

# SANDIA REPORT

SAND2012-3840  
Unlimited Release  
Printed May, 2012

## Navigating An Evolutionary Fast Path to Exascale – Expanded Version

R.F. Barrett, S.D. Hammond, C.T. Vaughan, D.W. Doerfler, M.A. Heroux

Center for Computing Research  
Sandia National Laboratories  
Albuquerque, NM 87185  
email: rfbarre,sdhammo,ctvaugh,dwdoerf,maherou@sandia.gov

J.P. Luitjens, NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA,  
95050 jluitjens@nvidia.com

D. Roweth, Cray UK Ltd, 2 Brewery Court, High Street, Theale, Reading, RG7  
5AH, United Kingdom, droweth@cray.com

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

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.

Approved for public release; further dissemination unlimited.



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

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

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](mailto: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](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



## Navigating An Evolutionary Fast Path to Exascale – Expanded Version

The computing community is in the midst of a disruptive architectural change. The advent of manycore and heterogeneous computing nodes forces us to reconsider every aspect of the system software and application stack. To address this challenge there is a broad spectrum of approaches, which we roughly classify as either revolutionary or evolutionary. With the former, the entire code base is re-written, perhaps using a new programming language or execution model. The latter, which is the focus of this work, seeks a piecewise path of effective incremental change. The end effect of our approach will be revolutionary in that the control structure of the application will be markedly different in order to utilize single-instruction multiple-data/thread (SIMD/SIMT), manycore and heterogeneous nodes, but the physics code fragments will be remarkably similar.

Our approach is guided by a set of mission driven applications and their proxies, focused on balancing performance potential with the realities of existing application code bases. Although the specifics of this process have not yet converged, we find that there are several important steps that developers of scientific and engineering application programs can take to prepare for making effective use of these challenging platforms. Aiding an evolutionary approach is the recognition that the performance potential of the architectures is, in a meaningful sense, an extension of existing capabilities: vectorization, threading, and a re-visiting of node interconnect capabilities. Therefore, as architectures, programming models, and programming mechanisms continue to evolve, the preparations described herein will provide significant performance benefits on existing and emerging architectures.

# Acknowledgment

The breadth of our work has required special efforts from a variety of entities and staff within the Department of Energy and our industrial collaborators. The test beds used for this research are funded by the Department of Energy's NNSA ASC program and the Office of Science Advanced Scientific Computing Research (ASCR) program. We acknowledge support from AMD Inc, Cray Inc and NVIDIA for providing detailed information on hardware platforms and information relating to optimization opportunities.

Ted Barragy of AMD Inc, and John Levesque and Jeff Larkin of Cray Inc, provided invaluable information and suggestions.

The CapViz team and others at Sandia provided the special skills and attention required to maintain and support the testbed systems used in this work.

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Related work.....	11
<b>2</b>	<b>Programming Models and Environments</b>	<b>13</b>
<b>3</b>	<b>Methodology</b>	<b>15</b>
<b>4</b>	<b>Case Study: Initial Porting of Mantevo Mini-Applications to Future Computing Architectures</b>	<b>17</b>
4.1	Processor Core Performance .....	17
4.2	Intra-node performance .....	19
4.2.1	Finite difference stencils .....	20
4.2.2	MiniFE on a GPU .....	21
4.2.3	The Impact of Memory Speeds .....	23
4.3	Inter-node performance .....	24
4.3.1	On the XK6 .....	25
4.3.2	Mapping processes to processors .....	26
4.3.3	Alternative communication strategies .....	29
<b>5</b>	<b>Conclusions and Future Work</b>	<b>33</b>
	<b>References</b>	<b>34</b>

# Appendix

<b>A</b>	<b>Programming Environment</b>	<b>39</b>
A.1	Cielo .....	39
A.2	Curie .....	39
A.3	GPU .....	40
A.4	Teller .....	40
A.5	Dual-Socket, Oct-core AMD 2.4GHz Magny-Cours 6136 .....	40
A.6	Dual-Socket, Quad-core Intel 2.93GHz Nehalem 5570 .....	40
<b>B</b>	<b>MiniMD code examples</b>	<b>41</b>
<b>C</b>	<b>MiniGhost code examples</b>	<b>45</b>
<b>D</b>	<b>NVIDIA miniFE Study</b>	<b>49</b>
D.1	Introduction .....	49
<b>E</b>	<b>Characterizing the Sensitivity of Charon and MiniFE to Variation in DDR Interface Frequency on Workstation-Class Multicore Processor Architectures</b>	<b>53</b>
E.1	Introduction .....	53
E.2	Test Beds and Method .....	53
E.3	Results .....	54
E.4	Summary and Conclusions .....	55
<b>F</b>	<b>Remapping the parallel processes in CTH</b>	<b>61</b>
<b>G</b>	<b>More on Cielo</b>	<b>63</b>

# List of Figures

3.1	Steps in an Evolutionary Path to Exascale Readiness . . . . .	15
4.1	Performance of varying miniGhost problems on an XK6 node . . . . .	20
4.2	Speedup of miniFE CUDA Implementation (NVIDIA Fermi M2090 vs. Hex-Core 2.7GHz E5-2680 . . . . .	21
4.3	Strong and Weak Scaling of MiniGhost on XK6 . . . . .	26
4.4	Cielo process map . . . . .	27
4.5	Performance of MiniGhost with MPI-rank remapping on Cielo . . . . .	29
4.6	CTH boundary exchange and computation . . . . .	30
4.7	Performance of miniGhost Communication Strategies on Cielo . . . . .	31
D.1	Speedup of miniFE CUDA Implementation (NVIDIA Fermi M2090 vs. Hex-Core 2.7GHz E5-2680 . . . . .	51
E.1	Memory Bandwidth: miniFE Finite Element Mini-Application . . . . .	57
E.2	Memory Bandwidth: Charon Device Simulation Application . . . . .	58
E.3	Memory Bandwidth: miniFE compared to Charon . . . . .	59
F.1	Performance of CTH with MPI-rank remapping on Cielo. . . . .	62
G.1	Cielo XE6 architecture. Image courtesy of Cray Inc. . . . .	63
G.2	The XE6 compute node architecture. Images courtesy of Cray, Inc. . . . .	64
G.3	The XE6 Gemini architecture. Images courtesy of Cray, Inc. . . . .	65

# List of Tables

4.1	Speedup of manual vectorization for miniMD force calculation and full application runtime. . . . .	18
4.2	Relative Runtime Slowdown of miniFE and Charon from Reduced Memory Frequency (Relative to Memory Frequency of 1333MHz, Lower is Worse) . . . .	24
4.3	miniGhost average hop counts on Cielo . . . . .	28
E.1	Test Bed Descriptions . . . . .	54

# Chapter 1

## Introduction

High Performance Computing (HPC) architectures of the coming decade will be significantly different in structure and design than today. We have already seen clock rates and node counts stabilize, and core counts increase. Now emerging are increased vector lengths, greater levels of hardware-enabled concurrency and new memory architectures that are strongly non-uniform and may soon lose cache coherency. Adoption of new, potentially immature, technologies presents many challenges including hardware reliability, scalability and, in the case of many proposed technologies, programmability and performance portability. Since proposed designs represent a radical departure from existing petascale technologies, new research projects have started in order to identify and develop solutions for many of these problems. One of the greatest concerns facing programs such as the Advanced Simulation and Computing (ASC) initiative in the United States is how best to port full applications that have been developed over nearly two decades. These applications are considerable in size, with many comprising of millions of lines of source implemented in a variety of programming languages (some of which use non-standard features) and utilize tens of supporting libraries. ASC in particular, many of these applications codify significant bodies of knowledge which have developed over multiple generations of scientists. Alongside the demands of porting such large codes are the continued requirements associated with on-going programmatic-level work. Put simply, science discovery cannot stop while applications and algorithms are rewritten for future systems and re-validated to ensure correct scientific output.

In this context, many in the HPC industry question how full science applications will be ported to new architectures. On one side of the debate is a view that significant application rewrites will be required in order to obtain the full potential performance of new hardware. On the other side is a view, described above, that the pedigree of existing code and the complex science which it embodies must be gradually evolved and augmented over time for new platforms so that scientific delivery can continue and the cost associated with porting is reduced or at least amortized. We regard these views as the *revolutionary* and *evolutionary* approaches to Exascale, respectively.

The work reported in this paper (and in expanded version in [5]) describes an initial series of experiments performed in the *evolutionary* context. We note that studies using revolutionary approaches are also underway and will be reported in future publications. Our work focuses on three levels of porting which we expect to be commonplace choices for the modification of codes on future platforms: (i) optimization within the processor core, typi-

cally investigating the improvement of vector-level parallelism and the modification of code to exploit near memory subsystems; (ii) optimization of code for the compute node as a whole using techniques such as thread-level parallelism, introduction of compiler directives to drive compute offloading, data motion reduction and adaptation for node-level topologies such as non-uniform memory architecture (NUMA) domains, and, (iii), optimization of inter-node communication to improve message pipelining or to utilize novel features in advanced network interconnects.

The specific contributions of this work are:

- Documented porting and analysis of several key Sandia mini-applications (miniapps) running on advanced computing architectures. The use of miniapps from the Mantevo suite enables us to draw conclusions that are relevant to production codes used by the National Nuclear Security Agency (NNSA) and ASC programs. We employ novel technologies to aid in this porting including traditional OpenMP pragmas as well as introduction of OpenACC directives to enable execution on GPUs, and intrinsic functions to improve levels of compiler vectorization;
- Benchmarking the effects on runtime of changes in hardware and environment configuration, in particular changes in memory bandwidth, MPI-rank placement and the varying use of threads/MPI ranks to use available processor resources. Such studies enable us to investigate the opportunity for optimizations outside of traditional changes in source code and reflects our on-going view that optimization of runtime encompasses a wide range of options including software configuration as well as design. We note that such options can have significant impact at scale and demonstrate the effect of such options for runs of over 16,000 processor cores;
- Highlighting of several architectural parameters which serve as bottlenecks or limits to further improvements in performance. A natural effect of early generations of hardware is that a number of optimization opportunities is likely to still be present in the design. In our work we use miniapps to identify these and discuss how, when addressed, these may provide improvement in runtime performance.

Due to the various, non-uniform levels of maturity of the systems used for this work we do not provide direct hardware-to-hardware comparison, instead preferring to use relative measures of improvement for each experiment. As the hardware and software stacks associated with platform develop further we will intend to provide future publications which will enable direct performance comparisons to be made.

## 1.1 Related work

The number and breadth of challenges associated with preparing for multi-petascale and exascale-class computing are significant and have helped to create a rich set of academic investigations touching on comparison of computer architectures [19, 2], optimization for specific classes of hardware [13, 8] and programming languages and mechanisms [25, 21, 16, 1].



# Chapter 2

## Programming Models and Environments

Programming models (abstractions used to reason about program design and implementation) and programming environments (compilers and tools used to implement software, correlated in design to one or more programming models) are some of the most challenging and dynamic elements on the path to exascale computing. Single program multiple data (SPMD) built on top of MPI has by far been the dominant programming model and environment pair since the emergence of distributed memory computing two decades ago. There is overwhelming evidence that single-level SPMD (statically assigning a process per core) is insufficient to achieve optimal performance. Even more importantly, this approach will not scale with the performance potential of future systems. If we are going to continue tracking future performance trends, we need to augment or replace existing strategies.

Revolutionary approaches in this area arguably include Chapel [14], ParalleX [16] and Swarm [1], since these languages or execution models, or both, would require a complete redesign of an existing application. Although future systems may demand such efforts, most application teams are using a more evolutionary approach, sometimes called MPI+X, by combining SPMD (via MPI) with one or more node-level programming environments such as OpenMP [12], Intel Threading Building Blocks (TBB) [23], CUDA [22] or C++11 threads.

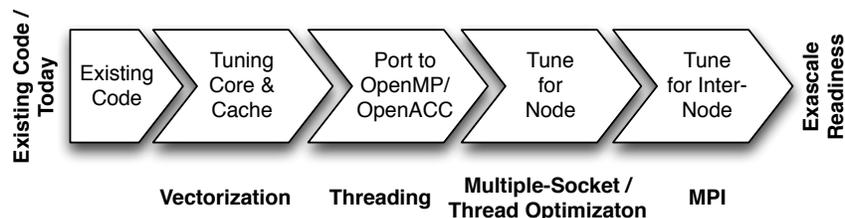
Although MPI+X may appear incremental, it is in fact very challenging to implement. In order to successfully scale on current and future heterogeneous and manycore nodes, the computational kernels of an existing (MPI) application must be redesigned to support dynamic partitioning and scheduling of work by the node-level runtime system. Typically this is most easily done by encapsulating kernels in stateless functions that can be parametrized to dynamically execute some runtime-selected portion of work such that, when scheduled to a processing element, the computation is performed efficiently. All node-level programming models and environments are compatible with this approach and some, such as TBB and CUDA, explicitly require it.

