CHAPTER 9

Xeon Phi Application Design and Implementation Considerations

Parallel programming on any general-purpose processor such as Intel Xeon Phi requires careful consideration of the various aspects of program organization, algorithm selection, and implementation to achieve the maximum performance on the hardware.

One of the key challenges in developing a parallel code for the Xeon Phi coprocessor is to develop and parallelize offload computations on as many as 244 threads to exploit the massive parallel computing power provided by the coprocessor. To achieve that degree of parallelism, application developers often face the following challenges:

- *Parallelization overhead*: OpenMP overhead to start and stop threads can kill the performance if there are not enough computations to amortize thread creations.
- *Load imbalance*: Load imbalance can reduce parallel efficiency drastically. If one or more of the cores are given more workload than other cores in the system, these cores will become the bottleneck as the rest of the system waits for these cores to finish processing before the whole program can end.
- *Shared resource access overhead*: Because so many threads or processes are working in parallel, they can easily be bottlenecked by concurrent memory or ring bus access latency and bandwidth.
- *Data and task dependency*: The large number of tasks and data divided among the cores causes dependencies and serialization that may not be present in the original serial version of the code. By Amdahl's law, discussed in the next section, these dependencies add to the serial component of the application execution time.

In order to counter these issues, the application developers need to manage data across the threads with minimal sharing, align data structures for vector operations for optimal compiler code generations, select algorithms to exploit higher flops or bytes, and balance workloads across MIC or Xeon and Cluster for load balancing.

Figure 9-1 shows various considerations you need to take into account to develop code for Intel Xeon Phi. These are broadly categorized as:

- *Workload considerations*: You must determine what size and shape of the workload fits into Xeon Phi and how to divide it among the computing devices, such as the host processor and one or more Xeon Phi coprocessors, so they can complete the job in minimal time.
- *Algorithms and data structures*: Parallel algorithm development needs a very different mindset in thinking about expressing the computational problem. Although our brains function in parallel, our thought processes as traditional programmers have developed in a serial fashion to be tractable. Parallel programming needs different ways of thinking about expressing

the same problem we ordinarily express serially. The algorithm that has been optimized for serial operation may not perform as well in parallel and may require a very different algorithm and data structure to solve the same task.

- *Implementation*: Massively parallel implementation on Xeon Phi needs support from programming languages, tools, and libraries. Because there are various way to parallelize, choosing the right parallel language suitable for programming the hardware for a specific problem will depend on the programmer, the problem at hand, and the specific library supports to perform optimally on the hardware. You can use *single program multiple data* (SPMD), loop structure, or thread fork joint models to do the parallel code implementation.

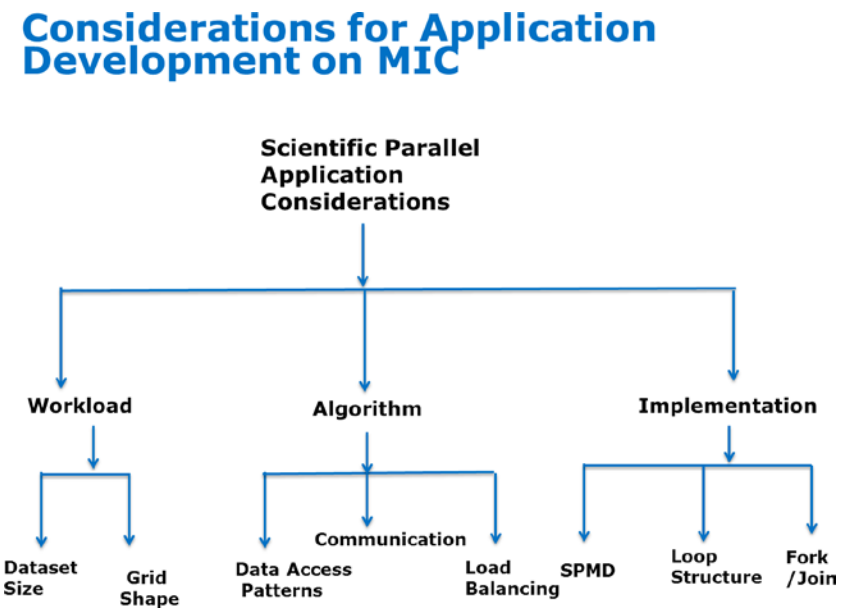

Figure 9-1. *Scientific parallel application development considerations*

This chapter will explain each of these broad categories and share some ideas on how to handle each challenging issue. These aspects of programming can in themselves fill up a book, but the goal here is to introduce the possible aspects of software development on Xeon Phi so you can pursue them further on your own.

