Optimization Techniques for the Intel MIC Architecture. Part 1 of 3: Multi-Threading and Parallel Reduction

Ryo Asai and Andrey Vladimirov

Colfax International

May 29, 2015

Abstract

This is part 1 of a 3-part educational series of publications introducing select topics on optimization of applications for the Intel multi-core and manycore architectures (Intel Xeon processors and Intel Xeon Phi coprocessors).

In this paper we focus on thread parallelism and race conditions. We discuss the usage of mutexes in OpenMP to resolve race conditions. We also show how to implement efficient parallel reduction using thread-private storage and mutexes.

For a practical illustration, we construct and optimize a micro-kernel for binning particles based on their coordinates. Such a workload occurs in such applications as Monte Carlo simulations, particle physics software, and statistical analysis. The optimization technique discussed in this paper leads to a performance increase of 25x on a 24-core CPU and up to 100x on the MIC architecture compared to a single-threaded implementation on the same architectures.

In the next publication of this series, we will demonstrate further optimization of this workload, focusing on vectorization.

Table of Contents

1. Introduction

This paper begins a 3-part series of educational publications on performance optimization in applications for Intel Xeon Phi coprocessors. In this publication, we will focus on some aspects of thread parallelism implementation in the OpenMP framework.

Optimization of multi-threading on Intel Xeon Phi coprocessors requires controlling synchronization, exposing sufficient parallelism, tuning thread affinity, improving load balance and avoiding false sharing. An extensive discussion of these techniques is presented in our book, "Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, 2nd Edition" [2]. In this paper, we demonstrate one of these techniques: control of synchronization in parallel reduction with the help of thread-private containers.

2. Motivating Example: Particle Binning

To illustrate our discussion of multi-threading and race conditions, we will consider the problem of binning. Suppose that we have data coming from a simulation or from an experiment on particles moving in a cylindrical particle detector (see Figure 1). The particle positions are reported in polar coordinates, and our interest is to bin these particles into bins defined in Cartesian coordinates. Workloads like this may occur in particle physics (for example, to detect particle tracks — see [3]), in Monte Carlo simulations and also in statistics where data transformation and binning takes place.

