically connected by additional unidirectional links. The system runs Linux with libnuma.

The Niagara 2 server, Figure 2(c), has two eight-core processors. Each core supports eight concurrent hardware threads, for a total of 128 concurrent hardware threads. Each core has its own bank in the L2 cache which is accessible from any core in that processor. Each cache bank pair shares a dual channel FBDIMM memory controller. This system runs Solaris 10 and supports the liblgrp interface.

5. Distributed data structures

Locality awareness can improve the performance of distributed data structures. Three data structure designs demonstrate this opportunity: a pool, an array, and a queue. These data structure types are cornerstones of application design, and are used in a wide range of high-performance applications. The key feature of these new designs is that they adapt to the topology of the system in use at runtime.



(a) Xeon Allocation **(b) Altix Allocation** **(c) Niagara 2 Allocation**

Figure 3. Multithreaded 44-byte memory allocations/sec, 100-million allocations.

5.1. Distributed memory pool

Memory allocation is a frequently overlooked detail that can significantly impact performance. Standard memory allocation libraries are typically designed for general-purpose allocation in single-threaded applications, balancing allocation speed with limiting fragmentation, without concern for locality. Memory locality is typically specified per-page and needs an abstraction for general-purpose use.

A set of memory pools with mutexes to provide thread-safe access is functional, but suffers from high mutex overhead. The qthread memory pool, qpool, is implemented as a set of location-specific lock-free stacks which provide fast access to local memory. Topology information is used to pull memory from nearby memory pools when necessary.

A qpool is created via qpool_create(). Once created, elements can be fetched from the pool with qpool_alloc() and returned to the pool using qpool_free(). A pool is destroyed by qpool_destroy(). Each allocation is pulled from the local stack. Local memory is allocated in large blocks and portioned out in chunks of the size specified at pool creation. Memory that is freed is pushed onto a local lock-free re-use stack. Upon allocation, this stack is checked first. If the local stack is empty, memory is pulled from the current local large allocated block. When this block is exhausted, the re-use stacks of neighboring shepherd pools are checked, in random order. If none of them have memory, a new large block of memory is allocated from the local node's memory, and added to a list of allocated blocks. If allocation fails, the allocated blocks of all pools are checked, in order of their distance from the requesting thread. If no memory can be found, the allocation request fails.

The graphs in Figure 3 compare the allocations per second of a multithreaded application that allocates, writes to, and deallocates 100 million separate 44-byte memory blocks (the size of a pthread_mutex_t on some systems). Three allocation methods are compared on the Linux systems: the qpool, a similar mutex-protected concurrent memory pool, and standard malloc. The qpool outpaces glibc malloc [10] at scale. The mutex-based memory pools suffer from Linux's slow mutex operations. The malloc implementation, while not returning location-specific memory, uses adaptive arena allocation to scale. The Solaris malloc library, designed for serial applications, is uniformly slow. Solaris's multithreaded mtmalloc library provides better performance for relatively low numbers of threads, but does not appear to be designed for more than sixteen threads and can only allocate blocks in power-of-two sizes.

5.2. Distributed array

The distributed array, or qarray, uses a basic "blocked" design, with node-specific array segments. We examine three aspects of this design: segment distribution pattern, segment size, and element size. The qarray distributes its memory when created, via qarray_create(). Once created, elements within the array can be accessed with qarray_elem(), or using pointer math within a segment. Convenient parallel iteration can be done with qarray_iter_loop(). This mechanism uses a user-specified function to process index ranges, ensuring that the function executes near the range it processes. Arrays are deallocated with qarray_destroy().

The distribution pattern and method of locating segments impacts performance. One option is to place each segment according to its order in the array, via a hash. This has the virtues of simplicity and uniformity, and avoids accessing memory to locate segments, but cannot relocate segments. Alternately, the location of each segment can be stored with the segment. Segments must be accessed to discover their location, but this enables a wider range of distribution patterns and segment relocation. Several options are compared in Figure 4. The "Static Hash" pattern uses the segment order to determine segment locations. The "Dist" patterns all store segment locations within the segments. "Dist Reg Stripes" distributes similarly to the Static Hash, "Dist Rand" distributes randomly, and "Dist Reg Fields" clusters sequential segments evenly. "Serial Iteration" is not a distribution pattern, but serves to compare the distribution patterns with typical non-qarray array iteration.

