Intel® Xeon Phi™ Application Design and Implementation Considerations

ABSTRACT: Parallel programming on any general purpose processor including Intel Xeon Phi™ coprocessor needs careful considerations of various aspects of program organization, algorithm selection and implementation to achieve maximum performance gain. The computer technology has made substantial progress over last couple of decades by introducing super scalar processors with pipelined vector architecture. We have also seen the rise of parallel processing in lowest computational segment like handheld devices. Today one can buy as much computational power of earlier supercomputers in less than a thousand dollars.

Massively parallel processors like Intel® Xeon Phi™ architecture has been developed to increase the computational power to remove these research barriers. However, to exploit a Many Integrated Core (MIC) coprocessor capable of providing more than Teraflops of double precision floating point performance in Technical computing applications will require careful design of algorithm, data structure and architecture understanding to match the hardware capabilities and idiosyncrasies of these processor.

This article is targeted to give the readers some pointers along the line of the parallel programming considerations that is needed for optimal performance on Parallel machines in general and Xeon Phi™ in particular. In it, we will discuss corresponding parallel data structure and algorithm used by various Technical Computing applications that will be suitable for such a coprocessor. We shall also look at the various source code level optimizations that can be done to exploit some of the processor features.

Some of the key challenges in developing a parallel code for Xeon Phi™ coprocessor are to develop and parallelize offload components to 240 threads. One often faces the challenge of:

- OpenMP overhead to start /stop threads can kill the performance if there are not enough computations to amortize thread creations.
- 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 rest of the system will wait for these cores to finish the processing for whole program to end.
- As many logical cores are working in parallel, they can easily be bottlenecked by concurrent memory / ring bus access latency and bandwidth.

In order to counter these issues, one need to manage data across threads with minimal sharing, align data structures for vector operations for optimal compiler code generations, prefer Algorithm to exploit higher flops/byte and balance workloads across MIC/Xeon and Cluster for load balancing.

There are various factors those affect parallel application performance such as Amdahl’s Law which describes how the parallel performance of an application will be limited by the serial portion of its execution time. Load imbalance: if the work shared between parallel cores is not equally distributed, all of the cores will end up waiting for the most overloaded core to finish the application. Some overhead like not present in serial version like task and adta dependencies.

One Needs Careful Considerations to produce optimal performance on this hardware

- To reduce the effect of Amdahl’s law
- Data set selection, Algorithm and data structure, and finally implementation

Figure 1 below shows various considerations that one needs to take into account to develop code for many parallel architectures including Intel Xeon Phi™.
One of the first steps to thinking about developing and running on a parallel computer is to think about what type of problem size of workload that will be run on the machine. Amdahl’s law (See Reference 1) familiar to all the parallel programmers describe that the amount of parallelization that can be achieved by parallelizing an application is limited by the percentage of the code that runs serially. Serialization is necessary when starting up the program, using a common resource like writing to a single file, etc.

Amdahl showed that the speed up that can be achieved by a parallel program on N number of processor can be given by Speedup = (s+p)/(s+(p/N)), where p is the parallel fraction, N number of parallel processors. If we use single processor runtime to normalize, that is set total runtime on a serial computer, s+p=1, we can define speedup as Speedup = 1/(s+(p/N)) ...(Equation 1).

Equation 1 indicates that the serial portion ‘s’ will dominate in massively parallel hardware like Intel® Xeon Phi™ limiting advantage of the many core hardware, when N tends to infinity, that is the parallel portion of the computation can be done in no time. This gave the computer scientist a pause and others to think that it is not worth creating parallel computers with more than 200 general purpose processing nodes. As it turns out they were wrong. People started building computers using thousands of cores and using them for wide variety of computational use, (See Reference 2), Dr. Alan Karp challenged Scientific community to demonstrate a speedup of at least 200 on a general purpose MIMD machine on three scientific apps.

