Title

Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications

Authors

Barrett, RF
Crozier, PS
Doerfler, DW
et al.

Publication Date

2015

DOI

10.1016/j.jpdc.2014.09.006

Peer reviewed

Assessing the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications

R.F. Barrett, P.S. Crozier, D.W. Doerfler, M.A. Heroux, P.T. Lin, H.K. Thornquist, T.G. Trucano and C.T. Vaughan

Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA

Abstract

Computational science and engineering application programs are typically large, complex, and dynamic, and are often constrained by distribution limitations. As a means of making tractable rapid explorations of scientific and engineering application programs in the context of new, emerging, and future computing architectures, a suite of "miniapps" has been created to serve as proxies for full scale applications. Each miniapp is designed to represent a key performance characteristic that does or is expected to significantly impact the runtime performance of an application program. In this paper we introduce a methodology for assessing the ability of these miniapps to effectively represent these performance issues. We applied this methodology to four miniapps, examining the linkage between them and an application they are intended to represent. Herein we evaluate the fidelity of that linkage. This work represents the initial steps required to begin to answer the question, "Under what conditions does a miniapp represent a key performance characteristic in a full app?"

1. Introduction

Over the past several years computer architectures commonly employed by the computational science and engineering communities have remained relatively stable, with subsequent generations characterized by faster processors, memories, and interconnects. These changes have typically resulted in predictably faster runtimes for application programs. Emerging and expected future architectures, however, are presenting special challenges and opportunities that, if not effectively exploited, could result in slower runtimes. Driven by stagnant clock speeds, memory constraints, and power consumption restrictions [2, 9], architects are exploring the capabilities and usefulness of things such as wider vector units, significantly more but less powerful processor cores, increased threading, more complex memory hierarchies, and differently balanced node interconnects.

The means for exploiting these capabilities must be investigated within the relevant context of their use. However, application programs targeting these machines are typically large, complex, dynamic, and often constrained by distribution limitations, and typically out live the computing environments they originally targeted. They may be constructed using hundreds of thousands to millions of source lines of code, written in multiple programming languages and linking in several third-party libraries, developed over decades by multiple generations of computational scientists. Thus examining and addressing the issues that will enable effective execution on these machines is prohibitive.

As a means of making tractable rapid explorations of scientific and engineering application programs in this context, a suite of "mini-apps" has been created to serve as proxies for full scale applications. Each miniapp is designed to represent a key performance characteristic that does, or is expected to significantly, impact the runtime performance of a scientific or engineering application program.

These miniapps enable rapid exploration of key performance issues that impact a broad set of scientific application programs. Within the Department of Energy (DOE) they are being used to explore the above issues by a broad set of participants, including staff at DOE laboratories, universities, and vendors. Yet how can we be sure that these proxies adequately represent that which they are intended?

Email address: rfbarre@sandia.gov (R.F. Barrett, P.S. Crozier, D.W. Doerfler, M.A. Heroux, P.T. Lin, H.K. Thornquist, T.G. Trucano and C.T. Vaughan)

The key contribution of the work described herein is a methodology, rooted in formal verification and validation (V&V) efforts that have been developed for experimental science, for determining the quality of the miniapp as it pertains to a large, complex application code. We applied this methodology to four miniapps, examining the linkage between them and an application they are intended to represent. Herein we evaluate the fidelity of that linkage. This work represents the initial steps required to begin to answer the question, "Under what conditions does a miniapp represent a key performance characteristic in a full app?"

1.1. Related work

Application proxies have been part of the code developers' tool kit for many years. The LINPACK benchmark came into existence [13] as what we are now calling a miniapp. Sweep3d [22], sPPM [3], and the NAS benchmarks [4] in some sense may also be viewed in these terms. Large scale proxies, such as LULESH [23], are serving related purposes. The three DOE Office of Advanced Scientific Computing Research (ASCR) Co-Design Centers [1] (ExMatEx, CESAR, and ExaCT) have identified the development of proxy applications as a key component of their efforts. Miniapps, and other kinds of application proxies, are being used as part of machine procurements [2]. The Mantevo project [19] solidifies the application proxy idea, bringing a community-based focused effort to bear on the wide variety of explorations that application proxies can enable.

The validation methodology presented herein is the first formal means, that we are aware of, for understanding if, and how, an application proxy may be used to represent the behavior of a full application program. This work is strongly informed by techniques developed for experimental validation, as will be discussed in the following sections.

2. Overview of the Mantevo Project

The Mantevo project [19] was motivated by questions arising from the Trilinos project [18]. These questions concerned the direction of some coding implementations targeting emerging and expected future architectures, including multi-core, many-core, and GPU-accelerated high performance computers. The goal was to create a suite of tools that placed important algorithms into an application-relevant context, enabling rapid exploration of issues and options and their mapping to computing platforms.

Mantevo miniapps are designed and developed to be a tool, useful throughout the co-design space [16], enabling agile exploration of a variety of issues that impact performance. Unlike a compact application, which is designed to capture some sort of physics behavior, miniapps are designed to capture some key performance issue in the full application. Unlike a skeleton application, which is designed for only focusing on inter-process communication perhaps involving a "fake" computation, miniapps create a meaningful context in which to explore key performance issues. Miniapps are developed and owned by application code teams. Miniapps are intended to be modified, and thus are generally limited to a few thousand source lines of code (SLOC), allowing for unconstrained modification. Once no longer useful for these purposes, a miniapp will be discarded. Mantevo miniapps are freely available as open source software under an LGPL license.

The current set of miniapps in the Mantevo project are listed in Table 1. The first miniapp was HPCCG, which formed and solved a sparse linear system of equations. Although it provides an important capability, it was soon realized that in order to provide a stronger tie to applications of interest, the context in which the linear system is formed needed strengthening. The result was miniFE, putting the linear system into the context of an implicit finite element solver. Thus although HPCCG continues to serve an important role, miniFE will be examined in detail for purposes herein. We anticipate that this sort of situation will continue to occur as these miniapps are used in different situations.

[1] http://science.energy.gov/ascr/research/scidac/co-design/

[2] http://www.nersc.gov/systems/trinity-nersc-8-rfp/draft-nersc-8-trinity-benchmarks/

Miniapp	*Description*
CloverLeaf	Solves the compressible Euler equations on a Cartesian grid, using an explicit, second-order accurate method.
CoMD	A simple proxy for the computations in a typical molecular dynamics application. The reference implementation mimics that of SPaSM.
HPCCG	Intended to be the best approximation to an unstructured implicit finite element or finite volume application in 800 lines or fewer.
miniFE	A proxy for unstructured implicit finite element codes. It is similar to HPCCG and pHPCCG but provides a much more complete vertical covering of the steps in this class of applications.
miniGhost	A difference stencil across a homogenous three dimensional domain, targeting the inter-process communication halo exchange operation.
miniMD	The force computations in a typical molecular dynamics applications. The algorithms and implementation used closely mimics these same operations as performed in LAMMPS.
miniXyce	SPICE-style circuit simulator [31].

Table 1: List of Mantevo miniapps, release 1.0

3. Methodology

Miniapps are designed to provide a predictive capability for some key performance issue in a full application. Ensuring that a miniapp completely fulfills its intent is a difficult and probably ongoing task. Further, the runtime behavior of a complex scientific application is typically problem dependent, and therefore it is important to understand the different ways that a code can be used and have a means for configuring the miniapp to mimic the important features under consideration. Thus our approach is to build up a "body of evidence" in support of the goals of a miniapp, combining formal verification and validation (V&V) techniques with our knowledge and experience bases.

Verification is the process of determining that a model implementation accurately represents the developers conceptual description of the model and the solution to the model. *Validation* is the process of determining the degree to which a model is an accurate representation of the "real world" (in this case the performance characteristics of the "real" application) from the perspective of the intended uses of the model. These terms are as defined by the American Society of Mechanical Engineers (ASME, 2006) and the American Institute of Aeronautics and Astronautics (AIAA, 1998), and this usage has basically been adopted by the United States DOE and Department of Defense (DoD). That is, within the context of the intent of the comparisons of a model with the "real world," we must verify that the applications ("real world") and miniapps ("model") compare well in the performance dimensions of interest. This is our defined means of assessing that the miniapps are accomplishing what they are designed and required to do (or not). All of the work (and possibly art) in this methodology will be in defining a set of comparisons that allow us to draw conclusions of this kind about the miniapps. We must also understand how close these comparisons should be for us to be able to conclude that the miniapps are suitably accurate models of real code performance, or that they aren't. There will clearly be significant components of judgment embedded in this methodology given the difficult nature of this problem, but the goal is to achieve some minimal level (at least) of objective evidence that informs us constructively about the fidelity of the miniapps. This approach requires extensive knowledge of, and experience developing, executing, profiling, maintaining, and extending multi-scale, multi-physics scientific and engineering application software, targeting highest performance computing platforms. It also requires a strong understanding of the miniapps and their intended use: what they are intended to represent and what they are not intended to represent. We combine this knowledge into a formal verification and validation (V&V) methodology that lets us examine experimental and predicted data.

This methodology adheres to the spirit of experimental validation as described in [32, 33, 34, 41] because validation referents are intended to be representative of the empirical (that is, "real") performance of the full applications. However, its important to note that this V&V process is executed within the goals of miniapps, and thus we are not concerned with ensuring any sort of V&V in regard to the application goals such as correctness of the algorithms and output. In other words, in considering the performance fidelity of miniapps, we are not addressing questions about the verification and validation of the full application; we are assuming that this has been done or, if not, that this

is an issue that is not relevant to the immediate goals of developing miniapps. The status of this assumption is in fact of interest, but beyond our immediate scope. Our focus is strictly on the computational runtime characteristics. Beyond this issue we also stress that miniapps are not intended to reproduce specific physics and mathematics represented by the full application. To some degree, we therefore have an operating assumption that a valid miniapp can approximate the runtime performance characteristics of a full application to a useful degree without reproducing the mathematics and physics of the full application to a useful degree. This may be an assumption that is worthy of fuller consideration also, but once again it is beyond the scope of our near term priorities.

3.1. Verification

Mantevo miniapps have been configured so that they produce some outcome that is measurable with some level of confidence of correctness. For example, miniFE solves a partial differential equation (PDE) such that the residual norm of the linear system solution is within an acceptable tolerance.

With regard to the application, we necessarily begin with the assumption it meets it V&V requirements. That said, our methodology still includes a strong element of software verification. In particular, because our validation approach is designed to expose differences in the runtime characteristics between the miniapp and application, the differences seen can point to areas where both the miniapp and application should be examined. In some sense, then, this is a code-to-code V&V exercise, whereby lack of agreement between the two can strengthen the (always ongoing) V&V efforts of each. This of course is not a definitive result, since both codes may be similarly incorrect, and therefore somewhat controversial. However, given the assumptions stated above, this sort of information can still provide a useful service if properly understood.

3.2. Validation

For a set of diagnostic runtime performance characteristics or elements, which we loosely refer to as the *performance domain*,

$$\{D\} = D_1, D_2, \ldots, D_n, \tag{1}$$

let

$$\{B\} = B_1, B_2, \ldots, B_n, \tag{2}$$

be a corresponding set of baseline full application observational referents, (the "validation data") and let

$$\{A\} = A_1, A_2, \ldots, A_n, \tag{3}$$

be a set of corresponding miniapp measurements.