Workload-Related Considerations

The first step toward thinking about, developing, and running applications on a parallel computer is to think about the size of the problem that will be run on the machine. Amdahl's law, which is familiar to many parallel application developers, states that the amount of parallelization that can be achieved by parallelizing an application is limited by the fraction of the code that runs serially during the whole application execution period.[1] Serialization is necessary when starting up the program, using a common resource such as writing to a single file and data consistency.

[1]Gene Amdahl, "Validity of the Single-Processor Approach to Achieving Large-Scale Computer Capabilities." *AFIPS Conference Proceedings* 30, 1967, pp. 483–485.

Amdahl showed that the speedup that can be achieved by a parallel program on N number of processors can be given by the expression $(s + p)/(s + (p/N))$, where s is the serial portion, p is the parallel fraction, and N is the number of parallel processors. If we use single processor runtime to normalize, which is the total runtime on a serial computer, we can define speedup as:

$$Speedup = 1/(s + (p/N)) \qquad \text{[Eq. 9-1]}$$

Equation 9-1 indicates that the serial portion s will dominate in massively parallel hardware such as Intel Xeon Phi, limiting the advantage of the manycore hardware. According to the formula, as N tends to infinity, the parallel portion of the computation can be completed in no time. So the application cannot be made infinitely fast by throwing more computing resources at the problem or by adding more computational cores. This was a big concern for computer scientists, and there was a prevalent feeling at that time in academia and industries that it is not a good return on investment to create parallel computers with more than 200 general-purpose processing nodes.

In 1985, Alan Karp of the IBM Palo Alto Research Center challenged the scientific community to demonstrate a speedup of at least 200 times on a general-purpose *multiple instructions and multiple data* (MIMD) machine on three scientific apps. John L. Gustafson and his colleagues showed that it is possible to achieve such performance on parallel computers.[2] The solution boiled down to better understanding of the problem characteristics that will be solved by these types of computers. Such problems follow Amdahl's law, but the problem size gets bigger on larger machines, such that the parallel portion of the workload keeps on increasing with the addition of more processors. This causes the serial section to become smaller and smaller, hence maintaining scalability. Gustafson et al. proposed a scaled speedup that represents scientific computing problems where data size is not fixed. As more processors are thrown into the mix, even larger problems can be solved.

Amdahl's law predicts the speedup curve as a function of the serial fraction, as shown in Figure 9-2, so that a maximum speedup of 24x is achieved with a serial fraction of only 4 percent.

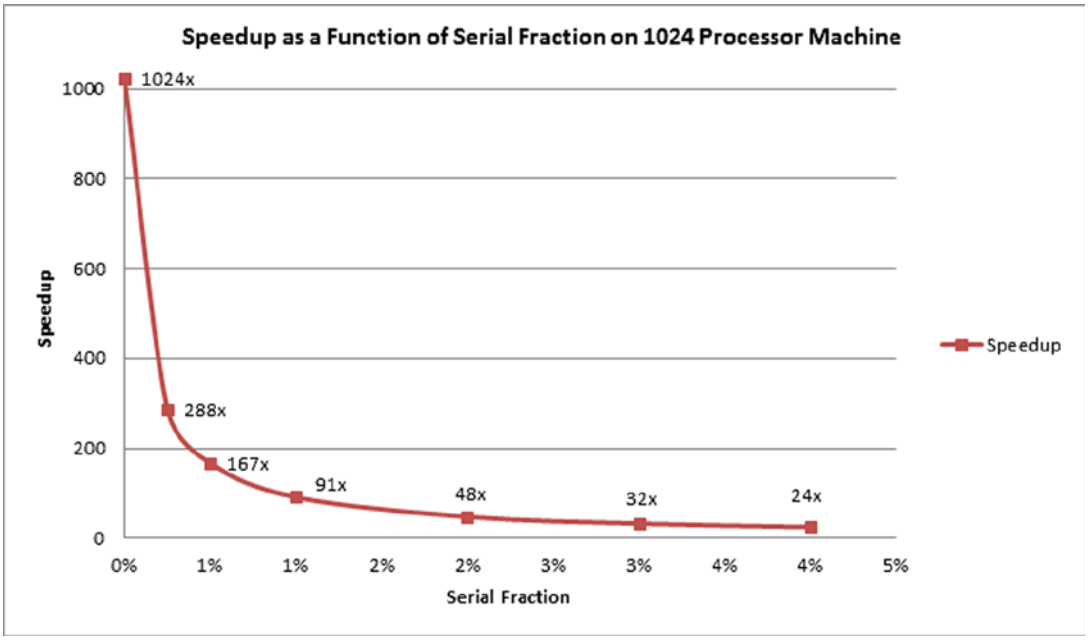

Figure 9-2. *Amdahl's law applied to a 1024 processor parallel MIMD machine*

[2]John L. Gustafson, Gary R. Montry, Robert E Benner, and C. W. Gear. "Development of Parallel Methods for a 1024-Processor Hypercube." *SIAM Journal on Scientific and Statistical Computing* 9(4), 1988, pp. 609–638.