It is worth noting that parallel iteration scales well on

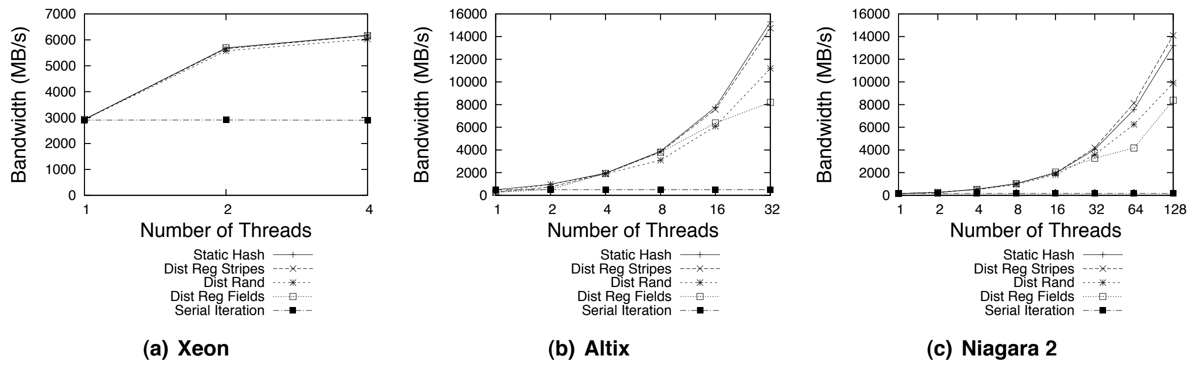


Figure 4. Impact of distribution pattern on multithreaded memory bandwidth over large arrays.

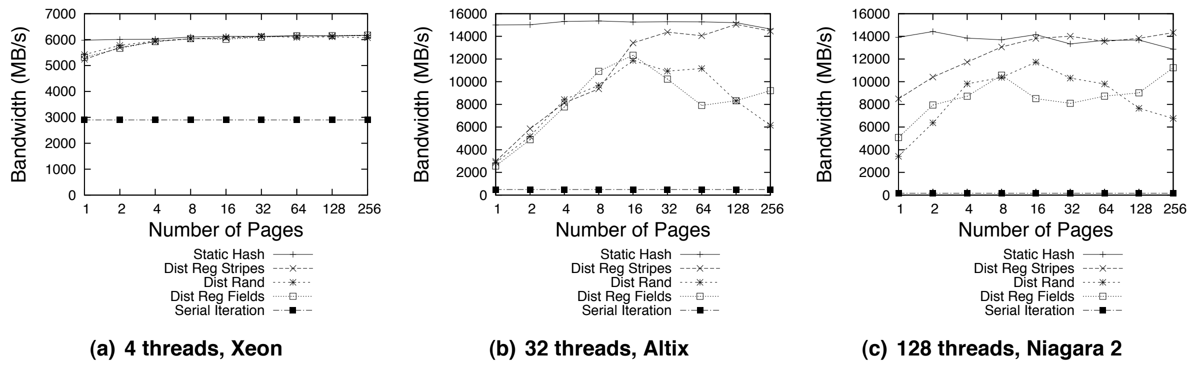


Figure 5. Impact of segment size and distribution pattern on multithreaded memory bandwidth over 100-million element arrays.