Interestingly, by going through the above refactoring processing, we not only make MPI+X work well, but we position ourselves well for any future programming and execution model. The effort of exposing and encapsulating parallelizable computations in this manner is intrinsically valuable. Furthermore, by optimizing node-level performance for runtime systems such as CUDA, we also move in the direction of hiding latency via task

concurrency. Optimizing occupancy rates in CUDA is a harbinger of the kind of reasoning needed for efficient use of ParalleX and Swarm.

# Chapter 3

## Methodology



**Figure 3.1.** Steps in an Evolutionary Path to Exascale Readiness

In this paper we document our experiences with attempting an evolutionary migration of code from existing petascale machines to exascale-class prototype hardware. The focus is therefore on maintaining as many of the existing investments in code as can be productively retained and, where possible, adapting the structure for new classes of hardware. Towards this end we have empirically found our code modifications follow a largely similar path regardless of the hardware technology being employed. Figure 3.1 represents our migration strategy that starts with our existing code and proceeds through a series of steps to port and optimize this for future systems. In practice some of these steps may occur in parallel or out of the order described. The principle steps of our porting have so far been the following:

1. Vectorization
2. Threading
3. Tuning for the Compute Node and
4. MPI/Inter-Node Parallelism

The stages of this process capture our intent to introduce improved levels of parallelism into our code which will be essential for application performance in the coming decade. Whilst the specifics of how this parallelism is presented to the compiler or the machine are unique to each hardware platform, the locating of parallelism is primarily a property of the algorithm

being used. Our experience is that the knowledge of parallelism obtained through this process can therefore be re-used between different compute architectures and programming models reducing the time and cost associated with porting.

# Chapter 4

## Case Study: Initial Porting of Mantevo Mini-Applications to Future Computing Architectures

In this last section of the paper we describe a series of short case studies which describe the value of mini-applications in assessing the capabilities and characteristics of future computing architectures and programming models. In so doing we are able to survey state of the art prototype computing architectures and provide initial commentary on how applications may be mapped to them using evolutionary modifications to the application. In order to separate the various levels of tuning being conducted, our results are split into three sections: (1) optimization within the processor core; (2) optimization within a whole compute node and, (3), optimization between compute nodes.

In this section we use the term *core*, *processor*, and *node* in a flexible context since precise terminology is still being refined throughout the community. For purposes herein, we view a core as being as a unit which is capable of performing calculation including traditional processor cores or lightweight cores as found in new hardware types such as a GPU; we view a processor as essentially being a socket and a node is a network endpoint.

### 4.1 Processor Core Performance

Future computing hardware is expected to provide increased parallelism in the processor core through the availability of, and increased width of, vector registers which are able to perform a single instruction over multiple pieces of data simultaneously. With the introduction of MMX instructions by Intel in 1995 and subsequent additions in the form of SSE and latterly AVX, many of the available floating-point operations in commodity processors require codes to exploit high levels of data parallelism in order to achieve a high proportion of peak chip performance. A well tuned compiler can very often detect opportunities for vectorization providing the code is structured in a manner that the compiler can safely determine the introduction of vectored instructions will not violate the original statement ordering. Where code control is complex or inter-statement dependencies exist, compilers are often unable to generate vectorized code resulting in slower execution. One potential approach to addressing

**Table 4.1.** Speedup of manual vectorization for miniMD force calculation and full application runtime.

	Force (Speedup (x))	Total (Speedup (x))
<b>AMD A8 3850 APU, 2.90GHz, SSE4a</b>		
Single Precision	1.26	1.21
Double Precision	1.56	1.49
<b>Intel Westmere 5690, 3.47GHz, SSE4.2</b>		
Single Precision	1.57	1.43
Double Precision	1.42	1.33
<b>Intel SandyBridge, 2.60GHz, SSE4.2</b>		
Double Precision	1.85	1.60

this issue is for the programmer to employ manual vectorization through the introduction of *intrinsic* operations – a library of routines which communicate how the developer would like vectorization to occur. Such code constructs allow developers to easily express low-level vector parameters and data operations that either expand directly to assembly instructions or provide programmer intent to the compiler allowing it to manipulate these statements into vectorized operations. Typically not for the faint of heart, we apply this approach herein in order to illustrate the potential performance advantages with a view that maturing compiler technology may provide additional opportunities for vectorization in the future and that the performance identified helps us to assess what the hardware may in fact be capable of.

Our initial inspections into poorer than expected performance of miniMD on commodity processors has shown that the force compute loop (which is responsible for up to 90% of serial runtime) cannot be vectorized due to the complex pointer behavior being employed as well as sparse operand loads from memory. The use of double pointer indirection to map data structures into the compute kernel prevents the compiler from determining whether vector instructions can be utilized without violating ordering constraints despite a valid vectorization being possible from an algorithmic perspective.

The force function computes the interaction forces between each pair of atoms that exist in a specific neighbor list. Due to the sparse nature of the atom information and the condition operations associated with identifying whether the atom pair is within a cut-off zone, vectorization of this code is particularly challenging. However, the high proportion of execution associated with the function makes this a candidate for optimization. In order to address the poor level of vectorization the main force compute loop was instrumented with vector intrinsic operations enabling the compiler to generate code with increased levels of vectorization. Table 4.1 shows the speedup obtained through the use of SSE4 intrinsic operations on several processors. Despite the SandyBridge being capable of executing AVX instructions, we have provided results from our SSE ports to enable a comparison.

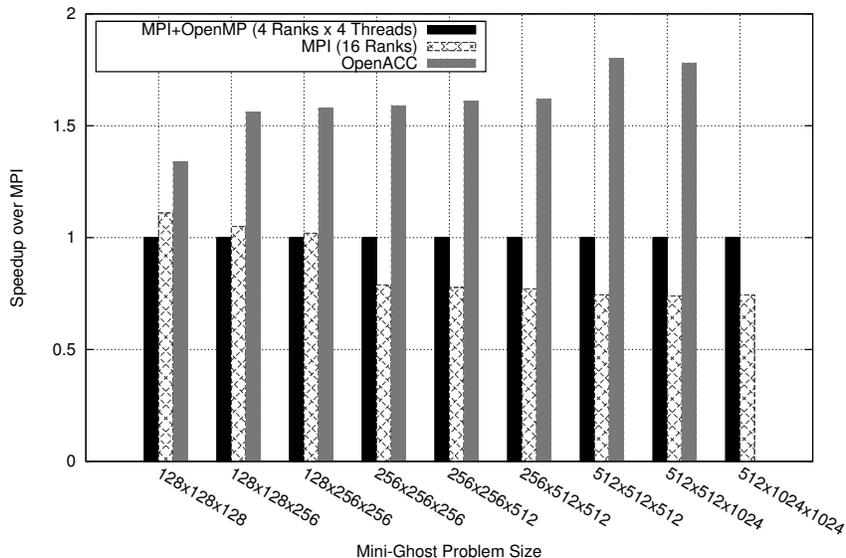
SSE provides a 128-bit wide vector unit (four single-precision operands or two double-precision operands) enabling a maximum speedup of floating point calculations equivalent

to the vector width. Although each operation within the force kernel is vectorized using an intrinsic operation, the speedup obtained is lower than the theoretical maximum as some instructions over vector registers are serialized within the processor core and other architectural bottlenecks such as memory operations are not executed in parallel. The output of the vector intrinsic instrumented codes is not *identical* to non-vectorized source as the operations may lead to a change in rounding effects, but has been thoroughly tested to ensure the results are acceptable to domain scientists. Our experience of rounding has shown that the introduction of intrinsics can yield subtle changes in output and even execution behavior and therefore careful design and post-implementation testing are required.

As we look forward the coming arrival of the Intel MIC architecture with an increased vector width, we predict that the addition of intrinsics to key application kernels may become more commonplace. We further expect the structure of the refactored intrinsic force kernel to be reusable on MIC platforms moving forward with only minor changes to adapt to the wide vector registers. Within the evolutionary context, adaption of key kernels using intrinsics therefore allows us to migrate applications to new platforms and provide significant improvement in execution speed with changes isolated to only short sections of code. Although intrinsics are not available for the GPU in the same sense, the knowledge of algorithm parallelism is fundamental to informing us of future ports to such architectures. Furthermore, many of the improvements being prepared for future exascale-class platforms can be reused on a number of our existing systems.

## 4.2 Intra-node performance

Effectively exploiting the performance potential provided by increasingly complex node architectures is currently seen as one of the major challenges for on-going code development. In this section we provide three examples illustrating several issues that the application developer may need to consider on an increasingly complex compute node. Specifically we focus on : (1) the porting of miniGhost to NVIDIA GPUs using the recently announced OpenACC compiler directive toolkit, demonstrating initial identification of parallelism in the algorithm and the performance obtained from OpenACC; (2) the porting of the miniFE Finite Element assembly phase to NVIDIA GPUs using CUDA which identifies the high level of register spilling present in the code and discusses how this will be addressed in future GPU systems and, finally, (3) a study of miniFE assembly and solve phase sensitivity to changes in memory bandwidth. The issues raised in these small studies demonstrate the value of mini-applications in identifying potential performance bottlenecks or sensitivities. The information being obtained through this work is able to drive analysis of larger applications as well as hardware and can have real impact in informing our hardware selection choices and subsequent optimization activities.



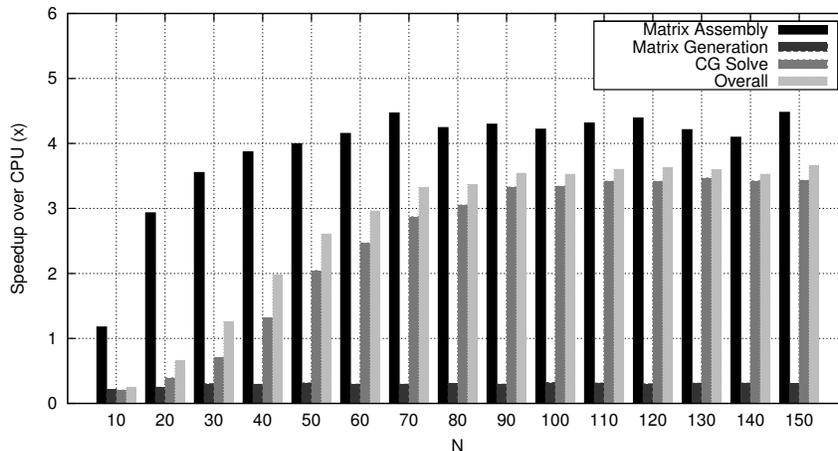
**Figure 4.1.** Performance of varying miniGhost problems on an XK6 node

### 4.2.1 Finite difference stencils

Computation in MiniGhost is based on a triply nested loop, whereby each point in the domain is updated as a function of the average over adjacent points. The simplicity of this computation and the ability to easily configure varying levels of compute complexity let us expose and explore numerous issues expected to significantly impact the performance of full applications that employ difference stencils and the halo exchange.

The hybrid MPI + OpenMP version began with a straightforward wrapping of the outermost loop the `!$OMP PARALLEL DO` directive. It was also then a straightforward port to OpenACC, replacing this directive with the `!$acc parallel loop` directive. However, on a node with a memory hierarchy, such as that on a dual-socket node, the memory affinity is different: with OpenMP, the arrays should be physically distributed across the memory hierarchy of the processing cores (via first-touch) while with a hybrid off-load system, the entire array should be resident on a single processor’s memory so that the movement to the device is straightforward. The MPI-everywhere version automatically maintains processor core and memory affinity.

The stencil loop required no additional use of OpenMP directives. The OpenACC version enabled the means for effectively mapping the data onto the GPU, required to achieve acceptable performance. The Fermi processor is organized into 16 groups of 32 cores, so a directive is used to map the computation as  $x - y$  slices of GRID to 16 blocks (*num\_gangs*), mapping them each as vectors (*vector\_length*) of length 32 onto the 32 core warps.



**Figure 4.2.** Speedup of miniFE CUDA Implementation (NVIDIA Fermi M2090 vs. Hex-Core 2.7GHz E5-2680)

Single node performance on Curie of the MPI-everywhere, MPI+OpenMP, and MPI+OpenACC implementations, illustrated in Figure 4.1, shows that the GPU gives a speedup over the MPI-everywhere ranging from around 25% to 80% as the amount of work increases. However, if the data to be operated on must be moved to and from the GPU, the advantage reverses. The best MPI+OpenMP configuration of 4 MPI ranks per node each with 4 OpenMP threads outperforms the MPI version by about 10%, but this quickly reverses as the problem size increases, with performance decreasing to 80% of the MPI version. It is beyond the scope of this paper to examine this more closely, though it is likely that this is an artifact of the first generation Interlagos node architecture and thus it is reasonable to expect that it will be addressed in the next generation Trinity node.

