SANDIA REPORT

SAND2009-5574
Unlimited Release
Printed September 2009

Improving Performance via Mini-applications

Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, Robert W. Numrich

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

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-5574
Unlimited Release
Printed September 2009

Improving Performance via Mini-applications

Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring
Computer Science Research Institute
Sandia National Laboratories
Albuquerque, NM 87175
{maherou,dwdoerf,pscrozi,jmwille}@sandia.gov

H. Carter Edwards, Alan Williams, Mahesh Rajan,
Eric R. Keiter, Heidi K. Thornquist
Joint Computational Engineering Laboratory
Sandia National Laboratories
Albuquerque, NM 87175
{hcedwar,william,mrajan,erkeite,hkthorn}@sandia.gov

Robert W. Numrich
Minnesota Supercomputing Institute
University of Minnesota
Minneapolis, MN 55455
rwn@msi.umn.edu

Abstract

Application performance is determined by a combination of many choices: hardware platform, runtime environment, languages and compilers used, algorithm choice and implementation, and more. In this complicated environment, we find that the use of mini-applications–small self-contained proxies for real applications–is an excellent approach for rapidly exploring the parameter space of all these choices. Furthermore, use of mini-applications enriches the interaction between application, library and computer system developers by providing explicit functioning software and concrete performance results that lead to detailed, focused discussions of design trade-offs, algorithm choices and runtime performance issues. In this paper we discuss a collection of mini-applications and demonstrate how we use them to analyze and improve application performance on new and future computer platforms.

Acknowledgment

The authors thank the Department of Energy (DOE) LDRD program, the Institute for Algorithms and Architectures, and the DOE ASC program for funding this research.

Contents

List of Figures

Introduction

Production-quality science and engineering applications are typically large, complicated and full-featured software products. As a result, they tend to be challenging to port to new computer platforms and require a well-trained user to do so. Although benchmarking of these applications on new platforms is essential as part of the design and implementation of a new computer system, the scope of this benchmarking is necessarily limited by the complexity of the software product, not to mention its demand for a full scope of system features that are only available after a new computer system reaches its near-production capabilities.

Characteristics that impact performance should be understood as early as possible in the analysis and design of new computers. Furthermore, it is often the case that there are multiple ways to design and implement the algorithms used in an application, and the choice can have a dramatic impact on application performance.

To address these needs, our recent work in application performance analysis takes advantage of two important properties of many applications. (i) Although an application may have one million or more source lines of code, performance is often dominated by a very small subset of lines. (ii) For the remaining code, these applications often contain many physics models that are mathematically distinct but have very similar performance characteristics.

To exploit the properties listed above, we have developed a growing collection of mini-applications (called *miniapps* for the remainder of the paper). Miniapps take advantage of the above two application properties by encapsulating only the most important computational operations and consolidating physics capabilities that have the same performance profiles. The large-scale application developer, who is tasked with developing the miniapp, guides the decisions, resulting in a code that is a small fraction of the original application size, yet still captures the primary performance behavior.

All of the work presented here is done as part of the Mantevo project [14], a project focused on developing tools to accelerate and improve the design of high performance computers and applications by providing application and library proxies to the high performance computing community.

The Role of Miniapps

There are many benchmarking efforts for scientific computing. The Top 500 High Performance Linpack (HPL) [12] and the HPC Challenge benchmark suite [18] are among the most popular. In addition, full-scale applications are often used for performance analysis, but usually on near-production systems. Between these two extremes there is a middle ground for small, self-contained programs that, like benchmarks, contain the performance-intensive computations of a large-scale application, but are large enough to also contain the context of those computations. The NAS Parallel Benchmarks [8] fall into this category and

have been commonly used, as have the compact or synthetic applications developed as part of the Department of Defense High Performance Computing Modernization Program [10]. SWEEP3D [16] also fits this category.

Despite this broad collection, we have found that there is room for many more miniapps. In fact, as we have progressed in this work, we have determined that any high-performance application project can benefit from having a miniapp that represents the performance-intensive aspects of the application. The availability of this kind of proxy greatly enhances the ability to study and improve application performance. Miniapps provide a category of tools that help in the following situations:

- **Interaction with external research communities:** Miniapps are open source software, in contrast to many production applications that have restricted access.
- **Simulators:** Miniapps are the right size for use in simulated environments, supporting study of processor, memory and network architectures.
- **Early node architecture studies:** Scalable system performance is strongly influenced by the processor node architecture. Processor nodes are often available many months before the complete system. Miniapps provide an opportunity to study node performance very early in the design process.
- **Network scaling studies:** Miniapps are easily configured to run on any number of processors, providing a simple tool to test network scalability. Although not a replacement for production applications, miniapps can again provide early insight into scaling issues.
- **New language and programming models:** Miniapps can be refactored or completely rewritten in new languages and programming models. Such working examples are a critical resource in determining if and how to rewrite production applications.
- **Compiler tuning:** Miniapps provide a focused environment for compiler developers to improve compiled code.

Miniapp Development Process

There have been many efforts to develop performance proxies for large-scale applications. Some efforts have started with the original application and cut out code that was not necessary for performance analysis. In related Mantevo project work, we have developed light-weight drivers (called minidrivers) to exercise production libraries in a way that focuses on performance issues. For miniapps we have found the following approach to work best.

Miniapp Development Resources

Miniapps are not just stripped down versions of large-scale applications or large benchmarks. A useful miniapp requires a good understanding of the class of applications it is intended to represent. As a result, we have found that the best miniapp developers are

the same people who develop the large-scale application. In fact, all of our miniapps are written by application developers, who set aside part of their time to develop and maintain their miniapp. These developers have come to view the miniapp as an essential part of their application project.

Miniapp Properties

Miniapps are intended to be self-contained, stand-alone codes. We have found that a simple makefile and instructions for configuring and building are more effective than a complex build environment. This is especially true when working in a simulator or other early design environment.

By keeping the code and build environment simple we have found that a variety of system researchers and benchmarkers can understand the basic anatomy and behavior of the miniapp and can even get insight into the performance characteristics of the corresponding large-scale application.

Data Generation and Cataloging

Although each miniapp is independently developed, we have found value in using a common output format for the purposes of collecting and analyzing data. There are many formats from which we can choose, but we have found that YAML [5] provides both a human readable form and the ability to process data into XML format or store it into a database. A related project called Mantevo Views [11] is focused on scanning YAML data to automatically generate database tables from YAML structures and reading a collection of compatible YAML results and analyzing them.

Overview of Current Miniapps

All of the miniapps discussed in this section are part of the Mantevo project. Each miniapp is available via the GNU Lesser General Public Licence (LGPL) [1] and is downloadable from the Mantevo website [14].

MiniFE: Implicit Finite Elements

Many engineering applications require the implicit solution of a nonlinear system of equations where the vast majority of time–as problem size increases–is spent in some variation of a conjugate gradient solver. As a result, any miniapp focusing on this area will necessarily have a conjugate gradient solver as the dominant computational kernel.

MiniFE (also known as HPCCG) is a miniapp that mimics the finite element generation, assembly and solution for an unstructured grid problem. The physical domain is a 3D box with configurable dimensions and a structured discretization (which is treated as unstructured). The domain is decomposed using a recursive coordinate bisection (RCB) approach and the elements are simple hexahedra. The problem is linear and the resulting matrix is symmetric, so a standard conjugate gradient algorithm is used with a general sparse matrix data format and no preconditioning.

This simple code–which is not intended to be a true physics problem–is sufficiently realistic for performance purposes. Furthermore, it contains approximately 1,500 lines of C++ code. MiniFE is written using C++ templates to support a variety of floating point and integer data, e.g., 32-bit and 64-bit variants. The RCB partitioning will provide a nearly perfect load and communication balance for a homogeneous problem definition, but MiniFE contains tuning parameters that can gradually increase work and communication imbalance for the purposes of studying scalability of competing computer systems.

Because of its small size and simplicity, MiniFE has been refactored and rewritten numerous times using OpenMP, CUDA, Qthreads [3], BEC [9] and the Trilinos Thread Pool Interface (Trilinos/TPI).

MiniMD: Molecular Dynamics

The MiniMD application is miniature version of the molecular dynamics (MD) application LAMMPS [28, 27, 2]. The source for MiniMD is less than 3,000 lines of C++ code. Like LAMMPS, MiniMD uses spatial decomposition MD, where individual processors in a cluster own subsets of the simulation box. And like LAMMPS, MiniMD enables users to specify a problem size, atom density, temperature, timestep size, number of timesteps to perform, and particle interaction cutoff distance. But compared to LAMMPS, MiniMD's feature set is extremely limited, and only one type of pair interaction (Lennard-Jones) is available. No long-range electrostatics or molecular force field features are available. Inclusion of such features is unnecessary for testing basic MD and would have made MiniMD much bigger, more complicated, and harder to port to novel hardware. The current version of LAMMPS includes over 130,000 lines of code in hundreds of files, nineteen optional packages, over one hundred different commands, and over five hundred pages of documentation. Such a large and complicated code is not ideally suited for answering certain performance questions or for tinkering by non-MD-experts.

A rewrite of the entire LAMMPS code base would be a daunting task, but a massive overhaul of MiniMD to test a new idea can be achieved fairly quickly. We have used MiniMD to test several MD software performance questions and ideas. One such idea featured changing MiniMD to single precision to investigate how much that would enhance performance. It was somewhat surprising to us that there was no appreciable performance enhancement on the typical CPU hardware that we tested. Future testing on other architectures may prove more interesting.

MiniMD has also been used to test the scaling performance of the spatial decomposition algorithm as the number of processors increased towards infinity. It was found that the fraction of time spent on computation did not approach unity (the fraction of time spent on communication did not approach zero). This finding demonstrated a limitation of the spatial decomposition algorithm for performing MD [25].

