6-9-2016

Simulating Nonlinear Neutrino Oscillations on Next-Generation Many-Core Architectures

Seyed Vahid Noormofidi

SeyedVahid Noormofidi

Candidate

Electrical and Computer Engineering

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Huaiyu Duan , Chairperson

Susan R. Atlas

Mark Gilmore

Yin Yang

Rouzbeh Allahverdi

Simulating Nonlinear Neutrino Oscillations on Next-Generation Many-Core Architectures

by

Vahid Noormofidi

B.S., Computer Science, Amirkabir University of Technology (Tehran Polytechnic), 2006

M.S., Computer Science, University of Tehran, 2008

DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Engineering

The University of New Mexico

Albuquerque, New Mexico

May, 2016

Dedication

To my mother and my wife, that without their sacrificial care and unconditional love none of this would be possible;

To my dear uncle, who has invariably encouraged me to pursue careers in science;

And to the memory of my cousin, Ali, who always supported me like a brother.

Acknowledgments

This work is supported in part by DOE grant DE-SC0008142 (Prof. H. Duan) and NSF grant MRI OCI-1040530 (Prof. S. R. Atlas) at the University of New Mexico.

We are grateful to the Texas Advanced Computing Center and UNM Center for Advanced Research Computing for providing some of the computational resources for the development of this work.

We are thankful to department of Physics and Astronomy, Prof. H. Duan, Prof. S. R. Atlas, Dr. John Cherry, Dr. Shashank Shalgar, Sajad Abbar, and Lei Ma for their great help and assistance during the development of this project.

Simulating Nonlinear Neutrino Oscillations on Next-Generation Many-Core Architectures

by

Vahid Noormofidi

B.S., Computer Science, Amirkabir University of Technology (Tehran Polytechnic), 2006

M.S., Computer Science, University of Tehran, 2008

Ph.D., Engineering, University of New Mexico, 2016

Abstract

In this work an astrophysical simulation code, XFLAT, is developed to study neutrino oscillations in supernovae. XFLAT is a hybrid modular code which was designed to utilize multiple levels of parallelism through MPI, OpenMP, and SIMD instructions (vectorization). It can run on both the CPU and the Xeon Phi co-processor, the latter of which is based on the Intel Many Integrated Core Architecture (MIC). The performance of XFLAT on various system configurations and physics scenarios has been analyzed. In addition, the impact of I/O and the multi-node configuration on the Xeon Phi-equipped heterogeneous supercomputers such as Stampede at the Texas Advanced Computing Center (TACC) was investigated.

Contents

Glossary

Accelerator	A processor that is typically installed as a card on the motherboard. The main processor can offload its tasks onto accelerator. Examples include GPU, FPGA, ...
API	Application Programming Interface (API) is a set of functions and protocols to build software applications.
Compute node	An independent machine on which one or more CPUs are installed. Some may also be equipped with co-processors. Compute nodes can be linked together via high-performance networks such as InfiniBand.
Co-processor	A processor that is typically installed as a card on the motherboard. The main processor can offload its tasks onto co-processor. Co-processors can also be employed independently.
CPU	A Central Processing Unit is an integrated circuit chip that executes instructions of a computer program by performing the arithmetic, logical, and control operations specified by the instructions.
GPU	Graphics Processing Unit. Traditionally utilized for computer graphics tasks. Modern GPUs such as NVidia Tesla can also be employed for high-performance computing applications.

HDD	Hard Disk Drive. A type of computer storage in which data is stored on one or more rapidly rotating metal disks.
Library	A collection of pre-written codes, procedures, classes, or values that may be employed for developing softwares.
MPI	Message Passing Interface is a portable and standardized message-passing system that can be used for inter-node communications. It provides a standard library APIs with which a program can be linked to.
Node	An independent machine on which one or more CPUs are installed. Some nodes may be equipped with co-processors as well. Nodes can be linked together via high-performance networks to compromise distributed parallel clusters.
OpenMP	Open Multi-Processing is an interface for parallel programming on shared memory multiprocessing environments. It provides a runtime system that can be employed via compiler directives.
PCI Express	Peripheral Component Interconnect Express. A high-speed serial bus that typically connects accelerators to processors.
RAM	Random-Access Memory. It is a type of memory that allows data elements to be accessed irrespectable of their physical location inside the memory. RAM is normally associated with volatile types of memory in which stored data is lost is the power is removed. Most modern computers' main memory is RAM.
Run Time	Run time or runtime is the time during which an application is executing.

SIMD	Single Instruction, Multiple Data is a parallel programming approach in which machines exploit data-level parallelism by issuing a single instruction applied to multiple data streams simultaneously.
Supercomputer	A powerful computer containing many individual computing nodes that are connected via a high-speed interconnection link.
Thread	A lightweight process that may share some part of its memory with the other threads.

Chapter 1

Introduction

1.1 Astrophysics, Supernovae, and Neutrinos

Human beings have always been curious about the universe. Astrophysics is the branch of science that attempts to understand the universe and the human future in it. The goal of astrophysics is "to ascertain the nature of the heavenly bodies, rather than their positions or motions in space." [Keeler, 1897]. There are a broad range of objects that are studied in astrophysics such as: extra-solar moons and planets, stars, supernovae, white dwarfs, neutron stars, black holes, galaxies, quasars, dark matter, dark energy, and the cosmic microwave background. Astrophysics address fundamental questions including the origin and evolution of stars, galaxies, and the universe.

At the end of its life, a massive star explodes as a *supernova* and at the same time its core collapses under its own gravity into a neutron star, which is the remnant of the core containing mostly neutrons [Woosley and Janka, 2005]. Therefore, during the explosion a supernova can briefly outshine the entire galaxy. In a few seconds a supernova emits as much energy as the Sun emits over its entire life span [Giacobbe, 2005].

The explosion expels almost all of a star's material at relativistic speeds, driving a shock wave into the surrounding interstellar medium [Schawinski et al., 2008]. Supernovae are crucial to the chemical evolution of the universe which had only hydrogen and helium immediately after the Big Bang. All other heavier elements are born inside stars, during supernovae, or potentially during other dramatic astrophysical events such as neutron star mergers. Therefore, without supernovae none of us would exist, since we are made up from heavy elements that are generated or distributed by supernovae.

Approximately, 99% of the total energy of a supernova is carried away by about 10^{58} particles called neutrinos within a minute of core collapse [Pagliaroli et al., 2009]. Neutrinos are extremely difficult to detect as they have no electric charge and they interact with matter only through the weak interaction. In fact, about 65 billion solar neutrinos per second pass through every square centimeter on the surface of the Earth [Bahcall et al., 2005]. There are three types or *flavors* of neutrinos in the particle physics Standard Model: electron neutrino, mu neutrino, and tau neutrino (ν_e, ν_μ, and ν_τ). Neutrinos play critical role in nucleosynthesis processes in supernovae. For example, the electron neutrino can affect the number density of protons and neutrons:

$$\nu_e + n \rightleftharpoons p + e^-$$
$$\overline{\nu}_e + p \rightleftharpoons n + e^+$$

where n, p, e^-, and e^+ are neutron, proton, electron and positron, respectively. Therefore, neutrinos can influence the production of heavy elements in supernovae ejecta by changing the proton to neutron ratio. In addition, since the supernova envelope is transparent to neutrinos, by investigating the neutrinos that are emitted from supernovae, astrophysicists can probe into supernovae themselves.