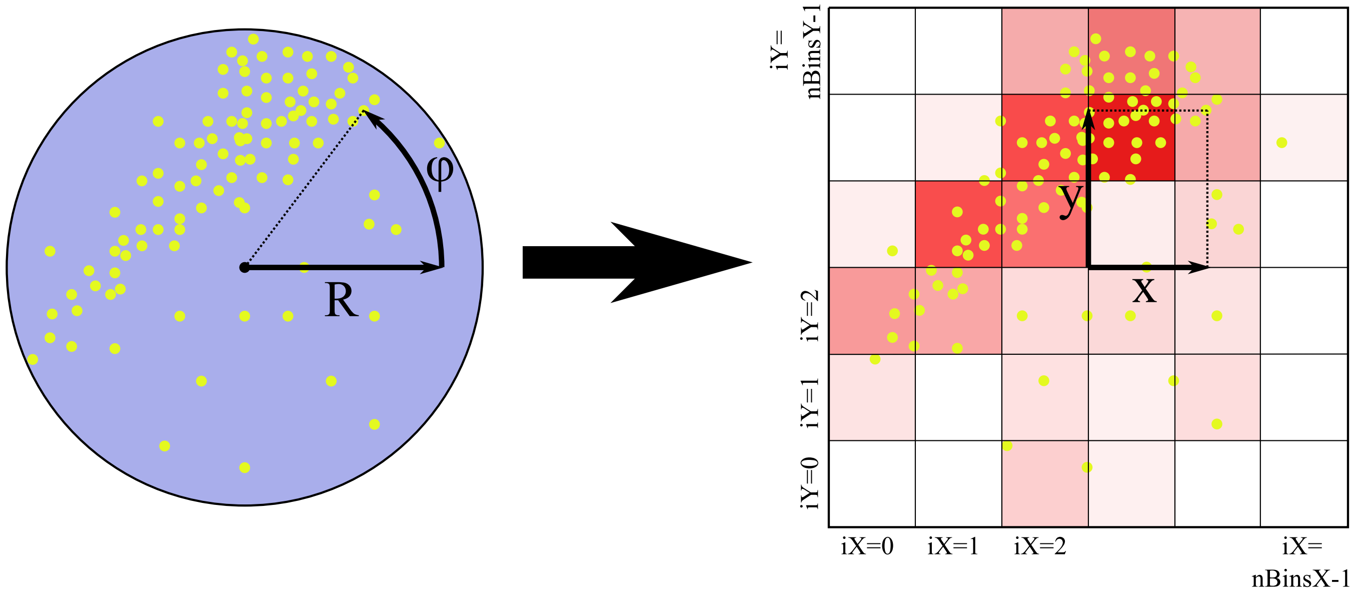


Figure 1: Workload illustration: take polar particle coordinates and compute particle counts in bins on the Cartesian grid.

For our specific problem, assume that the raw particle data comes in the form of a structure containing arrays `r` and `phi`. These arrays that contain the radii and the polar angles, respectively, of each particle. Listing 1 illustrates this data structure.

Our task is to compute the output data, which is a 2-dimensional array containing the counts of particles in the respective bins on a 2-dimensional Cartesian grid. Listing 2 illustrates the output data type.

```
struct InputDataType {
  int numDataPoints; // Using value 2^27
  FTYPE* r;    // FTYPE==real for single precision,
  FTYPE* phi; // FTYPE==double for double precision
};
```

Listing 1: Input data structure: polar coordinates of particles.

```
// Using nBinsX = nBinsY = 10
typedef int BinsType[nBinsX][nBinsY];
```

Listing 2: Output data structure: counts of particles in bins on the Cartesian grid.

We will assume `nBinsX=nBinsY=` 10 and `numDataPoints=` 2^{27} (a large enough number so that input arrays do not fit in the level-2 cache of the system).

A non-optimized scalar C code that performs the binning is shown in Listing 3. This code is not protected from situations when the converted `x` or `y` coordinates is outside the range (`xMin...xMax`) or (`yMin...yMax`), respectively. We assume that the user of the function is responsible for ensuring that the input arrays have only valid entries. For definitions of `COS` and `SIN`, see Section 4.2.

```
void BinParticlesReference(InputDataType  & inputData, BinsType & outputBins) {
  // Reference implementation: scalar, serial code without optimization

  // Loop through all particle coordinates
  for (int i = 0; i < inputData.numDataPoints; i++) {
    // Transforming from cylindrical to Cartesian coordinates:
    const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
    const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);

    // Calculating the bin numbers for these coordinates:
    const int iX = int((x - xMin)*binsPerUnitX);
    const int iY = int((y - yMin)*binsPerUnitY);

    // Incrementing the appropriate bin in the counter:
    outputBins[iX][iY]++;
  }
}
```

Listing 3: Non-Optimized binning code.

Code in Listing 3 is not optimal, because it does not use multi-threading to scale the application across multiple cores. On the 7100-series Intel Xeon Phi coprocessor, for example, this means that only one out of the 244 logical processors will be used by this application. Implementing thread parallelism is a pre-requisite for achieving good performance on massively parallel architectures one encounters in high performance computing platforms today. 1

3. OpenMP and Thread Parallelism

While there are many ways to run an application like in Listing 3 on multiple cores, we will use one of the most popular parallel frameworks: OpenMP. If you are not familiar with OpenMP, you can come to grips with it by continuing to read this paper. Additionally, a brief introduction to OpenMP may be found in our book [2], in Intel's video course [4], in the OpenMP standard specifications [5] or a number of tutorials available online and in print.

The OpenMP API consists of a set of directives, called pragmas, that the programmer inserts in the code to request running a piece of code in multiple threads. Threads are managed by the OpenMP runtime and usually one or two threads per core are used in Intel Xeon processors, and up to 4 threads per core in Intel Xeon Phi processors (MIC architecture).

