Performance Evaluation of Computation and Communication Kernels of the Fast Multipole Method on Intel Manycore Architecture

Mustafa Abduljabbar[1](✉), Mohammed Al Farhan[1](✉), Rio Yokota[2], and David Keyes[1]

[1] King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
{Mustafa.AbdulJabbar,mohammed.farhan,david.keyes}@kaust.edu.sa
[2] Tokyo Institute of Technology, Tokyo, Japan
rioyokota@gsic.titech.ac.jp

Abstract. Manycore optimizations are essential for achieving performance worthy of anticipated exascale systems. Utilization of manycore chips is inevitable to attain the desired floating point performance of these energy-austere systems. In this work, we revisit ExaFMM, the open source Fast Multiple Method (FMM) library, in light of highly tuned shared-memory parallelization and detailed performance analysis on the new highly parallel Intel manycore architecture, Knights Landing (KNL). We assess scalability and performance gain using task-based parallelism of the FMM tree traversal. We also provide an in-depth analysis of the most computationally intensive part of the traversal kernel (i.e., the particle-to-particle (P2P) kernel), by comparing its performance across KNL and Broadwell architectures. We quantify different configurations that exploit the on-chip 512-bit vector units within different task-based threading paradigms. MPI communication-reducing and NUMA-aware approaches for the FMM's global tree data exchange are examined with different cluster modes of KNL. By applying several algorithm- and architecture-aware optimizations for FMM, we show that the N-Body kernel on 256 threads of KNL achieves on average 2.8× speedup compared to the non-vectorized version, whereas on 56 threads of Broadwell, it achieves on average 2.9× speedup. In addition, the tree traversal kernel on KNL scales monotonically up to 256 threads with task-based programming models. The MPI-based communication-reducing algorithms show expected improvements of the data locality across the KNL on-chip network.

Keywords: AVX-512 · Fast multipole method · Intel knights landing

1 Preliminaries and Outline

Contemporary High-Performance Computing (HPC) systems are assembled from thousands of shared-memory compute nodes, which are progressively metamorphosing from multicore to manycore architecture with a hybrid layered memory

F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 553–564, 2017.
DOI: 10.1007/978-3-319-64203-1_40

hierarchy. Emerging manycore processors feature energy efficient, low frequency compute cores that support lightweight processing thread(s). For example, the second generation Intel Xeon Phi “Knights Landing” (KNL) can accommodate, in a single chip, up to 72 cores and 4 threads per core, which access an on-die high bandwidth memory. This immense computational power can be exploited by compute-intensive scientific algorithms.

The N-Body problem is used to sum up mutual interactions of discrete entities in $O(N^2)$ steps, which is a practical example of a compute-intensive kernel that can utilize the emerging manycore hardware. Its importance stems from the fact that it appears in many scientific applications such as electromagnetics, electrostatics, fluid mechanics, and astrophysics. In the form of the Fast Multipole Method (FMM), it is used either as a direct solver, or as an accelerator within an iterative solver for particular matrix-vector multiplications arising from the solution of Laplace or Helmholtz Partial Differential Equations (PDEs). N-Body methods can be considered as “matrix-free” methods, where a matrix is dynamically built before being multiplied by the source-point vector. This makes them favorable when the geometry of a problem changes rapidly such as particle-based methods where particles evolve every time step [15]. Tree codes like Barnes-Hut [3] build a geometric quad/oct tree to bring the quadratic complexity of N-body problem down to $O(N \log N)$. This is done by introducing a cutoff distance beyond which particles interact as cells located at the center of mass. FMM is an example of a tree code that uses hierarchical multipole expansions to approximate the far-field interactions up to specific error bound (ϵ) derived from the Multipole Acceptance Criteria (MAC) [7]. It solves the N-Body problem in asymptotically linear time complexity ($O(N)$). FMM is a highly computationally intensive algorithm that is favorable to manycore architectures.

1.1 Main Components of Parallel FMM

The general scheme of any parallel Fast Multipole solver consists of the several modules specified below. These typically execute in a fork-join sequence, with some exceptions mentioned in [2,17].