We intend to use MiniMD to test future ideas for enhancement of basic MD software performance on compute platforms that become available to us. The most useful ideas can then be migrated into LAMMPS for the benefit of the broader user community.

phdMesh: Contact Detection

Contact detection has been a performance-critical algorithm for parallel explicit dynamics simulation codes for over a decade [7]. In explicit dynamics simulations with large deformations each facet in an unstructured mesh may come into geometric proximity, and subsequently into contact, with any other facet in the mesh. These proximity conditions must be detected to support subsequent contact mechanics computations.

The parallel geometric proximity search algorithm consists of the following steps. (i) Partition the problem domain's geometric space among processors. In this step the objective is to generate a well-defined geometric domain decomposition that will load balance the number of facets within each subdomain. (ii) Communicate facet information from the originating processor to the processor(s) designated by the geometric domain decomposition. (iii) Perform an on-processor geometric proximity search within each geometric subdomain. (iv) Communicate facet-facet proximity results of the on-processor geometric proximity search back to the processors on which the facets originated. These results are used to duplicate off-processor facet data on one (or both) of the facets' processors to support subsequent on-processor contact mechanics computations.

The parallel geometric proximity search algorithm utilizes a combination of parallel reduce-to-all communications and problem-specific sparse all-to-all communications. Parallel scalability of the algorithm has been especially challenging [6] due to (i) having different parallel domain decompositions for the unstructured mesh and the geometric space and (ii) the all-to-all geometric search among facets. Numerous all-to-all geometric search algorithms have been developed with $N \log(N)$ performance (where N is the number of facets) as opposed to the naive N^2 algorithm; however, few of these geometric search algorithms are suitable for distributed memory parallel implementations.

The parallel heterogeneous dynamic mesh (phdMesh) is a library in Trilinos [13] that provides an in-memory mesh and field database for parallel, heterogeneous, dynamic, unstructured meshes. This library includes a parallel implementation of an oct-tree geometric proximity detection algorithm [17] with the state-of-the-practice $N \log(N)$ complexity. The phdMesh library and oct-tree geometric search algorithm are integrated to form a parallel geometric proximity search miniapp. This miniapp generates an unstructured mesh for a

grid of simple counter-rotating gears that have continuously changing contact conditions (Figure 1). The miniapp then runs the geometric proximity detection algorithm for the surface-facets of these gears.

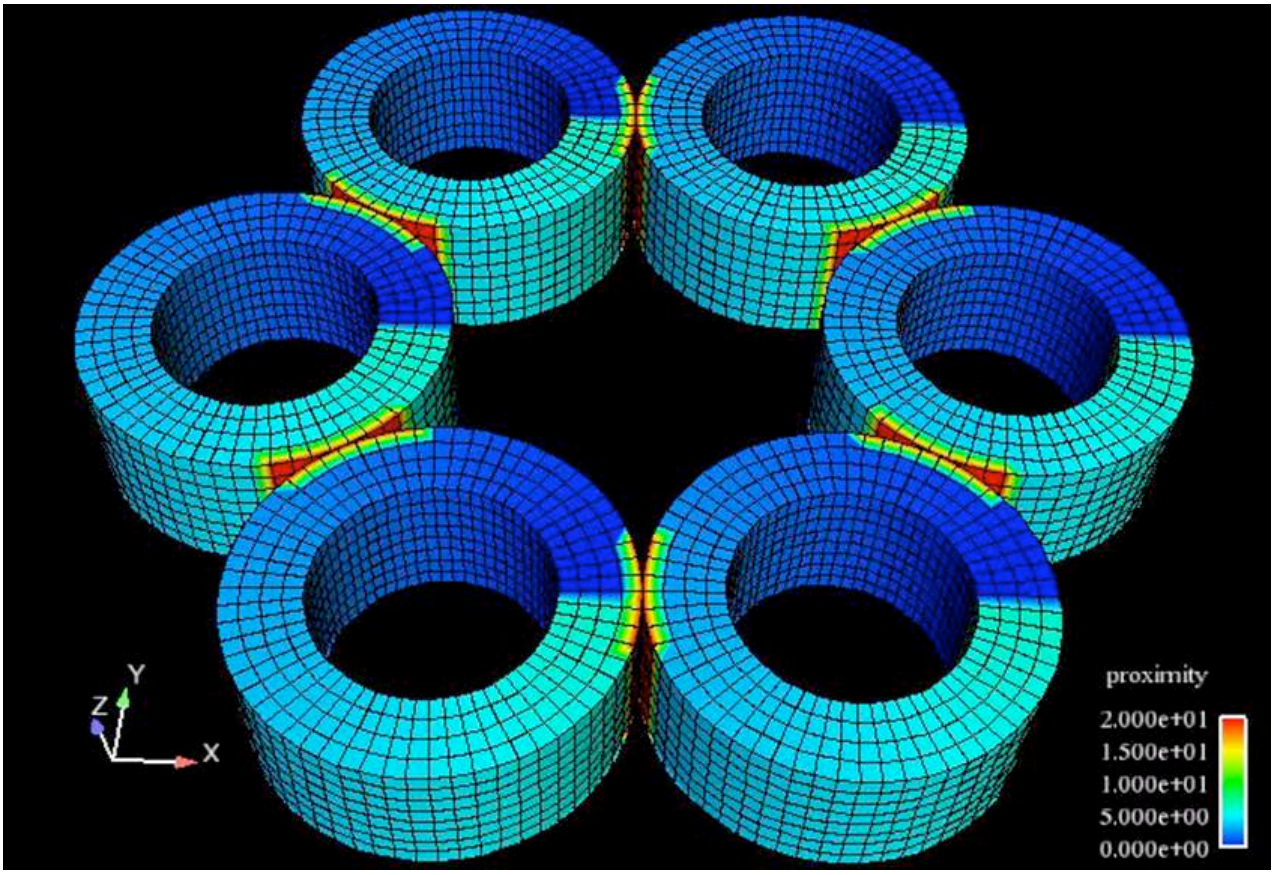

Figure 1: phdMesh miniapp scaling performance test case–a grid of counter-rotating gears.

The phdMesh miniapp provides a compact, self-contained, and portable code to assess performance and scalability of the performance-critical parallel geometric proximity search algorithm. The phdMesh library provides a full-capability parallel, unstructured mesh and field data structure including dynamic load balancing. This supports accurate performance assessment of communicating and manipulating the unstructured mesh, and the performance impact of the alignment between the original mesh domain decomposition and the geometric domain decomposition.

MiniXyce: Electrical Circuits

The MiniXyce application is a miniature version of Xyce [15], a circuit simulation application. Circuit simulation is the cornerstone of the electrical design automation (EDA) industry, and is a crucial part of commercial electrical design. Like most circuit simula-

tion tools, MiniXyce is based on a modified nodal analysis (MNA) formulation, resulting in Kirchoff Current Laws (KCL) being enforced across a potentially arbitrary network. The resulting system of differential-algebraic equations (DAEs) is solved implicitly using Newton-based methods. Traditional circuit codes have almost exclusively relied upon direct matrix solvers, but preconditioned GMRES is the method of choice for parallel simulation.

The network structure of circuits means that the parallel decomposition for MiniXyce is not based on spatial relationships. For example, it is common for digital circuits to have highly connected nodes, such as those connected to bus and clock elements, which directly drive components distributed throughout the entire circuit. This lack of locality poses unique problems for matrix solution. Circuit matrices tend to be ill-conditioned and are often non-SPD.

Xyce, the original code upon which MiniXyce is based, consists of over 500,000 lines of C++ code. However, much of the source is required to support capabilities that are not needed for MiniXyce. For example, the Xyce input file parser is very complicated, supporting user-defined expressions, hierarchical subcircuiting, as well as the physics (compact device) model library. For large circuit simulations, the input file itself can be so large as to exceed the memory constraints of a single processor. As a result, it is necessary for Xyce's parsing to be conducted in parallel.

In addition to IO parser support, a large fraction of the Xyce source is devoted to the library of device models. In circuit simulation, device models are used to enforce KCL equations by applying Ohmic relationships of discrete electrical components to branches of the circuit graph. Typical examples of such components include transistors, diodes, resistors, and capacitors. While some device models, such as the resistor, are quite simple, modern transistor models can be extremely complex. It is common for modern CMOS based transistor models to consist of over 10,000 lines of C/C++ code.

For MiniXyce, both source code burdens (IO and device models) can be avoided or mitigated. The approach taken for MiniXyce is based on the following ideas. (1) Most large circuits that could benefit from parallel computing methods will be CMOS integrated circuits. (2) CMOS integrated circuit designs can be divided into a few general categories and/or building blocks, such as memory, PLL, ADC, DAC, power grids and multipliers. As such, only a handful of device models are necessary: resistor, capacitor, voltage source, and a simplified MOSFET model. Additionally, a traditional circuit parser is not necessary, as the connectivity structure of many building blocks can be hard-coded with repeated unit cells. Realistic circuits will have more variability than can possibly be represented with such an approach, but this should be sufficient to investigate performance and scalability.

The conception and development of each general circuit category for MiniXyce is a valuable exercise in and of itself. For example, it has been observed, empirically, that circuits with feedback (such as PLL's) are much more difficult to solve using iterative methods than circuits that are unidirectional. While this is a fairly intuitive supposition, it bears further study. The development of miniapps provides a set of tools which can be used to investigate this issue in detail.

Prolego: A Configurable Miniapp

In addition to application-specific miniapps, we have invested in an alternative approach that uses a collection of code fragments that can be composed and calibrated to mimic a target application. This package is called Prolego.

