Advanced Programming Model Constructs Using Tasking on the Latest NERSC (Knights Landing) Hardware

Jeremy Kemp[1], Alice Koniges[2], Yun (Helen) He[2], and Barbara Chapman[3]

University of Houston, Houston, TX[1]
NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA[2]
Stony Brook University, Stony Brook, NY[3]

UNIVERSITY of HOUSTON
DEPARTMENT OF COMPUTER SCIENCE

Introduction

Most shared memory programming in HPC is done with highly synchronous constructs such as the "parallel for" in OpenMP. With the increasing core counts and non-uniformity in emerging hardware, a more asynchronous programming model is needed.

The goal of this summer was to explore OpenMP tasks on the Knights Landing (KNL) hardware that will be used in Cori Phase II to demonstrate the potential benefits of an asynchronous programming model. This is done with two kernels, LU decomposition and an iterative Jacob Kernel, as well as a proxy application, CoMD[1].

For each of these applications, a tasking version without data dependencies was developed to show the performance cost of moving to tasks. Then a version with task dependencies shows the improvements that can be gained from removing unnecessary synchronization, and finally several optimizations on the task dependency version show how much additional performance can be gained.

[1] unoptimized tasking version provided by Riyaz Haquw (UCLA) and Bronis deSupinski (LLNL)

Application Kernels

LU decomposition

The initial matrix is divided into a 2D matrix of blocks to improve cache usage, and enable parallelization. There are 4 distinct operations divided into 3 phases.

The worksharing (parallel for) version divides these into 3 phases for each iteration, with a barrier between each phase.

The tasking version has a similar structure to the parallel for version, spawning tasks inside of loops and then synchronizing on taskwait instead of barriers.

The task dependency version removes the synchronization between phases as well as between iterations, as Illustrated by diagram 1.

LU has no communication or synchronization between blocks. The task dependencies simply control access to the matrix so only one thread at a time is writing to it.

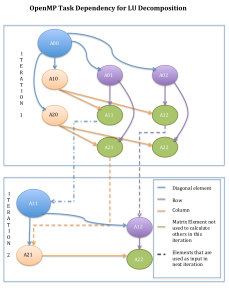

Diagram 1. Shows the dependencies between blocks in the LU Kernel.

Jacobi Solver

As diagram 2 shows, each element of the matrix depends on each of its neighbors. As a result, overwriting an element will change the result of its neighbor, so a second matrix is typically written to, and then swapped with the original matrix at the end of every iteration.

Each version of Jacobi divides up the 2D matrix into groups of whole rows instead of blocks. This reduces false sharing and enables very sequential access of memory.

Similar to LU, the tasking version is similar to the parallel for version, and the task dependency version removes the synchronization between iterations.

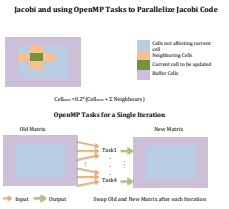

Diagram 2. Shows the dependencies between blocks in the Jacobi Kernel.

CoMD

Data in CoMD consists of atoms in 3 dimensional space, where each atom has a position, velocity, energy, and force. The 3D space is divided into boxes, and each atom is placed into a box. Each iteration is a timestep where each of the attributes is recalculated for each atom, and then atom is moved to its corresponding box, if it left its previous box.

The majority of the compute time is spent calculating force between particles, where the tasking version has one task for calculating forces between atoms in a pair of boxes. The worksharing and task dependency versions both divide up the work by boxes, calculating the interactions with all of its neighbors.

Diagram 3. Shows the division of space into boxes and the atoms involved in a single calculation

Optimizing the LU Kernel

Performance Overview

The parallel for loop is the default approach to parallelism in OpenMP. These two charts provide an overview of tasking performance relative to this, and how much benefit there is from additional optimizations, which easily surpass the performance of worksharing.

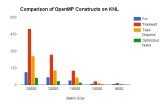

Fig 1. Shows the performance of the different approaches on KNL

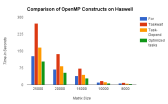

Fig 2. Shows the performance of the different approaches on Haswell

Memory Layout

Proper use of the high bandwidth memory can have a major impact on performance. This can either done with the memkind library, or numactl, or by using the quadcache configuration of the KNL. Changing the way the matrix is allocated can also improve performance when operating on blocks of memory, especially on larger matrix sizes.

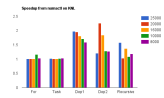

Fig 3. Shows the improvement from allocating in high bandwidth memory via numactl.