We then consider the difference between the application referents and the miniapp measurements in the performance domain defined by (1) as some kind of mathematical norm, which we will also call a *validation metric*:

$$X_i = \|B_i - A_i\|_i, \forall i. \tag{4}$$

Here, we have suggested that the difference measurement – the norm – might vary for each component of the performance domain. Clearly, this can become extremely complex and examples presented below will help clarify this. A very simple example could be a situation in which a positive number specifies every component of performance domain. In such a case, we then could simply have:

$$X_i = \|B_i - A_i\|_i = |B_i - A_i|, \forall i. \tag{5}$$

But one component of the performance domain might be specified this way, while another component might be as general as a functional or a time series, in which case the norm is far more complex than simply measuring the absolute value of the difference between two numbers. There is also the possibility that stochastic characteristics must be attached to one or more of the performance dimensions, which further complicates the mathematics that might underlie the validation metric. We do assume that all the components of the performance domain can have a norm distance definition attached to them. In something as extremely complicated as overall runtime performance behavior, even this assumption might fail if we had to deal with qualitative factors in performance. We do not see the need for this level of generality for this discussion.

The point of introducing a validation metric that measures the difference between miniapp performance and the application referent is to draw some conclusion about how well the miniapp is reproducing the performance behavior

of the full application. A simple illustration of this logic is as follows. Suppose that the validity of the miniapp was determined by how accurately it reproduces the performance of the full application in the performance domain. Then the differences in Equation (4) provide the means for assessing validity. Thus, given the measured set of values X_i, for $i = 1, \ldots, n$, suppose "valid" accuracy can be assessed using a set of threshold accuracies T_i^1, T_i^2, for$i = 1, \ldots, n$. Then assessment of the validation metric information might then be posed as:

$$V_i = \begin{cases} \text{predictive}, & \text{for } T_i^1 \leq X_i \leq T_i^2 \\ \text{caution}, & \text{for } T_i^2 \leq X_i \leq T_i^3 \\ \text{not predictive}, & \text{for } X_i \geq T_i^3 \end{cases} \tag{6}$$

where V_i is a validity statement attached to performance domain dimension i for some thresholds T_i^j, for $j = 1, \ldots, 3$.

While Equation (6) looks like a generally useful algorithm for assessment, we caution there is a great deal of overloading inherent in this simple expression. For example, the choice of thresholds could clearly be extremely difficult. The willingness to even evaluate validity based on a relatively direct threshold assessment is open to debate. And, developing the set $V_i, i = 1, \ldots, n$ leaves open the issue of how all of this information is combined into a single appraisal of the validity of the miniapp. Nonetheless, this logic is a clear illustration of the kind of ideal thinking that should underlie the validation assessment of miniapps.

Choice of diagnostics defining the performance domain is clearly a challenge. Example diagnostics for the set D include inter-process communication, which can then be further categorized, such as the number and distribution of partners, and the size and frequency of message traffic. Example observational referents B for the full application may be empirically measured, which we denote as B^M. Or the referents may need to be forecast or estimated using expert judgment, perhaps aided by a tool such as SST [38], denoted B^p. Using referents that are not empirically measured for validation is sometimes called face validation. While accepted by several communities, including DoD, clearly *face validation* has weaker validation inference associated with it than the use directly observed empirical referents. We expect that we might have to use a mixture of both in the validation methodology for miniapps. In general, we emphasize that measurements involving miniapps A must be carefully designed and captured. We also emphasize that having confidence in A and B requires having accumulated meaningful verification evidence.

This framework provides direct advantages. First, the input information D, B, and A and are open to challenge and refinement, are mutable and extensible, and thus the role interpretive judgment in the final results of validity assessment is transparent within the context of use. For example, new diagnostics, new or corrected baseline observations, and new or corrected measurements could be added to the model in the service of better assessment. Second, the way the results are computed is can be easily subjected to peer-review scrutiny.

The choice of validation metric can significantly influence differences (the validation metric $\| \cdot \|_i$) between validation data B_i and miniapp measurements A_i. For our validation work, we choose metrics based on their ability to emphasize or highlight differences between the application characteristic under consideration and the miniapp measurements.

For the diagnostics used in this study, the metric was usually configured as a relative comparison between the miniapp and the application. However, when available, the comparison is with some absolute comparison.

4. Making the link to full applications

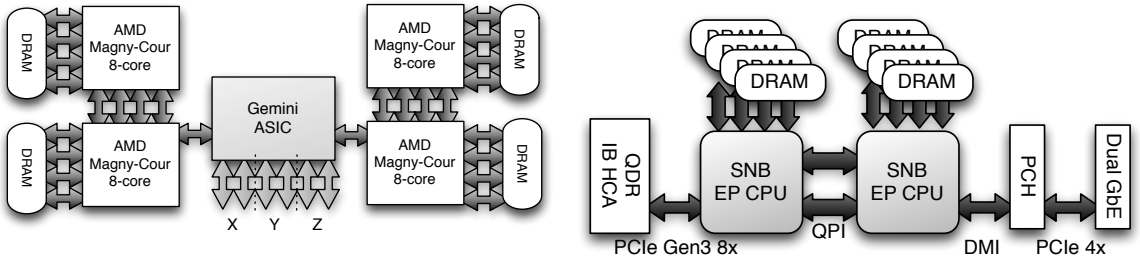
We applied our methodology to four miniapps, examining the linkage between them and an application they are intended to represent. Herein we evaluate the fidelity of that linkage. This represents the initial step required to begin to answer the question, "Under what conditions does a miniapp represent a key performance characteristic in a full app?" Runtime profiling information is presented in support of well-specified key performance issues for each code, providing some acceptable level of confidence that the miniapp is representative of their relevant computations, in a manner that will enable exploration and experimentation using the miniapp.

MiniFE was developed to examine Krylov-based linear equations solution methods applied within the context of a Krylov solver in an implicit finite element method application on unstructured meshes, such as that used in Charon, a semiconductor device simulator. MiniGhost was designed to provide a means for exploring alternatives to the Bulk-Synchronous Parallel programming model with data aggregation inter-process communication strategy, such as that found in CTH, a shock physics code. MiniMD was developed to model the molecular dynamics Lennard-Jones potential, such as that found in LAMMPS. MiniXyce, currently under development, is intended to represent Xyce, a circuit simulation code.

4.1. Experimental Platforms

We have developed and applied our methodology in a broad set of computing environments. Herein we focus on two large scale computers, Cielo and Chama, supplementing the explorations with additional computers where needed to focus on some particular issues. These machines are representative of the ways in which applications take advantage of current capabilities, but also provide hints regarding expected future capabilities.

Cielo, an instantiation of a Cray XE6, is composed of AMD Opteron Magny-Cours oct-core processors, connected using a Cray custom interconnect named Gemini, and a light-weight kernel operating system called Compute Node Linux (CNL). The system consists of 8,944 dual socket compute nodes, for a total of 143,104 cores. Each compute node is divided into two four processor core memory regions, called NUMA nodes (illustrated in Figure 1(a)), connected using HyperTransport version 3.



(a) Cielo/XE6 (b) Chama

Figure 1: Node Architectures

Nodes are connected using Cray's Gemini 3-D torus interconnect. A Gemini ASIC supports two compute nodes. The X and Z dimensions use twice as many links as the Y dimension (24 bits and 12 bits respectively) and therefore introduce an asymmetry to the nodes in terms of bandwidth in the torus. This needs to be taken into account when configuring a system in order to balance the bisection bandwidth of each dimensional slice in the torus. Cielo is configured as a $16 \times 12 \times 24$ 3-D torus. Injection bandwidth is limited by the speed of the Opteron to Gemini HyperTransport link, which runs at 4.4 GT/s. Links in the X and Z dimensions have a peak bi-directional bandwidth of 18.75 GB/s, and the Y dimension peaks at 9.375 GB/s.

Chama, constructed by Appro, Inc. (acquired by Cray in 2012), is designed for production capacity computing by the National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) Trilabs, i.e. Sandia National Laboratories (SNL), Los Alamos National Laboratory, and Lawrence Livermore National Laboratory. The system consists of 1,232 compute nodes, connected by a QLogic InfiniBand fat tree, using 12000 series switches and 7300 series adapters. Each node is composed of two oct-core Intel Xeon E5-2670 Sandy Bridge processors, illustrated in Figure 1(b), for a total of 19,712 cores, running a RHEL operating system.

4.2. A Molecular Dynamics code

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) is primarily a classical molecular dynamics (MD) code and can be run in serial or parallel using a spatial-decomposition of the simulation domain [35, 36]. LAMMPS allows simulation of soft materials, solid-state materials, and coarse-grained or mesoscopic systems. In general, it is a parallel particle simulator at the atomic, meso, or continuum scale.

The spatial-decomposition method that LAMMPS uses divides the physical domain into three dimensional sub-boxes, one per parallel process. Each process computes forces on atoms in its box using information from nearby processes (illustrated in Figure 2). As they migrate among processes, the atoms carry along their molecular topology. Communication is via a nearest-neighbor six-way stencil. Parallel scaling would be N/P if load is perfectly balanced. Computation scales as N/P, communication scales as $(N/P)^{2/3}$ (for large problems), and memory scales as N/P.

MiniMD is designed to model the force computations in typical molecular dynamics applications. The algorithms and implementation used closely mimic these same operations as performed in LAMMPS, with both using the Lennard-Jones potential in our tests.

The work involved in the parallel processing implementation of this model can be categorized as:

1. *Collect off-process ghost atom data ("comm")*: Point-to-point inter-process communication collects off-process ghost atom data onto the owning parallel process.

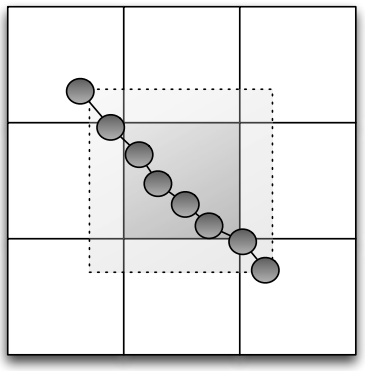


Figure 2: Molecular dynamics computation. Grid represents spatial-decomposition of the physical domain into subdomains for each parallel process. A single process owns the atoms in the central subdomain that shaded dark gray. Data from the atoms in the light gray ghost region are periodically communicated to that process in order to complete the interatomic force calculations.

2. *Construct neighbors list ("neigh")*: Each atom is assigned to a cell in a three-dimensional bin. Then using an inter-atomic cutoff distance, a neighbor list (array `neigh`) is constructed for each parallel process.
3. *Compute forces ("forces")*: Calculate the forces acting upon each atom by the other atoms.

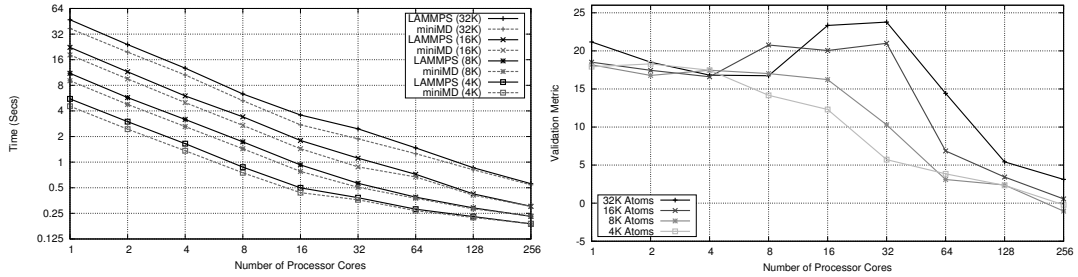
4.2.1. Model Abstractions

As stated throughout this paper, miniapps are not intended to capture all aspects of a particular application. Instead, they are designed to represent issues that critically impact the runtime characteristics of large scale application programs. In addition to enabling a stronger focus on a limited set of important issues in a particular application, this approach allows a miniapp to be representative of more than one application, and in some cases, represent a class of applications and algorithms.

