SANDIA REPORT
SAND2015-0454
Unlimited Release
Printed January 2015

Trinity Benchmarks on Intel Xeon Phi (Knights Corner)

Mahesh Rajan, Doug Doerfler, Si Hammond

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.

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

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

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

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

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

SAND2015-0454
Unlimited Release
Printed January 2015

Trinity Benchmarks on Intel Xeon Phi (Knights Corner)

Mahesh Rajan, Doug Doerfler, Si Hammond
Org.9326, Org. 1431
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS0807

Abstract

This report documents the early experiences with porting and performance analysis of the Tri-Lab Trinity benchmark applications on Intel Xeon Phi (Knights Corner) (KNC) processor. KNC, the second generation of the Intel Many Integrated Core (MIC) architectures, uses a large number of small P54C-x86 cores with wide vector units and is deployed as PCI bus attached process accelerators. Sandia has experimental test beds of small InifiniBand clusters and workstations to investigate the performance of the MIC architecture. On these experimental test beds the programming models that may be investigated are "offload", "symmetric" and "native". Among these program usage models our primary interest is in the so called "native" mode, because the planned Trinity system to be deployed in 2016 using the next generation MIC processor architecture called Knights Landing would be self-hosted. Trinity / NERSC-8 benchmark programs cover a variety of scientific disciplines and they were used to guide the procurement of these systems. Architectures such as the Intel MIC are well suited to study evolving processor architectures and a usage model commonly referred to as MPI + X that facilitates migration of our applications to use both coarse grain and fine grain parallelism. Our focus with the applications selected is on the efficacy of algorithms in these applications to take advantage of features like: large number of cores, wide vector units, higher-bandwidth and deeper memory sub-system. This is a first step towards understanding applications, algorithms and programming environments for Trinity and future exascale computing systems.

CONTENTS

Contents

FIGURES

TABLES

1. INTRODUCTION AND OBJECTIVES OF THIS INVESTIGATION

As of writing this report there are three dominant programming models on systems with KNC. The systems on which we investigated the KNC performance have Intel Xeon E5-2670 Sandy Bridge (SB) processors on the compute nodes (host) with one or more Xeon Phi Knights Corner coprocessors attached to the system PCIe bus. The programming models supported in such configurations are: MPI + Offload, Native Mode and Symmetric Mode. In all three approaches to using the KNC, an application could be built with a Hybrid (MPI + Threads) computational model. Our primary focus on this report is on the Native mode. This is because the recently announced procurement of NNSA/ACES Trinity system will have a large number of compute nodes (nearly half) using the Intel Xeon Phi Knights Landing (KNL) processor. The Xeon Phi Knights Landing on Trinity will operate in 'self-hosted' mode. An application built today to use the MIC's native mode runs entirely on KNC. The executable is not binary compatible with the host Sandy Bridge processor. The instruction set for MIC is similar to the Intel Pentium 4, but not all of the 64 bit scalar extensions are included. MIC also has 512 bit vector/SIMD registers but does not support MMX, SSE or AVX instructions. Neither can applications built for the host Sandy Bridge processor run on the MIC. The primary reason for our interest in studying the Native Mode usage model is that it lays the foundation for application migration to Trinity and this model is the least disruptive in adapting Sandia's production application to the MIC architecture. Even on the other half of the compute nodes on Trinity that incorporate the Intel Haswell processors, our investigation of applications on their efficacy in the use of a hybrid (MPI + OpenMP) programming model and vectorization would be relevant.

As described in the subsequent sections our systems with KNC, have 57 or 61 cores on the processor with each core capable of supporting four threads in hardware. Each hardware core has 512 wide SIMD/vector registers. All of the cores have fully coherent L1 and L2 caches and share 8GB or more of GDDR5 memory.

This investigation focuses on:

1) The KNC performance profile with different combinations of MPI tasks and OpenMP threads sweeping through a range spanning 1 MPI task and 240 OpenMP threads to 240 MPI tasks each with one OpenMP thread
2) Comparative performance on the host Sandy Bridge two processor nodes with 16 cores
3) The best choice of MPI task and OpenMP thread affinity settings to yield optimal performance.
4) Preliminary investigations on the ability to maximize performance through exploitation of 512 bit SIMD/vector registers.

Some of the Trinity benchmarks with a mature Hybrid (MPI + OpenMP) implementation were well suited to the investigation of balance between MPI tasks and OpenMP threads. We supplemented Trinity benchmarks with few other hybrid applications such as the NAS benchmarks to further gain insights on the optimal use of the KNC architecture. One of benefits of the Intel MIC architecture is the program development environment helps maintain a

single source code, while permitting migration to the many-core architectures with familiar x86 based compiler, development tools and libraries.

2. HARDWARE USED

2.1. Corner Workstation

For most of our performance investigation of the MIC/KNC architecture, a dual-socket workstation with dual 8 core Xeon E5-2670 (Sandy Bridge) processors with 64GB of RAM memory is used. It has been configured with two MIC PCIe attached Knight Corner processors; stepping C0 ES2, 1.2 GHz (1.3 GHz turbo), 61-core, 16 GB on 4 Gb technology. This workstation is named "Corner". A block diagram of the KNC processor with 61 cores is shown in Figure 1. The core architecture is shown in Figure 2.

The KNC processor is primarily composed of CPU cores, caches, memory controllers, PCIe client logic, and a high bandwidth, bidirectional ring interconnect. There are a total of 61 cores on each KNC. Each core uses a short in-order pipeline and is capable of supporting 4 threads in hardware. Each core has a private 32KB instruction and 32KB data L1 cache, and a private 512 KB L2 cache that is kept fully coherent by a global-distributed tag directory as shown in Figure 2. The memory controllers and the PCIe client logic provide a direct interface to the 8GB GDDR5 memory on the processor and the PCIe bus, respectively. All these components are connected together by the ring interconnect.

Multiple Intel Xeon Phi coprocessors can be installed in a single host system. Within a single system, the coprocessors can communicate with each other through the PCIe peer-to-peer interconnects without any intervention from the host. Similarly, the coprocessors can also communicate through a network card such as InfiniBand or Ethernet, without any intervention from the host.

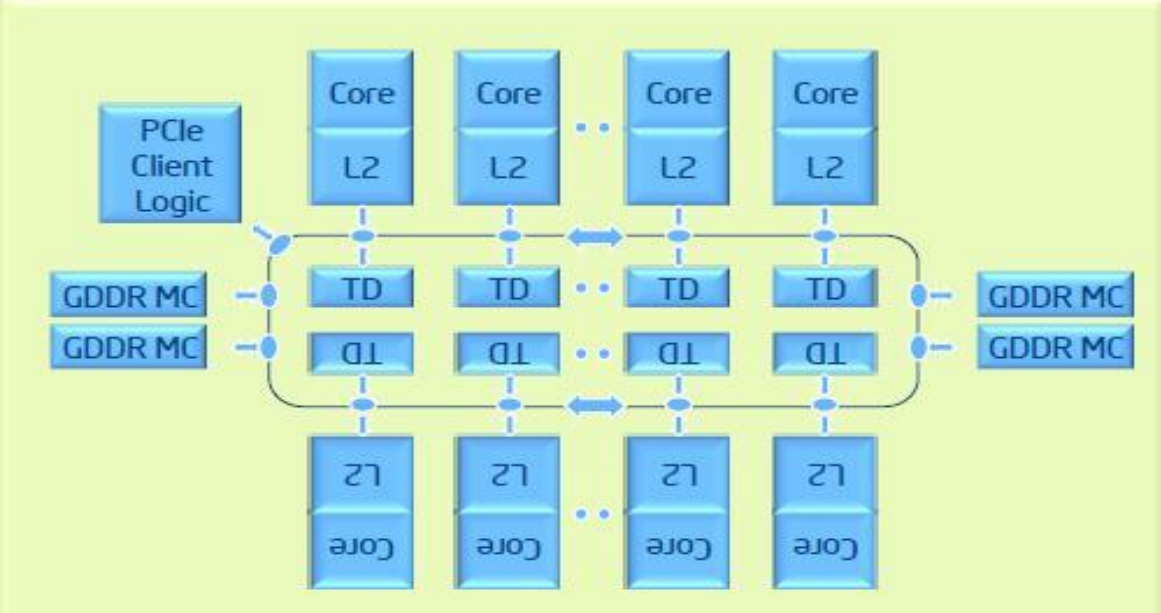

Figure 1 Knights Corner Block Diagram

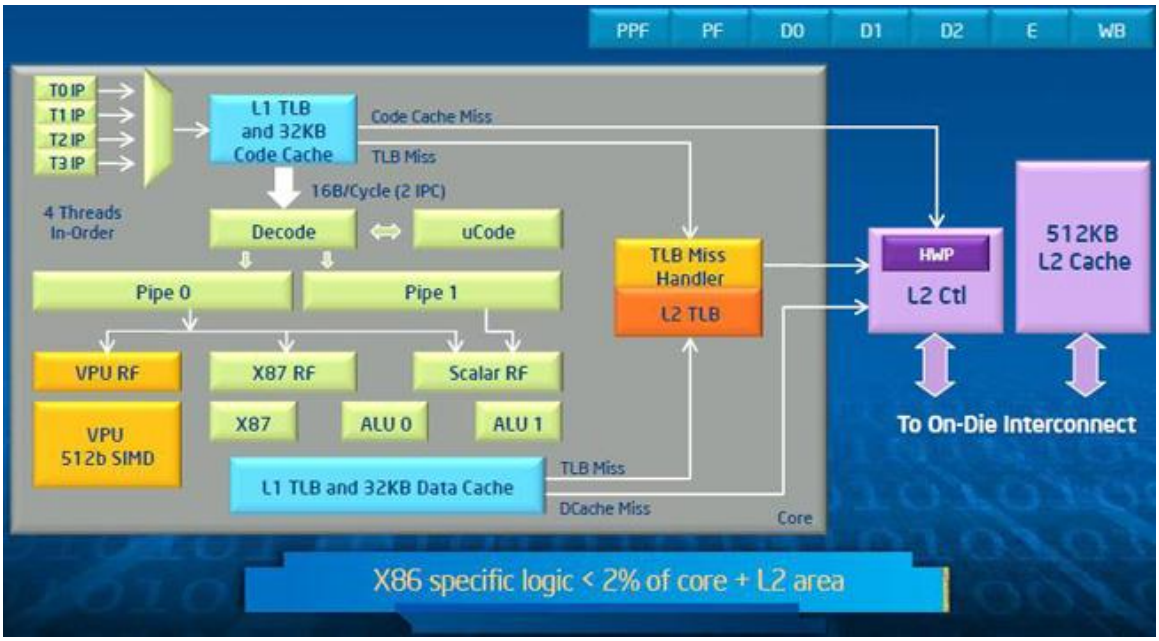

Figure 2 Knights Corner Core Architecture

An important component of the Intel Xeon Phi coprocessor's core is its vector processing unit (VPU), shown in Figure 3. The VPU features a 512-bit SIMD instruction set, officially known as Intel Initial Many Core Instructions (Intel IMCI). Thus, the VPU can execute 16 single-precision (SP) or 8 double-precision (DP) operations per cycle. The VPU also supports Fused Multiply-Add (FMA) instructions and hence can execute 32 SP or 16 DP floating point operations per cycle. It also provides support for integers.