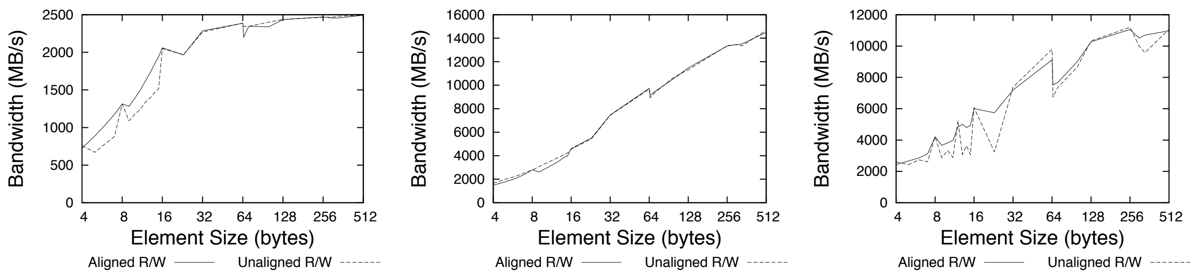
these systems. Iteration with 128 threads was 86.2x faster than serial iteration on the Niagara 2. Iteration with 32 nodes was 31.2x faster than serial iteration on the Altix. The Xeon workstation peaked at 2.1x faster than serial operation, likely because there are only two memory controllers, competing for a relatively low-bandwidth memory bus.

The static hash was the fastest on the Altix and Xeon because accessing remote memory to query its location is slow and causes incorrect prefetching. Thus, static hash is the default distribution method for all but the Niagara 2. The Niagara 2 benefits from fetching "incorrect" blocks because what is incorrect for one thread is likely correct for a nearby thread. Dist Reg Stripes leverages the Niagara 2's shared cache by clustering the working set of memory.

Distributed array performance also depends on the distribution granularity, or "segment size." Most locality-aware memory allocation methods operate on page-size memory blocks, thereby defining the minimum segment size. Thus, only large arrays can be efficiently distributed, and the segment size is a multiple of the page size. Figure 5 illustrates the impact of segment size. Larger segment sizes can either empower prefetching or create load imbalances. Distribution has a significant impact on optimal segment size. The static hash performance varied less than 12% on the Niagara 2 server, while the best performing segment size for Dist Reg Fields provides a 2.2x improvement over the worst performing size. The static hash distribution provided the best small segment size performance of the distribution methods tested; other methods need multiple pages to mask the cost of incorrect prefetching — 16 pages is a good default.

Alignment can be critical to taking full advantage of the cache as well as avoiding unnecessary bus traffic. Caches almost always load aligned data from memory. Accessing unaligned data can result in multiple load instructions, compositing instructions, and even crashes. For example, Figure 6(c) shows that the penalty for using unaligned data can be as high as a 50% reduction in bandwidth. The spikes and variances in bandwidth shown are repeatable, not the result of temporary issues. Avoiding such problems is a task that is often left to the compiler to handle, particularly for global and stack variables. Since a qarray performs layout at runtime, data alignment must be handled manually.

Figure 6 shows the impact of element size and alignment on performance. These graphs compare the memory band-



(a) 4 threads, Xeon **(b) 32 threads, Altix** **(c) 128 threads, Niagara 2**

Figure 6. Impact of alignment on multithreaded memory bandwidth over million-element arrays.

width achieved while accessing million-element arrays with a variety of element sizes. A packed array is compared to an 8-byte aligned array. Manually aligning data has a clear benefit for small element sizes, but has less benefit with large element sizes. This is likely because unaligned layout is more condensed, and that benefit outweighs the penalty for unaligned access beyond a certain size element. Thus, unless otherwise directed, the qarray automatically rounds element sizes under 64 bytes to the next-largest multiple of eight.

5.3. Distributed queue

Parallel queue designs depend on the use-case. If a queue is a buffer between two threads, assumptions can improve speed. For example, assuming low contention for the queue's head and tail, there need only be one of each. Queues are also used for distributing work among multiple threads, with multiple producers and multiple consumers. In that situation, the guarantees of a simpler queue, like global ordering, may be relaxed in order to increase speed. Such a queue may prefer dequeueing nearby elements over the oldest elements in the overall queue.

The qthread library provides both types of queue. The qlfqueue, based on Michael and Scott's lock-free queue [16], guarantees global ordering. The end-to-end ordered queue is the qdqueue. New qlfqueues are created with qlfqueue_create() and destroyed with qlfqueue_destroy(). Elements may be queued with qlfqueue_enqueue() and dequeued with qlfqueue_dequeue(). A quick element check may be performed with qlfqueue_empty(). Equivalent qdqueue functions begin with "qdqueue" instead of "qlfqueue."