We might infer from the highly parallel workload associated with a 4 percent serial fraction that it would be wasteful to build any machine with a 1024 processor. To understand how Amdahl's law may be misleading if generally applied to all parallel computing problems, let's apply it to a general scalability problem and see how Gustafson's law can provide a clearer idea of the value of parallel computation. The next section discusses further the important consideration identified by Gustafson and his team for understanding and practicing parallel application performance on these massively parallel machines.

Gustafson's Law

Gustafson and his Sandia colleague Edwin H. Barsis provided a speedup model that removed the barrier against parallel computation posed by Amdahl's model.[3] Gustafson showed that, for three practical applications with serial fraction (s = 0.4 - 0.8), his team was able to achieve more than 1,000x speedup on 1024 MIMD machines. In fact, the parallel portion of the "work done" in real applications scales with the number of processors, causing the overall serial fraction to be greatly reduced. Barsis proposed an inversion of Amdahl's model. Instead of asking how long a serial program will take to run on a parallel machine, he asked how long it will take a parallel program to run on a serial machine, and he came up with the definition of scaled speedup, which is given in the next section and shown in Figure 9-3.

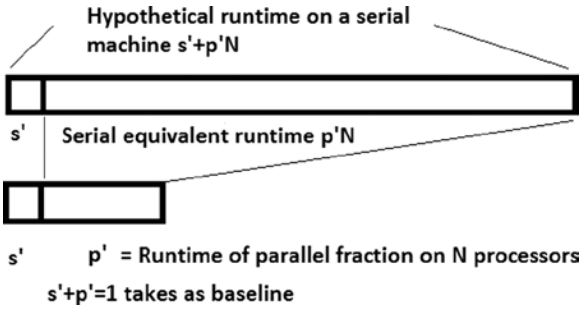

Figure 9-3. *Scaled speedup definition by E. Barsis. (Adapted from John L. Gustafson, "Reevaluating Amdahl's Law." Communications of the ACM May 1988.)*

Scaled Speedup

If we use s' and p' to represent serial and parallel time spent on the *parallel* system, then a serial processor would require time $s' + p' \times N$ to perform the task, where N is the number of processors. This reasoning yields Gustafson-Barsis's counterpoint to Amdahl's law (Figure 9-3):

$$\begin{aligned} \textit{Scaled speedup} &= (s' + p' \times N)/(s' + p') \\ &= s' + p' \times N = s' + (1 - s') \times N \\ &= N + (1 - N) \times s \end{aligned}$$

where $s' + p' = 1$ is the time spent on parallel computer taken as baseline.

[3]John L. Gustafson. "Reevaluating Amdahl's Law." *Communications of the ACM* 31(5), 1988, pp. 532–533.

As shown in Figure 9-4, Gustafson's law indicates that the scaled speedup is a line with moderate slope of 1 – *N*, where *N* is the number of parallel processors. Thus, increasing the number of processors can still provide fairly good performance for practical applications to be solved by such computers. Figure 9-4 shows that you can achieve 983x speedup on a 1024 core processor using Gustafson's model for a 4 percent serial fraction of the workload.