Prolego contains a collection of software fragments or kernels that can be composed at run-time using an XML input-file specification. These fragments represent performance-dominating pieces of target applications. The idea is that by selecting an appropriate set of fragments and giving them appropriate weighting, a benchmark can be calibrated to accurately represent an arbitrary target application.

As a simple example, consider a Krylov subspace solver such as linear Conjugate Gradients (CG). Three linear algebra kernels dominate CG performance: (i) matrix-vector product, (ii) collective operations (inner products and norms) and (iii) vector updates. Thus a benchmark consisting of those three kernels with appropriate weights and data sizes can be calibrated to match the performance of a conjugate gradient solve, even though the benchmark is not actually solving a linear system.

A more complicated application is of course harder to represent with great accuracy. Qualitative performance characteristics such as computational order of complexity are usually representable with a small number of fragments, and then the benchmark can be fine-tuned by adding more fragments to represent more detail as required. In this way we can model kernel performance and also performance coupling between kernels where temporal data locality is important.

Miniapp Usage

Here we discuss how miniapps have been used to study and improve performance. The purpose of this section is not to thoroughly explore any particular issue–that is reserved for other papers–but to illustrate the variety of ways these miniapps provide value to performance analysis activities.

Multicore Node Studies

Multicore nodes–including GPUs, Cell and soon manycore variants–are arguably the biggest architecture change for high performance computing in more than a decade. Early performance results are extremely important for planning and preparation in application development efforts. Mantevo miniapps have been used extensively to study a variety of multicore performance issues. Here we present three studies: (i) 8-core performance on four commodity dual-socket quadcore processors. (ii) MiniFE performance on 1 to 8 cores and (iii) the performance impact of memory placement on a NUMA memory system.

Figure 2 shows the performance of three miniapps on four commodity microprocessors. For this study, the Intel Clovertown results are used as the normalizing factor. The AMD Barcelona processor is representative of Sandia's Red Storm platform. The AMD Shanghai is the follow-on to Barcelona, and in this test increases memory bandwidth from 667 MHz DDR2 to 800 MHz DDR2. Both processors have integrated memory controllers with two 8 byte wide memory channels. Nehalem is Intel's latest workstation processor and is Intel's first processor to use an integrated memory controller as opposed to a front side bus. Nehalem provides significantly more memory bandwidth than the AMD processors via three 8 byte wide channels of 1066 MHz DDR3. These results imply that phdMesh is less sensitive

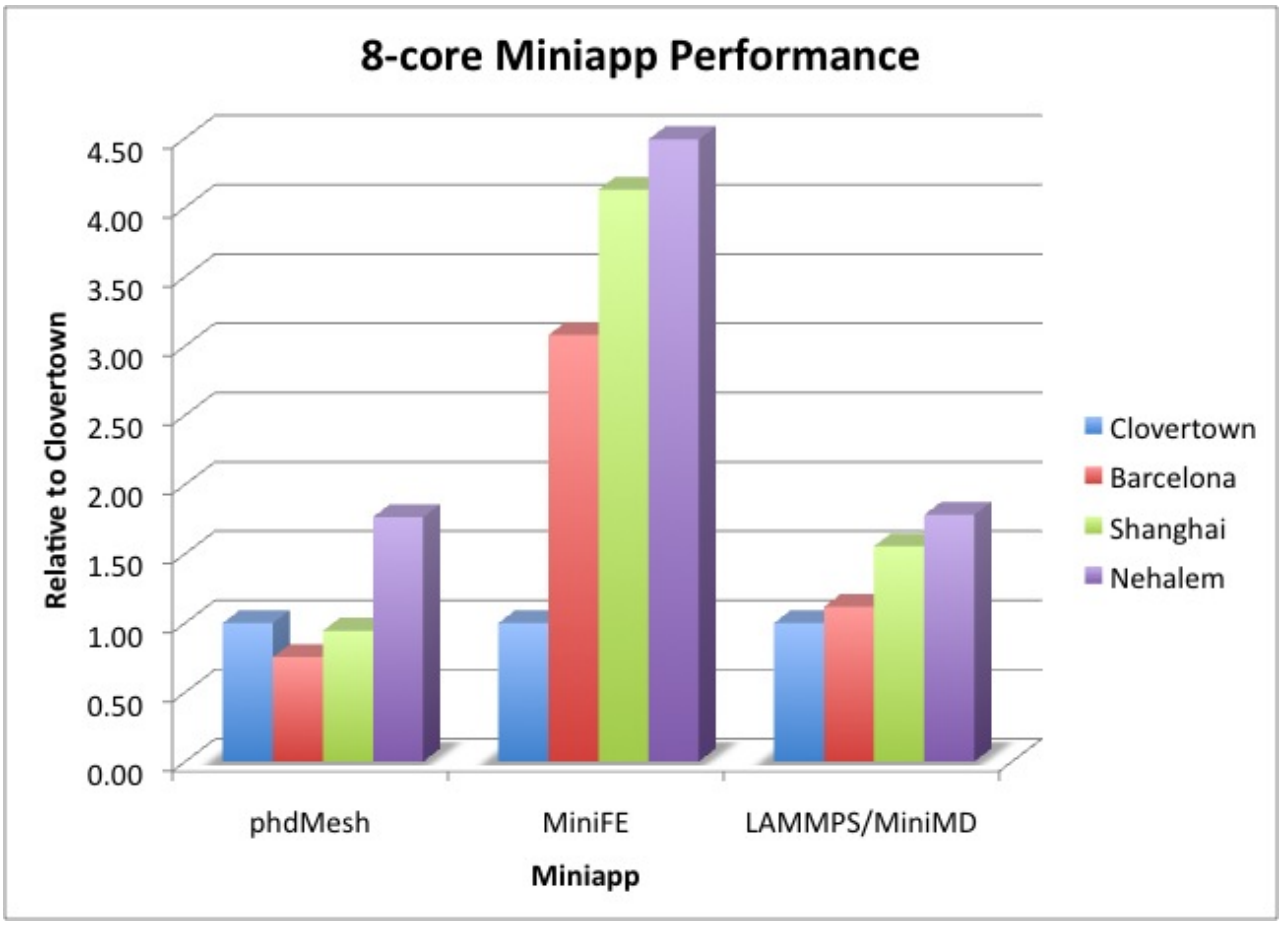

Figure 2: Performance of 8-core execution for three miniapps. All results normalized to Clovertown. These results clearly indicate the potential performance for each category of application, showing especially the poor relative performance of Clovertown for unstructured matrix computations, a result that was later seen in large-scale applications.

to main memory subsystem performance. MiniFE's memory access patterns put a much greater demand on memory and it can be seen that the extra bandwidth provided by the Shanghai processor increases performance significantly relative to the Barcelona. Cache size and performance is very important for LAMMPS, and it can be seen that memory bandwidth improvements do improve performance, but not to the degree of MiniFE.

Figure 3 shows results for MiniFE using 1M equations per core from 1-8 cores. These results clearly indicate that memory system performance is critical to being able to utilize all 8 cores, and also indicate that 32-bit computations can scale much better than 64-bit computations. These results have motivated aggressive efforts in our libraries to store as much data as possible in 32-bit mode, even if computations are performed in double precision. Furthermore, we are compelled to reduce memory bandwidth requirements as much as possible in order to use all cores.

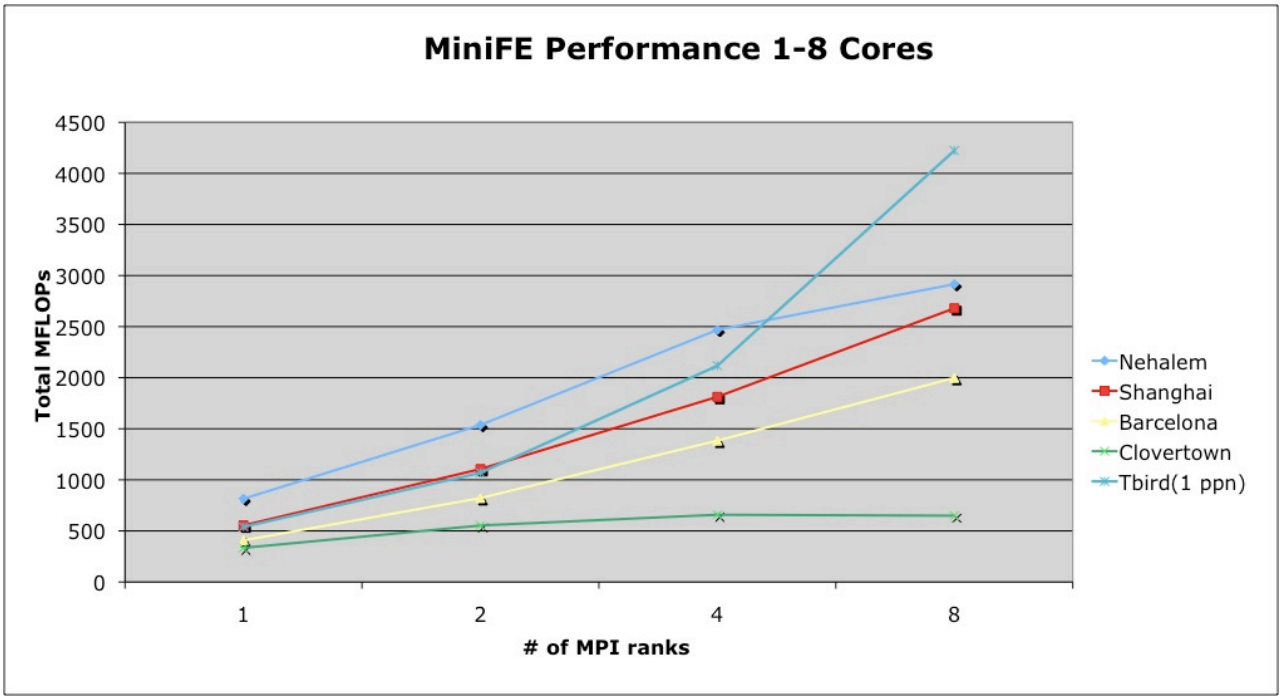

Figure 3: MiniFE results for 1-8 cores. These results show the importance of memory system performance for obtaining good core utilization and illustrate the potential belefit of using single precision data.