The OpenMP pragma that creates threads is `#pragma omp parallel`. It can be combined with the pragma `for` to distribute the iterations of a loop across these multiple threads. One might try to use these pragmas to enable multi-threading as shown in Listing 4.

```
 1 void BinParticles_INCORRECT(InputDataType  & inputData, BinsType & outputBins) {
 2   // Parallel, but incorrect implementation with OpenMP
 3 
 4   // Loop through all particle coordinates
 5   // from multiple threads
 6 #pragma omp parallel for
 7   for (int i = 0; i < inputData.numDataPoints; i++) {
 8     // Transforming from cylindrical to Cartesian coordinates:
 9     const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
10     const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);
11 
12     // Calculating the bin numbers for these coordinates:
13     const int iX = int((x - xMin)*binsPerUnitX);
14     const int iY = int((y - yMin)*binsPerUnitY);
15 
16     // Incrementing the appropriate bin in the counter...
17     // PROBLEM! Race condition in this line.
18     // Code produces incorrect and unpredictable results
19     outputBins[iX][iY]++;
20   }
21 }
```

Listing 4: Binning code. Parallel, but incorrect implementation: race condition in line 19.

The execution of this function will begin in one thread. When the code reaches the line with the `#pragma`, multiple threads will be spawned. By default, as many threads are created as there are logical processors in the system. For example, in an Intel Xeon Phi 7120P coprocessors, 244 threads are created. Threads will be distributed across cores, and therefore more computational resources will be available to the application. The iterations of the loop will be distributed across threads, and therefore the code execution time may be shortened.

However, this code has a big problem: it produces incorrect and unpredictable results. The next section gives more details on the situation.

1

3.1. Race Conditions and Mutexes

In applying thread-parallelism to an application, one of the most common pitfalls that software developers encounter is what is called a “race condition”. Race conditions occur when multiple threads attempt to access the same memory address concurrently and one or more of these accesses is a write operation. The order of these accesses cannot be guaranteed at runtime, and therefore the code may produce incorrect results, which will be different from run to run.

Figure 2 (left panel) demonstrates schematically how race conditions occur. When two threads must increment a variable shared between them, the two threads may simultaneously read the same state of the variable. In such a case, only the write of the slower thread survives; the first write is lost. Note that race conditions will not make the application crash; they will simply produce incorrect results.

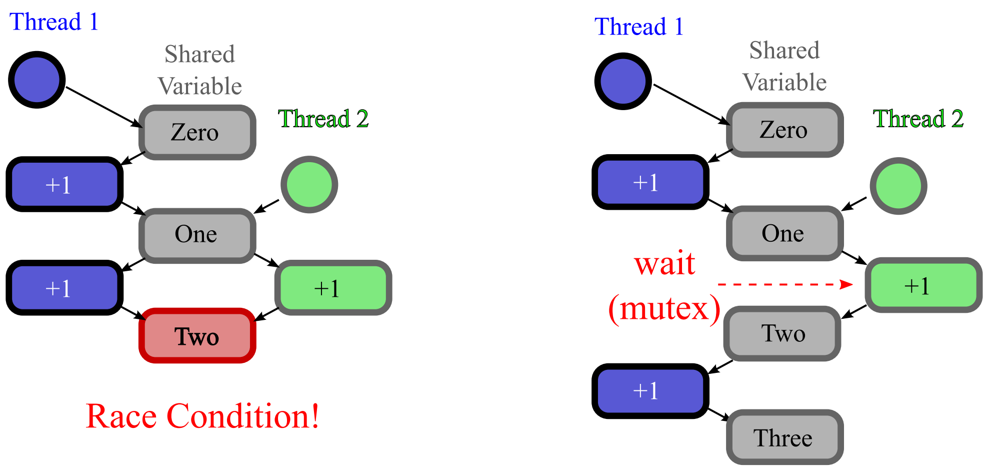


Figure 2: *Left*: Schematic illustration of race conditions. The result must be three, however, because of the race condition, the result is two. *Right*: Usage of a mutes resolves the race condition.

To ensure that every write to a shared variable is respected, we can use a thread synchronization construct called a “mutex” (mutually exclusive events). A mutex can limit certain operations to be carried out only by one thread at a time. Mutexes resolve race conditions because we can guarantee that once a thread, say thread 2, reads a shared variable, the shared variable is not read by any other thread until the write operation by thread 2 is completed. This is illustrated in Figure 2 (right panel).

OpenMP supports two types of mutexes: critical sections and atomic operations. Critical sections are created by using `#pragma omp critical` inside the parallel region, and placing the protected code in the scope of this pragma. Only one thread at a time will be able to execute this code. Atomic operations are executed by using `#pragma omp atomic` before certain operations (such as sums, multiplications, increments, assignments, reads, etc.) on scalar variables.