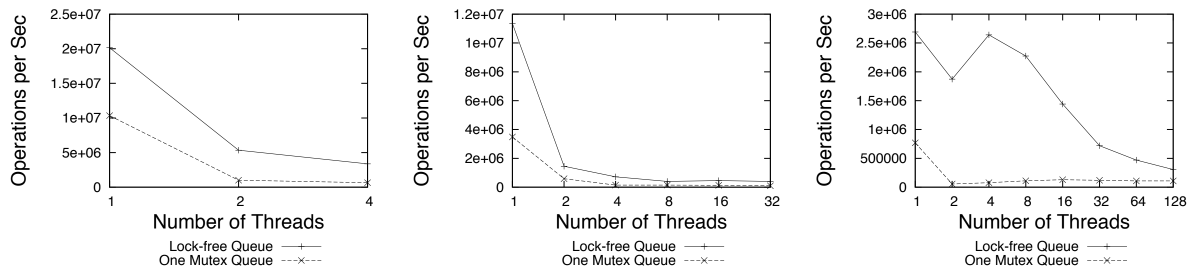
Figure 7 illustrates the effect of strict ordering on the scalability of a queue by comparing the lock-free qlfqueue to a single-mutex queue implementation from the cprops library [1]. Ensuring global ordering requires a serialized critical section. A lock-free queue is faster than a mutex-based queue because it uses hardware assistance to minimize ordering overhead. Nevertheless, this serialization acts as a bottleneck that precludes scaling.

The core of an end-to-end ordered queue is matching consumers to producers. A central matchmaker is a simple approach, but creates a bottleneck. Hierarchical matchmaking reduces consumer contention, but increases the work of producers without addressing their bottleneck issues. A "stochastic distributed queue" [13], where consumers repeatedly probe random producer queues until satisfied, is efficient when the queues are rarely empty. But if queues are frequently empty, random polling creates unnecessary work. Random polling can be avoided with advertisements.

Per location, the qdqueue uses a qlfqueue, a list of "ads" received, and a record of the last consumed element's source. Elements are enqueued locally. If the queue was not empty, ads are posted to nearby queues. An ad informs remote consumers that there is data available in this queue. The set of "neighbor" shepherds to receive advertisements from any given shepherd is determined at setup time, based on distance. Thus, the maximum work of a producer is fixed, independent of the size of the system. A producer can avoid resending ads by tagging them with a counter and tracking the highest consumed ad counter value. If the last-issued ads have not been consumed, ads need not be re-issued.

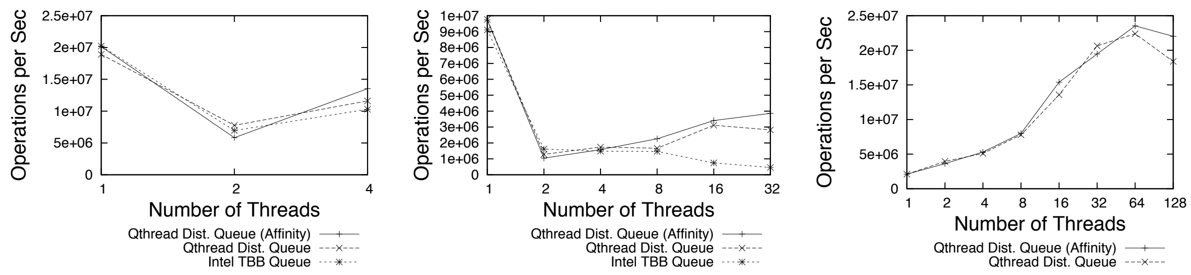
To dequeue, the local queue has priority. If the local queue was empty, any known ads are checked in order of distance. The source of the last-dequeued element is recorded locally. When responding to an ad, update the advertiser's record of ads consumed. If none of the ads result in an element, it is necessary to check *all* remote queues, in the order of distance from the consumer. If a remote queue is empty, that queue's last-consumed record is treated as an ad. If an ad is received while checking remote queues, it is checked immediately. Thus, consumers cooperate to find elements. If all remote queues are empty, the consumer must either return empty-handed or wait for an ad to be posted.