Many multicore nodes have a non-uniform access (NUMA) memory subsystem. We have found that memory placement makes a large impact on overall application performance. Figure 4 shows the impact of a processor using it's local memory subsystem vs. the memory subsystem on a neighboring socket. In the latter case, memory performance is dictated by the performance of the socket-to-socket interconnect, HypterTransport for the Barcelona and QuickPath for the Nehalem. As expected, MiniFE is much more sensitive to memory placement and its performance is primarily a function of the memory susbsystem. Since the Nehalem has exceptional local memory performance relative to QuickPath, its sensitivity is much greater than that of the Barcelona where local memory and HypterTransport performance are more closely matched. As was shown previously, phdMesh and MiniMD are less sensitive to local performance and hence also less sensitive to memory placement.

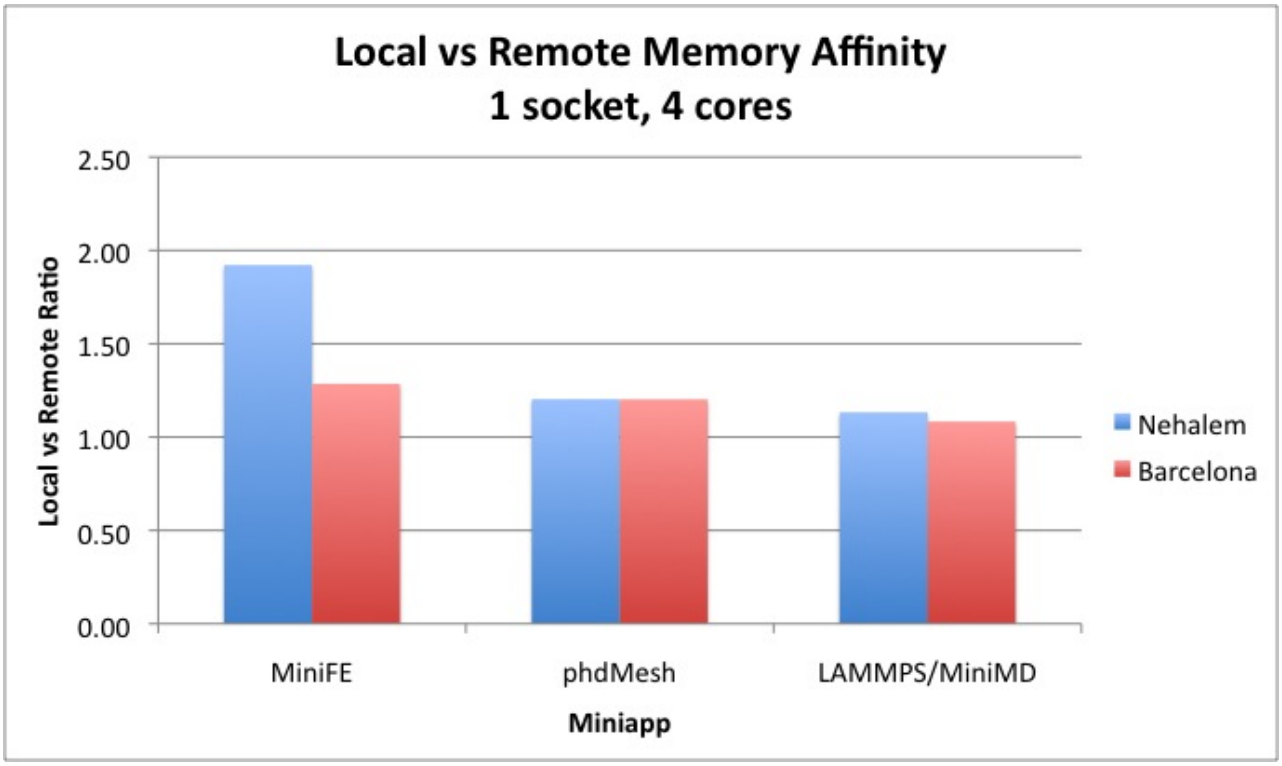

Figure 4: A study of the performance impact due to placing memory on local vs. remote memory sockets on a NUMA node memory system.

Scalable Multicore System Studies

The ease of building and running MiniFE, and straight-forward interpretation of the results are leveraged in use of this miniapp in the early evaluation of several system architectures used by the Sandia and Tri-Lab user community. The computation in MiniFE, as pointed out previously, is dominated by sparse matrix-vector multiplication. The communication is minimal, requiring exchange of nearest neighbor boundary information and global MPI_Allreduce operations required for the scalar computations in the CG algorithm. New MiniFE features to provide more challenging network communication patterns are still under development.

Weak scaling studies assigning identical computational load to each MPI task in a parallel simulation have been carried out on the Cray Red Storm/XT4, on the Tri_lab Capacity Clusters (TLCC), on an older Sandia capacity cluster called Thunderbird(T_Bird) and on the New Mexico Computer Application Center (NMCAC) supercomputer called Encanto. Figure 5 shows the total wall time as a function of the number of MPI tasks for each system. MiniFE clearly brings out the impact of memory architecture on application scaling. For instance on the Red Storm, which is a mildly heterogeneous system with both 2.2 GHz quad-core AMD Budapest nodes and 2.4 GHz AMD Opteron nodes, the former using the newer 800 MHz DDR2 DRAMs and the later using the older DDR 400 MHz DRAMs, we can see that for the dual-core nodes two MPI tasks saturate the memory access during the sparse

matrix-vector operations, while for the quad-core nodes four MPI tasks saturate it.

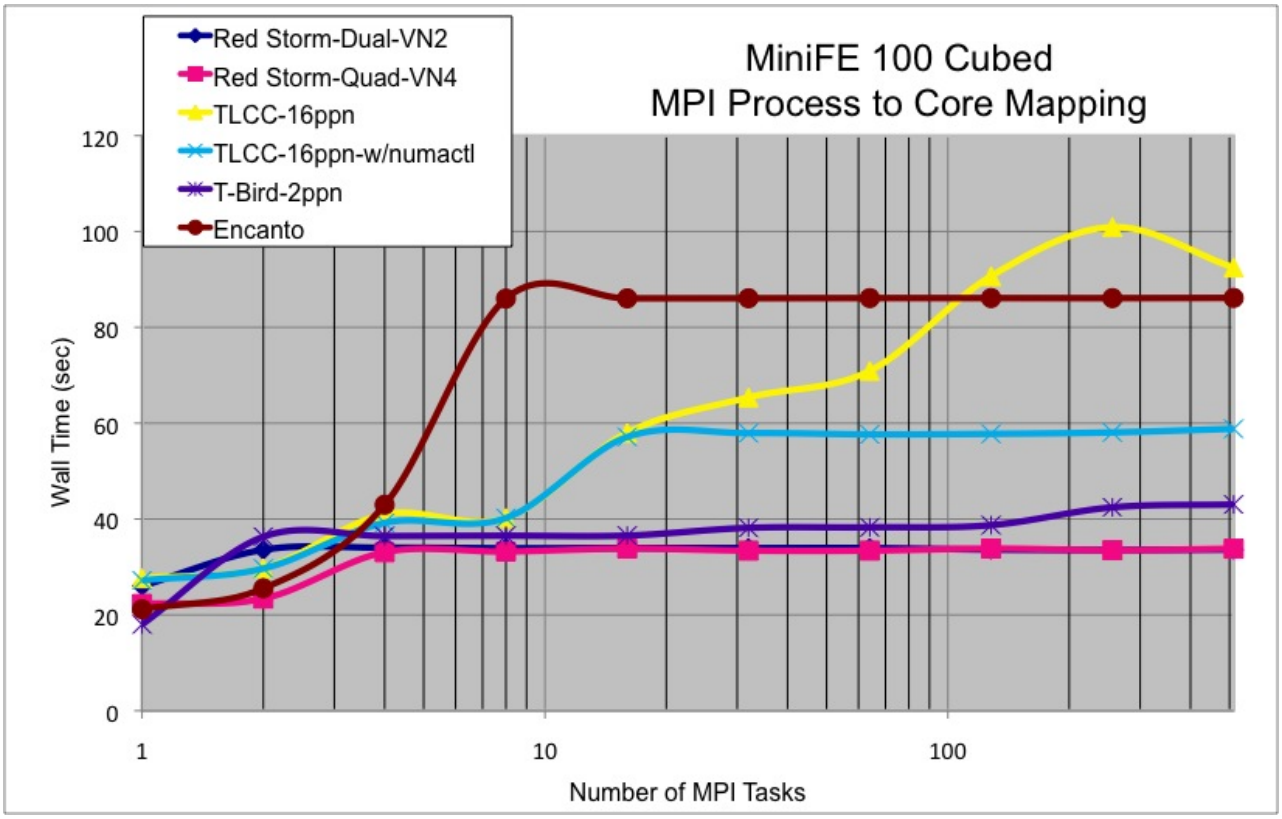

Figure 5: A study of the impact of NUMA architectures on scalability from 1 to 512 MPI tasks.