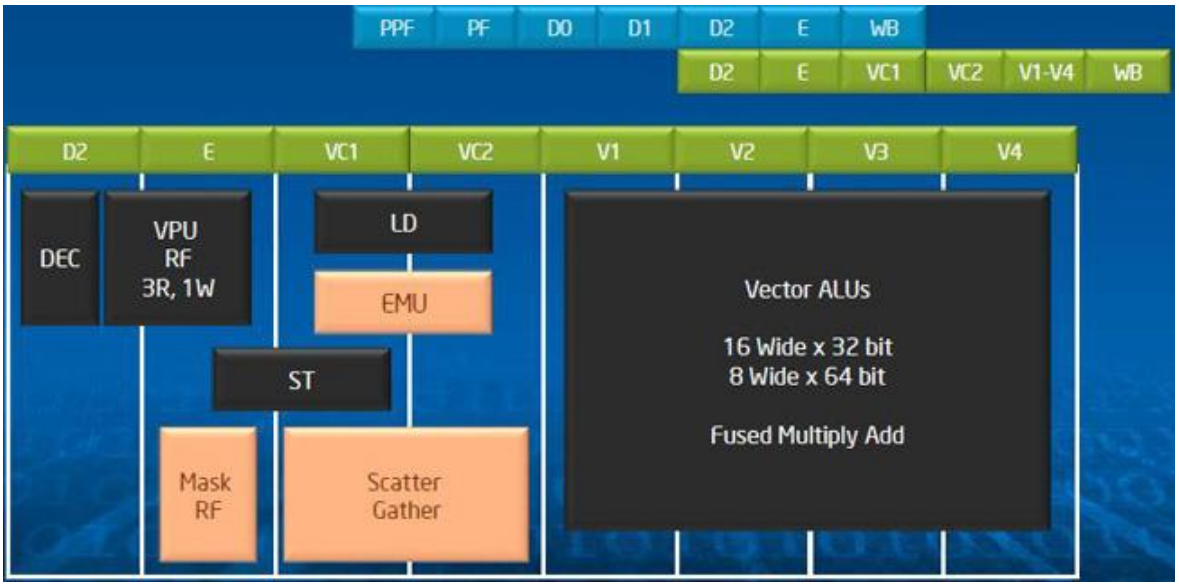

Figure 3 Knights Corner Vector/SIMD Unit

2.2. Compton InfiniBand cluster

The Compton cluster is an Appro InifiniBand cluster with 42 compute nodes and is similar to the production TLCC2 capacity clusters called Chama and Pecos. Each node has two 8-core Sandy Bridge, Xeon E5-2670 2.6 GHz processors. The nodes differ from Chama and Pecos nodes in that each node has two 1.1 GHz Knights Corner processors, each with 57 cores and 6GB of GDRR5 memory. The compute nodes are connected with a Mellanox Infiniscale QDR InfiniBand interconnect. For benchmarking purposes one must be aware that unlike on Chama and Pecos, on Compton Hyperthreading is active. The Knights Corner processor used in Compton comes from a different SKU to that used in the "corner" workstation and so has only 57 active cores.

3. SOFTWARE ENVIRONMENT

3.1. Corner Workstation

3.1.1. Intel Compiler and Composer version

All the needed software for program development is available in /opt/intel/composer_xe_2013_sp1. The latest install date is March 20, 2014. This environment consisted of compilers icc, ifort versions 14.0.2, with MKL libraries version 11.1 update 2, TBB libraries version 4.2 update 3. It also included Intel MPI, impi version 4.1.3 and vtune_amplifier_xe 2013 Update 16.

3.1.2. Setup for Host/Sandy Bridge compile and run

The setup for compiling and running on the host Sandy Bridge processors requires sourcing the file:

/opt/intel/bin/compilervars.sh intel64.

3.1.3. Setup for MIC/Xeon Phi compile and run in native mode

The steps involved in running on KNC in native mode are:

1) source /opt/intel/bin/compilervars.sh intel64
2) use -mmic flag to compile code to run native on MIC; -O2 and above compiler optimization flags invokes vectorization
3) Copy all the needed libraries (like MPI, MKL, etc.) to MIC. The script that was used for this called *load-mic.sh* shown below
4) scp the executable to MIC: (example; *sudo scp ./hello_mpi-mic mic0:/tmp*)
5) ssh to the MIC and cd to /tmp and run (example; sudo ssh mic0; cd /tmp; mpirun -n 4 ./hello_mpi-mic)

The *load-mic.sh* shown below copies the necessary libraries to MIC.

#!/bin/sh

sudo scp /opt/intel/impi/4.1.0/mic/bin/mpiexec.hydra mic0:/bin
sudo ssh mic0 ln -s /bin/mpiexec.hydra /bin/mpiexec
sudo scp /opt/intel/impi/4.1.0/mic/bin/mpirun mic0:/bin
sudo scp /opt/intel/impi/4.1.0/mic/bin/pmi_proxy mic0:/bin
sudo scp /opt/intel/impi/4.1.0/mic/lib/libmpi.so.4.1 mic0:/lib64
sudo ssh mic0 ln -s /lib64/libmpi.so.4.1 /lib64/libmpi.so.4
sudo scp /opt/intel/impi/4.1.0/mic/lib/libmpigf.so.4.1 mic0:/lib64
sudo ssh mic0 ln -s /lib64/libmpigf.so.4.1 /lib64/libmpigf.so.4
sudo scp /opt/intel/impi/4.1.0/mic/lib/libmpigc4.so.4.1 mic0:/lib64
sudo ssh mic0 ln -s /lib64/libmpigc4.so.4.1 /lib64/libmpigc4.so.4
sudo scp /opt/intel/impi/4.1.0/mic/lib/libmpi_mt.so.4.1 mic0:/lib64
sudo ssh mic0 ln -s /lib64/libmpi_mt.so.4.1 /lib64/libmpi_mt.so.4
sudo scp /opt/intel/impi/4.1.0.027/mic/lib/libmpi_dbg.so.4 mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/lib/mic/libimf.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/lib/mic/libsvml.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/lib/mic/libintlc.so.5 mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/lib/mic/libiomp5.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/lib/mic/libirng.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/lib/mic/libirng.so mic0:/lib64
sudo scp /opt/intel/mkl/lib/mic/libmkl_sequential.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/mkl/lib/mic/libmkl_intel_lp64.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/mkl/lib/mic/libmkl_intel_thread.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/mkl/lib/mic/libmkl_core.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/mkl/lib/mic/libmkl_cdft_core.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/mkl/lib/mic/libmkl_scalapack_lp64.so mic0:/lib64
sudo scp /opt/intel/composer_xe_2013/mkl/lib/mic/libmkl_scalapack_lp64.so mic0:/lib64
sudo scp /opt/intel/impi/4.1.0/mic/lib/libmpi_dbg_mt.so.4 mic0:/lib64

3.2. Compton InfiniBand cluster

The software environment on Compton is similar to Chama in the sense it has SLURM and supports modules. Both Intel MPI and OpenMPI can be used with Intel compilers or GNU compilers. The /home and /projects directories are visible when running in native mode on the MIC and so running a program on the MIC is simpler than on the corner workstation as there is no need to explicitly copy the executable or input/output files to and from the local memory on the MIC.

4. PROGRAM DEVELOPMENT TOOLS

4.1. Intel Parallel Studio XE

The primary program development tool used in this effort are packaged with the Intel Parallel Studio XE and consists of:

- C++ , C and Fortran Compilers
- Thread Building Blocks (TBB)
- Math Kernel Library (MKL)

- OpenMP, version 4.0
- Advisor XE for threading design and prototyping
- Inspector XE for memory and thread debugging
- VTune Amplifier XE for performance profiling
- MPI Library

Details of these software Intel software elements can be easily located on the Intel web site: https://software.intel.com/en-us/intel-parallel-studio-xe

5. MICRO BENCHMARKS

5.1. Matrix Multiply

The objectives of this benchmark are:

1) Evaluate performance of MKL's threaded SGEMM and DGEMM, and compare it to the theoretical peak performance. The performance is compared to the theoretical peak to gauge what percentage of the peak we are able to achieve.
2) Compare against host results and understand status of threading efficiency and MKL's ability to achieve fine grain parallelization and vectorization.
3) This benchmark also serves as a good upper bound of the sustained performance we can anticipate with code kernels that have good data locality.
4) Can also use this code with tools such as PAPI to understand performance tuning and hardware counter measures with this simple kernel.

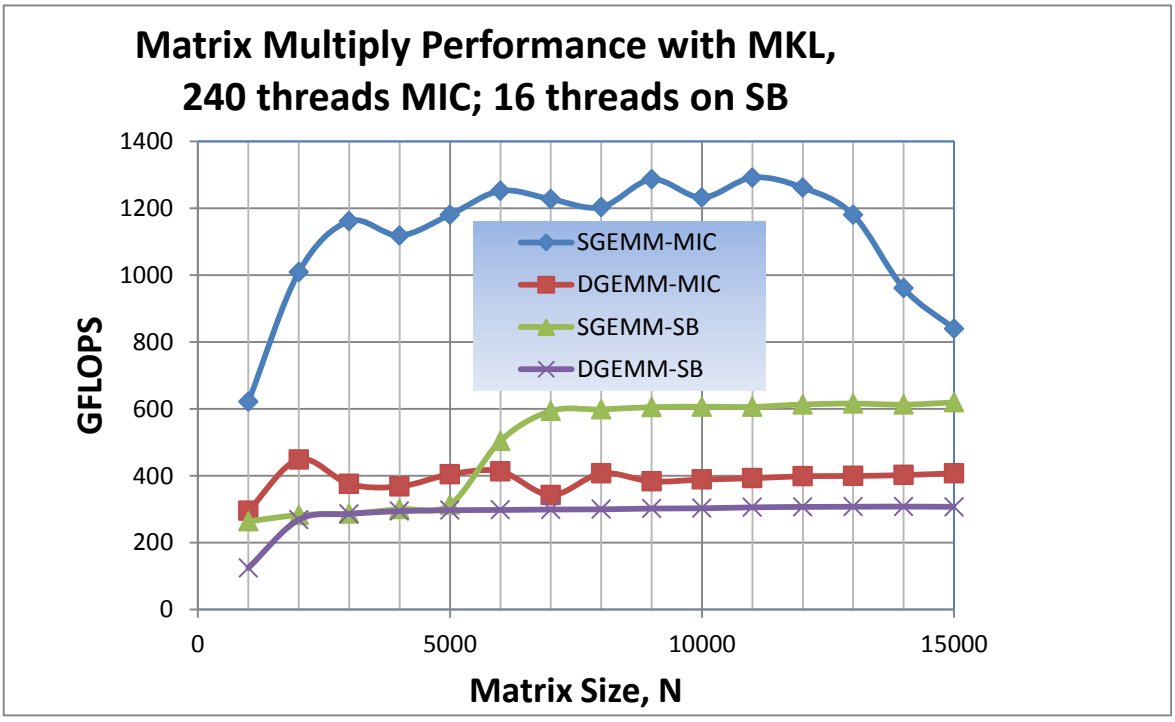

Figure 4 Matrix multiply performance on the MIC and the host SB with MKL

Figure 4 shows the measured performance on both the MIC and the dual-processor host Sandy Bridge node. For both the measurements the same program that call the threaded 'SGEMM' and 'DGEMM' MKL routines were used. While the measured performance for SGEMM on the MIC showed about 54% of the peak with 240 threads, the percent of peak on the host was close to 93%. Similarly for DGEMM the percentage of peak on MIC did not exceed 38% while on the host it was 93%. In a recent publication, Heinecke et.al. [1], outline an algorithm that takes full advantage of the KNC's salient architectural features to achieve close to 90% of the peak. This apparent large gap between our measured performance using MKL and this demonstrated high performance is not fully understood and needs further study.

An objective of this SAND Report is also to gain experience with use of hardware event counters, which are described in Intel Reference [2], and [3]. Micro-architectural performance tuning using the hardware events available through the built-in Performance Monitoring Unit (PMU) can be accessed through Intel's Vtune. We have recently installed a version of the TAU[4] performance tool on Compton and used it to measure hardware counter metric ratios like Vectorization intensity defined as:

Vectorization Intensity=VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED

For this matrix multiply benchmark using MKL's DGEMM, a few measurements of this metric ratio on the MIC with 4 and 8 threads gave a vectorization intensity value of 7.84. As suggested in Reference [3] this metric has an upper bound of 8 and so values close it suggest efficient use of MIC's SIMD units. However since the VPU_ELEMENTS_ACTIVE counter measures vector instructions like vector load/stores from memory, and instructions to manipulate vector mask registers, in addition to the double precision floating point instructions of interest to us, caution is needed in use of this metric for performance tuning. The fact that our measurements of this metric achieves close to the peak showing high vectorization intensity is misleading if our goal is to achieve high floating point operations throughput. The percentage of peak double precision floating point operations achieved in this test (which agrees with the values shown in Figure 4 when all the threads on the MIC's cores are utilized) is about 30%, which as stated previously is considerably less than published best performance of close to 90% [1]. However since at the present time the Xeon Phi does not have a PMU event to measure floating point performance, we still plan to use this Vectorization Intensity metric to give us some insights to the effective use of the SIMD units.