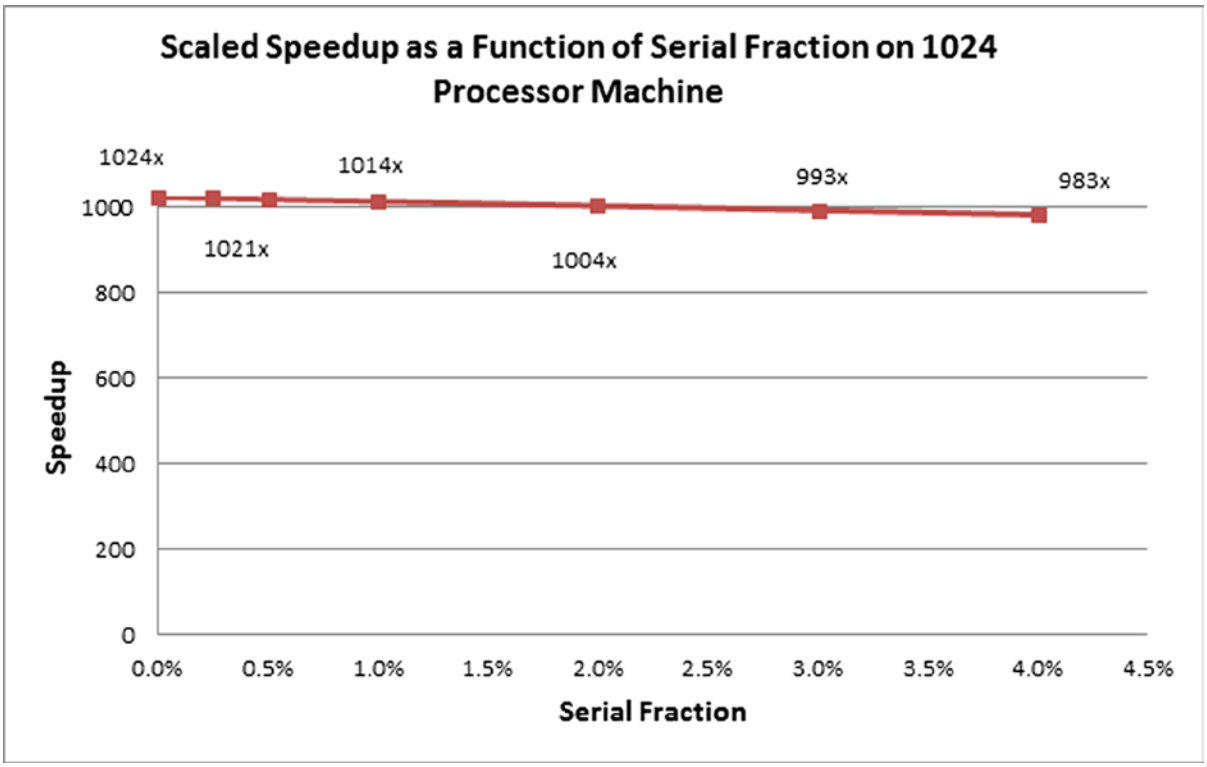

Figure 9-4. *Scaled speedup based on Gustafson's model*

Effect of Grid Shape on Performance

Technical computing applications often need to discretize the dataset into grids as part of the numerical representation of the problem. The grid size can sometimes drastically affect the performance of these apps on vector-based machine architectures, such as Intel Xeon Phi, as it dictates the memory access patterns. Often a developer or user has flexibility in picking the block shapes for simulation if they are aware of the advantage of one shape over the other.

As an example, let's look at two different shapes with the same volume, as shown in Figure 9-5. Both of these represent some simulation area—say, a region of earth where the weather patterns will be simulated.

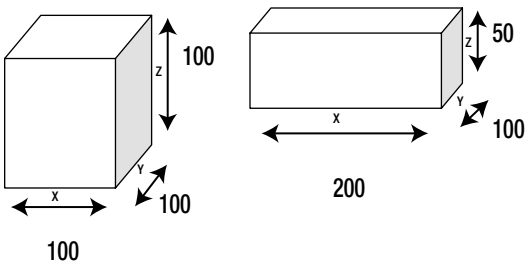

Figure 9-5. *Various shapes with the same 1 M cell volume*

In Figure 9-5, the left cuboid shows the grid size as 100 × 100 × 100 and the one on the right shows the grid size of 200 × 100 × 50 in X × Y × Z directions, respectively. The decomposition is done to exploit the memory capacity or maximum computational power available per node.

In a simulation code, this type of operation often turns into nested loops, as exemplified in the code segment in Listing 9-1 written in C-like syntax.

Listing 9-1. Example of a simulation code working on a 3D array

```
//Loop execution for cuboid in Figure 9-5
for ( z=zstart; z <zstart+100; z++){
    for(y=start;y<ystart+100;y++){
        for(x=start;x<xstart+100;x++)
        {
            Do simulation steps();
        }
   }
}

//Loop execution for rectangular cube in Figure 9-5
for ( z=start; z <zstart+50; z++){
    for(y=ystart;y<ystart+100;y++){
        for(x=start;x<xstart+200;x++)
        {
            Do simulation steps();
        }
   }
}
```

It needs to be recognized that if a big chunk of data is divided up among the cores of Xeon Phi, only the X dimension access for a pair of y and z values is expected to be in consecutive locations so that the processor can stream the data in for a given pair of y and z values. Once we move to a different pair of y and z values, the x start may be in a different memory page altogether. However, consecutive memory access for data elements along the x axis is consecutive if laid out properly.

This property will let the compiler vectorize the inner loop so that the memory access will be unit stride, allowing maximum memory access bandwidth in Xeon Phi's cache-based system. Having a longer x axis (say, 200 elements vs. 100 elements) allows the processors to do more work between resetting the X start address for the next pair of y and z values.

You can see the correlation between the innermost loop performance in the simulation computation and the X direction length of the volume being processed. In many cases, where the problem admits the flexibility of dividing the input dataset into different volume shapes, it may be possible to achieve high performance by doing so.