One of the most important breakthroughs in particle physics over the past twenty years was the discovery of neutrino (flavor) oscillations in which neutrinos of different flavors change to another during propagation [Olive et al., 2014]. In supernovae neutrinos of different flavors can have different luminosities (*i.e.,* total energy). In addition, they have different average energies which are characterized by their energy spectral functions. The energy spectrum of the neutrino is proportional to the number of neutrinos as a function of the neutrino's energy. Since only the electron-flavor neutrinos and anti-neutrinos play important roles outside the neutron star [Duan et al., 2006], and because neutrinos can change their flavors, neutrino oscillations are important to supernova dynamics and to the origin of the heavy elements.

There are two complementary ways to study neutrino oscillations in supernovae. The first approach is to detect neutrinos emitted by a supernova. However, supernovae are rare events, and on average only one supernova occurs in a century in our home galaxy [Hirata et al., 1987, Bionta et al., 1987]. The other approach is to study them via simulations and to model neutrino flavor oscillations.

In principle, the flavor quantum state of neutrinos emitted from a proto-neutron star [Thompson et al., 2001] in a supernova depends on seven parameters: time (t), distance (r), emitting position on the surface (Θ, Φ), energy (E), and trajectory direction (ϑ, φ). A complete study of neutrino evolution in supernovae depends on all seven variables and cannot be solved analytically or simulated computationally at present. In order to study neutrino oscillations simpler models are always employed [Duan et al., 2006]. For example, the *bulb model* is the simplest stationary model in which spherical symmetry is assumed [Duan et al., 2006]; thus, the only parameters are r, E, and ϑ. Even in this simple model, thousands of angle trajectories and hundreds of discretized energy bins are required in order to achieve good numerical resolution. In more complicated models, orders of magnitude more trajectories are required for each new parameter added back into the model. In addition, it is essential for a complete simulation to store the flavor quantum state of the many interacting particles in memory in order to perform the simulation. Therefore, the very large number of neutrino trajectories in the system as well as the complexity of the geometry and environment make these simulations very challenging, and the use of advanced computing architectures, including next-generation supercomputers, has the potential to enable increasingly sophisticated simulations of supernova physics.

One might wonder if within a few years this complexity and high computational demand could be largely addressed by Moore's law. Moore stated back in 1965 (reviewed in [Moore, 2006]) that the number of transistors in a dense integrated circuit doubles approximately every year (or doubles every two years in the 1975 revised version of Moore's law [Moore et al., 1975]); consequently microprocessor and compute capabilities correspondingly double at the same rate. However, while computing capability has indeed been growing exponentially since then (see Fig. 1.1), adding even one more parameter to a simulation can cause computational complexity to grow by several orders of magnitude (see Fig. 1.2). In order to perform a complete neutrino oscillation simulation, large numbers of neutrino trajectories will typically be required. This requirement dictating the simulations' fidelity has to be balanced against the ability to perform a feasible simulation on the current generation of supercomputers.

1.2 The History of Supercomputers

A supercomputer is a computer with a higher-level computational capacity than desktop machines or small servers. The performance of a supercomputer is measured in *floating point operations per second* (FLOPS). Modern supercomputers can perform quadrillions of floating point operations per second and it is expected that within a decade their performance will reach exaflops (10^{18} floating point operations per second) [Wikipedia, 2015g].

The history of supercomputers dates back to the 1960s when Seymour Cray de-

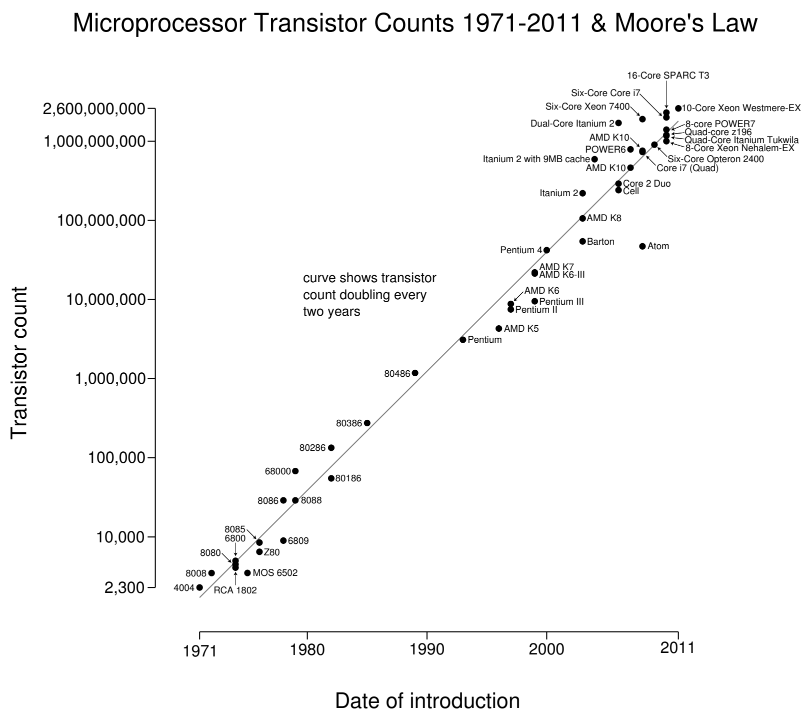


Figure 1.1: CPU transistor counts as a function of date of introduction [Wikipedia, 2015i].

signed the first commercial supercomputer, the CDC 6600, with nearly 1 megaflops performance [Wikipedia, 2015b]. The CDC 6600 *Central Processing Unit* (CPU) was dedicated solely to computations rather than handling all tasks such as memory mangement and I/O. This was the first example of what later came to be called *reduced instruction set computer* (RISC) design. The CDC 6600 was succeeded by the CDC 7600, which could deliver ten times more performance [Wikipedia, 2015c]. For the CDC 7600 Cray focused on the concept of an *instruction pipeline.* A pipeline improves the performance by feeding in the next instruction before the first has completed (similar to assembly line in a manufacturing process), thereby having each unit effectively work in parallel, as well as the machine as a whole (see Fig. 1.3). Cray continued his domination in supercomputing in the 1970s by introducing the Cray-