Direct comparison of performance between the host and device is problematic since memory sizes differ significantly, a situation common to their sorts of architectures. On Curie, the GPU device has significantly less memory than that available to the host (6 GBytes vs. 32 GBytes), and therefore the node can execute the larger problem shown on the graph while the GPU cannot. From an application perspective, the ultimate comparison then is multi-node strong scaling, examined in Section 4.3.1.

## 4.2.2 MiniFE on a GPU

MiniFE consists of three principle phases: generation of the matrix structure, assembly of the finite-element matrix, and the solution of the sparse linear system using the Conjugate Gradient method. Here we focus on key performance limiters in porting the matrix assembly phase of the algorithm to an NVIDIA GPU, using the CUDA programming model.

The assembly phase involves computing the element operators for each element and then summing the operators into a final matrix. We parallelize this phase by having threads operate on separate elements with the computation of the element operator and the summation into the linear system performed within a single kernel. Although the computation of the element operators is embarrassingly parallel, the summing into a linear system requires synchronization to avoid data race conditions. The use of a single kernel is preferred in this instance because it avoids having to store the state for the element operator and then having to later re-read that state during summing into the linear system.

By using one thread per element we were able to leverage the original code for the construction of the element operator subject to additions for compilation to a CUDA kernel and a modification of the code to use the ELL sparse-matrix representation [7] of the original compressed-row (CSR) form. Atomic addition operations are employed in the kernel to prevent race conditions in updating the global matrix.

The computation of the element operator involves a number of floating-point heavy operations including computing the matrix determinant and the Jacobian. The large number of floating point operations suggest that the performance should be FLOP limited but analysis using NVIDIA’s compute profiler has shown that the performance is in fact bandwidth bound due to register spilling.

The cause of this register spilling can be identified as the element operator which requires a large thread state, including 32 bytes for node-IDs, 96 bytes for node coordinates, 512 bytes for the diffusion matrix, 64 bytes for the source vector as well as data to store the Jacobian and matrix determinant. The Fermi GPU architecture supports up to 63 32-byte registers per thread limiting the total register storage to 252 bytes. As a result of this limit, any additional state must be spilled to at least L1 cache and potentially further to L2 or memory. Since the L1 cache can be up to 48kB in size but is shared by 512 threads this can result in as little as 96 bytes of L1 cache storage per thread. In addition, the L2 cache is 768kB shared by 8192 threads, again leaving only 96 bytes of storage per thread. Since L1 and L2 are insufficiently sized to store the required operator state, registers are spilled to global memory causing the computation to become bandwidth bound.

One method to improve the performance of bandwidth bound kernels is to increase the occupancy. However, in this case, the kernel’s occupancy is limited by register usage. Since the register usage is higher than is available in hardware it is not possible to increase this occupancy without further increasing register spilling.

We tuned the kernel to reduce register usage, including algorithmic changes that exploiting symmetry in the diffusion operator and reordering computations so that data is loaded immediately prior to being used. We have also applied several traditional optimization techniques including pointer restriction, inlining of functions, and unrolling of loops. Finally, we also position a portion of the state in shared memory and experimented with L1 cache sizes. The best performance is achieved by placing the source vector into shared memory and enabling a larger L1 cache. Whilst these optimization greatly reduce register spilling, 512 bytes of state is still spilled per thread. To ensure fair comparison, all optimizations

that were applicable to the original CPU code were back ported also improving the CPU performance.

The performance of the CUDA version of miniFE was compared to the MPI-parallel version of miniFE running on a Tesla M2090 and a hex-core Intel Xeon 2.7GHz E5-2680. We tested for various problem sizes of  $N^3$  hexahedral elements. The speedup for each of the three phases of the algorithm is reported in Figure 4.2.

The assembly realizes a 4x speedup and the solve phase is 3x faster. The generation of the matrix structure exhibits a slowdown because it is computed on the host in CSR format, transferred to the device, and then converted to ELL format. Whilst possible to move this computation to the device, the low proportion of time consumed by these operations does not dominate performance.

Future generations of NVIDIA systems are expected to address some of the findings from this study, including an increased number of registers per thread and increases in the size of L1 and L2 memories. Improvements in the CUDA compiler may also lead to a reduction in the number of register spills or the impact that register spills will have on execution time.

### 4.2.3 The Impact of Memory Speeds

It is widely accepted that the performance of the memory sub-systems used in future exascale computers will improve substantially. This is driven by a realization that application performance, even on contemporary systems, is frequently limited by the “memory wall” [20]. If compute devices experience rapid improvements in calculation throughput, memory will *need* to be improved significantly if applications are not to become entirely throttled on memory access. However, the question of whether the improvement in memory performance will exceed or even match that of compute hardware is still unanswered. Therefore, there is a very real risk that applications will need to execute using lower memory bandwidth in the future.

In this section we describe a study in which we alter the clock rate of memory components in a system through BIOS control, effectively slowing the rate at which memory can process requests. This enables an analysis of code performance where there is a growing divergence between compute throughput and that of memory (which may be the effect of significant improvement in compute hardware that is unmatched by equivalent improvement in memory subsystems). Table 4.2 presents the relative effect on sections of miniFE and Charon (the parent application to miniFE) runtime as the clock rate of memory is lowered from 1333MHz to 1066MHz and 800MHz. The processor used for this study is a dual-socket oct-core AMD Interlagos running at 2.9GHz. The assembly time in miniFE and the Jacobian generation of Charon is unaffected by the change indicating that these sections of code are not predominantly memory bound, whilst the runtime of the conjugate-gradient (CG) solver increases by approximately 16% in the 800MHz case. Of interest is that miniFE is able to closely track equivalent changes in its parent code for the solve phase (which in both codes

**Table 4.2.** Relative Runtime Slowdown of miniFE and Charon from Reduced Memory Frequency (Relative to Memory Frequency of 1333MHz, Lower is Worse)

	800MHz	1066MHz	1333MHz
<b>miniFE Mini-Application</b>			
Finite Element Setup	0.996	1.000	1.000
CG Solve	0.841	0.957	1.000
<b>Charon Device Simulation Application</b>			
Prec/Newt	0.960	0.980	1.000
Jac/Newt	1.000	1.000	1.000
Adv/Newt	0.920	0.970	1.000
Solve/Newt	0.840	0.940	1.000

is the dominant contributor to runtime) giving us confidence in the relevance of our studies using mini-applications.

From this short study we can begin to assess the likely impact that a reduced per-core memory bandwidth may induce. A runtime increase of 16% is approximately half of the drop in memory bandwidth, indicating that miniFE and Charon are clearly highly sensitive to memory bandwidth in their solve phases but that some of the bandwidth loss can be covered either through efficient use of the memory hierarchy or latency hiding by the processor through the use of prefetching and deep instruction pipelines.

### 4.3 Inter-node performance

Our goal of an evolutionary path includes the assumption that inter-node parallelism will continue, in the foreseeable future, to be implemented using functionality provided by MPI. (This does not rule out the use of MPI in a revolutionary approach.)

In this section we explore some issues associated with the ubiquitous nearest neighbor communication pattern. Our work is informed by CTH, an explicit three dimensional multi-material shock hydrodynamics code [18]. CTH models high-speed hydrodynamic flow and the dynamic deformation of solid materials, and solves the equations of mass, momentum, and energy in an Eulerian finite volume formulation. MiniGhost, shown to effectively represent the inter-process communication requirements of CTH, provides a tractable means for exploring strategies for improving the performance characteristics of the full application.

We begin by examining miniGhost on Curie. Next, we address a performance issue associated with process-to-processor mapping, noticeable only at large scales. Then we investigate an alternative to the very large message strategy implemented in CTH and many other applications.

### 4.3.1 On the XK6

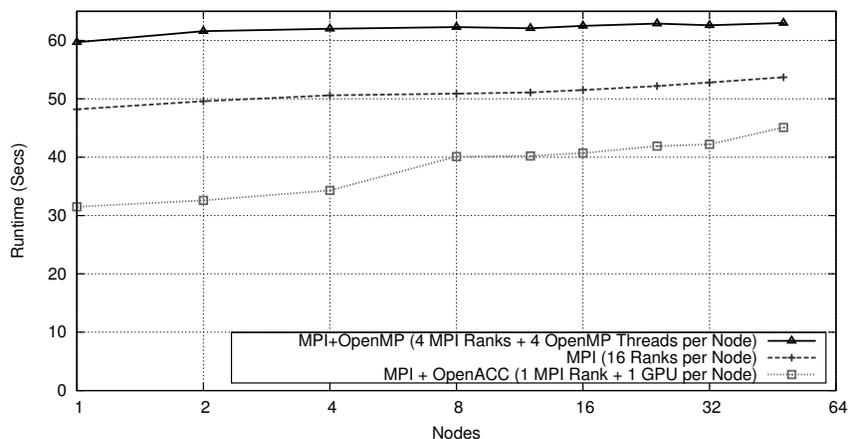
As seen above, Curie’s GPU provides a significant performance capability for the computation of difference stencils, but the cost of moving data between the host and device on a node overwhelms the performance of the computation. To minimize this expense, all arrays are maintained on the device, and only the halo is transferred to and from the host for MPI handling. The MPI+OpenACC implementation uses one MPI rank per node, a (current) limitation of the OpenACC implementation used here.

Problem sets were configured to mimic the profiles of CTH. Weak scaling experiments demonstrate the power of the GPU in comparison with all of the processor cores for the MPI and MPI+OpenMP implementations. Strong scaling experiments were configured to demonstrate the manner in which a domain scientist would use Curie’s capabilities, highlighting the effects of maintaining the state on the GPU. Representative results are shown in Figure 4.3.

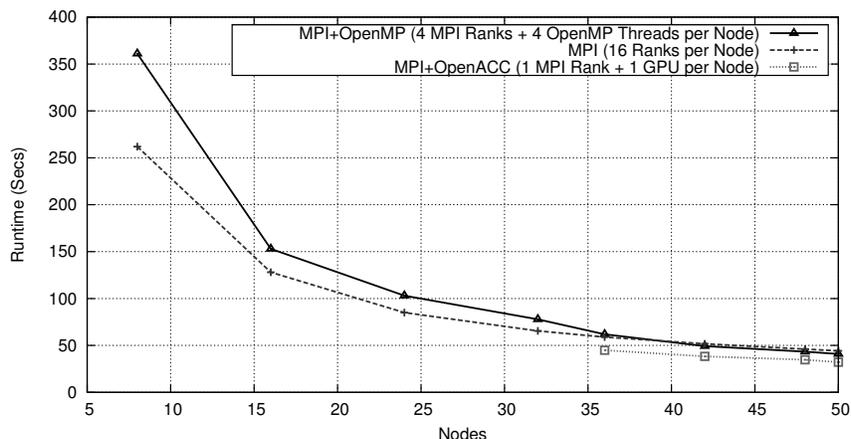
The weak scaling problem involves 16 variables on a  $256 \times 256 \times 512$  grid per node. This ensures that the MPI and OpenMP processes on a host node have a reasonable amount of work ( $128 \times 128 \times 128$  per rank or thread, resp.) while still allowing that work to fit onto the GPU device. Although the communication cost dominates the MPI+OpenACC runtime, the speed of the computation allows it to maintain its advantage over MPI and MPI+OpenMP. However, the gap closes at higher node counts.

The strong scaling problem involves 20 variables operating on a  $1024 \times 1024 \times 1024$  grid. Most notable is that due to the GPUs’ memory constraint relative to the host node, the OpenACC implementation requires a minimum of 32 nodes while the MPI implementations can run on eight nodes. However, at that scale, the MPI+OpenACC implementation out-performs the MPI implementation by about 40%. The MPI+OpenMP implementation becomes competitive with the MPI implementation at higher node counts, a trend we see for the strong scaling results as well as in the largest core counts on Cielo, discussed in the following sections.

The MPI and MPI+OpenMP implementations spent 10-20% and 5-10% of runtime, respectively, in communication, highlighting the computational issue for OpenMP. As seen in the following sections MPI+OpenMP outperforms MPI-everywhere on Cielo at very high scales. Interprocess communication is the sum of the work required to move data between the parallel processes, which includes the time spent packing and unpacking the message buffers as well as the time spent in MPI. For the OpenACC implementation, then, this includes the time spent moving the message buffers and partial sums between the host and device, resulting in mid-90% proportion of runtime spent in communication. We optionally aggregated the variable boundaries into a single array for host-device movement, but this increased the cost since the individual transfers could be partially hidden by the packing of the other buffers. This demonstrates that the data movement is impacted by the inject rate more than by either latency or bandwidth: the PCIx host-device connection is able to inject the transfer and quickly return to packing the next buffer.



