Abstract

This whitepaper studies the execution speed of the Intel Xeon Phi coprocessor when running a molecular dynamics proxy application. We aim to describe how particular code transformations influence the performance of different phases in the application code. Results demonstrate the performance response to code transformations on a single accelerator. The whitepaper will also present weak and strong scaling projections, and evaluate the potential for exascale simulations.

1. Introduction

Physical constraints of exascale computing make it prohibitively expensive to attain through upscaling conventional architectures with complex processing cores. Various accelerator technologies provide means to address this issue, by offering architectures which provide simplified, parallel cores in greater numbers, but this creates challenges with adapting application software to effectively utilize their resources.

Scientific application software frequently embodies many years of development work, and encodes large amounts of domain expertise, which requires substantial effort to rewrite and optimize for any novel architecture. Simultaneously, the essential performance bottlenecks of programs within a particular domain often stem from similar software/hardware interactions, due to intrinsic properties of the data structures and computations that solve similar problems.

Proxy applications reduce the engineering effort required to estimate application performance on novel architectures, by emulating the essential computational and memory access behaviour of larger applications. This behaviour is contained within limited size programs, to facilitate experiments that collect performance characteristics relevant to their application domains.

In this paper, we describe experiences collected from optimizing the CoMD [1] proxy application for Intel Xeon Phi accelerators [2].

As CoMD emulates essential behaviour of a broad class of molecular dynamics applications, our results are of interest to developers and users of programs such as NAMD and LAMMPS, to complement their perspectives on scalable programming techniques for accelerators.

a* Corresponding author. E-mail address: benjambj@idi.ntnu.no
2. **Xeon Phi Architecture**

2.1. **Computational Core Properties**

Some distinguishing features of the Xeon Phi architecture are a high number of cores run at a relatively low clock frequency, each supporting fine-grained multithreading and wide vector operations. Optimizations which raise the aggregate computational throughput of an application thus depend on exploiting this thread- and register-level parallelism as fully as possible, as performance per core is inferior to conventional superscalar architectures. Each core is in-order with two pipelines, but multiple dispatch is limited to certain instruction combinations (it is for instance not possible to issue two vector arithmetic instructions simultaneously). Each core also has four thread contexts, with fine-grained interleaving. Delays in the processor decode stage prohibits issuing instructions from the same thread contexts in two successive clock cycles, so at least two threads must be used to saturate the computational capabilities of the processor core. On a cache miss the suffering thread is completely stalled, so more than two threads may also be useful to hide the delay from cache misses. Most vector instructions also have a 4-cycle delay or more, so the use of multiple threads may also reduce the number of hazard-induced stalls.

The computational powerhouse in each core is its 512-bit wide vector units, which are capable of performing 8 double-precision or 16 single-precision operations at once. In order to make this SIMD paradigm as applicable as possible, the core has its own Instruction Set Architecture (ISA) nicknamed IMCI (Initial Many-Core Instructions). This ISA includes the following special support for vector operations:

- **Masked vector operations**: The ISA exposes 16-bit mask registers, which can contain the result of boolean expressions. These registers can be used to convert control-dependencies to data-dependencies by making a vector operation only apply to a subset of the vector elements.
- **Swizzling**: Many of the IMCI vector instructions also support reordering the elements of vector operands on the fly. This makes it possible to reorder data and do computation in the same instruction. The reordering is not perfectly flexible. It can only change the location of four adjacent elements in the vector, in the same way for each four-element group, and only with one of seven predefined permutations.

The ISA also contains a subset of Intel 64 instructions for scalar operations, but its focus is on operations using the wide vector units.

2.2. **Memory Configuration**

In addition to the architectural support for parallel execution, utilization depends on maximizing the effect of fast, local storage, in the form of cache memory. This is common to all contemporary high-performance architectures, but the effect is more pronounced on the Xeon Phi since the in-order cores are less capable of hiding memory latency than out-of-order cores. The memory hierarchy of the Xeon Phi is illustrated in Figure 1.
Figure 1. Xeon Phi Interconnection fabric (from [3]).