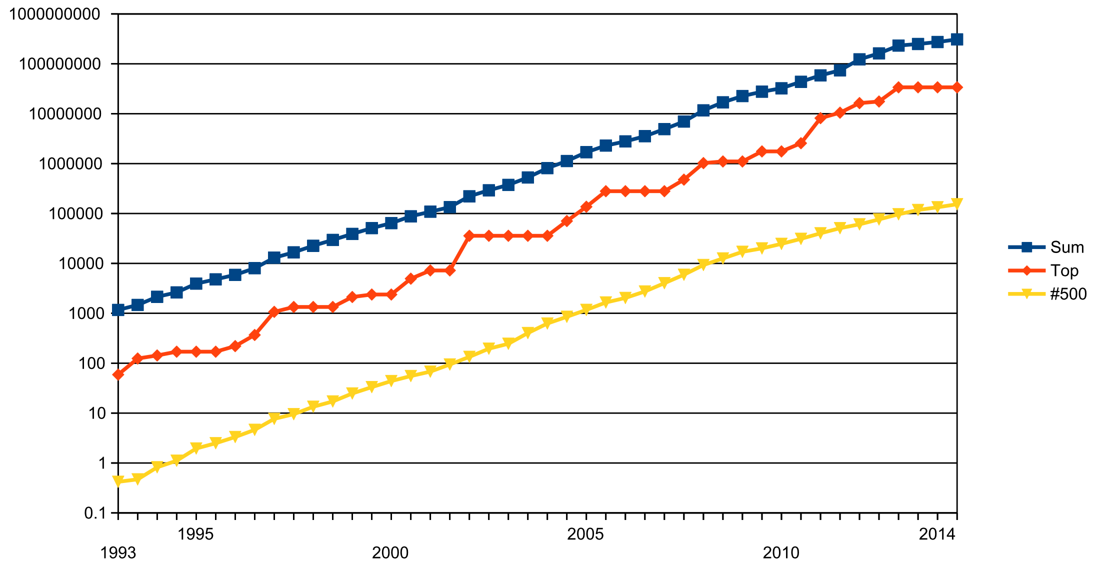
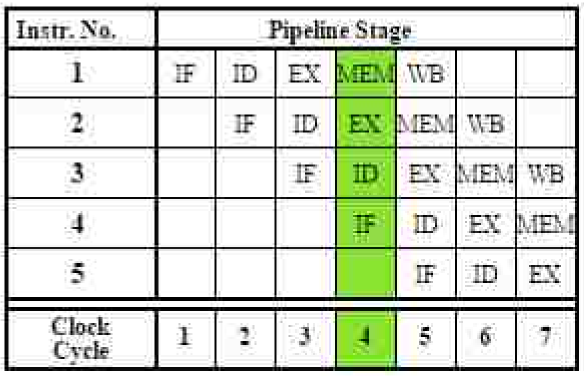


Figure 1.2: The combined performance of the 500 largest supercomputers (blue), the fastest supercomputer (red), and the supercomputer at 500th place (yellow) as functions of time [Wikipedia, 2015k]. Data based on Top500 list (`http://http://top500.org/`).

1, which could deliver ten times more performance than the previous record holder by utilizing vector processors (see Fig. 1.3) [Wikipedia, 2015e]. Compared to scalar processors, whose instructions operate on single data items, a vector processor is a CPU that implements an instruction set containing instructions that operate on an array of data items called *vectors*. The Cray-1 was succeeded in 1982 by the 800 megaflops Cray X-MP, the first Cray shared-memory parallel vector processor in which the memory was shared between two processors [Wikipedia, 2015f]. In late 1985 the nCUBE 10 was released, which was based on a set of custom chips. In each compute node of nCUBE10 there was a processor chip with a 32-bit Arithmatic Logic Unit (ALU) and a 64-bit Floating-Point Unit (FPU) [Wikipedia, 2015j]. Also, in 1985 the very advanced Cray-2, the first gigaflops-capable supercomputer capable of 1.9 gigaflops peak performance, succeeded the first two models as the world's fastest supercomputer [Wikipedia, 2015e]. During the 1980s there were also other designs such as the Thinking Machines CM-1 and CM-2 in which a hypercube design were employed [Wikipedia, 2015d]. Each CM-1 microprocessor had its own 4 kbits of RAM, and the hypercube array was designed to perform the same operation on multiple data points simultaneously, *i.e.*, to execute tasks in the single instruction, multiple data (SIMD) fashion. The supercomputing trend continued in the 1990s with the introduction of massively parallel supercomputers such as Thinking Machines CM-5/1024, the first supercomputer with more than one thousand processors [Wikipedia, 2015d]. In 1996, ASCI Red became the first supercomputer to break the 1 teraflops performance barrier. The ASCI Red computer was the first large scale supercomputer to be built entirely of common commercially-available components [Wikipedia, 2015a]. Prior to 2000, nearly all commercial CPU architectures were of single-core, single-threaded design; SIMD instruction sets were not even available in commodity CPUs before 1997 [Peleg et al., 1997]). During the 2000s the use of commercial CPUs in supercomputers accelerated. It was in 2008 that the IBM Roadrunner became the first supercomputer to perform more than 1 petaflops of computations [Wikipedia, 2015h] using Cell processors which were also used in the PlayStation 3 game console [Costigan and Scott, 2007]. Roadrunner was also an early hybrid supercomputer. Most previous supercomputers only used one type of processor, (CPU). Roadrunner, however, employed TriBlades in which two dual-core Opterons and four PowerXCell 8i CPUs interconnected to each other [Wikipedia, 2015h]. Roadrunner can be considered as an Opteron cluster with Cell accelerators, with each node consisting of a Cell attached to each Opteron core and the Opterons to each other. In the last decade mainstream microchip companies such as Intel and AMD have started packing more cores on a single die area [Geer, 2005], adding multi-threading (having multiple tasks run on a single core simultaneously) functionality onto each core [Magro et al., 2002], and increasing the length of SIMD registers [Thakkur and Huff, 1999, Firasta et al., 2008] by



Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 1.3: Basic five-stage pipeline (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline [Wikimedia, 2015].

several folds. Consequently, hierarchically-parallelized software has started to play a more important role in mainstream desktop applications such as web-browsers, video games and office applications.

In recent years, the High-Performance Computing (HPC) community has adopted the concept of heterogeneous computing and has shifted from employing only traditional CPUs to also incorporating accelerators and co-processors such as nVIDIA Tesla Graphics Processing Unit (GPU) [Kirk, 2007] and the Intel Xeon Phi with the Intel Many Integrated Core Architecture (Intel MIC) [Chrysos and Engineer, 2012]. Today, most petaflop class supercomputers have hybrid or heterogeneous designs, meaning that they are equipped with accelerators, such as GPU and Xeon Phi, in addition to traditional CPUs. These accelerators and co-processors, which are extension cards installed on computers' motherboards, are designed for fine-grained, massively parallel computation and possess superior computing capabilities. Modern supercomputers typically exploit several levels of parallelism. As illustrated in Fig. 1.4, a modern supercomputer may contain many compute nodes which are connected to each other via high-speed interconnect links. Each compute node may