(a) Weak scaling: 256x256x256 grid per node, 16 variables



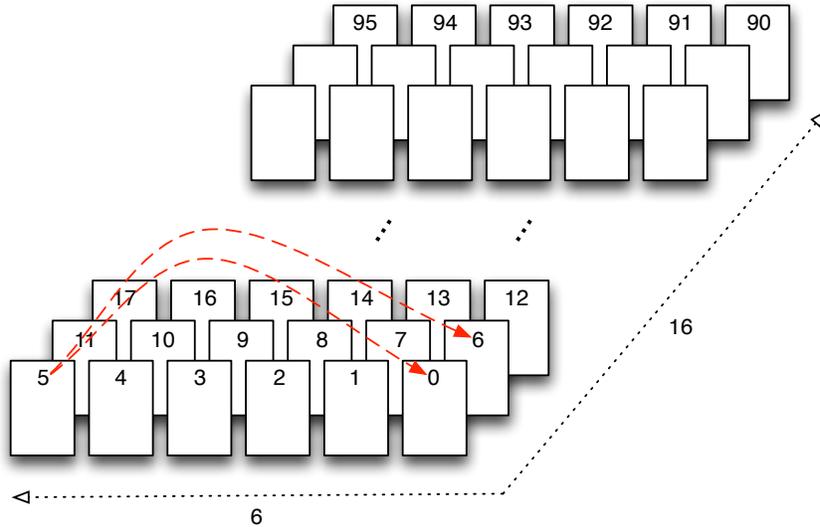
(b) Strong scaling, 1024x1024x1024 grid, 20 variables

**Figure 4.3.** Strong and Weak Scaling of MiniGhost on XK6

Its important to note that OpenACC is a new specification, and supporting compilers have only recently appeared. Our hope is that compilers will improve over time since we found this programming methodology to be rather easy to use.

### 4.3.2 Mapping processes to processors

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 an boundary exchange across those variables can be consolidated in to a single message per neighbor. But in a recently completed broad-based study of Cielo capabilities [3], the nearest neighbor boundary exchange encountered



**Figure 4.4.** Cielo process map

significant scaling degradation beyond 8,000 processor cores.

The problem was traced to the mapping of the parallel processes to the three dimensional torus topology, illustrated in Figure 4.4. 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. 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-mapped 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. Blocks of the mesh are assigned to cores by traversing the blocks of the mesh in the  $x$  direction of the mesh starting at one corner of the mesh. Once those blocks are assigned, the next block assigned is the block one over in the  $y$  direction of the mesh from the first block assigned. The mesh is again then traversed in the  $x$  direction and blocks are assigned to cores. This process is continued until there are no more blocks in the  $y$  direction. The next block assigned is then the first block in the  $z$  direction from the first block assigned. The blocks of the mesh with this  $z$  value are then assigned as the first blocks were assigned. This process is then repeated until all blocks in the mesh have been assigned to cores in the machine.

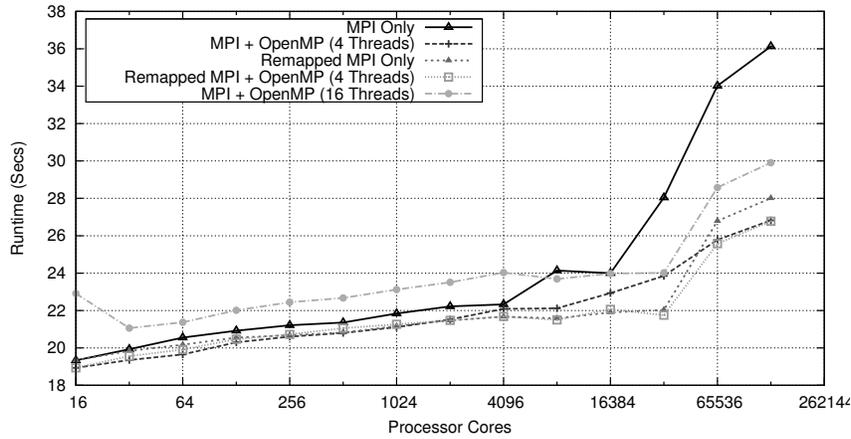
Number of MPI ranks	Regular Order			Reordered		
	X	Y	Z	X	Y	Z
16	0.0	0.0	0.0	0.0	0.0	0.0
32	0.0	0.0	0.0	0.0	0.0	0.0
64	0.0	0.0	0.3	0.0	0.3	0.0
128	0.0	0.0	1.0	0.0	0.5	0.0
256	0.0	0.0	1.0	0.0	0.5	0.3
512	0.0	0.1	2.0	0.0	0.6	0.4
1024	0.0	0.3	2.1	0.2	1.0	0.7
2048	0.0	0.3	2.7	0.3	1.2	1.2
4096	0.0	0.3	3.7	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 4.3.** miniGhost average hop counts on Cielo

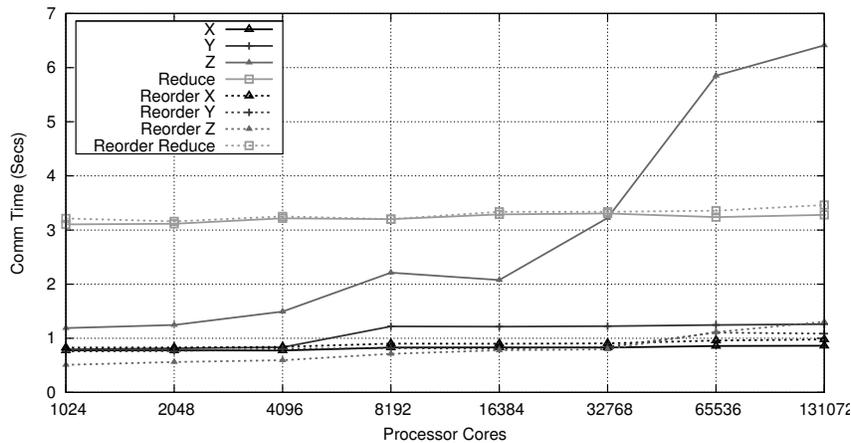
Our remapping algorithm assigns blocks of the mesh to the cores of the machine by groups. On Cielo, a group of blocks consists of a  $2 \times 2 \times 4$  group of blocks. These blocks are then assigned to nodes as above. The result is a slight increase in the average hop counts in the  $x$  and  $y$  directions, but a significant decrease in the average hop count in the  $z$  direction. A comparison of the number of hops between the two approaches is shown in Table 4.3.

This remapping strategy results in a significant improvement in scaling performance, illustrated in Figure 4.5(a). Figure 4.5(b) shows that this is attributable to controlling the time spent sending data in the  $z$  direction. We include the time spent in the reduction sum across each grid variable (inserted after computation on each variable to add application realism as well as a synchronization point), illustrating that this functionality is not the source of the issue, scaling well regardless of the processor mapping. We do see indications of the issue at the highest processor counts, though it is less pronounced. This remapping was incorporated into CTH, the results of which are described in Appendix F.

As discussed in the related work section above, we are exploring ways for incorporating these ideas into a more general interface. We are also exploring the use of `MPI_Datatype` in handling the non-contiguous (but patterned) face data.



(a) Time to solution



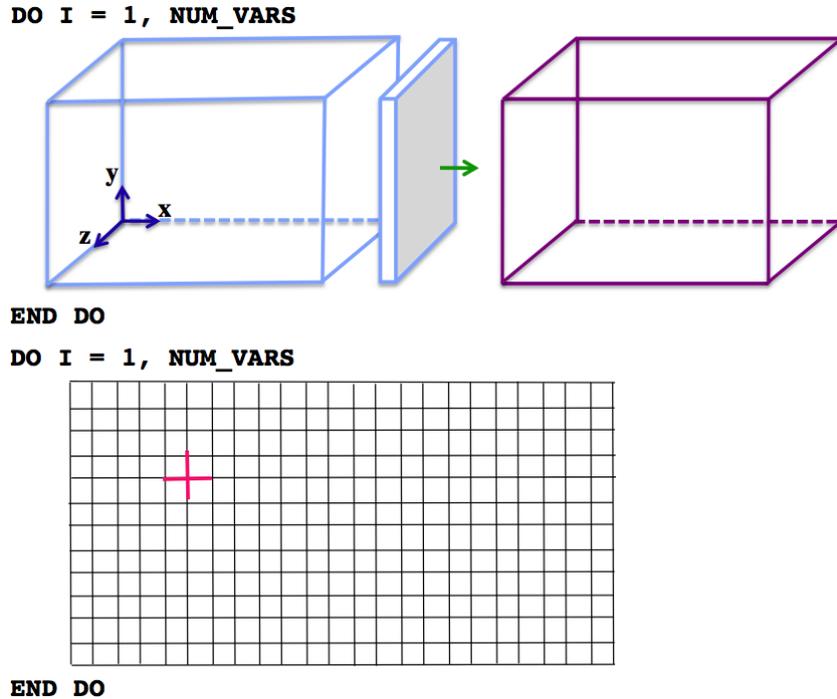
(b) Communication time per direction (MPI version)

**Figure 4.5.** Performance of MiniGhost with MPI-rank remapping on Cielo

### 4.3.3 Alternative communication strategies

Node interconnects are also evolving, driven by new node architectures as well as cost and energy conservation goals, encouraging exploration of new approaches within the context of application requirements. Interconnects are designed as a balance of global bandwidth (the ability of the interconnect to move data), inject bandwidth (the ability of the NIC to put data onto the interconnect), and injection rate (the ability of a node to place messages onto the NIC). Global bandwidth typically incurs the highest costs, both in terms of money and power consumption, and therefore we are preparing for a proportional decrease in that capability.

MiniGhost includes an application-relevant infrastructure for exploring alternative bound-



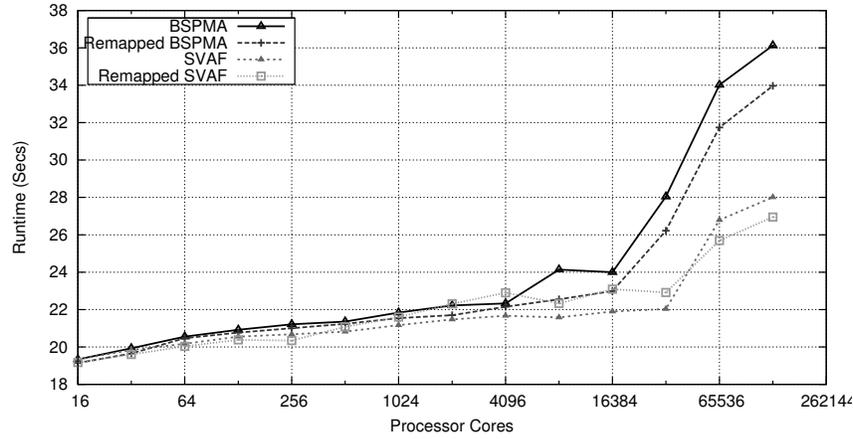
**Figure 4.6.** CTH boundary exchange and computation

ary exchange configurations [6]. The first configuration mimics that of CTH, which we call Bulk Synchronous Parallel with Message Aggregation (BSPMA), was used above, illustrated in Figure 4.6. The second, called Single Variable Aggregated Faces (SVAF) transmits data as soon as computation on a variable is completed, and thus six messages are transmitted for each variable (up to 40), one to each neighbor, each time step. (Looking at Figure 4.6, this eliminates the inner `END DO` and `DO I = 1, NUM_VARS`.) The two  $x - y$  faces are contiguous in memory, so each may be directly sent using a call to a single MPI function. The other four faces are aggregated into buffers, resulting in four messages to their neighbors. A third mode, called single variable, contiguous pieces, computational overlapping mode (SVCP), is designed for use on architectures that are strongly biased toward significantly increased message injection rates and injection bandwidth, a trend we see developing but not yet to the extent of supporting this configuration using MPI [25].

BSPMA and SVAF have been configured for MPI-everywhere as well as MPI+OpenMP. For the latter on Cielo and Curie, its best configuration is four MPI ranks on each node, each spawning four OpenMP threads. Note that this increases the size of each message in comparison with the MPI-everywhere version. Because of ghost cells, the message size in two directions almost doubles and the message size in the third direction almost quadruples. This is because the number of cells in two of the three directions is doubled. The size of the message is based the size of the face with ghost cells, so if the size of a face is  $x \times y$  cells for the MPI everywhere case, the size of the message is  $(x + 2) * (y + 2)$ . If the number of cells

in one of these directions (say  $y$ ) is doubled, then the size of the message is  $(x + 2) * (2y + 2)$ , which is not quite double the original size, but close. Similarly, if the number of cells in both directions are doubled, then the size of the message is almost quadrupled.

Performance of these implementations on Cielo are shown in Figure 4.7. Effective map-



**Figure 4.7.** Performance of miniGhost Communication Strategies on Cielo

ping of processes to processors is again critical to achieving good scaling, and as the number of processors increases, SVAF becomes the best strategy. This is of significant interest since it reduces demand on costly global bandwidth by a factor of  $N$ , where  $N$  is the number of variables aggregated (40 for the shaped charge problem.)



