KNL Performance Comparison: HLBM

August 2017

G.N. Barakos and M.A. Woodgate
George.Barakos@Glasgow.ac.uk Mark.Woodgate@Glasgow.ac.uk

CFD Laboratory
School of Engineering
University of Glasgow
Glasgow, G128QQ, Scotland, Uk

1. Compilation, Setup and Input

Compilation

The code was compiled using the Intel C++ compiler on both KNL and Xeon nodes. Version 17.0.0.098 was used on the KNL while the default version 15.0.2.164 was used on the Xeon. Currently we have no resources left on the Xeon to compare the performance of the difference between the two versions of the Intel compiler but it is not expected to have a large effect on the runtime of the code. The compiler optimization option used was -O3 and the compiler directives for the source code where -D_NEW_ALLOC -D_METHOD2 which is the most efficient memory layer currently implemented in the parallel code.

Setup

The code uses MPI for parallel communication and was run with up to 24 processes per node on the Xeon and 64 processes per node on the KNL. The performance across nodes shows linear speedup from the Xeon but, as it is currently only possible to run on two KNL nodes, there are not enough data points to draw any conclusions on the performance across nodes for the KNL system. The memory option used on the KNL system was quad_100 where all the MCDRAM is used to cache the main memory.

Input

The benchmark was a three-dimensional counter rotating vortex problem with periodic boundary conditions in all three directions. The lattice size was 121x241x241 and this was split into the number of MPI processes to be used. For example, the single process run only one block and for the case with 16 processes the lattice was split into 16 blocks each of size 61x61x61. Using only a single block per processes give the highest possible performance due to minimizing the halo data exchanges.

It should be noted that this problem size requires less than 16GB of memory and so all the data will be cached using the MCDRAM.

2. Performance Data

Figure 1 shows the parallel performance of HLBM while running on ARCHERs Xeon computer nodes. The scaling within a node shows a marked drop off in parallel performance when running on more than 4 cores per node – 8 cores in total. This is because the method is very memory bandwidth-intensive and general memory bandwidth has not kept pace with the ever-increasing number of cores on CPUs. However, the performance across nodes shows linear speedup going from 1 node (24 cores) to 64 nodes (1536 cores). This is because the number of lattice points per process dropped from 288,000 when on a single node to just 4500 when on 64. This means a much larger percentage of the data could be stored in the cache which increases the core performance, by about twenty percent. This gain in sequential performance offsets the communication costs.

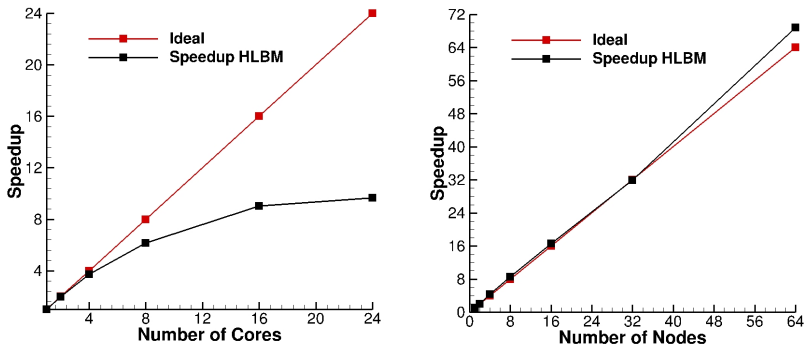

Figure 1: The speed up curve for running HLBM within and across ARCHER Xeon computer nodes – (Two 2.7GHz 12-core E5-2697 v2 Processors)

The code was also evaluated on nodes configured in cache mode with all 16GB of the on-chip Multi-Channel DRAM (MCDRAM) used to cache the system memory, and job sizes were small enough so all the data could fit within the cache. The MCDRAM is a high-bandwidth memory which fits well with the needs of a method like HLBM. The results can be seen in figure 2 and although the single core performance of a KNL processor was three times slower, mainly due to the lower clock speed, the parallel scaling was much better at high number of cores. Hence 24 processes on an ARCHER Xeon compute node run the same as 32 on a KNL node. This results in the KNL nodes being 80% faster when both nodes are full utilized.

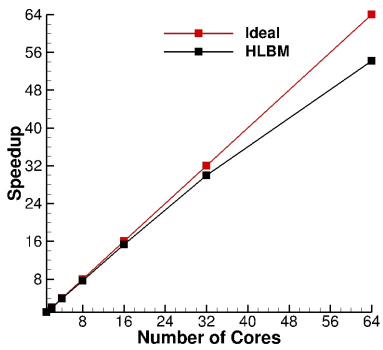

Figure 2: The speedup curve for running HLBM within an ARCHER KNL processor (model 7210) running at 1.3GHz

Table 4 shows the performance difference between the quad_100 configuration and the quad_0 configuration. The effect of having no caching between the cores and the main memory results in just over doubling the runtime of the code and shows the importance of having all the main memory cached for a code with low ratio of memory accesses to floating point operations.

Number of Xeon Cores	CPU time per iteration
1	2.1249s
2	1.0721s
4	0.56952s
8	0.34484s
16	0.23538s
24	0.21953s

Table 1: Performance data for running on different numbers of cores on an ARCHER Xeon compute node.