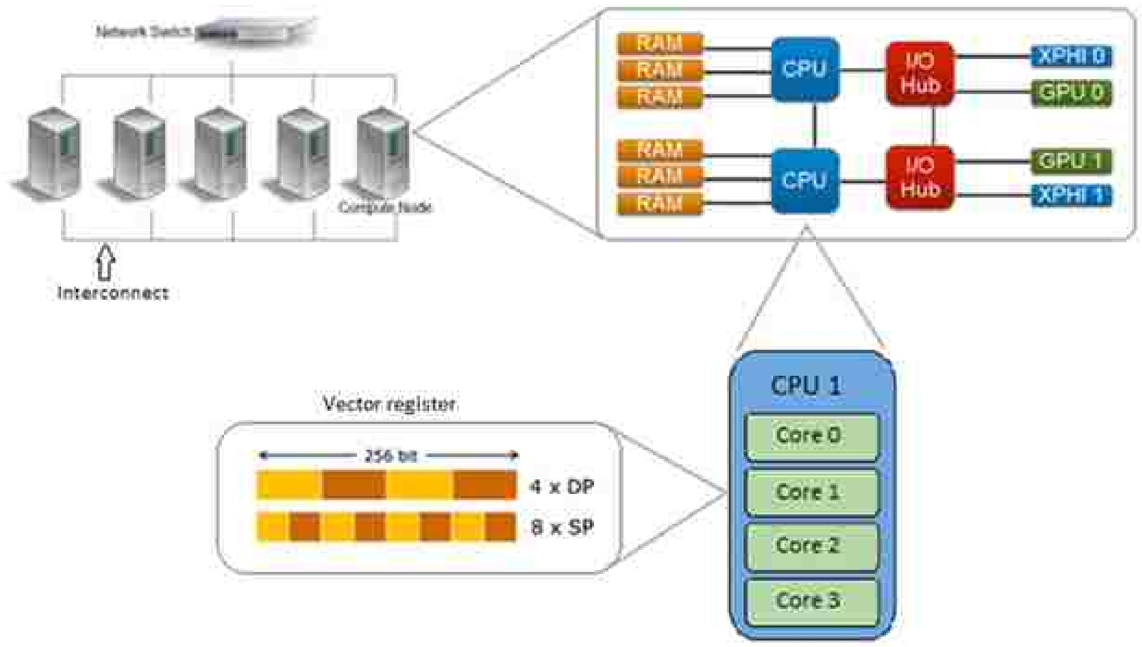


Figure 1.4: Cluster nodes and internal architecture of each node on a heterogeneous supercomputer. At the highest level compute nodes are interconnected via high-speed interconnect links. Inside each node, there may be multiple CPUs as well as accelerator cards. Typically, inside each processor there are a few to many cores. Finally, within each core, a vector instruction can be issued on multiple data simultaneously.

have several co-processors and CPUs. In each CPU or co-processor, there are a few to many cores available. Each of the cores may be able to handle several hardware-enabled threads. Finally, depending on the processors' architecture, the execution units of each core may be equipped with vectorized units, which can execute a single instruction on multiple data. Hybrid codes are required to fully utilize all of the different levels of parallelism on a modern supercomputer. Hybrid codes can utilize the vectorization units to process arrays of data, and different cores and processors and multiple compute nodes to process different chunks of data. Therefore, multiple order of magnitude speedups can be expected when a fully optimized and parallelized code runs on a modern supercomputer.

1.3 Accelerators

In a heterogeneous supercomputer, accelerators such as Graphics Processing Units (GPU) and the Intel Xeon Phi are employed to boost computing capability and improve the concurrency of each node. Initially, GPUs were introduced for computer graphics tasks, especially video game computations, so that CPUs could be freed up for other tasks. As video game complexity increased the demand for more powerful GPUs drove the evolution of GPUs to the point where GPUs were flexible enough for general computations other than graphics [Mark et al., 2003]. The trend of generating higher quality graphics has made GPUs even more powerful than CPUs [Kirk, 2007].

The latest generation of mainstream CPU architectures have typically less than ten cores available on the die area [Wikipedia, 2014], and largest part of the die area is dedicated to cache memory and the control unit [Magro et al., 2002]. Therefore, the execution units occupy only small part of the die area. On the other hand, accelerators and co-processors usually have many smaller and simpler cores than the CPU such that they can handle a larger number of light, in-order hardware-enabled threads (see Fig. 1.5), and most of the die area is dedicated to FPU and ALU, which are for instruction execution.

Despite the fact that GPUs can accelerate scientific computations, the programming model of GPUs can be challenging. Traditional CPU codes are not able to run on GPUs without modifications in their data structures, algorithms, and memory models. Therefore, new codes must be developed to run on GPUs. In addition, low-level knowledge of GPU architecture is needed, including GPUs' memory hierarchy and new parallel programming paradigms, in order to effectively harness the power of GPUs efficiently.

Having failed with their experimental Larrabee project (Intel GPU chip code-

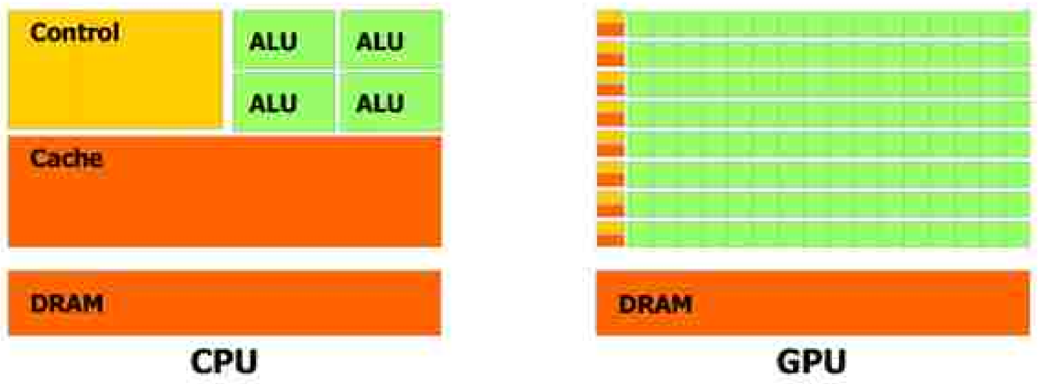


Figure 1.5: Comparison of the CPU and GPU die area. The CPU has more on-chip memory (shown in orange) and larger control unit (shown in yellow) than GPU. However, the GPU devotes more transistors to data processing (shown in green) than CPU [NVIDIA, 2012a].