5.2. STREAMS

The objectives of this benchmark are:

1) To measure the STREAMS memory bandwidth with an OpenMP version of the STREAMS benchmark.
2) To investigate impact of affinity settings on the measured performance

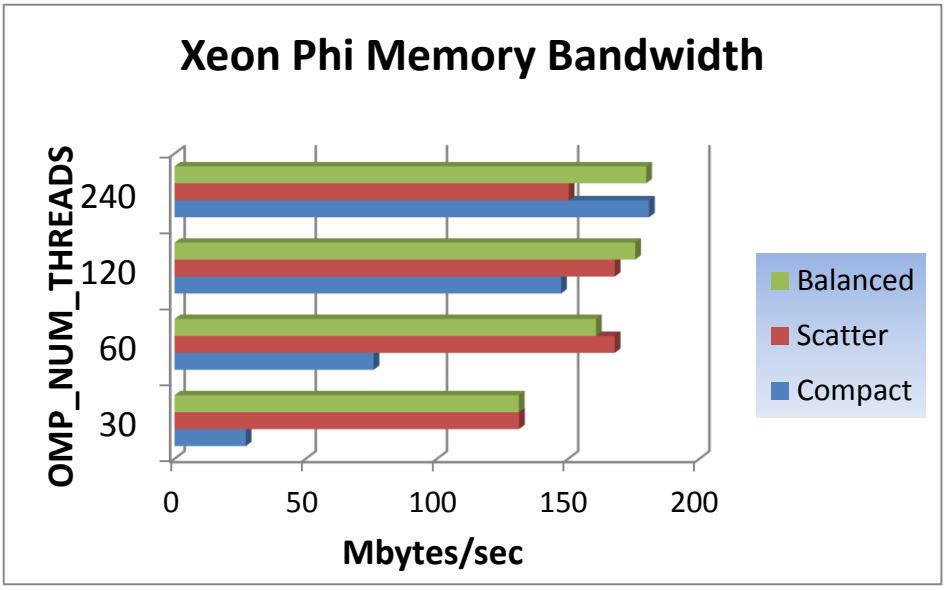

Figure 5 STREAMS Triad memory bandwidth on KNC

Figure 5 shows the KNC achieving an impressive 180 Gbytes/sec. This may be compared to about 74GBytes/sec on the host dual Sandy Bridge nodes. Runs on the MIC used the environment variables KMP_PLACE_THREADS=,4T (i.e. 4 threads per core) and KMP_AFFINITY=compact (or scatter or balanced).

Attempts to validate these measurements on Compton and Morgan, gave a maximum achieved STREAM Triad bandwidth of 133GB/s. Not sure as to the reasons for this discrepancy, but published numbers by other investigators are close to the lower values and suggest that the differences may be attributable to units with ECC correction enabled and higher values when it is disabled. This needs further investigation.

6. APPLICATION BENCHMARKS

6.1 miniFE

miniFE is a Finite Element mini-application which implements a couple of kernels representative of implicit finite-element applications. It assembles a sparse linear-system from the steady-state conduction equation on a brick-shaped problem domain of linear 8-node hex elements. It then solves the linear-system using a simple un-preconditioned conjugate-gradient algorithm.

Thus the kernels that it contains are:

- computation of element-operators (diffusion matrix, source vector)
- assembly (scattering element-operators into sparse matrix and vector)
- sparse matrix-vector product (during CG solve)
- vector operations (level-1 blas: axpy, dot, norm)

This version of miniFE has support for OpenMP. However, it is not deemed to be complete and/or optimal. There is scope to tune OpenMP sections to particular architecture. This version of miniFE corresponds to miniFE_ref_1.4b.

6.1.1 Hybrid (OpenMP+MPI) code performance

The code uses OpenMP pragmas in the 'for loops': computing BLAS daxpy type operations (in function waxpby(), computing a reduction BLAS ddot type operations (in function dot()) and in the computing Matrix-Vector products in function matvec().

The code was compiled with Intel C++ compiler version: 'icpc (ICC) 13.0.1 20121010' and with compiler flags '-mmic -O3 -openmp' for running on the Xeon Phi in the native mode. The same compiler was used for generating the executable that was run on the host with the same compiler flags except '–mmic' option. The process/thread affinity settings used were: KMP_AFFINITY=compact,granularity=fine and I_MPI_PIN_DOMAIN=omp.

As mentioned in the introduction one objective of this investigation is to find the optimal combination of MPI tasks and OpenMP threads that gives the best performance. Taking the smallest run time among three trials, Figure 6 shows the run time for the three dominant compute kernels, MATVEC, DOT and WAXPBY. The data for KNC shown on the left half of the chart and that for the host SB on the right half.

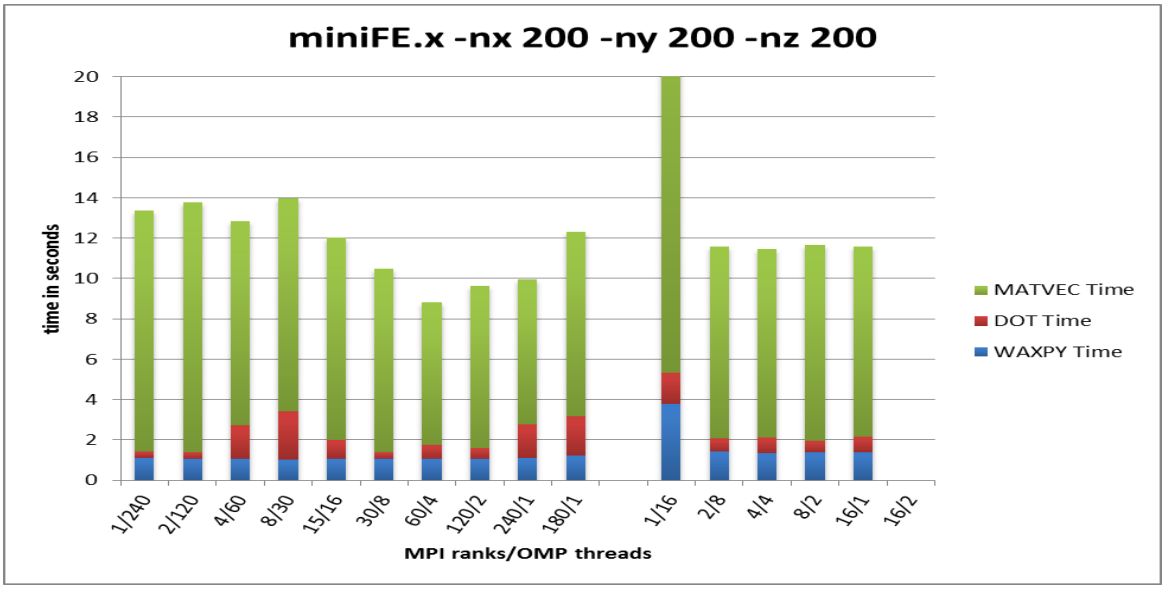

Figure 6 miniFE performance on KNC (left) and SB(right)

The best performance on KNC for the most time consuming kernel, MATVEC, was with 60 MPI tasks and 4 threads per MPI task. This was 1.3X faster than the best performance on the host with 16 MPI tasks and 1 task per thread. Interestingly the data shows that for the DOT compute kernel the optimal performance on KNC requires a different MPI tasks / OpenMP threads combination: 30/8.

The key to optimal usage of the MIC architectures is to efficiently use the 60/61 cores and the four hardware threads each core provides. Efficient usage could be viewed as achieving good thread level parallelism using the fewest MPI tasks on the processor. This is to permit future efficient multi-node scaling as a consequence of fewer MPI task on each node leading to better utilization of memory and smaller volume of inter-node message exchanges. Efficient usage also entails high level of vectorization of the compute loop kernels and full utilization of the excellent memory bandwidth Xeon Phi provides.

6.1.2 Vampir Profile to gauge coarse and fine grain parallelization effectiveness

Analysis of an application as to its efficient mapping on to an MPI + OpenMP programming model requires in addition to the scaling characteristics shown in Figure 6, an application function profile giving us an understanding of the fraction of time spent by the application in OpenMP compute loops, serial compute kernels, time spent in OpenMP overheads such as locks/barriers, time spent in MPI, time spent in MPI synchronizations. Towards this goal it is useful to obtain a profile of the application that reveals these wall-time components. The profile may be strongly influenced by scale and input. The input data set used for this analysis is the Trinity/NERSC8 "single-node" benchmark as described at [5].

For miniFE this data was gathered using, 9 nodes, 72 MPI tasks and 2 OMP threads per task on Sandia's TLCC2 system called Chama which has twin socket, 8-cores/socket Sandy Bridge nodes with Qlogic QDR InfiniBand Interconnect. ScoreP and Vampir are the profiling tools of choice to gather the desired information. Vampir API to bracket only code section that is of interest, namely conjugate gradient solver in miniFE is used to generate the plot shown in Figure 7.

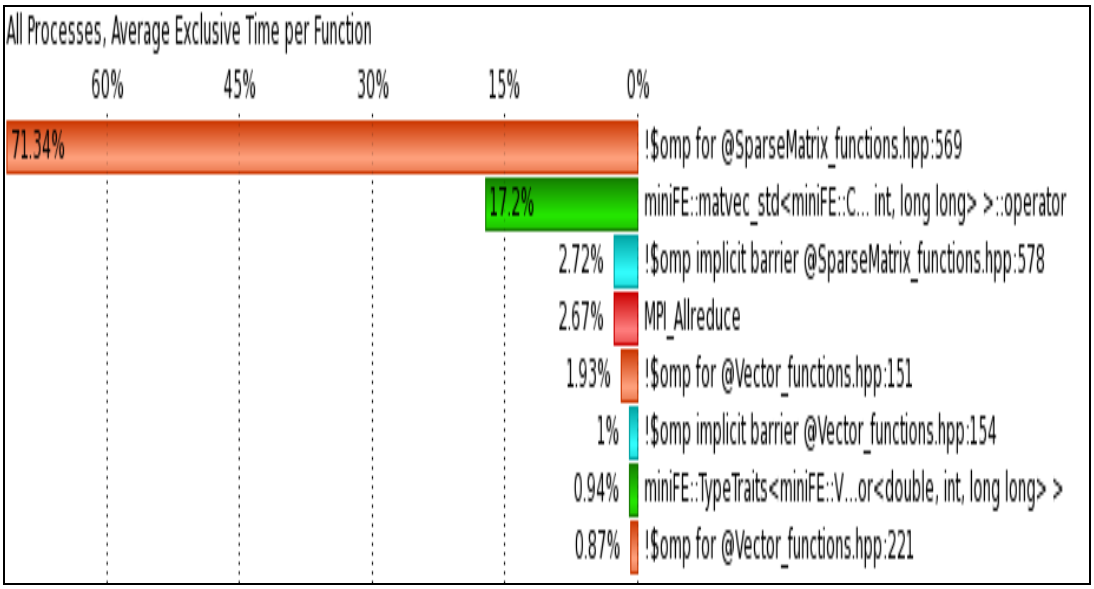

Figure 7 miniFE profile showing percentage run time fractions

From the profile it is clear why this application performs well with the hybrid programming model. The large fraction of the run time (> 72%) that is registered for an OpenMP construct and small fraction of the time in MPI gives this application the performance noted above.

6.1.3 Vectorization effectiveness

Another key factor in the efficient use of the Xeon Phi architecture is vectorization. As suggested in their book on Xeon Phi by Jeffers and Reinders [6], one approach to gauge effective vectorization is to compare run times of the application with and without auto vectorization by the compiler. Using the same input used for gathering the comparative performance between SB and KNC shown in Figure 6, picking the run with MPI tasks and OpenMP threads that yielded the best performance, the performance gain with compiler generated auto-vectorization for the host SB and the MIC measured on the Corner workstation is shown in Table 1.

Table 1 miniFE compiler auto vectorization performance for SB and KNC

Processor	Total CG time (secs) with Auto Vectorization	Total CG time (secs) with No Auto Vectorization
Intel MIC (60 MPI tasks/4 threads)	7.941	8.313
Intel Sandy Bridge(8 MPI tasks/4threads)	12.116	12.3219

Use of compiler vectorization report (with flag –vec-report3) shows that the inner loop in the most time consuming kernel, matrix-vector product, in SparseMatrix_functions.hpp, line 573 is reported as vectorized. However this vectorization, reported by the compiler, does not register as a significant performance gain for run time comparisons with and without vectorization. This needs to be further investigated, but suspected to be related to the indirect addressing required in the operation in line 574: sum += Acoefs[i]*xcoefs[Acols[i]]. Most of the small gain in run time seen in Table 1 comes from the vectorization of the other two key kernel computations: waxpby() and dot().

