FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Implementation and Evaluation of MLEM algorithm on Intel Xeon Phi Knights Landing (KNL) Processors

Rami Al Rihawi

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Implementation and Evaluation of MLEM algorithm on Intel Xeon Phi Knights Landing (KNL) Processors

Implementierung und Evaluierung von MLEM-Algorithmus auf Intel Xeon Phi Knights Landing (KNL) Prozessoren

Author:	Rami Al Rihawi
Supervisor:	Prof. Dr. rer. nat. Martin Schulz
Advisors:	M.Sc. Dai Yang
	Dipl.-Tech. Math. (Univ.) Tilman Küstner
Date:	September 26th, 2018

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, September 26th, 2018 Rami Al Rihawi

Acknowledgments

I would like to take this opportunity to thank my thesis advisors, Dai Yang and Tilman Küstner. Their constant support and mentorship has created an atmosphere where I could accomplish whatever I set my mind on. I also would like to express my gratitude to Prof. Dr. rer. nat. Martin Schulz for the opportunity to do this thesis as well as trusting in my skills and commitment to execute it within schedule. Finally, I thank the people who I have not mentioned but were there for me during the process of this thesis.

Abstract

With the rise of High Performance Computing Systems usage for computationally intensive algorithms, it is necessary to examine these systems for the best configurations to run those algorithms efficiently. This thesis presents a study of the various setups and configurations of Intel Xeon Phi Knights Landing (KNL) for running Positron Emission Tomography (PET) scan image reconstruction algorithm, Maximum Likelihood Expectation Maximization (MLEM). It focuses on analyzing the effect of KNL specific hardware configurations, memory and cluster modes, as well as parallel programming models, such as Message Passing and Shared Memory, and techniques on performance. This yields 20 different setups to be compared to derive the best configuration to utilize KNL for running MLEM. The results show that, on one node, OpenMP implementation with affinity, implicit High Bandwidth Memory (HBM) usage model, and either All-to-all or Quadrant cluster modes outperform MPI-only and Math Kernel Library (MKL) implementation with respect to all KNL configurations. The assessment criteria are average runtime (performance), speedup (scalability), and bandwidth memory utilization. Nevertheless, it is worth to point out that implicit HBM usage model is obtained through linking Intel's heap manager library, called memkind, for offloading to KNL's HBM without any code changes; proving that it is easy to run code on KNL chipset.

Contents

1 Introduction

High Performance Computing (HPC) Systems have seen an increase of complexity in the past decade; consisting of improvements in increasing parallelism as well as adoptions of heterogeneous and accelerator architectures [1]. The drive behind the development in HPC is due to the demand for greater applications performance [2] especially for application which use computationally intensive algorithms. Medical image reconstruction from projections is an example of such algorithms, where reducing the computation time is a vital for clinical diagnosis [3]. Thus, there is a need to find the best systems and configuration to run those algorithms utilizing the computational power those systems offer as efficiently as possible.

1.1 Image Reconstruction

The general problem of image reconstruction has repeatedly arisen over the last 50 years in a large number of scientific, medical, and technical fields with an immense applicability range [4]. Image reconstruction refers to creating an image from a set of data, as an input, describing the object. The object usually can not be directly measured; thus, the object is observed indirectly. The reconstruction algorithm has to eliminate the effects of the observation technique used for collecting the data; in addition to the fact that this data is usually also poor and limited.

The goal of image reconstruction in medicine is to create images of cross section of a human body, especially internal structure such as organs, soft tissues, and bones, without invasive surgery and based on either transmission tomography or emission tomography. One of those applications in the medical diagnostic area is Positron Emission Tomography (PET) scan, which provides valuable metabolic information that is needed for diagnostics in fields like clinical oncology [5].

1.2 Positron Emission Tomography Scan

PET is a medical imaging modality with proven clinical value for the detection, staging, and monitoring of a wide variety of diseases. It requires the injection of a radiotracer to the subject. Then, the subject is placed within a ring to be monitored externally. "A positron-emitting radioisotope can be detected indirectly, as positrons annihilate with electrons, creating two 511 keV gamma photons traveling in opposite directions. When two detectors each record a photon within a certain time window, an annihilation event is assumed somewhere along the line connecting the detectors. This line is called the line of response (LOR). The number of detected events influences the quality of the measurement, while the coverage of three-dimensional space of interest (field of view, FOV) by LORs affects the achievable resolution. The resolution is usually better at the center than at the edges of the field of view. The FOV is commonly divided into a three-dimensional grid, where each grid cell is called a voxel" [6]. Figure 1 shows a PET scanner called MADPET-2.

There are two commonly used iterative reconstruction algorithms which are Maximum Likelihood Expectation Maximization (MLEM) [7] and Ordered Subset Expectation Maximization (OSEM) [8]. Both of these methods allow a detailed mathematical description of the physical processes involved in tomography systems, such as the attenuation and scatter of photons in the body under study [9]. MLEM is discussed in more details in Chapter 4.

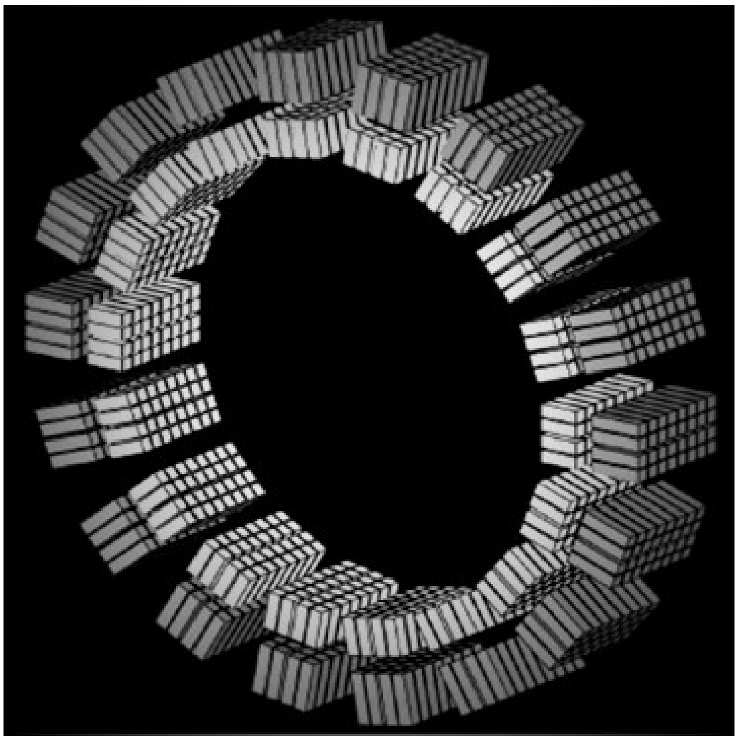


Figure 1: Illustration of PET Scanner MADPET-2 [10].

1.2.1 MADPET-2

It is a high-resolution 3D small animal PET scanner developed by Technical University of Munich at Klinikum Rechts der Isar to meet the demands of modern molecular imaging research. The unique detector design and data acquisition system increase sensitivity while preserving spatial resolution allowing new possibilities for data analysis [11]. As shown in Figure 1, the scanner is formed of detectors distributed into modules each consisting of two layers and 8 axial slices [12], which are all arranged in a shape of two concentric rings.

1.3 High Performance Computing Systems

For the past decades, manufacturers of computer chips has focus on improving the performance of chips by increasing number of transistors they contain in accordance to Moore's law. However, a limit has been reached in terms of making transistor smaller, adding more in a chip, and thus increasing the performance, namely capping the power usage and heat generation. The struggle is evident when examining chips' performance in the last decades: "Chip performance increased 60 percent per year in the 1990s but slowed to 40 percent per year from 2000 to 2004, when performance increased by only 20 percent, according to Linley Group president Linley Gwennap" [13]. As a result, manufacturers turned to building multicore chips instead of a powerful single core ones. Even though the multicore chip does not contain the most powerful cores, its overall performance is much better since it can exploits parallelism. Figure 2 shows analysis based on Intel tests using benchmarks (SPECint2000 and SPECfp2000) which reports that multicore processors perform much better than a single core processor and projects that relative advantage of multicore system will enhance over the next couple of years [14] as parallelism is exploited. Nevertheless, parallelism is discussed in more details in Chapter 2.

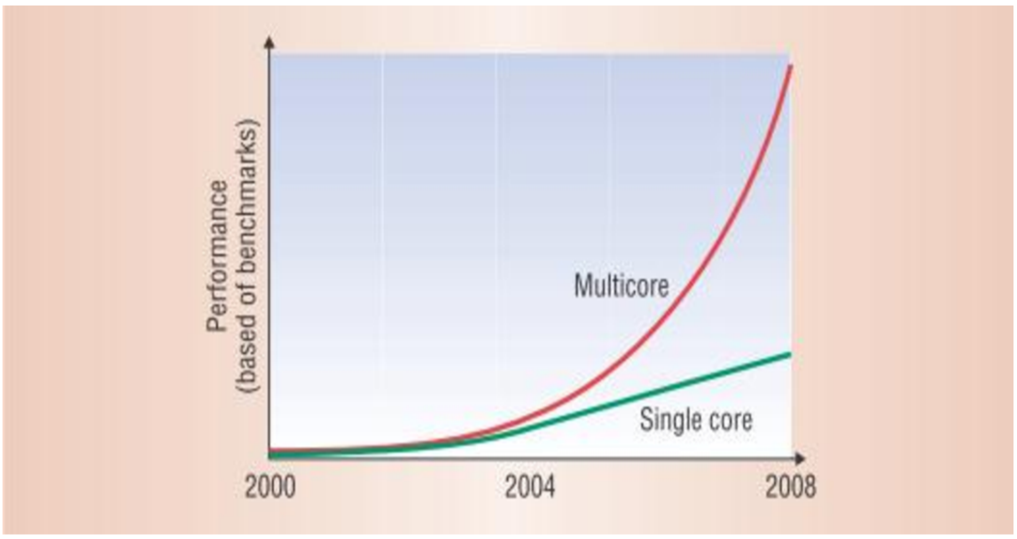


Figure 2: Performance comparison between a single core and multicore processor.

HPC Systems are found based on the same concept of increasing the processing cores or unit count as discussed in the previous paragraph. Those systems are categorized into either homogeneous or heterogeneous computing. Both differ on the kind of processing power they add to gain better performance. Homogeneous computing refers to using same or similar processing units; unlike heterogeneous computing which refers to using more than one kind. The benefits of increasing the processing cores or units in heterogeneous computing system is not limited to gains in performance only; but it also can incorporate specialized processing units that are capable of handling specific tasks. However, having multiple kind of processing units increases the complexity of the system. Thus, there are advantages and disadvantages for both HPC system types.

The new era of supercomputers was reshaped by Intel's Many Integrated Core (MIC) Architecture processors and Nvidia accelerators. Their computing capabilities and integration allowed for faster HPC systems in comparison to the traditional CPUs [15], [16]. According to June 2018 list of top500.org, most of the 10 fastest supercomputers are either based on Intel Xeon MIC (e.g. Trinity and Cori) or GPU (e.g. Titan and Summit). Based on Intel MIC architecture, Intel Xeon Phi product family targets HPC [17] especially Knights Landing (KNL) providing a large number of cores to achieve an aggregated performance with a relatively simple control of each core rather than increasing the CPU clock frequency, and providing highly functional multiple cores to save the total power consumption of the chip for a higher performance watt ratio [18] as well as a high bandwidth memory (HBM) all in one chip, which is discussed in more details in Chapter 3.

This thesis is only concerned with KNL, 2nd generation of Xeon Phi product family, and MLEM. It focuses on analyzing the various configurations for KNL as well as several optimizations and techniques of MLEM for best performance. The rest of it is organized as follows: Chapter 2 covers an overview of parallel concepts, techniques and libraries named in the work. Chapter 3 provides an overview of KNL architecture. Chapter 4 explains details and implementation of MLEM algorithms. Chapter 5 discusses the setup and the results of running MLEM in various configurations as well as analysis explaining the results. Chapter 6 reviews published work related to the presented work. Finally, Chapter 7 summarizes the work and proposes possible future plans.

2 Parallel Computing

The concept of parallelism is quite old. It goes back to 1958; Gill discussed the need for parallel programming with the concepts of branching and waiting [19] and IBM researchers discussed the use of parallelism in numerical calculations for the first time [20]. Few years later, in 1962, Burroughs Corporation introduced a four-processor computer that accessed up to 16 memory modules [21]. During the American Federation of Information Processing Societies Conference, in 1967, a debate established that Amdahl's law was coined to define the limit of speed-up due to parallelism [20].

In the meantime, IT industry has improved the cost-performance of sequential computing by about 100 billion times overall over the past 60 years. For most of the past 20 years, architects have used the rapidly increasing transistor speed and budget made possible by silicon. This led to innovations that were inefficient in terms of transistors and power but that increased performance while preserving the sequential programming model. After crashing into the power wall, architects were forced to find a new paradigm to sustain ever-increasing performance demand. That is when the industry started replacing the single power-inefficient core with many efficient cores on the same chip. This served as a statement of the new future of parallel computing led by increasing the number of processors and/or cores. Hence, the leap to multicore multiprocessor is not based on a breakthrough in programming or architecture and is actually a retreat from the more difficult task of building power-efficient, high-clock-rate, single-core chips [22].

The rest of this chapter is organized as follows: Section 2.1 covers a general overview of different parallel programming models with examples of libraries that are used in this work. Section 2.2 introduces Intel's Math Kernel Library (MKL). Finally, Section 2.3 expands on advanced parallel programming concepts that are also discussed in this work.

2.1 Parallel Programming Models

The variation among models are motivated by a couple of factors. First, there is a different amount of effort invested in writing parallel programs. Second, various machines and setups can support different parallelism approaches to exploit them. This section discusses those two factors in more details.

Transforming serial code into parallel is not an easy task especially since debugging becomes quite difficult. Thus, parallel programming models are categorized based on that yielding implicit and explicit parallel programming models. Explicit models requires a parallel algorithms to explicitly specify, with code, how the processors will cooperate in order to solve a specific problem. That allows the programmer to be in more control of the parallelization. While in implicit models, the compiler inserts constructs necessary to run the program in parallel. That makes it the easier model to use for the programmer since the majority of the burden of parallelization falls on the compiler [23].

Parallel Programming models can also be categorized based on memory setup. As illustrated in Figure 3, the models can be categorized into Shared Memory and Distributed Memory models. Both are explained in more details in the next section.

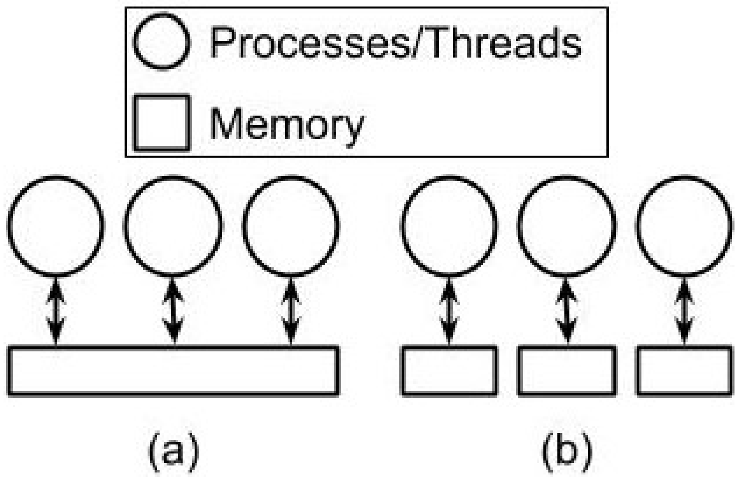


Figure 3: Illustration of the Parallel Programming Models based on their overlook on and use of memory. a) Shared Memory model b) Distributed Memory model.

2.1.1 Shared Memory

In a Shared Memory programming model, programmers view their programs as a collection of processes assessing a central pool of shared variables. Each processor can write to and read from a shared variable. That allows information sharing and ability to communicate at some level. It also introduces problems such as if two processes share a loop variable which both of them increment; that could result in undesirable behavior of misses loop executions. Shared Memory programming style is naturally suited to computers with a pool of shared memory [23].

Shared Memory programming model is classified into two types based on the shared memory setup, access, and latency which are Uniform and Non-uniform Memory

Access shown in Figure 4. Uniform Memory Access (UMA) implies that all processes share access to the physical memory uniformly; moreover, latency is the same for all processors requesting the same information from the same memory chip. Meaning, access time is independent of which processor makes the request as well as which memory chip contains the data. While, Non-Uniform Memory Access (NUMA) implies the opposite. A processor can access data from its own local memory faster than the non-local shared memory.

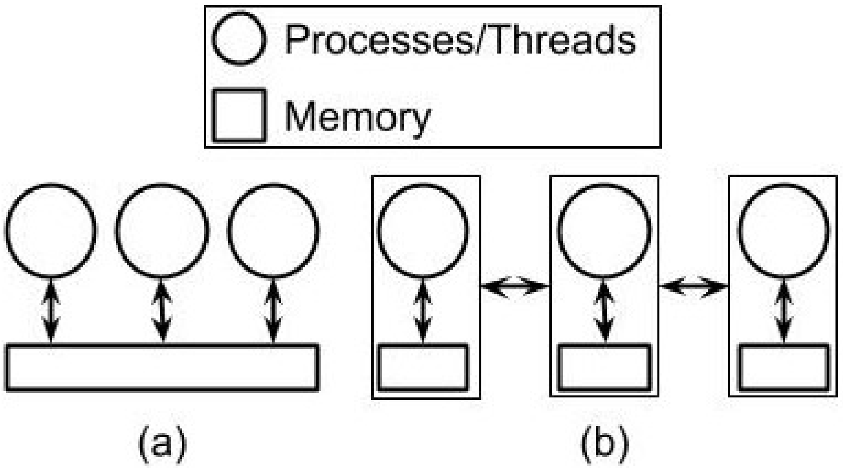


Figure 4: Illustration of the Shared Memory types (a) Uniform Memory Access (UMA) (b) Non-uniform Memory Access (NUMA).

The next section covers an overview of a popular library, called OpenMP, based on Shared Memory programming model. The following information is summarized and taken from [24], [25].

OpenMP

Pioneered by SGI and developed in collaboration with other parallel computer vendors. OpenMP is fast becoming the de facto standard for parallelizing applications. There is an independent OpenMP organization today with most of the major computer manufacturers on its board, including Compaq, Hewlett-Packard, Intel, IBM, Kuck & Associates (KAI), SGI, Sun, and the U.S. Department of Energy ASCI Program. The OpenMP effort has also been endorsed by over 15 software vendors and application developers, reflecting the broad industry support for the OpenMP standard.

OpenMP is a parallel programming model for shared memory and distributed shared memory multiprocessors shown in Figure 4 (a) and (b) respectively. Nevertheless, it is an Application Programming Interface (API) that provides an easy-to-use portable scalable parallel programming model for developers. The API supports C/C++ and

Fortran on a wide variety of architectures. Moreover, it comprised of three primary API components: compiler directives, runtime library routines, and environment variables as shown in Figure 5. OpenMP API is independent of the underlying machine/operating system; therefore, OpenMP compiler as well as standard implementation exists for all the major operating systems. Compiler directives express shared memory parallelism and are invoked using basic semantics, such as in C `#pragma`. Runtime routines allow the interaction and inquiries to OpenMP specific variables.

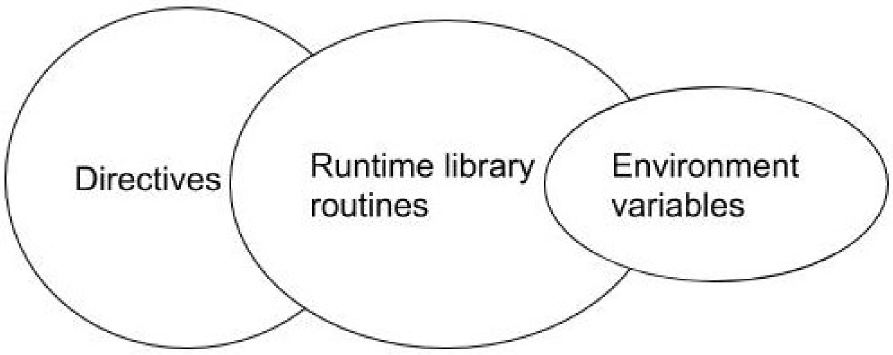


Figure 5: The components of OpenMP [25].

2.1.2 Distributed Memory/Message Passing

In Message Passing programming model, programmers view their programs as a collection of processes with private local variables and the ability to send and receive data between processes by passing messages; hence the Message Passing naming. There are no shared variables among processors; each has its own locally preserved variables and could send and receive data. Thus, this style is naturally suited to message-passing computers [23]. Figure 6 shows an illustration of Message Passing Model.

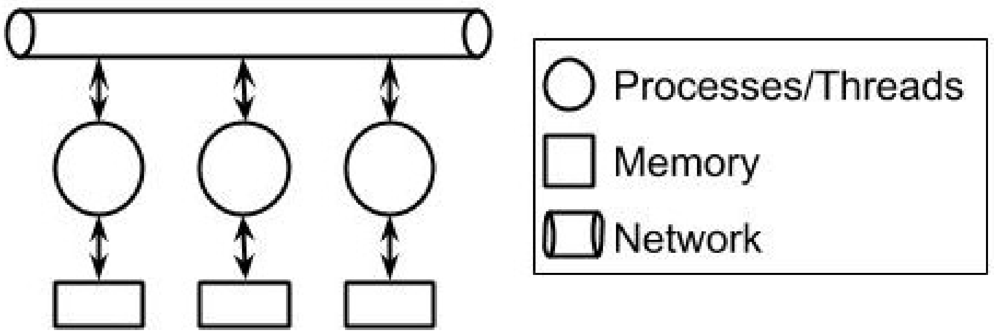


Figure 6: Illustration of Message Passing Model.

There are two key attributes that characterize the message-passing programming paradigm: 1) it assumes a partitioned address space, as shown in Figure 3 (b) 2) it supports only explicit parallelization [23]. Message passing libraries allow efficient parallel programs to be written for distributed memory systems; hence the Distributed Memory naming. These libraries provide routines to initiate and configure the messaging environment as well as sending and receiving packets of data. Currently, the two most popular high-level message-passing systems for scientific and engineering application are the Parallel Virtual Machine (PVM) from Oak Ridge National Laboratory and Message Passing Interface (MPI) defined by the MPI Forum [26].

The next section covers an overview MPI which is summarized and taken from [27]–[29].

MPI

It was designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The process of creating it began at a workshop on Message Passing Standardization in April 1992, and the MPI Forum organized itself at the Supercomputing 1992 Conference. During the next eighteen months the MPI Forum met regularly, and Version 1.0 of the MPI Standard was completed in May 1994. Since its release, the MPI specification has become the leading standard for message-passing libraries for parallel computers. More than a dozen implementations exist, on a wise variety of platforms. Every vendor of high-performance parallel computers offer an MPI implementation as part of the standard system software, and there are a number of freely available implementations for heterogeneous networks of workstations and symmetric multiprocessors. That is a result of the representation on the MPI Forum, which designed MPI: all segments of the parallel computing community, vendors, library writers, and application scientists.

MPI is a standardized and portable message passing standard that facilitates the development of parallel applications and libraries. The standard defines the syntax and semantics of a core library routines that can be used to specify the communication among a set of processors forming a concurrent program. MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes. Moreover, it provides three classes of services, environmental inquiry, basic timing information for application performance measurement, and a profiling interface for external performance monitoring.

2.1.3 Hybrid

Best of both worlds, Shared and Distributed Memory. It allows the utilization of both models strengths. Having said that, it may not always be beneficial, since it depends heavily on the problem statement's computational and communication loads. However, if it fits for a problem, it can give considerable speedup and better scaling [30].