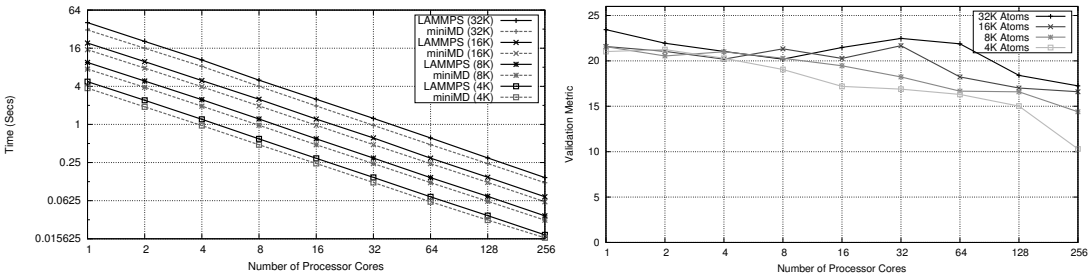
In this case, miniMD does not include the complex logic present in the LAMMPS implementation, resulting in shorter runtimes. Lennard-Jones is one particular type of atomic interaction model that can be used in molecular dynamics. In the future, miniMD may be expanded to include more complex atomic interaction models in order to perform an expanded set of comparisons with LAMMPS. But for now, miniMD cannot explore the broad problem space that is available to LAMMPS, which includes a wide array of complex interatomic interaction potentials and a very long list of auxiliary capabilities (see http://lammps.sandia.gov for more details about LAMMPS's capabilities).

That said, LAMMPS(LJ) and miniMD are quite similar, providing a strong candidate for testing our methodology. This is because the LJ problem and miniMD that we've chosen to simulate are quite representative of a typical MD simulation that an MD practitioner would run, particularly in terms of the computational load and communication patterns. The spatial decomposition, neighborhood list builds, force calculations, radial cutoff, and the time integration simulation of a condensed matter fluid are all common themes that are found in most parallel MD calculations. Further, the computations and data access patterns of the force calculations stress the computational capabilities and memory systems in ways that are important to computations found in many other science domain areas.

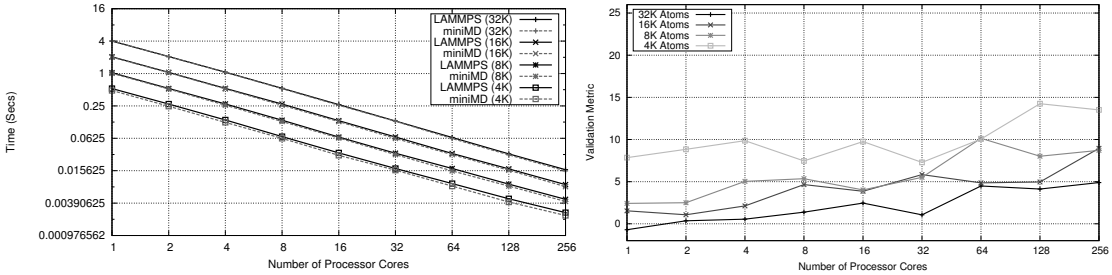
We have chosen to study scaling in a strong sense, where the number of atoms is held constant with increasing core count, instead of weak scaling, where the number of atoms simulated would be proportional to the number of cores used. This choice is intentional and seems more representative of what a typical MD practitioner would want: run a relatively small ensemble of atoms out to as-long-as-possible timescales, rather than simulate a relatively large problem for a short time. Strong scaling is more difficult from a computer science perspective, but more desirable from an MD practitioner's perspective. We note, however, that both LAMMPS and miniMD can be used to study strong or weak scaling, and that the conclusions we draw from this strong scaling app-to-miniapp comparison would likely hold for a weak scaling comparison as well.



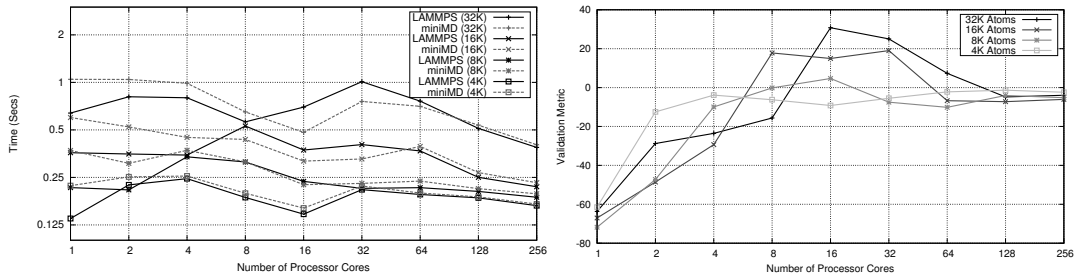
(a) Total Time



(b) Force Compute Time

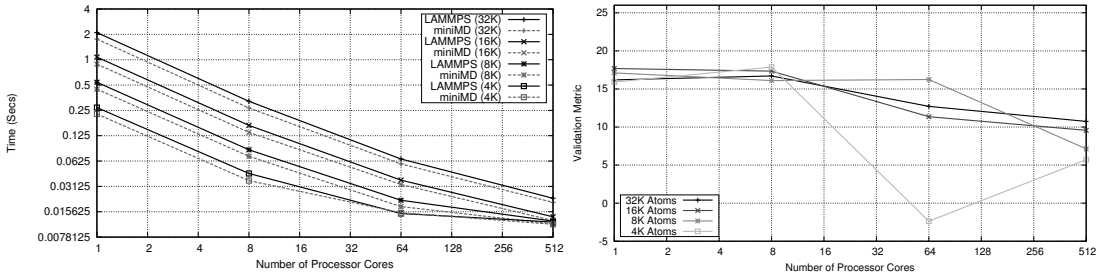


(c) Neighborlist Construction Time

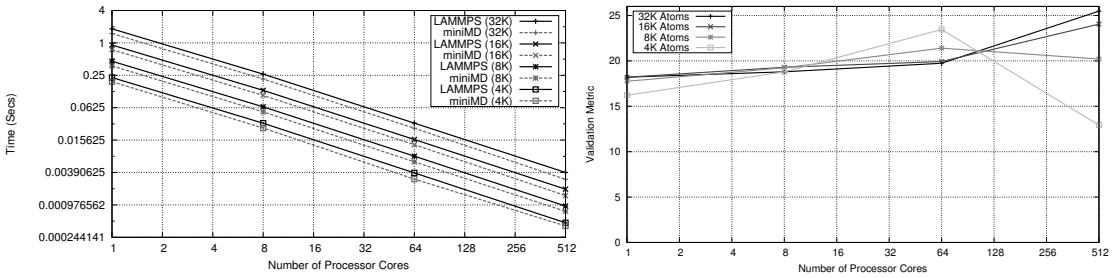


(d) Inter-process Communication Time

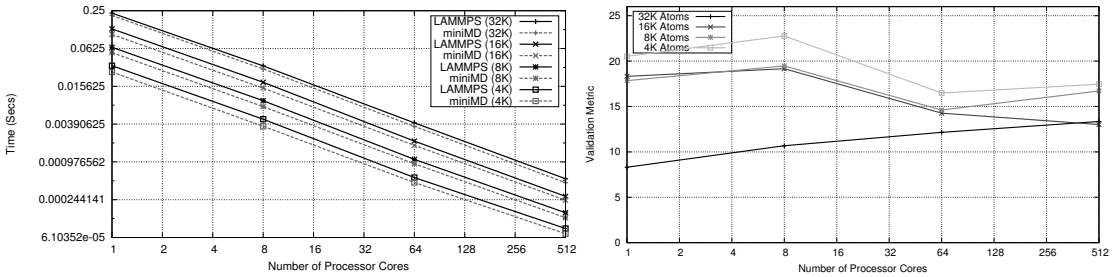
Figure 3: LAMMPS and miniMD Strong Scaling on Muzia: Time (left column) and Validation Metric (right column).



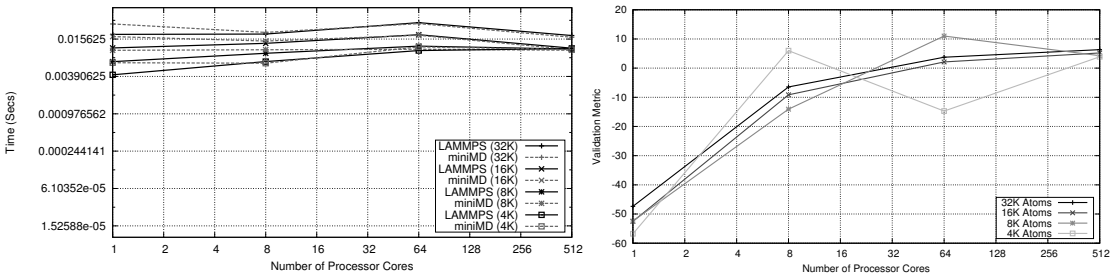
(a) Total Time



(b) Force Compute Time



(c) Neighborlist Construction Time



(d) Inter-process Communication Time

Figure 4: LAMMPS and miniMD Strong Scaling on Chama: Time (left column) and Validation Metric (right column).

Processor Cores	L(32k)	MD(32k)	L(16k)	MD(16k)	L(8k)	MD(8k)	L(4k)	MD(4k)
	32k Atoms		16k Atoms		8k Atoms		4k Atoms	
Nehalem Workstation								
1	0.4	0.3	0.3	0.00	0.5	0.1	0.2	0.5
2	0.5	0.6	1.1	0.00	1.2	0.8	0.4	0.9
4	1.0	2.9	0.6	0.01	1.7	0.8	0.7	1.0
8	4.3	1.0	1.0	0.11	7.1	10.2	4.8	0.0
Muzia								
1	0.84	0.14	0.83	0.06	0.03	0.01	0.01	0.03
2	0.67	0.13	0.03	0.05	0.07	0.09	4.28	0.05
4	0.05	0.13	0.03	0.14	5.26	0.06	5.92	0.03
8	0.05	0.18	0.11	0.17	1.80	0.09	2.72	0.06
16	0.07	0.04	1.72	0.02	6.54	0.03	0.18	0.04
32	0.42	0.98	1.45	0.18	0.38	0.15	0.67	0.11
64	4.02	1.69	1.97	1.51	0.21	1.58	0.31	0.12
128	0.98	0.90	1.73	0.53	0.04	0.62	1.70	0.27
256	0.97	0.21	0.84	0.65	0.83	0.25	0.03	0.12

Table 2: Standard deviations, as a percentage of the time, for LAMMPS and miniMD experiments (denoted by "L" and "MD", resp.).

4.2.2. Performance Domain

The strong connection of miniMD to LAMMPS(LJ) provides a straightforward means of applying our methodology. The diagnostics are defined as the time to solution for each computational phase:

D_1 : Total time

D_2 : Force calculation time

D_3 : Time for construction of neighbor list

D_4 : Time for inter-process communication

Four different problem sets were defined by varying the number of atoms in the simulation (4000, 8000, 16000, and 32000), each using LJ density $\rho^* = 0.8442$, and LJ temperature $T^* = 1.444$, iterated for 1000 time steps, with an LJ time step size of 0.00462.