# Chapter 5

## Conclusions and Future Work

With the goal of enabling an effective piecewise evolutionary path to making effective use of Exascale computing architectures, we described our explorations of a breadth of issues throughout the codesign space. While faced with uncertain choices of programming models, mechanisms, and perhaps languages designed to run in an uncertain computing environment, we have demonstrated a variety of ways the application developer can begin to concretely and effectively prepare for an unknown specific machine but a widely accepted architectural approach. Although this work is presented in terms of processor, node, and inter-node strategies, we also see that a system-wide design is required to achieve overall reductions in runtime. The common theme is, not unexpectedly, the organization of our data and the way it is moved around and presented to the various components of the architecture. Most reassuring in terms of modifications to large code bases, we have demonstrated an evolutionary path that can also significantly improve performance on current and emerging architectures. Additional information in support of this paper is presented in [5], and deeper studies on some of the topics herein will be presented in papers in preparation.

Our use of miniapps significantly improves our ability to rapidly explore ideas, and these miniapps have been demonstrated to be predictive of full application codes with regard to some key performance issues [4], guiding our focus here. For example, the remapping strategy has proven beneficial to CTH. That said, we reiterate that the output of a miniapp is information that must be interpreted within the context of the full application, and therefore the application developer must apply and probably extend the experiences described in this paper.

We are also studying revolutionary options, including less commonly used and new languages (e.g. [21, 26, 10, 11]). It is also possible that a completely new architecture could emerge from the exascale initiatives. Regardless, it appears that the fundamental concepts for exploiting these architectures will remain: presenting data to the compute engine in a manner that allows it to operate on the data in a vectorized multi-threaded fashion, sharing that data with the parallel processes in efficient ways and exposing sufficient parallelism to effectively hide ever-increasing relative latencies. The lesson learned from our incremental evolutionary approach will not only help applications in the near to medium term, but also set the stage for a smoother transition to revolutionary environments.



# References

- [1] D.A. Bader, V. Kanade, and K. Madduri. SWARM: A Parallel Programming Framework for Multi-Core Processors. In *First Workshop on Multithreaded Architectures and Applications (MTAPP)*, March 2007.
- [2] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho. Entering the Petaflop Era: the Architecture and Performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08*, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press.
- [3] B.W. Barrett 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.
- [4] R.F. Barrett, P.S. Crozier, S.D. Hammond, M.A. Heroux, P.T. Lin, T.G. Trucano, and C. Vaughan. Assessing the Validity of the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications. Technical Report SAND2012-TBD, Sandia National Laboratories, 2012. In preparation.
- [5] R.F. Barrett, S.D. Hammond, C.T. Vaughan, D.W. Doerfler, M.A. Heroux, J.P. Luitjens, and D. Roweth. Navigating An Evolutionary Fast Path to Exascale. Technical Report SAND 2012-TBD, Sandia National Laboratories, 2012. [http://www.sandia.gov/~rffbarre/pubs\\_list.html](http://www.sandia.gov/~rffbarre/pubs_list.html).
- [6] R.F. Barrett, C.T. Vaughan, and M.A. Heroux. MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies Using Stencil Computations in Scientific Parallel Computing. Technical Report SAND2011-5294832, Sandia National Laboratories, May 2011. <https://software.sandia.gov/mantevo/publications.html>.
- [7] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, 2009.
- [8] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.
- [9] L. Brown, P.M. Bennett, M. Cowan, C. Leach, and T.C. Oppe. Finding the Best HPCMP Architectures Using Benchmark Application Results for TI-09. *HPCMP Users Group Conference*, 0:416–421, 2009.
- [10] B.L. Chamberlain, D.Callahan, and H.P. Zima. Parallel programming and the Chapel language. *International Journal on High Performance Computer Applications*, 21(3):291–312, 2007.

- [11] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA)*, October 2005.
- [12] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.
- [13] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2008 (SC'08)*, pages 1–12, November 2008.
- [14] R.E. Diaconescu and H.P. Zima. An Approach to Data Distribution in Chapel. *International Journal on High Performance Computer Applications*, 21(3), 2007.
- [15] Roland W. Freund. A Transpose-Free Quasi-Minimum Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14(2):470–482, 1993.
- [16] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. ParalleX: A Study of A New Parallel Computation Model. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, March 2007.
- [17] Roger G. Grimes, David R. Kincaid, William I. MacGregor, and David M. Young. Itpack report: Adaptive iterative algorithms using symmetric sparse storage. Technical Report CNA-139, Center for Numerical Analysis, University of Texas, 1978.
- [18] E.S. Hertel and others. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings, 19th International Symposium on Shock Waves*, 1993.
- [19] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupati, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.
- [20] S.A. McKee. Reflections on the Memory Wall. In *First Conference on Computing Frontiers*, 2004.
- [21] R.W. Numrich and J.K. Reid. Co-Array Fortran for Parallel Programming. *ACM Fortran Forum*, 17(2):1–31, 1998.
- [22] NVIDIA Corporation. CUDA programming guide. [http://www.nvidia.com/object/cuda\\_home.html](http://www.nvidia.com/object/cuda_home.html).
- [23] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

- [24] 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.
- [25] H. Shan et al. A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI. In *Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, PMBS '11, pages 13–14, New York, NY, USA, 2011. ACM.
- [26] UPC. Consortium, UPC Language Specification. May 31 2005.



# Appendix A

## Programming Environment

In this section we list the programming environments and other details used for the studies reported in this paper. The general approach was to use default settings and basic compiler and runtime options. Future work will include deeper dives into the various alternative configurations.

### A.1 Cielo

Work performed within the context of `PrgEnv-cray/4.0.36`. Compiled using Cray Fortran compiler version 8.0.1 with the `-O3` optimization flag. Application launch using `aprun`, with settings ensuring appropriate placement of processes.

### A.2 Curie

Work performed within the context of `PrgEnv-cray/4.0.36`. Compiled using Cray Fortran compiler version 8.0.3 with the `-O3` optimization flag. OpenACC version 1.0 as implemented as provided by module `craype-accel-nvidia20`, Cray CUDA version 4.0.17a.

The Cray OpenACC compiler provides a significant amount of profiling. Setting environment variable `CRAY_ACC_DEBUG` to an integer value from 0 to 3 shows the movement of data to and from the device. Other profiling data reported by the CUDA profile log version 2.0.

An OpenACC enabled PGI compiler became available during our work. A comparison between the two is not in the scope of this paper, but such a comparison will be made in future work.

## **A.3 GPU**

The work in Section 4.2.2 was performed using cud a 4.0 on a Tesla M2090 and a six-core Intel Xeon E5-2680 at 2.7 GHz. Compile flags: `-O3 arch=sm_20`. Thread block size was 64.

## **A.4 Teller**

Results gathered on the Teller testbed used the `openmpi-gnu/1.5` TOSS/CHAOS MPI environment module and GNU system installed compilers.

## **A.5 Dual-Socket, Oct-core AMD 2.4GHz Magny-Cours 6136**

This platform was used in the memory speed study (Section 4.2.3 and Appendix E). Intel compilers 11.0.081, with compiler flag `-O3`. MPI implementation: OpenMPI 1.4.3,

## **A.6 Dual-Socket, Quad-core Intel 2.93GHz Nehalem 5570**

This platform was used in the memory speed study (Section 4.2.3 and Appendix E). Intel compilers 11.0.081, with compiler flag `-O3`. MPI implementation: OpenMPI 1.4.3.

# Appendix B

## MiniMD code examples

The miniMD force calculation with modifications targeting vectorization are shown in this section. The main focus of miniMD is on the force calculation, with the code segment shown in Figure B.1. Changing to single pointers allowed the compiler to identify opportunities for vectorization, illustrated in Figure B.2. Vector intrinsic functions illustrated in Figure B.3.

**Code B.1.** miniMD force calculation vectorization

```
double **x,**f;

for (i = 0; i < nlocal; i++) {
    neighs = neighbor.firstneigh[i];
    numneigh = neighbor.numneigh[i];
    xtmp = x[i][0];
    ytmp = x[i][1];
    ztmp = x[i][2];

    for (k = 0; k < numneigh; k++) {
        j = neighs[k];
        delx = xtmp - x[j][0];
        dely = ytmp - x[j][1];
        delz = ztmp - x[j][2];
        rsq = delx*delx + dely*dely + delz*delz;

        if (rsq < cutforcesq) {
            sr2 = 1.0/rsq;
            sr6 = sr2*sr2*sr2;
            force = sr6*(sr6 - 0.5)*sr2;
            f[i][0] += delx*force;
            f[i][1] += dely*force;
            f[i][2] += delz*force;
            f[j][0] -= delx*force;
            f[j][1] -= dely*force;
            f[j][2] -= delz*force;
        }
    }
}
```

**Code B.2.** miniMD force calculation vectorization

```
double** restrict x;
double** restrict f;

for (i = 0; i < nlocal; i++) {
    neighs = neighbor.firstneigh[i];
    numneigh = neighbor.numneigh[i];
```

```

    xtmp = xx[i];
    ytmp = xy[i];
    ztmp = xz[i];
    res1 = fx[i];
    res2 = fy[i];
    res3 = fz[i];

#pragma simd reduction(+:res1,res2,res3)
    for (k = 0; k < numneigh; k++) {
        j = neighs[k];
        delx = xtmp - xx[j];
        dely = ytmp - xy[j];
        delz = ztmp - xz[j];

        rsq = delx*delx + dely*dely + delz*delz;

        if (rsq < cutforcesq) {
            sr2 = 1.0f/rsq;
            sr6 = sr2*sr2*sr2;
            force = sr6*(sr6-0.5f)*sr2;
            res1 += delx*force;
            res2 += dely*force;
            res3 += delz*force;

            fx[j] -= delx*force;
            fy[j] -= dely*force;
            fz[j] -= delz*force;
        }
    }

    fx[i] += res1;
    fy[i] += res2;
    fz[i] += res3;
}

for (i = 0; i < nall; i++) {
    f[i][0] = fx[i];
    f[i][1] = fy[i];
    f[i][2] = fz[i];

    x[i][0] = xx[i];
    x[i][1] = xy[i];
    x[i][2] = xz[i];
}

```

**Code B.3.** miniMD force with SSE intrinsics

```

const double one = 1.0;
const double half = 0.5;

const __m128d one_vec = _mm_load1_pd(&one);
const __m128d half_vec = _mm_load1_pd(&half);
const __m128d cutforce_vec = _mm_load1_pd(&cutforcesq);

double fjx_store[2] __attribute__((aligned(16)));
double fjy_store[2] __attribute__((aligned(16)));
double fjz_store[2] __attribute__((aligned(16)));

__m128d fi0_vec;
__m128d fi1_vec;
__m128d fi2_vec;

for (i = 0; i < nlocal; i++) {
    neighs = neighbor.firstneigh[i];
    numneigh = neighbor.numneigh[i];
}

```

```

xtmp = x[i][0];
ytmp = x[i][1];
ztmp = x[i][2];

k = 0;
const int numneigh_2 = numneigh - 2;

__m128d xtmp_vec = _mm_load1_pd(&xtmp);
__m128d ytmp_vec = _mm_load1_pd(&ytmp);
__m128d ztmp_vec = _mm_load1_pd(&ztmp);
__m128d fi0_vec = _mm_setzero_pd();
__m128d fi1_vec = _mm_setzero_pd();
__m128d fi2_vec = _mm_setzero_pd();

for (; k < numneigh_2; k = k + 2) {
    const int j_0 = neighs[k];
    const int j_1 = neighs[k+1];

    __m128d delx_vec = _mm_sub_pd(xtmp_vec, _mm_set_pd(x[j_1][0], x[j_0][0]));
    __m128d dely_vec = _mm_sub_pd(ytmp_vec, _mm_set_pd(x[j_1][1], x[j_0][1]));
    __m128d delz_vec = _mm_sub_pd(ztmp_vec, _mm_set_pd(x[j_1][2], x[j_0][2]));

    const __m128d rsq_vec = _mm_add_pd(
        _mm_add_pd(
            _mm_mul_pd(delx_vec, delx_vec),
            _mm_mul_pd(dely_vec, dely_vec)),
        _mm_mul_pd(delz_vec, delz_vec));

    const __m128d sr2_vec = _mm_div_pd(one_vec, rsq_vec);
    const __m128d sr6_vec = _mm_mul_pd(_mm_mul_pd(sr2_vec, sr2_vec), sr2_vec);
    const __m128d cond_vec = _mm_cmplt_pd(rsq_vec, cutforce_vec);

    const __m128d force_vec = _mm_and_pd(cond_vec, _mm_mul_pd(sr6_vec,
        _mm_mul_pd(_mm_sub_pd(sr6_vec, half_vec), sr2_vec)));
    delx_vec = _mm_mul_pd(delx_vec, force_vec);
    dely_vec = _mm_mul_pd(dely_vec, force_vec);
    delz_vec = _mm_mul_pd(delz_vec, force_vec);

    fi0_vec = _mm_add_pd(fi0_vec, delx_vec);
    fi1_vec = _mm_add_pd(fi1_vec, dely_vec);
    fi2_vec = _mm_add_pd(fi2_vec, delz_vec);

    delx_vec = _mm_sub_pd(_mm_set_pd(f[j_1][0], f[j_0][0]), delx_vec);
    _mm_store_pd(fjx_store, delx_vec);
    f[j_0][0] = fjx_store[0];
    f[j_1][0] = fjx_store[1];

    dely_vec = _mm_sub_pd(_mm_set_pd(f[j_1][1], f[j_0][1]), dely_vec);
    _mm_store_pd(fjy_store, dely_vec);
    f[j_0][1] = fjy_store[0];
    f[j_1][1] = fjy_store[1];

    delz_vec = _mm_sub_pd(_mm_set_pd(f[j_1][2], f[j_0][2]), delz_vec);
    _mm_store_pd(fjz_store, delz_vec);

    f[j_0][2] = fjz_store[0];
    f[j_1][2] = fjz_store[1];
}

_mm_store_pd(fjx_store, fi0_vec);
_mm_store_pd(fjy_store, fi1_vec);
_mm_store_pd(fjz_store, fi2_vec);

f[i][0] += fjx_store[0] + fjx_store[1];
f[i][1] += fjy_store[0] + fjy_store[1];
f[i][2] += fjz_store[0] + fjz_store[1];

// handle the less than 2 case