Hybrid approach for parallelism is a trend in HPC for the current hybrid hardware architectures. Lots of research look into using MPI-OpenMP; in the same time, new parallel performance libraries are adapting both models. The next sections discuss an overview on MPI-OpenMP.

MPI-OpenMP

HPC systems are nowadays exclusively of the distributed-memory type at the overall system level but use shared-memory compute nodes as basic building blocks (Hybrid Architecture) [31]. Hence, the rise of MPI-OpenMP programming model that allows any MPI process to spawn a team of OpenMP threads to deal with the shared memory, while communicating with other MPI processes on distributed memory level. Hager et. al. in [31] defines two basic hybrid programming approaches based on MPI-OpenMP, Vector and Task Mode. They differ in the degree of interaction between MPI calls and OpenMP directives. [31] explains them in more details as follows:

Vector Mode. All MPI subroutines are called outside OpenMP parallel regions, i.e., in the "serial" part of the OpenMP code. A major advantage is the ease of programming, since an existing pure MPI code can be turned hybrid just by adding OpenMP work sharing directives in front of the time-consuming loops and taking care of proper NUMA placement.

Task Mode. It is most general and allows any kind of MPI communication within OpenMP parallel regions. Based on the thread safety requirements for the Message Passing library, the MPI standard defines several different levels of interference between OpenMP and MPI. Before using task mode, the code must check which of these levels is supported by the MPI library. Functional task decomposition and decoupling of communication and computation are two areas where task mode can be useful.

To explain both modes in more depth, a 3D Jacobi solver algorithm is adapted to reflect the modes. Algorithm 1 and 2 show pseudocode of a 3D Jacobi solver to demonstrate the differences between Vector and Task mode respectively. Jacobi method is prototypical for many stencil-based iterative methods in numerical analysis and simulation. In its most straightforward form, it can be used for solving the diffusion equation for a scalar function.

2.2 Math Kernel Library

MKL includes routines and functions optimized for Intel that take advantage of vectorization and shared memory multiprocessing capabilities as well as distributed memory

Algorithm 1 Vector Mode hybrid implementation of a 3D Jacobi solver.

```
do iteration=1,MAXITER
...
!$OMP PARALLEL DO PRIVATE(..)
    do k = 1,N
        ! Standard 3D Jacobi iteration here
        ! updating all cells
        ...
    enddo
!$OMP END PARALLEL DO

! halo exchange
    ...
    do dir=i,j,k

        call MPI_Irecv( halo data from neighbor in -dir direction )
        call MPI_Isend( data to neighbor in +dir direction )

        call MPI_Irecv( halo data from neighbor in +dir direction )
        call MPI_Isend( data to neighbor in -dir direction )
    enddo
    call MPI_Waitall( )
enddo
```

Algorithm 2 Task Mode hybrid implementation of a 3D Jacobi solver.

```
!$OMP PARALLEL PRIVATE(iteration,threadID,k,j,i,...)
  threadID = omp_get_thread_num()
  do iteration=1,MAXITER
  ...
  if(threadID .eq. 0) then
  ...
  ! Standard 3D Jacobi iteration
  ! updating BOUNDARY cells
  ...
  ! After updating BOUNDARY cells
  ! do halo exchange
    do dir=i,j,k
      call MPI_Irecv( halo data from neighbor in -dir direction )
      call MPI_Send( data to neighbor in +dir direction )
      call MPI_Irecv( halo data from neighbor in +dir direction )
      call MPI_Send( data to neighbor in -dir direction )
    enddo
    call MPI_Waitall( )

  else ! not thread ID 0

  ! Remaining threads perform
  ! update of INNER cells 2,...,N-1
  ! Distribute outer loop iterations manually:

    chunksize = (N-2) / (omp_get_num_threads()-1) + 1
    my_k_start = 2 + (threadID-1)*chunksize
    my_k_end = 2 + (threadID-1+1)*chunksize-1
    my_k_end = min(my_k_end, (N-2))

 ! INNER cell updates
    do k = my_k_start , my_k_end
      do j = 2, (N-1)
        do i = 2, (N-1)
          ...
    enddo enddo enddo
  endif ! thread ID

!$OMP BARRIER
  enddo
!$OMP END PARALLEL
```

parallelism using MPI. MKL includes several groups of routines which span across various mathematical computational challenges; the following are some of the popular group of routines [32] :

- Basic Linear Algebra Subprograms (BLAS):
  - Level 1 BLAS: vector operations.
  - Level 2 BLAS: matrix-vector operations.
  - Level 3 BLAS: matrix-matrix operations.
- Super BLAS Level 1, 2, and 3 (basic operations on sparse vectors and matrices) .
- LAPACK routine for solving systems of linear equations, least squares problems, eigenvalues and singular value problems, Sylvester's equations, and auxiliary and utility LAPACK routine.
- ScaLAPCK computational, driver, and auxiliary routines for solving systems of linear equations across a computer cluster.
- PBLAS routines for distributed vector, matrix-vector, and matrix-matrix operation.
- MKL PARDISO, direct sparse solver routines implementing LU factorization and Cholesky factorization for matrices stored in sparse data format.

MKL provides both static and dynamic libraries for KNL. Furthermore, it offers a *Single Dynamic Library* interface to simplify linking [32].

2.3 Advance concepts

2.3.1 Affinity (Thread/Process Pinning)

Enables the binding of threads and/or processes to specific cores results in exploiting data locality hence improving the performance. There are multiple strategies for affinity, such as scatter, balanced, and compact. Figure 7 illustrates the differences between the three affinity strategies. Compact strategy pins the threads as close as possible to main thread. Balanced strategy distributes the threads equally on the cores. Scatter strategy is similar to balanced, but it distributes threads such that threads with IDs in close numerical proximity are pinned on different cores.

Since effects of affinity is a widely studied concept and programs can benefit from it, several implementations/libraries tackled affinity. For example, Intel introduce an environment variable, `KMP AFFINITY`, allowing Intel's compliers to bind OpenMP threads to cores [33]. Then, July 2013 OpenMP version 4.0 was released introducing affinity of data (first touch policy, privatization, etc.), threads, and the mapping of work to threads (e.g. work-sharing constructs) [34].

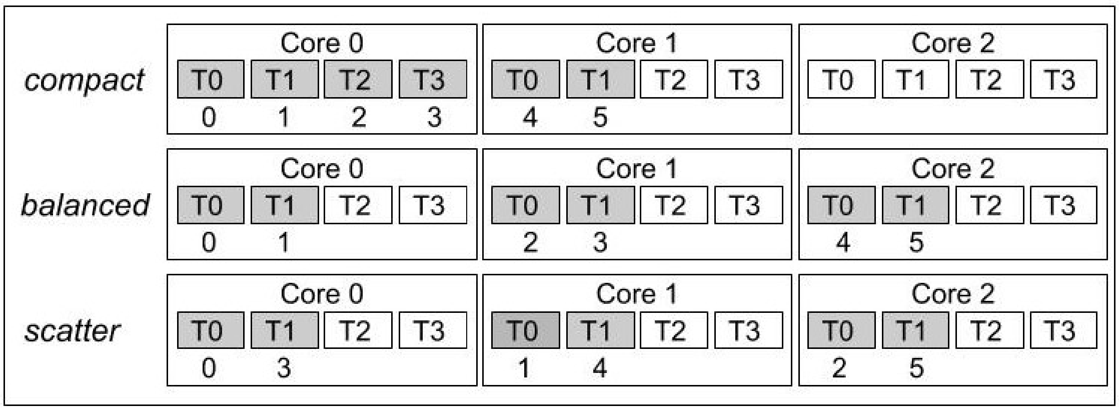


Figure 7: Illustration of affinity compact, balanced, and scatter pinning 6 threads applied on three cores with four threads

3 Xeon Phi Knights Landing

Xeon Phi Knights Landing (KNL) is part of the Xeon Phi family processors which is based on Intel MIC architecture. Xeon Phi family integrates many x86 Xeon Phi processors which deliver massive parallelism and vectorization to support demanding high-performance computing applications [35]. As a result, making Xeon Phi family able to handle your most-demanding HPC applications [36].

Xeon Phi family starts with Knights Ferry, a prototype board that was never released commercially. In 2011, Intel showed an early silicon version of its first commercial Xeon Phi board from Xeon Phi family named Knights Corner (KNC), which offers 60 cores per chip. In 2013, Intel revealed details of its second generation Intel Xeon Phi products. Later Intel canceled the third generation named Knights Hill. However, eventually, it re-released an updated verion of it based on the same architecture and specification named Knights Mill in late 2017.

As mentioned previously in Chapter 1, KNL is the second generation. This generation brings big changes from the previous generation, KNC [17]; namely, 2D mesh interconnect instead of 1D ring interconnect, stand alone processors instead of coprocessors, 72 cores out-of-order cores instead of 60 in-order cores, and two 512 Vector Processing Units (VPU) instead of one. The terms and hardware components mentioned are introduced and covered in more details later in this chapter.

The rest of this chapter summarizes the KNL architecture based on [17], [32] and it is organized as follows: Section 3.1 introduces hardware components and terminology. Section 3.2 covers an overview of KNL architecture. Section 3.3 details memory specific configuration and mode that is offered by KNL. Section 3.4 details cluster modes that are offered by KNL.

3.1 Terminology

In order to discuss KNL architecture especially low level hardware specific component in HPC systems, this section introduces hardware components and terminology that is discussed in more details in this chapter.

- **Core** It is the electronic circuitry that performs instructions specified in a program by performing the basic arithmetic, logical, control and input/output (I/O) operations.

- **Coprocessors**. It is microprocessor designed to supplement the capabilities of the primary processor.
- **Central Processing Unit (CPU)**. It is a central processor where most of the operations and calculations happen. A single CPU contains at least one core.
- **Vector Processing Unit (VPU)**. A functional unit designed with arithmetic pipelining for vector processing is attached to a base data processor from which it receives vector instructions and operands for processing [37].
- **Dynamic Random-Access Memory (DRAM)**. It is a type of random access semiconductor memory that stores each bit of data in a separate tiny capacitor within an integrated circuit corresponding to charged or discharged which represents 0 and 1.
- **Double Data Rate (DDR)**. It refers to transferring data on both the rising edge and falling edge of the DRAM clock signal; as a result, doubling the peak data rate [38].
- **Multichannel DRAM (MCDRAM)**. It is a 3D stacked DRAM which has about 5 times bandwidth of the DDR4 memory (up to 475 GB per second) [39]. It is considered as a High Bandwidth Memory (HBM) and can be used in different modes with different usage models, which will be explained in details in Section 3.3.
- **High Bandwidth Memory (HBM)**. The evolution of the DRAM has been driven by ever-increasing speed requirements mainly dictated by the microprocessor industry [40]. To keep up with that, a standard interface to interact with the new generation of the DRAM, which represents High Bandwidth Memory devices, was established.
- **Peripheral Component Interconnect Express (PCIe)**. It is a high-speed serial computer expansion bus standard for I/O.
- **Direct Media Interface (DMI)**. It is a connection between the processor and the Platform Controller Hub [41].
- **Mesh**. It refers to the network setup connecting multiple processors/computers to facilitate their communication.
- **Cache**. High-speed buffer memory used in modern computer systems to hold temporarily those portions of the contents of main memory which are (believed to be) currently in use. Information located in cache memory may be accessed in much less time than that located in main memory. Thus, CPU with a cache memory needs to spend far less time waiting for instructions and operands to be fetched and/or stored [42].

- **Cache Miss**. It refers to the state of when data requested by a processor but it is not found in the cache memory. As a result, it triggers a request for the memory to be fetched from the next cache or main memory; causing execution delays. Figure 8 illustrates the communication caused when a processor requests data in a memory system with two levels of cache.

- **Level-1/2 (L1/2) Cache**. It is a cache that is usually built onto the microprocessor chip itself. It is the closest to a core in the CPU, with the lowest latency, and it is usually only few KB. While, L2 is a cache that is slower than L1 cache and could be shared between multiple cores with a size of less than or equal to MB. Figure 8 illustrate a memory system with two level of cache and its communicate with the processor when it requests data.

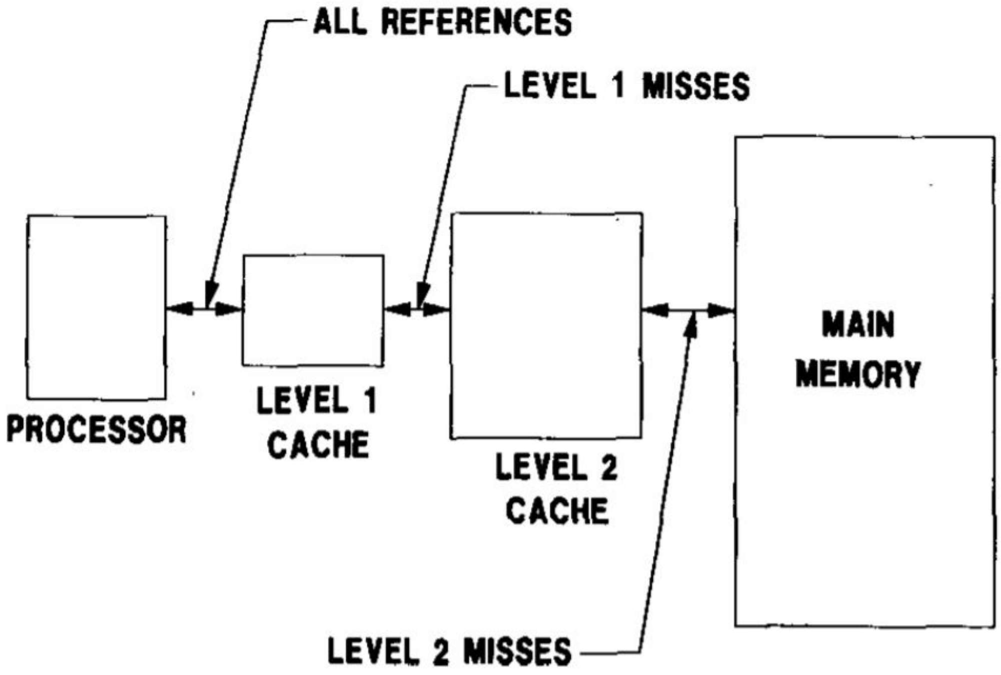


Figure 8: Illustration of a memory system with two levels of cache, L1 and L2 cache [43].

- **Caching and Home Agent (CHA)**. It holds a portion of the Distributed Tag Directory, which is discussed and explained in more details in Section 3.3, and also serves as a connection point between the tile and the mesh.

3.2 Architecture

In general, KNL chipset consists of tiles, MCDRAM, DRAM, and I/O, as shown in Figure 9. Components are as follows:

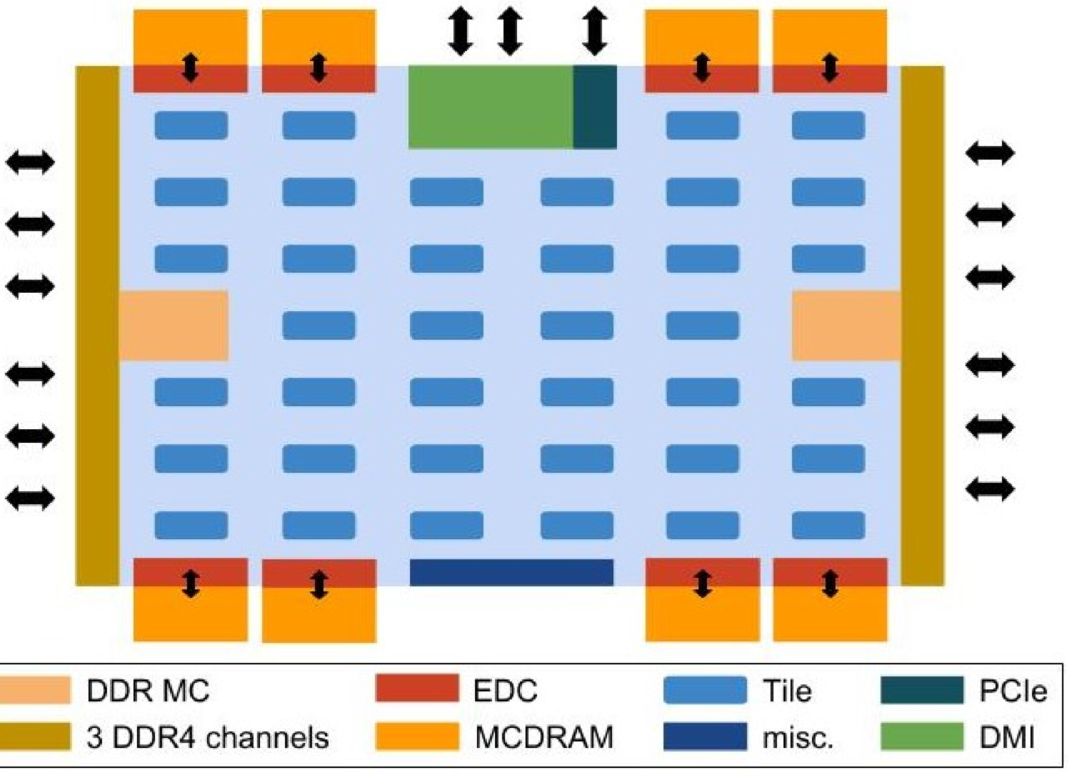


Figure 9: Diagram demonstrating an overview of the KNL Architecture.

- **Tile**. Composed of two cores, four VPUs, L2 cache, and a CHA, as demonstrated in Figure 10. The two cores share a 16 way associative, 1 MB unified L2 cache. The cores are Intel Atom based codename Slivermont; each supports up to 4 threads running at 1.3 to 1.5 GHz with a 32 KB L1 cache and two 512 bit VPUs. Nevertheless, CHA is a distributed cache directory to keep cache coherent.

- **MCDRAM**. Eight 2 GB of MCDRAM. All eight MCDRAM devices together provide an aggregate Stream triad benchmark bandwidth of more than 450 GB per second.

- **DDR4**. The total DDR memory capacity supported is up to 384 GB.

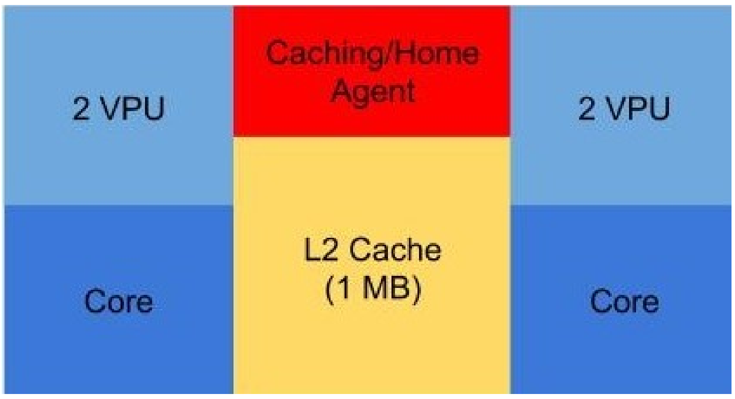


Figure 10: Diagram demonstrating an overview of a tile in KNL Architecture.

- **I/O**. Eight MCDRAM controllers (EDC), two DDR memory controllers that support 6 channels with a bandwidth up to 90 GB per second, 2x16 lanes and 4 lanes PCIe Gen3 each at 15.75 GB per second, and 4 lanes of DMI.

To sum up, a KNL chipset consists of a 2D mesh interconnect of 36 tiles (comprise of 72 cores and 144 VPUs), 8 MCDRAM (2G each; 16 GB in total), 2 DDR memory controllers supporting up 384 GB DD4 RAM in total, 36 lanes PCIe Gen 3, and 4 lanes DMI.

3.3 Memory Mode

As discussed before, MCDRAM is a HBM that offers three different ways to be used, namely Cache, Flat and Hybrid.

3.3.1 Cache

HBM can be used as a cache and it is treated as a Last Level Cache. Meaning, it is located between the L2 cache and addressable memory, in this case the DDR4 RAM. Figure 11 illustrates the memory system of KNL where MCDRAM is cache.

3.3.2 Flat

The entire HBM memory is added to the address space extending the space of the existing DDR4 Memory. In this mode, it is possible to allocate memory in the HBM using one of the HBM usage models, Implicit and Explicit, which are supported through memkind interface.

"Intel's memkind library is a user extensible heap manager built on top of jemalloc [44] which enables control of memory characteristics and a partitioning of the heap

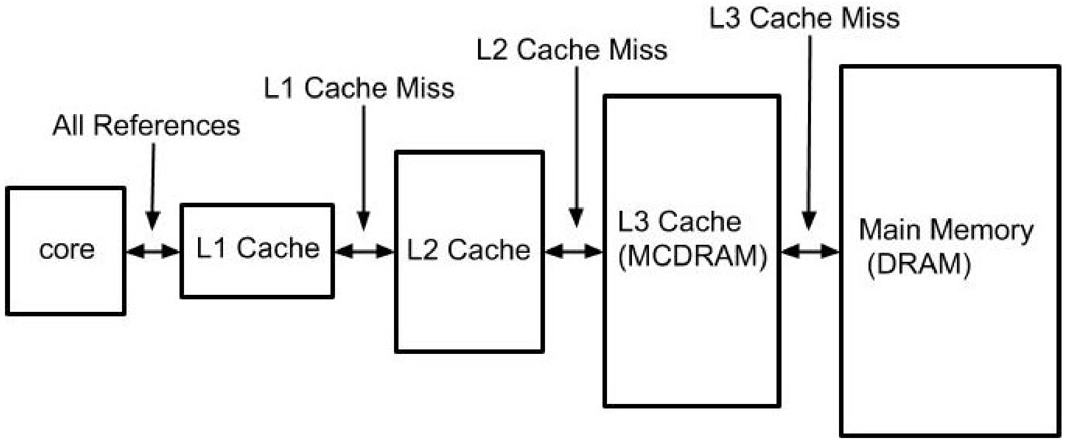


Figure 11: Illustration of KNL's memory system with MCDRAM as cache.

between kinds of memory. The kinds of memory are defined by operating system memory policies that have been applied to virtual address ranges. Memory characteristics supported by memkind without user extension include control of NUMA and page size features. The jemalloc non-standard interface has been extended to enable specialized arenas to make requests for virtual memory from the operating system through the memkind partition interface. Through the other memkind interfaces the user can control and extend memory partition features and allocate memory while selecting enabled features" [45].

Implicit. Memkind library manages memory allocations and offloads to the HBM automatically depending on the declared preferences, such as minimum memory size, as environment variables. As a result, it requires no changes to be made on the code to use the HBM.

Explicit. Memory can be offloaded to the HBM by specifying it through code using memkind Library. Hence, requiring code changes in thr program to declare which memory allocations to be offloaded to the HBM, by using for example `hbw_malloc` instead of `malloc` which allocates in the DDR4 RAM.

3.3.3 Hybrid

The best of both worlds. It allows the HBM to be partitioned to part cache and part flat by specifying a ratio between the two, either 75% - 25%, 50% - 50%, or 25% - 75%.

3.4 Cluster Mode

There are three main cluster modes, All-to-all (a2a), Quadrant (quad), Hemisphere (hemi), Sub-NUMA Clustering 4 (snc-4), and Sub-NUMA Clustering 2 (snc-2). In each of these cluster modes, there are advantages and disadvantages based on the application and the differences it introduces to memory and memory access in KNL. Before tackling the differences between the modes, an explanation of Tag Directory (TD) and CHA is needed.

- **Tag Directory**. It keeps track whether data is in L2 cache and which tile's L2 cache has that data.
- **Distributed Tag Directory**. Each tile has an equal portion of the address space and tile's CHA services queries about its portion of the tag directory.