The performance for this experiment on Muzia (a surrogate for Cielo) and Chama are illustrated in Figures 3 and 4, respectively. These are the minimum times of three trials for each phase. The graphs on the left are direct comparisons, in terms of time. The graphs on the right show the results of our validation methodology. Here we use the normalized form

$$X_i = (B_i - A_i)/B_i, \tag{7}$$

converted to a percentage.

4.2.3. Discussion

Above we've shown the minimum times of three trials for each phase. The average standard deviations are shown in Table 2. Results of similar experiments executed on an Intel Nehalem processor are shown in [6].

Figures 3 and 4 show that miniMD closely mimics LAMMPS performance and scaling characteristics in all cases tested. In nearly every case, miniMD total time is slightly less than LAMMPS total time, with miniMD faster by up to almost 25% and LAMMPS faster in a few cases by a few percent. Examination of each code's respective force calculation inner loops illustrates why miniMD is typically faster: its inner loop logic is simpler. Although miniMD closely mimics the Lennard-Jones algorithm in LAMMPS, it does so within the context of a single, focused goal. MiniMD allows simulation of only a single LJ component with fixed LJ parameters, whereas LAMMPS allows simulation of multi-component LJ materials with user-prescribed LJ parameters. In LAMMPS the algorithm is contained within the goals of a much larger scope, requiring additional complexity so that code may be shared across algorithms, etc. Consequently, LAMMPS's additional flexibility takes a small toll in terms of overall performance. This effect becomes less pronounced at larger core counts due to its falling cost relative to other parts of the calculation.

For both LAMMPS and miniMD, the neighbor list construction times are consistently about 12% of the cost of the force calculations. Since both of these costs are computation dominated and similar in nature, this ratio remains fairly constant versus core count, code, and platform. Both codes have been tuned to use pair distance cutoffs that necessitate neighbor list updates at about 20 timestep intervals, near the minimum total computational cost. Calculation costs other than those for neighbor list construction, force calculations, and inter-processor communication were typically less than 3% of the total and never more than 8% in all of the cases examined.

In these strong scaling tests, we see that computation cost dominates at low core count and communication cost dominates at high core count, as expected. This is because the force and neighbor list construction times exhibit near perfect scaling in these tests whereas the inter-process communication times remain essentially constant with core count. If these tests were extended to even higher core counts, parallel scalability would continue to deteriorate, yielding no advantage to using more cores.

LAMMPS typically outperforms miniMD in terms of inter-process communication time, presumably due to LAMMPS's more sophisticated and highly optimized communication code. Interestingly, at higher core counts, the relative performance in this regard tightens, demonstrating remarkable similarity in scalability between the two codes on both platforms. Since communication costs dominate at higher core counts, and since LAMMPS and miniMD use very similar communication patterns, we expect good agreement between the codes in overall timings near the scaling limit. Indeed we do see remarkably similar performance between the codes at the highest core counts, with the maximum difference in total performance less than 11%.

MiniMD computes the forces acting upon each atom, and therefore the actions of each atom (potentially) on the others. This coupling of actions is reflected in a coupling of the computation, which has implications with regard to the computational capabilities of current processors. Vectorization mechanisms, such as the Streaming SIMD Extensions (SSE) and the Advanced Vector Extensions (AVX) instruction sets can effectively manage these sorts of interactions, but threading approaches, such as pthreads [10] and OpenMP [11], are problematic. MiniMD is serving as a valuable tool for exploring these issues as they pertain to LAMMPS methods.

4.3. A Semiconductor Device Simulation code

Charon is a semiconductor device simulation computer program [17, 29] developed at SNL. It is a transport reaction code used to simulate the performance of semiconductor devices under irradiation. Charon employs the drift-diffusion model, which is a coupled system of nonlinear partial differential equations (PDEs). Finite element discretization (stabilized Galerkin formulation) of these equations in space on an unstructured mesh produces a sparse, strongly coupled nonlinear system. These equations are solved using a Newton-Krylov approach, resulting in a large sparse linear systems of the form

$$AM^{-1}(Mx) = b, \tag{8}$$

for $A \in \mathbb{C}^{N\times N}$, x and $b \in \mathbb{C}^N$, and some preconditioner M.

Using functionality from Trilinos [18], the linear systems are solved either using BiCGStab [43] or GMRes [39] Krylov solver. An algebraic multigrid preconditioner [15], with local incomplete factorization as smoothers significantly improves scaling and performance [26].

An example 2D steady-state drift-diffusion solution is illustrated in Figure 5 for a bipolar junction transistor (BJT).

MiniFE is intended to mimic the finite element assembly (FEA) and linear solver computations for a problem on unstructured meshes. MiniFE solves the steady-state scalar conduction equation as presented in [37]. It assembles finite element matrices into a global matrix and vector, then solves the linear system using the Conjugate Gradient method [21]. Each finite element is a hexahedron with 8 vertex-nodes. These equations are solved on a three-dimensional box of hexahedra.The domain dimensions are in terms of elements, so for example, a $2 \times 2 \times 2$ box describes eight elements each of which has eight nodes, so it is a $3 \times 3 \times 3$ node domain (27 nodes). The coordinate origin is at the corner of the global box where $x = 0, y = 0, z = 0$. The box extends along the positive x-axis, positive y-axis, and the *negative* z-axis. Each node corresponds to a row in the matrix. A global identifier, assigned using coordinates and global box dimensions, adds coding convenience to some aspects of matrix-structure generation and finite-element assembly.

The domain is partitioned using the Recursive Coordinate Bisection method [8], and thus some processors own non-contiguous blocks of global node identifiers. Since it is convenient for matrices and vectors to store contiguously-numbered blocks of rows, global node identifiers are mapped to a separate space of row numbers such that each processor's nodes correspond to a contiguous block of row numbers.

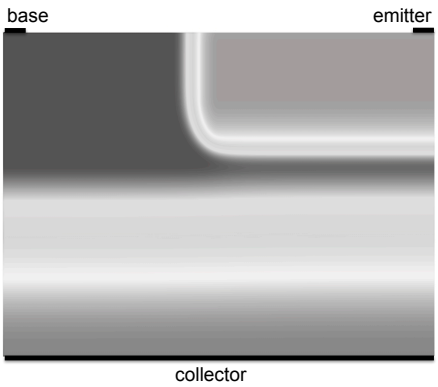


Figure 5: Charon steady-state drift-diffusion solution for BJT with voltage bias. The base, emitter and collector are located in the upper left corner, upper right corner and bottom edge respectively. The shades of gray represent the electric potential, which highest voltage along the bottom and lowest in the upper left portion of domain.

4.3.1. Model abstractions

Charon steady-state drift-diffusion problems (examined herein) have three degrees of freedom (DOF) per mesh node. MiniFE solves a linear scalar equation (one degree of freedom per mesh node). Here MiniFE was configured such that the structure of the linear systems mimic those of Charon. It is to be determined if miniFE can be configured to represent this level of complexity.

Runtime for typical Charon problems is focused in the Newton-Krylov solver. The conjugate gradient solver in miniFE is intended to be sufficiently realistic to be representative of the performance characteristics of the Krylov solver portion in an application code, e.g. to be representative of the scaling of a single Krylov iteration. As miniFE solves a different set of physics, it is not intended to predict the either the number of Krylov iterations required or the number of Newton steps required. For our studies, the number of Krylov iterations for miniFE and a Newton step of Charon will be the same.

4.3.2. Performance Domain

The runtime performance of Charon is dominated by the Newton-Krylov solver, which is itself dominated by the Krylov solver. These computations are characterized by irregular memory accesses, including across nodes, which informs the choice of diagnostics:

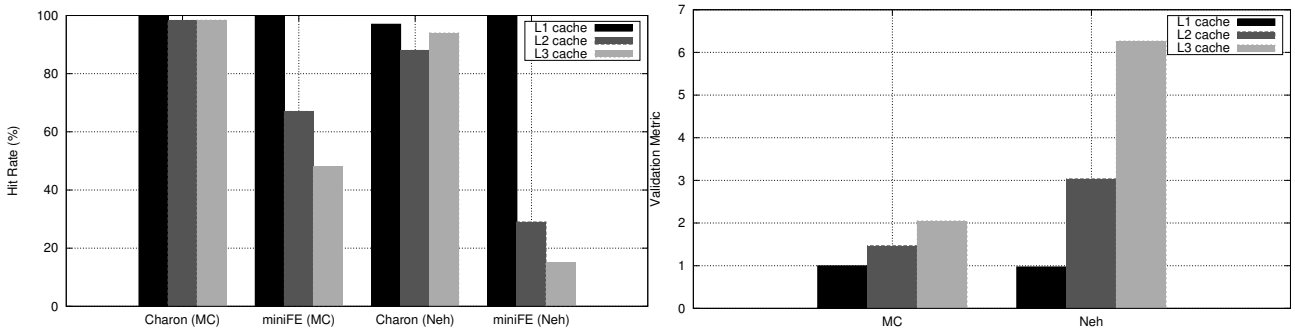
D_1 : Cache hit-to-miss ratio
D_2 : Node memory bandwidth
D_3 : Weak scaling

In addition to the two main target architectures in this study (Cielo and Chama), we include Red Sky and two workstation nodes. Red Sky is similar to Chama in that it is composed of Intel Xeon processors, though the older generation Nehalems, and its InfiniBand interconnect is configured as a three dimensional torus (like Cielo) produced by Mellanox. The individual nodes, described below, enable configuration of different memory speeds.

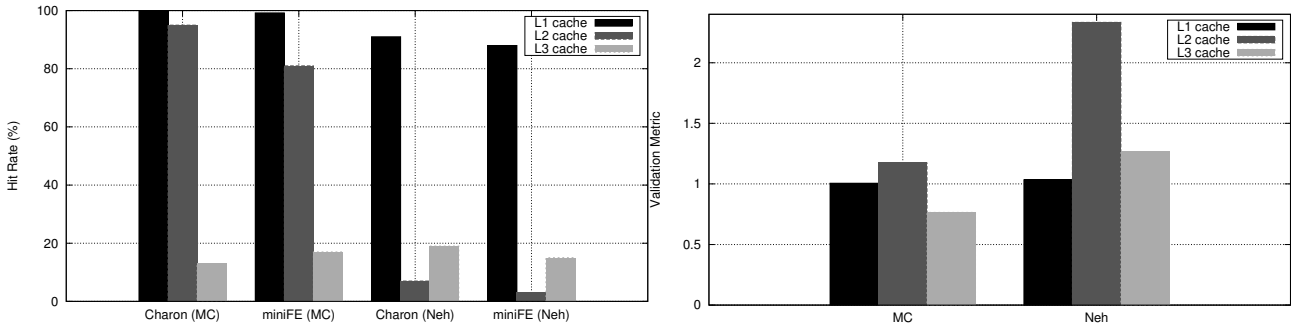
4.3.3. Diagnostic: Cache performance

Caches provide the first defense against on-node memory access constraints, so the impact of cache design is of significant interest. This diagnostic considers cache behavior, again with the separation of between the FEA and solver phases, and again using the Nehalem and Magny-Cours nodes, each with three levels of cache (L3 is shared across cores in a die; one die per socket for Nehalem and two dies per socket for Magny-Cours). The hit rate, defined as the proportion of the number of times the processor finds needed data in a cache with the total number of times it looks for data in that cache, plays a significant role in processor performance. This provides us with an absolute performance, so our validation metric is

$$\frac{H_{app} - H_{miniapp}}{100}, \tag{9}$$



(a) FEA



(b) Solver

Figure 6: Cache behavior of the FEA and solver phases of Charon and miniFE. Hit rates are shown in left column and the validation metrics in the right column.

where H is the measured cache hit rate.