name for GPU) [Seiler et al., 2008], Intel introduced the Many Integrated Core Architecture (MIC) in Xeon Phi co-processors in 2012 [Chrysos and Engineer, 2012]. Due to the challenges of GPU programming, Intel decided to stick with its established x86 microprocessor architecture for its teraflops co-processor. The Xeon Phi can be installed on a compute node as an extension card (similar to GPU cards), and it runs an embedded micro-kernel Linux Operating System (OS) in memory. Intel MIC is principally a many-core x86-64 microchip with several extensions [Seiler et al., 2008, Chrysos and Engineer, 2012]. As a result, legacy codes may be compiled and run on the Xeon Phi without significant modifications. Being able to write a parallel code once and run it on both CPU and MIC architectures is a major benefit for developers, thus a great advantage for Intel MIC over GPU programming. However, many applications do not fully exploit multiple levels of parallelism. As a result, code optimization is necessary in order for legacy codes to run efficiently on the Intel MIC. Since there are still several hardware differences between the Intel MIC and traditional CPU, code optimizations are invariably needed in order to make the chip appropriate for high-performance parallel applications. For example, the chip has a smaller cache area to make room for a higher number of cores. In fact, Intel completely removed the Level 3 cache [Chrysos and Engineer, 2012]. Additionally, in comparison to traditional out-of-order x86 CPU cores, each core on Xeon Phi has simpler in-order design [Chrysos and Engineer, 2012], and is capable of handling four hardware threads in contrast to just two on the CPU. Moreover, each thread on the Xeon Phi has access to wider Single Instruction Multiple Data (SIMD) vector registers [Chrysos and Engineer, 2012]. Nevertheless, by taking the advantage of the same optimization techniques for both the CPU and the MIC, performance improvement can be obtained on both architectures. However, to truly take advantage of the power of the Xeon Phi, a computer code must be carefully designed so as to utilize its special features, such as 512-bit-long registers and 200+ hardware threads.

The goal of this work was to design and create a new high-performance software framework for astrophysics simulations of neutrino oscillations in supernovae which can run on next-generation many-core architectures hardware as well as existing compute platforms. The new code, XFLAT, is capable of employing several levels of parallelism and is shown to scale very well on multi-node systems. It is designed with a hybrid architecture in order to exploit multiple layers of parallelism simultaneously. It is also designed to be modular so that new physical simulations can be performed by simply swapping in new modules.

Chapter 2

Accelerators and Co-processors

The trend of employing non-general purpose processors for scientific computations has increased dramatically in recent years. The introductions of General-Purpose computation on Graphics Processing Units (GPGPU) and Intel Many Integrated Core (MIC) Architecture can be seen as two pivot points in the high-performance computing trend in recent years.

2.1 GPUs

CPUs have been evolving and adding new parallelism capabilities over the past twenty years. By contrast, GPUs were invented from the start to perform image processing tasks on a large set of pixels in parallel. As a reuslt, there is commonly more than one logical and computational unit in GPU architecture design. Hence, GPU designs have been parallelized from the start. However, their architecture was a fixed-function rendering pipeline (the rendering pipeline is the sequence of steps that are taken when rendering objects) for many years until 2001 when programmable pipelines were introduced [Lindholm et al., 2001]. A fixed-function pipeline contains

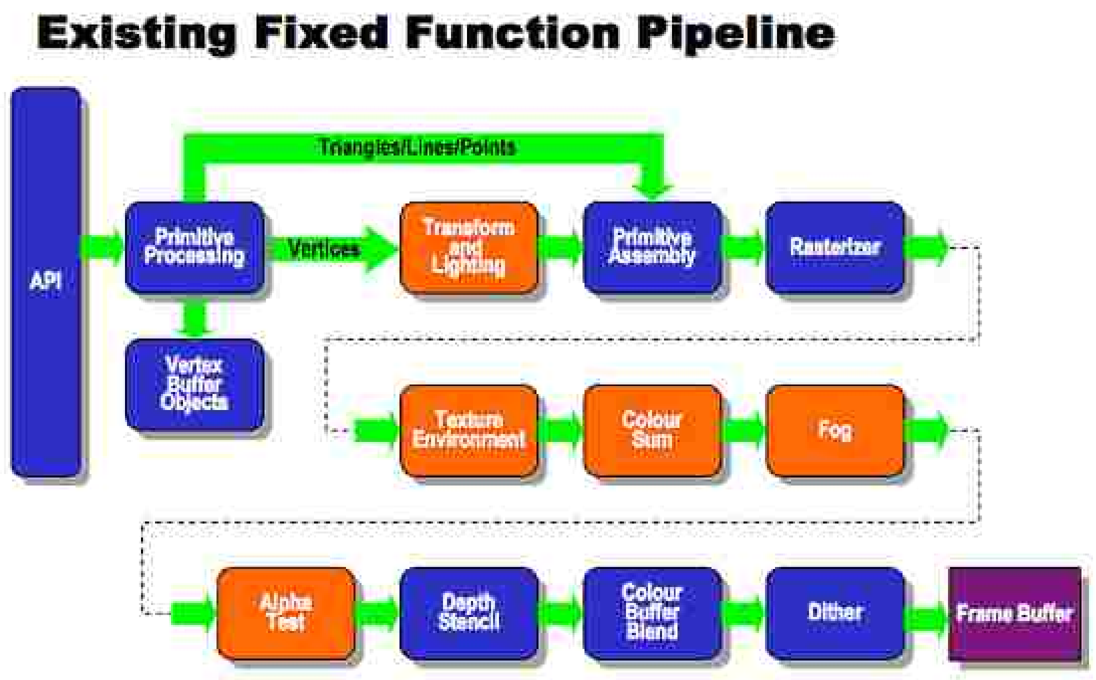


Figure 2.1: Fixed function pipeline hardware prior to the introduction of programmable pipelines in GPUs. Each box in orange has a fixed and non-programmable functionality. Following the introduction of programmable shaders, these were replaced with vertex and fragment shader units. The boxes in blue remain identical before and after the introduction of programmable pipelines (image from khronos.org 2015).

a set of configurable processing states that are accessible by a set of callable functions [OpenGL, 2015a]. A graphics processing unit that contains a fixed-function pipeline exposes a different, predefined rendering pipeline based on user-provided configurations, rather than user-provided programs [OpenGL, 2015a]. Therefore, based on rendering criteria, user can decide which one of the predefined rendering pipeline configuration have to be loaded onto the hardware (see Fig. 2.1).

In 2001 NVIDIA introduced its programmable pixel *shaders* pipeline architecture in their GPUs [Lindholm et al., 2001], which allowed programmers to develop customized shading programs on GPUs for the graphics and image processing fields such as stereo matching, linear algebra, and Fourier transformations [Purcell et al., 2002, Krüger and Westermann, 2003, Sugita et al., 2003, Moreland and Angel, 2003] and other scientific research branches and purposes including [Thompson et al., 2002, Mark et al., 2003, Atanassov et al., 2003]. A shader program is a piece of code which performs graphics shading and apply special color effects to objects. This transition from fixed-function pipeline, which is a non-flexible pipeline for performing fixed tasks, to a programmable pipeline which is shader-based, introduced General Purpose computing on GPUs (GPGPU) and opened a new door for high performance computing as well [1].