Once the best performance within a node–as determined by memory bandwidth among the competing cores–is determined, the weak scaling study shows near perfect scaling (flat curve). Similarly for the quad-socket, quad-core TLCC node, we can see that for 1 and 2 MPI tasks we get perfect scaling, some degradation in performance at 4 and 8 MPI tasks due to partial memory contention and at 16 MPI tasks the performance has degraded by 1.5x. For multiple nodes on TLCC using all the cores on each node we see subsequent perfect weak scaling when memory and processor affinity are forced. On the other hand, this miniapp also brings out the destructive impact of OS jitter and thread migration as evident by looking at the scaling curve for TLCC when no 'numactl' is used. Similar conclusions about the impact of memory architecture on the newer multi-core multi-socket nodes emerges from looking at the performance of 'Encanto' whose memory architecture is bus-based as opposed to TLCC's NUMA nodes with independent memory controllers.

Programming Model Studies for Manycore

Manycore nodes appear to be inevitable for scalable computing. A significant strategic question is how to maximize application performance, maintainability, and portability on an-

ticipated HPC architectures with manycore nodes. Application programming model studies are underway to assess the performance benefits of a hybrid approach combining inter-node MPI parallelism with intra-node thread parallelism (including portability to GPGPUs) and the impact of such a programming model on programmability and maintainability of application code.

Miniapps provide an ideal testbed for these application programming model studies. Components of a miniapp are easily re-implemented with intra-node thread parallelism for objective assessment of performance and subjective assessment of programmability and maintainability.

Our first study assessed a hybrid parallel implementation of the HPCCG (precursor to MiniFE) on CPU and GPGPU multicore nodes. This study suggests that the application programming model/architecture illustrated in Figure 6 can enable hybrid parallelism, under the following constraints:

- The conventional inter-node distributed memory domain decomposition parallel programming model is applied in the top three layers ("global control flow" through "node-local control flow").
- Applications' computational work components are separated into resource management components and computational kernel components.
- Computational kernels become "stateless" functions in that they perform their computations on data provided by a resource management component, and never maintain data internally to the kernel. Furthermore, effort is made to expose vector/SIMD constructs to the compiler.

This separation of concerns between computational work and resource management allows node-local threads to be treated as a resource and kernels to be safely called in thread-parallel. Furthermore, kernel programming can be simplified to improve the likelihood of portability between CPU and GPGPU based implementations. For example, a C-language computational kernel devoid of internal states and memory allocation (or other resource management) constructs is not far from a CUDA implementation.

A reimplementation of the HPCCG miniapp using this hybrid programming model has demonstrated the potential for a significant performance gain as compared to a pure-MPI programming model. Intra-node parallelism is implemented with standard pthreads; however, thread management details are abstracted by the Thread Pool Interface (TPI) library in Trilinos. The TPI library provides a simple interface to dispatch computational kernels to a pool of threads, and to reduce results from those kernels as needed (e.g. a parallel dot product must sum its results to a single value).

For modest-sized sparse matrices (less than 100,000 rows with 27 non-zeros per row) on a standard dual socket quadcore (2x4core) Intel Clovertown workstation, performance of the compress row storage (CRS) sparse matrix-vector multiply operation is significantly better for the TPI implementation versus the MPI implementation (Figure 7). The difference be-

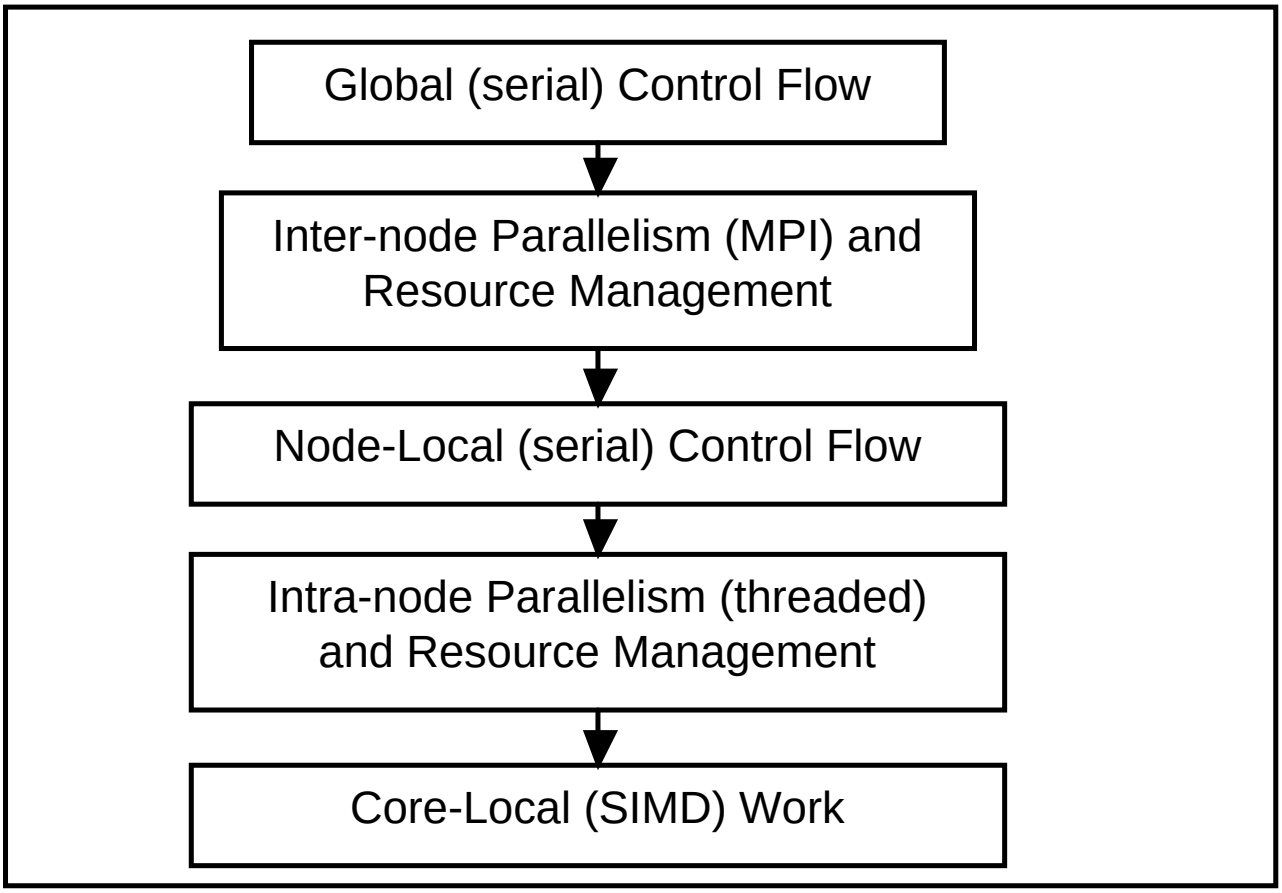

Figure 6: Suggested Application Programming Model / Architecture for Hybrid Parallelism.

tween these two implementations is that the MPI-based implementation must communicate portions of the input vector among MPI processes to apply the sparse matrix-vector multiply kernel while in the TPI-based implementation the kernel simply accesses its designated portion of the input vector from each thread. Thus the entire communication step is eliminated.

Furthermore, in the TPI implementation, the node-local sparse matrix and vectors are maintained in contiguous spans of physical memory, as opposed to each MPI process maintaining its portion of this data in its own allocated portion of the node-local physical memory.

We hypothesize that much of this performance gain is due to improved cache utilization. In the TPI-base implementation the sparse matrix and vector data is stored in contiguous spans of memory, which reduces the chance that portions of these data arrays will occupy the same cache line. Furthermore, in the MPI implementation communication data is allocated and the matrix-vector multiply implementation diverts from the computational code path into a communication/MPI code path, thus increasing the probability of ejecting segments of the sparse matrix and vector data from cache memory. This hypothesis is reinforced by the observation (see Figure 6) that once the size of the sparse matrix and vectors become so large that sustained cache-residency is impossible then the difference in performance between