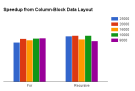

Fig 4. Shows the performance gained from allocating the matrix in columns of blocks in place of a single allocation.

Blocking Optimizations

When the number of tasks are mapped to the number of blocks either the size of the block gets larger to accommodate larger matrix sizes, or the number of blocks increases. In order to avoid the overhead of too many tasks, or the loss of locality from oversized blocks, tasks must be mapped to multiple blocks, and if possible multiple iterations over the same blocks. Figure 6 shows the performance improvement of 3 optimizations for task dependencies; combining multiple blocks per tasks (block), combining multiple blocks and iterations per task (block-iter), and a recursive cache oblivious version (block-rec)that operates on multiple blocks and iterations.

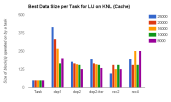

Fig 5. Shows how much the size of data (possibly multiple blocks) that each task operates on varies for different implementations and matrix sizes.

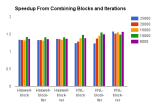

Fig 6. Shows the performance gained from combining multiple blocks into a task and the performance gained by combining multiple blocks and iterations per task.

Applying Optimizations

Jacobi

Jacobi has very little cache reuse, as it only writes a given element once per iteration. As a result, dividing up the matrix into blocks has no benefit, so the block combination optimizations don't apply.

Iteration combining is possible, but more complex, due to overlapping read/writes between neighbors. For the Jacobi iteration optimization, the second matrix is removed, and replaced by 3 threadlocal scratch rows and 4 synchronization rows per chunk of rows. Each task performs an iteration for 3 rows writing the results into the scratch rows, and then writing the second iteration back to the original matrix.

Whole rows are too large to fit in the L2 of the KNL, so the cache reuse does not improve. Whereas the Haswell performance more than doubles due to to the rows fitting into the very large L3 cache. Further implementation work is needed on an iteration combining version that operates on blocks that fit inside of smaller Caches.

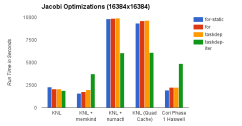

Fig 7. Shows the performance of the different versions of Jacobi

CoMD

With CoMD, the initial conversion of the force function to task dependencies improved performance. The all-task-dep conversion of the application replaced all parallelism and synchronization in the application with task dependencies, including serial regions with data dependencies. Combining blocks would have been a better first optimization, as it would have only reduced overhead where the full conversion introduced too much overhead and hurt performance.

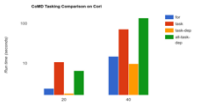

Fig 8. Shows the performance of the different versions of CoMD

Hardware

KNL (From the CARL NERSC KNL testbed) has 64 cores, each with 4 hardware threads. Two cores are grouped in to pairs as a tile and share 1 MB of L2 cache. There is no L3 cache, but there is 16 GB of high bandwidth memory that can be configured as cache or allocated manually in different modes.

The results on this poster use the *quadflat* and *quadcache* configurations. The cache configuration turns the MCDRAM into cache that is no longer programmable, while the quadflat configuration enables the use of memkind or numactl to more finely control how memory is used.

For comparison, Cori Phase 1 nodes have 2 Haswell processors with a total of 32 cores, 2 hyperthreads per core. Each core has 256 KB of L2 cache, and each processor has 40 MB of L3 cache.

Conclusions

Locality is important, and difficult to do with OpenMP tasks; the consistent poor performance of tasks (without task dependencies) shows how much overhead is introduced and how much locality is given up. On the other hand, unoptimized task dependencies demonstrate how much can be gained by removing unnecessary synchronization. The optimizations on the task dependencies can then improve the locality and achieve much better performance than the parallel for loops.

There are two major differences when programming for KNL over Haswell; KNL has very little cache per thread relative to Haswell, and using MCDRAM properly can drastically improve performance. The Jacobi results illustrate both of these points very well. The fastest version on Haswell performs the worst on KNL due to the cache size, and the versions that move through memory sequentially perform much better.

Future work includes further applications of the optimizations explored with LU to Jacobi, CoMD, and other applications.

References

1. http://colfaxresearch.com/knl-numa/
2. Heller, Thomas, Hartmut Kaiser, and Klaus Iglberger. "Application of the ParalleX execution model to stencil-based problems." Computer Science-Research and Development 28.2-3 (2013): 253-261.
3. http://www.exmatex.org/comd.html
4. http://www.nersc.gov/users/computational-systems/cori/
5. https://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/knl-white-boxes/