In internal experiments with similar data shapes for a simulation, we are able to achieve around a 35 percent boost in performance by changing the simulation grid shape to make vectorized code be more efficient.

Workload Consideration 1

- Choose the right problem size that will benefit from the Intel Xeon Phi.
- Amdahl's law can be misleading in designing for parallel performance.
- Gustafson's law is a better representation of parallel scientific computations.

Workload Consideration 2

- Choose a grid shape that is rectangular cuboid by making the unit stride dimension of the inner loop long enough.
- For better use of cache lines, map the unit stride long dimension to the inner loop in nested loop iterations.

Algorithm Considerations

In order to get optimal parallel performance on Xeon Phi, you need to identify algorithms and data structures that may need modification from the serial version of the code. It is often found that the existing parallel machines are limited by memory bandwidth rather than compute bandwidth for most of the applications, while the serial codes are often limited by the compute bandwidth of the single core. So the algorithm has to be redesigned or a different algorithm may need to be used to achieve high performance on the Xeon Phi manycore architecture.

An example of change in the choice of algorithm appropriate to solve a problem as we move from a sequential to parallel machine is the common data-sorting algorithm. We know that Quicksort is the algorithm of choice for serial computation for many applications because of its optimal average performance. But Quicksort involves walking over the array elements in the first step to divide them into two parts based on the pivot element and requires the step to be performed in a single processor, which seriously limits the performance. Computer scientists have proven that bucket sort, a seemingly simple algorithm, may be more appropriate in the parallel world.

The optimal performance algorithm chosen should match the practical hardware-supported flops-to-bytes metric. This metric provides the ratio of peak floating-point operations the processor can perform to peak bytes of data the hardware is able to sustain for the duration of computation. If you know the peak flops and peak bandwidth of a piece of hardware, this metric can be easily computed. The physical limit of Phi hardware determines the work-to-memory access ratio supported and influences the type of algorithm chosen to work optimally on a piece of hardware.

Many applications of practical importance nowadays are *memory bound*, that is, the compute waits on the data to arrive. In these cases, if F represents the algorithms compute-to-data ratio (flops/bytes) and B represents the maximum bandwidth (BW) bytes/sec of the platform, then the maximum performance achievable on such a platform is equal to $F*B$ flops. However, you need to use the appropriate BW for the platform, taking into account that some of the data may reside in the cache line.

Take, for example, an optimized DGEMM routine working on the L1 cache for its computation. Assuming that an optimized DGEMM routine can deliver a 3 flops/bytes ratio and the L1 cache can sustain 300 GB/sec BW on Xeon Phi, a DGEMM routine can achieve 3 × 300 = 900 GFlops/sec. On the other hand, the stream triad benchmark works on three double-precision numbers and performs two floating-point operations on them, providing a flops-to-bytes ratio equal to 2 / (3 × 8 = 24) = 0.083. So the highest achievable flops for this benchmark is 0.083 × 180 = 14.9 GFlops, limited by the memory interconnect bandwidth of approximately 180 GB/sec.

Parallelizing algorithms introduces issues that may not be present in the serial version of the code. As stated by Amdahl's law, once the application has been parallelized, you need to examine the sequential portion of the algorithm to reduce the serial fraction of the algorithm for optimal performance scaling and reap the benefit of parallel processing. This includes the communication overhead introduced by the parallel algorithm, which may not be present in the serial version of the code.

Communication and synchronization between the threads and processes introduced by parallelizing the algorithm usually show up as big overhead during the runtime of an application running on Xeon Phi. Experiments have shown that 30 percent of runtime overhead can in some cases be attributed to OpenMp or MPI communication and synchronization overhead. We would need to determine, for a given programming model, how to divide the task to minimize the communication-to-computation overhead. There are various patterns for parallel program and algorithm development available for programmers to design their applications optimized for their needs.[4] Intel provides a parallel programming library, the Intel Threading Building Blocks (TBB), and Intel Cilk plus language support and runtime to help reduce parallel overheads.

Our development experience with Xeon Phi has shown that the data-parallel design and algorithm patterns work well for Xeon Phi architecture. As Xeon Phi has a cache-based vector architecture, data-decomposition algorithm patterns result in better memory reuse and require less memory per core (as needed by task-parallel codes), because the total dataset does not need to be replicated to each individual node and can work on a subset of the total dataset. Having access to data to be worked on in parallel is a necessary condition for vector processing architecture.

Another side effect of data-parallel applications is less communication between the parallel processes, except in boundary exchanges. The data-parallel pattern allows the developer to focus on organizing data and can lead to better cache utilization and vector code generation by the compiler, resulting in better performance. However, some of the parallel applications that have been shown to perform well on Xeon Phi, such as ray tracing, are inherently task-parallel and do not need to be forced into data-parallel mode.