Each core has a 32 KiB L1 data cache, a 32 KiB L1 instruction cache, and a 512 KiB L2 cache, all shared by all four thread contexts. The caches are 8-way set-associative to reduce cache conflicts among the threads. The access latency to the L1 cache is 1 cycle and to the local L2 cache it is 11 cycles, while the work by Fang et al. estimates the access latency for data in remote L2 caches to around 250 cycles and accesses to main memory at approximately 300 cycles [4]. The core has prefetching mechanisms to minimize the amount of exposed latency. There is hardware prefetching for sequential memory streams to the L2 cache. Prefetching to the L1 cache is done using software instructions, which can be executed alongside vector instructions in a superscalar fashion. The Xeon Phi coprocessor comes with various core configurations. The specimen used to conduct our experiments contain 57 cores, each running at 1.1 GHz. It has 12 memory channels, which gives it a maximum theoretical main memory bandwidth of 240 GB/s.

3. Application Characteristics

3.1. Structure of the Computation

CoMD simulates pairwise atomic interactions in materials with short range interatomic potentials, such as metals. It can be configured to use either Lennard-Jones (LJ) or Embedded Atom Method (EAM) potential functions, and uses a spatial decomposition with a cutoff radius for computing interactions between all pairs. Particle position determines its governing link cell, necessitating force interactions between atoms in neighbouring link cells, and particle migration. The key concepts and data structures are illustrated in Figure 2. Spatial decomposition in CoMD. The CoMD kernel operates in three dimensions, but our illustrations will use only two, for the sake of clarity.
This spatial decomposition maps to a Cartesian grid of MPI processes, with segments of the domain allocated to each process. Because atoms interact across segments, it maintains a halo region of cells belonging to neighboring processes, and performs border exchanges with periodic boundary conditions after each time step. In single-process cases, the periodic exchange is carried out using memory copy operations in place of MPI communication calls.

Performance is reported in terms of the rate of atomic updates [atoms/µs], and total run-time in number of seconds. The simulation can be configured with respect to the number of atoms in x, y and z directions, initial temperature, and the choice of inter-atomic potential. We generally observe that larger problem sizes tend to yield higher update rates, and initial temperature influences the amount of particle movement in the simulated material. The latter increases the amount of work in updating the data structures maintaining particle data, and may also increase the load imbalance between segments in the spatial decomposition.

3.2. Reference Problem Configuration

Our experiments are based on the hybrid OpenMP/MPI port of CoMD, as the Xeon Phi supports these programming models also without program modifications, although this results in modest performance. To establish a baseline for comparison, we initially observe performance figures of 1.2 atoms/µs and total run time of 39.2 seconds, for a 52x52x52 problem size at an initial temperature of 600 degrees Kelvin.

All following performance improvements are relative to this baseline, using average values over three runs. The variance between runs is at most 0.5 seconds. Experiments with initial temperature at 5000 Kelvin were also performed. This rarely impacts the efficiency of program transformations, but will be explicitly noted when it does.

The arithmetic intensity of the application is affected by the type of inter-atomic potential. Our experiments are carried out with the Lennard-Jones potential which is configured by default in the OpenMP version of CoMD, which this work is based on. The reader is advised that comparisons with the CUDA version of CoMD [5] must take into account how that port uses the EAM potential, which has a higher arithmetic intensity.
Figure 3. Initial run time distributions of the unmodified CoMD program. shows how run time is distributed among phases of execution for 3 different cubic problem sizes and the program is left unmodified. The identified stages of execution are:

- **commHalo**: transmission of buffers with atom data during border exchanges
- **commReduce**: MPI reduction operations used to accumulate scalars across MPI processes
- **force**: computation of interatomic forces
- **position**: update of particle positions
- **redistribute**: migrate particles that move to another link cell
- **atomHalo**: border exchange of atom information, discounting buffer transmission time

4. **Optimizations**

The optimization strategies we investigate can be classified into four categories, according to the architectural overview in Section 2:

- **Thread parallelism** optimizations focus on exploiting the available core count, as well as individual core multithreading capabilities.
- **Memory utilization** optimizations focus on adapting the access pattern of the application data structures to the Xeon Phi memory system.
- **Vectorization** optimizations focus on exploiting SIMD parallelism inherent in the computation.
- **Algorithmic modifications** investigate whether the computational steps can be reorganized to improve utilization without modifying the result.
4.1. **Thread Parallelism**

*Parallel Link Cell Updates*

Figure 3. Initial run time distributions of the unmodified CoMD program shows that the *redistribute* phase does not improve with increasing thread counts. The update can be parallelized over link cells, but this introduces potential race conditions. These race conditions can be prevented by using one lock per link cell, and locking both link cells involved in a particle migration during the update, as illustrated in Figure 4. A thread-safe link cell update.