- Partitioning stage: Domain-decomposes the input while maximizing locality across processes. Foundations can be found in [12].
- Oct/Quad tree construction.
- Upward pass: A bottom-up sweep of the tree to execute the Particle-to-Multipole (P2M) and Multipole-to-Multipole (M2M) kernels. FMM kernels are explained in [7].
- Traversal: A depth-first local and global tree traversal to calculate near-field interaction by calling the Particle-to-Particle (P2P) or aggregate multipoles to local expansions for the far-field (i.e., Multipole-to-Local (M2L)).
- Communication: The far-field cells are propagated to other processes in a sender-initiated fashion.
- Downward pass: A top-down traversal that reduces local expansions using Local-to-Local (L2L) and Local-to-Particle (L2P).

Sections 2 through 4 briefly explain the P2P, traversal and communication modules, which contribute to the bulk of the total execution time, in the context of manycore parallelism.

Major fundamental and incremental contributions describe parallel FMM algorithms and implementations on shared and distributed memory [6]. Among these contributions are parallel FMM libraries that include PVFMM [13], pfalcON, PEPC and ExaFMM [16]. Our choice for this paper is ExaFMM due to its reported efficient shared-memory optimizations, which range from adaptability to different task-based threading models to low-level Advanced Vector eXtension (AVX) vectorization. On a single socket Intel Xeon X560, ExaFMM outperforms the traditional FMM libraries [14]. Furthermore, Bédorf et al. provide an implementation and analysis of a gravitational N-Body tree code, that has been redesigned to run on top of the GPU architecture [4]. This results in a processing rate of 2.8 million particles/second.

1.2 Paper Contributions

The main contributions of this paper are:

- Exploit aggressive Single Instruction Multiple Data (SIMD) optimizations to vectorize the N-Body kernel, and also optimize the outer and the inner loop via certain loop tiling techniques with a specific stride size.
- Perform performance comparisons and analysis of the N-Body kernel between: (1) handwritten vectorization code using Intel Intrinsics and the compiler's auto-vectorization, and (2) inner and the outer loop tiling, on the state-of-the-art manycore and multicore Intel architectures.
- Carry out in-depth performance analysis and benchmarking of different task-based programming paradigms to parallelize the Tree Traversal kernel of FMM on KNL architecture.
- Analyze the performance of various MPI-based NUMA-aware communication algorithms of FMM within a single node to overcome the hurdles of cache line transfer inside the on-chip network of KNL, and study multiple cluster modes of KNL.

2 Direct N-Body Kernel on Modern Intel Architectures

The direct N-Body Kernel is manifested as the P2P near-field interactions within FMM. Along with the M2L kernel, P2P contributes to the bulk of execution time by performing the largest share of FMM computations [9]. The number of Floating Point Operations (FLOPs) per each P2P call is $20 \times n_i \times n_j$, where n_i is the size of the target cell (outer loop), n_j is the size of the source cell (inner loop), and 20 is the number of operations needed to calculate: (1) the smoothed Laplace potential ($\phi_i = \sum_{j=1}^{N} \frac{m_j}{r_{ij}}$), (2) the accelerations ($a_i = \nabla\phi_i = -\sum_{j=1}^{N} \frac{m_j r_{ij}}{r_{ij}^3}$), and (3) the distance between bodies located at x_i and x_j, given ϵ as the smoothing factor ($r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 + \epsilon^2}$). We store source and target fields in separate vectors to avoid loop conflict dependencies. This facilitates automatic and handwritten SIMD optimizations to exploit each core's two Vector Processing Units (VPUs) [10]. The outer loop is parallelized using OpenMP and the effect of hyperthreading is explored with two and four threads per a KNL core. A classical question that arises in the situation of nested for loops is at which level the loop should be vectorized. Along with the SIMD optimization techniques, this question is addressed thoroughly in the subsequent results section.

3 Task-Based Traversal of ExaFMM

As mentioned in Sect. 1.1, the traversal stage calculates near-field or self interactions (P2P); whereas far-field potentials are aggregated to well-separated cells through M2L kernel calls. The essential difference between pure tree codes and FMM is that the former usually constructs the tree using a linked-list data structure; the tree is traversed in a recursive top-down manner, and well-separated cells are identified by applying the MAC. In contrast, adaptive FMM does not traverse the tree, nor does it construct a linked-list between parent and child cells. It, however, constructs a Hilbert/Morton key by interleaving bits of $X-Y-Z$ cell coordinates. Typically the parent's neighbor's child cell is considered well-separated. Nonetheless, there are several downsides of this technique, which are highlighted in [14].