A detailed description of the cluster modes can be found in [17], [32]. An abstract view of the differences between the cluster modes is presented through the following scenario:

A process running on a specific tile requesting data from the memory address, given that it is not present in the local cache. The tile's CHA queries the Distributed Tag Directory resulting in the tile requesting the data. However, assuming it is not present in the cache, it requests the data from the memory controller responsible for this address. Figure 12 illustrates the example for each of the cluster modes.

3.4.1 All-to-all

In this cluster mode, memory addresses are uniformly distributed across all TDs plus the memory (MCDRAM and DDR) is set to UMA. Any tile may request data at an address that is tracked by a CHA in any part of the chip, and the memory location may reside in any part of the chip. It can route physical addresses from any tile to any CHA, to any memory controller. As a result, on average a L2 cache miss transaction will traverse longer distance since locality is not targeted. To sum up, it lacks any affinity between the tile, TD, and memory.

3.4.2 Quadrant

It is a virtual concept, not a hardware property, that divides the tiles into 4 virtual quadrants. It is much like a2a, where memory is set to UMA, memory addresses are distributed uniformly across all TDs, and it can route physical addresses from any tile to any CHA, to any memory controller. But, having four quadrants introduces affinity in CHA and memory. Meaning, data associated to a TD will be in the same quadrant that the TD is located. To sum up, it introduces affinity between TD and memory.

3.4.3 Hemisphere

It is a variant of quad. The only difference is that instead of dividing the tiles into 4 groups, it divides the tiles into 2 virtual halves, called hemispheres.

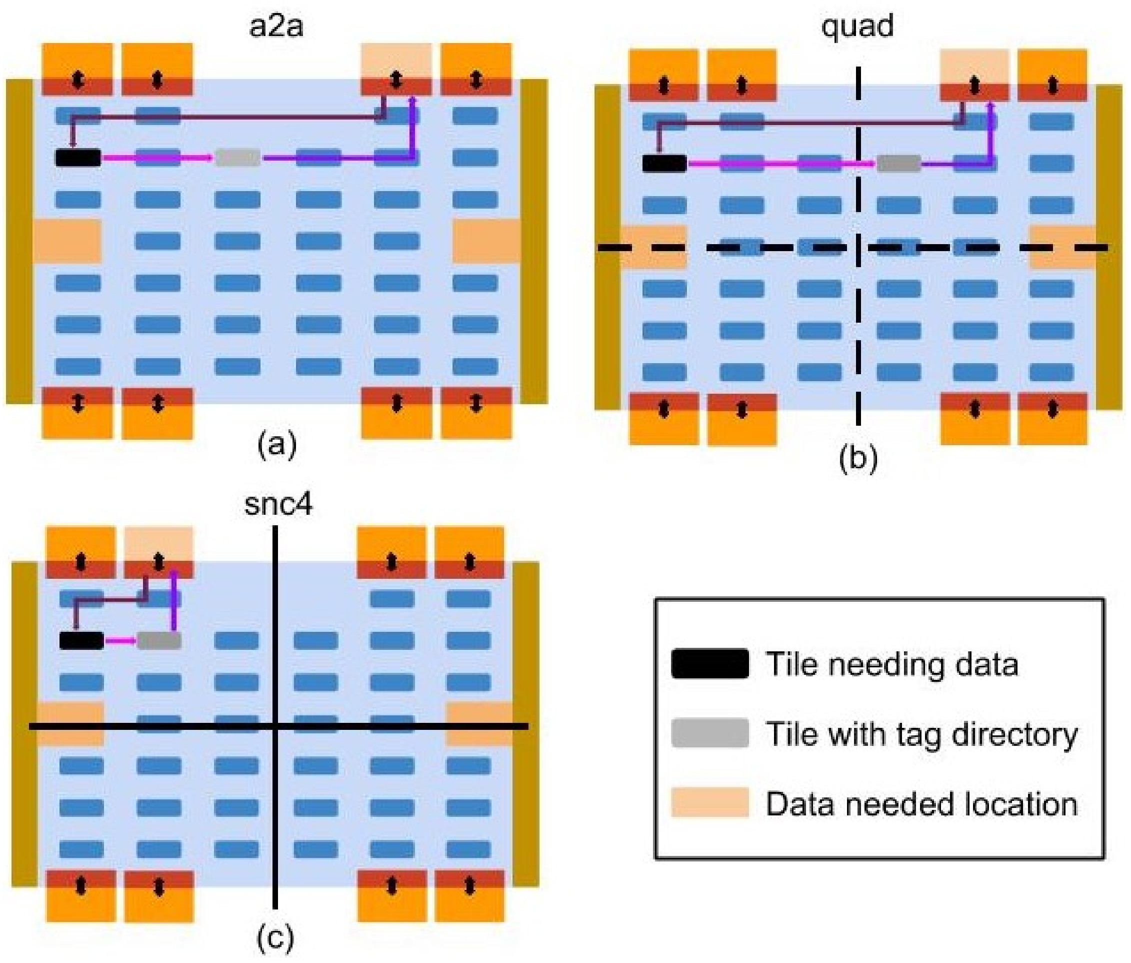


Figure 12: Illustration of L2 cache miss scenario for each cluster mode.

3.4.4 Sub-NUMA-4

Unlike quad, snc-4 is not only based on software; it divides the tiles into 4 clusters resulting in separate cache-coherent clusters. The memory addresses are distributed such that a continuous region of memory is mapped into each cluster and the TD for the memory addresses mapped in a cluster are in CHAs that are also within the same cluster. Meaning, the memory access transactions are completed locally within the same cluster. To sum up, it introduces affinity between tile, TD, and memory.

3.4.5 Sub-NUMA-2

It is a variant of snc-4. The only difference is that instead of dividing the tiles into 4 clusters, it divides the tiles into 2 clusters.

4 MLEM Algorithm

The most widely used iterative reconstruction methods for Emission Tomography is the Maximum Likelihood (ML) Reconstruction using the Expectation Maximization (EM) algorithm [46], which was proposed by Shepp et. al. [7] in 1982. The algorithm tries to solve a system of linear equations in iterative manner, which is represented in Equation 1, where N: number of voxels, M: number of detector pairs, f: 3D image that has to be reconstructed of size N, A: 2D system matrix of size $N \times M$ which describes the geometrical and physical properties of the scanner (a_{ij} represents the probability of a gamma photon discharge from a voxel j being recorded by a given pair of detectors i), g: measured data of size M.

$$f_j^{(q+1)} = \frac{f_j^q}{\sum_{l=1}^{N} a_{lj}} \sum_{i=1}^{N} \left(a_{ij}\left(\frac{g_i}{\sum_{k=1}^{M} a_{ik} f_k^q}\right)\right) \tag{1}$$

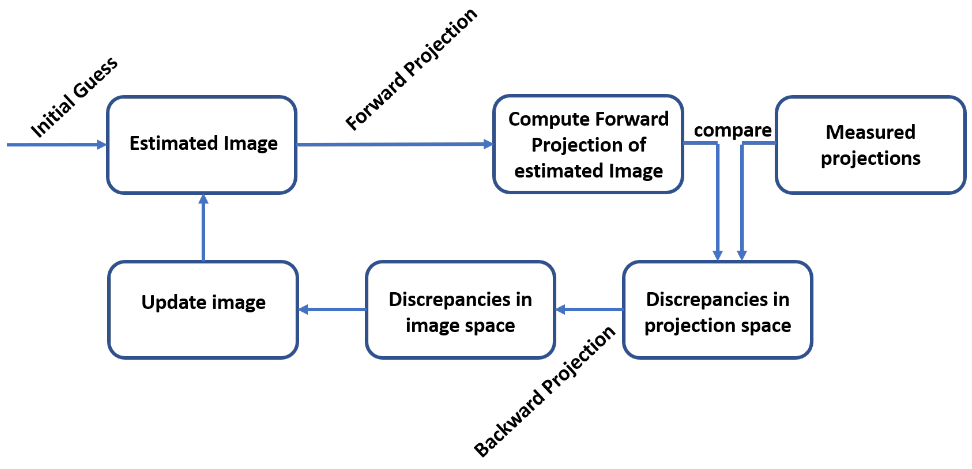

Initial Guess
Estimated Image
Forward Projection
Compute Forward Projection of estimated Image
compare
Measured projections
Update image
Discrepancies in image space
Discrepancies in projection space
Backward Projection

Figure 13: Flowchart of MLEM Algorithm.

Figure 13 is a flowchart of the algorithm [47], which is made of the following steps:

- **Initial image estimate**: sum elements of g, measured data, divided by sum of A, geometry matrix, elements, attaining Equation 2.

$$
\begin{aligned}
f^0 &= \frac{\sum_{i=1}^{N} g_i}{\sum_{j=1}^{M} n_j} \\
where, n_j &= \sum_{l=1}^{N} a_{jl}
\end{aligned} \tag{2}
$$

- **Forward Projection**: multiplication of A and the image vector f, attaining Equation 3.

$$w_i = \sum_{k=1}^{M} a_{ik} f_k^q \tag{3}$$

- **Comparison**: comparison based on the calculated correlation between the actual measurement, g and forward projection, w, attaining Equation 4.

$$c_i = \frac{g_i}{w_i} \tag{4}$$

- **Backward Projection**: multiplication of transpose of A and correlation vector, c, generating update vector, u, which used in the next step to continue to convergence, attaining Equation 5.

$$u_j = \sum_{l=1}^{N} a_{ij} c_i \tag{5}$$

- **Update Image**: image estimation updated based on update, u, and norm, n, vectors, attaining Equation 6

$$f_j^{q+1} = \frac{f_j^q}{n_j} u_j \tag{6}$$

Based on Figure 13 and Equation 1, 3, and 5, a general MLEM, forward projection, and backward projection algorithms can be derived, shown in Algorithm 3, 4, and 5 respectively. The MLEM algorithm is considered expensive due to the sparse matrix multiplication, which occur in forward and backward projection. Sparse matrix multiplication runtime is driven by the limitation of bandwidth, loading the matrix to the memory. Chapter 5 discusses runtime, bandwidth, and scalability in more details.

The rest of this chapter is organized as follows: Section 4.1 provides a background on sparse matrix storage format, since matrix A is a large sparse matrix. Section 4.2 specifies a previous implementation for MLEM, which is based on the figures and equations presented. Finally, Section 4.3 discusses the new implementation of MLEM in terms of the code changes and the reasoning behind it.

Algorithm 3 MLEM Algorithm

```
1: function MLEM($A, g, nIterations$)
2:     initalizeImage($A, g, n, image$)
3:     for ($iteration < nIterations$) do
4:         forwardProjection($A, image, w$)
5:         correlation($w, g, c$)
6:         backwardProjection($A, c, u$)
7:         updateImage($u, n, image$)
8:     end for
9: end function
```

Algorithm 4 Forward Projection Algorithm

```
1: function FORWARDPROJECTION($A, image, w$)
2:     for (row in A.rows) do
3:         $start = A.rowIndexVector[row], end = A.rowIndexVector[row + 1]$
4:         for (rowIndex in [start, end)) do
5:             $temp+ = A.value[rowIndex] \times image[A.column[rowIndex]]$
6:         end for
7:         $w[row] = temp$
8:     end for
9: end function
```

Algorithm 5 Backward Projection Algorithm

```
1: function BACKWARDPROJECTION($A, c, u$)
2:     for (row in A.rows) do
3:         $start = A.rowIndexVector[row], end = A.rowIndexVector[row + 1]$
4:         for (rowIndex in [start, end)) do
5:             $u[A.column[rowIndex]]+ = A.value[rowIndex] \times c[row]$
6:         end for
7:     end for
8: end function
```

4.1 Sparse Matrix Storage Format

Typically a matrix is stored as one continuous block of memory as an array of the matrix's dimensions. This representation allows fast access on costs of storage and space. In the case of huge sparse matrices, there are techniques to address storage format compressing these matrices. The technique can be applied on a 2D matrix generating a sparse matrix; moreover, there are multiple formats to compress and store the sparse matrix. Although this paper summarizes the several storage formats, a more detailed and comprehensive explanation can be found in [48].

Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) are widely used sparse matrix storage format. Both of these formats are based on translating a 2D sparse matrix into three vectors of Rows, Columns, and Values, storing the nonzero elements of the matrix for compressed storage. Each of these formats have its own strengths, which are explained below with an example based on Matrix *M* (7).

$$M = \begin{pmatrix} 12 & -2 & 0 & 6 & 0 \\ -3 & 11 & 0 & 0 & 0 \\ 0 & 0 & -8 & 10 & 7 \\ -9 & 0 & 3 & 5 & 0 \\ 0 & 2 & 0 & 0 & -1 \end{pmatrix} \tag{7}$$

4.1.1 Compressed Sparse Row

The format is based on row indices matrix compression, hence the name. Meaning, compressing the row vector into indices indicating the columns and the nonzero values appearing in the matrix in the order from left-to-right top-to-bottom. Since the storage format is a data model, there are multiple interpretations and implementations that could be derived. This paper presents two variants, Coupled Row Index Vectors and Single Row Index Vector. Both consist of a *Columns* and a *Value* vectors of the nonzero elements, which length is equal to the number of nonzero elements in the matrix; however, their row index representation differ:

- **Coupled Row Index Vectors**: It consists of four vectors: *Columns*, *Values*, *Row Pointer Start*, and *Row Pointer End*. The two vectors, *Row Pointer Start* and *Row Pointer End*, are coupled to map each row starting and ending indices of *Columns* and *Values* vectors. Meaning that, both vectors, *Row Pointer Start* and *Row Pointer End* have the same length which is equal to the number of rows in the matrix. In more depth, *Row Pointer Start* of row *x* represents the first element index in the *Columns* and *Values*; while *Row Pointer End* of row *x* last element index in the *Columns* and *Values* vectors. Table 1 is an example of Matrix M (7).

- **Singular Row Index Vector**: It consists of three vectors, *Values*, *Columns*, and *Row Pointer*. Both *Values* and *Columns* vectors are the same as described in **Coupled**

Values	12	-2	6	-3	11	-8	10	7	-9	3	5	2	-1
Columns	0	1	3	0	1	2	3	4	0	2	3	1	4
Row Pointer Start	0	3	5	8	11								
Row Pointer End	3	5	8	11	13								

Table 1: Coupled Row Index Vector CSR representation of Matrix (7).

Row Index Vectors, while *Row Pointer* is combining both *Row Pointer Start* and *Row Pointer End*. Thus, the length of *Row Pointer* is equal to the number of rows in the matrix + 1. Table 2 is an example of Matrix M (7).

Values	12	-2	6	-3	11	-8	10	7	-9	3	5	2	-1
Columns	0	1	3	0	1	2	3	4	0	2	3	1	4
Row Pointer	0	3	5	8	11	13							

Table 2: Singular Row Index Vector CSR representation of Matrix (7).

4.1.2 Compressed Sparse Column

Similarly to CSR, CSC is based on compressing the column indices, hence the name. The main differences are 1) it has vectors representing column indicies instead of row indices 2) order of compressing the elements is top-to-bottom left-to-right instead of left-to-right top-to-bottom. Table 3 and Table 4 are examples of CSC format Coupled and Singular Column Index Vector respectively.

Values	12	-3	-9	-2	11	2	-8	3	6	10	5	7	-1
Rows	0	1	3	0	1	4	2	3	0	2	3	2	4
Column Pointer Start	0	3	6	8	11								
Column Pointer End	3	6	8	11	13								

Table 3: Coupled Column Index Vector CSC representation of Matrix (7).

4.2 Existing Implementation

4.2.1 Matrix Representation

The matrix representation in the original code is based on CSR format with few difference, namely:

- **Row index representation**: Based on *Singular Row Index* representation; however without the first element of the vector, which is by default is a zero (0).

Values	12	-3	-9	-2	11	2	-8	3	6	10	5	7	-1
Rows	0	1	3	0	1	4	2	3	0	2	3	2	4
Column Pointer	0	3	6	8	11	13							

Table 4: Singular Column Index Vector CSC representation of Matrix (7).

- **Columns and values representation**: Instead of having a vectors for each, it uses one vector of tuples for both of them since the vector indices correspond to each other.

(values, columns)	(12, 0)	(-2, 1)	(6, 3)	(-3, 0)	(11, 1)	(-8, 2)	(10, 3)	(7, 4)	(-9, 0)	(3, 2)	(5, 3)	(2, 1)	(-1, 4)
Row Index	3	5	8	11	13								

Table 5: Example of the existing code CSR matrix representation of Matrix (7).

The code corresponding to that representation is shown in Code Snippet 1.

```
template<typename T> class RowElement
{
  uint32_t column;
  T value;
}

class Csr4Matrix
{
  uint32_t nRows;
  uint32_t nColumns;

  uint64_t* rowidx;
  RowElement<float>* data;
}
```

Snippet 1: Existing code of CSR matrix implementation.

4.2.2 OpenMP

The code does not exactly support thread pinning, due to the matrix representation and thread assignment, and certain KNL hardware configuration. Forward and backward projections code are shown in Code Snippet 3 (Left) and 4 (Left) respectively. The full code is available online and can be found on LRZ GitLab[1].

4.2.3 MPI-only

The code contains unnecessary complexity, does not support hybrid (MPI and OpenMP) approach, and other minor issues. Forward and backward projections code are shown in Code Snippet 5 (Left) and 6 (Left) respectively. The full code is available online and can be found on LRZ GitLab[1].

4.3 Proposed Implementation

4.3.1 Matrix Representation

The existing matrix implementation is good from a Software Engineering point of view. Having said that, it is over-engineered and as a result it increases the complexity when dealing with multiprocessing and multithreading. Therefore, we propose a simpler and more straight-forward implementation based on CSR storage format that is:

- Support threading and multiprocessing.
- General; it could be used without any changes in different implementations.

The proposed OpenMP, MPI-only, and Hybrid (MPI-OpenMP) implementation use the following matrix structure based on *Singular Row Index*:

```
struct CSR
{
  uint32_t startRow, endRow;
  int nRowIndex, nColumn;

  uint32_t *columns;
  uint64_t *rowIndex;
  float *values;
}
```

On the other hand, the MKL offers routines for creating a handle for CSR matrix, show in Code Snippet 2, as well as functions to perform operations such as matrix-vector multiplication. Thus, MKL implementation uses the provided library, which is based on *Coupled Row Index*.

[1]https://gitlab.lrz.de/lrr-tum/research/mlem/commit/f1b5d3827adb9bece7c12010c9c9934123868e0b

```
sparse_status_t mkl_sparse_s_create_csr (
  sparse_matrix_t *A, sparse_index_base_t indexing,
  MKL_INT rows, MKL_INT cols,
  MKL_INT *rows_start, MKL_INT *rows_end,
  MKL_INT *col_indx, float *values
)
```

Snippet 2: MKL function to create handle for a CSR format matrix.

4.3.2 OpenMP

Given the matrix implementation change, it was necessary to change the OpenMP implementation to:

- Be simpler, which makes it easier to read, understand and modify.
- Support thread pinning, which is affected by matrix partitioning.
- Support KNL hardware features, specifically HBM.

Matrix Partitioning

In the existing implementation, the main thread allocates the memory for the whole matrix, then assigns ranges for each. Given KNL architecture, this implementation could be problematic to certain cluster modes, such as snc-4. Moreover, it is not efficient since the memory could be located rather far from the tile. Therefore, each thread should allocate its own chunk of the matrix in the memory, which mostly would be the nearest memory it can access, and store it in CSR format. Algorithm 6 highlights the logic explained.

OpenMP Code

The change in the matrix representation effects the functions that takes it as an input, which most importantly are forward projection and backward projection. Code Snippet 3 (Right) and 4 (Right) shows the proposed implementation for forward and backward projection respectively; moreover, it also highlights the changes made in comparison to the existing implementation in a side-by-side comparison style.

4.3.3 MPI-only and Hybrid

Similarly to OpenMP, it is necessary for the new MPI implementation to:

- Be simpler, which makes it easier to read, understand and modify.
- Support process pinning, which is affected by matrix partitioning.

Algorithm 6 OpenMP Matrix Partitioning

```
 1: function PARTITION($A$)
 2:     Allocate shared array storing all CSR struct the pointers    ▷ Master thread only
 3:     Allocate CSR struct, $thread\_CSR$                           ▷ Ran by each thread
 4:
 5:     Calculate: average = total elements / number of threads
 6:     for ( row in A.rows ) do
 7:         sum += elements in row
 8:         if ($sum >= average$) then
 9:             Assign thread range                                  ▷ Starting and ending row
10:             $threadID$++; $sum$=0;
11:         end if
12:     end for
13:
14:     function BUILD_CSR                                           ▷ Ran by each thread
15:         for ( row in [startRow, endRow) ) do
16:             Copy row index, values, columns from $A$ to $thread\_A$
17:         end for
18:     end function
19: end function
```

- Support Hybrid approach by design; hence, thread pinning.
- Support KNL hardware features, specifically HBM.

Matrix Partitioning

It is an adaptation of OpenMP matrix partitioning to be used for MPI-only and Hybrid. Theoretically, there is not much difference other than some variables and memory accesses are not shared. The code for matrix partitioning for MPI-only and Hybrid can be derived from OpenMP partitioning algorithm, Algorithm 6.

MPI-only and Hybrid Code

Similarly to OpenMP implementation, forward and backward projection are affected by the new matrix representation. Moreover, in the case of Hybrid, forward and backward projection is a combination of MPI-only and OpenMp. Code Snippet 5 (Right) and 6 (Right) shows the proposed implementation for forward and backward projection respectively; moreover, it also highlights the changes made in comparison to the existing implementation in a side-by-side comparison style.

