Available online at www.sciencedirect.com

ScienceDirect

Procedia Engineering 00 (2015) 000–000

Procedia Engineering

www.elsevier.com/locate/procedia

24th International Meshing Roundtable (IMR24)

A Case Study - Scaling Legacy Code on Next Generation Platforms

William Roshan Quadros*

[1]*Sandia National Laboratories, PO Box 5800, MS 0897, Albuquerque, NM – 87185 USA*

Abstract

This research note discusses a case study that summarizes the procedure followed in porting a legacy code to scale on next generation platforms. Here, the legacy Laplace mesh smoothing algorithm is used on a hex mesh as a hotspot for the performance study. The case study was conducted on a testbed that has a similar hardware architecture with many integrated cores (MIC) as that of the next generation supercomputer called Trinity. The case study has shown good scaling using the hybrid MPI+X programming model.

© 2015 The Authors. Published by Elsevier Ltd.
Peer-review under responsibility of organizing committee of the 24th International Meshing Roundtable (IMR24).

Keywords: Parallel Programming, MPI+X, Hybrid Programming Model, Laplace Smoothing, Next Generation Platforms

1. Introduction

The next generation platforms built towards exascale computing are taking advantage of the many integrated core (MIC) revolution that can be characterized by the trends of increasing thread count and decreasing memory per thread. High performance computing (HPC) on these next generation architectures requires codes to exploit every opportunity for thread-level parallelism and architecture specific memory access pattern performance constraints. In this study, the target next generation platform is the Trinity supercomputer [1] to be hosted at Las Alamos National Laboratory. Trinity will have a Cray XC30 platform architecture containing Intel Knights Landing (KNL) MIC processors. In this case study, a testbed containing seven nodes with Intel Knight Corners (KNC) MIC coprocessors

*Corresponding author. Tel.: +1 505-220-9458; fax:+1 505-284-2418.
E-mail address: wrquadr@sandia.gov

1Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000

1877-7058 © 2015 The Authors. Published by Elsevier Ltd.
Peer-review under responsibility of organizing committee of the 24th International Meshing Roundtable (IMR24).

[2] (see Figure 1) was used to analyze node performance with the Laplace smoothing algorithm. This case study has enabled us to develop the following procedure for scaling a legacy code.

2. Procedure for Scaling Legacy Code on Next-Generation Platforms

Step 1: Profile legacy code to choose between refactor and rewrite

Profile the legacy code to determine its characteristics such as number of hotspots, distribution of hotspots, etc. In our case study, TAU [3] has been used, which is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements. If there are too many hotspots distributed throughout the legacy code, then it may be better to rewrite rather than refactor the code to perform on the next generation platforms.

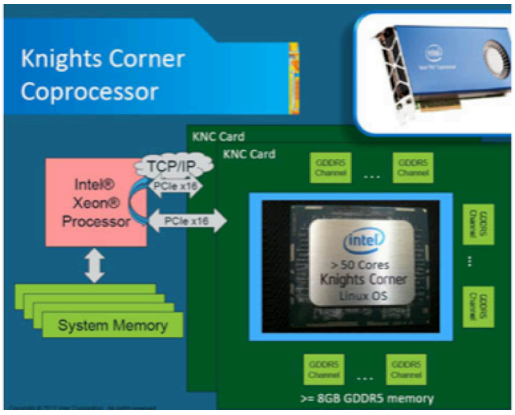

Figure 1. Intel Knights Corner Coprocessor

In our case study, the TAU profiling report on Sculpt [4], a grid-based hex mesh generator in CUBIT [5] showed that on average, the Laplace volume smoothing algorithm in CAMAL [6] was taking about 30% of the overall meshing runtime on our test cases. Therefore, we decided to refactor the volume smoothing hotspot in CAMAL rather than rewrite the entire mesh generator.