Due to potential task-based parallelism, we configure ExaFMM to use Dual-Tree-Traversal (DTT), which traverses the source and target cells at the same time. Classical OpenMP threading is not applicable for the lack of an outer-loop over target cells. DTT takes a source and a target cell, and expands the larger until either MAC is satisfied or both are leaf cells. Algorithms 1 and 2 demonstrate the general structure of DTT code. Nested task-parallelism can be effectively incorporated by passing an integer *nspawn* that indicates the size of cells that can spawn a task as shown in Line 8 of Algorithm 2.

Algorithm 1. DualTreeTraversal(C_i, C_j)

```
1 if $C_i > C_j$ then
2     foreach $c_i$ in $C_j.Children$ do
3         Interact ($c_i$,$C_j$)
4 else
5     foreach $c_j$ in $C_i.Children$ do
6         Interact ($C_i$,$c_j$)
```

Algorithm 2. Interact(C_i, C_j)

```
1  if $C_i$ and $C_j$ are leafs then
2      P2P($C_i$,$C_j$)
3  else
4      if $C_i$ and $C_j$ satisfy MAC then
5          M2L($C_i$,$C_j$)
6      else
7          if SizeOf ($C_i$,$C_j$)> nspawn then
8              Spawn (DualTreeTraversal ($C_i$,$C_j$))
9          else
10             DualTreeTraversal ($C_i$,$C_j$)
```

4 NUMA-Aware Communication Reducing Algorithms

The local essential tree (LET) is the union of trees representing the entire domain as perceived by the local process. LET communication is known to be the major factor that hinders FMM's perfect scaling. References [1,8] describe specific communication protocols named $\mathcal{HSDX}$ and $\mathcal{NBX}$ respectively. They provide optimizations that are specific to distributed sparse data exchange, which generally suits the communication structure of FMM's global tree. We explore the effect of different communication strategies within the KNL chip. Note that we implemented all of these strategies on an ExaFMM branch [16]. Table 1 briefly highlights various techniques that we benchmark. Note that "Hierarchical" protocol means that the data is aggregated along a structured hierarchy such as graphs and trees, whereas "sparse-aware" protocol avoids direct communication with partitions without or with very little data to exchange (almost negligible). In the context of NUMA systems, hierarchical protocols tend to maximize locality of the data within each local caches, and in the case of data exchange, each process requires the data only from its neighboring MPI ranks. Hence, the communication is mostly localized inside the NUMA socket. However, if the required data happens to be in different NUMA socket, then MPI would communicate the cache line from the socket's memory, which is very negligible in proportion to locality maximizing communication protocols.

5 Results and Discussions

5.1 Experimental Setup

For KNL experiments, we used two Linux servers that run CentOS Linux 7.3.1611 Operating System. Both servers are powered by Intel Xeon Phi CPU 7210, which is equipped with 64 hardware cores that execute at 1.30 GHz clock frequency, and both have access to 116 GB of DRAM. The typical specifications of the KNL chip that we used here can be found in [10]. For Broadwell experiments, we used a Linux server that runs Ubuntu 14.04.5 LTS Operating System.

Table 1. MPI-based communication paradigms

Name	MPI calls	Complexity	Hierarchical	Sparse-aware
Alltoallv	MPI_Alltoallv	MPI specific	Yes	No
Hierarchical Alltoallv	MPI_Comm_Split MPI_Alltoallv	MPI specific	Yes	No
Point-to-Point	MPI_Isend MPI_Irecv MPI_Wait	$O(P)$	No	No
Hypercube	MPI_Comm_Split MPI_Isend MPI_Irecv	$O(\log P)$	Yes	Yes
$\mathcal{NBX}$	MPI_Ssend MPI_Srecv MPI_Ibarrier	$O(\log P)$	No	Yes
$\mathcal{HSDX}$	MPI_Distgraph_create MPI_Neighbor_alltoallv	$\Omega(\log P)$ $O(\log^2 P)$	Yes	Yes
One-sided	MPI_Win_create MPI_Get	$O(P)$	No	No