Data Structure

Another important consideration in achieving optimal performance on Xeon Phi is the data structure used by the algorithm. As a cache-based vector coprocessor, Intel Xeon Phi is heavily dependent on data layout and data access patterns for optimal performance. In designing the data structure for an algorithm, you need to give careful attention to the memory access patterns expected from or influencing the algorithm. The goal here is to maximize the reuse of data already residing in the L2 and L1 caches. The hardware consists of physical components that limit how fast the data can be accessed from the different levels of the memory hierarchy. For a given designed flops rate and bandwidth limit, there is a physical limit to the floating point-to-data bus bandwidth of these processors.

As an example of data structure selection, consider an algorithm working on an array of structures where structure elements are only partially accessed at one time. In the code segment in Listing 9-2, the position structure contains the x, y, and z coordinates of the object. Keep in mind that such data structures are more complex in real-world applications, with many more attributes added to them.

If the object is to move along the x-direction, we need to update the x-positions of the objects by accessing the objectPosition array of structures for the x position and updating them. Although we need only the x elements, owing to cache behavior the whole cache line containing the objects' position structure and consecutive object position

[4]T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming.* Reading, MA: Addison-Wesley Professional, 2004.

structures will be pulled into the cache. In addition to unnecessary data related to y and z elements polluting the cache line, it also wastes valuable interconnect bandwidth by using only the x component of the structure in the computation while reading all three x, y, and z values from memory, as shown in Listing 9-2.

Listing 9-2. Inefficient data structure layout for the given computation

```
typedef struct position
{
   double x;
   double y;
   double z;
}POSITION;

POSITION objectPositions[NUMOBJECTS];

Foo()
{
   // move the object along X direction by DIST amount
   for(int i=0;i<NUMOBJECTS; i++){
      objectPositions[i].x += DIST;
   }
}
```

If, however, the data structure is laid out such that the x-positions of all the objects are kept together, we will make optimal use of the data cache and bandwidth by pulling in x elements only, as shown in Listing 9-3. This common technique for parallel processors is known as the *Array of Structures-to-Structure of Arrays* (AoS-to-SoA) *transformation*.

Listing 9-3. AoS-to-SoA transformation

```
typedef real POSITION;
POSITION objectPositionsX[NUMOBJECTS], objectPositionsY[NUMOBJECTS],
objectPositionsZ[NUMOBJECTS];

Foo()
{
   // move the object along X direction by DIST amount
   for(int i=0;i<NUMOBJECTS; i++){
      objectPositionsX[i] += DIST;
   }
}
```

Offload Overhead

Because Xeon Phi is an attached coprocessor on a host system, the computational data have to be sent back and forth between the card and host memory to perform the computation. The algorithm must be chosen so that the data communication between the host and the card is minimized, be it an MPI or OpenMP or other threading-based application. The offload overhead can be considered part of the communication overhead of the algorithm for analysis purposes. When exporting computations to the Xeon Phi card, the data and computation code have to be exported to the card using the offload programming model.

Algorithms that allow asynchronous execution of various steps provide better parallelization and may reduce offload overhead by overlapping phases of computation and communication. Another aspect of algorithm design is to choose algorithms that allow for load balancing of computational tasks by dividing them between the host cores and the Xeon

Phi coprocessor cores. This can often be achieved with an algorithm using a pipelining technique to do its computation and creating a cost model for each of the computational pipeline stages. For example, in Linpack computation on a node with a Xeon Phi processor, you can design the algorithm so that the factorization, broadcast, and update of the matrices happen asynchronously. This will allow pipelining, the overlapping of various computational phases between the host and the coprocessor and prioritize performance-critical tasks to allow dependent tasks to progress.

Load Balancing

One of the key differentiating factors between a good and bad parallel algorithm when implemented is how well they divide the work among parallel processing units so that they are equally loaded. Load imbalance can lead to major performance loss. *Load balance* can be defined as the ratio between the average time to finish all of the parallel tasks (T_{avg}) and the maximum time to finish any of the parallel tasks (T_{max}). For a perfectly balanced load, the time spent by each of the processing units is equal and the load balance equals 1.

Load balance limits the parallel efficiency achievable by an algorithm on a piece of hardware. It can be shown that parallel efficiency is less than or equal to load balance for an algorithm on a piece of hardware.[5]

There are various algorithm patterns that result in load-balanced works, such as the "nearest neighbor" algorithm.

Algorithm and Data Structure Considerations

- Parallel algorithms that allow vectorization (such as data-parallel algorithms) are preferable. It is possible to gain 8x/16x for DP and SP code respectively, over nonvectorized code if fully optimized.
- Understand the loops and data layout in algorithms for optimal performance for Xeon Phi hardware.
- Try to maintain load balance for optimal performance.