4.3.4 MKL

An implementation using MKL routines was crucial for comparison. Since MKL has its own CSR structure and routines, forward and backward projection are based on MKL matrix multiplication function provided by the library; making the code significantly different than the other implementations, which is shown in Code Snippet 7.

```
void
forwardProjection(A, image, w) {
 #pragma omp parallel for



 for (uint32_t row = 0;
  row < A.rows();
  ++row) {
  float res = 0.0;

  std::for_each(A.beginRow2(row),
   A.endRow2(row),
   [&](RowElement<float>& e) {
    res += e.value() *
     image[e.column()];
   });
  w[row] = res;

 }
}
```

```
void
forwardProjection(image, w) {
 #pragma omp parallel
 {
  int tidx = omp_get_thread_num();
  A_thread = global_A[tidx];
  uint32_t startRow, endRow;
  for (row = startRow;
   row < endRow;
   ++row) {
   float res = 0.0;
   uint32_t startCVIndex, endCVIndex;
   for (j = startCVIndex;
    j < endCVIndex;
    j++)
    res += A_thread.value[j] *
     image[A_thread.columns[j]]

   w[row] = res;
  }
 }
}
```

Snippet 3: OpenMP forward projection implementation side-by-side comparison. Left side shows the existing implementation; while, the right side shows the proposed implementation. The grey highlights details the changes.

```
void
backwardProjection(A, c, u) {
 #pragma omp parallel
 {
  Vector<float> temp(u.size(),0);


  #pragma omp for schedule(dynamic)
  for (uint32_t row = 0;
    row < A.rows();
    ++row) {

    std::for_each(
     A.beginRow2(row),
     A.endRow2(row),
     [&](RowElement<float>& e) {
      temp[e.column()]
       += e.value() * c[row];
     });
  }
  #pragma omp critical
  {
   size_t i = 0;
   for (; i < u.size(); ++i)
    u[i] += temp[i];
  }
 }
}
```

```
void
backwardProjection(c, u) {
 #pragma omp parallel
 {
  Vector<float> temp(u.size(),0);
  int tidx = omp_get_thread_num();
  A_thread = global_A[tidx];
  uint32_t startRow, endRow;
  for (row = startRow;
   row < endRow;
   ++row) {

   uint32_t startCVIndex, endCVIndex;
   for (j = startCVIndex;
    j < endCVIndex;
    j++)
    temp[A_thread.columns[j]] +=
    A_thread.value[j] * c[row]

  }
  #pragma omp critical
  {
   size_t i = 0;
   for (; i < u.size(); ++i)
    u[i] += temp[i];
  }
 }
}
```

Snippet 4: OpenMP backward projection implementation side-by-side comparison. Left side shows the existing implementation; while, the right side shows the proposed implementation. The grey highlights details the changes.

```
void
forwardProjection(A, image, w) {
 std::fill(w, w + w.size(), 0.0);

 auto& myrange = ranges[mpi.rank];
 matrix.mapRows(myrange.start,
  myrange.end - myrange.start);


 for (int row = myrange.start;
  row < myrange.end;
  ++row) {
  float res = 0.0;

  std::for_each(A.beginRow2(row),
   A.endRow2(row),
   [&](RowElement<float>& e) {
    res += e.value() *
     image[e.column()];
   });
   w[row] = res;
  }

 MPI_Allreduce(MPI_IN_PLACE, w,
  w.size(), MPI_FLOAT, MPI_SUM,
  MPI_COMM_WORLD);
}
```

```
void
forwardProjection(image, w) {
 std::fill(w, w + w.size(), 0.0);

 #pragma omp parallel
 {
  int tidx = omp_get_thread_num();
  A_thread = global_A[mpi.rank][tidx];
  uint32_t startRow, endRow;
  for (row = startRow;
   row < endRow;
   ++row) {
   float res = 0.0;
   uint32_t startCVIndex, endCVIndex;
   for (j = startCVIndex;
    j < endCVIndex;
    j++)
    res += A_thread.value[j] *
     image[A_thread.columns[j]]

  w[row] = res;
 }

 MPI_Allreduce(MPI_IN_PLACE, w,
  w.size(), MPI_FLOAT, MPI_SUM,
  MPI_COMM_WORLD);
}
```

Snippet 5: MPI/Hybrid forward projection implementation side-by-side comparison. Left side shows the existing implementation; while, the right side shows the proposed implementation. The grey highlights details the changes.

```
void
backwardProjection(A, c, u) {
 std::fill(u, u + w.size(), 0.0);

 auto& myrange = ranges[mpi.rank];
 matrix.mapRows(myrange.start,
  myrange.end - myrange.start);


 for (int row = myrange.start;
  row < myrange.end;
  ++row) {


  std::for_each(
   A.beginRow2(row),
   A.endRow2(row),
   [&](RowElement<float>& e) {
    u[e.column()] +=
     e.value() * c[row];
   });
 }





 MPI_Allreduce(MPI_IN_PLACE, ,
  u.size(), MPI_FLOAT, MPI_SUM,
  MPI_COMM_WORLD);
}
```

```
void
backwardProjection(c, u) {
 std::fill(u, u + w.size(), 0.0);
 #pragma omp parallel
 {
  Vector<float> temp(u.size(),0);
  int tidx = omp_get_thread_num();
  A_thread = global_A[mpi.rank][tidx];
  uint32_t startRow, endRow;
  for (row = startRow;
   row < endRow;
   ++row) {

   uint32_t startCVIndex, endCVIndex;
   for (j = startCVIndex;
    j < endCVIndex;
    j++)
    temp[A_thread.columns[j]] +=
    A_thread.value[j] * c[row]
  }
  #pragma omp critical
  {
   size_t i = 0;
   for (; i < u.size(); ++i)
     u[i] += temp[i];
  }
 }
 MPI_Allreduce(MPI_IN_PLACE, ,
  u.size(), MPI_FLOAT, MPI_SUM,
  MPI_COMM_WORLD);
}
```

Snippet 6: MPI/Hybrid backward projection implementation side-by-side comparison. Left side shows the existing implementation; while, the right side shows the proposed implementation. The grey highlights details the changes.

```
sparse_matrix_t A;
struct matrix_descr descr;

void forwardProjection(image, w)
{
  mkl_sparse_s_mv(SPARSE_OPERATION_NON_TRANSPOSE, 1.0, A,
    descr, image, 0.0, w);
}

void backwardProjection(c, u)
{
  std::fill(u, u + u.size(), 0.0);

  mkl_sparse_s_mv(SPARSE_OPERATION_TRANSPOSE, 1.0, A, descr, c, 0.0, u);
}
```

Snippet 7: MKL implementation for forward and backward projection.

5 Evaluation

This chapter is organized as follows: Section 5.1 describes the datasets and parameters for running MLEM. Section 5.2 discusses the different approaches based on parallelism concepts and hardware configurations to obtain the best performance. Section 5.3 details 20 different configurations to be compared yield from the previous section. Section 5.4 portrays and represents the results of the 20 different setups. Section 5.5 analyzes and provides reasoning for the setup performance as well as highlights the best performing setup. Finally, Section 5.6 discusses limitations in this work.

5.1 Data

As discussed previously in Chapter 4 and shown in Algorithm 3, MLEM requires four parameters:

- *nIteration*: Number of iterations before it terminates. It is set to 50 iterations.
- *image*: Initial estimation, which is calculated using Equation 2.
- *g*: Measurement vector generated from the MADPET-2 scanner.
- *A*: System matrix, 2D matrix that describes the geometrical and physical properties of the scanner. Since the measurement were taken by MADPET-2, *A* describes the characteristics of MADPET-2, which consists of voxels divided into a $140 \times 140 \times 140$ grid and 1152 detectors arranged in two concentric rings, as shown in Figure 1. A more detailed characteristics of the *A* presented in Table 6.

5.2 Design

To utilize KNL as efficiently as possible, the effect of different approaches based on various hardware configurations and parallelism concepts on performance are investigated:

- **Affinity (Process/Thread Pinning)**. As mentioned previously, it enables the binding of threads and/or processes to CPUs results in exploiting data locality usually. Moreover, there are multiple strategies for affinity, such as scatter, balanced, and compact.

Parameter	Value
Total Size(Bytes)	12,838,607,884
Rows (Pair of detectors)	1,327,104
Columns (Voxels)	784,000
Total Non Zeros(NNZ)	1,603,498,863
Matrix Density(%)	0.1541
Max Value	8.90422e-05
Min Value	5.50416e-24
Max NNZ in a row	6537
Min NNZ in a row	0
Avg NNZ in a row	1208
Most repeating NNZ in row	0
Occurrence of Most repeating NNZ in row	822530
2nd Most repeating NNZ in row	3
Occurrence of 2nd Most repeating NNZ in row	2034
Max NNZ in a column	6404
Min NNZ in a column	0
Avg NNZ in a column	2045
Most repeating NNZ in column	0
Occurrence of Most repeating Mode NNZ in row	215488
2nd Most repeating NNZ in row	231
Occurrence of 2nd Most repeating NNZ in row	260

Table 6: MADPET-2 Matrix Characteristics [49].

Nevertheless, in all proposed implementations excluding MKL, each thread allocates its own part of the matrix, which accentuates the pinning effect on exploiting data locality as well as deals with NUMA memory configuration.

- **Cluster Mode**. As stated in Section 3.4, the modes are either based on NUMA or UMA. Thus, this evaluates the possibility of data locality, introduced by the cluster modes, influence on performance. There are certain type of applications where NUMA results in better performance than UMA and visa versa. Nevertheless, it is important to examine the effect of those modes with the influence of other variables, such as Parallelism Library.
- **HBM Configuration**. As stated in section 3.3, there are three modes for HBM: cache, flat, and hybrid. Considering the proposed implementations, flat mode is better than cache mode since 1) matrix fits in the memory; hence, offloading the matrix to HBM (flat) where the bandwidth is the same as to cache 2) flat mode allows faster writes generated by MLEM matrix-vector multiplication. Given that 1) hybrid is cache and flat mode 2) flat is better than cache for MLEM, then we can deduce that flat performance is better or the same as hybrid. Thus, we are only concerned with flat mode in the evaluation.
- **HBM Usage Model**. As stated in section 3.3, there are two ways to use the HBM in flat mode, Implicit and Explicit. In the Implicit, the matrix and all other variables are allocated in the HBM. While in the Explicit, the implementations have only the matrix to be allocated in the HBM.
- **Parallelism Library**. Performance of the two different parallel paradigms, Message Passing and Shared Memory, and their mix, hybrid approach, is a constant question that shows up in research. In Section 4.3, we presented implementations based on different parallelism approaches and libraries, OpenMP, MPI-only, MKL, and Hybrid (MPI-OpenMP).

5.3 Setup

The different approaches based on various hardware configurations and parallelism concepts discussed in the previous section yield 20 different setups, as shown in Table 7. More details concerning running the implementation spanning thread count to compiler information is explained below:

- **Thread Range**: Since the maximum performance of KNL is achieved using one thread per core (without hyperthreading), the maximum number of threads is 64. The implementations are run over a range of thread numbers starting from 1 to 64 in an increasing order in powers of two, 2^i. Note that the Hybrid implementation runs on two nodes, hence in total the maximum number of threads is 128 and it follows an increasing order of 2^{i+1}. Nevertheless, it is ensured that there is no

Parallelism Library	Cluster Mode	HBM Usage Model	Affinity	Label
MKL	snc-4	Implicit	Yes	mkl_a2a_m_pinning
			No	mkl_a2a_m
Open MP	a2a	Implicit	Yes	omp_a2a_m_pinning
			No	omp_a2a_m
		Explicit	Yes	omp_a2a_x_pinning
			No	omp_a2a_x
	quad	Implicit	Yes	omp_quad_m_pinning
			No	omp_quad_m
		Explicit	Yes	omp_quad_x_pinning
			No	omp_quad_x
	snc-4	Implicit	Yes	omp_snc4_m_pinning
			No	omp_snc4_m
		Explicit	Yes	omp_snc4_x_pinning
			No	omp_snc4_x
MPI-only	a2a	Explicit	Yes	mpi_a2a_x_pinning
	quad	Explicit	Yes	mpi_quad_x_pinning
	snc-4	Explicit	Yes	mpi_snc4_x_pinning
Hybrid	a2a	Explicit	Yes	hyb_a2a_x_pinning
	quad	Explicit	Yes	hyb_quad_x_pinning
	snc-4	Explicit	Yes	hyb_snc4_x_pinning

Table 7: All the different setups with their labels.

imbalance; each node has the same number of threads.

To sum up, the OpenMP, MPI-only, and MKL implementation run with 1, 2, 4, 8, 16, 32, and 64 threads; While, Hybrid runs with 2, 4, 8, 16, 32, 64, and 128 threads.

- **Affinity (Process/Thread Pinning)**: Considering that at maximum thread number is 64, 1 thread per core, the environment variable for affinity set to bind each thread to a core using a compact strategy, `KMP_AFFINITY="granularity=core,compact"`. On the other hand, in the case of no pinning, there is no strategy set.
- **HBM Usage Model**: In the case of Implicit, an environment variable is set for the least size of allocation to be redirected to HBM, `AUTO_HBW_SIZE=1B` plus adding memkind libraries to shared libraries path, `LD_PRELOAD=libautohbw.so:libmemkind.so`.
- **Compiler Flags**: a comprehensive list of the compiler flags is in the Appendix, Table 8.
- **Run command**. The OpenMP and MKL implementations run using `numactl`; while Hybrid and MPI-only implementations run using `mpirun` Version 2017 Update 4 Build 20170817. As mentioned previously, in the case of Implicit, we prepend `LD_PRELOAD=libautohbw.so:libmemkind.so` to link `memkind` for automatic HBM allocation. A comprehensive list of the run commands as well as environment variables are found in Snippet 8 in the appendix.
- **KNL Cluster**: The implementations run on CoolMUC cluster provided by Leibniz Supercomputing Center. An overview of CoolMUC3 characteristic is in the Appendix, Table 9.

5.4 Results

Each setup runs 10 time. During each run, the algorithm records the iteration time for forward projection and backward projection, and the total iteration time. In total generating 500 data points, since the iteration number is set to 50.

To evaluate the all different setups, a measure for performance, scalability, and memory bandwidth is required. From data collected, average runtime, speedup, and memory bandwidth can be calculated as indicators for performance, scalability, and memory bandwidth utilization respectively. Runtime indicates the performance of the algorithm given all the variables and configuration associated with the setups. Speedup indicates the improvement with increasing the number of processors/threads which in turn indicates scalability. Memory bandwidth indicates the KNL's HBM memory utilization as well as indicates the existence of a bottleneck since MLEM is memory-bound algorithm.

Average Runtime

Let *iteration_runtime* denote the time for one iteration of MLEM algorithm loop and *runtime* denote the total time for MLEM algorithm executed for 50 iterations. Then, *average_runtime* is the total *runtime* of 10 runs over 10. Equation 8 defines average runtime given a number of threads and setup, *t* and *s* respectively. Figure 14 provides a full overview of all setups average runtime, but it is rather hard to read. While Figure 15 and 16 breaks down the average runtime into groups for easier reading. The error is not presented in the graphs since the standard deviation for most of the measurements is equal to or less than 1%.

$$average\_runtime(t,s) = \frac{\sum_{i=1}^{10} \left(runtime_i(t,s)\right)}{10} \tag{8}$$

Speed up

Speedup is the runtime using one thread/processor, sequential, over the runtime using *p* processors/threads. Since there are different setups, a common sequential implementation must be defined, *sequential_baseline*. Equation 9 defines speedup, where *s* is setup, *t* number of threads, and *sequential_baseline* is the sequential code runs with HBM in Implicit and cluster mode in a2a. Note, by definition, speedup is upper bound by *p* which is considered the theoretical optimal value. Figure 17 provides a full overview of all setups speedup curves, but it is rather hard to read. While Figure 18 and 19 breaks down the speedup into groups for easier reading.

$$speedup(t,s) = \frac{sequential\_baseline}{average\_runtime(t,s)} \tag{9}$$

Memory Bandwidth

Memory bandwidth is the total amount of memory loaded over the time spent for the operation. Equation 10 defines memory bandwidth, where *memory_size* is the memory loaded for a given function. Based on that, forward and backward projection memory bandwidth can be calculated. Figure 20 provides a full overview of all setups memory bandwidth, but it is rather hard to read. While Figure 21 and 22 breaks down the memory bandwidth into groups for easier reading.

$$bandwidth(t,s) = \frac{average\_runtime(1)}{average\_runtime(t,s)} \tag{10}$$

5.5 Discussion

Figure 14, 17, and 20 indicate performance, scalability, and memory bandwidth utilization of the setups according to the influence of the different approaches, based on various hardware configurations and parallelism concepts, presented previously.

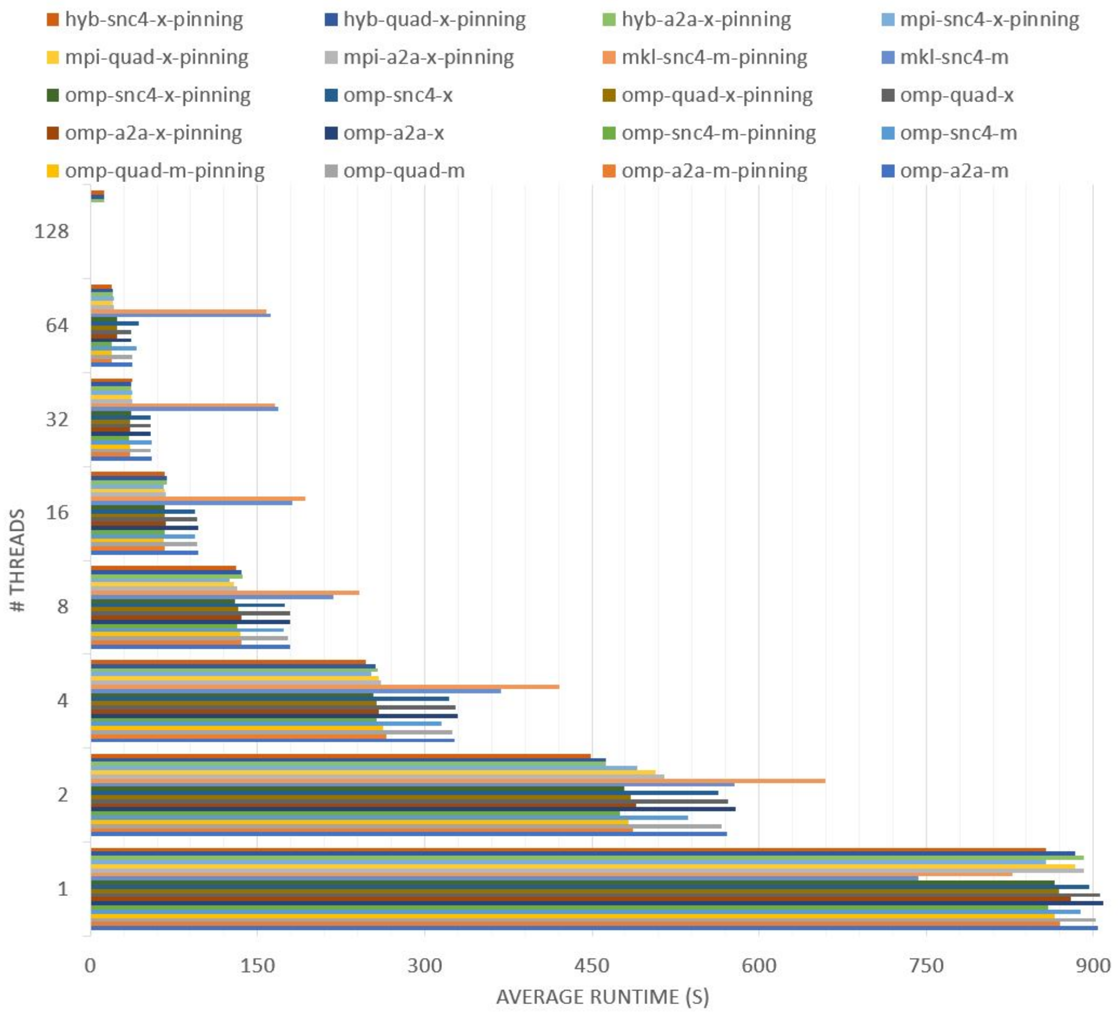
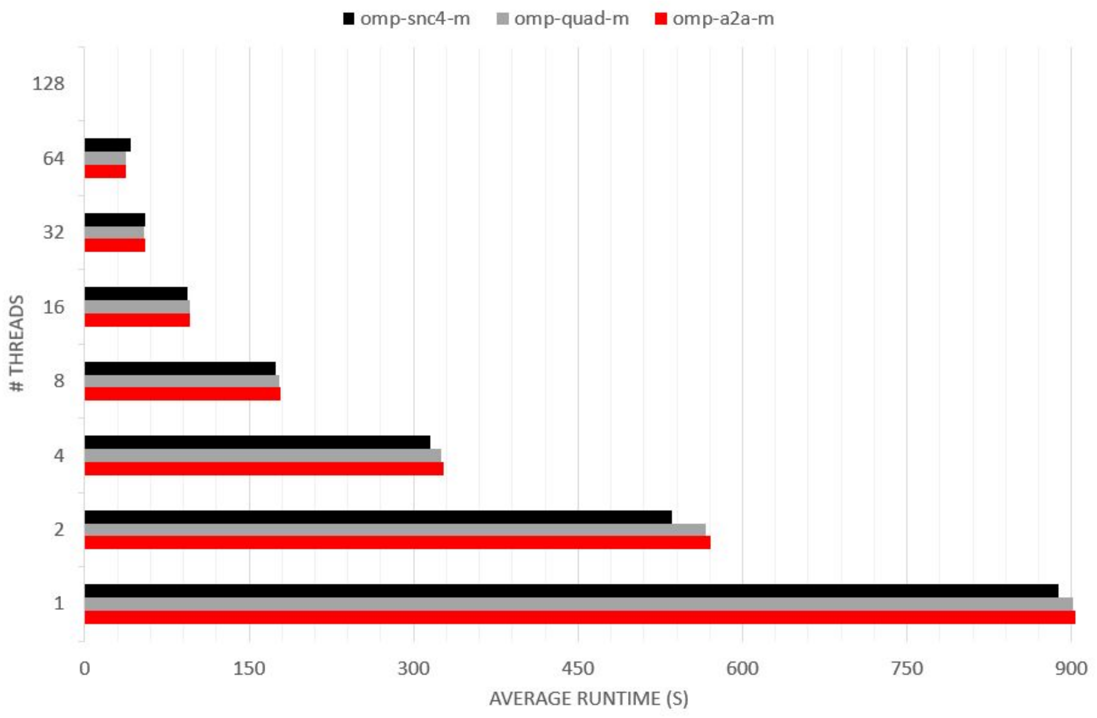
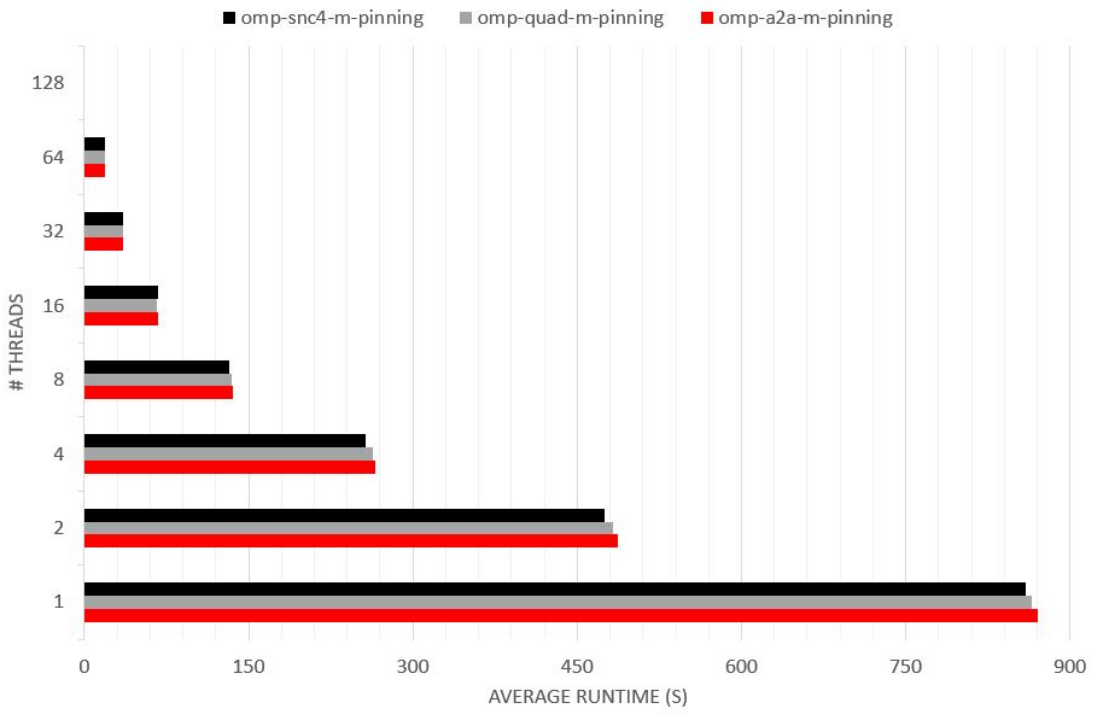


Figure 14: Average runtime achieved by all setups.