KNL quad_100 Cores	CPU time per iteration
1	6.772s
2	3.503s
4	1.743s
8	0.880s
16	0.442s
32	0.226s
64	0.126s

Table 2 : Performance data for running on different numbers of cores on an ARCHER KNL compute node in quad_100 configuration.

Number of Nodes	Xeon Nodes (24 Cores)	KNL Nodes (64 Cores)
1	0.21953s	0.12835s
2	0.10560s	0.06350s
4	0.05050s	0.03305s
8	0.02586s	0.01729s

Table 3: Comparison of scaling across nodes for ARCHER Xeon and KNL nodes.

KNL Configuration	Quad_100	Quad_0
CPU time per iteration	0.1286s	0.2942s

Table 4: Comparison of performance on a since KNL compute node in two different configurations.

The same test case has also been benchmarked on two other HPC systems. The first was ARCHIE-WeST (Academic and Research Computer Hosting Industry and Enterprise in the West of Scotland) were the 276 compute nodes are 2 Intel Xeon X5650 2.66 Ghz CPU with 6 cores each and 48 GB of memory connected together by 4xQDR Infiniband interconnect.
The second was through early access to the Cirrus HPC service which has 280 computer nodes containing two 2.1 GHz, 18-core Intel Xeon E5-2695 (Broadwell) series processors, 256 GB of memory and is connected with a single FDR infiniband interface with a bandwidth of 54.5 Gb/s.

It should be noted that the single compute node case scales differently from the ARCHER case since the default behaviour of the queue of fully loading the first CPU before running processes on the second was not overridden. It was found when testing on ARCHER than the parallel efficiency of running on either N cores of a single CPU or N cores on both CPU was very similar and hence the halving of the CPU time per iteration when going from 6 to 12 cores on ARCHIE-WeST and 18 to 36 cores on Cirrus.

Table 5 shows the performance of HLBM on ARCHIE_WeST. The single core performance is 35% slower than an ARCHER core and scaling is not as good with only a 3% increase in performance when increasing the number of cores used on a CPU from 4 to 6. When running across multiple nodes the scaling shows super linear speedup as the size of the blocks decrease as the problem size remains constant.

ARCHIE-WeST Cores (Nodes used)	CPU time per iteration
1 (1)	2.6361s
2 (1)	1.4362s
4 (1)	1.0449s
6 (1)	1.0124s
12 (1)	0.5252s
24 (2)	0.2647s
48 (4)	0.1342s
96 (8)	0.06389s
192 (16)	0.03121s
384 (32)	0.01623s

Table 5: Performance of HLBM on different numbers of cores and nodes on an ARCHIE-WeST

Table 6 shows the performance of HLBM when run on the Cirrus HPC system. Although the Cirrus nodes are slower (2.1Ghz vs 2.7 GHz) the overall single core performance is very similar. Cirrus also scales better within the CPU. However, just like an ARCHER node after 8 processes are running within a CPU there is very little performance gain. This means for HLBM there is very little advantage to be gaining from using more than 16 cores per computer node which is just half the number available. The code is actually slower when running with the maximum number of cores possible. The internode performance is the same as the other HPC systems and shows excellent scaling up to many thousands of cores.

Cirrus Cores (Nodes used)	CPU time per iteration
1 (1)	2.0763s
2 (1)	1.0325s
4 (1)	0.5376s
8 (1)	0.3501s
16 (1)	0.3244s
18 (1)	0.3345s
36 (1)	0.1669s
72 (2)	0.7711s
144 (4)	0.04013s
288 (8)	0.02011s
576 (16)	0.01042s
1152 (32)	0.005224s
2304 (64)	0.002108s

Table 6: Performance of HLBM on different numbers of cores and nodes on Cirrus.

3. Summary and Conclusions

HLMB shows very similar behavior on all the HPC systems it's been benchmarked on. There is a marked difference to the scaling within a compute node, compared to the scaling between compute nodes. On modern machines with many cores per CPU only a small subset of the cores can effectively be used due to the low floating-point operation count of the Lattice Boltzmann method. Indeed, on Cirrus only about 40% of the core can effectively be used. This does present an opportunity of possibly being able increase the floating-point workload without increasing the CPU time per iteration thought say a more accurate and hence computationally more expensive equilibrium function or the addition better sub-grid scale turbulent models.

Although the single core performance of the KNL is much slower than that of the Xeon, mainly due to the much slower clock speed, the performance per node was 80% greater. Since HLBM is aimed at the computation of wakes in real time any performance gain is welcome. Due to the smaller lattice sizes used in this type of application the situation where there is not enough cache memory on the KNL nodes will not arise. For larger lattices which cannot be fully cached the performance advantage of the KNL nodes my well be lost.

A compact openMP version of the code is also being tested on the KNL nodes which has been highly optimized to reduce the work load but loop unrolling removing redundant calculations etc. However, due all these optimizations it fails to take advantage of the vector processing units with the KNL node. Different computational kernels are being tested to see if it possible to rework the code so it can make use of the VPUs. The basic code using just the compiler to the optimization is slower in number of lattice updates per second but much faster looking at raw floating point operations. It is thought there could be a version in between these two extremes which could be faster than the current optimized version. It has already been determined that a simple padding of the workspace so that the inner loop always has memory-aligned access is detrimental to the overall performance of the code.