Results are shown in Figure 6. For the FEA phase, Charon and miniFE show strong use of level 1 cache, with a proportional difference of no more than 3%. However, level 2 and 3 hit rates are significantly different, with miniFE being a factor of 3 and 6 times different, respectively, from Charon, leading us to claim that the cache performance of FEA in miniFE is not predictive of that for Charon. For the solver phase, we believe that cache performance is predictive. Although the thresholds for acceptance for level 2 and 3 are arguably high (20%) the trends are clear.

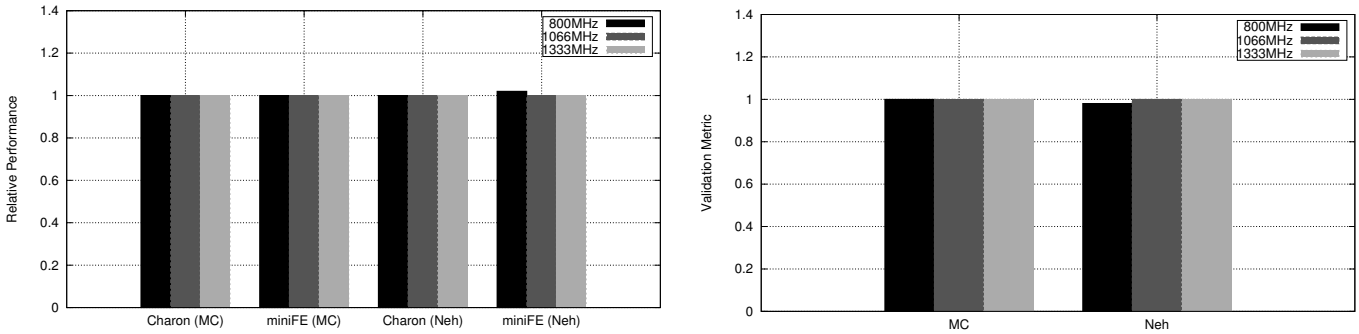
Most interesting is that the Charon/Aztec solver's surprisingly low level 2 hit rate seen on the Nehalem is also seen with miniFE. Given that this is unexpected given Magny-Cours and other past observations, care must be taken with regard to attribution, but given our experience on a variety of processors, this issue is likely due to a hardware configuration issue. Additional experiments are required to make strong causal claims, since it is possible that measurement intrusion is to blame, or perhaps a hardware configuration issue.

4.3.4. Diagnostic: Node memory bandwidth

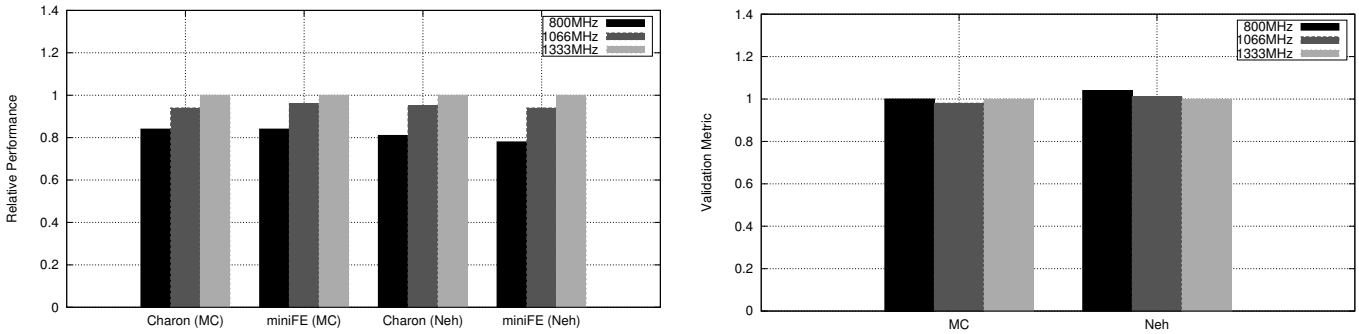
Memory bandwidth within a multicore processor based node is seen as having a significant impact on the performance of many applications [1, 14], including Charon [28]. A typical means for exploring this issue is to vary the number of processor cores employed on the node and comparing the resulting performance efficiency. Although this does vary the amount of memory bandwidth available to a core, it also varies other resources, such as the amount of shared cache. An effective validation study requires stronger evidence to make this claim. However, the fact that the application and miniapp both encountered this issue strengthens the claim that miniFE is representative of the impact of memory speeds on Charon.

Using a dual-socket quadcore Intel Nehalem 5560 clocked at 2.8 GHz processors and a dual-socket 8-core AMD Magny-Cours 6136 clocked at 2.4 GHz, experiments were configured to better focus on memory bandwidth. The machines were configured to provide memory speeds of 800 MHz, 1066 MHz, and 1333 MHz.

The validation metric is defined as follows: we began by normalizing the performance of the FEA and solver performance (in terms of time), with the performance of the fastest memory setting, and then compared those results between Charon and miniFE, shown in Equation 10.



(a) FEA



(b) Solver

Figure 7: Effects of memory speeds on the FEA and solver phases of Charon and miniFE. Times, relative to 1333 MHz are shown in the left column and the validation metrics are in the right column.

$$\frac{t^{charon}(m_{max}) - t^{charon}(m_i)}{t^{charon}(m_{max})} - \frac{t^{minife}(m_{max}) - t^{minife}(m_i)}{t^{minife}(m_{max})}, \quad \text{for} \quad i = 800, 1066, 1330\text{MHz}, \tag{10}$$

where $t(m^{max})$ is the time spent in the computation at the highest memory speed (1330 MHz). That is, we made a relative comparison between the app and miniapp of the normalized absolute comparison within the app or miniapp.

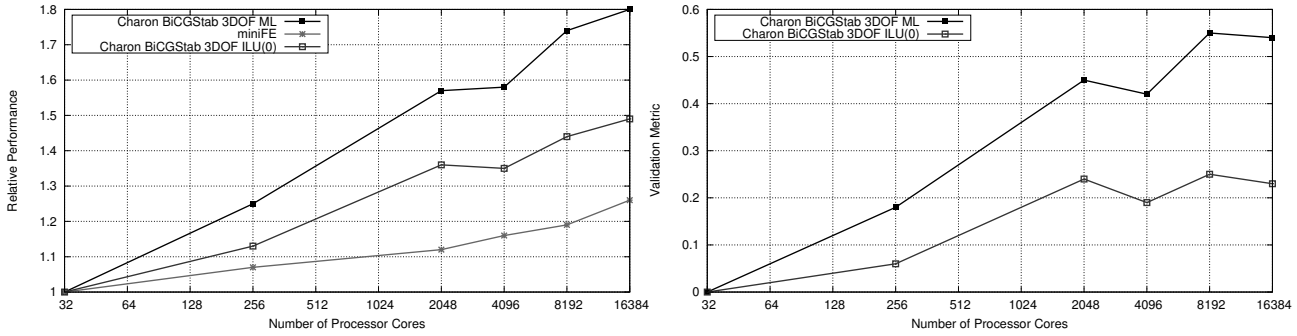
Figure 7 shows the normalized absolute comparison within the app or miniapp. In line with our assumptions regarding the impact of memory speed on these computations, the FEA phases for Charon and miniFE are not impacted by the change in bandwidth, while their solvers are. For the latter, a 66% increase in memory speed resulted in a 19% increase in solver performance on the Magny-Cours processor for both Charon and miniFE, and on the Nehalem, a 23% and 28% increase for Charon and miniFE, respectively.

The validation metrics, also illustrated in Figure 7, shows that miniFE is within 4% of all measures of Charon, leading us to claim that, with regard to on-node memory bandwidth, miniFE is predictive of Charon for these two key performance critical computational phases.

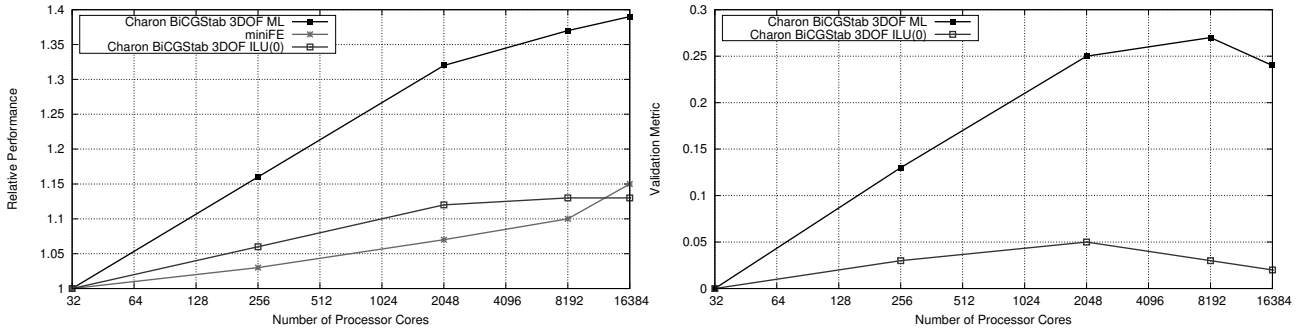
4.3.5. Diagnostic: Weak scaling

Target simulations involving Charon require execution on very large parallel processing architectures. Effective runtime performance in these environments requires managing the inter-node sharing of data. Effective algorithmic behavior requires managing the convergence characteristics of the linear systems through proper choice and application of preconditioning techniques.

Here we examine weak scaling characteristics of Charon and miniFE up to 16k core counts. Diagnostics include the Charon/Aztec BiCGStab solver with two preconditioning strategies, an incomplete factorization algorithm with no fill (ILU(0)) and a multilevel/multigrid (ML) algorithm. Results for each are analyzed in comparison to miniFE, which does not employ a preconditioner. The idea is that Krylov solvers perform common computations (e.g. addition and scaling of vectors, inner products, and sparse matrix-vector products. Further, applications typically use a breadth of preconditioners, so our goal is to understand where specificity is required and where it is not



(a) Chama



(b) Cielo

Figure 8: Charon and miniFE weak scaling for time per Krylov iteration, on Chama and Cielo. Times, relative to 32 processors are shown in the left column and the validation metrics are shown in the right column.

necessary.

Performance is illustrated in Figure 8. We have not yet determined an effective means for analytically comparing scaling behavior. Instead we can reason about the curves by first noting that performance is different on different architectures, which we speculate is a function of the difference preconditioning strategies. Although the difference between Charon and miniFE with multigrid preconditioning is large, this is not reason enough to reject the relationship. Instead, we claim that miniFE is *not* predictive of Charon with multigrid because miniFE does not include the sorts of computations found in multigrid. Further, an analysis of the interprocess message passing requirements shows that multigrid sends over 40% more message per core than do the other configurations.