Gustafson and team ([3]) showed that it is possible to achieve such performance. The solution boiled down to a misunderstanding of the problem that will be solved by these types of computer, that follows Amdahl’s law, but the parallel portion of the workload keeps on increasing with adding more processors as a result the serial section becomes smaller and smaller. This was first attributed to a paper published by Gustafson and team developing parallel methods for 1024 Intel processor based hyper cube at Sandia National Labs. (See Reference 3)

Gustafson proposed a scaled speedup which represents scientific computing where data size is not fixed. As more processors are thrown into the mix, one can solve even larger problems. As shown in Figure 2, if one assumes Amdahl’s law, one would get a speedup curve as function of serial fraction and will achieve a maximum speedup of 24x with only 4% serial fraction. Which seems it wasteful to build a machine with 1024 processors given that 4% serial fraction is itself is recognized as a highly parallel workloads. Figure 3 shows what Gustafson recognized as important in understanding parallel application performance on these massively parallel machines.
Gustafson’s Law:
In a paper “Reevaluating Amdahl’s Law” published in the Communications of the ACM in March 1988, Gustafson provided a speedup model that removed the barrier posed by Amdahl’s model against parallel computing. (See Reference 4)

During that time period there was considerable skepticism against the viability of parallel computers in scientific community. He showed that for three practical applications with serial fraction (s=0.4-0.8), his team received more than 1000x speedup on 1024 MIMD machines. In fact the parallel portion of the “work done” in real applications scales with the number of processors, causing overall serial performance fraction to be much reduced. E. Barris of Sandia proposed an inverse Amdahl’s model where 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 came up with the definition of scaled speedup as given below.

**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 $N$= number of processors. This reasoning gives an alternative to Amdahl’s law suggested by E. Barsis at Sandia:

$$\text{Scaled speedup} = \frac{(s' + p' \times N)}{(s' + p')} = s' + p' \times N = s' + (1 - s') \times N = N + (1 - N) \times s'$$

Where $s' + p' = 1$ is the time spent on parallel computer taken as baseline Gustafson’s law indicates that the scaled speed up is a line with moderate slope of $1-N$, where $N$ is the number of parallel processors.
Effect of Grid Shape on Performance

Technical computing applications often need to make the data set discreet to form grids as part of the numerical representation of the problem. The grid size may sometime affect the performance of these apps drastically on some vector based machine architecture like Intel Xeon Phi™ coprocessors, as they dictates the memory access patterns. Often a developer/user have flexibility in picking the grid size for simulation if they are aware of the advantage of one grid size over the other. For example, experiments have shown that rectangular cuboid grid shape is more amenable to vectoring due to longer length in the inner most loop. So often it may be beneficial to break a 100x100x100 grid shape into 200x100x50 for optimal performance.

Workload Considerations

Consideration 1:
- Choose the right problem size that will benefit from the Intel® Xeon Phi™ architecture
- Amdahl’s law can be misleading in designing for parallel performance
- Gustafson’s law a better representation of parallel scientific computations

Consideration 2:
- Make sure where possible to choose a grid size that is rectangular cuboid
• Results in more iterations in the inner loop
• By mapping longest dimension to inner loops in nested loop iterations.
• Causing better use of cache lines.

Algorithm and Data Structure Considerations

In order to get optimal parallel performance one needs to think of algorithms and data structures that may need modification from the serial version of the code. One needs to reexamine every sequential portion of the algorithm to reduce the serial fraction of the algorithm; this includes communication overhead induced by the parallel algorithm not present in the serial version of the code.

Another important aspect is the data structure used by the algorithm. As we shall see, the Intel Xeon Phi™ coprocessor being a cache-based vector processor is heavily dependent on data layout and data access pattern for optimal performance.

Communication between the threads and processes are also third considerations that one needs to think through carefully. We would need to find out for a given programming model how to divide up the task to minimize the communication to computation overheads. The offload overhead will be included in part of the communication overhead in this book.

The fourth component to this equation is load balance and work division granularities. An imbalanced implementation of the algorithm can lead to severe performance degradations.