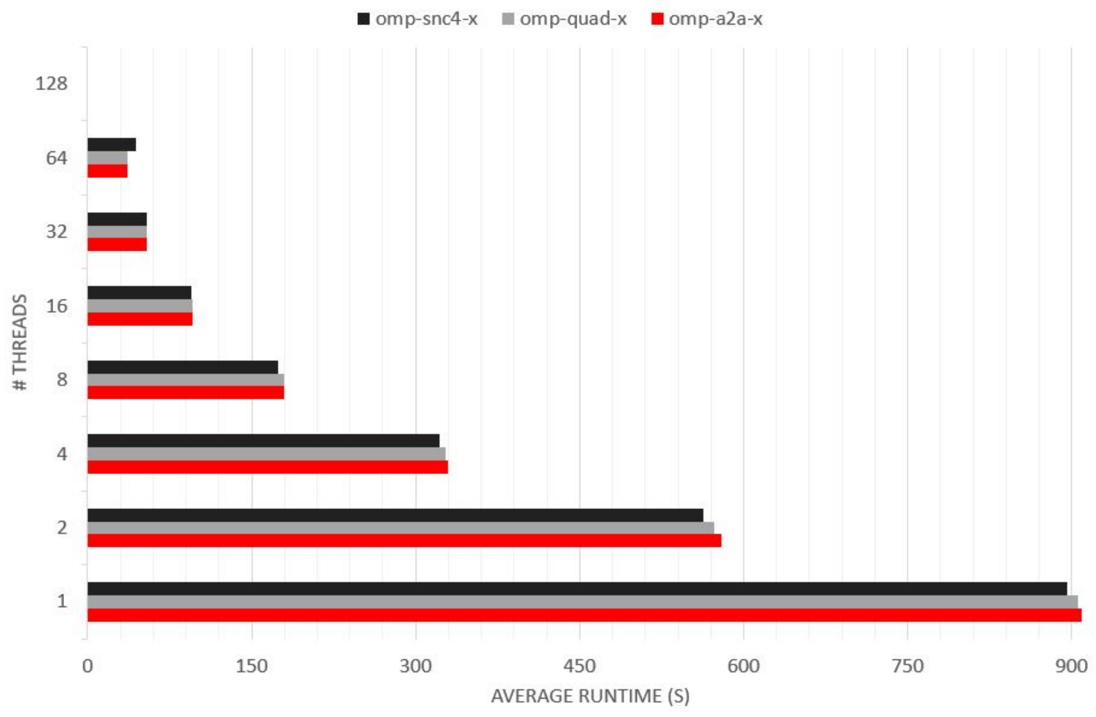
■ omp-snc4-m ■ omp-quad-m ■ omp-a2a-m
# THREADS
128
64
32
16
8
4
2
1
0
150
300
450
600
750
900
AVERAGE RUNTIME (S)

(a)

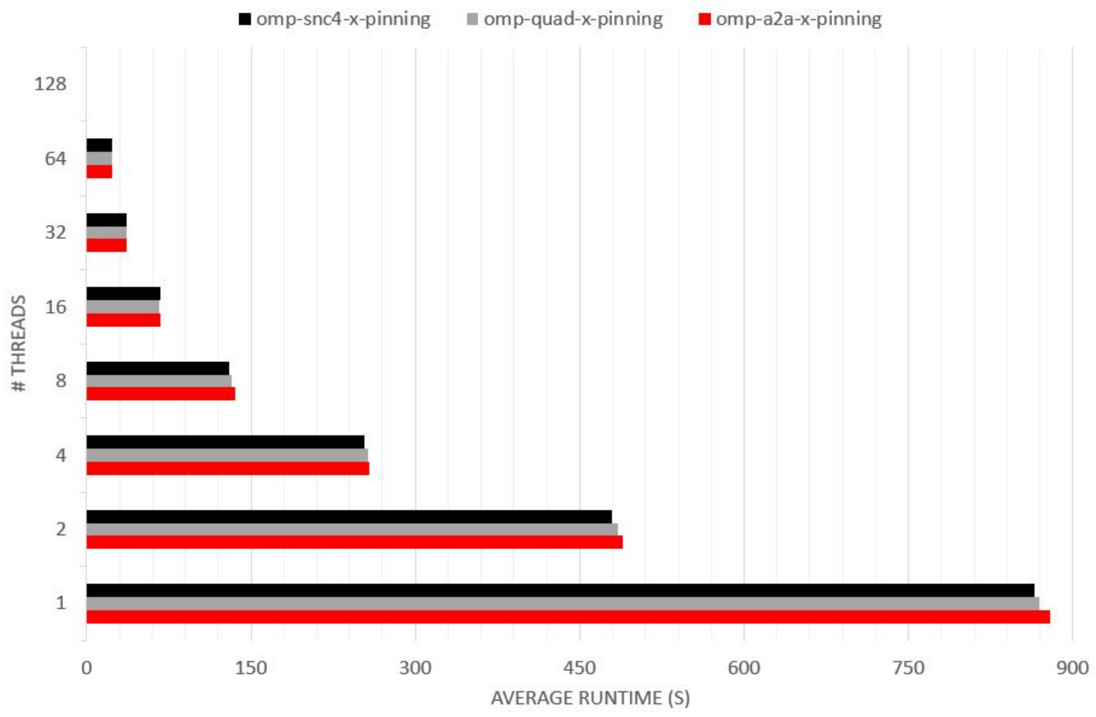


■ omp-snc4-m-pinning ■ omp-quad-m-pinning ■ omp-a2a-m-pinning
# THREADS
128
64
32
16
8
4
2
1
0
150
300
450
600
750
900
AVERAGE RUNTIME (S)

(b)

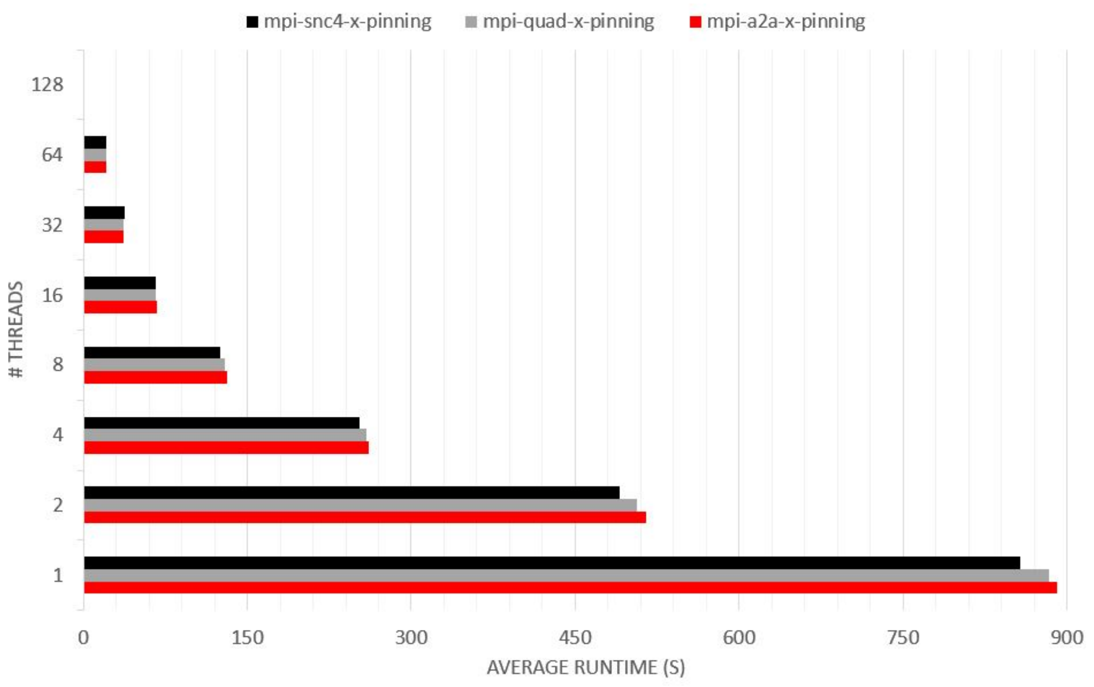


(c)

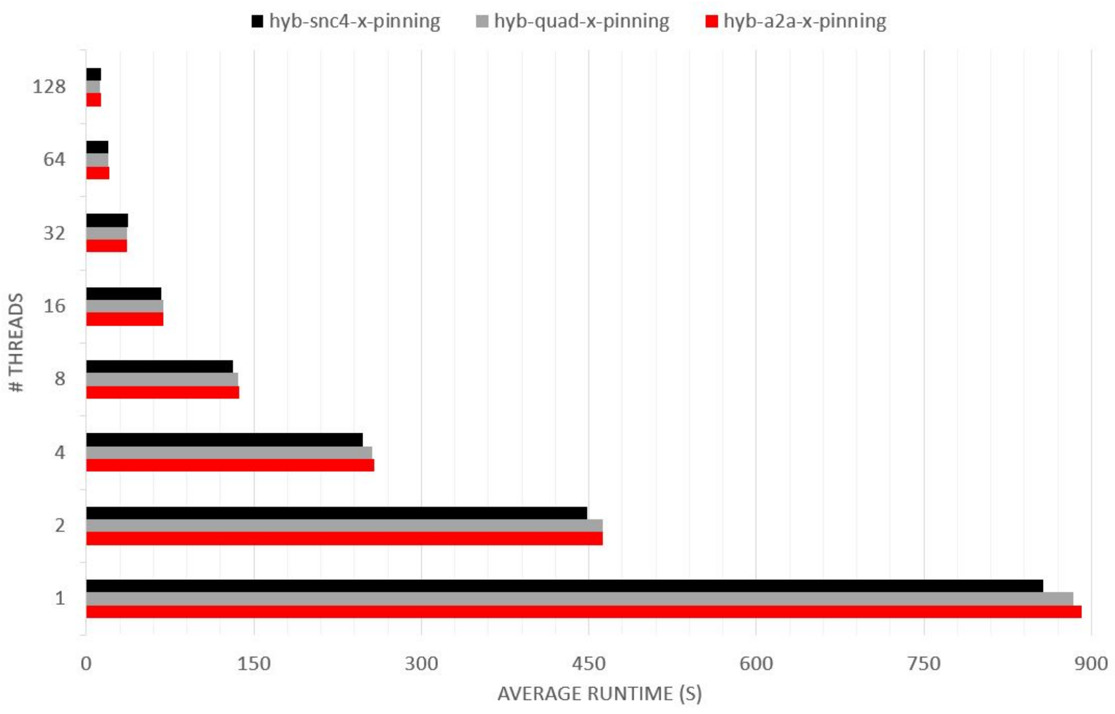


(d)

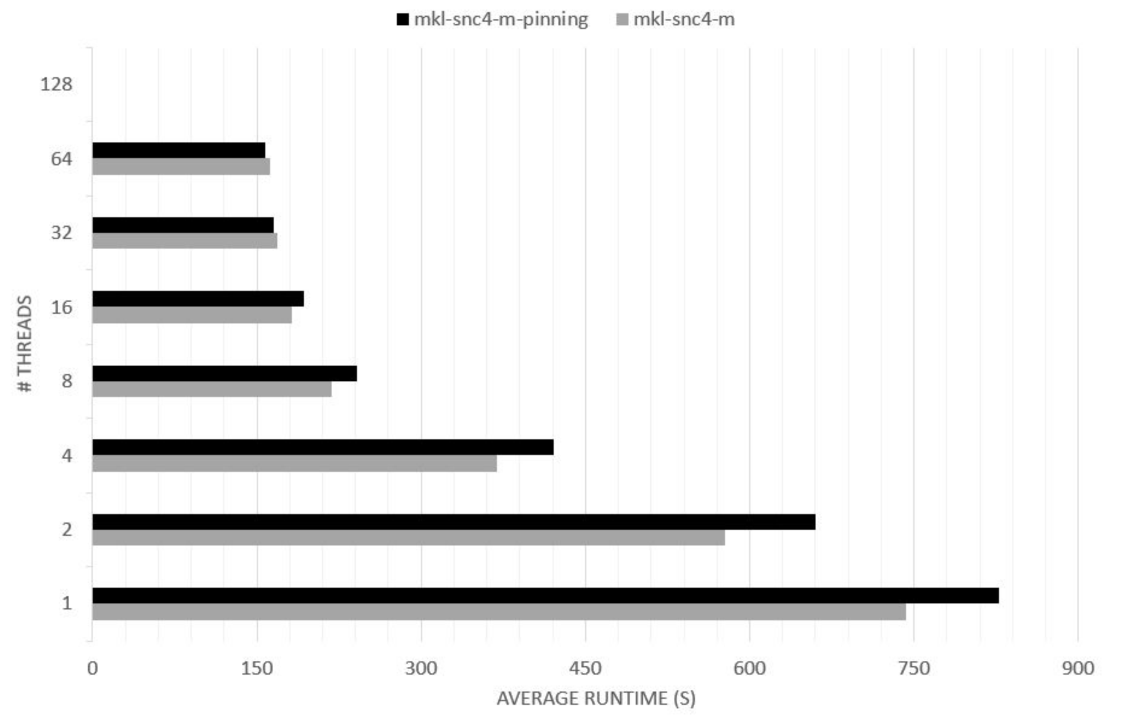
Figure 15: Average runtime achieved by OpenMP setups grouped in (a) Implicit (b) Implicit + Pinning (c) Explicit (d) Explicit + Pinning


mpi-snc4-x-pinning
mpi-quad-x-pinning
mpi-a2a-x-pinning
# THREADS
128
64
32
16
8
4
2
1
0
150
300
450
600
750
900
AVERAGE RUNTIME (S)

(a)


hyb-snc4-x-pinning
hyb-quad-x-pinning
hyb-a2a-x-pinning
# THREADS
128
64
32
16
8
4
2
1
0
150
300
450
600
750
900
AVERAGE RUNTIME (S)

(b)



(c)

Figure 16: Average runtime achieved by (a) MPI-only (b) Hybrid (c) MKL

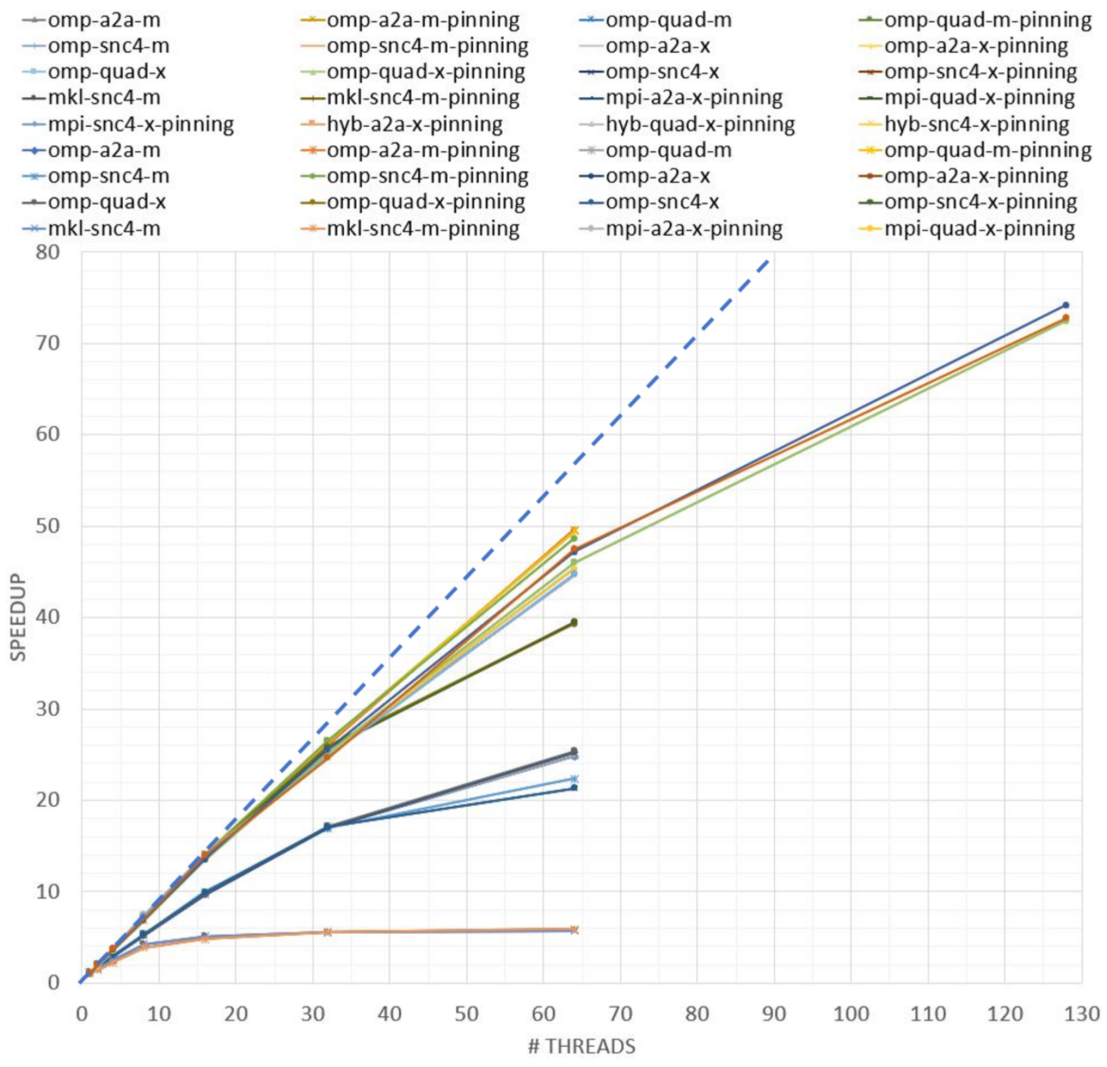
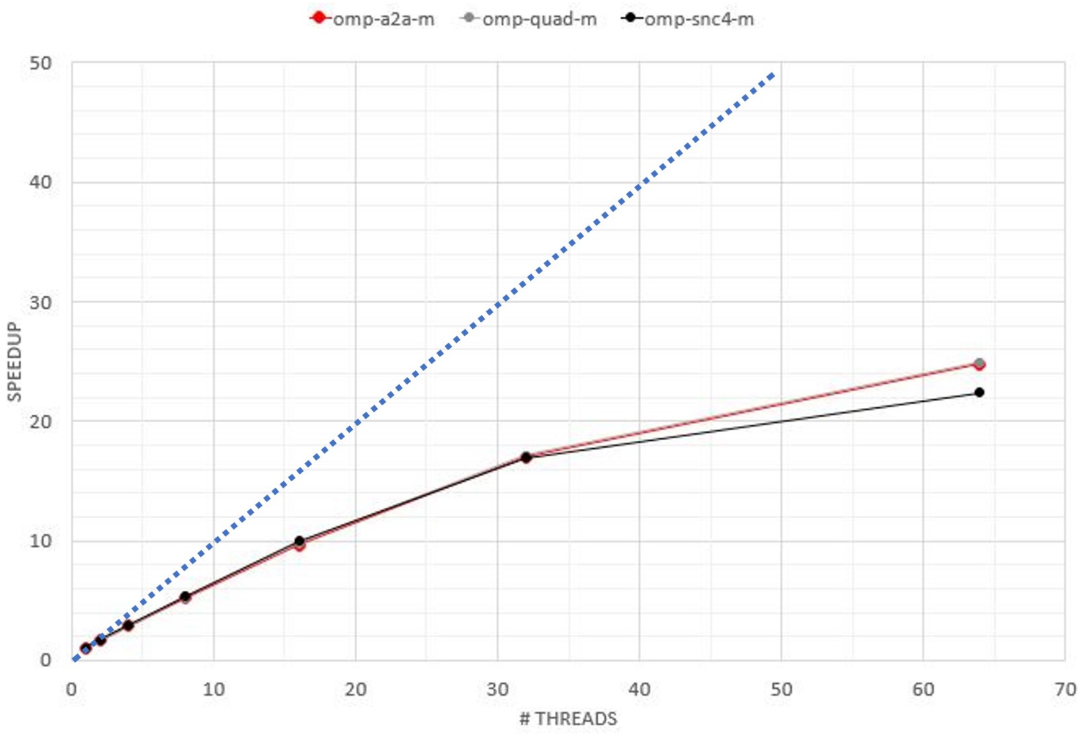
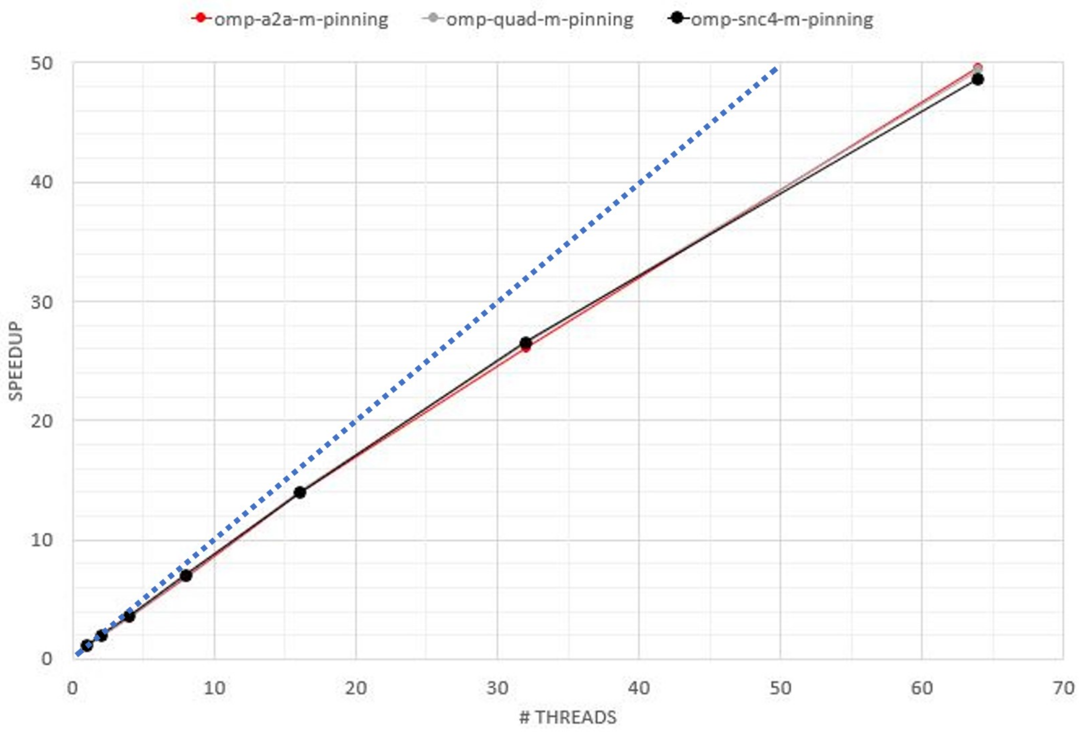