Another measure of vectorization that we wish to gain further experience is with the use hardware performance counters. In reference [2], Shannon Cepda presents various metrics using hardware events counts from the processing core's Performance Monitoring Unit (PMU). PMU hardware counters can be programmed to count occurrences of various events. Intel VTune provides developers the ability to collect and view sampled data from the Xeon Phi. Recently we have installed the TAU performance monitoring tool on Compton. TAU can also provide access to the PMU through PAPI. Of particular interest in investigating effective vectorization are two counters: VPU_INSTRUCTIONS_EXECUTED and VPU_ELEMENTS_ACTIVE. The number of vector elements active given by the second counter above is a measure of the number of vector operations and provides correct estimates of the multiple vector operations in both single and double precision for each instruction executed. So the metric ratio VPU_ELEMENTS_ACTIVE/VPU_INSTRUCTION_EXECUTED, called vectorization intensity is very useful in gauging how well the compute intensive sections of the code are vectorized. For double precision vectors optimal vectorization is achieved if vectorization intensity is close to 8 and for single precision close to 16.

MiniFE was instrumented with TAU with the following build script which sets the necessary environment variable and replaces the mpiicxx in the makefile with tau_cxx.sh.

```
#!/bin/bash
source /projects/tau/tau.bashrc
export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt-openmp-opari
export TAU_OPTIONS='-optVerbose'
make CXX=tau_cxx.sh
```

The instrumented executable produced a run profile on Compton. However the profile did not readily provide the desired information for the most time consuming sparse Matrix-Vector operations in the conjugate gradient solve. To focus on the section of code of interest, the TAU API for selective instrumentation was used to introduce in cg_solve.hpp calls to TAU_PROFILER_START and TAU_PROFILER_STOP functions bracketing the function call to matvec(). Execution of the instrumented miniFE on the MIC using similar parameters to that used for the data in Figure 4, gave the vectorization intensity of: 7.634e08/4.363e08 = 1.75. Since this value is not close to 8, the previous conclusion on the need to improve vectorization for the compute intensive MATVEC kernel is reinforced.

6.2 AMG

AMG is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids.

6.2.1 Hybrid (OpenMP+MPI) code performance

The code uses OpenMP pragmas to invoke OpenMP threads for the Hypre library GMRES solver kernel operations. For the Laplace solver benchmarked, the dominant GMRES kernel OpenMP operations (called by hypre_GMRESSolve) are in the source code files in the directory src/seq_mv, in the files csr_matvec.c (line numbers 344,317,215,134,126) and vector.c (line numbers 445, 419, 391, 320, 265.

The code was compiled with Intel C++ compiler version: 'icpc (ICC) 13.0.1 20121010' and with compiler flags '-mmic -O3 -openmp' for running on the Xeon Phi in the native mode. The same compiler was used for generating the executable that was run on the host with the same compiler flags except '–mmic' option. The process/thread affinity settings used KMP_AFFINITY=compact, granularity=fine and I_MPI_PIN_DOMAIN=omp.

The optimal combination of MPI tasks and OpenMP threads that gives the best performance was investigated with this benchmark. Taking the smallest run time among three trials, Figure 8 shows the GMRES solve wall clock time. The data for KNC shown on the left half of the chart and that for the host SB on the right half.

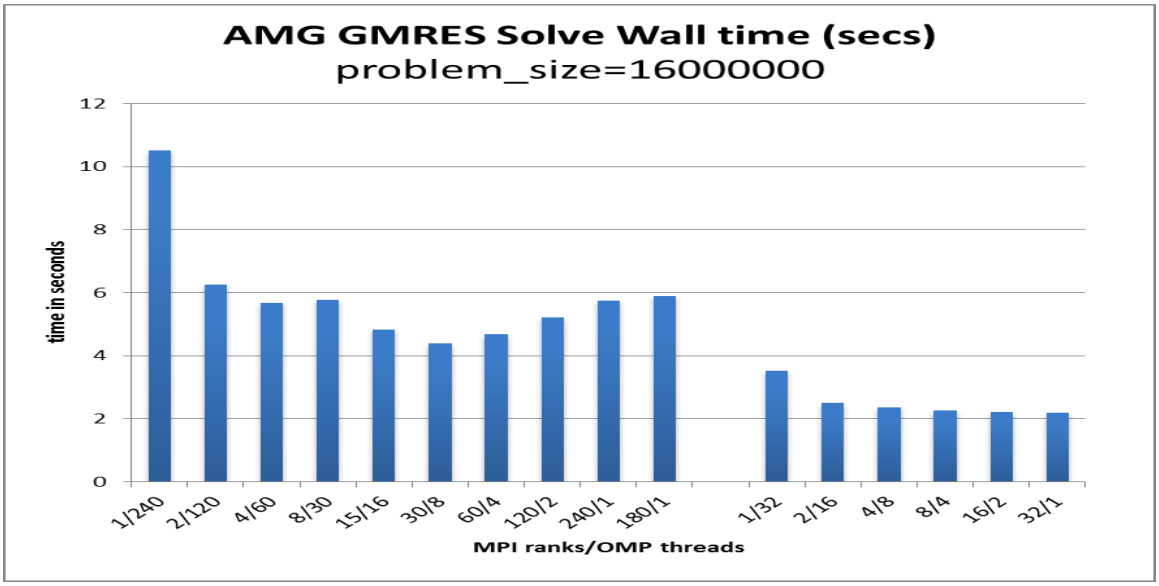

Figure 8 AMG performance on KNC (left) and SB (right)

The best performance on KNC was with 30 MPI tasks and 8 threads per MPI task and in the SB 32 and 1. The host SB performance is 2X faster than KNC.

6.2.2 Vampir Profile to gauge coarse and fine grain parallelization effectiveness

A profile of AMG is needed to reveal various compute time components. The profile may be strongly influenced by scale and input. The input data set used for this analysis is the Trinity/NERSC8 "single-node" benchmark as described at [5]. For AMG this data was gathered using, 6 nodes, 48 MPI tasks and 2 OpenMP threads per task on Sandia's TLCC2 system called Chama with an mpiexec command as shown below:

mpiexec –n 48 –npersocket 4 –bind-to-core ./amg2013 –P 4 4 3 –n 189 189 189 –solver 2

ScoreP and Vampir are the tools used to gather the desired trace information. ScoreP API is used to bracket only code section that is of interest. SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION_END bracket the call to HYPRE_GMRESSolve function in the file amg2013.c Figure 9. Shows the function profile as percentage of the run time.

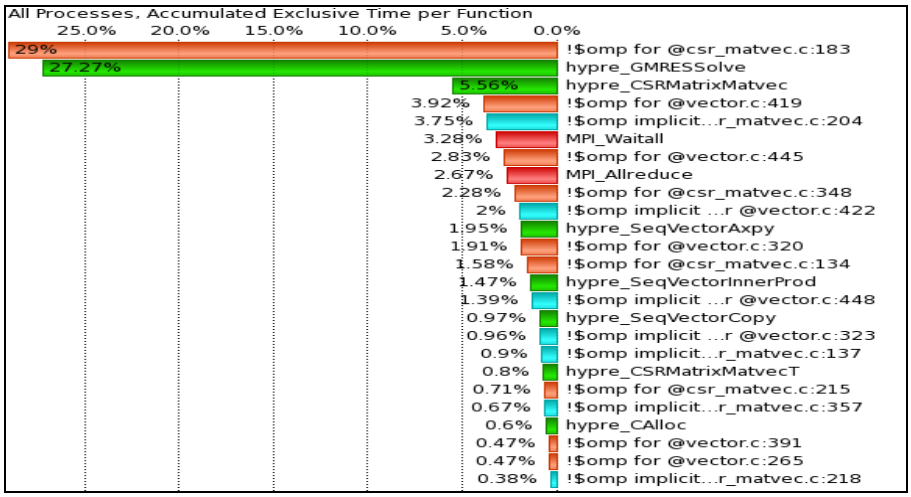

Figure 9 AMG profile showing percentage run time fractions

From the run time fraction percentage we see that while there is substantial fraction of the run time in OpenMP 'for' loops it is not as high as we saw with miniFE and consequently we see small gains in run time when using MPI_tasks + OpenMP threads as opposed to only MPI tasks using the same number of cores in a given run. We also see from the profile that the fraction of the run time spent in MPI is less than 6% which helps with using large number of MPI tasks on the Xeon Phi.

6.2.3 Vectorization effectiveness

Using the same input used for gathering the comparative performance between SB and KNC shown in Figure 8, picking the run with MPI tasks and OpenMP threads that yielded the best performance, the performance gain with compiler generated auto-vectorization for the host SB and the MIC measured on the Corner workstation is shown in Table 2.

Table 2 AMG compiler auto vectorization performance for SB and KNC

Processor	Total CG time (secs) with Auto Vectorization	Total CG time (secs) with No Auto Vectorization
Intel MIC (30 MPI tasks/8 threads)	3.342	3.124
Intel Sandy Bridge(16 MPI tasks/2 threads)	2.210	2.219

Use of –vec-report3 compiler flag shows that the Intel icc compiler was unable to vectorize any of the loops in the most compute intensive function as per the profile shown in Figure 9, namely the functions in csr_matevc.c. From Table 2, We see that both on SB and KNC very little gain in performance with vectorization is measured. The repeatable (using 3 measurements not recorded here) slightly better performance without vectorization on the MIC is not fully understood. We could measure the vectorization intensity with PMU counters as shown in the section on miniFE, but it may not add to our analysis of this application.

6.3 UMT

The UMT benchmark is a 3D, deterministic, multigroup, photon transport code for unstructured meshes.

6.3.1 Hybrid (OpenMP+MPI) code performance

The optimal combination of MPI tasks and OpenMP threads that gives the best performance was investigated with this benchmark. Taking the smallest run time among three trials, Figure 10 and Figure 11 shows the two metrics of interest: CumulativeWork Time and AngleLoop Time. The data for KNC shown on the left half of the chart and that for the host SB on the right half.

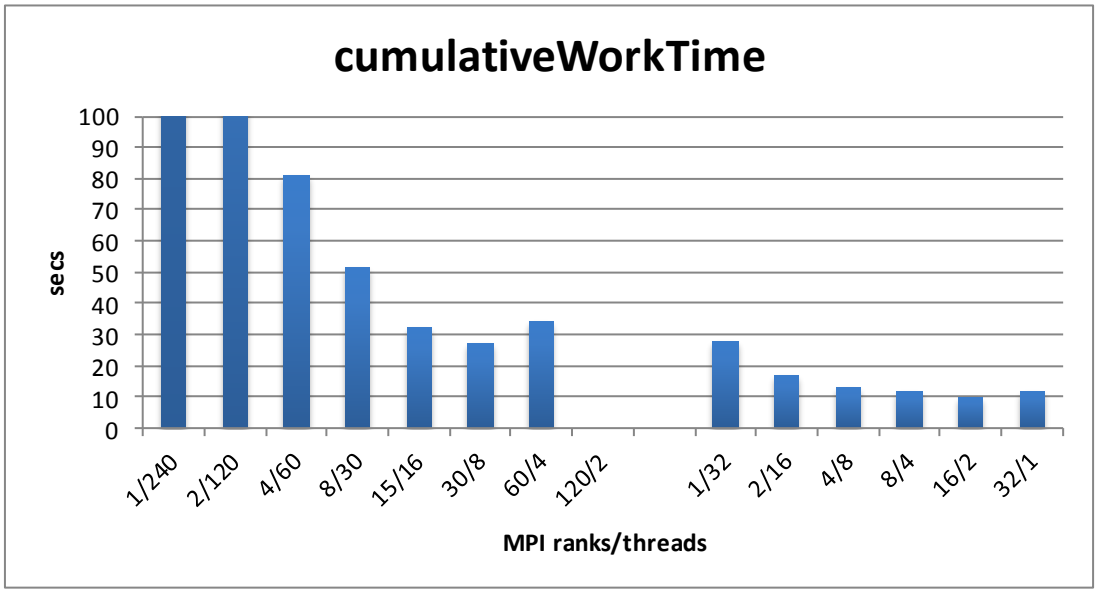

Figure 10 UMT performance (Cumulative Work Time) on KNC (left) and SB (right)