For a long time after the introduction of programmable shaders, developers had to write their own shaders in GPU's assembly language [2]. Later, several higher-level languages were introduced which made GPU programming easier. One of the first high level GPU programming languages was NVIDIA Cg (C for graphics) [Mark et al., 2003]. Cg was a flexible C-like language that was designed for GPU shader programming. It allowed graphics programmers to configure the vertex

[1]Previously GPU computational power was only available for video games and rendering applications, even though GPUs can be capable of doing more single precision floating point operations per second (FLOPS) than CPUs [Nyland et al., 2007]. Before the introduction of programmable pipelines in GPUs, in order to use GPUs for scientific research, scientists had to develop low-level understanding of the rendering pipeline. Furthermore, most of the time there was no suitable way to map a scientific problem onto the fixed-function pipeline architecture. Consequently, the use of GPUs for general purpose sicentific research was limited [Swanson, 1995]

[2]After the introduction of programmable shaders for the rendering pipeline in GPUs, scientists started to study the appropriateness of GPUs in other research fields. In the early generation of programmable GPUs, it was possible to write simple programs for the *Vertex shader* and *Pixel shader* units of the GPU. Since the basic polygon for representing a graphical scene is a triangle, a special dedicated hardware unit is needed for performing calculations on triangles' vertices in any GPU. Vertex Shader is the programmable shader stage in the rendering pipeline that handles the processing of individual vertices [OpenGL, 2015c]. Similarly, the other important stage of the rendering pipeline relates to the processing of individual pixels in which most of the GPU computations occur. Hence, the Pixel Shader or Fragment Shader is a user-supplied program that performs pixel processing such as setting of colors and depth values [OpenGL, 2015b]

and pixel shader units and, therefore, change the behavior of the GPU rendering pipeline without dealing with the low-level assembly language. Using Cg programmers were able to write high-level structures and algorithms with the use of standard programming constructs such as for-loops, if-else statements, functions etc. Because each vertex is addressed by using three numbers (for the x, y, and z directions) and each pixel contains multiple color channels (red, green, blue, and alpha blending), a functionality similar to SIMD was present in Cg which together with multiple ALUs on vertex and pixel shader units provides significant parallelism for the programmers to utilize at a high level.

The GPGPU trend continued further following the introduction of the Compute Unified Device Architecture (CUDA) by NVIDIA in 2007 [Kirk, 2007, Buck, 2007]. CUDA platform enable developers to harness GPU power for general high-performance computing research and applications. Since then, the number of cores on GPUs has increased dramatically, from tens in the first generation CUDA-enabled GPUs, to thousands in the GPUs in Kepler and Maxwell generations [NVIDIA, 2012b]. As shown in Fig. 2.2, there are many simple cores in one GPU block in contrast to a few complex cores in a typical CPU. In high-end GPUs such as the Kepler GK110 illustrated in Fig. 2.3, there is more than one core block (a.k.a. Streaming Multiprocessors) on the die area. The parallel nature of the GPU itself plays an essential role in scientific computations. Thus, by employing CUDA or other GPU programming languages such as the Open Computing Language (OpenCL) [Stone et al., 2010] or C++ Accelerated Massive Parallelism (C++ AMP) [Gregory and Miller, 2012], scientists can write general applications which can execute on the GPU without the need to be familiar with computer graphics or the graphics pipeline.

The adaptation of GPU for scientific research has grown dramatically in the past decade [Ryoo et al., 2008, Nickolls et al., 2008]. Porting legacy codes to GPUs has helped researchers to accelerate many applications. For example, molecular dy-

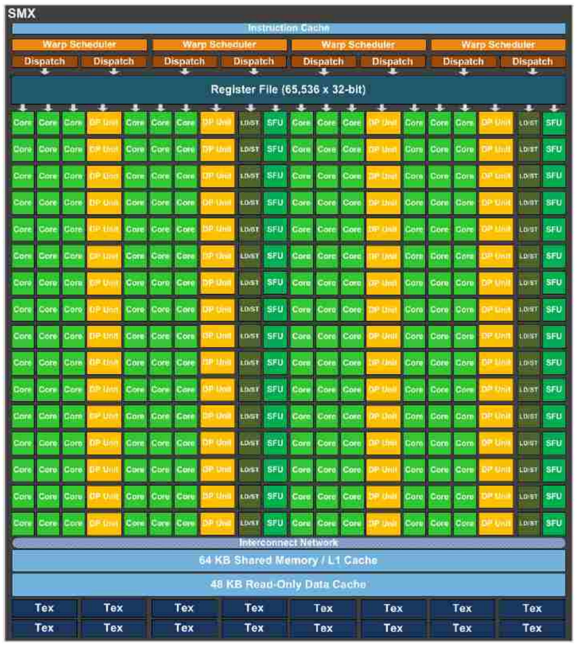

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 2.2: A cluster of tiny cores for the NVIDIA Kepler GPU architecture. There are hundreds of single floating point units (shown in light green) as well as double precision units (shown in yellow). The other important units are the special function units (shown in green) which are responsible for the computations of transcendental functions (Figure from NVIDIA 2012).

namics research codes LAMMPS [Brown et al., 2011, Brown et al., 2012], NAMD [Phillips et al., 2008], and GROMACS [Hess et al., 2008] have accelerated key computational kernels using GPUs. GPUs have also been utilized in particle simulations, such as cosmological simulations [Hamada et al., 2009, Spurzem et al., 2009, Belleman et al., 2008, Nyland et al., 2007], artificial neural networks [Jang et al., 2008], graph theory [Harish and Narayanan, 2007, Vineet and Narayanan, 2008], Fast Fourier Transforms [Nukada et al., 2008], bioinformatics [Ligowski and Rudnicki, 2009], and weather prediction [Michalakes and Vachharajani, 2008].

The wide adoption of GPGPU in the past few years is largely due to the CUDA parallel programming model. CUDA is accessible to software developers via ex-