Listing 5 demonstrates the usage of these mutexes. The advantage of the critical section is that it can be used for any instruction, including user-defined functions, and can mark a whole block of code to be executed by only one thread at a time. Its drawback is that it is more expensive than an atomic operation. On the other hand, the drawback of `omp atomic` is that it is only supported for certain operations on scalar variables, and can only protect the line immediately following the pragma (see details in [2, 4, 5]).

1

```
int sharedVariable = 0;
#pragma omp parallel for
for (int i = 0; i < n; i++){
#pragma omp critical
  { // Heavy-weight critical section mutex:
    sharedVariable++;  // Only one thread at a time will enter this scope
  }
// Alternatively, a light-weight atomic mutex:
#pragma omp atomic
  shared_variable++;  // Only one thread at a time increments shared_variable
}
```

Listing 5: Usage of `#pragma omp atomic` and `#pragma omp critical`.

Let's see what happens if we protect the race condition in the increment operation in our binning code with an atomic construct as in Listing 6.

```
void BinParticles_0(const InputDataType  & inputData, BinsType & outputBins) {
  // Thread-parallel implementation with atomics

  // Loop through all particle coordinates with multiple threads
#pragma omp parallel for
  for (int i = 0; i < inputData.numDataPoints; i++) {

    // Transforming from cylindrical to Cartesian coordinates:
    const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
    const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);

    // Calculating the bin numbers for these coordinates:
    const int iX = int((x - xMin)*binsPerUnitX);
    const int iY = int((y - yMin)*binsPerUnitY);

    // Incrementing the appropriate bin in the counter,
    // protecting race condition with atomics
#pragma omp atomic
    outputBins[iX][iY]++;
  }
}
```

Listing 6: Binning code. Parallel and correct, but inefficient implementation.

Now our code is parallel, and it produces correct results. However, as performance measurements show (see Section 4.2), the performance of this code is unsatisfactory. On the CPU we did not gain any performance over the single-threaded implementation (Listing 3). On an Intel Xeon Phi coprocessor we achieved a speedup up to 12x, however, this is much less than the number of cores in the coprocessor.

The problem is that mutexes make the results correct at the cost of serializing the application. Even a small fraction of serialization in the context of running 244 threads on the Intel MIC architecture leads to very steep performance penalties (see Amdahl's law [6]). To optimize the code, we have to use thread synchronization sparingly. This can be done with a parallel reduction implementation shown in Section 3.2.
1

3.2. Parallel Reduction

Parallel algorithms that require synchronization only to modify a common quantity can be expressed in terms of parallel reduction. This is possible when the operation with which the common quantity is modified is *associative*, i.e., the order of operations does not affect the result. OpenMP has built-in functionality for reduction on scalar variables in the form of `reduction` clause (see, e.g., [2, 4, 5]). However, in the case of particle binning, we are reducing into a 2-dimensional array, so we will demonstrate an alternative approach. We will instrument a reduction algorithm using private variables and minimal synchronization.

With the private variable method, each thread must have a private variable of the same type as the global reduction variable. In each thread, the reduction operation is applied to that private variable without synchronization with other threads. At the end of the loop, critical sections or atomic operations are used to reduce (i.e., accumulate) the values of the private variables from each thread into the global variable. Figure 3 illustrates this approach.