Step 2: Choose a suitable parallel programming model

The next generation platforms would require both distributed memory and shared memory programming models. In our study, hybrid Message Passing Interface (MPI) [7] and the OpenMP [8] programming model were used for two levels of parallelism: (1) distributed memory parallelism supported through the Intel MPI library and (2) thread level parallelism on the MIC device using OpenMP. A third level of parallelism can be achieved by using compiler flags that generate vector instructions for the 512-bit SIMD Vector Processing Unit (VPU) of KNC.

As performance portability and preserving the legacy code from potentially detrimental parallel directives were important, we used Kokkos [9], which provides a minimal overhead API that isolates user code from device specific hardware architecture. Kokkos currently uses device specific back-ends such as CUDA [10], pthreads [11], and OpenMP [8]. Using Kokkos should assist with the portability of the legacy code with the MPI+"X" programming model to scale on GPU-based next generation platforms such as Sierra [12], which will be hosted at Lawrence Livermore National Laboratory in 2017.

Step 3: Implement selected programming model

It is better to first convert the serial code to parallel using the MPI library for distributed level parallelism. This will enable the parallel code to run on both distributed memory and shared memory architectures. Note that each process has its own private data, i.e., it will not be able take full advantage of shared memory architecture. MPI would require domain decomposition considering load balancing and minimizing communication between the domains.

Next, threading can be implemented to take advantage of the shared memory architecture at each KNC card. This would require making the legacy code thread safe, which could be a significant challenge. Many of the serial legacy codes use libraries that are not thread-safe. It is recommended to first refactor the hotspot to make it thread-safe by avoiding dependencies on any non-thread-safe library, avoiding global and static variables, using "const" functions and variables in C++, etc. OpenMP is best suited for loop level data parallelism for shared memory via various directive and clauses. For fine control, pthreads can be used instead of OpenMP.

In order to achieve performance portability on supercomputers with different architectures containing GPUs, OpenMP/pthread can be replaced with Kokkos at the hotspots. A sample performance portable pseudo code is given below:

```
#ifdef ENABLE_KOKKOS

// data parallelism on N nodes using Kokkos
MyClass::class_method( function arguments )
   ...
   // 1st argument: number of nodes
   // 2nd argument: this object
   Kokkos::parallel_for( N, *this);
   ...


// operator() for Kokkos::parallel_for
MyClass::operator()( int k ) const
   ...
   // Laplace smoothing at node k
   laplacian_smooth_at_node(k);
   ...

#endif
```

It is generally beneficial to wrap the threaded code with the preprocessor macros `#ifdef` and `#endif` as this allows for the Kokkos code to be enabled or disabled during the performance study. The `Kokkos::parallel_for` is a C++ functor, which has a work callback and shared input parameters that are operated on. Thread safety in `operator()` is achieved by using the `const` keyword. The function `laplacian_smooth_at_node(k)` calculates the new coordinates of the kth node as given in Equation 1. Additionally, the Intel compiler option -vec-report flag can be used for diagnostic information regarding vectorization to stdout. In CAMAL, the C++ standard template library (STL) containers such as `std::map` was not refactored as it is found to be thread safe in the read operation.

Step 4: Determine optimal number of MPI processes and number of threads per node for job submission
In order to achieve high scaling, it is important to specify the optimal number of MPI processes and OpenMP threads per MPI process based on the underlying hardware architecture. This requires a series of studies that vary these two main parameters on a series of test cases. Various factors that could influence the outcome include MPI communication cost, overhead incurred through thread synchronization, shared communication state, computational cost, etc. Increasing the number of MPI processes per node might decrease the MPI communication cost. However, increasing the number of threads per MPI process enables better utilization of node resources and can decrease computational cost. Thus, the hybrid MPI+"X" (e.g. MPI+OpenMP for Trinity) applications must make a delicate tradeoff between the number of processes and the number of threads per process at the time their jobs are launched.