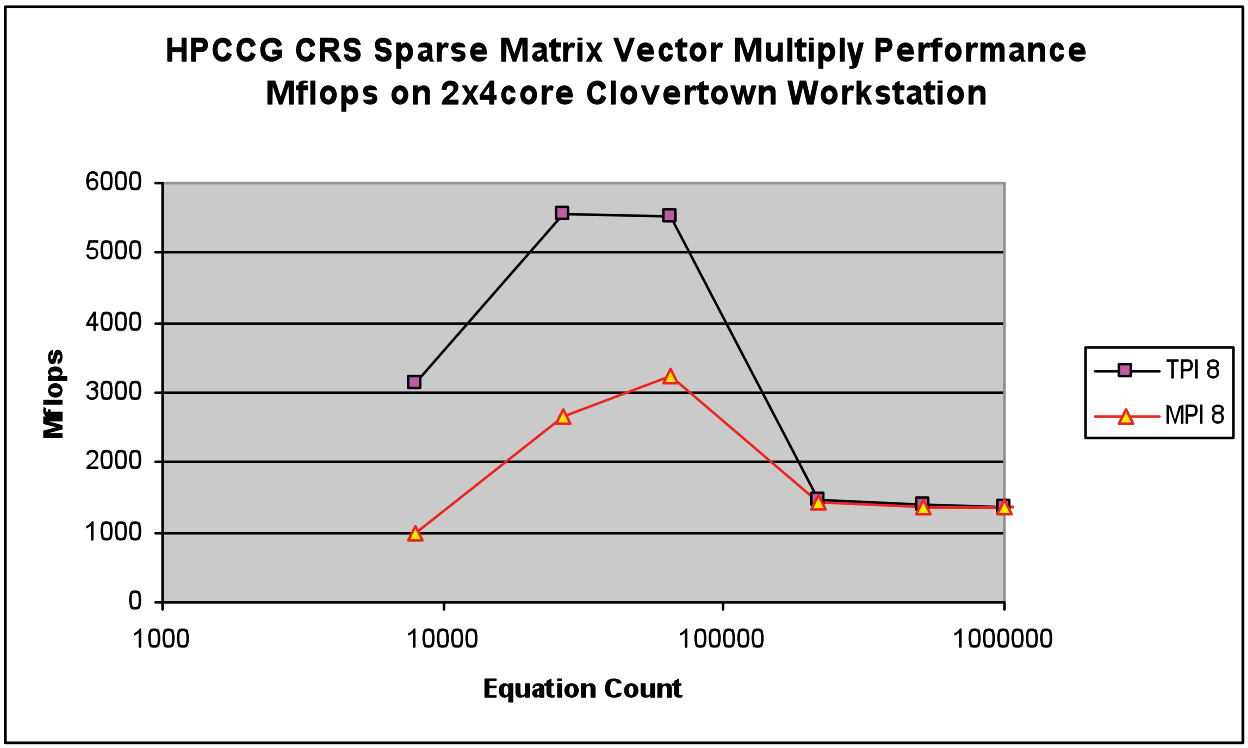

Figure 7: Comparison of Thread-Parallel versus MPI-Parallel Sparse Matrix-Vector Multiply Performance within MiniFE.

the TPI and MPI based implementations becomes negligible.

Prolego Results

Prolego is still a fairly new effort. Therefore, our first results are from attempting to see if Prolego can predict the performance of our other miniapps. In this section we show results for predicting MiniFE performance. The script in Figure 8 was used to configure Prolego to produce the results in Figure 9 on the previously mentioned TLCC cluster. Using the 16-core results for calibration, we get excellent correlation on up to 1024 cores.

Performance Modeling

Miniapps are small enough that explicit timing formulas include all aspects of scaling as a function of problem size and processor count. Our analysis of the MiniFE (HPCCG) miniapp revealed an interesting self-similarity property for parallel systems [24]. Like many simple algorithms, the execution time is the sum of three terms,

$$t = t_C + t_B + t_L \ , \tag{1}$$

a time t_C for computation, a time t_B for communication determined by bandwidth, and a time t_L for a global reduction determined by latency. The fraction of time spent in computation,

$$f_{\text{comp}}(u_L, u_B) = (1 + u_L + u_B)^{-1} \ , \tag{2}$$

is a function of two time ratios, $u_B = t_B/t_C$ and $u_L = t_L/t_C$, with the vector,

$$\vec{\mathbf{r}}(u_L, u_B) = [u_L, u_B, f_{\text{comp}}(u_L, u_B)] \ , \tag{3}$$

defining points on a surface. The coordinates are functions of the problem size n and the number of processors p, such that $u_B = u_B(n, p)$ and $u_L = u_L(n, p)$, and the vector (3) traces a path along the surface parameterized by n and p.

Figure 10 reveals that two machines, an SGI Altix and an IBM Blade cluster studied in a previous paper [24], are self-similar. Roughly speaking, two machines are self-similar if they follow the same path on the surface as the problem size and the number of processors change. More precisely, define the dimensionless parameter $\sigma = u_L/u_B$ and rewrite the fraction (2),

$$f_{\text{comp}}(\sigma u, u) = (1 + (1 + \sigma)u)^{-1} \ . \tag{4}$$

On average, the curve defined by this function with $\sigma = 5$, independent of n and p, represents all the measurements made for the two machines. The parameter σ is the ratio of two computational forces, one related to the latency of the machine's network and the other related to the bandwidth of the machine's network [19, 20, 22, 24]. It defines an equivalence class of machines for this miniapp [24].

The MiniMD miniapp shows analogous behavior [25]. The formula for the execution time can be written as the sum of three terms although the terms have quite different meanings from those of the MiniFE miniapp. Nonetheless, another, but of course different, self-similarity relationship holds specific to this particular application [25].

Simulation

Single-processor performance is best understood using a full-scale, instruction-level simulation of the program as it executes. Such a simulation is out of the question for a complete application code, but miniapps can be limited to just the important parts of an application. The computational kernel of the MiniFE (HPCCG) miniapp is a good candidate for simulation because it involves a sparse matrix-vector multiplication and stresses the local memory hierarchy.

We used the Structural Simulation Toolkit [4] to simulate the inner loop for sparse matrix-vector multiplication. The simulator produces a detailed trace of the program as it executes

from which we extracted the issue time at clock-tick k_j for each instruction and the completion time at clock-tick $k_j + \kappa_j$. The issue time is determined by the machine's hardware constraints, whether, for example, the operands are ready, and the completion time depends on whether, for example, an address hits or misses in one of the caches.

The instruction trace defines an energy spectrum for the program [21, 23],

$$T(k) = \frac{n}{2} - \frac{1}{2}\sum_{j=1}^{n} \cos(\omega_j(k - k_j)) \ , \tag{5}$$

where n is the number of instructions in the trace with frequencies, $\omega_j = 2\pi/\kappa_j$, determined by the number of clock-ticks consumed by each instruction. The Fourier transform of the energy spectrum, $(\mathcal{F}T)(\omega)$, for positive frequencies $\omega \geq 0$, yields the formula,

$$(\mathcal{F}T)(\omega) - \frac{n}{2}\delta(\omega) = -\frac{1}{4}\sum_{j} n_j e^{-i\omega k_j}\delta(\omega - \omega_j) \ , \tag{6}$$

where n_j is the number of instructions with frequency ω_j. The absolute value of (6) yields a spectrum in the frequency domain,

$$4\left|(\mathcal{F}T)(\omega) - \frac{n}{2}\delta(\omega)\right| = \sum_{j} n_j \delta(\omega - \omega_j) \ , \tag{7}$$

a collection of delta functions at each characteristic frequency with height equal to the number of instructions at that frequency.

Examination of the trace shows that the delta function at $\kappa = 4$ represents a branch instruction, and the one at $\kappa = 5$ represents a fused multiply-add instruction. The critical bottlenecks, not surprisingly, are references to memory. There are fifty-five memory references corresponding to L_1 hits at $\kappa = 6$; ten that correspond to L_2 hits at $\kappa = 29$; and sixteen that correspond to L_2 misses at $\kappa = 106$ for a total of $55 + 10 + 16 = 81$ memory references. The L_1 hit ratio, then, is $h_1 = 55/81 = 0.68$; the L_2 hit ratio is $h_2 = 10/81 = 0.12$; and the main-memory hit ratio is $h_3 = 16/81 = 0.20$.

Although the issue times, k_j, do not affect the absolute value of the frequency spectrum, they determine the execution time of the program. The phase factors on the right side of (6),

$$e^{-i\omega k_j} = \cos(\omega k_j) - i\sin(\omega k_j) \ , \tag{8}$$

lie on the unit circle in the complex plane. To follow the evolution of the program, plot the instructions along the helix,

$$x = \cos(\alpha k) \qquad y = -\sin(\alpha k) \qquad z = k \ , \tag{9}$$

with $\alpha = 2\pi/K$ where $K = \max(k_j + \kappa_j)$ is the total execution time of the program. As shown in Figure 12, as the instructions traverse one trip around the circle, they rise from the plane $z = 0$ as time advances. Instruction cluster in groups followed by gaps that correspond to constraints on issue and completion times.

If each instruction waits for the one ahead of it with no overlap, the total execution time,

$$t = \sum_j n_j \kappa_j \ , \tag{10}$$

is the sum of the individual execution times [26, 29, 30, 31]. This sum is the computational action generated by the program [23], the scalar product of a vector containing the height of each delta function and a vector containing the position of each delta function. It estimates the execution time as $t = 4409$, much larger than the actual execution time, $t = 2156$. The goal of optimization is to reduce the computational action [21] by reducing the number of instructions and by overlapping them.

Since this miniapp is dominated by its memory instructions, the execution time from formula (10),

$$\begin{aligned} t &= 81 \cdot (6h_1 + 29h_2 + 106h_3) \\ &= 81 \cdot (6 \times 0.68 + 29 \times 0.12 + 106 \times 0.20) \\ &= 2316 \ , \end{aligned} \tag{11}$$

can be estimated by taking the total number of instructions equal to the number of memory instructions, $n = 81$, as if nothing else were happening, and using the cache-miss ratios to compute the number of instructions at each frequency. This estimate exceeds the actual execution time, $t = 2156$, by about nine percent.

Although the combined hit ratios to the L_1 and L_2 caches, $h_1 + h_2 = 0.80$, is a reasonably high value, the long latency to main memory, for the remaining references that miss both caches, dominates the execution time. The result suggests that the L_2 cache is of limited value in reducing the execution time for this application.

Conclusions

Application performance is determined by a large collection of inter-related issues. As a result, we need a large and varied toolset to explore the design space when performing research and development of high performance systems and applications. Although benchmarks and large-scale applications have been used extensively in this process, we believe that miniapps are an effective and underdeveloped class of tools that can greatly accelerate and enhance the decision making process.