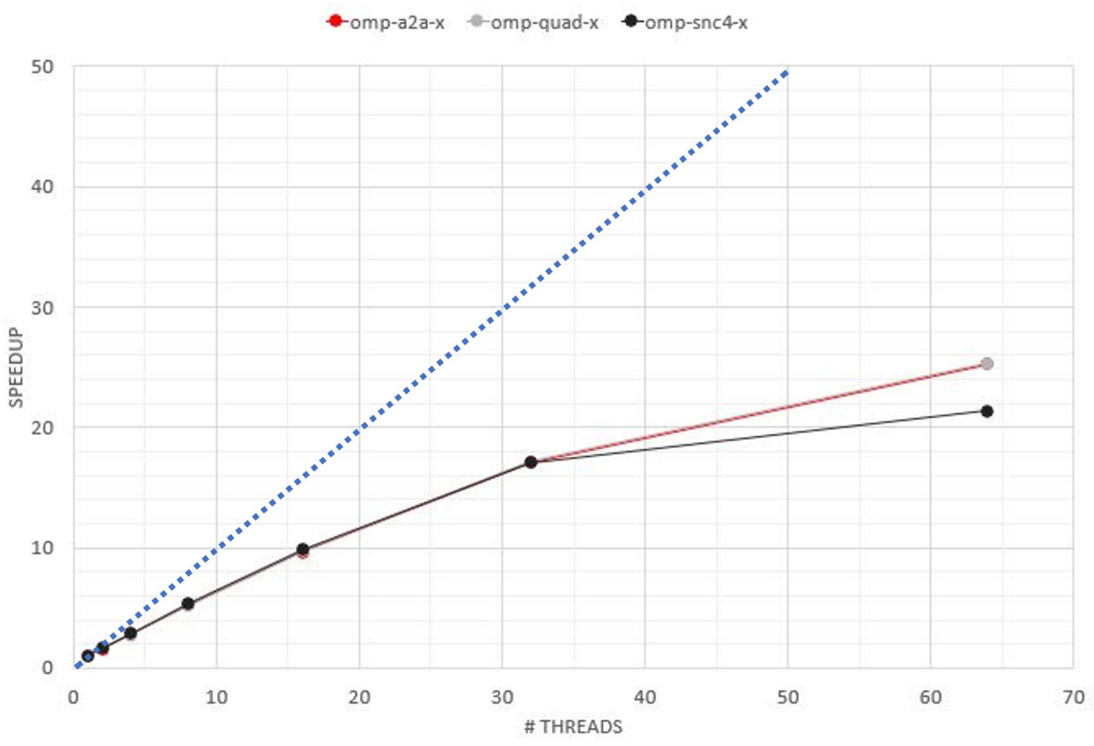
Figure 17: Speedup achieved by all setups.


omp-a2a-m
omp-quad-m
omp-snc4-m
SPEEDUP
# THREADS
0
10
20
30
40
50
60
70

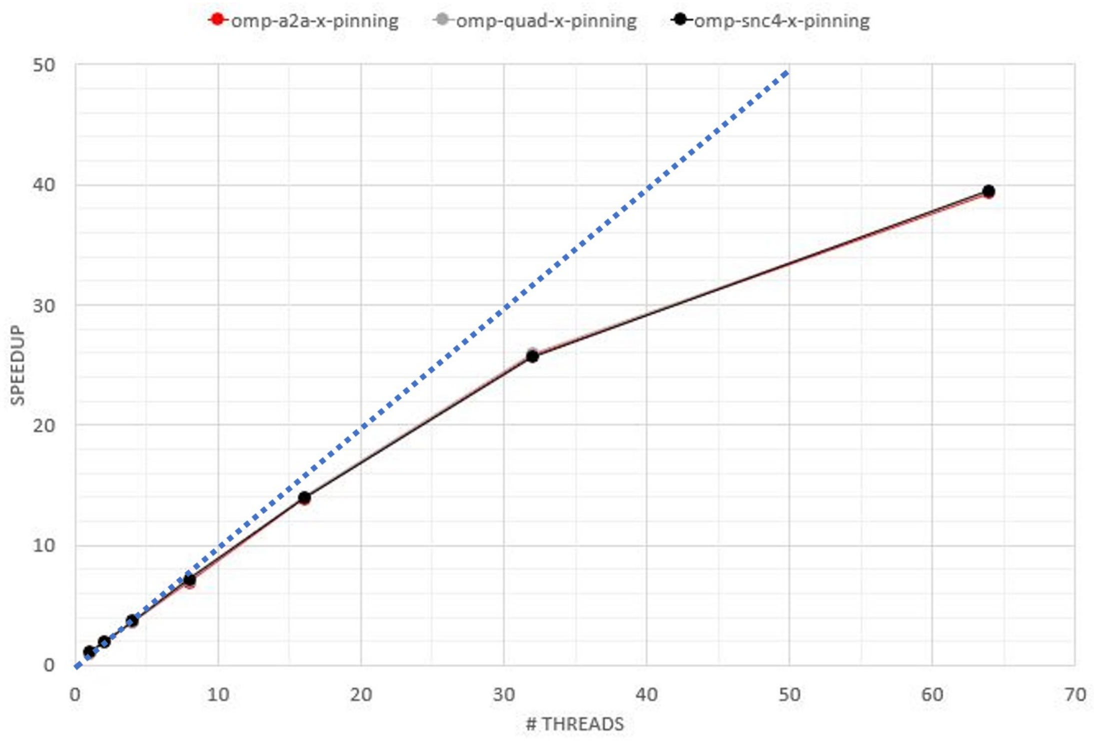
(a)


omp-a2a-m-pinning
omp-quad-m-pinning
omp-snc4-m-pinning
SPEEDUP
# THREADS
0
10
20
30
40
50
60
70

(b)

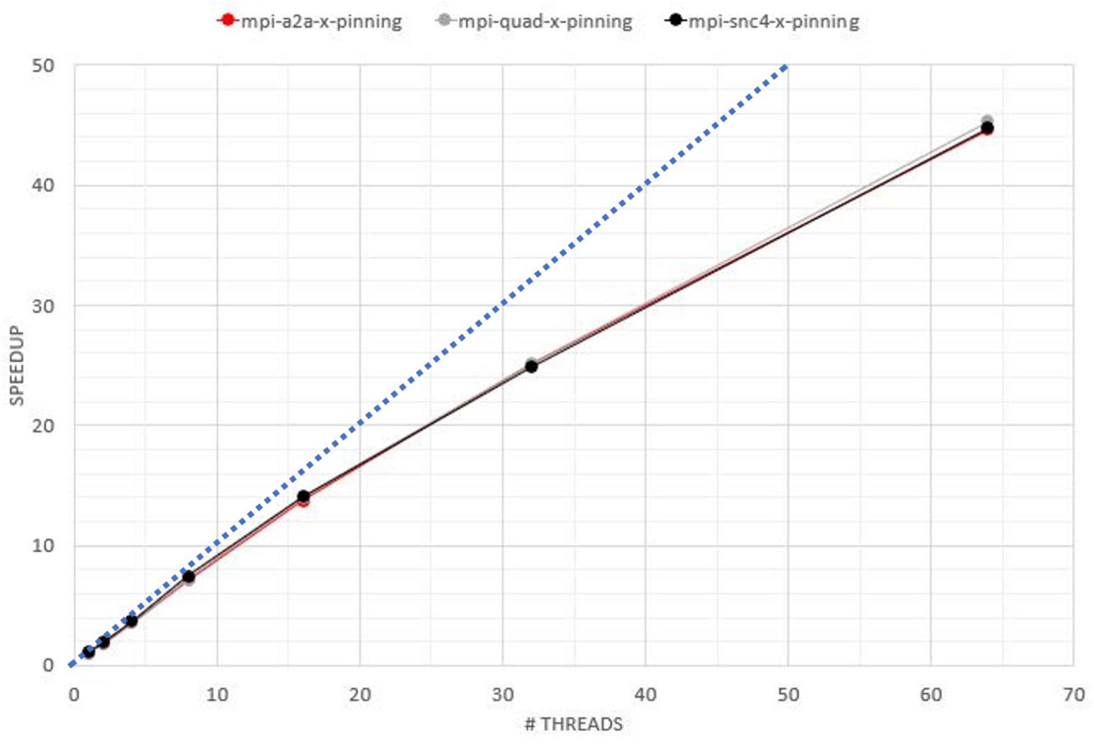


(c)

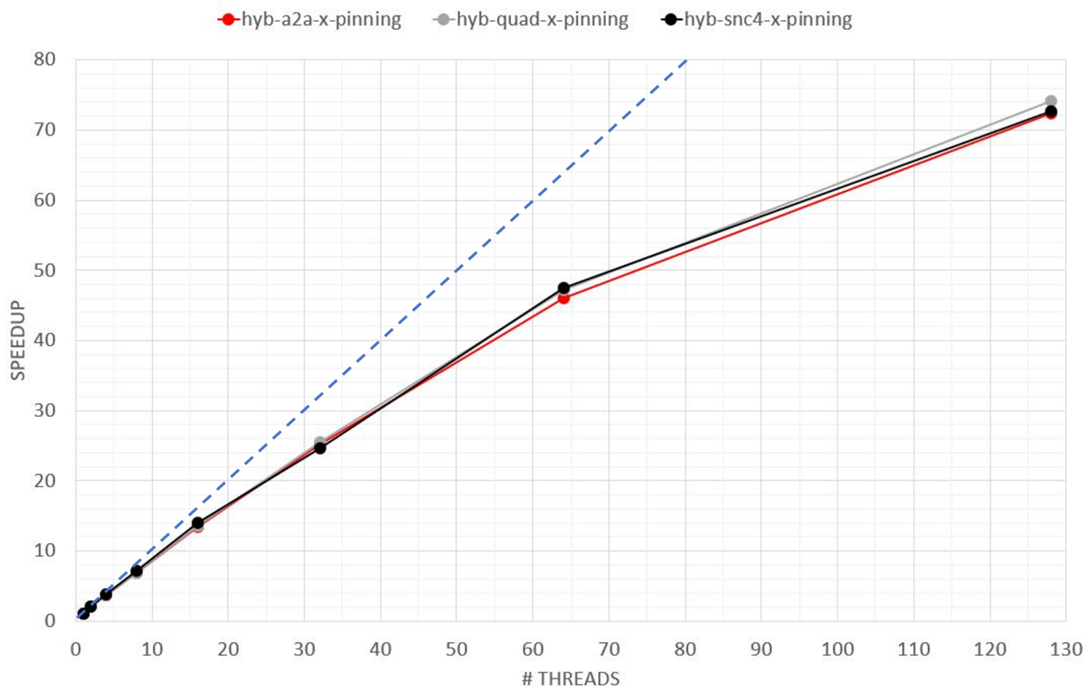


(d)

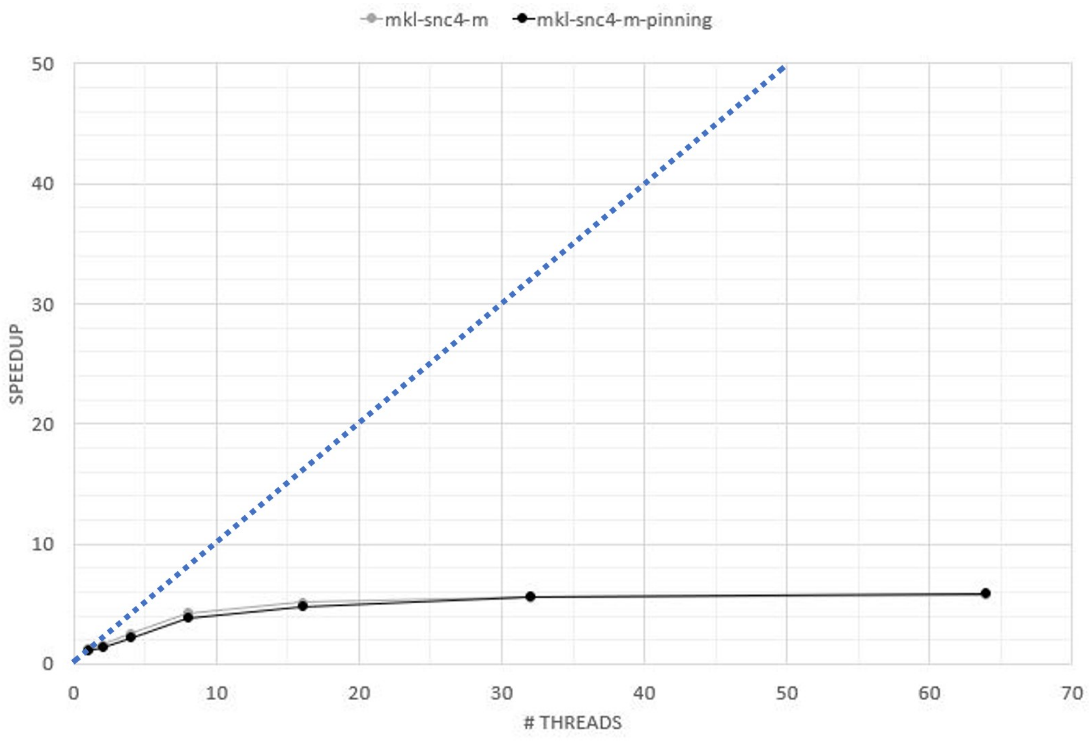
Figure 18: Speedup achieved by OpenMP setups grouped in (a) Implicit (b) Implicit + Pinning (c) Explicit (d) Explicit + Pinning.


mpi-a2a-x-pinning
mpi-quad-x-pinning
mpi-snc4-x-pinning
SPEEDUP
# THREADS
0
10
20
30
40
50
60
70

(a)


hyb-a2a-x-pinning
hyb-quad-x-pinning
hyb-snc4-x-pinning
SPEEDUP
# THREADS
0
10
20
30
40
50
60
70
80
90
100
110
120
130

(b)



(c)

Figure 19: Speedup achieved by (a) MPI-only (b) Hybrid (c) MKL. Blue dotted line shows the theoretical optimal speedup.

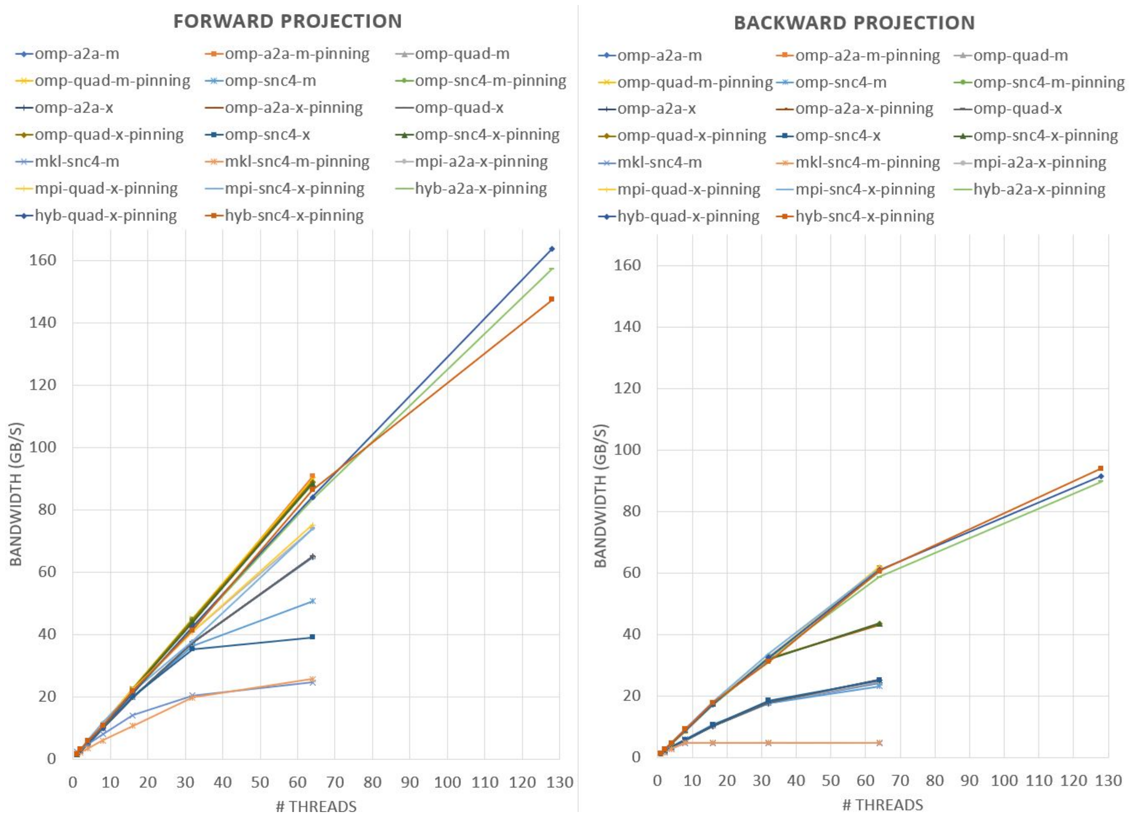
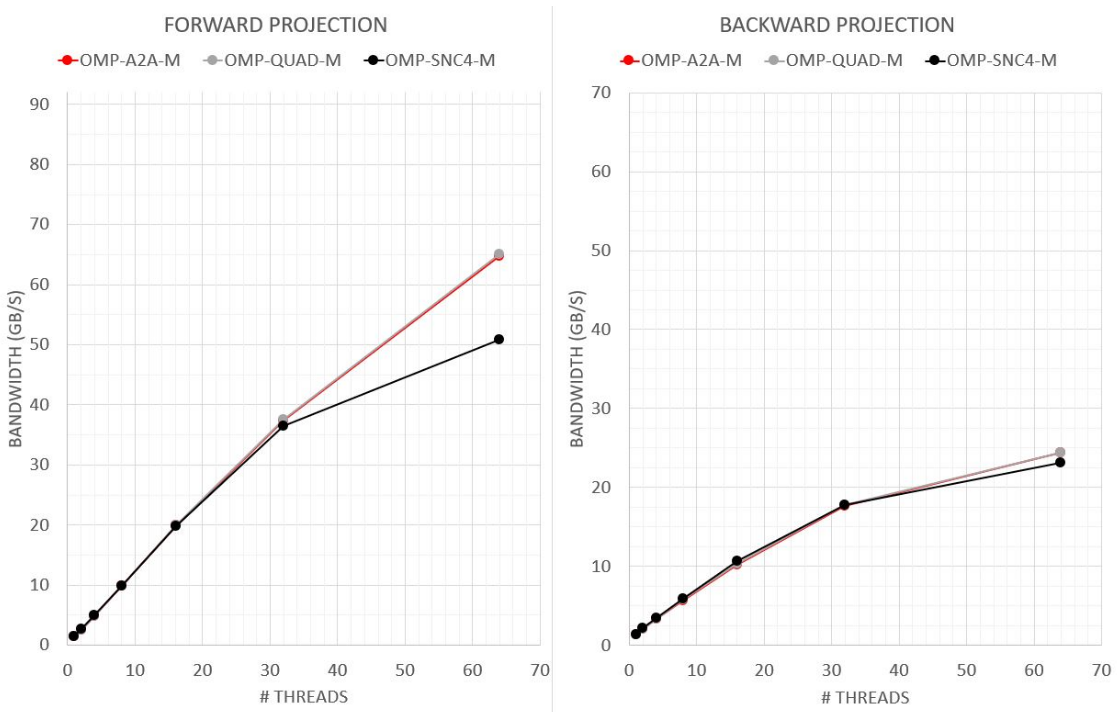
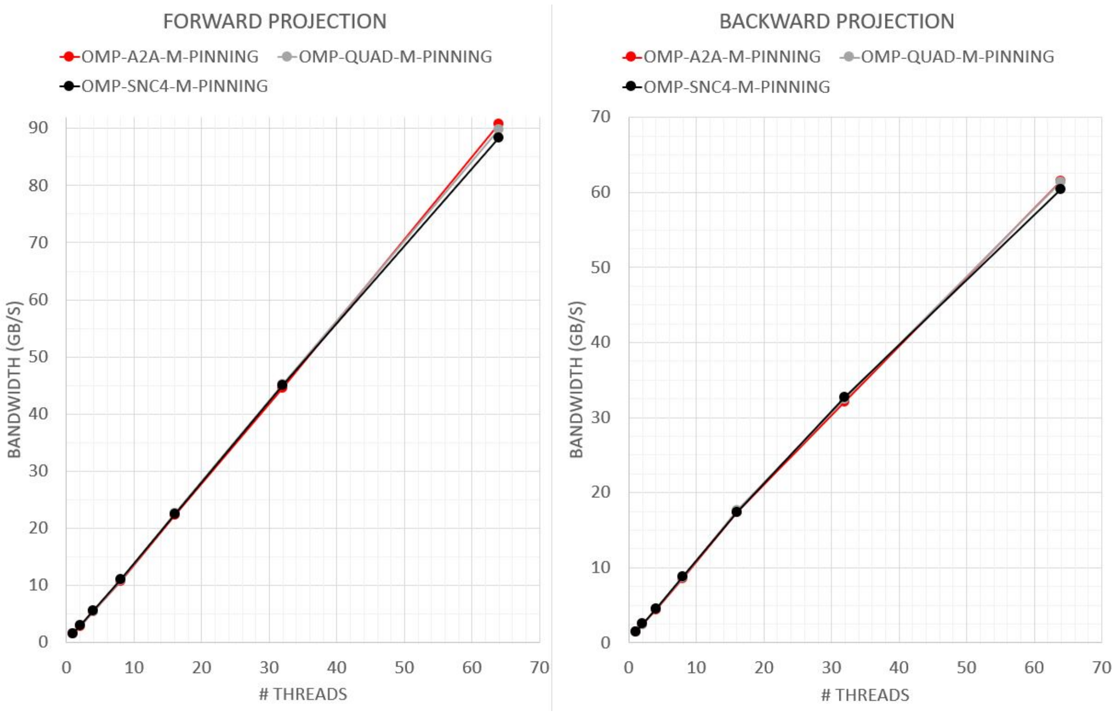


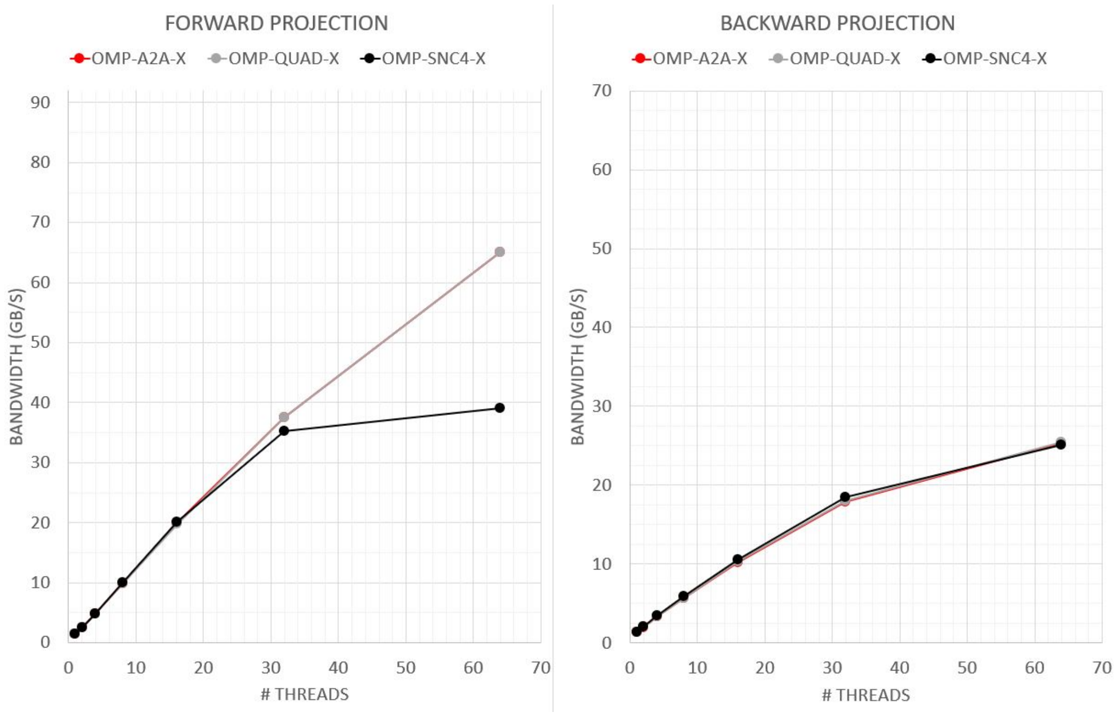
Figure 20: Bandwidth achieved by all setups.