Implementation Considerations

After the design and algorithm development for Xeon Phi, you need to look at the implementation issues. This involves understanding the hardware microarchitecture, the language constructs to implement the algorithm, the parallelization libraries, and the code optimization techniques to maximize underlying hardware efficiency.

Memory Management

Memory hierarchy enforces different optimizations for different parts of the code. Use of software prefetches and scatter gathers to get data to the L2 cache, where the hardware prefetchers are not effective, is one way to solve the issue. Hardware prefetchers in Xeon Phi will not be able to prefetch data into the cache if the data access is random or crosses page boundaries. Carefully experiment with software prefetching when necessary. Hardware prefetch helps on predictable strides and only prefetches data to the L2 cache. So use software prefetches to get data to the L1 cache, even in cases where the hardware prefetcher may be active. Sometimes you can use loop unrolling to work on multiple data items in the innermost loop, hiding instruction latencies and improving the flops-to-bytes ratio.

[5] L. R. Scott, T. W. Clark, and B. Bagheri. *Scientific Parallel Computing*. Princeton, NJ: Princeton University Press, 2005.

Figure 9-6 shows various memory latency considerations in optimizing the code. It shows estimated nonload latencies in different cache hierarchies on a Xeon Phi processor. It can be seen that there are 600 to 1,000 cycle latencies for getting data to the L2 cache line, so although a vector instruction can execute at four cycles, any stall due to data not available in the L2 cache line will incur hundreds to thousands of cycle latencies. It is important to have the data available in the cache line with some form of linear or scatter gather prefetches. This type of cache miss often happens when the data are accessed randomly from memory, and only the developer may have some way to prefetch the data in place for a given application. Figure 9-6 also shows that the fetching from L2 can itself cause tens of cycles if not already put into the L1 cache. So for latency-sensitive code, it may be necessary to further prefetch the data into the L1 cache with L1 prefetch instructions.

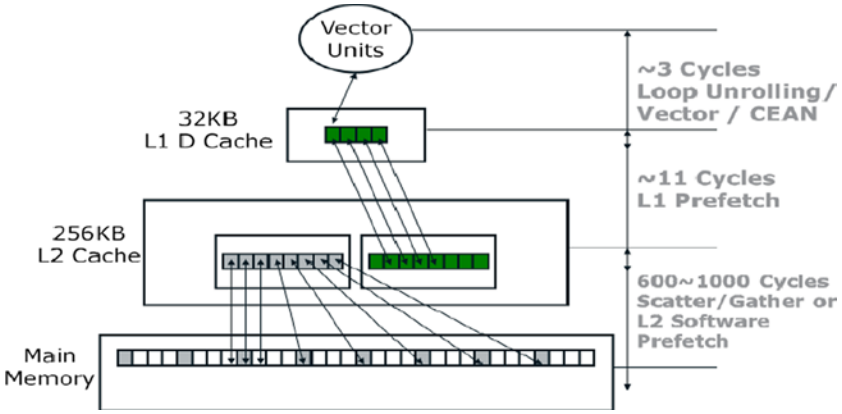

Figure 9-6. *Memory latency considerations during code optimization in selecting prefetch distances*

Mixed-Precision Arithmetic

Mixed-precision arithmetic indicates the use of single- and double-precision mathematics in a computation. Many problems can be reformulated using mixed precision with no loss in the final precision of the resulting value. Using lower precision—say single-precision instead of double-precision numbers—can result in a large speedup thanks to reduced memory traffic, because single-precision data are half the size of double-precision data, thus providing a better flops-to-bytes ratio. As explained in the architecture sections, there is hardware support in Xeon Phi that allows you to use single-precision transcendental functions in computations, also providing significant speedup. Some examples include iterative solvers, such as the Richardson iterative solver—where using single precision in the approximation step does not result in loss of computational accuracy but may in some cases improve performance. The Richardson iterative process seeks a solution to the matrix equation $Ax = b$, where A and b are given. It starts by approximating x values to x_k. The first step is to compute the residual using double-precision arithmetic. The second step finds a delta value to approximate the next x value until it converges. Because this is an approximation, it is possible to use single-precision arithmetic in this case, as shown in step 2 of the loop (Listing 9-4). This can yield substantial performance improvement.

Listing 9-4. Richardson's Iteration

```
While (|r_k| > e){
   r_k=b-Ax_k     <- DP SpMV  -------- (step 1)
   p_k=A^-1 r_k   <- SP approximation (step 2)
   x_k+1=x_k+p_k  <- DP
}
```

Similar mixed-precision arithmetic can be used in preconditioned Krylov solvers. The preconditioner can use single precision and the outer solver can use double precision without have much effect on the solution, while achieving a performance gain over default implementations.