The difference between Charon with ILU(0) preconditioning and miniFE is less clear, with reasoning driven from the position in the codesign space. For example, from the perspective of some hardware architects, these two approaches are not predictive. However, from the perspective of an algorithm developer perhaps investigating new programming models, miniFE performance could be reasonably predictive. Even with the Charon/Aztec BiCGStab solver being similar to CG in miniFE, there still are substantial differences between the two cases, such as the significant difference in how the domain decomposition is performed. Clearly further investigation into this issue is needed. Therefore we assign this diagnostic a *caution* assessment.

4.3.6. Discussion

The evidence shows that miniFE is representative of the cache and memory bandwidth performance of Charon. Both of these conclusions are aligned with an understanding of the algorithmic implementations of the FEA and Krylov solvers. However, the scaling studies for large numbers of compute nodes shows meaningful differences between Charon and miniFE. In particular, not surprisingly, the behavior of the multigrid preconditioned Charon is not captured by the unpreconditioned miniFE. This is an important consideration for future large-scale simulations which are expected to require such sophisticated preconditioning in order to achieve acceptable convergence.

This work illustrates the importance of the comparison metric. Whereas the time-based diagnostics for LAMMPS and miniMD compelled relative comparisons, the availability of an absolute value for cache performance (100%) elucidated the shared behavior between the miniapp and app, especially important for the L2 cache issue on the Red

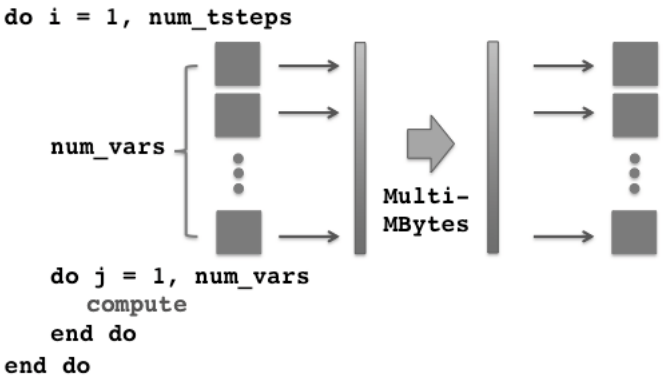


Figure 9: CTH and miniGhost computation and communication interaction

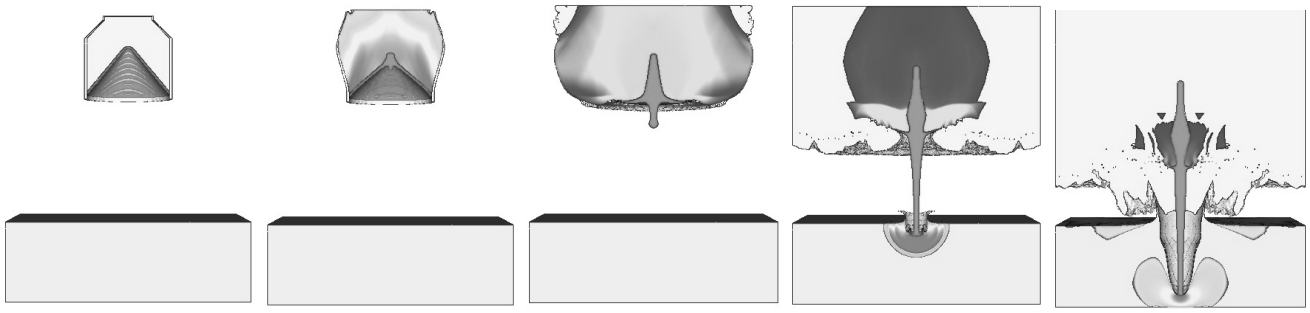


Figure 10: CTH shaped charge simulation. Time progresses left to right.

Sky Nehalem processor. Normalizing the comparison to the fastest memory speed clearly showed the relationship of algorithm to this important architectural capability. The lack of a meaningful metric for the scaling behavior, while a weakness, in this case was overcome by a firm understanding of the algorithmic characteristics, and provided the basis for important discussions by the application team. We intend to work toward a meaningful metric in this case, possibly informed by the work below.

4.4. A Shock Physics code

CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories [20]. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials, using second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results.

Several times within a time step boundary information is aggregated and exchanged with up to six neighbors in the grid of processors. In the BSP/message aggregation (BSPMA) model, data from multiple (logical) memory locations are combined into a user-managed array with other data, then subsequently transmitted to the target process. Illustrated in Figure 9 this work incurs three costs: memory utilization (the message buffers), on-node bandwidth (copies into the buffer), and synchronization (leading up to and including the data transfer).

MiniGhost is designed as a proxy for the boundary exchange functionality (likewise also called ghost- or halo-exchange), embedded into the context of difference stencils. This notion of mapping a continuous problem to discrete space and the inter-process communication requirement induced by spacially decomposing the grid across parallel processes adheres to the bulk-synchronous parallel programming model (BSP [42]), arguably the dominant model for implementing high performance portable parallel processing scientific applications [5].

Two distinct problems are commonly modeled by CTH. The meso-scale impact in a confined space problem is computationally well-balanced across the parallel processes. This problem involves 11 materials, inducing the boundary exchange of 75 variables. The shaped charge problem, illustrated in Figure 4.4, involves four materials, inducing the boundary exchange of 40 variables. (This problem was used in the acceptance testing for the NNSA ASC campaign's capability computer, Cielo [12].) For the shaped-charge problem these messages average 4.1 MB

and for the meso-scale problem these messages average 10.4 MB. Process 0 and a couple of other processors near it for the shaped charge problem have more work since they are the genesis of the explosion and thus have additional work relative to the other processes. The runtime trace shows significantly less waiting time than the other cores.

4.4.1. Model abstractions

CTH is a finite volume code, and thus it has values that are based at cells and at the nodes. Its computations are complex, whereas miniGhost computation is rather simple. Prior to sending a message containing boundary data to a neighbor, a CTH MPI process waits for a message from the target processes, which alerts the sending process that the matching receive is posted. The intent is to avoid unexpected messages, potentially a serious issue given the typically large amount of data transmitted. This is a meaningful approach on some architectures, such as Red Storm [40], but is of no help on most, which include a built-in acknowledgement handshake prior to sending messages. Further, the MPI specification provides functionality managing this. Regardless, our intent is to test the capabilities of the MPI implementation on the target architecture external of some means for adapting to the specific capabilities of a particular implementation. That said, one use of a miniapp is to provide a lower impact means for testing other approaches and ideas.

CTH manages data as sets of two dimensional slices, contained in a single pool of allocated memory. This memory management scheme is a relic of past language constraints. MiniGhost allocates distinct three dimensional arrays, each representing a material.

MiniGhost is intended to capture the behavior of the halo exchange typically employed in finite difference and volume codes. Its ability to do so has beem demonstrated, and has contributed to performance improvements for CTH [7].

4.4.2. Performance Domain

MiniGhost is designed to capture the inter-process communication requirements of CTH. The diagnostics are defined to measure the message passing characteristics of these requirements:

D_1 : Number of communication partners (neighbors)

D_2 : Number of messages per boundary exchange.

D_3 : Message volume (bytes per neighbor)

D_4 : Weak scaling

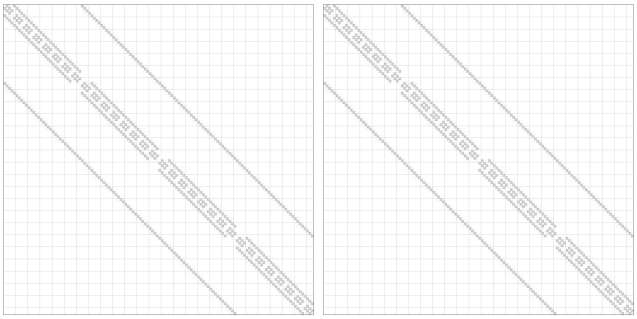
Two problem sets described above (shaped charge and meso-scale) provide the drivers for these measurements. In order to ensure that miniGhost accurately reflects the inter-process communication behavior, it is important to understand the inter-process communication infrastructure, including physical interconnect, logical to physical process maps, system software, and ultimately to how the application manages its communication requirements.

For example, when MPI communication is initiated, buffers are configured for managing message queues, unexpected messages, etc. In the distinct communication and computation phases of the full application, the compute stage touches enough data to ensure that all communication data structures have been flushed from the processor cache hierarchy and must be refetched from main memory upon the initiation of the communication phase. The work in miniGhost must be enough in order for this to occur. Thus for example, a single variable weak scaling problem of dimension $100 \times 100 \times 100$ in 8-byte precision means that an eight MByte variable is operated on, stored into another eight MByte variable. Thus 16 MBytes of memory has been traversed by the processor, effectively flushing a cache of that size.

Alternative methods for moving the data are being explored using miniGhost, and therefore the experimenter must take this into consideration.

4.4.3. Diagnostics: Boundary exchange characteristics

MiniGhost was configured to match the number and relative position of communication partners, the number of variables, and the dimensions of those variables such that the size of the messages involved in the boundary exchange are the same as with CTH. Figure 11(a) illustrates the communication pattern of CTH problems, replicated by miniGhost. Diagnostics D_1, D_2, and D_3 were configured and verified to be equivalent between miniGhost and the CTH problem sets under consideration, thus by any rational metric, the difference would be zero. This lays the foundation for a more meaningful exploration of communication characteristics, described below.



(a) Original mapping (b) Re-ordered mapping

Figure 11: CTH and miniGhost logical processor mapping

Number of	Original Order			Re-ordered		
MPI ranks	X	Y	Z	X	Y	Z
16	0.0	0.0	0.0	0.0	0.0	0.0
32	0.0	0.0	0.0	0.0	0.0	0.0
64	0.0	0.0	0.3	0.0	0.3	0.0
128	0.0	0.0	1.0	0.0	0.5	0.0
256	0.0	0.0	1.0	0.0	0.5	0.3
512	0.0	0.1	2.0	0.0	0.6	0.4
1024	0.0	0.3	2.1	0.2	1.0	0.7
2048	0.0	0.3	2.7	0.3	1.2	1.2
4096	0.0	0.3	3.7	0.3	1.2	1.2
8192	0.0	0.5	5.1	0.2	1.1	2.0
16384	0.0	0.5	4.9	0.2	1.1	2.2
32768	0.0	0.5	5.6	0.2	1.1	2.5
65536	0.0	1.1	10.2	0.2	1.6	2.8
131072	0.0	1.1	10.1	0.2	1.6	3.1

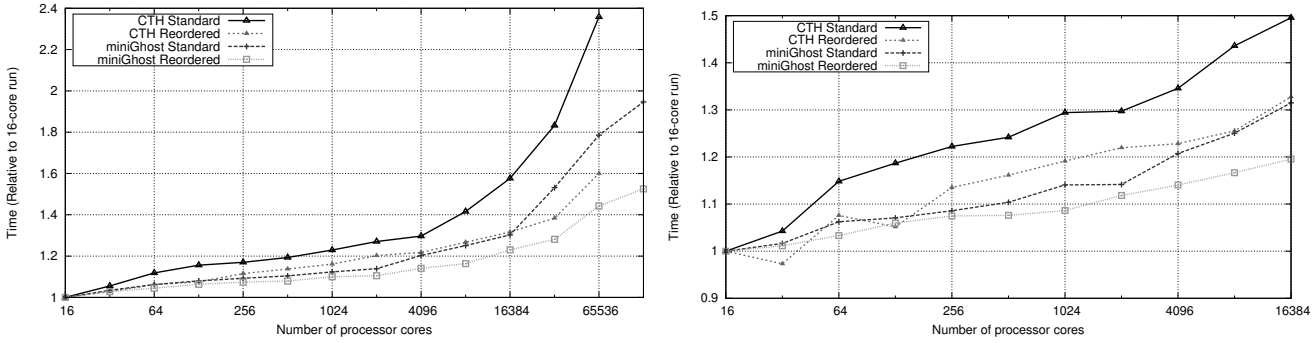
Table 3: CTH and miniGhost average hop counts on Cielo