```

```

for (; k < numneigh; k++) {
    j = neighs[k];

    delx = xtmp - x[j][0];
    dely = ytmp - x[j][1];
    delz = ztmp - x[j][2];

    rsq = delx*delx + dely*dely + delz*delz;

    if (rsq < cutforcesq) {
        sr2 = 1.0/rsq;
        sr6 = sr2*sr2*sr2;
        force = sr6*(sr6-0.5)*sr2;

        f[i][0] += delx*force;
        f[i][1] += dely*force;
        f[i][2] += delz*force;
        f[j][0] -= delx*force;
        f[j][1] -= dely*force;
        f[j][2] -= delz*force;
    }
}

```

# Appendix C

## MiniGhost code examples

The miniGhost 3D 27-point stencil calculation with modifications targeting vectorization are shown in this section. The main focus of of miniGhost is on the 3D 27-point stencil calculation, with the code segment shown in Figure C.1.

**Code C.1.** miniGhost 3D 27-point stencil

```
1 DO K = 1, NZ
2   DO J = 1, NY
3     DO I = 1, NX
4       G2(I,J,K) = ( &
5         G1(I-1,J-1,K-1) + G1(I-1,J,K-1) + G1(I-1,J+1,K-1) + &
6         G1(I ,J-1,K-1) + G1(I ,J,K-1) + G1(I ,J+1,K-1) + &
7         G1(I+1,J-1,K-1) + G1(I+1,J,K-1) + G1(I+1,J+1,K-1) + &
8
9         G1(I-1,J-1,K)   + G1(I-1,J,K)   + G1(I-1,J+1,K) + &
10        G1(I ,J-1,K)   + G1(I ,J,K)   + G1(I ,J+1,K) + &
11        G1(I+1,J-1,K)   + G1(I+1,J,K)   + G1(I+1,J+1,K) + &
12
13        G1(I-1,J-1,K+1) + G1(I-1,J,K+1) + G1(I-1,J+1,K+1) + &
14        G1(I ,J-1,K+1) + G1(I ,J,K+1) + G1(I ,J+1,K+1) + &
15        G1(I+1,J-1,K+1) + G1(I+1,J,K+1) + G1(I+1,J+1,K+1) &
16
17      ) * TWENTYSEVENTH
18
19   END DO ! End NX
20 END DO  ! End NY
21 END DO  ! End NZ
```

## Code C.2. Block version: miniGhost 3D 27-point stencil

```

1  YBLOCKS = 5
2  ZBLOCKS = 5

4  !OMP PARALLEL DO COLLAPSE(2)
5  DO ZZ = 1, NZ , ZBLOCKS
6      DO YY = 1, NY , YBLOCKS

8          ZMAX = ZZ + ZBLOCKS
9          IF ( ZMAX .GT. NZ ) ZMAX = NZ

11         DO K = ZZ, ZMAX
12             YMAX = YY + YBLOCKS
13             IF ( YMAX .GT. NY ) YMAX = NY

15             DO J = YY, YMAX

17         !DIR VECTOR NONTEMPORAL
18             DO I = 1, NX

20                 G2(i,j,k) = ( &
21                     G1(I-1,J-1,K-1) + G1(I-1,J,K-1) + G1(I-1,J+1,K-1) + &
22                     G1(I ,J-1,K-1) + G1(I ,J,K-1) + G1(I ,J+1,K-1) + &
23                     G1(I+1,J-1,K-1) + G1(I+1,J,K-1) + G1(I+1,J+1,K-1) + &

25                     G1(I-1,J-1,K) + G1(I-1,J,K) + G1(I-1,J+1,K) + &
26                     G1(I ,J-1,K) + G1(I ,J,K) + G1(I ,J+1,K) + &
27                     G1(I+1,J-1,K) + G1(I+1,J,K) + G1(I+1,J+1,K) + &

29                     G1(I-1,J-1,K+1) + G1(I-1,J,K+1) + G1(I-1,J+1,K+1) + &
30                     G1(I ,J-1,K+1) + G1(I ,J,K+1) + G1(I ,J+1,K+1) + &
31                     G1(I+1,J-1,K+1) + G1(I+1,J,K+1) + G1(I+1,J+1,K+1) &

33                 ) * TWENTYSEVENTH

35             END DO ! End NX
36         END DO ! End NY
37     END DO ! End NZ
38 END DO ! YBLOCKS
39 END DO ! ZBLOCKS

```

**Code C.3.** Distributed version: miniGhost 3D 27-point

```

1  !OMP PARALLEL DO stencil
2
3  DO K = 1, NZ
4  DO J = 1, NY
5  DO I = 1, NX
6
7      G2(i,j,k) = &
8      G1(I-1,J-1,K-1) + G1(I-1,J,K-1) + G1(I-1,J+1,K-1) + &
9      G1(I,J-1,K-1) + G1(I,J,K-1) + G1(I,J+1,K-1) + &
10     G1(I+1,J-1,K-1) + G1(I+1,J,K-1) + G1(I+1,J+1,K-1)
11  END DO
12  !dir nofusion
13  DO I = 1, NX
14     G2(i,j,k) = G2(i,j,k) + &
15     G1(I-1,J-1,K) + G1(I-1,J,K) + G1(I-1,J+1,K) + &
16     G1(I,J-1,K) + G1(I,J,K) + G1(I,J+1,K) + &
17     G1(I+1,J-1,K) + G1(I+1,J,K) + G1(I+1,J+1,K)
18  END DO
19  !dir nofusion
20  DO I = 1, NX
21     G2(i,j,k) = G2(i,j,k) + &
22     G1(I-1,J-1,K+1) + G1(I-1,J,K+1) + G1(I-1,J+1,K+1) + &
23     G1(I,J-1,K+1) + G1(I,J,K+1) + G1(I,J+1,K+1) + &
24     G1(I+1,J-1,K+1) + G1(I+1,J,K+1) + G1(I+1,J+1,K+1)
25
26     G2(i,j,k) = G2(i,j,k) * TWENTYSEVENTH
27
28  END DO ! End NX
29  END DO ! End NY
30  END DO ! End NZ

```



# Appendix D

## NVIDIA miniFE Study

This appendix expands on Section 4.2.2.

### D.1 Introduction

A study of the mini-FE application in CUDA was undertaken by NVIDIA. The purpose of this study was to identify key performance limiters for finite element applications on CUDA enabled architectures and use that knowledge to drive future hardware and software designs. The mini-FE algorithm is composed of three phases, they are the following: generate matrix structure, assemble the finite-element matrix, and solve. Of these the majority of time is spent in either the assembly or the solve. The matrix structure can often be reused and as such the cost of generating this structure is often small compared to the overall runtime.

The solve phase involves a sparse linear solve. There are many algorithms to solve sparse linear systems. Most of these algorithms are dominated by the performance of Sparse Matrix-Vector Multiplies (SPMVs). Mini-FE solves the system of linear equations using the Conjugate-Gradient algorithm. The performance of SPMV is bandwidth bound and is sensitive to the sparse-matrix format. A common sparse-matrix format is Compressed Sparse Row (CSR). This format, while ideal for serial processors, has uncoalesced memory accesses when accessed in parallel. We used an alternative matrix format known as ELLPACK which avoids the memory coalescing problems present in CSR [7, 17]. The effects of these formats on performance, as well as their limiters, are well understood and were not the focus of this work. As such, they will not be discussed further. For more details on sparse-matrix vector formats and their performance in CUDA see [7].

The assembly phase involves computing the element operators for each element and then summing those operators into the final matrix. We parallelized this phase by having each thread operate on separate elements. In addition, the computation of the element operator and the summation into the linear system was performed in a single kernel call. This scheme was chosen because the computation of the element operators is embarrassingly parallel. However, summing into the linear system does require synchronization to avoid race conditions. Using a single kernel is preferred over multiple kernels because it avoids having to store the state for the element operator and then reread that state back in when

summing into the linear system.

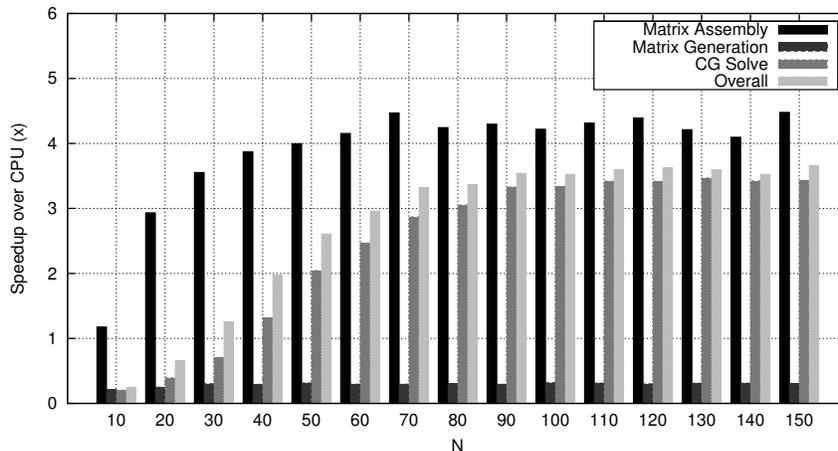
By using one thread per element we were able to leverage the original code for the construction of the element operator. Most of the changes for this portion of the code were limited to marking the original host functions with `__device__` so that they could be called from CUDA. The summation into the linear system also required a few changes. The major changes in this operation were to use the ELL matrix format instead of the CSR matrix format and to update the matrix using `atomicAdd`. The use of atomics prevents race conditions that arise from updating the global matrix in parallel.

The computation of the element operator involves a number of flop heavy operations including computing the determinant and the inverse of the Jacobian. These large number of flops suggest that the performance should be flop limited but analysis using NVIDIA's visual profiler showed that the performance is bandwidth bound due to register spilling. The element operator requires over 900 bytes of state for the computation. The large state is caused by the math heavy operations which require a lot of intermediates. The usage is further increased because the calculation is 3D and in double precision. The Fermi architecture only supports up to 63 32-byte registers per thread, limiting the total amount of register storage to 252 bytes. As a result of this limit the additional state must be spilled first to L1 cache, then to L2 cache, and finally to global memory. The L1 cache is configurable to be either 16K or 48K bytes per SM. In addition, the L2 cache size is 768K bytes and is shared across all SMs. The current implementation uses 512 threads per SM. This leaves 96 bytes of each L1 and L2 cache per thread. Since L1 and L2 are insufficiently sized to store the required state, registers are spilled to global memory causing the computation to become bandwidth bound.

The summation into the element operator is also bandwidth bound. This is expected as this operation adds contributions from the diffusion matrix to the global matrix. Atomics are another potential limiter for this operation. However, there are very few potential collisions and thus the number of instruction replays due to atomics is low. Experiments have shown that the use of atomics affected the total performance by less than a half percent.

One way to improve the performance of bandwidth bound kernels is to increase the occupancy. However, in this case, the kernel's occupancy is limited by register usage. Since the register usage is much higher than what is available in hardware it is not possible to increase this occupancy without also increasing register spilling.

Another method for increasing performance is to reduce the amount of register spilling. We made a number of optimizations to the kernel in order to reduce its register usage. We were able to reduce register pressure by making algorithmic changes including exploiting symmetry in the diffusion operator and reordering computations so that data was loaded in immediately before it was used. We were also able to utilize traditional optimization techniques including restricting pointers, inlining functions, and unrolling loops. These optimizations allow the compiler to be more aggressive and can have a large impact on performance. Finally we also placed some of the state in shared memory and experimented with L1 cache sizes. In the end, the best performance we found was when placing the source vector into



**Figure D.1.** Speedup of miniFE CUDA Implementation (NVIDIA Fermi M2090 vs. Hex-Core 2.7GHz E5-2680)

shared memory and enabling the larger L1 cache. These optimization greatly reduced register spilling. However, there was still 512 bytes of state being spilled per thread. All of these optimizations that were applicable to the original CPU code were back ported to the CPU code.