The server is powered by dual sockets of Intel Xeon CPU E5-2680 v4, each of which is equipped with 14 hardware cores that execute at 2.40 GHz clock frequency. Each socket has access to a single address space of size 64 GB of DRAM. Therefore, the server has a NUMA node of in total 28 hardware cores and 128 GB of DRAM. For KNC experiments, we used a Linux server that runs Scientific Linux release 6.4 (Carbon) Operating System. The server is powered by two Intel Xeon Phi 7120P coprocessors, each of which is equipped with 61 hardware cores that run at 1.238 GHz clock frequency, and each has access to 16 GB of DRAM. The typical specifications of the KNC chip that we used here can be found in [5]. The two KNC chips are hosted by a dual socket Intel Sandy Bridge E5-2670 CPU. Each socket consists of 8 hardware cores (in total 16 cores). The CPU clock speed is 2.6 GHz. Both sockets share a 64 GB DRAM (32 GB per socket). All of the experiments here were run with Intel Parallel Studio XE 2017 as the main software stack that comes with Intel ICC, MPI, TBB, OpenMP, and Cilk. The data sets are based on a single precision Laplace kernel with Cartesian coordinates, and the FMM order of expansion is set to 4. For the KNL results, all of the experiments are ran with `-xMIC-AVX512`, and for KNC, we use `-mmic` compiler option. For Broadwell, on the other hand, we use `-xHost` compiler flag. All of the experiments are compiled with `-O3` compiler optimization flag. All of the experiments here are summarized using the arithmetic mean of the CPU wall clock time across 10 independent runs, which forms the sample space, and an error bar is drawn to show the +/− standard deviation of the mean for each experimental sample.

5.2 SIMD Optimizations of the N-Body Kernel

The N-Body kernel is constructed with two nested `for loops`. The outer loop is the target loop and the inner loop is the source loop. We explore loop

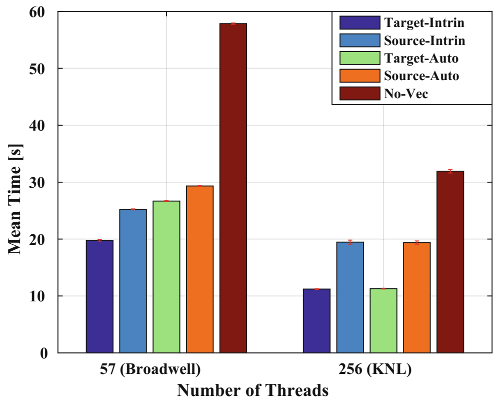

Fig. 1. Direct N-Body kernel running on two Intel architectures, KNL (quadrant cluster and flat memory modes) and Broadwell [Problem Size: 1 million Particles]

tiling on each loop, with 16 stride size for KNL. Therefore, in a single CPU cycle, each OpenMP thread fetches 16 bytes of data into the vector unit. In the case of two threads per core, each thread processes 16 bytes simultaneously utilizing the two vector units per core of KNL. However, if one thread per core is running, the next 16 bytes are pipelined in the second vector unit, and the thread scheduler alternates between them in a serialized manner, which keeps the core busy as much as possible. Furthermore, when the full number of threads per core are running, the threads are pipelined to process the data of both vector units. Therefore, with four threads per core, KNL utilizes both vector units and the pipelining potentials available in the out-of-core execution of the core's instruction pipeline. We observe that tiling targets' as opposed to sources' wins consistently in KNL; in each outer loop iteration, cache lines pertaining to elements in the target vector are loaded only once to AVX512 register using `_mm512_load_ps` intrinsic. This in turn does not require calling `_mm512_reduce_add_ps` after iterating over sources, which must be done otherwise because vectorizing effects of source fields must eventually be reduced to one value at target. Note that this kernel is run independent of FMM to explore the effect of the used techniques in detail, hence the chosen problem sizes are relatively small due to the quadratic compute and memory complexities. Figure 1 presents the performance of the N-Body kernel running on KNL comparing five different optimization techniques: (1) Target-Intrin: N-Body outer-loop tiling. (2) Source-Intrin: N-Body inner-loop tiling. (3) Target-Auto: outer-loop wrapping with `#pragma simd`. (4) Source-Auto: inner-loop wrapping with `#pragma simd`. (5) No-vec: scaler code.