Figure 8 compares the scaling of the qdqueue with the TBB concurrent queue, which have similar ordering guarantees. The benchmark, in both cases, transfers word-size data using a variable number of enqueueing threads with an identical number of dequeueing threads. The qdqueue can use CPU pinning — labeled "(Affinity)." In single-



(a) Xeon **(b) Altix** **(c) Niagara 2**

Figure 7. The ops/sec of strictly ordered multithreaded queues.



(a) Xeon **(b) Altix** **(c) Niagara 2**

Figure 8. The ops/sec of end-to-end ordered multithreaded queues with small elements.

threaded mode the qdqueue is just as fast as the qlfqueue, but improves on the qlfqueue's multi-thread performance. At scale, the qdqueue performed 9.6x better than the qlfqueue on the Altix, 72x better on the Niagara 2, and 4x better on the Xeon. The Niagara 2's performance is a result of cheap communication via shared cache. Cache-coherency penalizes shared information on the Altix and Xeon. Figure 9, illustrates the impact of using larger (1024-byte) blocks of data. Large inter-node transfers impose a higher penalty than small ones, magnifying the benefit of location affinity.

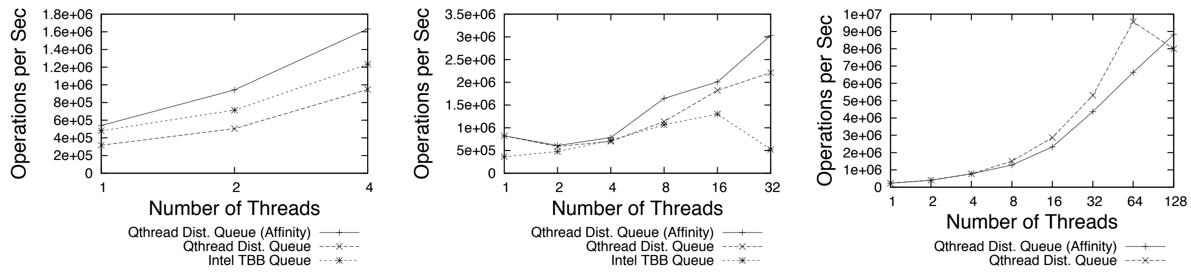
6. Future work

There are many additional array operations common in scientific computing that benefit from locality-awareness. Array stencils are common in signal processing, image processing, and solving partial differential equations. Foreknowledge of the stencils can be used to influence distributed array layout, such as by aligning stencil and segment boundaries and avoiding unnecessary thread spawns. Array combination, such as in string comparison, genetic research, matrix multiplication, and multi-dimensional arrays, are also common. These array combination operations have well-understood memory access patterns that provide opportunities for locality-aware optimization. A key question is where to best execute operations with disperse input.

MapReduce [7] is a powerful asynchronous and easily pipelined data-processing abstraction popularized by Google. It could be implemented with distributed queues instead of the usual "master" node approach, naturally preserving locality between stages. This design may be useful for exploring the best number of workers at each stage, the optimal information transfer method, and the effect of worker loss and worker migration.

7. Conclusion

Developing portable locality-aware data structures, even with the advantage a threading library with integrated locality, is a significant challenge. We have presented a portable threading interface that integrates locality, several data structure designs that use locality information, and examined the selection of optimal system-specific design parameters. We have demonstrated the effectiveness of locality-aware design in distributed data structures. Memory pools can be up to 155 times faster than traditional `malloc()` while providing location-specific memory. The qarray distributed array design supports strong-scaling on large NUMA systems, executing 31.2 times faster with 32 nodes than with one. The qdqueue distributed queue design provides up to a 47x improvement over a fast lock-free queue by providing only end-to-end ordering. Locality-awareness



(a) Xeon **(b) Altix** **(c) Niagara 2**

Figure 9. The ops/sec of end-to-end ordered multithreaded queues with large (1024 byte) elements.

provides up to an 8.3x improvement in performance over the state-of-the-art concurrent queues on a large NUMA system. Importantly, the use of locality information does not significantly impact serial performance or performance on small systems with uniform memory access latencies. Exposing hardware non-uniformity is becoming common, and the unification of thread handling and topology is an important direction for future shared-memory data structure research.