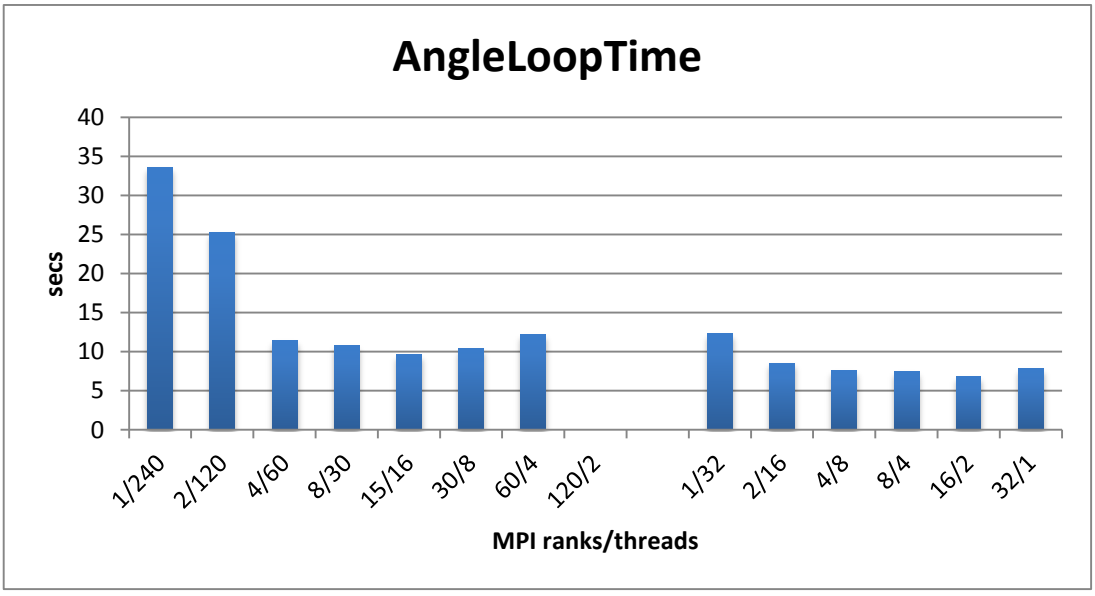

Figure 11 UMT performance (AngleLoop Time) on KNC (left) and SB (right)

6.3.2 Vampir Profile to gauge coarse and fine grain parallelization effectiveness

Initial attempts on Chama to collect this profile with Vampir/ScoreP ran into some link time errors. This will have to be pursued after either building a version of ScoreP that generates dynamic libraries or a setup on Chama that uses GNU compilers. The data from Figure 10, 11 suggest that fine grain parallelism with OpenMP is quite effective resulting in improved performance with up to 16 or 8 OpenMP threads on the MIC.

6.3.3 Vectorization effectiveness

Using the same inputs as used for the runs in Figure 11 and picking the combinations of MPI tasks and OpenMP threads that lead to the best performance the impact of vectorization on the MIC and on the host SB node was investigated. The data recorded for the AngleLoop Time and the Cumulative Work time are shown in Table 3.

Table 3 UMT compiler auto vectorization performance for SB and KNC

processor	AngleLoop time/Cum.Work time (secs) with Auto Vectorization	AngleLoop time/Cum.Work time (secs) with No Auto Vectorization
Intel MIC (30 MPI tasks/8 threads)	8.10/23.61	11.98/27.09
Intel Sandy Bridge(16 MPI tasks/2 threads) on Chama	6.68/9.88	8.35/11.52

Vectorization gives about 20% better performance for the SB and 32.4% for the KNC for the Angle Loop Time metric, which is of more interest in this application.

6.4 GTC

GTC is used for Gyrokinetic Particle Simulation of Turbulent Transport in Burning Plasmas. It is a fully self-consistent, 3D Particle-in-cell code (PIC) with a non-spectral Poisson solver and a grid that follows the magnetic field lines (twisting around the torus). It solves the gyro-averaged Vlasov equation in real space; the Vlasov equation describes the evolution of a system of particles under the effects of self-consistent electromagnetic fields. The unknown is the flux, f(t,x,v), which is a function of time t , position x, and velocity v, and represents the distribution function of particles (electrons and ions) in phase space.

6.4.1 Hybrid (OpenMP+MPI) code performance

Among all the applications studied here, GTC is best set up to use OpenMP for thread parallelization of many computationally intensive loops. The compiler vector report indicates several of the loops in key functions like pushi, chargei get vectorized.

The code was compiled with Intel Fortran compiler ifort with compiler flags '-mmic -O3 -openmp' for running on the Xeon Phi in the native mode. The same compiler was used for generating the executable that was run on the host with the same compiler flags except '–mmic' option. The process/thread affinity settings used are: KMP_AFFINITY=compact,granularity=fine and I_MPI_PIN_DOMAIN=omp.

The optimal combination of MPI tasks and OpenMP threads that gives the best performance was investigated with this benchmark. Taking the smallest run time among three trials, Figure 12 shows the NERSC Time used as the metric for this benchmark. The data for KNC shown on the left half of the chart and that for the host SB on the right half. Some combination of MPI tasks and OpenMP threads (1/240, 2/120, 4/60) on the KNC and (1/32, 2/16) on SB SEGFAULTED, but possible causes for this have not been investigated.

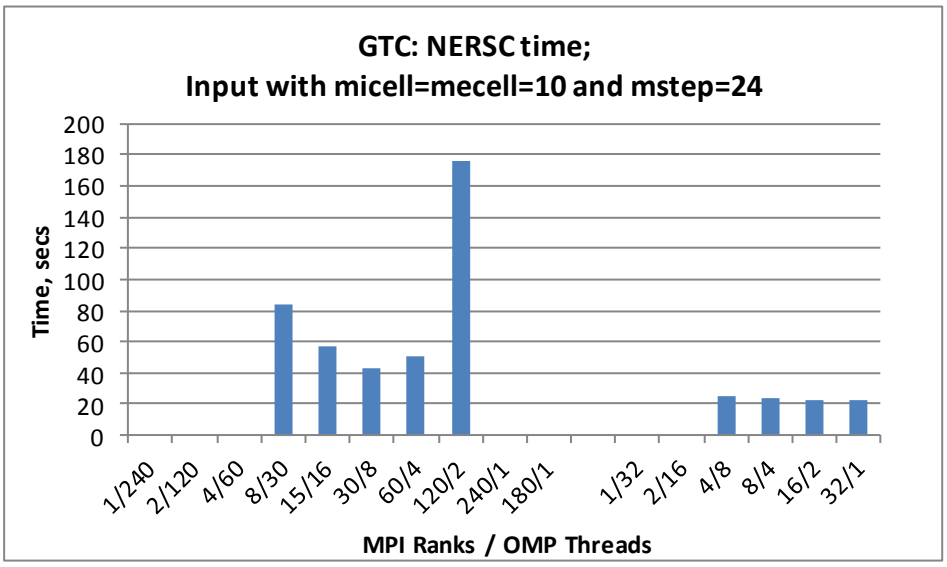

Figure 12 GTC performance (NERSC Time) on KNC (left) and SB (right)

6.4.2 Vampir Profile to gauge coarse and fine grain parallelization effectiveness

A profile ofGTC is needed to reveal various compute time components. The profile may be strongly influenced by scale and input. The input data set used for this analysis is the Trinity/NERSC8 "single-node" benchmark as described at [5]. For GTC this data was gathered using, 8 nodes, 64 MPI tasks and 2 OpenMP threads per task on Sandia's TLCC2 system called Chama with an mpiexec command as shown below:

mpiexec –n 64 –npersocket 4 –bind-to-core ./gtcomp

ScoreP and Vampir are the tools used to gather the desired trace information. For GTC the entire code was instrumented with ScoreP. Figure 13 shows the function profile as percentage of the run time.

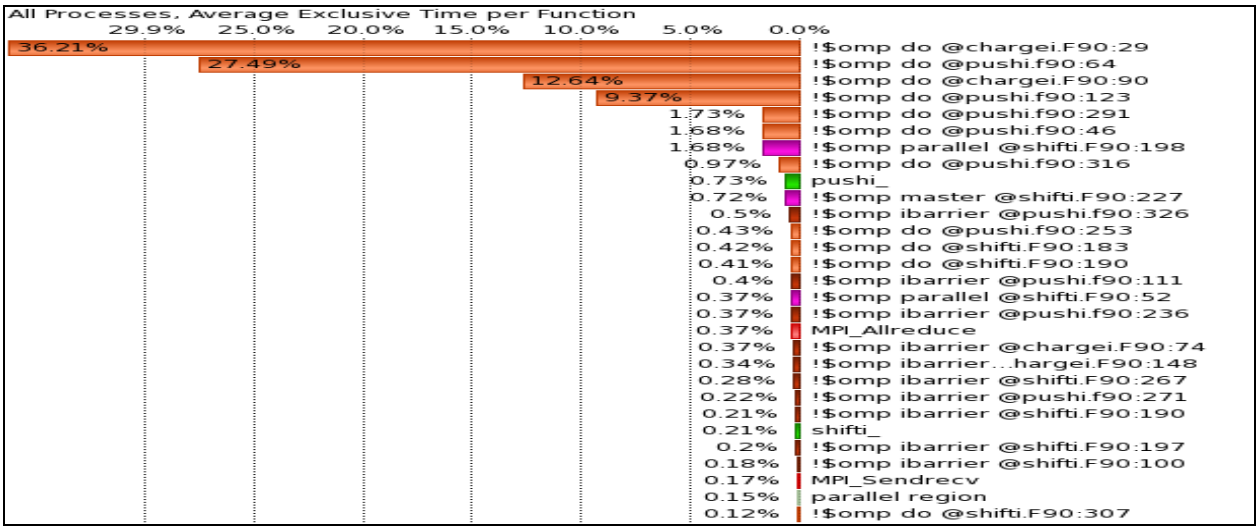

Figure 13 GTC profile showing percentage run time fractions

From the run time fraction percentage we see that substantial fraction of the run time is in OpenMP 'for' loops. On the MIC 30 MPI_tasks with 8 OpenMP threads gives a run time 80% longer than the best seen on the host SB. This application demonstrated important performance elements for MIC, like good thread level parallelism, vectorization effectiveness and low MPI overhead.

6.4.3 Vectorization effectiveness

Using the same input used for gathering the comparative performance between SB and KNC shown in Figure 12, picking the run with MPI tasks and OpenMP threads that yielded the best performance, the performance gain with compiler generated auto-vectorization for the host SB and the MIC measured on the Corner workstation is shown in Table 4.

Table 4 GTC compiler auto vectorization performance for SB and KNC

Processor	NERSC time (secs) with Auto Vectorization	NERSC time (secs) with No Auto Vectorization
Intel MIC (30 MPI tasks/8 threads)	39.735	46.869
Intel Sandy Bridge(16 MPI tasks/2 threads)	21.969	30.677

Use of –vec-report3 compiler flag shows that the Intel ifort compiler was able to vectorize the key compute loops in chargei, pushi and shifti functions From Table 4. We see that both on SB 18% gain and on KNC about 39% gain in performance with vectorization is measured. We could measure the vectorization intensity with PMU counters as shown in the section on miniFE for the compute intensive chargei and pushi functions. Initial attempts with TAU on Compton led to run time failures only when the PAPI counters were turned on. This needs to be further discussed with the TAU developers. We should also try VTune with the same PMU counters.

6.5 MILC

The benchmark code MILC represents part of a set of codes written by the MIMD Lattice Computation (MILC) collaboration used to study quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics. It performs simulations of four dimensional SU(3) lattice gauge theory on MIMD parallel machines. "Strong interactions" are responsible for binding quarks into protons and neutrons and holding them all together in the atomic nucleus. The MILC collaboration has produced application codes to study several different QCD research areas, only one of which, ks_dynamical simulations with conventional dynamical Kogut-Susskind quarks, is used here. QCD discretizes space and evaluates field variables on sites and links of a regular hypercube lattice in four-dimensional space time. Each link between nearest neighbors in this lattice is associated with a 3-dimensional SU(3) complex matrix for a given field. The version of MILC used here uses matrices ranging in size from 8^4 to 128^4.

6.5.1 Hybrid (OpenMP+MPI) code performance

As per the README file provided with this benchmark, OpenMP directives currently exist only in source code in the generic_ks directory (specifically, in the files d_congrad5_fn.c and dslash_fn2.c). Since these two functions did not appear to consume significant fraction of the run time this benchmark is not well suited to investigate impact of a hybrid programming model. Also noted in the README file, the inlined SSE instructions available in MILC have been disabled as they have been observed to not always work between different compilers. So this benchmark as set up is not suited for investigating vectorization.

The code was compiled with Intel C compiler icc with compiler flags '-mmic -O3 -openmp' for running on the Xeon Phi in the native mode. The same compiler was used for generating the executable that was run on the host with the same compiler flags except the '–mmic' option. The process/thread affinity settings used KMP_AFFINITY=compact,granularity=fine and I_MPI_PIN_DOMAIN=omp.