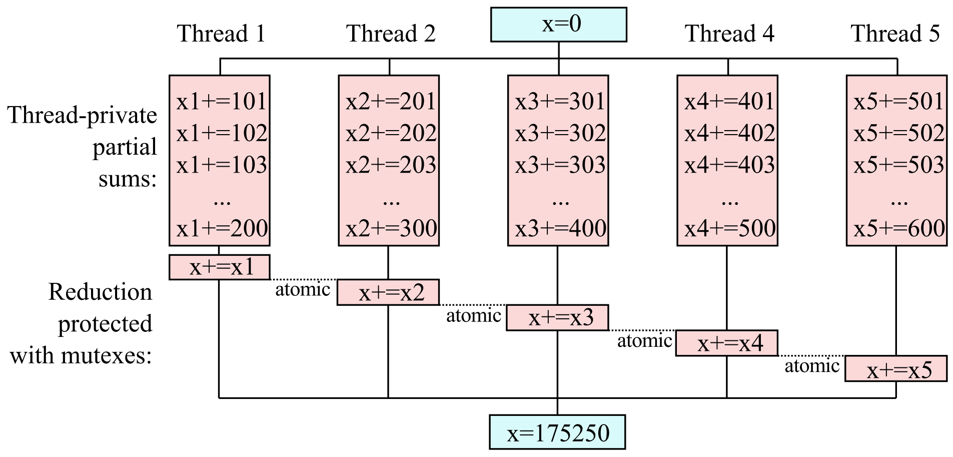


Figure 3: *Left*: Schematic illustration of race conditions. The result must be three, however, because of the race condition, the result is two. *Right*: Usage of a mutes resolves the race condition.

Code implementing parallel reduction with private variables schematically looks like in Listing 7.

```
int sharedVariable = 0; // We want the result to go here
#pragma omp parallel
{
  privateVariable = 0; // Each thread gets a private copy of this variable
#pragma omp for
  for (int i = 0; i < n; i++)
    privateVariable++; // No race condition: each thread writes its own copy

 // Still inside of parallel region. Reducing partial sums into sharedVariable
#pragma omp atomic
   sharedVariable += privateVariable;
}
```

Listing 7: Implementation of parallel reduction with thread-private containers.

Obviously, we still have the construct `#pragma omp atomic` in the code. However, this construct is outside of the innermost loop. So atomic operations in Listing 7 will be performed only T times, whereas if we did it as in Listing 5, atomic operations would be performed n times. Here T is the number of threads and n is the number of loop iterations. Therefore, parallel reduction discussed here works well only if the loop count n is much greater than the number of threads T. This condition is certainly true for our binning code for an Intel Xeon Phi coprocessor where T is up to 244 and $n = 2^{27}$.

Listing 8 shows an implementation of the binning algorithm where we use parallel reduction.

```
 1 void BinParticles_1(const InputDataType  & inputData, BinsType & outputBins) {
 2   // Thread-parallel implementation. Race conditions are avoided by using
 3   // parallel reduction with thread-private containers
 4 #pragma omp parallel
 5   {
 6     // Declare thread-private containers for bins
 7     BinsType threadPrivateBins;
 8     for (int i = 0; i < nBinsX; i++)
 9       for (int j = 0; j < nBinsY; j++)
10         threadPrivateBins[i][j] = 0;
11 
12     // Loop through all bunches of particles
13 #pragma omp for
14     for (int i = 0; i < inputData.numDataPoints; i++) {
15       // Transforming from cylindrical to Cartesian coordinates:
16       const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
17       const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);
18 
19       // Calculating the bin numbers for these coordinates:
20       const int iX = int((x - xMin)*binsPerUnitX);
21       const int iY = int((y - yMin)*binsPerUnitY);
22 
23       // Incrementing the appropriate bin in the thread-private counter:
24       threadPrivateBins[iX][iY]++;
25     }
26     // Reduction outside the parallel loop
27     for(int i = 0; i < nBinsX; i++)
28       for(int j = 0; j < nBinsY; j++) {
29 #pragma omp atomic
30         outputBins[i][j] += threadPrivateBins[i][j];
31       }
32   }
33 }
```

Listing 8: Binning code. Parallel and correct implementation using reduction with thread-private containers.

Just like in Listing 7, we start the parallel region and inside of it declare a container for the partial sums. Such a container is created in every thread. Inside the main loop in Line 14 we bin particles into these thread-private containers. There are no race conditions here because each thread writes into its own region of memory. After the main loop, we reduce the partial sums into the global sum and protect these increment operations with an atomic mutex. 1