The performance of the CUDA version of miniFE was compared to the MPI-parallel version of miniFE. Timings were performed on the same system which included a Tesla M2090 and a six-core Intel Xeon E5-2680 at 2.7 GHz. We tested for various problem sizes of  $N^3$  hexahedral elements. In addition, the matrix structure was reused for 50 assembles and solves. The speedup for each of the three phases along with the overall speedup is reported in Figure D.1. Here we can see that the assembly phase achieves over a 4x speedup while the solve phase is over 3x speedup. The generation of the matrix structure exhibits a slowdown because it is generated on the host in CSR format, transferred to the device, and then converted to ELL format. It would be possible to move this computation to the device. However, in these runs the time for the generation of the structure did not dominate the overall performance and over a 3x speedup was observed overall.

One of the goals of the Mantevo project is to help steer future hardware and software development. In this case, the project was a success. From the analysis above we can conclude that the current limiter is bandwidth due to register spilling. There are a number of things that NVIDIA could do to improve the performance of such apps in the future. Hardware changes could include increasing the number of registers per thread, increasing the size of shared/L1 memory and L2 memory, improving the speed of register spilling, and improving the bandwidth utilization at lower occupancy. In addition, NVIDIAs compiler could more aggressively target register usage for large state applications. These findings have been shared within NVIDIA and will impact the design of their future hardware and compilers. In addition, the CUDA implementation could potentially be improved further.

One option would be to use multiple threads per element. For example, one thread per node would be a logical choice. This would potentially divide the amount of state required by the number of nodes. This increases the parallelism expressed in the algorithm and will likely be required as the demand for parallelism increases in the future.

# Appendix E

## Characterizing the Sensitivity of Charon and MiniFE to Variation in DDR Interface Frequency on Workstation-Class Multicore Processor Architectures

This appendix expands on Section 4.2.3.

### E.1 Introduction

It's well known that many of today's modeling and simulation applications are memory bound, but to what degree? Most high-end workstations allow programming the signaling frequency between a processor's memory controllers and the attached main memory DDR DIMMs. Using this feature, it's possible to perform an empirical study looking at the effect of main memory bandwidth on application performance. In this study a comparison is made between a mini-application and a production application in terms of their sensitivity to memory bandwidth. Charon is a semiconductor device simulation application based on a finite element method. MiniFE is a mini-application from the Mantevo project and its intent is to emulate the performance of the solve phase of a finite element application, such as Charon. One question often raised in the use of a mini-application is how representative is it of the application it is meant to model? In addition, just as valid a question is what it does not represent? In this study, Charon's and MiniFE's sensitivity to memory bandwidth is analyzed to help provide some insight.

### E.2 Test Beds and Method

The two test beds used for this study are described in Table E.1. The study is performed on two separate processor architectures in order to provide a higher level of confidence in the

Blade Model	HP ProLiant BL460c G6	HP ProLiant BL465c
Processor	Intel Nehalem 5570	AMD Magny-Cours 6136
Compiler Suite	Intel 11.0.081	Intel 11.0.081
MPI	OpenMPI 1.3.3	OpenMPI 1.4.3
# sockets	2	2
# cores/socket	4	8
Total # of cores	8	16
CPU frequency (GHz)	2.93	2.4
Memory Capacity	24 GB (3x4 GB/socket)	32 GB (4x4 GB/socket)
DDR3 (maximum)	1333	1333
Peak FLOP/s	93.76 GFLOP/s	153.6 GFLOP/s
Peak memory bandwidth	64.9 GB/s	85.3 GB/s
Nominal byte/FLOP	0.692	0.555
Misc	Turbo mode off	

**Table E.1.** Test Bed Descriptions

results. Initially, the method was applied to an AMD Magny-Cours based workstation. To see if the results are repeatable for different processor architectures, the method was performed on an Intel Nehalem based system. Both test beds are based on multicore processors that are representative of the node architecture used by many HPC platforms in operation today. Both of the target workstations are blade based and are contained in a Hewlett-Packard BladeSystem c3000 chassis. They are managed as standalone, independent workstations. Each test bed is capable of varying the memory controller to DIMM interface frequency at the rates of 800 Mhz, 1066 Mhz and 1333 Mhz. By default, the blades auto-configure the frequency to the capability of the DIMMs populated in the system, in the case of the two blades under test in this study 1333 Mhz DDR3. By reconfiguring via the BIOS, the interface frequency can be set to be a maximum of 1066 Mhz or 800 Mhz. Using this feature, results for key runtime metrics associated with Charon and MiniFE were collected for all three BIOS settings.

For miniFE, two run time metrics are reported: the array assembly time and the total time for the conjugate gradient solver. For Charon, four metrics are reported: preconditioner time/iteration, solver time/iteration, Jacobian time/iteration and total advance time/iteration. Absolute run times are not presented in this study. A more relevant metric is performance relative to a baseline reading. All results shown are relative to the nominal 1333 Mhz result.

### E.3 Results

Results for miniFE are shown in Figure E.1. Two observations are noteworthy. First, the

finite assembly step is not sensitive to memory bandwidth, which implies that this step is completely compute bound. Second, the CG solver phase is sensitive to memory bandwidth and hence is to some degree a memory bound problem. Third, for the CG phase the AMD and Intel results are similar and hence the microarchitecture of the processor doesn't affect the degree to which the problem responds to decreasing memory bandwidth, at least to within a few percent. Note that this is not a statement about the absolute performance of the respective processors. But an observation that a given decrease in memory bandwidth has the same relative affect on overall performance. This is a powerful observation and supports the notion that memory bandwidth is a better indicator of performance than the common use of peak FLOP/s.

It's interesting to note that on the Nehalem workstation, the assembly time is slightly faster at the slower 800 Mhz frequency. This is repeatable and is not a measurement anomaly.

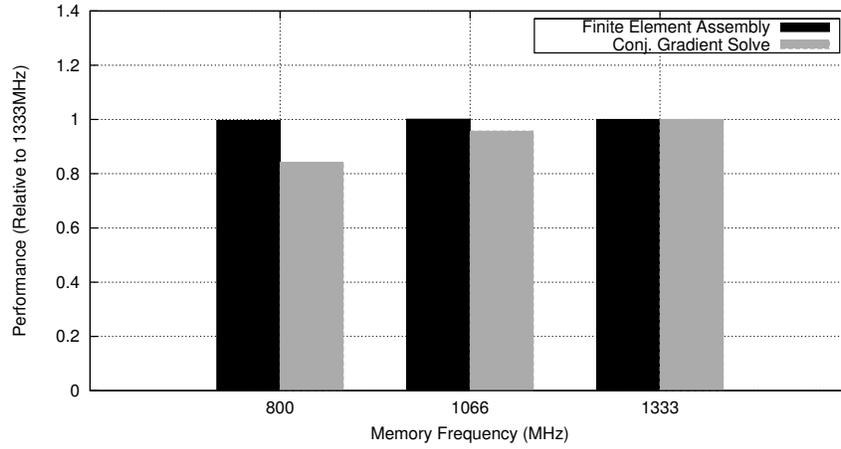
Results for Charon are shown in Figure E.2. Again, both processor architectures react similarly, within a few percent, to varying memory bandwidth. The preconditioner step is relatively insensitive to varying bandwidth. As was seen with miniFE, the solver is the most sensitive. The Jacobian step is unaffected, and total advance time is a function of the prior steps and since the solve phase is the most dominant total time it also reflects the same trend.

One of the fundamental questions when using a mini-application is - In what way does it represent a "real" application. Above, it was shown that the calculation that is most sensitive to memory bandwidth is the solver phase. In miniFE, a relatively fundamental CG algorithm is used for the solver, while the Charon application, solving a nonsymmetric system, uses TFQMR [15] or GMRes [24]. In Figure E.3, the relative solver performance for each is charted. These results show that in the case of the solver, miniFE is a very accurate proxy for Charon in terms of studying the affect of varying memory bandwidth. MiniFE tracks Charon's sensitivity to within a few percent. Again, it can be seen that the results using the Intel processor is consistent with that seen using the AMD processor.

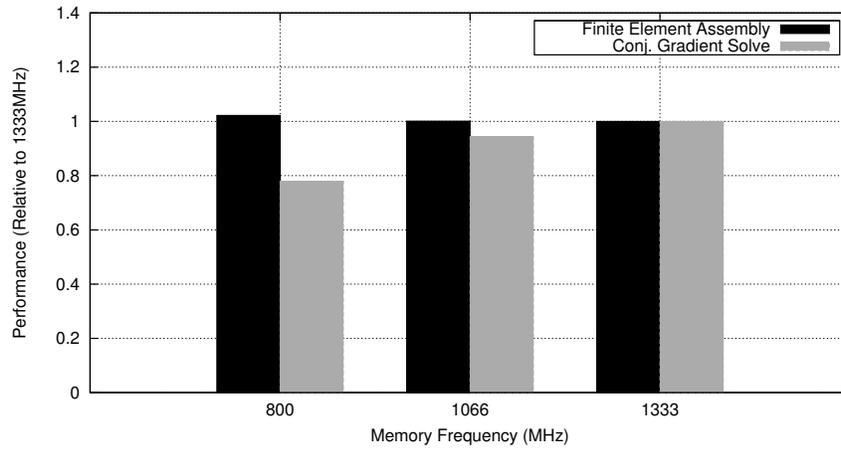
## E.4 Summary and Conclusions

In this study, an empirical method was used to measure the impact of main memory bandwidth on application performance. This was performed by varying the signaling frequency of the DDR interface between the memory controllers and their respective DIMMs. The miniFE mini-application and Charon application were studied. For each, metrics of interest were identified and analyzed as the memory interface frequency was set at 800 Mhz, 1066 Mhz and 1333 Mhz. Some metrics, such as assembly time in miniFE and the preconditioner step in Charon showed to be insensitive to memory bandwidth. However, the solvers in each application were sensitive to memory bandwidth. This is not surprising, but it is interesting to have a quantitative measure. It was also demonstrated that the same trends were seen using two different workstation architectures, which provides confidence in the assertion that

for solver based applications the important metric is memory bandwidth, not FLOP/s. Finally, it was shown that miniFE's CG method is a good proxy for more advanced solvers found in Charon. This provides the users of miniFE confidence that memory subsystem performance architecture studies are valid.