The optimal combination of MPI tasks and OpenMP threads that gives the best performance was investigated with this benchmark. Taking the smallest run time among three trials, Figure 14 shows the NERSC Time used as the metric for this benchmark. The data for KNC shown on the left half of the chart and that for the host SB on the right half. Some combination of MPI tasks and OpenMP threads (15/16, 30/8, 60/4) on the KNC produced an error message: "Can't layout lattice, not enough factors of 5". Possible ways to go past this hurdle with modifications to the input file was not pursued.

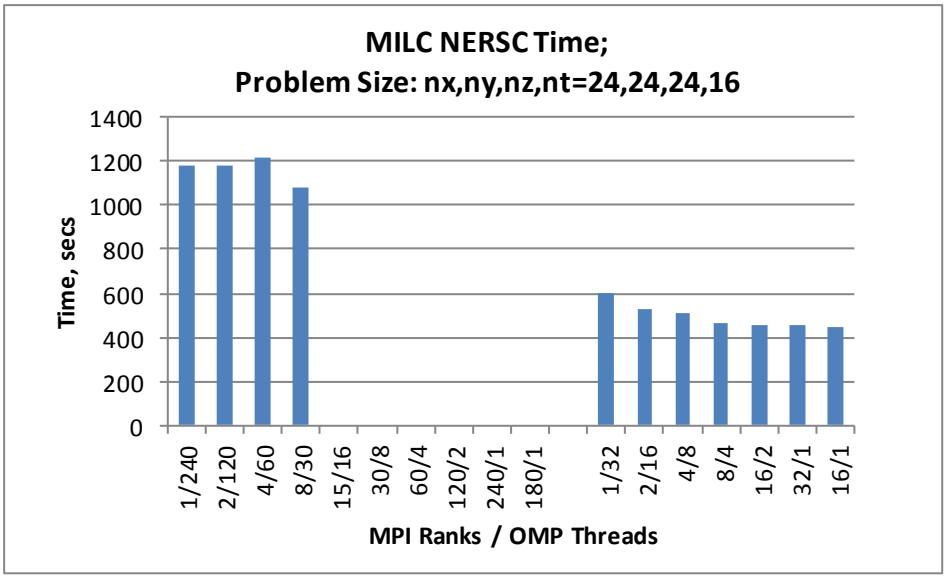

Figure 14 MILC performance (NERSC Time) on KNC (left) and SB (right)

6.5.2 Vampir Profile to gauge coarse and fine grain parallelization effectiveness

A profile of MILC is needed to reveal various compute time components. The profile may be strongly influenced by scale and input. The input data set used for this analysis is the Trinity/NERSC8 "single-node" benchmark as described at [5]. For MILC this data was gathered using, 4 nodes, 24 MPI tasks and 2 OpenMP threads per task on Sandia's TLCC2 system called Chama with an mpiexec command as shown below:

mpiexec –n 64 –npersocket 3 –bind-to-core ./su3_rmd < n8_single.in

ScoreP and Vampir are the tools used to gather the desired trace information. For MILC the code was instrumented using a scoreP filter file to limit the size of the trace file. In the filter file all the regions were excluded from tracing, except functions: MPI, OMP, update, main, update_h, f_meas_imp and load_ferm_link. Figure 15 shows the function profile as percentage of the run time.

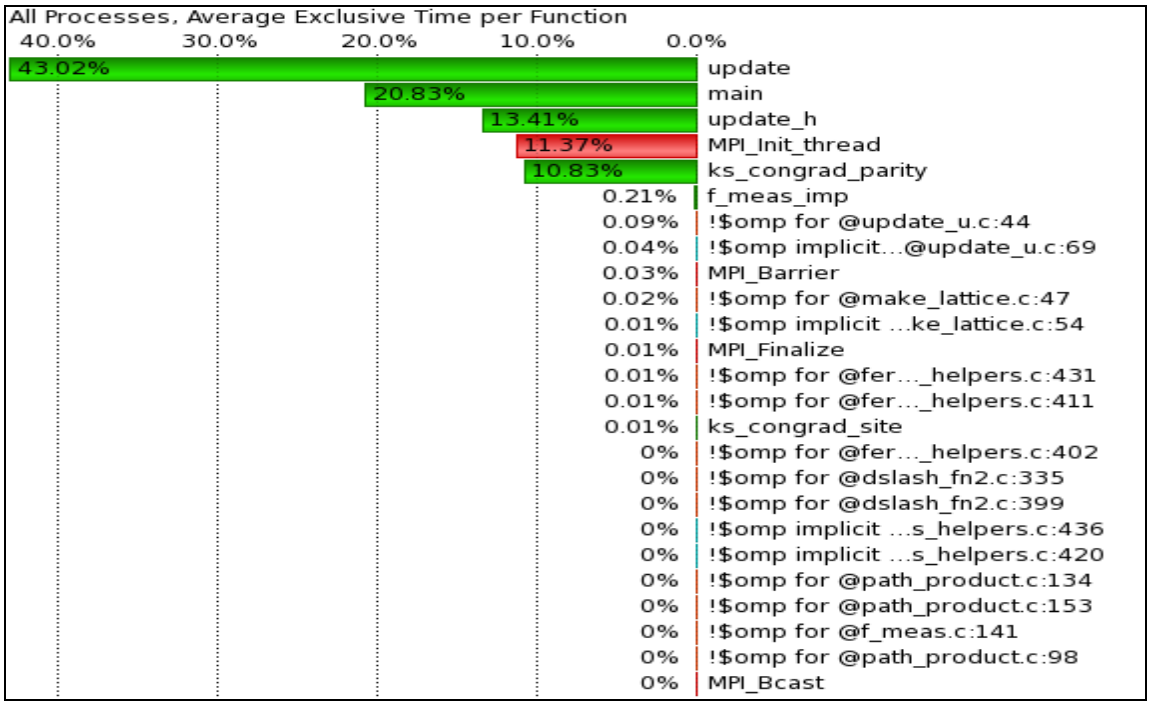

Figure 15 MILC profile showing percentage run time fractions

From the run time fraction percentage we see that only a very small fraction of the run time is in OpenMP 'for' loops. We need to investigate if the version of MILC provided as part of the Trinity benchmarks is a version that has the latest development in introducing OpenMP constructs. On the MIC, 8 MPI_tasks with 30 OpenMP threads gives a run time 2.6X the best seen on the host SB. MILC because of its importance to physicists and because it consumes large node-hours on number of NSF/University/DOE ASCR systems has a long history of developments and performance enhancements. Further investigation of the port of MILC to MIC should be pursued in collaboration with the domain scientists with deep knowledge of this application.

6.6 SNAP

SNAP is a proxy application to model the performance of a modern discrete ordinates neutral particle transport application. SNAP may be considered an update to Sweep3D, intended for hybrid computing architectures. It is modeled on the Los Alamos National Laboratory code PARTISn. PARTISn solves the linear Boltzmann transport equation (TE), a governing equation for determining the number of neutral particles (e.g., neutrons and gamma rays) in a multidimensional phase space. SNAP itself is not a particle transport application; SNAP incorporates no actual physics in its available data, nor does it use numerical operators specifically designed for particle transport. Rather, SNAP mimics the computational workload, memory requirements, and communication patterns of PARTISn. The equation it solves has been composed to use the same number of operations, use the same data layout, and load elements of the arrays in approximately the same order. Although the equation SNAP solves looks similar to the TE, it has no real world relevance.

6.6.1 Hybrid (OpenMP+MPI) code performance

The solution to the time-dependent TE is a "flux" function of seven independent variables: three spatial (3-D spatial mesh), two angular (set of discrete ordinates, directions in which particles travel), one energy (particle speeds binned into "groups"), and one temporal. PARTISN, and therefore SNAP, uses domain decomposition over these dimensions to coherently distribute the data and the tasks associated with solving the equation. The parallelization strategy is expected to be the most efficient compromise between computing resources and the iterative strategy necessary to converge the flux.

The iterative strategy is comprised of a set of two nested loops. These nested loops are performed for each step of a time-dependent calculation, wherein any particular time step requires information from the preceding one. No parallelization is performed over the temporal domain. However, for time-dependent calculations two copies of the unknown flux must be stored, each copy an array of the six remaining dimensions. The outer iterative loop involves solving for the flux over the energy domain with updated information about coupling among the energy groups. Typical calculations require tens to hundreds of groups, making the energy domain suitable for threading with the nodes' provided accelerator. The inner loop involves sweeping across the entire spatial mesh along each discrete direction of the angular domain. The spatial mesh may be immensely large. Therefore, SNAP spatially decomposes the problem across nodes and communicates needed information according to the KBA method . KBA is a transport-specific application of general parallel wavefront methods. Lastly, although KBA efficiency is improved by pipelining operations according to the angle, current chipsets operate best with vectorized operations. During a mesh sweep, SNAP operations are vectorized over angles to take advantage of the modern hardware.

The code was compiled with Intel Fortran compiler ifort with compiler flags '-mmic -O3 -openmp' for running on the Xeon Phi in the native mode. The same compiler was used for generating the executable that was run on the host with the same compiler flags except '–mmic' option. The process/thread affinity settings used are: KMP_AFFINITY=compact,granularity=fine and I_MPI_PIN_DOMAIN=omp.

The optimal combination of MPI tasks and OpenMP threads that gives the best performance was investigated with this benchmark. Taking the smallest run time among three trials, Figure 16 shows the Solve Time used as the metric for this benchmark. The data for KNC shown on the left half of the chart and that for the host SB on the right half.

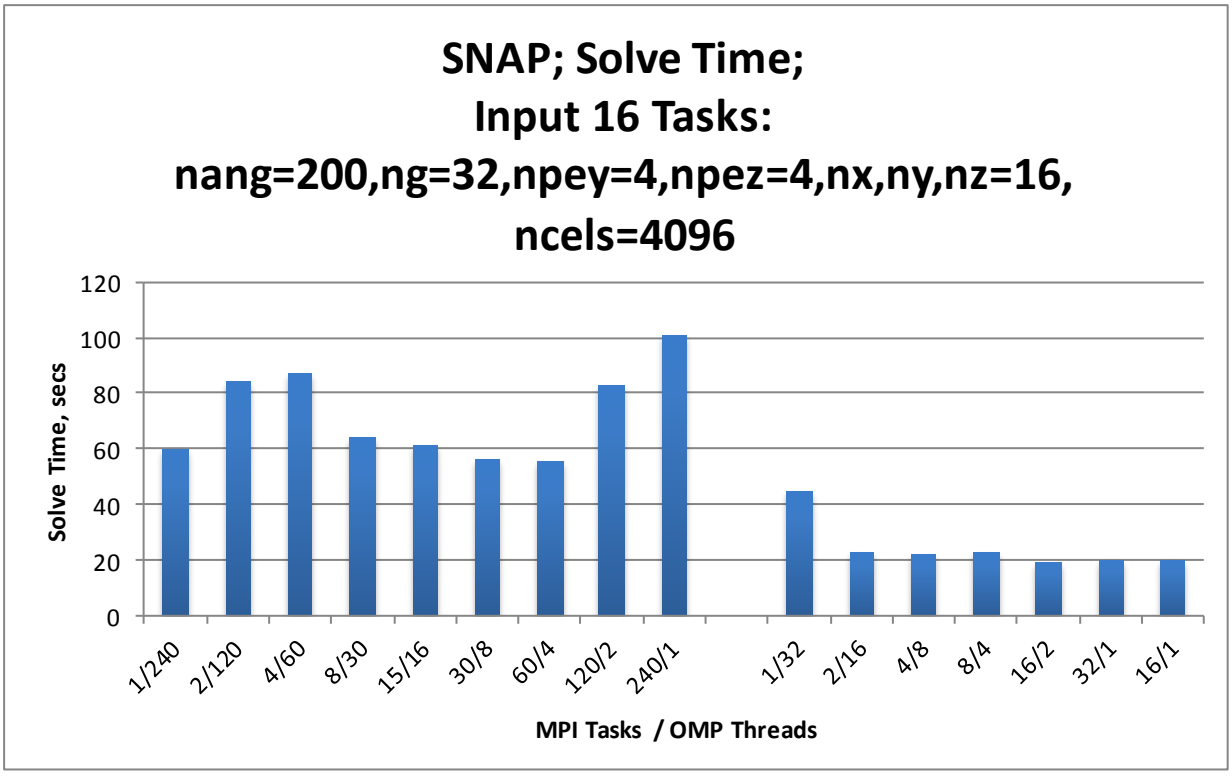

Figure 16 SNAP performance (Solve Time) on KNC (left) and SB (right)

6.6.2 Vampir Profile to gauge coarse and fine grain parallelization effectiveness

A profile of SNAP is needed to reveal various compute time components. The profile may be strongly influenced by scale and input. The input data set used for this analysis is the Trinity/NERSC8 "single-node" benchmark as described at [5]. For SNAP this data was gathered using, 12 nodes, 48 MPI tasks and 4 OpenMP threads per task on Sandia's TLCC2 system called Chama with an mpiexec command as shown below:

mpiexec –loadbalance –n 48 ./snap ./small-4nodes-input ./small-4nodes.output

ScoreP and Vampir are the tools used to gather the desired trace information. For SNAP the entire code was instrumented with ScoreP. Figure 17 shows the function profile as percentage of the run time.