![Figure 4. A thread-safe link cell update.](image)

This optimization yields a speedup of 1.44x when the initial temperature is 600K, and 1.45x when it is 5000K.

*Parallel Atom Halo Copy Buffer Operations*

Like the link cell update phase, the atomHalo phase is executed sequentially by default, and most of its run time requirement is spent in preparing buffers for transmission. Atom boundary information is sparsely distributed throughout the link cell data structure. To reduce message size, the information is transferred to a compact buffer, sent or copied, and finally, unpacked into the appropriate link cells upon reception. This final unpacking operation can be parallelized by splitting the received buffer, and ensuring correctness by using link cell locking to avoid race conditions, as illustrated in Figure 5.

![Figure 5. Parallel buffer unloading with thread-safe insertion.](image)

This optimization gives speedups of 1.17x in the absence of loop fission, and 1.11x in conjunction with loop fission.

*Loop Fission*

The load operation is somewhat more involved, since the location of an entry into a compact buffer depends on the count of elements to be placed before it. We parallelize the operation by first fetching the number of atoms in each link cell in parallel, and calculating the prefix sum for the link cell count buffer. The prefix sum is implemented sequentially, as parallelization did not yield any benefit. Based on the prefix sum, link cells can be processed in parallel, transmitting information from four separate arrays of atom ids, types, positions, and momentum, as illustrated in Figure 6. In order to attain speedup, it was necessary to copy these arrays in separate loops without barrier synchronization.

![Figure 6. Loop fission.](image)
Dynamic Load Balancing

The force computation phase is parallelized by assigning a portion of the link cells to each thread. When some link cells contain more atoms than others, this can lead to load imbalance. In order to mitigate this, we use the OpenMP dynamic scheduling policy on the main loop. This improves both scaling behaviour and performance, as shown in Figure 7.
The resulting speedup is 1.06x when the force kernel is not optimized, and 1.01x when the force kernel is optimized. Unfortunately, dynamic scheduling makes it hard to improve memory system utilization through traversal order improvements, as described in Subsection 4.2.

4.2. Memory Utilization

Traversal Order

The force computation uses data from neighbouring link cells. It is therefore possible to exploit memory locality by using a good link cell traversal order. Any sequential traversal order will visit a neighbour next, but a regular linear dimension traversal will only reuse neighbour data along one dimension. The Hilbert traversal order is depicted in Figure 8. The Hilbert traversal order, and will reuse data from neighbours along multiple dimensions.

![Hilbert traversal order](image)

Figure 8. The Hilbert traversal order.

Hilbert order is unfortunately only possible to compute when the dimension sizes are powers of two. There are other alternatives, such as the Peano order, which generalizes to dimensions where all sizes are odd numbers. The effect of implementing these two additional traversal orders was a speedup of 1.05x at best. Combined with dynamic load balancing, however, performance was unaffected by altering the traversal order.

Blocking

To ensure temporal locality when computing forces, the force kernel iterates over link cell pairs before atom pairs, as illustrated in Figure 9.

![Blocking](image)

Figure 9. Use of blocking to improve temporal memory locality.

If we reorder the two inner loops, the run-time increases by 1.8 seconds, suggesting that the blocking accounts for 4.3%. We experimented with gathering all the neighbour atoms in range into a compact list before computing forces, in order to have fewer remainder loops in the vectorization, as discussed in Subsection 4.3. Using blocking in this situation increases the amount of temporary storage required, since we build compact neighbour lists for all atoms in the current link cell simultaneously. Therefore, the run-time increases when we block on the neighbour boxes in this situation.

4.3. Vectorization

Automatic Vectorization Enabling

Simple syntactic source code transformations affect the efficiency of compiler-driven automatic optimizations, by enabling detection of vectorizable constructs. Unrolling loops which iterate per dimension, and adding the ‘restrict’ annotation to pointers with no aliases gives a speedup of 1.14x.
Active Data Compression

When calculating the force acting on a given atom from all the neighbours in a neighbour box, the effect of vectorization can be limited by the number of neighbours who are out of range and whose force contributions should be ignored. To mitigate this, we split the inner force computation loop into two, as illustrated in Figure 10.

![Figure 10. Splitting the inner loop of the force computation.](image)

This optimization yields a speedup of 1.06x.

All-neighbours Active Data Compression

We have also tried to gather all neighbours across all neighbour boxes before computing forces. This optimization gave a 1.18x (compared to a non-vectorized approach), and is thus less effective than the neighbour list approach discussed in the algorithmic modifications section.