We note that the handwritten vectorization does not improve much over auto-vectorization in KNL. It even appears that the ICC compiler was able to detect the event of reciprocal square root known as `_mm512_rsqrt28_ps`. Overall, vectorization benefits the kernel and shows significant improvements compared to

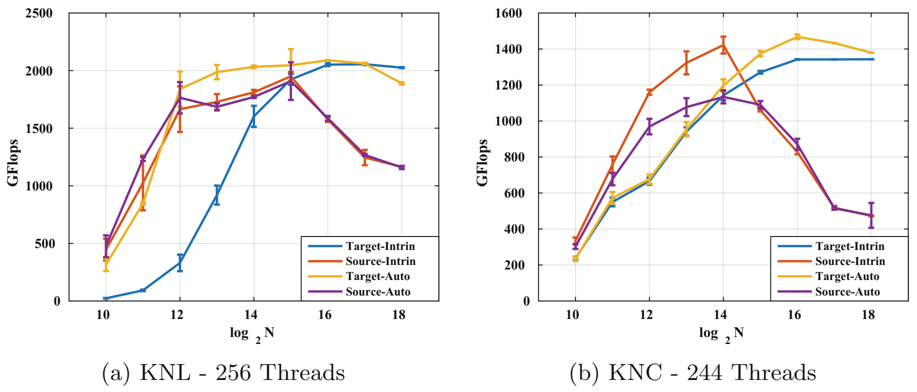

Fig. 2. FLOP/s performance across the 2 Intel manycore generations, KNL (quadrant cluster and flat memory modes) and KNC

the non-vectorized version of the code. This is not entirely the case in Broadwell; the variations are not proportional to their rivals in KNL, which suggests that outer-loop manual tiling cannot be avoided in Broadwell. We infer that KNL's AVX512 has a more sophisticated mechanism of matching correct vectorization than AVX2.

Figure 2 draws a comparison between floating point applicabilities of the 2 manycore generations by Intel, namely KNC and KNL, in terms of the aforementioned vectorization techniques. Error bars suggest reasonable stability in clocking frequency in both generations. Auto-vectorization in KNL reaches maximum FLOPs rate in an at least four times smaller problem, which strongly suggest that it utilizes local caches in a much more efficient manner. The drop in performance for slow versions happens exactly at the time when the performance of manual target vectorization saturates. This also suggests that the drop happens when prefetching and cache reuse could no more hide the overhead caused by source vectorization [5], which is 2^{15} in KNL (Fig. 2a) and 2^{14} in KNC (Fig. 2b).

5.3 Dual Tree Traversal with Task-Based Threading

Figure 3 shows traversal scalability using several threading libraries. The purpose of this test is to assess the DTT (Algorithm 1) performance using task-based/lightweight threading libraries on manycore architectures. Error bars are hardly observable, because frequency scaling has been disabled on KNL to stabilize performance. As expected, there is a general loss of scalability aspect when hyperthreading is enabled. Intel TBB perfectly scales up to 64 threads (1 thread/core). Scaling to the full chip, i.e., 256 threads, its relative speedup is $14, 94$, with an efficiency of 0.469, compared to 0.4249 in Intel Cilk and 0.1912 in OpenMP tasks. It is observed that there is a weak separation between user-level and OS-level threads in OpenMP tasks. This is due to the very marginal performance gain from enabling hyperthreading in OpenMP tasks [5] ($1.1\times$ speedup

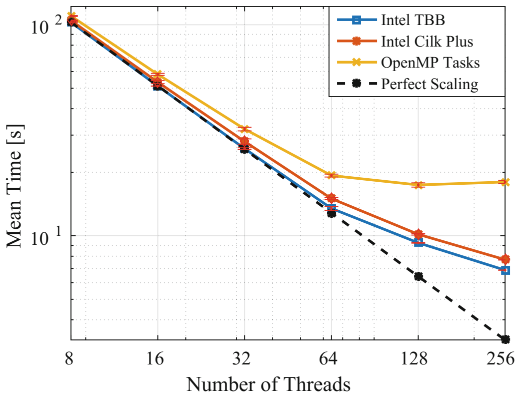

Fig. 3. Dual tree traversal using different task threading models. [Problem Size: 100 million Particles]

between 64 and 128 threads). Intel TBB, therefore, has the lowest task creation overhead, due to its efficient performance for heavily recursive tasks. However, Cilk does not seem to pose significant degradation in performance although it has minimal development time since it is integrated as a C++ language extension in modern Intel compilers.