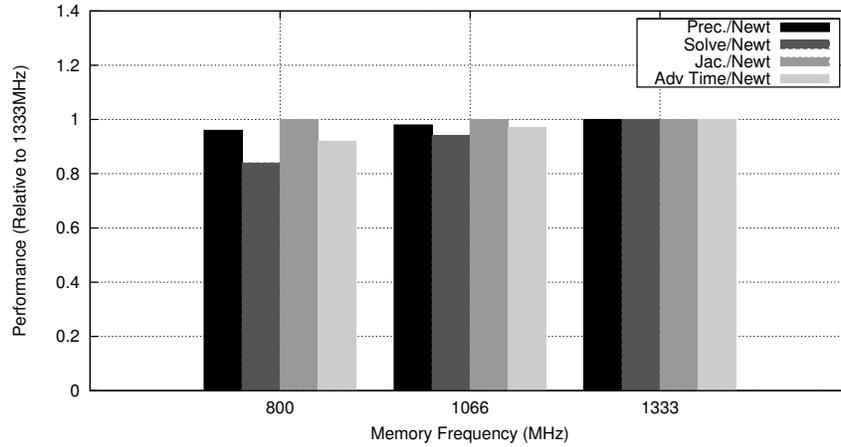


(a) Dual-Socket, Oct-core AMD 2.4GHz Magny-Cours 6136

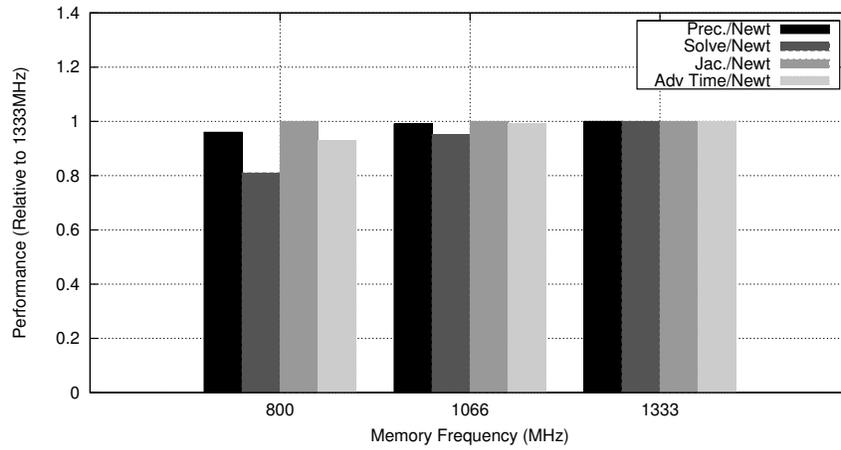


(b) Dual-Socket, Quad-core Intel 2.93GHz Nehalem 5570

**Figure E.1.** Memory Bandwidth: miniFE Finite Element Mini-Application

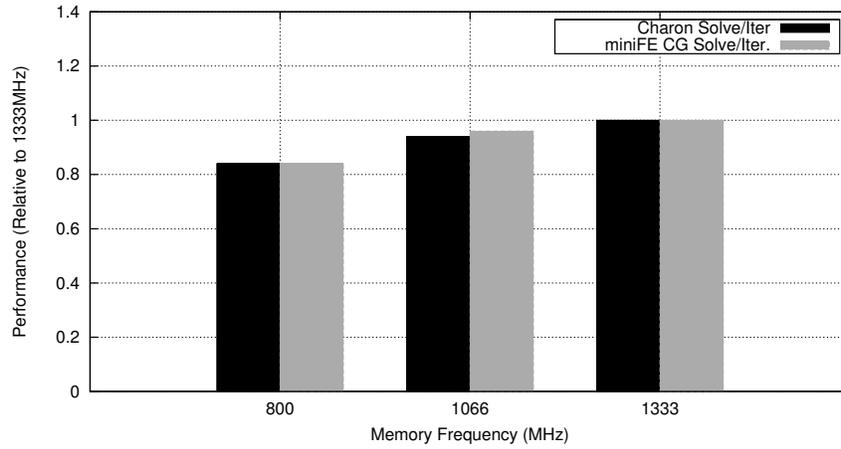


(a) Dual-Socket, Oct-core AMD 2.4GHz Magny-Cours 6136

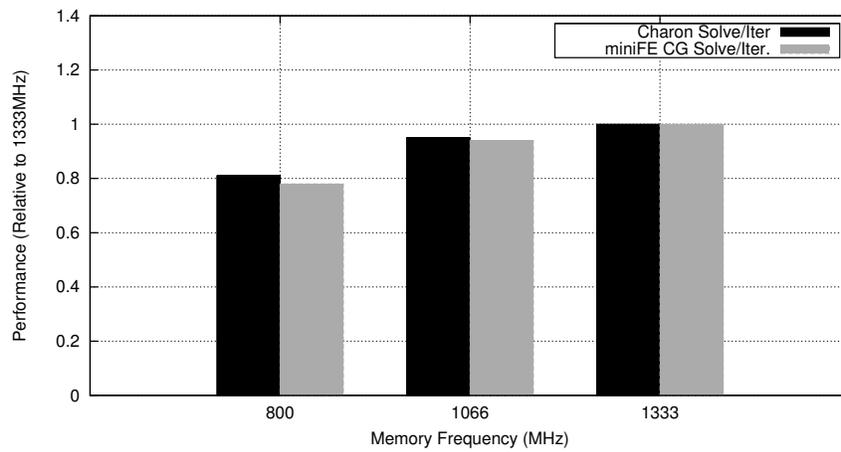


(b) Dual-Socket, Quad-core Intel 2.93GHz Nehalem 5570

**Figure E.2.** Memory Bandwidth: Charon Device Simulation Application



(a) Dual-Socket, Oct-core AMD 2.4GHz Magny-Cours 6136



(b) Dual-Socket, Quad-core Intel 2.93GHz Nehalem 5570

**Figure E.3.** Memory Bandwidth: miniFE compared to Charon



# Appendix F

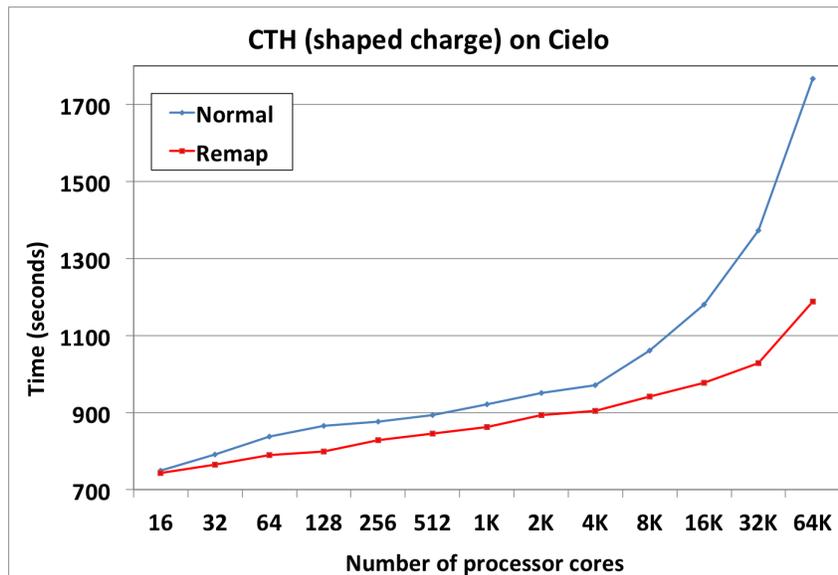
## Remapping the parallel processes in CTH

This appendix expands on Section 4.3.2.

Work as part of the recently completed L2 milestone, “Report of Experiments and Evidence for ASC L2 Milestone 4467 – Demonstration of a Legacy Application’s Path to Exascale” [3] exposed a scaling issue on Cielo in the boundary exchange in miniGhost. MiniGhost was configured to serve as a proxy for CTH. CTH, used throughout the United States DOE complex, and is part of the US Department of Defense (DoD) High Performance Computing Modernization Program (HPCMP) test suite [9], is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories [18]. Solving the Lagrangian equations using second-order accurate numerical methods and mesh remap to reduce dispersion and dissipation, it includes models for multi-phase, elastic viscoplastic, porous and explosive materials, CTH divides the domain into three dimensional regions (illustrated in Figure 4.6), which are mapped to parallel processes. For the shaped charge simulation considered herein, each time step CTH exchanges boundary data (two dimensional “faces”) 19 times, in each of the three dimensions, with three calls to propagate data across faces. Each boundary exchange aggregates data from (up to) 40 arrays, representing 40 variables, resulting in messages on the order of a few MBytes for representative problems.

The MPI rank remap strategy described in Section 4.3.2 showed a significant scaling benefit for miniGhost. The issue was seen clearly in the time spent sending boundary data to neighbors in the  $z$  direction, illustrated in Figure 4.5(b). Its also interesting to note that there is no effect on the performance of the global reduction function `MPI_Allreduce`.

We implemented this strategy in CTH, the application miniGhost was configured to represent. Results, shown in Figure F.1, illustrate a similar benefit to the full application, demonstrating the predictive capability miniGhost is designed to provide. These runs were made on Cielo in non-dedicated mode, so the scaling issue that seems to be appearing at 64k cores may be the result of competing jobs. Additional work in this area, including remapping of processes in the alternative SVAF strategy will be reported in subsequent publications.



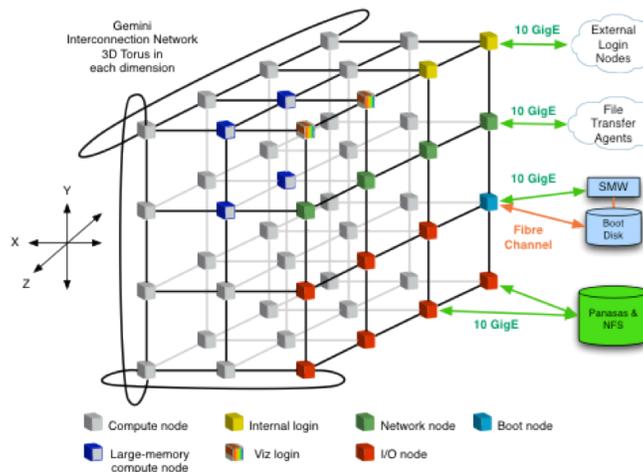
**Figure F.1.** Performance of CTH with MPI-rank remapping on Cielo.

# Appendix G

## More on Cielo

Cielo is the latest Advanced Simulation and Computing (ASC) capability machine. Although not technically a testbed system since the architecture is widely available, Cielo provides some new capabilities and hardware features which we expect will be extended and refined in future architectures.

An instantiation of a Cray XE6, Cielo is composed of AMD Opteron Magny-Cours processors, connected using a Cray custom interconnect named Gemini, and a light-weight kernel (LWK) operating system called Compute Node Linux. The system, illustrated in Figure G.1, consists of 8,944 compute nodes, for a total of 143,104 cores.

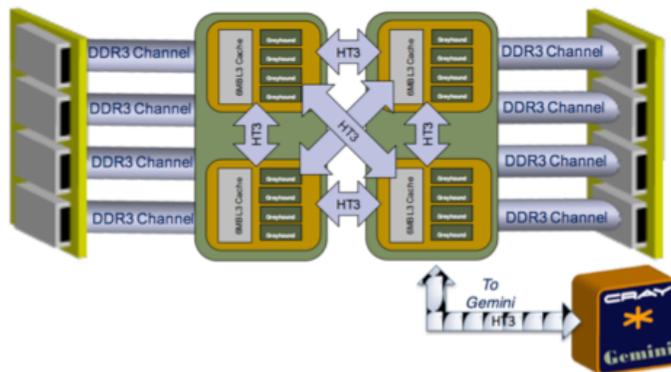


**Figure G.1.** Cielo XE6 architecture. Image courtesy of Cray Inc.

Each Cielo node consists of two oct-core AMD Opteron Magny-Cours processors<sup>1</sup>. Each Magny-Cours processor is divided into two memory regions, called NUMA nodes, each con-

<sup>1</sup>Magny-Cours processors are also available with 12 cores divided into 6-core NUMA nodes, which form the basis of the new Hopper II computer at NERSC (<http://www.nersc.gov/nusers/systems/hopper2/>).

sisting of four processor cores (illustrated in Figure G.2). Thus each compute node consists



**Figure G.2.** The XE6 compute node architecture. Images courtesy of Cray, Inc.

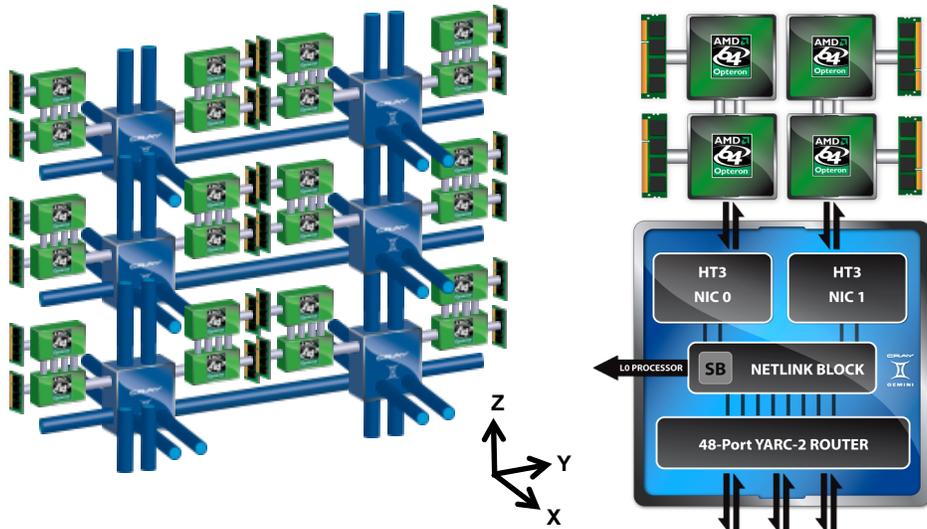
of 16 processor cores, evenly divided among four NUMA nodes, which are connected using HyperTransport<sup>2</sup> version 3. The links between NUMA nodes run at 6.4 GigaTransfers per second (GT/s). Each core has a dedicated 64 kByte L1 data cache, a 64 kByte L1 instruction cache, and a 512 kByte L2 data cache, and the cores within a NUMA node share a 6 MByte L3 cache (of which 5 MBytes are user available).

Cielo compute nodes are connected using Cray’s Gemini 3-D torus high-speed interconnect, illustrated in Figure G.3. 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 introduces 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. 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.

Cielo’s node interconnect, called Gemini, is configured as a three dimensional torus topology as  $16 \times 12 \times 24$ .

Figure 4.4 (in the main body of this paper) shows the CTH nearest neighbor communication pattern as it is mapped to Gemini’s three dimensional torus topology for Cielo.

<sup>2</sup><http://www.hypertransport.org>



(a) Cray XE6 Gemini Architecture

(b) XE6 Node Architecture with AMD Magny-Cours processors and Cray Gemini high-speed interconnect

**Figure G.3.** The XE6 Gemini architecture. Images courtesy of Cray, Inc.







**Sandia National Laboratories**