3. Results

Multiple studies have been conducted on the testbed for Trinity and this section highlights one of the findings. Two compute nodes were allocated in the seven-node testbed with each compute node containing two KNC cards. As Kokkos does not currently manage the threads on multiple MIC devices, each MPI process can use one MIC device at most. Therefore, in our studies, four or more MPI processes were used. The older KNC MIC device contains 57 cores and the newer MIC device contains 61 cores. Therefore, the OpenMP threads per MPI process were varied from 1 to 64.

Both CUBIT and Kokkos were compiled using ntel compiler 13.4.183 and intelmpi 4.1.1.036. It is important to compile the code for MIC architecture using compiler flag –mmic. Also, -lmpi_mt option enables MPI with multi threading. For threading, Kokkos used compiler flag –openmp for OpenMP. The compiler's vectorization flags were not varied in the below reported results.

CAMAL's Laplace volume smoothing algorithm was used as the hotspot kernel. Ten iterations of Laplace smoothing were performed on a 5 million node hex mesh. Equation 1 shows the Laplace equation, where N is the number of adjacent nodes to node k, $x_{i,j}$ is the coordinate of the jth adjacent node of node k in the ith iteration, and $x_{i+1,k}$ is the new $i+1^{th}$ iteration coordinate set for node k.

$$x_{i+1,k} = \frac{1}{N}\sum_{j=1}^{N} x_{i,j} \qquad \text{Equation 1}$$

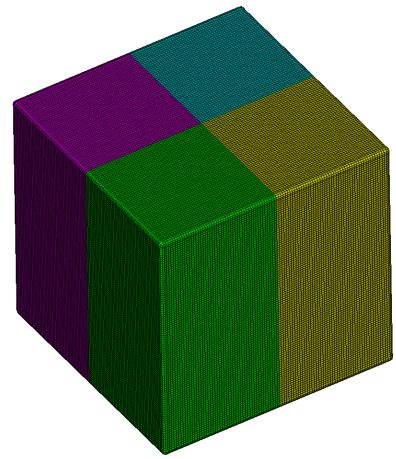

Figure 2. 5 million node mesh on 4 MPI processes

Note that two arrays were used to store coordinates of ith and $i+1^{th}$ iterations for 1.25 million nodes in each process. Use of two arrays doubles the memory requirement but avoids the thread safety and parallel-serial consistency issues.

One of the studies focused on OpenMP threading performance on the KNC processors. Therefore, MPI-related parameters were kept constant. As shown in Figure 2, equal size domains were used on all four MPI processes with identical symmetry to keep the MPI communication cost equal among MPI processes. As shown in the 1st column of Table 1, the number of MPI processes is kept constant, i.e., one MPI process on each of the 4 MIC processors. The OpenMP threads per MPI process were varied from 1 to 64 as shown in the 2nd column. The fourth column shows the 5-trial average runtime of Laplace smoothing kernel on a 5 million nodes hex mesh. The fifth column shows the ideal time for linear scaling as we increase the number of threads per process. The final column shows the percentage deviation from the ideal linear scaling. As we increase the threads per process, the deviation from the linear scaling increases due to thread startup and other overhead costs as shown in Figure 3. On a 57/61core KNC coprocessor, this study observed a 95% reduction in runtime (a 20X speedup) as the single process runtime of 278.88 seconds dropped to 14.24 seconds.

Table 1. 5-trial average result of volume smoothing on a 5 million node hex mesh

Number of Processes	Thread per Process	Process X Thread	Actual Runtime (sec)	Ideal Runtime (sec)	Percentage Deviation
4	1	4	278.88	278.88	0%
4	2	8	140.48	139.44	0.74%
4	4	16	73.22	69.72	5.02%
4	8	32	40.23	34.86	15.40%
4	16	64	24.16	17.43	38.61%
4	32	128	23.64	8.71	171.25%
4	64	256	14.24	4.35	226.79%