5.4 Communication Reducing on KNL

As compute nodes are packaged with more low frequency cores, it is essential that MPI communication scales within main memory or across the NUMA sockets. Therefore, we apply various MPI communication reducing algorithms from Table 1 to FMM's tree communication. Results that are shown in Fig. 4 are executed with 64 MPI ranks, and a single thread per each rank, so that the effect of locality-maximizing behavior can be clearly observed.

$\mathcal{HSDX}$ (Distgraph) performs better that the others, and this is due to restricting exchanges to neighbors only, which makes it potentially NUMA-aware and yields acceptable on-chip performance. In other words, in $\mathcal{HSDX}$ algorithm, we tend to maintain a load balance between the KNL tiles, so that each tile acts like a sender and receiver of the cache lines. Thus, this model of communication prevents any long distance cache line transfer inside the chip, and maintains load balance of the cache line distributions across the tiles. To further prove this, we investigate this phenomena when we change the cluster mode of KNL. As you can see in Fig. 5, the $\mathcal{HSDX}$ the cluster modes of KNL do not have significant performance impact on the algorithm, and the performance differences between different modes are very negligible. Note that SNC-2 and SNC-4 modes are still experimental modes [11].

One-sided communication has a large overhead for shared window creation using `MPI_Win_create`, which requires soft locking prior to data access. This latency cannot be hidden when fetching sparse data either from the memory or

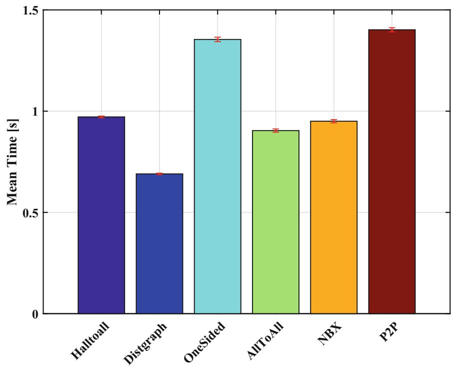

Fig. 4. Comparison of different MPI communication algorithms of LET communication kernel running on KNL (quadrant cluster/flat memory modes) [Problem Size: 80 million Particles]

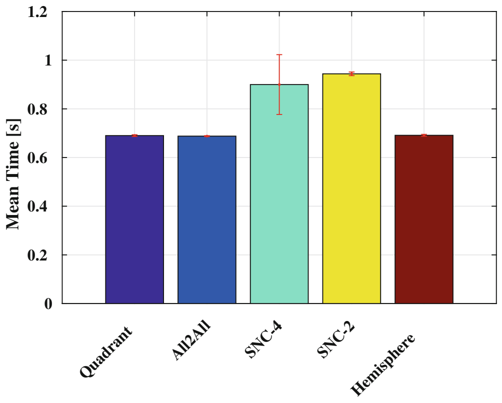

Fig. 5. Comparison of different cluster modes of KNL running Distgraph communication algorithm with 64 MPI ranks and 1 TBB Thread per MPI rank [problem size is 80 million particles]

from the other L2 caches. Even though, KNL has a great support for AVX512 prefetching instructions, locking the window before accessing the data imposes an implicit synchronization barrier on every data read. This creates a significant overhead on a cache-coherent systems.

6 Conclusion and Future Work

Facing manycore processors with high degree of fine-grained thread parallelism within a single shared-memory compute node, practitioners are now compelled to investigate strong thread scaling. In this paper, we present optimizations and thorough analysis of an FMM code on a modern high performance Intel manycore architecture, KNL. We extract the potential SIMD and thread-level parallelisms of three different computationally intensive kernels, namely P2P, tree traversal, and LET communication kernels. We demonstrate several shared-memory optimizations on these kernels, including different task-based threading paradigms, vectorization, loop tiling, and NUMA-aware communication-reducing. Our shared-memory optimizations present significant improvements that are reflected on the N-Body kernel compare to the out-of-the-box compilation of the non-optimized version. These feature in excess of 2.8x speedup on two Intel multi and many architectures, KNL and Broadwell. Furthermore, the task-based parallelism of the tree traversal kernel shows almost linear scaling, within a massively parallel single compute node, up to 64 thread contexts of KNL. With hyperthreading the performance gain becomes slightly monotonic. The NUMA-aware communication algorithm based on optimizing MPI alltoall communication protocol to maintain load balancing and shorter cache line transfers within a chip are explored. It is found that $\mathcal{HSDX}$ performs considerably faster than any other communication models; even across different cluster modes of KNL it still maintains marginally the same performance.