Parallel Programming Patterns for Xeon Phi™ Coprocessors

In the book PARALLEL PROGRAMMING PATTERNS by TIM MATTSON, ET AL., the authors describe various algorithm design patters for parallel application development. Our development experience with Xeon Phi™ coprocessors has shown it to prefer the data parallel design and algorithm patterns. As this is a cache based vector architecture, data decomposition algorithm patterns results in better memory reuse, need smaller memory per core (as needed by task parallel codes) since the total dataset does not need to be replicated to each individual node and can work on subset of the total data set, and having access to data to be worked on in parallel is a needed 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 with this design pattern in mind lead to better cache utilization and vector code generation by the compiler resulting in better performance.

However, some of the parallel applications that have shown to perform well on Xeon Phi™ coprocessors like Ray tracing is inherently task parallel and does not need to be forced into data parallel mode.

Once designing the data structure for an algorithm, one need to give careful attention to memory access patterns expected from or influences the algorithm. The goal here is to maximize the re-use of data already residing in L2 and L1 cache. The hardware is consists of physical components that limit how fast the data can be accessed from the main and different level of memory hierarchy. Give a designed flops rate of these hardware and a given bandwidth limit, there is a physical limit to floating point to data bus bandwidth of these processors.

For optimal performance algorithm chosen should match the practical hardware supported Flops/Byte. The physical limit of Xeon 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 now days of practical importance are memory bound, that is the compute waits on data to arrive. In these cases, if F represents the algorithms compute to data ratio (flops/bytes) and B represents the maximum BW bytes/sec of the platform, then the maximum performance achievable on such a platform is = F*B flops. However, one needs to use the appropriate BW for the platform taking into account that some of the data may be residing in the cache line.
Data Alignment Considerations

One of the key performance benefits may be achieved by making sure, the data is aligned to cache line boundary on host and KNC and the algorithm to work on cache line sizes. This allows better code generation from the compiler and low latency memory access. However, keep in mind while creating such data access pattern that each individual thread to be able to access aligned data. This often slips ones attention, but aligned overall data set does not necessarily imply that each thread working on it will have aligned access. Since each thread access a part of the data structure and individual access to shared data structure may not be aligned. [note to self: Put an explanation box here]. Remember to let the compiler know that the data is aligned. Often compiler cannot figure it out 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 maximize without waiting for memory load latencies caused by unaligned access.

Load Balancing

One of the key differentiating factor between a good and bad parallel algorithm is how optimal they are in divvying up the work among parallel processing unit 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) to maximum time to finish any of the parallel tasks (T max). For a perfectly balanced load, the time spent by each of the processing unit is equal and thus, can achieve 1. Load balance limits the parallel efficiency achievable by an algorithm on a hardware. It can be shown that parallel efficiency is less than or equal to load balance for an algorithm on a hardware. So if LB=load balance of an algorithm for a workload and PE= parallel efficiency, it can be shown that PE <= LB. (See Reference 5) There are various algorithm patterns that results in load balanced works such as “nearest neighbor” algorithm.

Communications

Once a program is 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 startup phase where the data/tasks are communicated to each process/threads by broadcast or one to one communications, during runtime exchange information like boundary condition between the threads/processes by one to many or one to one communications and at the end to reduce and gather the results in the final output of the tasks through some sort of reduction method. Since communication could contribute to serial part of a program, it is necessary to reduce the communication overheads. The hardware architecture as well as algorithm plays a big role on what type of communication needs to be done between the cores and processes.

Depending on the hardware architecture, one often needs to decide between shared memories vs. Message passing mode of communication. Shared memory system makes message passing much less overhead, however, access to shared resources could impose high overhead due to the need of synchronization. If you have a lot of communication and less shared resources, OpenMP may be better than MPI and vice versa. If you are using offload programming model to communicate with Xeon Phi™ coprocessors, asynchronous data transfer often leads to lower communication overhead than synchronous transfers where possible. Same is try for message passing algorithms.
Algorithm and Data Structure Considerations

Consideration 1:
- Parallel Algorithms that allow vectorization is preferable. Example: Data parallel algorithms
- Possible to gain 8x/16x over non-vectorized code if fully optimized.