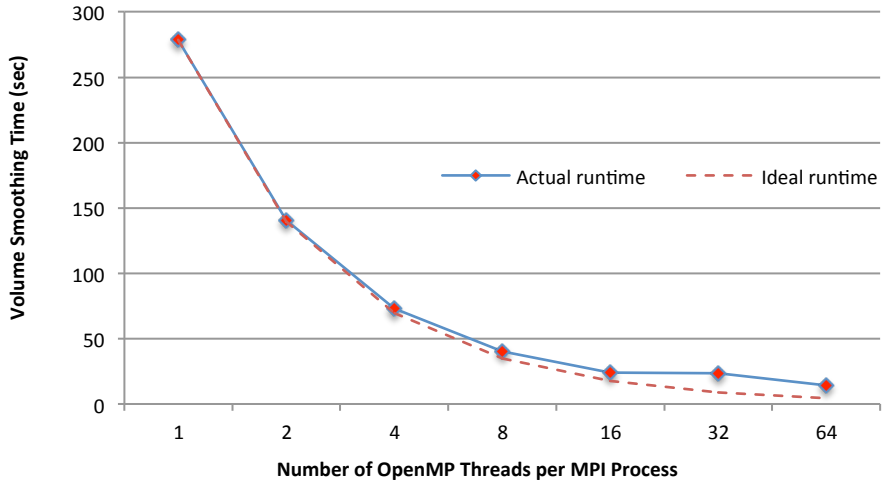

Figure 3 Linear scaling and actual scaling graphs

4. Conclusion

This research note outlines a procedure for scaling legacy codes on next generation platforms based on a case study. The case study used a legacy Laplace volume smoothing algorithm of CAMAL as the kernel for the hybrid MPI+OpenMP programming model on a testbed of Trinity. The case study enabled us to become familiar with profiling tools to identify hotspots, debugging tools, and optimization tools, and to come up with a procedure for scaling other legacy modules in CUBIT. The results recommend use of a high-level performance portable library, such as Kokkos, that can handle multiple advanced architecture specific memory access pattern performance constraints without having to modify the user code. Here only data parallelism was implemented and currently no effort has been made on task parallelism. The data parallelism using Kokkos achieved node performance speedup of 20X on a KNC MIC device.

References

[1] Trinity Supercomputer, http://www.lanl.gov/projects/trinity/ (accessed Aug 12, 2015)
[2] Intel Knights Corner Coprocessor, http://ark.intel.com/products/codename/57721/Knights-Corner (accessed Aug 12, 2015)
[3] TAU Performance System, http://www.paratools.com/TAU (accessed Aug 12, 2015)
[4] Steven Owen, Matthew Staten, Marguerite Sorensen, "Parallel Hex Meshing from Volume Fractions", 20th International Meshing Roundtable, Springer-Verlag, pp.161-178, October 23-26, 2011
[5] CUBIT Geometry and Meshing Toolkit, https://cubit.sandia.gov/public/14.1/Cubit-14.1-announcement.html, Version 14.1, Released Jan 13, 2014.
[6] CAMAL – The Cubit Adaptive Meshing Algorithm Library, https://cubit.sandia.gov/public/camal.html (accessed Aug 12, 2015)
[7] MPI, http://www.mpi-forum.org/docs/docs.html (accessed Aug 12, 2015)
[8] OpenMP API Specification for Parallel Programming, openmp.org, Jun. 2013.
[9] Carter Edwards, Christian Trott, "Kokkos: Enabling performance portability across manycore architectures", Extreme Scaling Workshop (XSW), IEEE, pp. 18–24, 2013
[10] CUDA home page, www.nvidia.com/object/cuda home new.html, (accessed Jun. 2013).
[11] Pthreads, IEEE Std 1003.1, 2004 Edition, 2004.
[12] Sierra Supercomputer, https://www.llnl.gov/news/next-generation-supercomputer-coming-lab (accessed Aug 12, 2015)