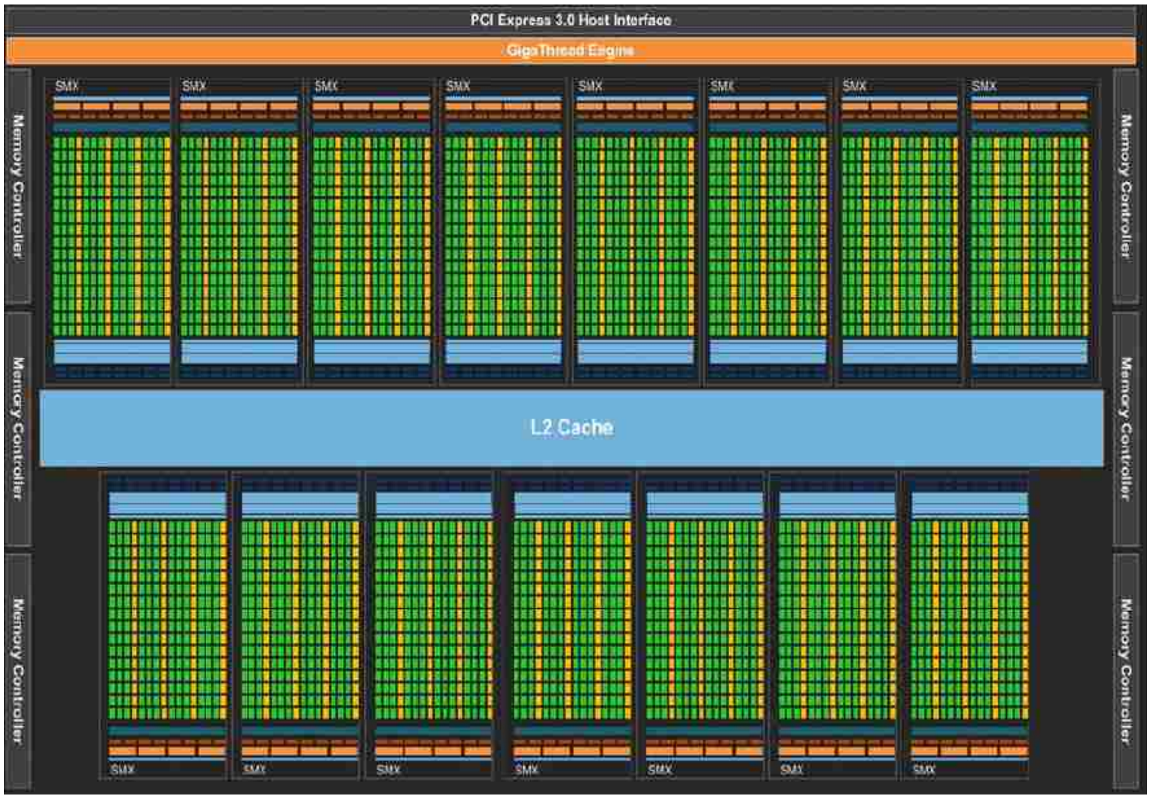


Figure 2.3: Full chip block diagram of NVIDIA GK110 architecture containing clusters (Streaming Multiprocessors) of CUDA cores. Blocks can share data using the shared L2 cache unit (shown in light blue in the middle), and each one can access the global memory outside the chip through memory controllers (shown in grey on the sides) (Figure from NVIDIA 2012).

tensions to standard languages such as C, C++ and FORTRAN. There are three abstractions in CUDA: a hierarchy of thread groups, shared memories, and barrier synchronization. These layers of abstraction provides fine-grained data parallelism and thread parallelism. They are nested within coarse-grained data parallelism and task parallelism. As a result, programmers can partition a problem into coarse sub-problems which can be solved independently in parallel via blocks of threads. Each sub-problem can then be partitioned into finer pieces which can be solved cooperatively in parallel by all threads within the block. At the same time, scalability

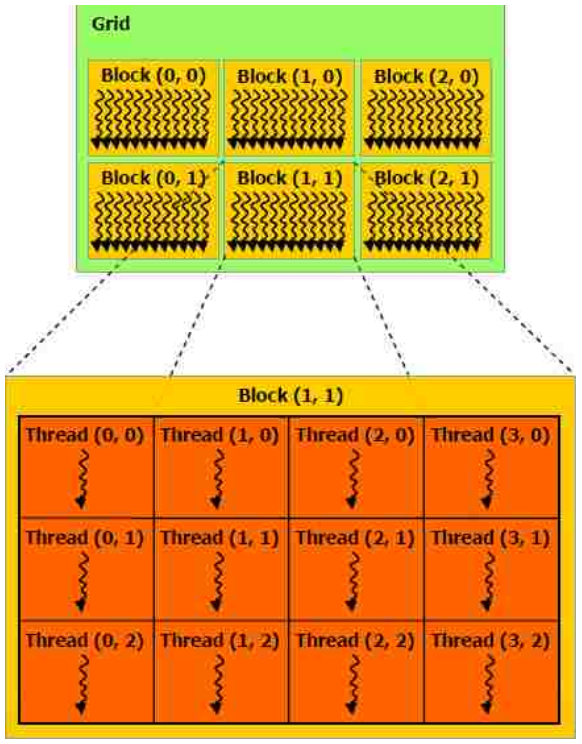


Figure 2.4: Grid of thread blocks. A grid (shown in green) may contain a multi-dimensional array of blocks (shown in yellow). Within each block there is an array of threads (shown in orange) (Figure from NVIDIA 2014).

is preserved since each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially.

In order to execute a computation task on a GPU (a.k.a. the *device*), programmers define special functions, or "kernels" which, when called from the CPU (a.k.a. the *host*), are executed in parallel on streaming multiprocessors. A kernel contains a grid of blocks of threads which have equal numbers of threads (see Fig. 2.4).

CUDA threads may access data from multiple memory regions, as illustrated in Fig. 2.5. Each thread possesses a private local memory where local variables and data are stored. Similar to registers, access time to this type of memory is

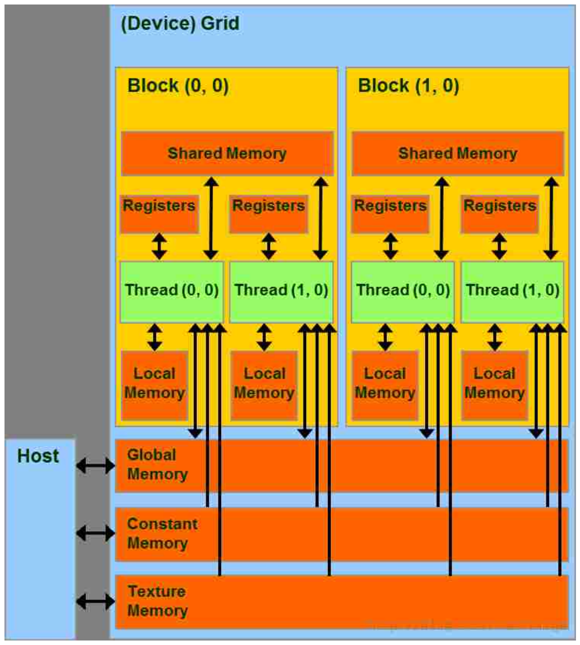


Figure 2.5: CUDA memory hierarchy. Threads (shown in green) can access to different memory locations each with their own advantages and disadvantages (shown in orange) (Figure from NVIDIA 2014).

short. However, if the programmer allocates too much of such memory, the allocated memory will reside on GPU global memory (the off-chip memory). Each block has a shared memory space visible to all threads of the block within the lifetime of the block. The shared memory is also an on-chip memory; thus, the access time is short. In addition, all threads have access to the same global memory, which resides on the graphics card's RAM. There are also two read-only memory spaces accessible by all threads: the *constant* and *texture* memory spaces, which are optimized for different memory usages. As a result, programmers must understand the advantages and disadvantages of each type of memory in order to write codes which can run efficiently on a GPU.