Consideration 2:
- Understand the loops and data layout. Understanding and communicating this info to compiler can help generate efficient vector code.

Consideration 3:
- Try to maintain Load balance
- It is sometime ok to leave one core idle to achieve load balance for better performance

Consideration 4:
- Select proper parallelism constructs, OpenMP, MPI and others to reduce communication to computation overhead.

Implementation Considerations

Once you have completed the design and algorithm development for the Xeon Phi™ coprocessor you need to look at the implementation issues. This involves a greater understanding of the hardware microarchitecture; the language constructs to implement the algorithm, the parallelization libraries and code optimization techniques to maximize underlying hardware efficiency.

Memory management

Memory hierarchy enforces different optimizations for different parts of the code. Use prefetches and Scatter Gathers to get data to L2 cache where the hardware prefetechers are not effective. Carefully experiment with software prefetching when necessary. Hardware prefetch helps on predictable strides. Use software Prefetches to get data to L1 cache. Sometimes one can use loop unrolling to work on multiple data items in the innermost loop, optimizing cache reuse, improving flops/byte ratio. The Xeon Phi coprocessor needs two threads to dispatch one instructions/ cycle. Figure 5 below shows various memory latency considerations while thinking about optimizing the code.

Mixed Precision Arithmetic

Many problems can be reformulated using mixed precision with no loss in final precision. Large speed up due to reduced memory traffic as SP data is half the size of DP, providing better flops/byte ratio. There is
also hardware support in Xeon Phi™ architecture that allows one to use SP transcendental functions in computations providing large speedup. An example, shown in Figure 6, includes Iterative solver-Richardson iteration where the using in the approximation step does not result in loss of computational accuracy but may improve performance in some cases.

While \(|p_k| > e|\{
\begin{align*}
    r_k &= b - Ax_k \\
    p_k &= A^{-1} r_k \\
    x_{k+1} &= x_k + p_k
\end{align*}
\}

Figure 6  Richardson's iteration

Example 1: In preconditioned Krylov solvers, The pre-conditioner can use single precision and outer solver can use DP.

Example 2: Mixed Precision MGEMM: Edgar et al. proposed mixed precision GEMM where Low precision data summed in high precision accumulator. [REF 8:http://www.nvidia.com/content/GTC/posters/72_Edgar_MGEMM.pdf]

Optimizing Memory Transfer BW

As the Xeon Phi™ coprocessor is an attached coprocessor, the input data set has to be sent to coprocessor of PCIe Gen2 bus that connects it to the core. One challenge often comes out is how to get the maximum BW out of this communication channel. Use following techniques to optimize the data transfer BW.

1. One technique to keep in mind that all the scratch data buffer can be allocated on the coprocessor for offload model to reduce the data transfer.
2. Align the data buffers to be transferred to 4KB memory boundary to enable the DMA transfer between host and the coprocessor for optimal transfer BW. There are several language constructs and functions like __mm_malloc for this purpose.
3. Use asynchronous transfer protocol provided by the offload compiler to do the transfer. This techniques will be covered in details in the programming section (Chapter ?)
4. Use persistent data on the card to reduce duplicate transfers. (Ref: Chapter ?)

Implementation Considerations

- Consideration 1:Use Array vector notation and elemental function to help vectorization
- Consideration 2: Will need some code restructuring for optimal vectorization
- Considerations 3: Used mixed precision arithmetic where possible and exploit HW transcendental functions

References

REFERENCE 1 - G. AMDAHL, VALIDITY OF THE SINGLE-PROCESSOR APPROACH TO ACHIEVING LARGE-SCALE COMPUTER CAPABILITIES, AFIPS CONFERENCE PROCEEDINGS 30, (1967), PP. 483–485.)

REFERENCE 2 - “SCIENTIFIC PARALLEL COMPUTING” – SCOTT RIGDEWAY ET. AL)