4.4.4. Diagnostic: Weak Scaling

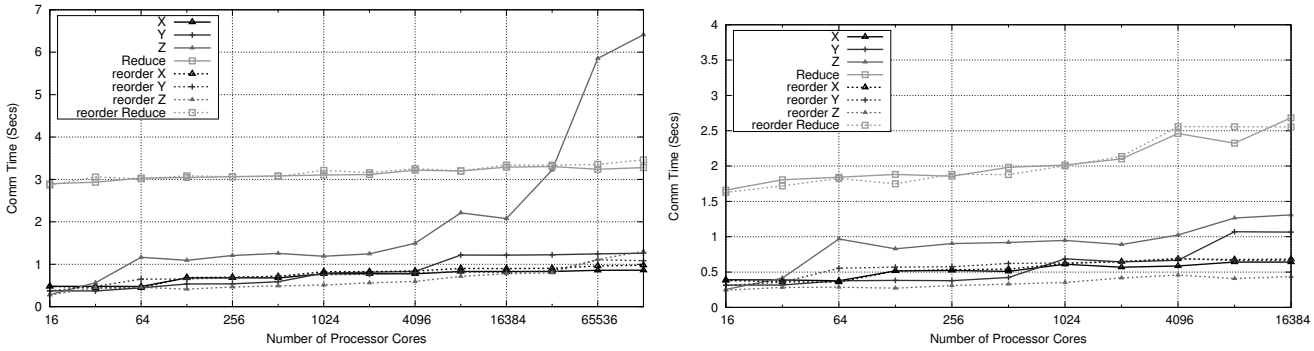
CTH provides a typical example of a code team adapting to computing architectures: in order to avoid message latencies and exploit global bandwidth, computation is performed across as many variables as possible before a boundary exchange involving those variables can be consolidated into a single message per neighbor. But in a recently completed broad-based study of Cielo capabilities [25], and reproduced on Chama, the nearest neighbor boundary exchange encountered significant scaling degradation beyond 8,000 processor cores. This issue is predicted by miniGhost, illustrated in Figure 12(a). Scaling performance is shown relative to that of one node, i.e. 16 processor cores.

The problem was traced to the mapping of the parallel processes to the three dimensional torus topology. Neighbors in the x direction required a maximum of one hop and in the y direction a maximum of two hops. But the number of hops across the network (referred to as the *Manhattan distance*) was shown to increase significantly in the z direction, as shown in Table 3. This combined with the very large messages of a typical CTH problem set (e.g. for the "shaped charge" problem, 40 three dimensional state variable arrays generated message lengths of almost 5 MBytes) resulted in poor scaling beginning at 8k processes, a trend that accelerated after 16k processes.

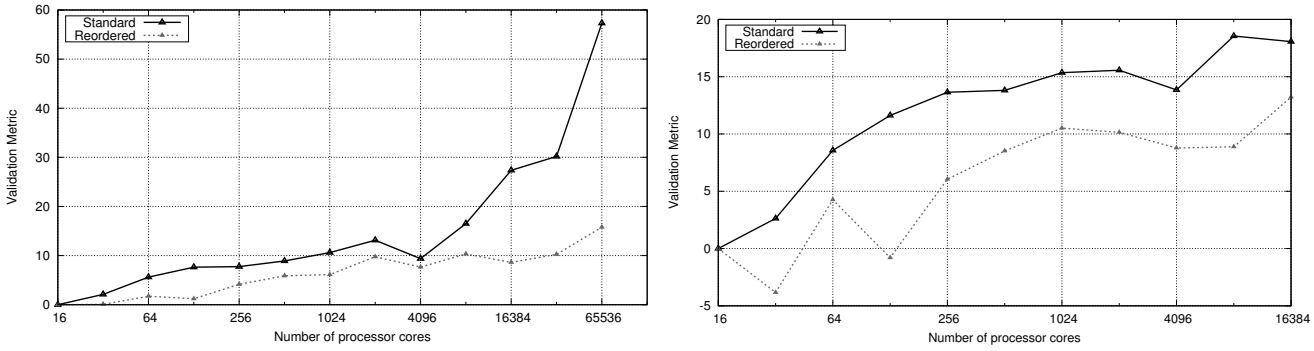
In response, we implemented a means by which the parallel processes could be logically re-ordered to take advantage of the physical locality induced by the communication requirements. In the normal mode, CTH (and miniGhost) assigns blocks of the mesh to cores in a manner which ignores the connectivity of the cores in a node. On Cielo, as with other Cray X-series architectures, cores are numbered consecutively on a node, and this numbering continues on the next node. This process re-ordering strategy, illustrated in Figure 11(b), explored using miniGhost and incorporated into CTH, resulted in a significant improvement in scaling performance. As seen in Figure 12(b),



(a) Relative Scaling



(b) miniGhost inter-process communication time



(c) Validation Metric

Figure 12: Performance of CTH and miniGhost. The graphs on the left are from Cielo, the graphs on the right are from Chama.

this improvement is attributable to a significant improvement in communication time in the z direction. The remaining scaling issue has been traced to synchronization requirements of the boundary exchange, an issue requiring further study.

The validation metrics for Cielo and Chama are defined as the difference between the scaling normalized to 16 cores, as follows:

$$\frac{t_i^{cth}}{t_{16}^{cth}} - \frac{t_i^{minighost}}{t_{16}^{minighost}}, \quad \text{for} \quad \text{time} \quad t; \quad i = 1, \ldots, \text{numpes}. \tag{11}$$

The comparison is shown in Figure 12(c). Two things are interesting to note. First, the process re-ordered versions have stronger agreement. Second, at least for Cielo where higher process scales were possible, that agreement remained longer as processor counts increased. This is not surprising since the the increasing Manhattan distance is expected to more strongly impact a more complex code. However, because of the observations of the Manhattan distances, before and after re-ordering, combined with the impact on the communication times, as well as additional runtime profiling that showed the computation time remained constant and the node interconnect stalls decreased [7], we are confident that miniGhost is predictive of the weak scaling behavior of CTH.

4.4.5. Discussion

Three diagnostics, D_1, D_2, and D_3, were defined in order to configure miniGhost to capture the general CTH inter-process communication requirements. This created an application relevant context for exploring CTH's communication behavior.

An issue with scaling performance, tracked to the way in which processes were physically mapped onto a computer, was identified and effectively addressed. We intend to explore the ability of miniGhost to represent the inter-process communication behavior of similar applications that employ dynamic meshes, i.e. adaptive mesh refinement (AMR). This will provide challenges for these diagnostics, but they may inform a process reordering strategy that would improve the application's scaling behavior. If this is not the case, AMR may need to be added to miniGhost. Either way, this will compel the definition of an additional diagnostic involving the Manhattan distance.

4.5. A Circuit Simulation code

Xyce is a circuit modeling tool [24] developed at Sandia National Laboratories. It is designed to perform transistor-level simulations for extremely large circuits on large-scale parallel computing platforms of up to thousands of processors. Xyce is a traditional analog-style circuit simulation tool, similar to the Berkeley SPICE program[30].

Circuit simulation adheres to a general flow, as shown in Fig. 13. The circuit, described in a netlist file, is

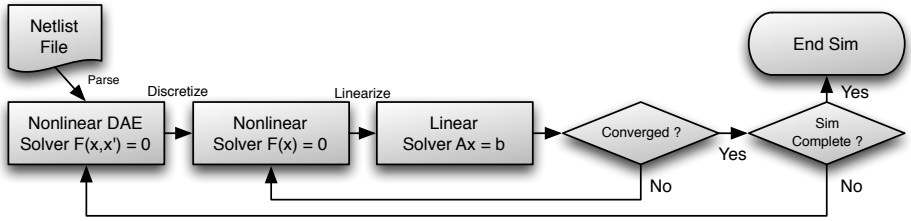


Figure 13: General Circuit Simulation Flow in Xyce

transformed via modified nodal analysis (MNA) into a set of nonlinear differential algebraic equations (DAEs)

$$\frac{dq(x(t))}{dt} + f(x(t)) = b(t), \tag{12}$$

where $x(t) \in \mathbb{R}^N$ is the vector of circuit unknowns, q and f are functions representing the dynamic and static circuit elements (respectively), and $b(t) \in \mathbb{R}^M$ is the input vector. For any analysis type, the initial starting point is this set of DAEs. The numerical approach employed to compute solutions to Equation (12) is predicated by the analysis type.

Circuit simulation can be coarsely divided into three phases that play a distinct role in the overall performance of a simulation. The first two phases result naturally from the evaluation and the solution of the circuit Equation (12)

and are called the "Device Evaluation" and "Linear Solve" phases. In general, load balancing each of these two phases has competing objectives, indicated by Fig. 14, which requires a rebalancing of the problem between those phases.

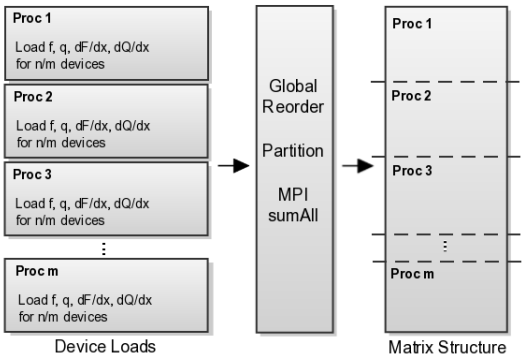


Figure 14: Different Load Balance/Partitioning for Device Evaluation and Linear Solve

The relative amount of time spent in each of those two phases is problem-dependent. For smaller problems, the device evaluation phase should dominate the runtime, especially when the circuit includes modern transistors. As the problem size increases, the linear solve phase will dominate, as it should scale super-linearly, while the device evaluations should scale linearly. This is because linear solution methods (whether they be direct or iterative) are generally communication intensive, while the communication volume required during the device evaluations is relatively small.

As a result, the device evaluation phase has historically been naively balanced by taking into account only the computational work required, while the matrix partitioning has been designed to minimize communication volume. How this communication volume is measured and how it is optimized is an active area of research for many types of numerical simulation problems. Since the device evaluation and linear solve phases have different load balance requirements, Xyce has been designed to have completely different parallel partitioning for each. A simplified representation of this is shown in Fig. 14.

The third phase is "Parsing", where the hierarchical netlist file, describing the network elements and connectivity, is read in and the set of DAEs is constructed and partitioned across processors. In the total runtime of a simulation the netlist parsing doesn't take a large percentage, but the decisions made about partitioning devices over processors in this phase can possibly have a significant effect for emerging architectures, making it a phase worth studying.

Given the hierarchical structure possible in the netlist file, parsing is a largely serial process, where devices are naively partitioned according to a "first-come-first-served" basis. This process is not guided by circuit topology or computational cost of the individual device evaluations, which can vary widely. This design has not been troublesome on the distributed-memory machines that Xyce was designed for. However, future architectures may prove this approach is too simplistic.