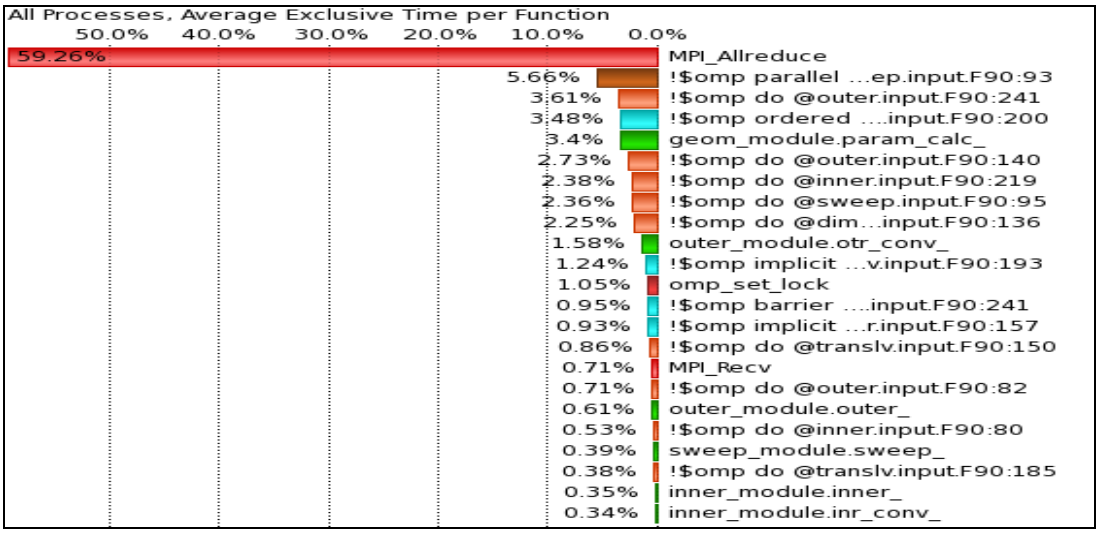

Figure 17 SNAP profile showing percentage run time fractions

For this particular analysis/input and run on Chama the large fraction of time spent in Allreduce suggests that this benchmark may need careful study before it can run efficiently on the MIC. The profile does show good use of OpenMP thread level parallelization in several functions.

6.6.3 Vectorization effectiveness

Using the same input used for gathering the comparative performance between SB and KNC shown in Figure 15, picking the run with MPI tasks and OpenMP threads that yielded the best performance, the performance gain with compiler generated auto-vectorization for the host SB and the MIC measured on the Corner workstation is shown in Table 5. We see a 19.5% improvement with vectorization on the MIC and a 13.2% improvement on the host SB.

Table 5 SNAP compiler auto vectorization performance for SB and KNC

Processor	Solve time (secs) with Auto Vectorization	Solve time (secs) with No Auto Vectorization
Intel MIC (60 MPI tasks/4 threads)	3.4377e01	4.1082e01
Intel Sandy Bridge(16 MPI tasks/2 threads)	1.5965e01	1.8086e01

We measured performance gain with vectorization of 18% on SB and about 39% on KNC. We could measure the vectorization intensity with PMU counters as shown in the section on miniFE for the compute intensive chargei and pushi functions. Initial attempts with TAU on Compton led to run time failures only when the PAPI counters were turned on. This needs to be further discussed with the TAU developers.

6.7 miniDFT

MiniDFT is a plane-wave density functional theory (DFT) mini-app for modeling materials. Given a set of atomic coordinates and pseudopotentials, MiniDFT computes self-consistent

solutions of the Kohn-Sham equations using either the LDA or PBE exchange-correlation functionals. For each iteration of the self-consistent field cycle, the Fock matrix is constructed and then diagonalized. To build the Fock matrix, Fast Fourier Transforms are used to transform orbitals from the plane wave basis (where the kinetic energy is most readily competed) to real space (where the potential is evaluated) and back. Davidson diagonalization is used to compute the orbital energies and update the orbital coefficients.

6.7.1 Hybrid (OpenMP+MPI) code performance

The code was compiled with Intel Fortran compiler ifort with compiler flags '-mmic -O3 -openmp' for running on the Xeon Phi in the native mode. The same compiler was used for generating the executable that was run on the host with the same compiler flags except '–mmic' option. The process/thread affinity settings used are: KMP_AFFINITY=compact,granularity=fine and I_MPI_PIN_DOMAIN=omp.

A special input file called Mg0442.in was constructed after discussions with the author of miniDFT at NERSC. This was because the input files provided with the trinity benchmark could not be easily modified to permit runs within the 8GB GDDR5 on the MIC. It is also not quite straight forward to construct weak-scaling-study inputs, as the computational complexity of the key compute kernels (FFT, solver) has non-linear dependence on key input parameters. With the Mg0442.in as input, the optimal combination of MPI tasks and OpenMP threads that gives the best performance was investigated with this benchmark. Taking the smallest run time among three trials, Figure 17 shows the Benchmark_Time reported on output as the metric for this benchmark. The data for KNC shown on the left half of the chart and that for the host SB on the right half. Some combination of MPI tasks and OpenMP threads (1/240, 2/120, 4/60, 8/30,15/16,30/8) on the MIC led to run time failures with the MKL Cholesky solver aborting the run. This needs to be investigated with the Intel development team.

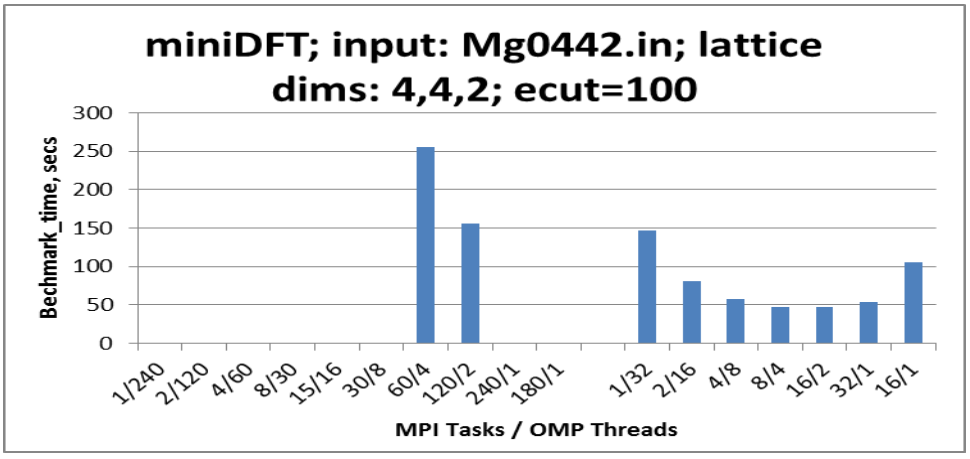

Figure 18 miniDFT performance (Benchmark Wall time) on KNC (left) and SB (right)

6.7.2 Vampir Profile to gauge coarse and fine grain parallelization effectiveness

A profile of miniDFT is needed to reveal various compute time components. The profile may be strongly influenced by scale and input. The input data set used for this analysis is the

Trinity/NERSC8 “single-node” benchmark with input *titania_3_120.in* as described at [5]. For miniDFT this data was gathered using, 3 nodes, 24 MPI tasks and 2 OpenMP threads per task on Sandia’s TLCC2 system called Chama with an mpiexec command as shown below:

mpiexec –n 24 –npersocket 4 –npernode 8 ./mini_dft –in titania_3_120.in

ScoreP and Vampir are the tools used to gather the desired trace information. For miniDFT the entire code was instrumented with ScoreP. Figure 19 shows the function profile as percentage of the run time. From the figure it is clear that OpenMP paralleized loops constitute a small fraction of the run time for miniDFT. However as miniDFT calls MKL math kernels like ZGEMM and FFT it takes advantage of fine grain parallelism in MKL.

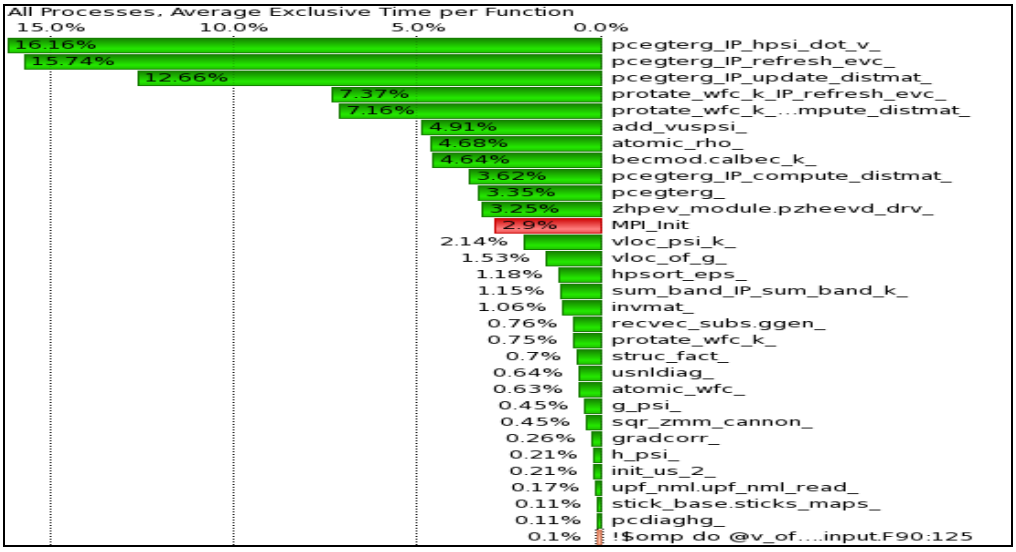

Figure 19 miniDFT profile showing percentage run time fractions

6.7.3 Vectorization effectiveness

Using the same input used for gathering the comparative performance between SB and KNC shown in Figure 18, picking the run with MPI tasks and OpenMP threads that yielded the best performance, the performance gain with compiler generated auto-vectorization for the host SB was measured on the Corner workstation and is shown in Table 6. We see a small 2% improvement with vectorization on the host SB node. However this result is misleading in view of the percentage of the peak FLOPS achieved in this benchmark that is discussed in the following section 7. miniDFT computations are dominated by highly tuned library functions like matrix multiply (ZGEMM) and FFT. This approach of gauging vectorization by comparing performance with and without the compiler flag “-no-vec” affects only loops in the source code that get vectorized and therefore highly optimized and vectorized library routines, which dominate miniDFT are unaffected by this compiler flag.

Table 6 miniDFT compiler auto vectorization performance for SB and KNC

Processor	Benchmark Wall time (secs) with Auto Vectorization	Benchmark Wall time (secs) with No Auto Vectorization
Intel Sandy Bridge(8MPI	48.19	49.21

tasks/4 threads)		

6.8 NPB

NPB is used as a sanity check to gain confidence in understanding hybrid code performance and tools for analysis. NPB BT-MZ solves a discretized version of unsteady, compressible Navier-Stokes equations in three spatial dimensions. BT (Block Tri-diagonal) solves three sets of uncoupled systems of equations, first in the X dimension, then in the Y dimension, and finally in the Z dimension; these systems are block tri-diagonal with 5x5 blocks. The benchmark performs 200 time steps on a regular 3 dimensional grid. The code is implemented in 20 or so Fortran77 source modules. Multi-zone versions of NPB (NPB-MZ) are designed to exploit multiple levels of parallelism in applications and to test the effectiveness of multi-level and hybrid parallelization paradigms and tools.

Taking the smallest run time among three trials, Figure 20 shows the Benchmark Time, reported on output as a metric for this benchmark. The data for KNC shown on the left half of the chart and that for the host SB on the right half. NPB has both good coarse grain and fine grain parallelism. The best performance on MIC is just 1.3X slower than best on SB node.