References

[1] I. Aelion. cprops - C prototyping tools. http://cprops.sourceforge.net, March 2009.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification v. 1.0β*. Sun Microsystems, 2007.

[3] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proc. of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 10–22, New York, NY, 1999. ACM.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, 1995. ACM.

[5] J. Chapin, A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proc. of the 1995 ACM SIGMETRICS Joint International Conf. on Measurement and Modeling of Computer Systems*, pages 1–13, New York, NY, 1995. ACM.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, 2005. ACM.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of the 6th Symp. on Operating Systems Design & Implementation*, page 10, Berkeley, CA, 2004. USENIX Assoc.

[8] R. Diaconescu and H. Zima. An approach to data distributions in chapel. *International Journal of High Performance Computing Applications*, 21(3):313–335, 2007.

[9] T. El-Ghazawi and L. Smith. Upc: unified parallel c. In *Proc. of the 2006 ACM/IEEE Conf. on Supercomputing*, page 27, New York, NY, 2006. ACM.

[10] W. Gloger. Dynamic memory allocator implementations in Linux system libraries. http://www.dent.med.uni-muenchen.de/ wmglo/, May 1997.

[11] Institute of Electrical and Electronics Engineers. *IEEE Std 1003.1-1990: Portable Operating Systems Interface (POSIX.1)*, 1990.

[12] Intel Corp. *Intel® Threading Building Blocks v. 1.6*, 2007.

[13] T. Johnson. Designing a distributed queue. In *Proc. of the 7th IEEE Symp. on Parallel and Distributed Processing*, pages 304–311, Washington, DC, 1995. IEEE Comp. Soc.

[14] A. Kleen. An NUMA API for Linux. http://halobates.de/numaapi3.pdf, August 2004.

[15] U. Manber. On maintaining dynamic information in a concurrent environment. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 273–278, New York, NY, 1984. ACM.

[16] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th Ann. ACM Symp. on Princ. of Dist. Comput.*, pages 267–275, New York, NY, 1996. ACM.

[17] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and D. Ayguadé. Is data distribution necessary in OpenMP? In *Proc. of the 2000 ACM/IEEE Conf. on Supercomputing*, page 47, Washington, DC, 2000. IEEE Comp. Soc.

[18] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[19] Open MPI Team. Portable Linux processor affinity. http://www.open-mpi.org/projects/plpa/, March 2009.

[20] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 2.5 edition, May 2005.

[21] Sun Microsystems, Inc., Santa Clara, CA. *Memory and Thread Placement Optimization Developer's Guide*, 2007.

[22] J. Tao, W. Karl, and M. Schulz. Memory access behavior analysis of NUMA-based shared memory programs, 2001.

[23] K. Wheeler, R. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proc. of the 22nd IEEE International Parallel & Distributed Processing Symp.* IEEE Comp. Soc., April 2008.

DISTRIBUTION:

1	MS 1322	John Aidun, 1435
1	MS 1319	Jim Ang, 1422
1	MS 1319	Brian Barrett, 1423
1	MS 1319	Ron Brightwell, 1423
1	MS 1320	Scott Collis, 1414
1	MS 1319	Doug Doerfler, 1422
1	MS 1322	Sudip Dosanjh, 1420
1	MS 1319	K. Scott Hemmert, 1422
1	MS 1318	Bruce Hendrickson, 1410
1	MS 9151	Howard Hirano, 8960
1	MS 9158	Curtis Janssen, 8961
1	MS 1319	Sue Kelly, 1423
1	MS 0801	Rob Leland, 9300
1	MS 0321	John Mitchner, 1430
1	MS 0321	James Peery, 1400
1	MS 1316	Danny Rintoul, 1409/1412
1	MS 1319	Arun Rodrigues, 1423
1	MS 0822	David Rogers, 1424
1	MS 1318	Suzanne Rountree, 1415
1	MS 1318	David Womble, 1540
2	MS 9018	Central Technical Files, 8944
2	MS 0899	Technical Library, 4536

Sandia National Laboratories