Iteration Ordering

The compaction step does not have to store the atoms in order, since the force calculation is independent and the subsequent reduction is commutative. We therefore write a hand-vectorized implementation which uses as few shuffle operations as possible to gather x, y and z coordinates into vectors in some compatible order, and then computes distances without reordering them.

The compacted storage then stores the data for the atoms which are in range in the order which they are in the vector registers. This reduces the required amount of shuffling.

![Figure 11. Order-independent neighbour treatment reduces shuffling requirements in the vectorized force kernel.](image)

The optimization is illustrated in Figure 11, and yields a speedup of 1.05x.

Remainder Loop Vectorization

Because there are at most 64 atoms in a box, there are at most eight full iterations of the inner force calculation loop. In any non-full boxes, the remainder loop will therefore be at least 12.5 % of the work, when relevant.

Vectorizing remainder loops can be challenging, since it may be illegal to request full vectors of memory from the end of the data structure. In CoMD, however, we can exploit the fact that all atom boxes will have a size that is evenly divided by the vector length. Since the main data structure is also aligned at the vector length, we can
request three full cache lines from memory in the remainder loop without risking memory faults. After the distance calculation is performed, some of the elements must be ignored as they contain garbage data. We use a look-up table to get an appropriate mask for the number of elements present in the remainder loop. The code transformation is illustrated in Figure 12, and the attained speedup was 1.03x.

![Figure 12](image1.png)

Figure 12. Use of padding and masking to vectorize remainder loop.

**Combined Computation and Shuffling**

When using neighbour lists as described in Section 4.4, Pennycook et al. [6] recommend combining shuffling and calculation by using the Xeon Phi ISA swizzle capabilities. This reduces the amount of instructions required. Figure 13 illustrates the principle.

![Figure 13](image2.png)

Figure 13. Combined computation and shuffling by swizzling 2nd operand of vector addition.

We implemented this optimization for neighbour lists, observing that the original approach assumes padded structures, causing some unaligned load operations in their absence. We therefore observe a speedup of less than 1%.

**4.4. Algorithmic Modifications**

**Symmetric Force Updates**

Since force exchange is symmetric, it is possible to calculate this only once per atom pair. This increases the required synchronization logic, because each core also updates force contributions in atoms placed in other link cells. We implemented this optimization, and used per-link-cell locks to synchronize increments to the forces acting on atoms in the link cell. We store updates to each neighbor atom in a thread-local array, and commit the updates only when the neighbor cell is completely processed. This gives a speedup of 1.02x over a baseline vectorized implementation. However, if the force kernel has a compressed force calculation, the use of symmetric force calculation increases run-time. We attribute this to the necessity of also compacting the neighbor atom indices, leading to more time-consuming compaction. The stack memory use increases by 256 bytes compared to the compaction-free symmetric update, which may have a slight effect, although it is small compared to the 2048 bytes increased stack use from temporary neighbor force contributions required for the symmetric computation in the non-compacted case.

**Result Caching**

When compacting all neighbors before using them as described in the All-neighbors Active Data Compression subsection above, and blocking on neighbors, we can reduce the amount of work by caching the transposed
coordinates as suggested by Pennycook et al. [6]. This technique is illustrated in Figure 14, and gives a speedup of 1.05x.

Neighbour Lists

If we use neighbour lists, we need to check fewer neighbours per atom. This data structure enhancement is depicted in Figure 15.

Using this data structure gives a speedup of 1.48x when the temperature parameter is set to 600 Kelvin. When the temperature is higher there is more movement, and the neighbour list must be rebuilt more often. With the initial temperature set to 5000 Kelvin, the speedup is diminished to 1.45x.

5. Results

The obtained performance improvements are summarized in Figure 16, which refers to specific optimizations by the abbreviations listed in Table 1. Figure 16 reflects results obtained at 600K, comparing each optimization to the reference problem separately.
Table 1. Abbreviated optimization names.