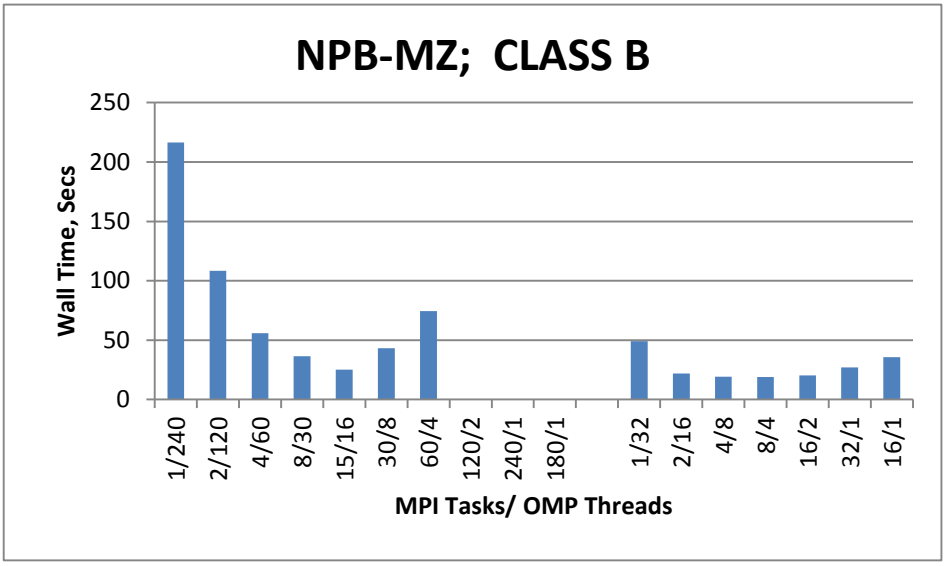

Figure 20 NPB BT-MZ performance (Wall Time) on KNC left and SB right

6.8.2 Vampir Profile to gauge coarse and fine grain parallelization effectiveness

A profile of NPB-MZ is needed to reveal various compute time components. The profile may be strongly influenced by scale and input. The input data set used for this analysis is the CLASS B benchmark. For NPB this data was gathered using, 1 node, 8 MPI tasks and 2 OpenMP threads per task on Sandia's TLCC2 system called Chama with an mpiexec command as shown below:

mpiexec –n 8 ./bt-mz.B.8

ScoreP and Vampir are the tools used to gather the desired trace information. For restricting the size of the trace file generated by scoreP the code was modified to do only 20 time steps instead of 200. For NPB the entire code was instrumented with ScoreP. Figure 21 shows the function profile as percentage of the run time.

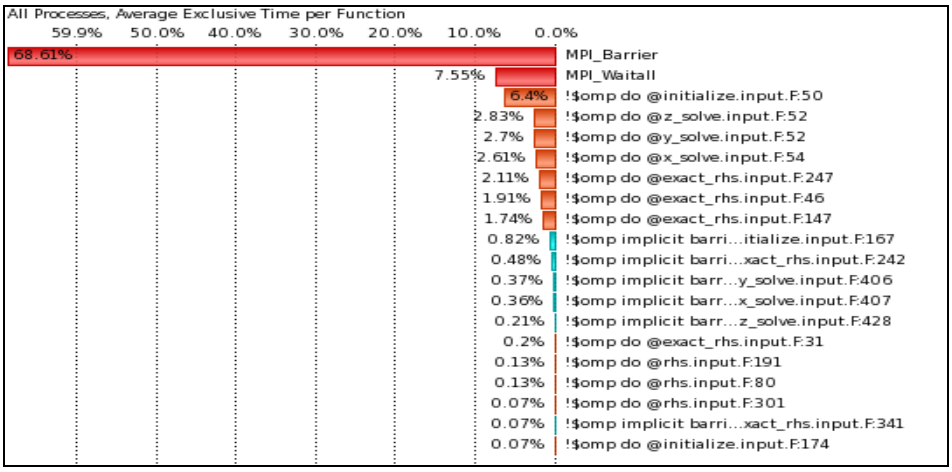

Figure 21 NPB BT-MZ profile showing percentage run time fractions

6.8.3 Vectorization effectiveness

Using the same input used for gathering the comparative performance between SB and KNC shown in Figure 20, picking the run with MPI tasks and OpenMP threads that yielded the best performance, the performance gain with compiler generated auto-vectorization for the host SB and the MIC measured on the Corner workstation is shown in Table 7. We see a 7% poorer performance with vectorization on the MIC and a 6.2% improvement on the host SB. Cause for the poorer performance on the MIC may be related to the large (16) threads per MPI task, but needs to be investigated further.

Table 7 miniDFT compiler auto vectorization performance for SB and KNC

Processor	Benchmark Wall time (secs) with Auto Vectorization	Benchmark Wall time (secs) with No Auto Vectorization
Intel MIC (15 MPI tasks/16 threads)	23.73	22.16
Intel Sandy Bridge(16 MPI tasks/2 threads)	17.47	18.56

7. FLOPS PERFORMANCE

A recent white paper by Leland et.al [5] investigates a response to a question raised in briefing to Dr. John Holdren, the President's Science Advisor, on the National Strategic Computing Initiative being developed within the Office of Science and Technology Policy. The question, raised in implicit form, was *whether we should focus on improving the efficiency of supercomputing systems and their use rather than on building, larger and ostensibly more capable systems that are used at low efficiency*. In that context one of the metrics often

considered is the percentage of the peak FLOPS. This is motivated by the increasing gap between sustained and peak performance and is quite relevant to an investigation on Intel MIC as it is the first X86-64 TFLOPS processor. While the motivation for the above question was the behavior of applications at very large scale, increasingly with the new generations of many-core processor nodes often with a hardware FLOPS accelerator, achieving a good fraction of the peak node performance is hugely important. Data gathered with the Trinity "single-node" benchmarks sheds some light on this question. This data was gathered on Cielo using the CrayPat tool. Figure 21 shows the measured performance as a percentage of the peak.

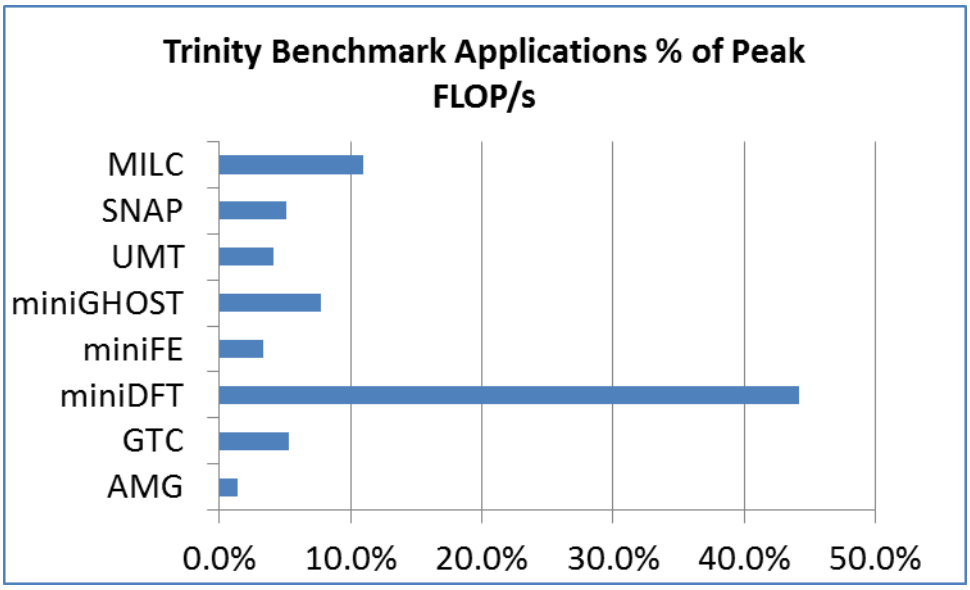

Figure 22 Trinity "single node" benchmarks FLOPS performance on Cielo

These results while signifying the need for significant performance optimization for the Trinity benchmarks, should be viewed in the light of percentage of peak floating point performance (5% — 20% and averages approximately 15%.) for highly tuned representative applications run on the NSF Blue Waters system at the NCSA [7].

8. CONCLUSIONS AND FUTURE WORK

The data from the sections above may be summarized comparing the ratio of the run time taking the best performance measured on MIC and the best performance on the host Sandy Bridge node. Table 8 provides such a summary.

Table 8 Wall clock run time ratio Knights Corner / Sandy Bridge node

Benchmark	miniFE	AMG	SNAP	UMT	GTC	MILC	miniDFT
Best KNC/ Best SB	0.8	2.01	2.95	1.38	1.95	2.32	3.30

From the data in Table 8, Figure 21, and, the vectorization effectiveness investigated for the applications, a conclusion that emerges is that much effort is required to fully exploit the architectural features of MIC to bring the performance in par with what Sandia users are used to

seeing on the TLCC2 clusters like Chama. We do recognize that the current targeted use of Xeon Phi is predominantly as a node accelerator to boost performance of compute intensive kernels. In other words native mode is not the intended usage model for Knights Corner. Publications in the literature [8], that show case 2X performance on the Knights Corner over Sandy Bridge based nodes for certain classes of applications and algorithms is encouraging as it attest to the potential of this architecture for applications that can exploit the many-cores/thread architecture and also benefit from the 512 bit vector units. Based on our experience with the Trinity benchmarks documented here, we may draw some conclusions in the light of our interest in the Knights Landing processor targeted for Trinity.

1) The analysis procedures laid out here to evaluate hybrid programming models, namely investigations to find the right balance between MPI tasks and threads at a node would be a necessary step before looking to scale an application to 10,000 or so nodes. The objective is to have as few MPI tasks as possible in a node to minimize data flow through the high speed inter node network.
2) Performance profile as shown in the applications sections to identify time spent in the application kernels, OpenMP or other thread parallel compute loops, OpenMP or threading overheads, MPI and MPI overheads is essential to optimally map applications to these many-core architectures.
3) Effective vectorization and procedures to measure it will be very important to close the gap between peak and sustained performance. Working with Intel to expose PMU counters that help us measure various vector and memory usage performance metrics will be very fruitful.
4) High performance thread level parallel MKL routines for the math kernels of interest to Sandia like sparse matrix-vector operations will facilitate rapid port of applications.
5) A simple model of the observed miniFE KNC to Dual SB time ratio of 0.8 is: (threads_SB/threads_KNC)*(BW-per-Th_SB / BW-per-Th_KNC) ; i.e. greater parallelism helps when MPI & OpenMP overheads are very small
6) For the other apps we are not seeing a performance improvement over dual SB due to different reasons: higher MPI /OMP overhead with greater parallelism, compiler did not take advantage of 512 bit SIMD, lower MKL performance , higher serial fraction & poorer core performance

REFERENCES

1. Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, George Chrysos, Pradeep Dubey, "Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor", IPDPS 2013.
2. https://software.intel.com/sites/default/files/forum/278102/intelr-xeon-phitm-pmu-rev1.01.pdf
3. Shannon Cepeda, "Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors, Part 2: Understanding and Using Hardware Events," 11/12/2012, https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding
4. http://www.cs.uoregon.edu/research/tau/home.php
5. https://www.nersc.gov/users/computational-systems/nersc-8-system-cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks.
6. Jim Jeffers and James Reinders "Intel Xeon Phi Coprocessor High Performance Programming", Intel Press, 2013
7. http://www.ncsa.illinois.edu/assets/pdf/news/BW1year_apps.pdf
8. James Baker, "Xeon Phi and The Art of Accelerator Optimisation; High Performance Computing with Unconventional Architecture", CSIRO Advanced Scientific Computing Report, 30th April, 2013

DISTRIBUTION

1	MS1318	Bruce Hendrickson	1400
1	MS1319	Doug Doerfler	1422
1	MS1319	Si Hammond	1422
1	MS1319	Jim Ang	1422
1	MS1319	Richard Barrett	1423
1	MS1319	Courtenay Vaughan	1423
1	MS0845	Michael Glass	1545
1	MS0801	Tom Klitsner	9320
1	MS0807	Mahesh Rajan	9326
1	MS0823	Dino Pavlakos	9326
1	MS0807	Dennis Dinge	9326
1	MS0807	Joel Stevenson	9326
1	MS0807	Anthony Agelastos	9326
1	MS0807	Bob Ballance	9328
1	MS0801	John Noe	9328
1	MS0899	Technical Library	9536 (electronic copy)


Sandia National Laboratories