Optimizing Memory Transfer Bandwidth over the PCIe Bus

Because Xeon Phi is an attached coprocessor, the input dataset has to be sent to the coprocessor of the PCIe Gen2 bus, which connects it to the core. A challenge we often face is how to get the maximum data transfer BW out of this communication channel. Use the following techniques to optimize the data transfer BW:

1. If the scratch-data buffers are required, they can be allocated on the coprocessor for the offload model to reduce the data transfer, so they do not need to be sent explicitly as they are already available on the card.
2. Align the data buffers to be transferred to the 4KB memory boundary to enable the DMA transfer between the host and the coprocessor for optimal transfer BW. There are several language constructs and functions, such as __mm_malloc, for this purpose.
3. Use the asynchronous transfer protocol provided by the offload compiler to do the transfer.
4. Use persistent data on the card to reduce duplicate transfers.

Data Alignment Considerations

One of the key performance benefits may be achieved by ensuring that the data are aligned to the cache line boundary on the host and Xeon Phi and that the algorithm works on cache line sizes. This allows better code generation from the compiler and low latency memory access. But keep in mind while creating such data access patterns that each individual thread is able to access aligned data. This often slips one's attention, but alignment of the overall dataset does not necessarily imply that each thread working on it will have aligned access. Because each thread accesses a part of the data structure, individual access to shared data structure may not be aligned. Remember to let the compiler know that the data are aligned by using the `aligned` pragma. Often the compiler cannot figure this out by itself, so conveying such information, where you have carefully laid out the data structure, will allow the compiler to generate optimal code. This will allow vector units throughput to be maximized without waiting for memory load latencies caused by unaligned access.

Communication

Once a program has been converted to a parallel version, there is often a need for various parallel components to work together and communicate with each other to finish a task. Some of the communications are at the startup phase, where the data/tasks are communicated to processes/threads by broadcast or one-to-one communications during runtime exchange information, such as the boundary condition between the threads or processes by one-to-many or one-to-one communication, and at the end to reduce and gather the results in the final output of the tasks through some sort of reduction method. Because communication could contribute to the serial part of a program, it is necessary to reduce the communication overhead. The hardware architecture as well as the algorithm play a big role in what type of communication needs to be done between the cores and processes.

Depending on the hardware architecture, you often need to decide between shared memories vs. the message-passing mode of communication. Although the shared memory system has much less overhead than message passing, access to shared resources can impose a high overhead owing to the need of synchronization. If you have a lot of communication and fewer shared resources, OpenMP may be better than MPI or vice versa. If you are using an offload programming model to communicate with Xeon Phi coprocessors, asynchronous data transfer often leads to a lower communication overhead than that of synchronous transfers. The same is true for message-passing algorithms.

File I/O

There are three ways to perform file input/output (I/O) on a Xeon Phi–based system. The coprocessor native I/O uses card memory for the file system. This consumes valuable GDDR memory space but is faster than other solutions. The second option is to use proxy I/O. In this configuration, you set MIC_PROXY_FS_ROOT to point to the host directory where you want the files to be proxied to. MPSS proxy I/O manager manages the I/O from the coprocessor file system to the proxied host file system. The third method is mounting a *network file system* (NFS) directory to the Xeon Phi coprocessor. In this case, all the writes from the coprocessor are written to the host-exported NFS directory.

The proxied and NFS-based file I/O are slower, as they have to transfer I/O data over the PCIe bus, but they do not consume any valuable memory resources. For optimal I/O performance and memory usage balance, it is beneficial to store smaller scratch files or performance-sensitive files on the coprocessor RAM drive and other files on the NFS-mounted or proxy file system. Proxy file I/O is based on NFS and, as such, does not provide any performance benefit over the NFS-mounted file system.

Implementation Considerations

- Use array vector notation and elemental function to help vectorization.
- Do some code restructuring for optimal vectorization.
- Used mixed-precision arithmetic where possible. Exploit hardware transcendental functions.
- Select proper parallelism constructs, OpenMP, MPI, and others to reduce communication to computation overhead.
- Strike a balance between native RAM file system and NFS or proxy file system for optimal file I/O performance without wasting coprocessor memory for file storage.

Summary

This chapter has looked at the various considerations that developers must take into account in designing and developing code for the Intel Xeon Phi coprocessor in order to extract maximum performance out of the hardware. You have seen the importance of selecting the proper workload to make full use of the raw compute power. You have also seen that the algorithm and data structure developed for bigcore processors such as Intel Xeon needs to be adjusted for the manycore architecture of Xeon Phi. Finally, you have seen that, even if proper design choices are made in the first two steps, implementation issues such as communication overhead may play a big role in application performance.

The next chapter looks at specific optimizations you can incorporate in your code to make it perform well on a Xeon Phi coprocessor.