FORWARD PROJECTION
OMP-A2A-M
OMP-QUAD-M
OMP-SNC4-M
BANDWIDTH (GB/S)
# THREADS
BACKWARD PROJECTION
OMP-A2A-M
OMP-QUAD-M
OMP-SNC4-M
BANDWIDTH (GB/S)
# THREADS

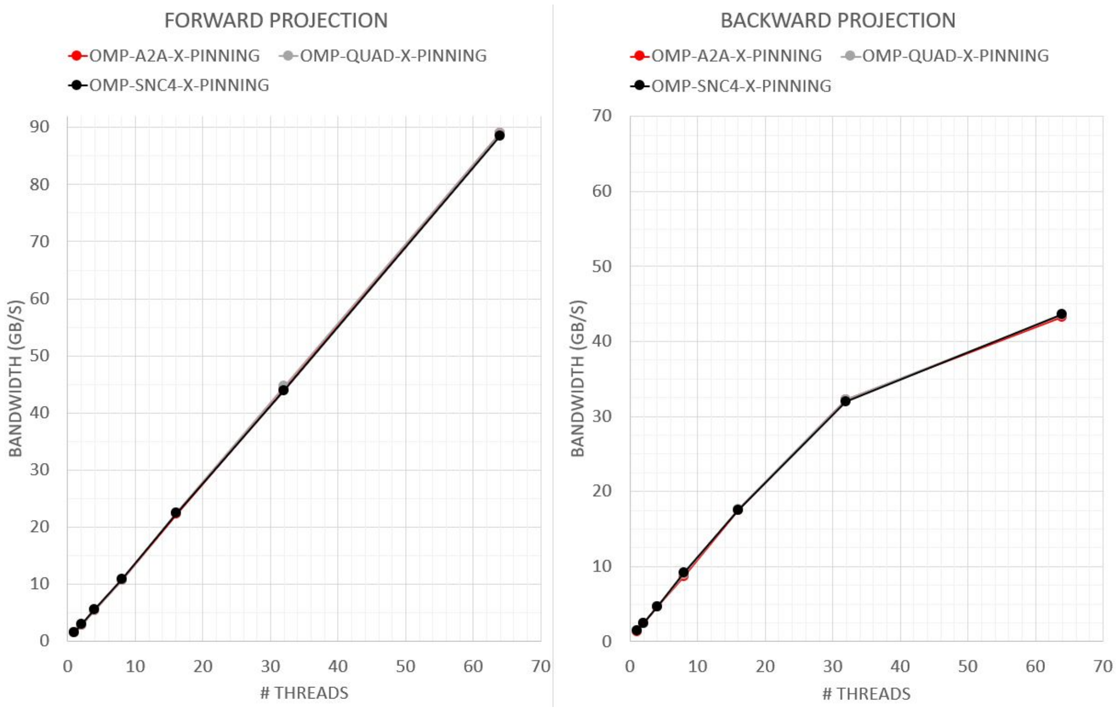
(a)


FORWARD PROJECTION
OMP-A2A-M-PINNING
OMP-QUAD-M-PINNING
OMP-SNC4-M-PINNING
BANDWIDTH (GB/S)
# THREADS
BACKWARD PROJECTION
OMP-A2A-M-PINNING
OMP-QUAD-M-PINNING
OMP-SNC4-M-PINNING
BANDWIDTH (GB/S)
# THREADS

(b)

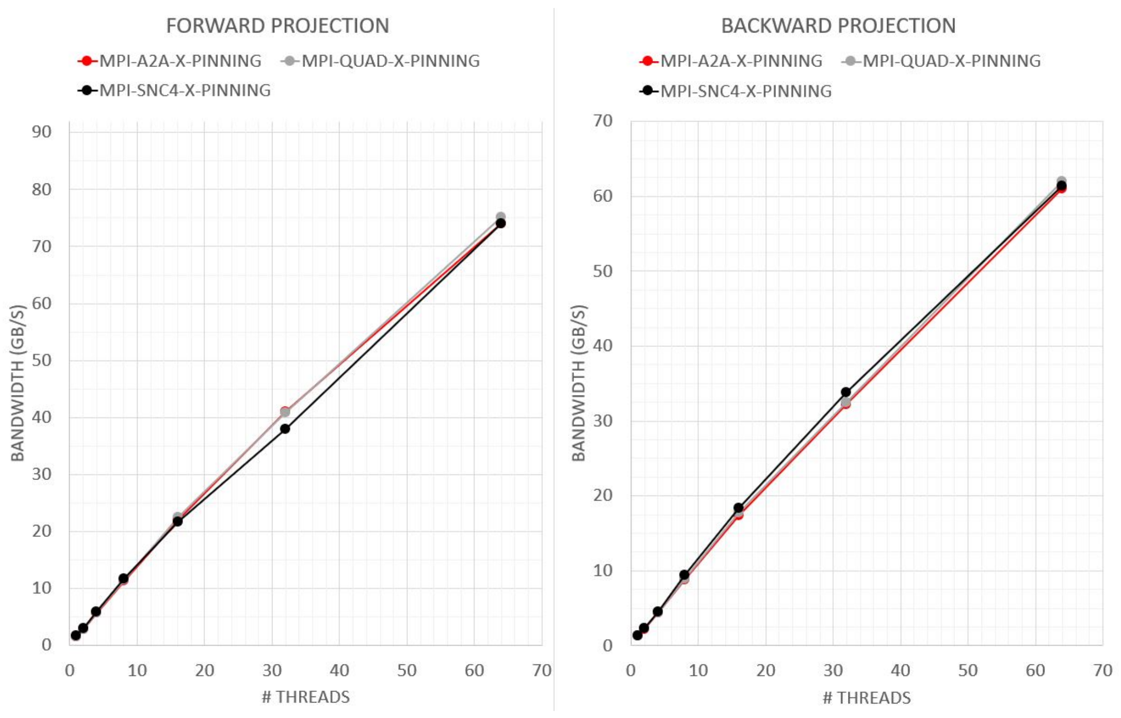


(c)



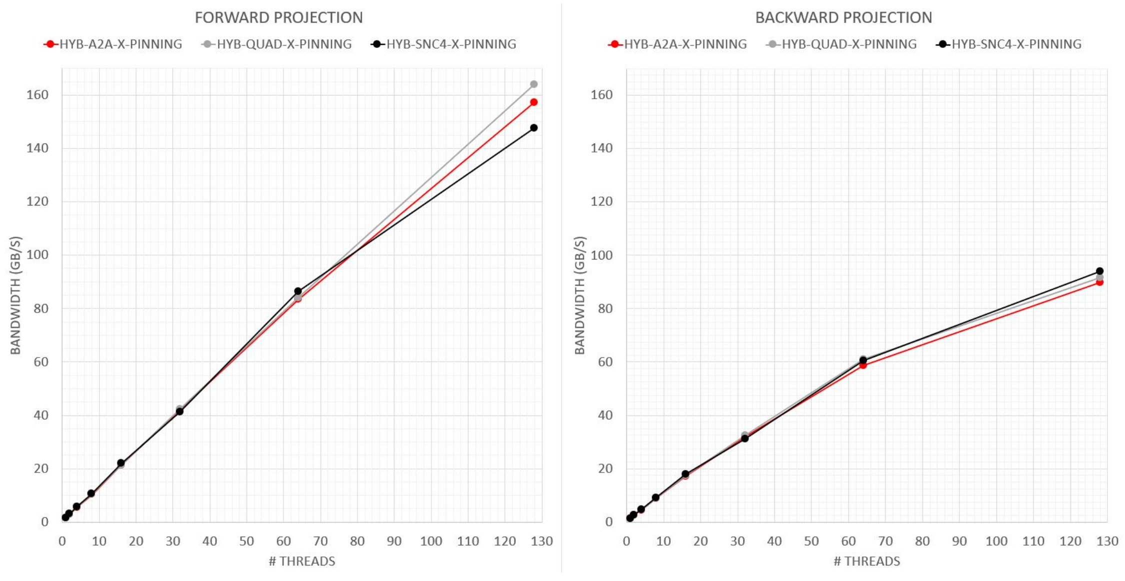
(d)

Figure 21: Memory bandwidth achieved by OpenMP setups grouped in (a) Implicit (b) Implicit + Pinning (c) Explicit (d) Explicit + Pinning.



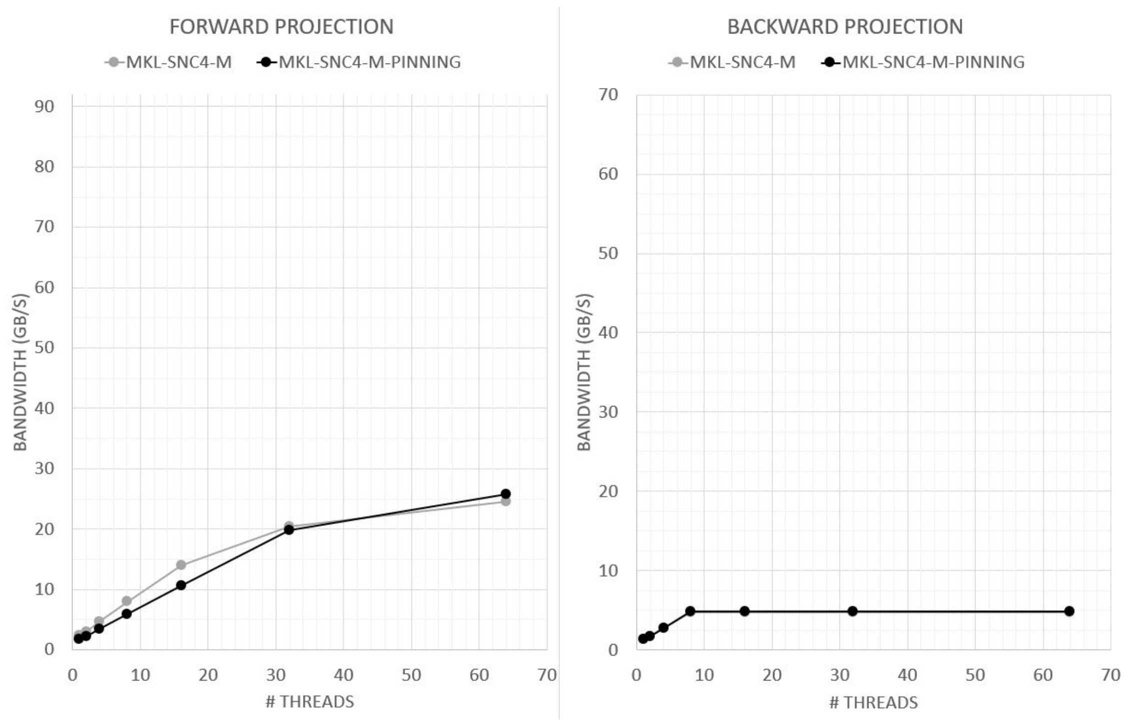
FORWARD PROJECTION
MPI-A2A-X-PINNING
MPI-QUAD-X-PINNING
MPI-SNC4-X-PINNING
BANDWIDTH (GB/S)
# THREADS
BACKWARD PROJECTION
MPI-A2A-X-PINNING
MPI-QUAD-X-PINNING
MPI-SNC4-X-PINNING
BANDWIDTH (GB/S)
# THREADS

(a)



FORWARD PROJECTION
HYB-A2A-X-PINNING
HYB-QUAD-X-PINNING
HYB-SNC4-X-PINNING
BANDWIDTH (GB/S)
# THREADS
BACKWARD PROJECTION
HYB-A2A-X-PINNING
HYB-QUAD-X-PINNING
HYB-SNC4-X-PINNING
BANDWIDTH (GB/S)
# THREADS

(b)



(c)

Figure 22: Memory bandwidth achieved by (a) MPI-only (b) Hybrid (c) MKL.

5.5.1 Affinity (Process/Thread Pinning)

There are 14 setups in total that can be compared to examine the effect of affinity, as detailed in Table 7. These setups are tackled in the next paragraphs based on their parallelism library breaking them into 12 and 2 setups corresponding to OpenMP and MKL respectively.

OpenMP. Figure 15 (a-d), where average runtime data are compared from (a) and (c) to (b) and (d) respectively as non-pinning vs pinning setups, shows a significant performance increase for pinning setups spanning all the different cluster modes. Figure 18 (a-d), where average runtime data are compared from (a) and (c) to (b) and (d) respectively as non-pinning vs pinning setups, shows a significant scalability increase for pinning setups spanning all the different cluster modes. Figure 21 (a-d), where memory bandwidth data are compared from (a) and (c) to (b) and (d) respectively as non-pinning vs pinning setups, shows a significant memory bandwidth increase in forward and backward projection in pinning setups spanning all the different cluster modes.

MKL. Figure 16 (c), non-pinning vs pinning average runtime, shows a significant performance increase non-pinning setups. Figure 19 (c), non-pinning vs pinning speedup, shows no observable difference in scalability between both setups. Figure 22 (c), non-pinning vs pinning memory bandwidth, shows an alternating behavior for memory bandwidth for forward projection between pinning and non-pinning; while no observable difference for backward projection. Despite the alternating behavior for forward projection, at 64 threads both, pinning and non-pinning, have similar memory bandwidth.

OpenMP pinning setups perform and scale better than non-pinning setups as expected. Pinning setups allow threads to allocate their required data at the nearest memory available to location where they are pinned. As a result, these setups can utilize the memory bandwidth more effectively, which is evident in Figure 21 (a-d). In the case of MKL, since MKL uses its own matrix format which does not allow the threads to allocate their own required data, pinning could have either positive or negative affect which is seen in its alternating bandwidth utilization in Figure 22 (c).

To conclude, this highlights the importance for developers to exploit data locality for its possible huge effect on performance as it can be seen here.

5.5.2 Cluster Mode

There are 18 setups in total that can be compared to examine the effect of cluster modes, as detailed in Table 7. These setups are tackled in the next paragraphs based on their parallelism library breaking them into 12, 3, and 3 setups corresponding to OpenMP, MPI-only, and Hybrid respectively.

OpenMP. Figure 15 (a-d), where each figure shows the average runtime of the three different cluster modes, reveals no significant difference in performance overall, especially for setups with pinning. Furthermore, snc-4 performance is slightly less than a2a and quad for setups with non-pinning and 64 threads. Figure 18 (a-d), where each figure shows speedup of the three different cluster modes, reveals no observable difference in scalability between the cluster modes in all setups with pinning; while setups without pinning show snc-4 having lower speedup for 64 threads. Figure 21 (a-d), where each figure shows memory bandwidth of the three different cluster modes, shows overall no observable difference in the performance between the three cluster modes in pinning setups. While the non-pinning setups, snc-4 utilizes less memory bandwidth in comparison to quad and a2a. It is noticeable for 64 threads.

MPI-only and Hybrid. Figure 16 (a-b), where each figure shows the average runtime of the three different cluster modes, reveals no significant difference in performance overall. Figure 19 (a-b), where each figure shows speedup of the three different cluster modes, reveals no observable difference in scalability between the the cluster modes. Figure 22 (a-d), where each figure shows memory bandwidth of the three different cluster modes, shows that MPI-only setups have no observable difference between the utilized memory bandwidth; while Hybrid setups have slight difference, namely quad utilizes the most in forward projection and snc-4 utilizes the most in backward projection. Having said that, this slight difference does not affect much the runtime and speedup showing that it is rather insignificant. Overall the performance, scalability, and memory bandwidth of MPI-only and Hybrid is similarly to OpenMP with pinning.

As mentioned previously in Section 3.4, the difference between the cluster modes is what affinity it introduces and how. snc-4 introduces affinity between tile, TD, and memory; quad introduces affinity between TD and memory; while, a2a is the most general introducing no affinity between tile, TD, and memory. That explains why snc-4 performs slightly better than quad and a2a as shown in Figure 15 (a-d) and 16 (a-b). However, OpenMP non-pinning at 64 threads setups, it seems that it performs slightly slower than a2a and quad, which is due conflicting affect of non-pinning (threads moving around in the chipset while the data is allocated still at the same place) and cluster modes. In other words, the affinity introduced by the cluster modes is damaged by threads moving around (non-pinning). The small difference in the performance affects scalability and reflects in memory bandwidth utilization which is more noticeable in Figure 18 (a-d) and 21 respectively. While in the case of OpeMP pinning, MPI-only, and Hybrid, cluster modes have more uniform performance, scalability, and memory bandwidth utilization with insignificant difference. That is because of the pinning implementation which allows each thread to allocate its own chunk of memory; thus, it introduces data affinity regardless of the cluster mode.

To conclude, a2a and quad have overall similar performance; and even though a2a and quad have the overall better performance and scalability in comparison to snc-4, cluster modes do not affect performance significantly especially when data affinity is introduced through pinning processes near to the data they require.

5.5.3 HBM Usage Model

There are 12 setups in total that can be compared to examine the effect of HBM Usage model, as detailed in Table 7. Figure 15 (a-d), where average runtime data are compared from (a) and (b) to (c) and (d) respectively as Implicit vs Explicit setups, shows no significant difference for setups with non-pinning; while observable difference in setups with pinning where Implicit performs better than Explicit. Figure 18 (a-d), where speedup data are compared from (a) and (b) to (c) and (d) respectively as Implicit vs Explicit setups, shows no significant difference for setups with non-pinning; while observable difference in scalability in setups with pinning where Implicit scales better than Explicit. Figure 21 , where memory bandwidth data are compared from (a) and (b) to (c) and (d) respectively as Implicit vs Explicit setups, shows no significant difference for setups with non-pinning; while an observable significant difference in setups with pinning where Implicit utilizes better memory bandwidth in backward projection than Explicit.

Implicit better performance, scaling, and memory bandwidth utilization than Explicit is expected. Implicit offloads the matrix as well as all the vectors and variables to the HBM; unlike Explicit which only offloads the matrix to the HBM. Even though the size vectors and variables is not as large as the matrix size, however their effect us noticeable as it is reflected on the memory bandwidth utilization shown in Figure 21.

To conclude, using the HBM has a great effect on performance. It is evident when comparing the performance of Implicit and Explicit, where Implicit takes full advantage of the HBM unlike the Explicit.

5.5.4 Parallelism Library

Figure 14 shows the average runtime of all the setups grouped by parallelism library. Their overall performance ranking in a descending order as follows: OpenMP, Hybrid, MPI-only and MKL. OpenMP pinning and Implicit, Hybrid, and MPI-only setups having similar performance verifies the implementations. Even though MPI is based on Message Passing paradigm, however the MPI-only setups uses MPI's Shared Memory Programming Model (SHM). As a result, it is expected to have a similar performance to OpenMP. On the other hand, MKL performance is rather under our expectations. Having said that, it is in agreement with other published work which is presented later in Chapter 6.

Figure 17 shows the speedup of all the setups grouped by parallelism library. Their overall scalability ranking in a descending order is the same as their performance order. That is because scalability is defined by average runtime, as shown in Equation 9. The linear increasing curve of OpenMP pinning and Implicit, Hybrid, and MPI-only setups is good when compared to the theoretical optimal speedup curve. While the other setups seems to suffer from data shuffling since their required data is not placed nearest (non-pinning setups).

Figure 20 shows the memory bandwidth of all the setups grouped by parallelism. Their overall memory bandwidth utilization ranking in a descending order is the same as speedup and average runtime. Average runtime and speedup is reflected in the memory bandwidth utilized by the setups. Hence, all the top performing setups, OpenMP pinning and Implicit, Hybrid, and MPI-only show high memory bandwidth utilization. The highest memory bandwidth utilized on one KNL node is 90 GB per second. Since this is a bit less than one third of the available memory bandwidth, it indicates the possibility to improve the implementations even more to utilize the HBM more effectively.

5.6 Limitation

5.6.1 CoolMUC3

There are two main limitations:

- **Cluster Modes**. There are two other cluster modes, hemi and snc-2, which are not mentioned in the thesis since they are not supported.
- **Runtime**. Rarely, bizarre iteration times are recorded, which presumed are caused by a malfunction. Those defected iterations are removed by replacing the full run where they occur with a new full run.

5.6.2 MKL

Collecting results of MKL is difficult due to killed jobs. This is happens specially for a2a and quad randomly for jobs of low thread number and to almost all jobs of high thread number. We could not find a reason that would explain MKL processes running successfully in snc-4 cluster mode but not in the others. Thus, we reached out to experts to investigate the issue, but we have not heard back yet.

6 Related Work

Speeding up iterative emission tomography image reconstruction algorithms, such as MLEM algorithm [7], [8], [50], [51], has been an important research topic with great practical importance [52]. This thesis evaluates KNL for running MLEM and more importantly approaches to speed up MLEM on KNL through utilizing various hardware configurations it offers as well as different parallelism libraries and concepts.

Before discussing the different parallelism libraries and concepts, an introduction to several mini applications and benchmarks used in the reviewed work is necessary:

- MiniFE [53]: It is a finite elements application which solves a nonlinear system of equations using Conjugate-Gradient algorithm. It is a memory-bound application, which makes it an optimal candidate to study the impact of the different cluster modes introduced previously in Section 3.4 [54].
- MiniMD [53]: It is a "Molecular dynamics code. It implements spatial decomposition, where each processor works on subsets of the simulation box. MiniMD computes atoms movements in a 3D space using the Lennard-Jones pair interaction. It follows a stencil communication pattern where neighbors exchange information about atoms in boundary regions. Because of these characteristics, it provides good weak scaling" [54].
- LBS3D [55]: It is a multiphase Lattice Boltzmann Code based on the Free Energy method of Zheng et. al. [55]. This code simulates the flow of two immiscible, isothermal, incompressible fluids with great spatial and temporal detail [54].
- XSBench [56]: It is a proxy app that models the most computationally intensive part of a typical Monte Carlo transport algorithm - the calculation of macroscopic neutron cross sections, which accounts for 85% of the total runtime of OpenMC (Monte Carlo particle transport simulation code focused on neutron criticality calculations). It is a memory-bound application and usually used for investigating on node parallelism issues [57].
- Graph500 [58]: "It represents data-analytics workloads. The memory access pattern is data-driven with poor temporal and spatial locality. Thus this application is featured with random access pattern" [59].
- GUPS [60]: It is a "synthetic benchmark that measures the Giga-updates-per-second (GUPS) by reading and updating uniformly distributed random addresses in a

table. The memory access pattern is random with poor data locality. This synthetic problem is often used for profiling the memory architecture" [59].

- DGEMM [60]: It is a benchmark that performs dense-matrix multiplication measuring Giga floating point operation as its performance evaluation. The memory access pattern is sequential and optimization in data locality is crucial [59].
- PARSEC [61]: Princeton Application Repository for Shared-Memory Computers (PARSEC) "is a benchmark suite composed of multithreaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors" [62].
- XGC1 [63]: It is a "full distribution function global 5D gyrokinetic Particle-In-Cell (PIC) code for simulations of turbulent plasma phenomena in a magnetic fusion device. It is particularly well-suited for plasma edge simulations due to an unstructured mesh used in the Poisson equation solver that allows the simulation volume to encompass the magnetic separatrix, the Scrape-Off-Layer (SOL) and the edge pedestal. The main building blocks of the code are the particle pusher, the collision operator and the Poisson solver" [64].

Cluster Modes. Rosales et. al. in [54] investigates the effect of cluster modes on performance of HPC applications. The paper uses mini application (MiniFE [53], MiniMD [53], and LBS3D [55]) to observe the performance differences in a2a and quad cluster modes running given 1 to 256 threads (in powers of 2). The results from MiniFE and MiniMD, using DRAM or MCDRAM, show that a2a scales better than quad; however it performs either better or same as quad. On the other hand, the results from LBS3D using MCDRAM show alternating behavior, where given certain number of threads a2a performs better, worse, or the same, while using DRAM a2a performs better or the same. Moreover, a2a scales slightly better than quad using MCDRAM and significantly better using DRAM. Malhanov et. al. in [65] assess the effect of cluster modes on the performance of couple of parallelism approaches. The paper tests the performance of PARSEC [61] in quad and snc4 cluster modes with parallelism approach as MPI only and Hybrid (MPI and OpenMP) as well as with MCDRAM as flat or cache. The results show that quad outperforms snc-4 in all setups. Carrier et. al. in [64] examines the effect of clusters modes, quad and snc4, on one node thread scaling of XGC1 [63]. The results display that quad performs slightly better when using MCDRAM or DDR. Ultimately, the effect of the cluster modes is dependent on the application as demonstrated in the work stated above.