<table>
<thead>
<tr>
<th>Abbreviated optimization name</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCU</td>
<td>Parallel Link Cell Updates</td>
</tr>
<tr>
<td>PAHC</td>
<td>Parallel Atom Halo Copy Buffer Operations</td>
</tr>
<tr>
<td>LF</td>
<td>Loop Fission</td>
</tr>
<tr>
<td>DLB</td>
<td>Dynamic Load Balancing</td>
</tr>
<tr>
<td>TO</td>
<td>Traversal Order</td>
</tr>
<tr>
<td>BL</td>
<td>Blocking</td>
</tr>
<tr>
<td>AVE</td>
<td>Automatic Vectorization Enabling</td>
</tr>
<tr>
<td>ADC</td>
<td>Active Data Compression</td>
</tr>
<tr>
<td>ANAC</td>
<td>All-Neighbours Active Data Compression</td>
</tr>
<tr>
<td>IO</td>
<td>Iteration Ordering</td>
</tr>
<tr>
<td>RLV</td>
<td>Remainder Loop Vectorization</td>
</tr>
<tr>
<td>SFU</td>
<td>Symmetric Force Updates</td>
</tr>
<tr>
<td>RC</td>
<td>Result Caching</td>
</tr>
<tr>
<td>NL</td>
<td>Neighbour Lists</td>
</tr>
</tbody>
</table>

Figure 17 shows the categories of optimizations, and their relative weight in the overall performance gain.

![Figure 17. Fraction of total improvement by optimization category.](image-url)
6. Communication and Scalability Considerations

Our code transformations have focused on single-accelerator performance. In order to attain exascale performance, however, a high number of such accelerators must be used in concert. In this section, we analyse the communication requirements and expected scalability characteristics of the proxy application. We focus only on the communication characteristics when the LJ potential is used. In CoMD, each MPI process independently calculates atom movements for the atoms in its own local domain. After each time-step, all the processes synchronize by sending border exchange messages. Each process sends six messages, one in each direction along each dimension. This is illustrated in Figure 18. The message contains 56 bytes of atom information per atom in link cells in the border of the local domain. Information is also sent for atoms which migrated to halo link cells in the most recent time step. If CoMD is run with a higher temperature, the atom movement will increase. Although more atoms move out of the local domain, more atoms also move out of border cells and further into the local domain. The communication requirements are therefore relatively unaffected by increases in temperature, as depicted in Figure 19a. However, atoms which move into edge cells are sent twice and atoms which move into corner cells are sent three times. The increased movement therefore leads to a small net increase in the number of bytes sent, which Figure 19b demonstrates.

Since processes only exchange information about atoms in the border cells, the communication requirements are determined by the number of atoms in these cells. The fraction of cells in the local domain which are border cells is given by \( B = \frac{\text{xy} + \text{xz} + \text{yz}}{\text{xyz}} \). For a given domain size, the total number of atoms is fixed. If we assume that the average atom count in a border cell is independent of the domain size, then the number of atoms in border cells will be proportional to \( B \), and so will the communication requirements. Figure 20 plots \( B \) for some CoMD problem sizes. The plot demonstrates how an increasing number of processes increases the fraction of border cells with a fixed domain size, and how an increasing problem size will reduce the fraction of border cells for a fixed number of MPI processes. The fraction of border cells is also reduced if the local domains form cubes instead of prisms, which is the effect that gives multiple points in the plot for a given problem size and MPI process count.
Strong Scaling

From our previous analysis of the fraction of border cells, we would expect the communication requirements to increase if we increase the number of processes for a fixed problem size. This is also what happens, as is demonstrated in Figure 21. The figure plots the total number of bytes sent for CoMD instances with various domain- and MPI dimensions. For a fixed problem size, the number of bytes sent grows similarly to the growth in the fraction of border cells, and confirms that as the local domain sizes shrink the amount of communication increases. This will limit the strong scaling potential of the application, in addition to the intrinsic limits that a fixed problem size enforces on the available parallel force calculation work.

Weak Scaling

Each process sends a constant number of messages each iteration. As long as the problem size is increased in proportion to an increase in the number of processes, so that the local domain size remains fixed, the communication requirements will not increase. As such, we can expect good weak scaling properties from the application. The latency may increase, depending on the topology of the node interconnect.
7. Conclusions

In this paper, we have examined the effectiveness of applying 14 different optimization techniques to the CoMD molecular dynamics proxy application, using the Intel compiler toolchain and with Xeon Phi accelerator technology. In addition to the common observation that application performance on this platform depends strongly on exploiting OpenMP threads and vectorization features, we also found that more than 1/4 of the overall optimization potential was realized by making modifications to algorithms and data structures that are specific to molecular dynamics. This suggests that adapting similar applications to many-core architectures for the purpose of running exascale simulations not only requires porting to an appropriate programming paradigm, but that it also mandates revision of design decisions pertaining to problem representation and algorithms.

References


Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU’s Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838.