Presently, node architecture changes and the resulting intense effort to develop the next generation of node programming models pose a serious challenge to HPC application development. Furthermore, extreme scale systems continue to grow in node count reaching a level where existing scalable algorithms are challenged. All of these issues and more can be addressed by the use of miniapps.

```
 <prolego_input>

<ParameterList name="cg_int_double">
  <Parameter name="vector_length" type="int" value="27000"/>
  <Parameter name="num_iterations" type="int" value="49"/>
  <Parameter name="share_data" type="bool" value="true"/>

  <ParameterList name="vecdot_int_double">
  </ParameterList>

  <ParameterList name="mpi_ops_int_double">
    <Parameter name="MPI_OPERATION" type="string" value="MPI_Allreduce"/>
  </ParameterList>

  <ParameterList name="vecaxpy_int_double">
  </ParameterList>

  <ParameterList name="crsmatvec_int_double">
    <!-- Don't supply num_rows or vector_length here,  get from cg above. -->
    <Parameter name="nnz_per_row" type="int" value="27"/>
    <!-- num_matvecs here means num-matvecs-per-iteration...-->
    <Parameter name="num_matvecs" type="int" value="1"/>
  </ParameterList>

  <ParameterList name="vecdot_int_double">
  </ParameterList>

  <ParameterList name="mpi_ops_int_double">
    <Parameter name="MPI_OPERATION" type="string" value="MPI_Allreduce"/>
  </ParameterList>

  <ParameterList name="vecaxpy_int_double">
  </ParameterList>

  <ParameterList name="vecaxpy_int_double">
  </ParameterList>

</ParameterList>


</prolego_input>
```

Figure 8: XML Script to configure Prolego so that it mimics the performance of MiniFE. This script was used to produce the results in Figure 9.

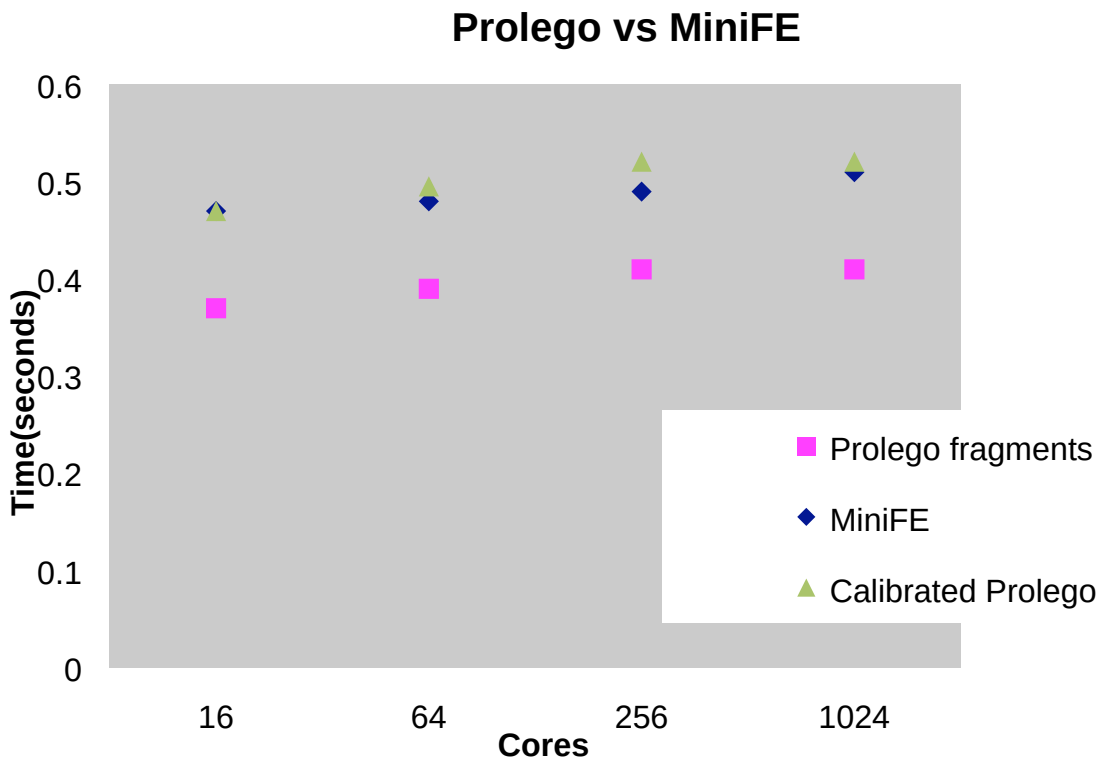

Figure 9: Comparison of MiniFE performanc with performance predicted by Prolego using the script in Figure 8. The calibrated results come from scaling the Prolego results such that the 16 processor results of MiniFE and Prolego match.

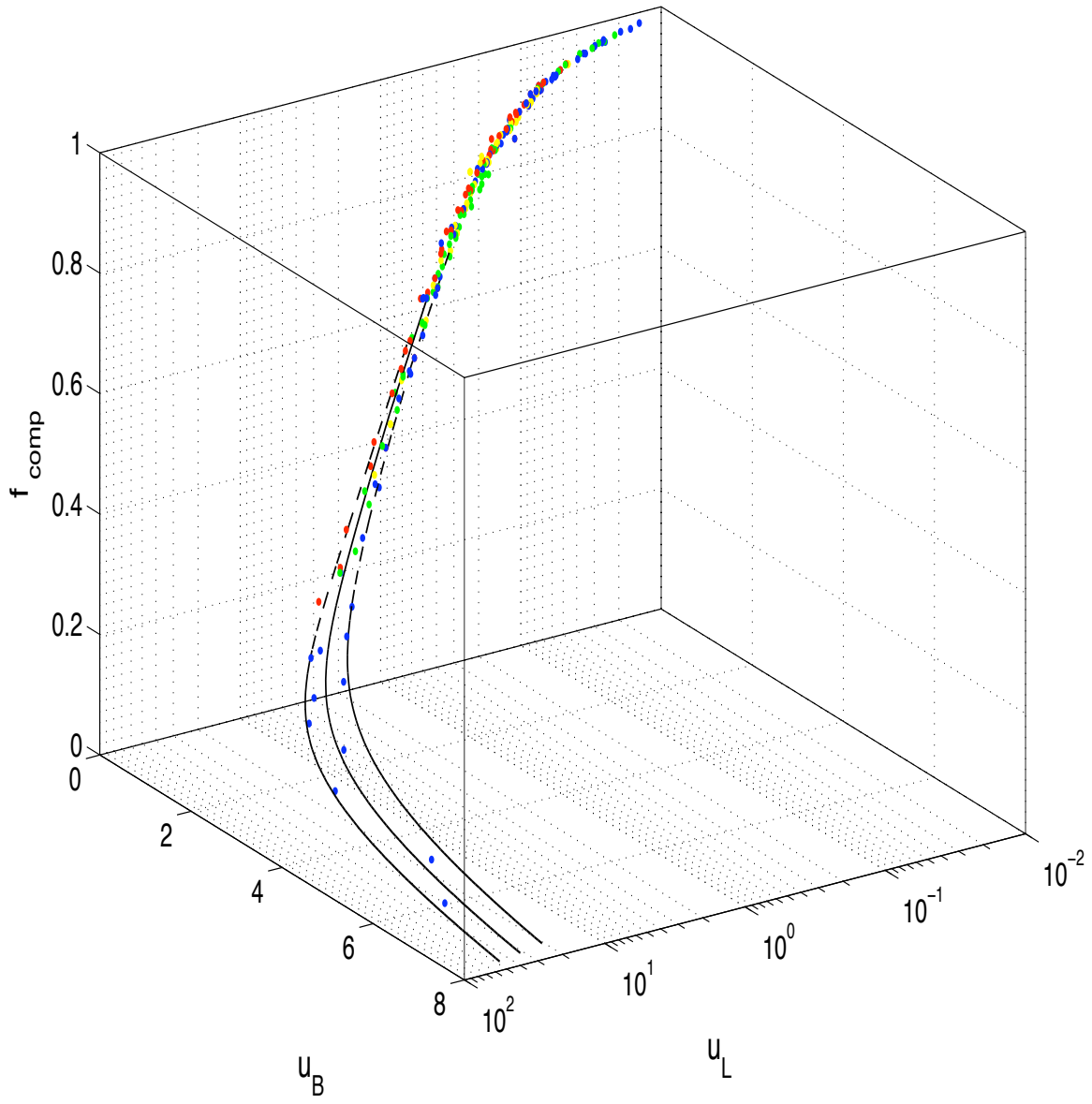

Figure 10: Fraction of time spent in computation as a function of the coordinates $u_L(n,p)$ and $u_B(n,p)$ for $p = 16, 32, 64$ and $n = pn_x^3$ with $n_x = 8, \ldots, 64$. The blue and green bullets mark measured values for the SGI machine. The red and yellow bullets mark measured values for the IBM machine. The central solid line is function (4) with $\sigma = 5$, and the two lines on either side correspond to $\sigma = 7$, on the left side, and $\sigma = 2.5$, on the right side. Notice that the u_L axis is logarithmic.