Memory Mode. Smith et. al. [66] compares the effect of MCDRAM memory mode, flat and cache, on the performance of MTTKRP using several databases. The results reveal that both, cache and flat mode, perform identically when the database fit in MCDRAM; while when the database is larger, flat mode performs better than cache with significantly lower runtime. Peng et. al. in [59] inspects MCDRAM memory mode effect thoroughly over several applications and benchmarks. The paper benchmarks the performance of XSBench [56] over a range of problem sizes, Graph500 [58] over a range of graph sizes, GUPS [60] over a range of table sizes, MiniFE over a range of matrix sizes, and DGEMM [60] over a range of array sizes in flat and cache. The results of XSBench and DGEMM shows similar performance, Graph500 and GPUs shows alternating performance, and MiniFE shows flat outperforms cache. Rosales et. al. in [54] also observe the effect of MCDRAM memory modes on the performance of HPC application. In the same evaluation of MiniFE, MiniMD, and LBS3D performance in a2a cluster mode, the effect of flat and cache memory modes is noted. The results show insignificant difference in the performance of flat and cache. To conclude, the best memory mode configuration is dependent on the application memory usage and definitely on whether the dataset fits in the MCDRAM.

Parallelism Library. Malhanov et. al. in [65], as mentioned previously, assess the effect of cluster modes on the performance of couple of parallelism approaches, Hybrid and MPI-only, on KNL. The results show that Hybrid performance has less imbalance; however, MPI only scales better than Hybrid although Hybrid allows better memory usage. Zhao et. al. in [67] examine the adaptation of Hybrid approach parallelism in comparison to pure. The work compares Hybrid (MPI and OpenMP) and MPI only on VASP [68], widely used materials science code. The results show that Hybrid outperforms MPI-only by 2 to 3 times. Yan et. al. in [69] investigates the performance several implementations of parallel matrix multiplcation algorithms. The paper presents several OpenMP implementation of matrix multiplcaition based forward parallel algorithms, Loop Chunking and Recursive Tiling, and advance parallel algorithms, Hybrid Tiling and Strassen's Algorithm [70]. It also includes MKL routine, *cblas_dgemm*, to the comparison. The results show that MKL routine performance over a range of 1 to 48 threads (multiples of 2) worse than Hybrid Tiling and Strassen and alternating to Recursive Tiling. Furthermore, it has rather bad L2 cache hit ratio in comparison to the rest of the algorithms. Cramer et. al. in [71] evaluates the performance of OpenMP running simple benchmarks and kernels. The work observes the scalability and performance of the CG Kernel of OpenMP for loop of a sparse matrix-vector multiplication in comparison to equivalent MKL routine, over a range of threads and given the matrix is stored in CSR format. The results show better speedup accomplished by OpenMP overall; however, MKL reaches the same speedup as OpenMP when all the hardware threads are used. To sum up, the performance of the parallelism libraries varies depending on the application, whether communication constant synchronization is needed or not; meaning that some libraries are more suitable than others depending on the problem characteristics.

Affinity. Jabbie et. al. in [72] observes the performance of the classical elliptic test problem of the Poisson equation on KNL. The work tests two pinning techniques, scatter and balanced, on the performance of the test over a range of processes and thread using a hybrid approach. The results show no observable difference in runtime behavior. Cramer et. al. in [71], presented in the previous paragraph, also examine whether affinity effects memory bandwidth over a range of threads. The results show that balanced affinity memory bandwidth access curve grows faster than scatter; however it normalizes and drops to the same bandwidth as scatter when all hardware threads are used. To conclude, affinity matters since it is developer's interest to maintain the locality achieving the lowest latency and greatest bandwidth of communication with caches [73].

7 Conclusion

This thesis focused on evaluating High Performance Computing (HPC) System, Xeon Phi Knights Landing (KNL), for running a medical image reconstruction algorithm, Maximum Likelihood Expectation Maximization (MLEM), for Positron Emission Topography (PET). Several implementations were provided based on parallelism libraries, such as Message Passing. Furthermore, those implementations are assessed in all the different configurations based on KNL hardware features such as High Bandwidth Memory (HBM). To compare all these setups for best configuration and implementation, runtime, speed up and bandwidth are collected and computed from running them. These are the criteria used for determining the best hardware configuration as well as best parallelism approach (implementation) for running MLEM on KNL.

The results' findings are categorized based on the various hardware configurations as well as parallelism concepts used and they are as follows:

- **Parallelism Library**. On one node, Open-MP and MPI-only had similar performance, speedup, and memory bandwidth for equivalent setups; while MKL setups' performance, speedup, and memory bandwidth were inadequate in comparison to the rest, which might be related to the routine used in the implementation. MPI-only setups uses MPI's Shared Memory Model (SHM) based on Shared Memory paradigm since it is running on one node, even though MPI is basically based on Message Passing paradigm. This allows us to verify both implementations correctness but more importantly shows that libraries based on Shared Memory paradigm run MLEM the best on KNL. That is because Message Passing paradigm on one KNL node would have overhead communication for synchronization in MLEM implementation. That observation is based on Amdahl's law [74]; the limiting effect caused by inter-node communication becomes severe as the parallel computation gets faster [52].

- **Cluster Modes**. As explained previously, the main difference between the cluster modes is the affinity they introduce and how. Cluster modes has an observable effect on non-pinning setups that results in a2a and quad performing, scaling, and utilizing the memory bandwidth better than snc-4. On the other hand, pinning setups are not affected by cluster modes. This is because pinning introduces the affinity regardless of the cluster modes. This concludes that if a programmer wants to run an optimized or a non-optimized implementation, quad or a2a would yield the best configuration.

- **HBM Usage Model**. As explained previously, there are two usage models for using the HBM as flat, implicit and explicit. Implicit setups outperform equivalent explicit setups. This is expected since 1) the matrix fits in the HBM 2) in the implicit setups the matrix and all other variables and vectors are allocated in the HBM; while, in explicit setups the matrix only is allocated in the HBM. This reveals the HBM effect on performance, speedup, and memory bandwidth utilization. It also stresses the importance of using HBM wisely; for example, offloading large frequently used data in the application to the HBM.
- **Affinity**. Introducing data affinity by pinning threads or processes close to the data location is a known parallel programming technique. Affinity has a significant observable affect on performance, speedup, and memory bandwidth utilization. That is because in affinity setups each process/thread allocates its chunk of the data as close as possible to itself. Thus, utilizing memory bandwidth effectively and reducing wait time for processors. Furthermore, the `compact` pinning strategy allows the exploitation of L2 Cache data locality.

It is worth to point out that *Implicit* HBM usage model is a useful feature. HPC need and usage is spreading to various fields, such as astronomy and biology. More scientists from different disciplines are interested in using HPC systems. Thus, there is a necessity to equip HPC systems with features and functionality allowing those scientists use with ease such as minimal code adaptation. In conclusion, the results show that the best performing setup is OpenMP with implicit, pinning and either a2a or quad cluster mode.

7.1 Future Work

The next steps is to evaluate the effect of different matrix storage format as well as HPC systems and architectures on MLEM performance.

There are advantages and disadvantages of each matrix storage format as discussed in Section 4.1. There has been research on the performance of Compressed Sparse Blocks (CSB) in comparison to Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) in certain applications; yielding certain storage format outperforming the others. Thus, a more comprehensive study is essential.

The matrix size used in this work fits in HBM; while this is not necessary the case for larger PET scanners. Moreover, there has been studies examining various configurations and ways for optimizing the performance of KNL when the dataset does not fit in the HBM. Extending this work to include matrices of various sizes specially ones that don't fit in the HBM is fundamental.

Although there has been research on optimizing MLEM performance and evaluating it on different architectures, there has not been a published paper comparing several architectures and HPC system performance highlighting the differences. A survey study that evaluates several different architectures running MLEM at their best configuration is vital. The study could be used as reference for a range of algorithms that are memory bound algorithms and applications.

8 Bibliography

[1] S. Salehian and Y. Yan, "Evaluation of Knight Landing High Bandwidth Memory for HPC Workloads", in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ACM, 2017, p. 10.

[2] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[3] J. Ramírez, J. M. Górriz, M. Gómez-Río, A. Romero, R. Chaves, A. Lassl, A. Rodríguez, C. G. Puntonet, F. Theis, and E. Lang, "Effective Emission Tomography Image Reconstruction Algorithms for SPECT Data", in *Computational Science – ICCS 2008*, Springer Berlin Heidelberg, 2008, pp. 741–748.

[4] G. T. Herman, *Fundamentals of computerized tomography: image reconstruction from projections*. Springer Science & Business Media, 2009.

[5] P. Conti and L. Strauss, "The applications of PET in clinical oncology", *J Nucl Med*, vol. 32, pp. 623–648, 1991.

[6] T. Küstner, J. Weidendorfer, J. Schirmer, T. Klug, C. Trinitis, and S. Ziegler, "Parallel MLEM on multicore architectures", in *International Conference on Computational Science*, Springer, 2009, pp. 491–500.

[7] L. A. Shepp and Y. Vardi, "Maximum Likelihood Reconstruction for Emission Tomography", *IEEE Transactions on Medical Imaging*, vol. 1, pp. 113–122, 1982.

[8] H. M. Hudson and R. S. Larkin, "Accelerated image reconstruction using ordered subsets of projection data", *IEEE transactions on medical imaging*, vol. 13, pp. 601–609, 1994.

[9] C. Vazquez, M. Rodriguez-Alvarez, C. Correcher, A. González, F. Sánchez, P. Conde, and J. Benlloch, "Parallelization of MLEM algorithm for PET reconstruction based on GPUs", in *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2014 IEEE*, IEEE, 2014, pp. 1–4.

[10] D. McElroy, M. Hoose, W. Pimpl, V. Spanoudaki, T. Schüler, and S. Ziegler, "A true singles list-mode data acquisition system for a small animal PET scanner with independent crystal readout", *Physics in Medicine & Biology*, vol. 50, p. 3323, 2005.

[11] D. P. McElroy, W. Pimpl, M. Djelassi, B. J. Pichler, M. Rafecas, T. Schuler, and S. Ziegler, "First results from MADPET-II: a novel detector and readout system for high resolution small animal PET", in *Nuclear Science Symposium Conference Record, 2003 IEEE*, IEEE, vol. 3, 2003, pp. 2043–2047.

[12] M. Rafecas, B. Mosler, M. Dietz, M. Pogl, A. Stamatakis, D. P. McElroy, and S. I. Ziegler, “Use of a Monte Carlo-based probability matrix for 3-D iterative reconstruction of MADPET-II data”, *IEEE Transactions on Nuclear Science*, vol. 51, pp. 2597–2605, 2004.

[13] D. Geer, “Chip Makers Turn to Multicore Processors”, *Computer*, vol. 38, pp. 11–13, 2005.

[14] A. Roy, J. Xu, and M. H. Chowdhury, “Multi-Core Processors: A New Way Forward and Challenges”, in *Microelectronics, 2008. ICM 2008. International Conference on*, IEEE, 2008, pp. 454–457.

[15] A. Heinecke, M. Klemm, and H.-J. Bungartz, “From gpgpu to many-core: Nvidia fermi and intel many integrated core architecture”, *Computing in Science & Engineering*, vol. 14, pp. 78–83, 2012.

[16] P. J. Ungaro, “The changing role of supercomputing”, in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ACM, 2012, pp. 1–2.

[17] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, “Knights landing: Second-generation intel xeon phi product”, *Ieee micro*, vol. 36, pp. 34–46, 2016.

[18] Y. Hirokawa, T. Boku, S. A. Sato, and K. Yabana, “Performance evaluation of large scale electron dynamics simulation under many-core cluster based on knights landing”, in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ACM, 2018, pp. 183–191.

[19] S. Gill, “Parallel programming”, *The Computer Journal*, vol. 1, pp. 2–10, 1958.

[20] G. V. Wilson, “The history of the development of parallel computing”, *URL: http://ei.cs.vt.edu/history/Parallel.html*, 1994.

[21] G. Anthes, “The power of parallelism”, *Computerworld. Retrieved on*, pp. 01–08, 2008.

[22] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, *et al.*, “A view of the parallel computing landscape”, *Communications of the ACM*, vol. 52, pp. 56–67, 2009.

[23] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Redwood City, 1994, vol. 400.

[24] M. Sato, “OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors”, in *Proceedings of the 15th international symposium on System Synthesis*, ACM, 2002, pp. 109–111.

[25] R. Chandra, L. Dagum, D. Kohr, D. Maydan, R. Menon, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[26] L. M. Silva and R. Buyya, “Parallel programming models and paradigms”, *High Performance Cluster Computing: Architectures and Systems*, vol. 2, pp. 4–27, 1999.

[27] M. Snir, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI–the Complete Reference: The MPI core*. MIT press, 1998, vol. 1.

[28] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard", *Parallel computing*, vol. 22, pp. 789–828, 1996.

[29] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the MPI standard", *Communications of the ACM*, p. 18, 1995.

[30] K. Kedia, "Hybrid programming with OpenMP and MPI", Technical Report 18.337 J, Massachusetts Institute of Technology, Tech. Rep., 2009.

[31] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[32] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[33] A. E. Eichenberger, C. Terboven, M. Wong, and D. an Mey, "The design of OpenMP thread affinity", in *International Workshop on OpenMP*, Springer, 2012, pp. 15–28.

[34] S. Pophale and O. Hernandez, "Evaluating OpenMP Affinity on the POWER8 Architecture", in *International Workshop on OpenMP*, Springer, 2016, pp. 35–46.

[35] *Intel® Xeon Phi™ Processors*, Accessed: 2018.

[36] *Intel® Xeon Phi™ Product Family*, Accessed: 2018.

[37] L. C. Garcia, D. C. Tjon-Pian-Gi, S. G. Tucker, and M. W. Zajac, *Vector processing unit*, US Patent 4,791,555, 1988.

[38] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[39] Z. Zhao and M. Marsman, "Estimating the performance impact of the MCDRAM on KNL using dual-socket Ivy Bridge nodes on Cray XC30", *Proceedings of the Cray User Group–2016*, 2016.

[40] P. Gillingham and B. Millar, *High Bandwidth Memory interface*, US Patent 6,510,503, 2003.

[41] J. Turley, *White Paper Introduction to Intel® Architecture: The basics*, Accessed: 2018.

[42] A. J. Smith, "Cache memories", *ACM Computing Surveys (CSUR)*, vol. 14, pp. 473–530, 1982.

[43] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory", *IEEE Transactions on computers*, vol. 41, pp. 1054–1068, 1992.

[44] *jemalloc*, Accessed: 2018.

[45] Intel, *memkind*, Accessed: 2018.

[46] A. Szlávecz, G. Hesz, T. Bükki, B. Kári, and B. Benyó, "GPU-based acceleration of the MLEM algorithm for SPECT parallel imaging with attenuation correction and compensation for detector response", in *Proceedings of the 18th IFAC World Congress. Milan, Italy*, 2011, pp. 6195–6200.

[47] A. Słomski, Z. Rudy, T. Bednarski, P. Białas, E. Czerwiński, Ł. Kapłon, A. Kochanowski, G. Korcyl, J. Kowal, P. Kowalski, *et al.*, "3D PET image reconstruction based on the maximum likelihood estimation method (MLEM) algorithm", *Bio-Algorithms and Med-Systems*, vol. 10, pp. 1–7, 2014.

[48] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computations", 1990.

[49] A. Gupta, "Implementation and Evaluation of MLEM-Algorithm on GPU using CUDA", Masterarbeit, Technische Universität München, 2018.

[50] A. J. Reader, K. Erlandsson, M. A. Flower, and R. J. Ott, "Fast accurate iterative reconstruction for low-statistics positron volume imaging", *Physics in Medicine & Biology*, vol. 43, p. 835, 1998.

[51] R. M. Lewitt and S. Matej, "Overview of methods for image reconstruction from projections in emission computed tomography", *Proceedings of the IEEE*, vol. 91, pp. 1588–1611, 2003.

[52] J. Cui, G. Pratx, B. Meng, and C. S. Levin, "Distributed MLEM: An iterative tomographic image reconstruction algorithm for distributed memory architectures", *IEEE transactions on medical imaging*, vol. 32, pp. 957–967, 2013.

[53] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications", *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[54] C. Rosales, J. Cazes, K. Milfeld, A. Gómez-Iglesias, L. Koesterke, L. Huang, and J. Vienne, "A comparative study of application performance and scalability on the Intel Knights Landing processor", in *International Conference on High Performance Computing*, Springer, 2016, pp. 307–318.

[55] H. Zheng, C. Shu, and Y.-T. Chew, "A lattice Boltzmann model for multiphase flows with large density ratio", *Journal of Computational Physics*, vol. 218, pp. 353–371, 2006.

[56] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis", *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[57] J. Tramm, *The XSBench Mini-App - A Discussion of Theory*, Accessed: 2018.

[58] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500", *Cray User's Group (CUG)*, vol. 19, pp. 45–74, 2010.

[59] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "Exploring the Performance Benefit of Hybrid Memory System on HPC Environments", in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, IEEE, 2017, pp. 683–692.

[60] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, "Introduction to the HPC challenge benchmark suite", 2005.

[61] C. Bienia, "Benchmarking Modern Multiprocessors", PhD thesis, Princeton University, 2011.

[62] Princeton University, *The PARSEC Benchmark Suite*, Accessed: 2018.

[63] S. Ku, C.-S. Chang, and P. Diamond, "Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry", *Nuclear Fusion*, vol. 49, p. 115 021, 2009.

[64] T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. He, T. Kurth, T. Koskela, M. Lobet, T. Malas, *et al.*, "Evaluating and optimizing the NERSC workload on knights landing", in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on*, IEEE, 2016, pp. 43–53.

[65] A. Malhanov, A. J. Biller, and M. Chuvelev, "Optimizing PARSEC for Knights Landing", in *Proceedings of the 23rd European MPI Users' Group Meeting*, ACM, 2016, pp. 213–214.

[66] S. Smith, J. Park, and G. Karypis, "Sparse tensor factorization on many-core processors with high-bandwidth memory", in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, IEEE, 2017, pp. 1058–1067.

[67] Z. Zhao, M. Marsman, F. Wende, and J. Kim, "Performance of hybrid MPI/OpenMP VASP on Cray XC40 based on Intel Knights landing many integrated core architecture", *CUG Proceedings*, 2017.

[68] G. Kresse and J. Furthmüller, "Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set", *Computational materials science*, vol. 6, pp. 15–50, 1996.

[69] Y. Yan, J. Kemp, X. Tian, A. M. Malik, and B. Chapman, "Performance and power characteristics of matrix multiplication algorithms on multicore and shared memory machines", in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, IEEE, 2012, pp. 626–632.

[70] V. Strassen, "Gaussian elimination is not optimal", *Numerische mathematik*, vol. 13, pp. 354–356, 1969.

[71] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "Openmp programming on intel r xeon phi tm coprocessors: An early performance comparison", in *Proc. Many Core Appl. Res. Community (MARC) Symp*, 2012, pp. 38–44.

[72] I. A. Jabbie, G. Owen, and B. Whiteley, "Performance comparison of Intel Xeon Phi Knights Landing", *SIAM Undergraduate Research Online (SIURO)*, vol. 10, 2017.

[73] V. Mironov, Y. Alexeev, K. Keipert, M. D'mello, A. Moskovsky, and M. S. Gordon, "An efficient MPI/openMP parallelization of the Hartree-Fock method for the second generation of Intel® Xeon Phi™ processor", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2017, p. 39.

[74] D. P. Rodgers, "Improvements in multiprocessor system design", in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society Press, vol. 13, 1985, pp. 225–231.

9 Appendix

Evaluation

Setup

Parallelism Library	Compiler	Compiler Flags
MKL	icpc	$-DMKL\_ILP64 - I\$\{MKLROOT\}/include$ $-O3 - std = c + +11 - xMIC - AVX512$ $-fma - align - finline - functions$
OpenMP	icpc	$-O3 - std = c + +11$ $-I\$\{BOOST\_INCLUDE\}$ $-D\_MWAITXINTRIN\_H\_INCLUDED$ $-D\_FORCE\_INLINES$ $-D\_\_STRICT\_ANSI\_\_$ $-fopenmp$
MPI-only & Hybrid	mpiicpc	$-O3 - std = c + +11$ $-I\$\{BOOST\_INCLUDE\}$ $-D\_MWAITXINTRIN\_H\_INCLUDED$ $-D\_FORCE\_INLINES$ $-D\_\_STRICT\_ANSI\_\_$ $-fopenmp$

Table 8: Compiler flags based on Parallelism Libraries

Hardware	
Number of nodes	148
Cores per node	64
Core nominal frequency	1.3 GHz
Memory (DDR4) per node	96 GB (Bandwidth 80.8 GB/s)
High Bandwidth Memory per node	16 GB (Bandwidth 460 GB/s)
Bandwidth to interconnect per node	25 GB/s (2 Links)
Number of Omnipath switches (100SWE48)	10 + 4 (each 48 Ports)
Bisection bandwidth of interconnect	1.6 TB/s
Latency of interconnect	2.3 μs
Peak performance of system	394 TFlop/s
Infrastructure	
Electric power of fully loaded system	62kVA
Percentage of waste heat to warm water	97%
Inlet temperature range for water	30 ... 50°C
Temperature difference between outlet and inlet	4 ... 6°C
Software (OS and development environment)	
Operating system	SLES12 SP2 Linus
MPI	Intel MPI 2017, OpenMPI
Comiplers	Intel icc, icpc, ifort 2017
Performance libraries	MKL, TBB, IPP
Tools for performance and correctness analysis	Intel Cluster Tools

Table 9: An overview of CoolMUC3 characteristics

```
# Only for Implicit
export AUTO_HBW_LOG=0
export AUTO_HBW_SIZE=1B

# Only for Affinity
export KMP_AFFINITY=verbose,granularity=core,compact
export KMP_HW_SUBSET=64c,1t

# Extending Affinity only for MPI-only and Hybrid
export I_MPI_PIN=1
export I_MPI_PIN_MODE=lib
# only for MPI-Only
export I_MPI_PIN_DOMAIN=core
# only for Hybrid
export I_MPI_PIN_DOMAIN=node

# Run command for OpenMP Implicit
OpenMP_M_Command="LD_PRELOAD=libautohbw.so:libmemkind.so numactl --membind=$membind"
                   "./openmpcsr4mlem A g $outputFile nIterations"

# Run command for OpenMP Explicit
OpenMP_X_Command="numactl --membind=$membind
                   ./openmpcsr4mlem A g $outputFile nIterations"

# Run command for MKL Implicit
MKL_M_Command="LD_PRELOAD=libautohbw.so:libmemkind.so numactl --membind=$membind"
                "./mklcsr4mlem A g $outputFile nIterations"

# Run command for MPI Explicit
MPI_X_Command="mpirun -n 64 ./mpicsr4mlem A g $outputFile nIterations 0"

# Run command for Hybrid Explicit (2 nodes
HYB_X_Command="mpirun -genvall -n 2 -ppn 1 numactl --membind=$membind"
                "./mpicsr4mlem A g $outputFile nIterations 64"
```

Snippet 8: The environment variable declaration and run commands for all setups.

List of Figures

List of Tables