4. Performance

4.1. System Configuration

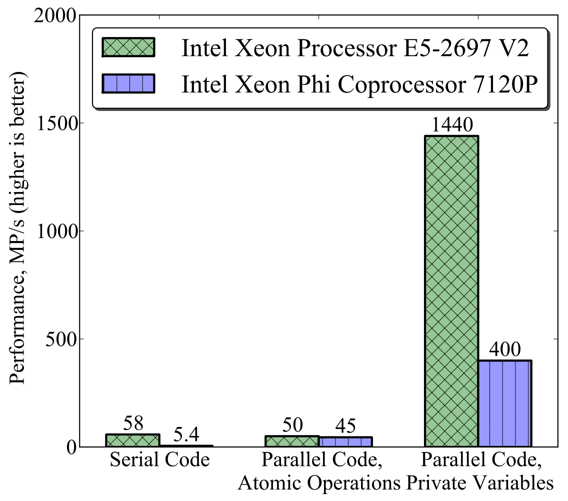
All of the benchmarks presented in this section were taken on a Colfax ProEdge™ SXP8600 workstation based on a dual-socket Intel Xeon E5-2697 v2 processor (12 cores per socket, 24 physical cores with two-way hyper-threading). The Intel Xeon Phi coprocessor benchmarks presented in this section were taken using native execution on 7120P Xeon Phi coprocessor (61 physical cores with four-way hardware-threading) installed in that system. The Xeon Phi benchmarks are taken using the coprocessor alone, i.e., the CPU was not computing in tandem with the coprocessor. For compilation we used the Intel C++ compiler version 15.0.3.187 on a CentOS 7.0 Linux OS with Intel MPSS 3.5.

4.2. Performance Benchmarks of Particle Binning

We benchmarked three versions of the code: the serial code from Listing 3, the parallel code with atomic constructs from Listing 6 and the parallel code with reduction implemented using thread-private containers from Listing 8. For each benchmark the binning workload was repeated 10 times, and the average of the latter 8 iteration is reported. We chose to exclude the first two iterations because they tend to be much slower due to various initialization overheads on both Xeon CPUs and Xeon Phi Coprocessors. They do not represent sustained performance that is achieved if the application has a long execution time.

For all benchmarks, we used either double precision or single precision. For double precision, we had `FTYPE==double`, `SIN==sin` and `COS=cos`. For single precision, we had `FTYPE==real`, `SIN==sinf` and `COS==cosf`. Those are single-precision implementations of the sine and cosine functions, respectively. Note that in C++, `sin` and `cos` are overloaded only in namespace `std`, but not in the global namespace. Therefore, for single precision, the correct functions are either `sinf` and `cosf`, or `std::sin` and `std::cos`.

Figure 4 reports the performance benchmarks of the binning algorithms. The performance is measured in MP/s, where 1 MP/s is 10^6 particles binned per second.



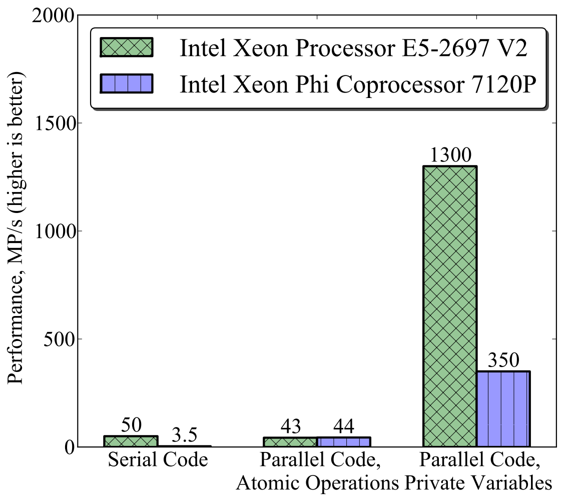


Figure 4: Benchmarks of the binning algorithm in single precision (left) and double precision (right). Optimization process is not complete. Refer to Part 2 for additional techniques that benefit the Intel Xeon Phi coprocessor performance.

5. What's Next

5.1. What We Learned

In this paper we introduced the parallel framework OpenMP that allows to add thread parallelism to applications for Intel Xeon processors and Intel Xeon Phi coprocessors featuring the MIC architecture. We discussed the concept of race conditions in thread-parallel applications and their resolution with mutexes. Then we introduced an optimization technique for parallel reduction which uses mutexes sparingly to achieve high parallel scalability.

We illustrated the optimization technique on an example that performs binning of particles based on their coordinates. The implementation with parallel reduction achieved a speedup of 25x on our 24-core Intel Xeon processor and a speedup of 72x (in single precision) and 100x (in double precision) on our 61-core Intel Xeon Phi coprocessor. These results show good parallel scalability for a code that is bound by the arithmetic throughput of the processor. Speedup in excess of 61x on the 61-core Xeon Phi is expected: each core in the coprocessor has 4 hardware threads that work in the dual-issue mode. That is, each hardware thread issues an instruction only every other cycle. Therefore, our baseline case, which is a single-threaded application, uses only 50% of the performance potential of one core.