In the future, we plan to carry out a comprehensive comparison study across other FMM codes optimized for multi and manycore architectures. We are extending the study to include other x86 architectures, including IBM POWER8, and the bleeding edge release of Intel Xeon (i.e., Skylake). In addition, we are exploring multiple problem sizes to study the performance effects of workload variations on KNL. We are applying certain algorithmic optimizations to improve the performance of FMM on KNL, especially to overcome the stagnation and saturation of performance with hyperthreading enabled. To exploit the MCDRAM, we are working on optimizing the tree traversal kernels, which include the N-Body kernel. This is achieved by issuing simultaneous memory accesses throughout the kernel execution and utilizing the AVX512 prefetching instructions. Finally, we are building an extensive performance model to analyze the behavior of hybrid programming paradigms (MPI+TBB) on KNL. Multiple strategies are being developed to extract the best combinations of different programming models within a chip. These include thread/task pinning to a thread, core, tile, quadrant, and node, through low-level programming interfaces.

References

1. Abduljabbar, M., Markomanolis, G.S., Ibeid, H., Yokota, R., Keyes, D.: Communication reducing algorithms for distributed hierarchical N-body problems with boundary distributions. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) ISC 2017. LNCS, vol. 10266, pp. 79–96. Springer, Cham (2017). doi:10.1007/978-3-319-58667-0_5
2. AbdulJabbar, M., Yokota, R., Keyes, D.: Asynchronous execution of the fast multipole method using CHARM++. arXiv preprint arXiv:1405.7487 (2014)
3. Barnes, J., Hut, P.: A hierarchical $o(n \log n)$ force-calculation algorithm. Nature **324**(6096), 446–449 (1986)

4. Bédorf, J., Gaburov, E., Zwart, S.P.: A sparse octree gravitational N-body code that runs entirely on the GPU processor. J. Comput. Phys. **231**(7), 2825–2839 (2012)
5. Farhan, M.A.A., Kaushik, D.K., Keyes, D.E.: Unstructured computational aerodynamics on many integrated core architecture. Parallel Comput. **59**, 97–118 (2016). Theory and Practice of Irregular Applications
6. Greengard, L., Gropp, W.D.: A parallel version of the fast multipole method. Comput. Math. Appl. **20**(7), 63–71 (1990)
7. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comput. Phys. **73**(2), 325–348 (1987)
8. Hoefler, T., Siebert, C., Lumsdaine, A.: Scalable communication protocols for dynamic sparse data exchange. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 159–168. ACM, New York (2010). http://doi.acm.org/10.1145/1693453.1693476
9. Ibeid, H., Yokota, R., Keyes, D.: A performance model for the communication in fast multipole methods on high-performance computing platforms. Int. J. High Perform. Comput. Appl. **30**, 423–437 (2016)
10. Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming (Knights Landing Edition), 2nd edn. Morgan Kaufmann, Boston (2016)
11. Ramos, S., Hoefler, T.: Capability models for manycore memory systems: a case-study with xeon phi KNL. In: Proceedings of the 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS 2017). IEEE, May 2017
12. Warren, M.S., Salmon, J.K.: A fast tree code for many-body problems. Los Alamos Sci. **22**(10), 88–97 (1994)
13. Ying, L., Biros, G., Zorin, D., Langston, H.: A new parallel kernel-independent fast multipole method. In: 2003 ACM/IEEE Conference Supercomputing, p. 14. IEEE (2003)
14. Yokota, R.: An FMM based on dual tree traversal for many-core architectures. J. Algorithms Comput. Technol. **7**(3), 301–324 (2013)
15. Yokota, R., Abduljabbar, M.: N-body methods. In: Reinder, J., Jeffers, J. (eds.) High Performance Parallelism Pearls - Multicore and Many-Core Programming Approaches, Chap. 10, pp. 175–183. Elsevier, Amsterdam (2014). 1 edn
16. Yokota, R., et al.: ExaFMM (2016). https://github.com/exafmm/exafmm
17. Zandifar, M., Abdul Jabbar, M., Majidi, A., Keyes, D., Amato, N.M., Rauchwerger, L.: Composing algorithmic skeletons to express high-performance scientific applications. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015, pp. 415–424. ACM (2015)