4.5.1. Model Abstractions

At this time, miniXyce is a simple linear circuit simulator with a basic parser that performs transient analysis on any circuit with resistors (R), inductors (L), capacitors (C), and voltage/current sources. The parser incorporated into this version of miniXyce is a single pass parser, where the netlist is expected to be flat (i.e. no hierarchy via subcircuits is enabled). Simulating the system of DAEs generates a nonsymmetric linear problem, which is solved using GMRes [39], without preconditioning. The time integration method used in miniXyce is backward Euler with a constant time-step.

The development of the first version of miniXyce resulted in something closer to a compact application than a miniapp since more focus was put on the simulator returning the correct answer, than modeling performance characteristics of interest. Further analysis of Xyce has called out particular performance issues in the three phases discussed in Section 4.5. These issues will inspire enhancements to, and a second version of, miniXyce. For complex simulation codes, developing a representative miniapp will likely require an iterative process, where performance issues are investigated in order of an application-based priority.

4.5.2. Model Enhancements

Enhancements to the first version of miniXyce will focus on two of the three phases discussed in Section 4.5: "Parsing" and "Device Evaluation". The third phase, the "Linear Solve", shares performance characteristics and issues with other implicit application codes, like Charon. While this phase dominates the runtime for large-scale circuits, it is uncertain if focusing on that aspect of miniXyce will be a duplication of effort with miniFE. However, the "Parsing" and "Device Evaluation" phase are unique to circuit simulation. By focusing on these two phases, it is anticipated that miniXyce will provide a different performance characterization than any other miniapp. Alternatively, it will give Xyce a unique opportunity to explore the impact of naively partitioning the network and devices.

5. Summary and future work

Miniapps have been shown to provide a tractable means for rapid exploration of a broad set of issues associated with effectively executing large scale scientific and engineering applications on current, emerging, and future architectures. As such, it is crucial to have data-driven evidence of what the miniapp is able to represent, and just as important, a firm understanding of what the miniapp is not able to represent. We presented a methodology for understanding the relationships between an application and the application proxy (the miniapp) in the manner in which it is intended to represent.

All validation work must be driven by usage requirements, which requires a strong understanding of the system under study. Here, runtime profiling points us to the important performance portions of an application, informing a choice of diagnostics. Configuring the appropriate validation metric is critical to elucidating the differences in between the application and the miniapp. We used a variety of formulas for this purpose, in some cases driven by incorrectly chosen metric, in order to clarify differences. The thresholds for acceptance are the most nebulous part of this process. Unlike experimental validation, where a threshold is specified, the threshold in our methodology is viewed more as a guideline than a hard number. Given this, the methodology, in a real sense, provides a formal framework for discussion, interpretation, and challenge. Therefore the process for accessing the miniapp to application connection must be open, iterative, and refined.

The goal of miniapps is to provide actionable information to the application developer. Four distinct applications provided the means for demonstrating this link. MiniMD, closely aligned with the application (LAMMPS) it is intended to represent, enabled calibration of our approach. MiniGhost provides a means for focusing on the inter-process communication requirements of a finite difference or finite volume based application (here CTH), with the computation serving to create a relevant separation between the transmission of data between the parallel processes. Within certain limits, miniFE provides a means to investigate linear solver performance for an implicit finite element method on an unstructured mesh application code (e.g. Charon; for a more detailed study see [27]). Through this work it was determined that miniXyce requires additional analysis, design, and implementation in order for it to serve the performance goals of the Xyce team.

Several issues for each application were found that will require further study. For example, the behavior of the multigrid preconditioner in Charon must be added to miniFE. Our work with molecular dynamics illustrates the need for algorithms in addition to Lennard-Jones. MiniGhost, executed at very large scale, pointed to an issue that will need to considered as anticipated improvements to interconnect technologies are realized.

Development of additional miniapps in Mantevo is progressing in several areas: a solid mechanics contact algorithm, adaptive refinement of an Eulerian mesh (AMR), some graph-based applications, a new miniapp for Xyce, and others. This work is being meaningfully driven by our validation methodology, guiding implementation strategies and configurations, and enabling an integrated code development model that provides a shorter feedback loop.

Finally, its important to note that just as the methodology provides a means for building up a body of evidence for a miniapp's predictive capabilities, we anticipate refining the methodology as it is exercised in an increasingly broader context. In particular, the choice of metric can significantly alter the apparent outcome. In this work we have purposely used somewhat simplistic measures. However, future work will examine the use of more sophisticated metrics. Validation is a never-ending process of discovery.

Acknowledgements

Support for this work was provided through the Advanced Simulation and Computing (ASC) program funded by U.S. Department of Energy's National Nuclear Security Agency. This effort was greatly enhanced by interactions

with staff throughout Sandia as well as many external organizations. It is heartening to discover the active interests in this work, supported by broad and deep expertise, of the computational science community.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] S.R. Alam, R.F. Barrett, J.A. Kuehn, P.C. Roth, and J.S. Vetter. Characterization of Scientific Workloads on Systems with Multi-core Processors. In *IEEE International Symposium on Workload Characterization*, 2006.

[2] S. Amarasinghe et al. ASCR Programming Challenges for Exascale Computing. In *Report of the 2011 Workshop on Exascale Programming Challenges*, July 2011.

[3] S.E. Anderson and P.R. Woodward. simplified PPM. Laboratory for Computational Science and Engineering, University of Minnesota, 1998.

[4] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, D. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] R.F. Barrett, S. Ahern, M.R. Fahey, R. Hartman-Baker, J.K. Horner, S.W. Poole, and R. Sankaran. A Taxonomy of MPI-Oriented Usage Models in Parallelized Scientific Codes. In *The International Conference on Software Engineering Research and Practice*, 2009.

[6] R.F. Barrett, P.S. Crozier, D.W. Doerfler, S.D. Hammond, M.A. Heroux, P.T. Lin, H.K. Thronquist, T.G. Trucano, and C.T. Vaughan. Summary of Work for ASC L2 Milestone 4465: Characterize the Role of the Mini-Application in Predicting Key Performance Characteristics of Real Applications. Technical Report SAND2012-4667, Sandia National Laboratories, 2012.

[7] R.F. Barrett, C.T. Vaughan, S.D. Hammond, and D. Roweth. Reducing the Bulk of the Bulk Synchronous Parallel Model. *Parallel Processing Letters*, 23(4), December 2013.

[8] M.J. Berger and S.H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. Comput.*, 36:570–580, May 1987.

[9] Keren Bergman et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor and Study Lead, 2008.

[10] D. Buttlar, B. Nichols, and J.P. Farrell. *pthreads Programming.* O'Reilly & Associates, Inc., 1996.

[11] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46 –55, 1998.

[12] D.W. Doerfler, M. Rajan, C. Nuss, C. Wright, and T. Spelce. Application-Driven Acceptance of Cielo, an XE6 Petascale Capability Platform. In *Proc. 53rd Cray User Group Meeting*, 2011.

[13] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency Computat.: Pract. Exper.*, 15:803–820, 2003.

[14] J.J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The Impact of Multicore on Computational Science Software. *CTWatchQuarterly*, 3(1), February 2007.

[15] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, and M.G. Sala. ML 5.0 Smoothed Aggregation User's Guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[16] A. Geist and S. Dosanjh. IESP Exascale Challenge: Co-Design of Architectures and Algorithms. *Int. J. High Perform. Comput. Appl.*, 23:401–402, November 2009.

[17] G.L. Hennigan, R.J. Hoekstra, J.P. Castro, D.A. Fixel, and J.N. Shadid. Simulation of Neutron Radiation Damage in Silicon Semiconductor Devices. Technical Report SAND2007-7157, Sandia National Laboratories, 2007.

[18] M. A. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software*, 31:397–423, September 2005.

[19] M.A. Heroux, D.W. Doerfler, P.S. Crozier, J.M. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E.R. Keiter, H.K. Thornquist, and R.W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009.

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

[21] M.R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[22] Accelerated Stratigic Computing Initiative. The ASCI SWEEP3D Benchmark Code. `www.llnl.gov/asci_benchmarks`, 1995.

[23] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, August 2013.

[24] E.R. Keiter and others. Parallel Transistor-Level Circuit Simulation. In Peng Li, Luis Miguel Silveira, and Peter Feldmann, editors, *Advanced Simulation and Verification of Electronic and Biological Systems*. Springer, 2011.

[25] S.M. Kelly et al. Report of Experiments and Evidence for ASC L2 Milestone 4467 - Demonstration of a Legacy Application's Path to Exascale. Technical Report SAND2012-1750, Sandia National Laboratories, 2012.

[26] P.T. Lin. Improving Multigrid Performance for Unstructured Mesh Drift-Diffusion Simulations on 147,000 cores. *International Journal for Numerical Methods in Engineering*, 91:971–989, 2012.

[27] P.T. Lin, M.A. Heroux, A.B. Williams, and R.F. Barrett. Assessing a mini-application as a performance proxy for a finite element method engineering application code. *In preparation*.

[28] P.T. Lin and J.N. Shadid. Towards Large-Scale Multi-Socket, Multicore Parallel Simulations: Performance of an MPI-only Semiconductor Device Simulator. *Journal of Computational Physics*, 229(19):6804–6818, 2010.

[29] P.T. Lin, J.N. Shadid, M. Sala, R.S. Tuminaro, G.L. Hennigan, and R.J. Hoekstra. Performance of a Parallel Algebraic Multilevel Preconditioner for Stabilized Finite Element Semiconductor Device Modeling. *Journal of Computational Physics*, 228(17):6250–6267, 2009.

[30] L.W. Nagel. SPICE 2, a Computer Program to Simulate Semiconductor Circuits. Technical Report Memorandum ERL-M250, University of California, Berkley, 1975.

[31] L.W. Nagel and D.O. Pederson. SPICE (Simulation Program with Integrated Circuit Emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973.

[32] W.L. Oberkampf and C.J. Roy. *Verification and Validation in Scientific Computing*. Cambridge University Press, 2010.

[33] W.L. Oberkampf and T.G. Trucano. Verification and Validation in Computational Fluid Dynamics. *Progress in Aerospace Sciences*, 38, 2002.

[34] M. Pilch, T.G. Trucano, J. Moya, G. Froehlich, A. Hodges, and D. Peercy. Guidelines for Sandia ASCI Verification and Validation Plans - Content and Format: Version 2.0. Technical Report SAND2000-3101, Sandia National Laboratories, 2000.

[35] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117, 1995.

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

[37] J.N. Reddy and D.K. Gartling. *The Finite Element Method in Heat Transfer and Fluid Dynamics*. CRC Press, 2nd edition, 2001.

[38] A.F. Rodrigues, R.C. Murphy, P. Kogge, and K.D. Underwood. The structural simulation toolkit: exploring novel architectures. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'06)*, page 157, New York, NY, USA, 2006. ACM.

[39] Y. Saad and M.H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[40] J.L. Tomkins, R. Brightwell, W.J. Camp, S. Dosanjh, S.M. Kelly, P.T. Lin, C.T. Vaughan, J. Levesque, and V. Tipparaju. The Red Storm Architecture and Early Experiences with Multi-core Processors. *International Journal of Distributed Systems and Technologies (IJDST)*, 1(2):74–93, May 2010.

[41] T.G. Trucano, M. Pilch, and W.L. Oberkampf. General Concepts for Experimental Validation of ASCI Code Applications. Technical Report SAND2002-0341, Sandia National Laboratories, 2002.

[42] L.G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33:103–111, August 1990.

[43] H. van der Vorst. Bi-CGSTAB: a Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.