5.2. Vectorization

Looking at the last sets of bars in the plots in Figure 4, one may wonder why the Intel Xeon Phi coprocessor yields less performance than an Intel Xeon processor. The answer is that the optimization process is not complete. While we achieved thread parallelism to utilize all cores, we do not yet use all the computational resources inside of each core. To get there, we must vectorize the code.

In Part 2 of this series of papers we will discuss an optimization technique called strip-mining that will help us to expose data parallelism in the code to the compiler and vectorize the binning application.

5.3. Memory Allocation for Thread-Private Variables

The containers for thread-private storage in our example were placed on the stack. Each thread in OpenMP has its own stack, and there is no penalty for simultaneously creating multiple containers on stacks of multiple threads. However, if we were to use heap allocation (e.g., using the call `malloc` or `new`), that could have resulted in a performance hit. That is because the heap address space is shared between all threads, so `malloc` serializes execution in order to maintain thread-safety. If the amount of parallel work is not very large, the overhead of serialization for heap memory allocation can ruin performance in user applications.

The same applies to allocatable arrays in Fortran. One must avoid using the call `ALLOCATE` inside parallel regions for the purpose of creating thread-private storage, because if the array is big enough, it will be allocated on the heap, incurring serialization overhead. This effect will be particularly heavy in the Intel MIC architecture where cores are numerous and run at a low clock speed.

But what if we had to allocate a storage container that is too large to fit on the stack? And what if we wanted to use the information in individual thread-private containers after the end of the parallel region? The short answer is that it can be done using a single large shared container with "private compartments" for each thread, but it is important to control false sharing with this approach. Read part 3 of this series of papers for more details.

5.4. Reduction Algorithms

We must note here that the sequential reduction algorithm that we used in Listing 8 (as illustrated in Figure 3) is easy to implement, but it is not the most efficient one. It completes in $O(T)$ time. There exist tree-based algorithms that take advantage of parallelism to complete in $O(\log T)$ time (e.g., [7]).

5.5. Variable Sharing and Fortran

In Listings 7 and 8 we declared thread-private containers inside the scope of `#pragma omp parallel`. This method works well in C and C++, however, in Fortran, where variables must be declared at the beginning of a function or subroutine, it cannot be done. It is still possible to have thread-private variables in Fortran, and it can be achieved by using the clause `private` or `firstprivate` in `#pragma omp parallel`. See, e.g., [2, 4, 5] for details.

5.6. Closing Words

If you found this publication useful, stay tuned for Part 2 where we talk about implementing vectorization in our particle binning code, and for Part 3 where we discuss another common pitfall in parallel applications: false sharing. These publications will be posted in the next couple of weeks at research.colfaxinternational.com. Code used for benchmarks will be posted together with the last publication in the series.

The topics discussed in these 3 publications are only a subset of optimization subjects necessary for efficient, future-proof applications for Intel architectures. For a thorough guide to optimization on Intel Xeon Phi coprocessors, see our book [2].

References

[1] Ryo Asai and Andrey Vladimirov. Optimization Techniques for the Intel MIC Architecture. Part 1 of 3: Multi-Threading and Parallel Reduction, 2015 *(landing page for this paper)*.
http://research.colfaxinternational.com/post/2015/05/29/Techniques-1of3.aspx.

[2] Andrey Vladimirov, Ryo Asai, and Vadim Karpusenko. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. Colfax International, 2nd edition, March 2015.
http://www.colfax-intl.com/nd/xeonphi/book.aspx.

[3] Parallel Computing in the Search for New Physics at LHC.
http://research.colfaxinternational.com/post/2013/12/02/LHC.aspx.

[4] Tim Mattson. Intel OpenMP Videos.
https://www-ssl.intel.com/content/www/us/en/education/university/intel-many-core-curriculum-list/openmp-videos.html.

[5] OpenMP Specifications.
http://openmp.org/wp/openmp-specifications/.

[6] Wikipedia, the free encyclopedia. Amdahl's law.
http://en.wikipedia.org/wiki/Amdahl%27s_law.

[7] Michael McCool, Arch D. Robinson, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.
http://parallelbook.com/.

1