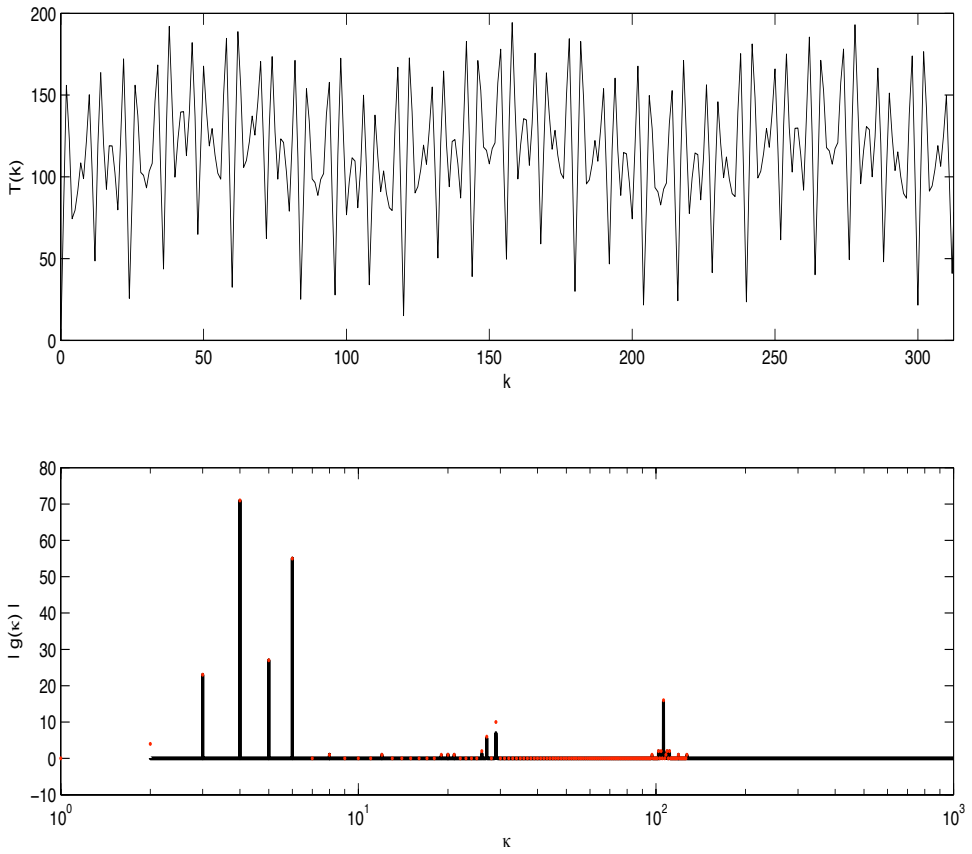

Figure 11: The energy spectrum on the top, as a function of clock-tick k, and its Fourier transform on the bottom, as a function of the logarithm of the reciprocal of frequency, $\kappa = 2\pi/\omega$. The red dots in the bottom figure are the number of instructions at each frequency counted directly from the simulation trace.

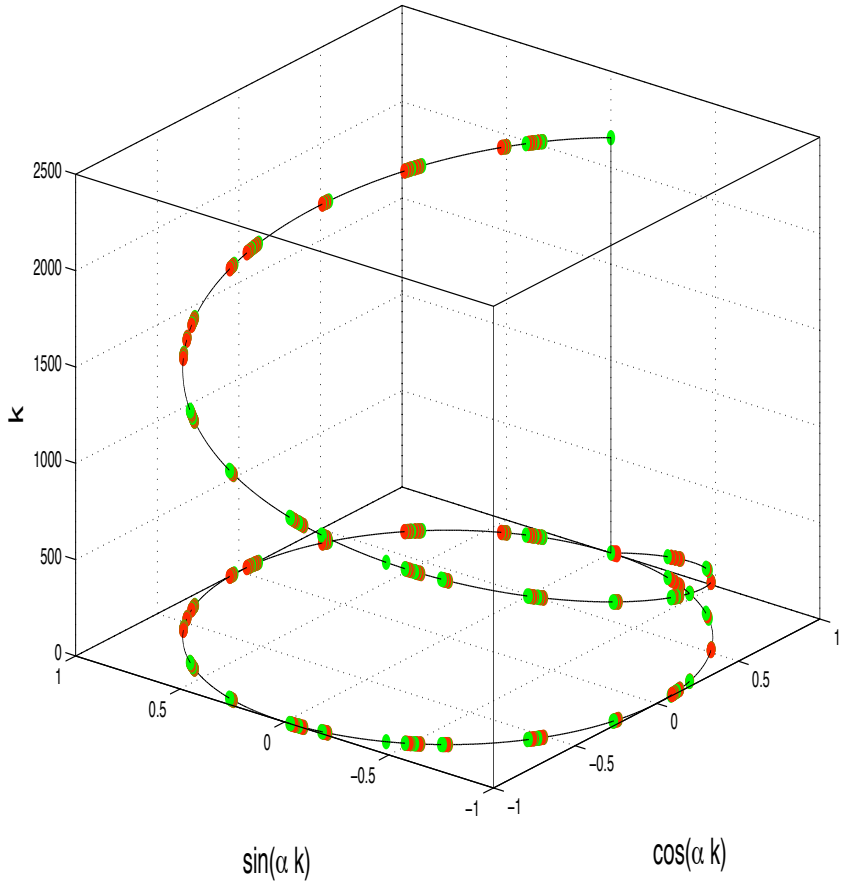

Figure 12: Instructions traversing the helix (9). The red bullets mark the issue time for each instruction, and the green bullets mark the completion time for each instruction. Program execution begins at the first red bullet at $k = 0$ and ends at the last green bullet at $K = 2156$.

References

[1] GNU Lesser General Public License - GNU Project, 2009. http://www.gnu.org/licenses/lgpl.html.

[2] LAMMPS Molecular Dynamics Simulator, 2009. http://lammps.sandia.gov/index.html.

[3] Sandia National Laboratories: Qthreads, 2009. http://www.cs.sandia.gov/qthreads.

[4] Sandia National Laboratories: Structural Simulation Toolkit, 2009. http://www.cs.sandia.gov/sst.

[5] The Official YAML Web Site, 2009. http://www.yaml.org.

[6] S. Attaway, K. Brown, D. Gardner, B. Hendrickson, S. J. Plimpton, and C. Vaughan. Transient Solid Dynamics Simulations on the Sandia/Intel Teraflop Computer. In *Supercomputing '97 Proceedings*, San Jose, CA, 1997. ACM/IEEE.

[7] S. Attaway, S. Plimpton, D. Gardner, C. Vaughan, K. Brown, and M. Heinstein. A Parallel Contact Detection Algorithm for Transient Solid Dynamics Simulations Using PRONTO3D. *Computational Mechanics*, 22:143–159, 1998.

[8] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Advanced Supercomputing (NAS) Division, 1994.

[9] Jonathan L. Brown, Sue Goudy, Mike Heroux, Shan Shan Huang, and Zhaofang Wen. An envolutionary path towards virtual shared memory with random access. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–117, New York, NY, USA, 2006. ACM.

[10] Laura C. Carrington, Michael Laurenzano, Allan Snavely, Roy L. Campbell, and Larry P. Davis. How well can simple metrics represent the performance of hpc applications? In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 48, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Cameron S. Christensen. The Design and Implementation of an Automatic Data Collection and Analysis Tool, 2009. B.S. Honors thesis, St. John's University.

[12] J. Dongarra, H. Meuer, and E. Strohmaier. Top 500 Supercomputer Sites. Technical report, University of Tennessee, Knoxville, TN, USA, 1999.

[13] M. A. Heroux. Trilinos Home Page, 2003. http://trilinos.sandia.gov.

[14] M. A. Heroux. Mantevo Home Page, 2008. http://software.sandia.gov/mantevo.

[15] Eric R. Keiter, Ting Mei, Thomas V. Russo, Eric L. Rankin, Roger P. Pawlowski, Richard L. Schiek, Keith R. Santarelli, Todd S. Coffey, Heidi K. Thornquist, and Deborah A. Fixel. Xyce Parallel Electronic Simulator: Users' Guide, Version 4.1. Technical Report SAND2008-6461, Sandia National Laboratories, 2008.

[16] Darren J. Kerbyson. Software — Performance and Architecture Laboratory (PAL) CCS-3 — Los Alamos National Laboratory (LANL), 2009. http://www.c3.lanl.gov/pal/software.shtml.

[17] B. J. Lucchesi. A Parallel Linear Octree Collision Detection Algorithm. Master's thesis, University of Nevada, Reno, 2002.

[18] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, New York, NY, USA, 2006. ACM.

[19] Robert W. Numrich. Computational force: A unifying concept for scalability analysis. In *Proceedings of the International Conference ParCo 2007*, pages 107–112. John von Neumann Institute for Computing (NIC) and Jülich Supercomputing Centre, 2007.

[20] Robert W. Numrich. Computational forces in the Linpack benchmark. *Journal of Parallel and Distributed Computing*, 68(9):1283–1290, September 2008.

[21] Robert W. Numrich. A metric space for computer programs and the principle of computational least action. *The Journal of Supercomputing*, 43(3):281–298, March 2008.

[22] Robert W. Numrich. Computational forces in the SAGE benchmark. *Journal of Parallel and Distributed Computing*, 69:315–325, 2009.

[23] Robert W. Numrich. The computational energy spectrum of a program as it executes. *The Journal of Supercomputing*, in press, February 2009.

[24] Robert W. Numrich and Michael A. Heroux. Self-similarity of parallel machines. under review, November 2008.

[25] Robert W. Numrich and Michael A. Heroux. A performance model with a fixed point for a molecular dynamics kernel. In *Proceedings International Supercomputing Conference '09*, June 23-26, Hamburg, Germany, 2009.

[26] Bernard L. Peuto and Leonard J. Shustek. An instruction timing model of cpu performance. In *Proceedings 4th Annual Symposium on Computer Architecture*, pages 165–178. ACM Press, New York, 1977.

[27] S. 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*, pages 8–21, Minneapolis, MN, 1987. SIAM.

[28] Steve Plimpton. Fast Parallel Algorithms for Short-range Molecular Dynamics. *J. Comput. Phys.*, 117(1):1–19, 1995.

[29] Rafael H. Saavedra and Alan J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, November 1996.

[30] Rafael H. Saavedra and Alan Jay Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, October 1995.

[31] Rafael H. Saavedra and Alan Jay Smith. Performance Characteristics of Optimizing Compilers. *IEEE Transactions on Software Engineering*, 21(7):615–627, July 1995.

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic)
1	MS 0123	D. Chavez, LDRD Office, 1011

Sandia National Laboratories