When a kernel is launched from a C code on the CPU (host), only the grid of thread blocks and their associated memory reside on the GPU (device) and the

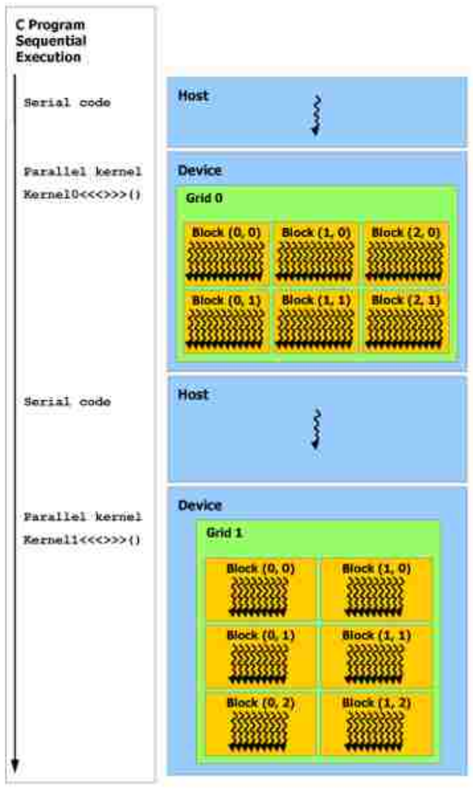


Figure 2.6: Launch of kernels from host. Host and device can have their own executions (shown in blue). Multiple grids (shown in green) can be launched from the host (Figure from NVIDIA 2014).

rest of the program remains on the CPU. Therefore, a host program can continue its execution while the kernels are run on the GPU (see Fig. 2.6). The CUDA programming model assumes that both the host and the device maintain their own separate memory spaces in their own RAM, referred to as the host memory and device memory, respectively. A program on the host manages the global, constant, and texture memory spaces visible to device kernels through function calls to the CUDA runtime system. This includes device memory allocation and deallocation as well as data transfer between the host and device memories.

There are a number of challenges involving the use of GPUs for general-purpose programming. First, traditional C/C++ codes cannot run on GPUs without extensive modifications. Programmers must learn a new language, new programming techniques, and adopt new coding strategies. Second, in order to use CUDA or the other GPGPU programming platforms efficiently, developers must still deal with many low-level details such as micromanaging the shared memory, controlling memory flow between CPU and GPU, and with new versions of old concepts such as grids, blocks, and threads. For these reasons, programmers frequently have to restructure their codes and data structures and develop more optimized algorithms, as well as rewrite many parts of their code so that they can run on GPUs efficiently.

2.2 MICs

2.2.1 The Intel Many-Integrate Core architecture (MIC)

Intel had a GPU project in 2008 that eventually failed [Seiler et al., 2008]. After failing with their experimental GPU, they introduced the Many Integrated Core Architecture (MIC) in Xeon Phi co-processors in 2012 [Chrysos and Engineer, 2012]. Unlike the architectures of GPUs which may change with each generation, Intel decided to stick with their well-established x86 architecture for its teraflops co-processor.

The architecture of the Intel Xeon Phi is depicted in Fig. 2.7. There are about 60 simple x86 cores on the microchip die area. Each core is connected to the other cores via a high-speed bi-directional ring interconnect, which results in full coherency of the Level 2 cache units. Therefore, despite the fact that there is no Level 3 cache for the Xeon Phi, each core can access the L2 cache associated with other cores via the ring interconnect in a constant access time that is not influenced by the location of the cores [Fang et al., 2014]. Thus the entire set of L2 cache units can act in a fashion similar to a distributed L3 cache [Fang et al., 2014]. The type

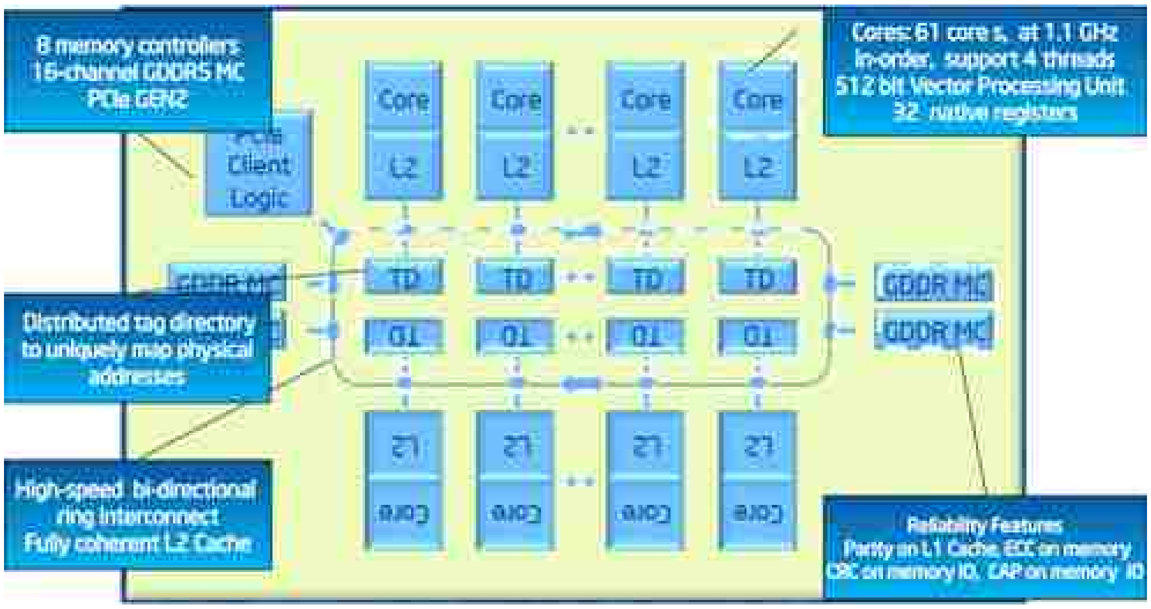

8 memory controllers
16-channel GDDR5 MC
PCIe GEN2
PCIe Client Logic
Core
L2
TD
GDDR MC
Cores: 61 core s, at 1.1 GHz
In-order, support 4 threads
512 bit Vector Processing Unit
32 native registers
Distributed tag directory to uniquely map physical addresses
High-speed bi-directional ring interconnect
Fully coherent L2 Cache
Reliability Features
Parity on L1 Cache, ECC on memory
CRC on memory IO, CAP on memory IO

Figure 2.7: Intel Xeon Phi (MIC) architecture (Figure from Intel 2013).

of memory chip on the card is Graphics Double Data Rate (GDDR) memory, the same memory that also used in contemporary GPU cards. GDDR memory provides higher bandwidth and lower access time in comparison to the main memory for CPUs [Chrysos and Engineer, 2012]. Nevertheless, the Intel MIC sees this memory in exactly the same way that the CPU sees the main memory. All caching tasks are performed in the background and are transparent to the programmer (*i.e.*, the data is fetched from main memory to the L2 cache; then to the L1 cache inside the x86 code and into scalar or vector registers automatically if necessary). In this way, developers do not need to be concerned about understanding different types of memory in the hierarchy or micro-managing the underlying memory caches.

There are two pipelines inside each Xeon Phi core as illustrated by Fig. 2.8. Therefore, the core itself is dual issue per cycle (one for scalar and one for vector) [Jeffers and Reinders, 2013, Intel, 2013] and four hardware threads are available on each core. Most of the vector instructions have a 4-clock latency, which can be hidden

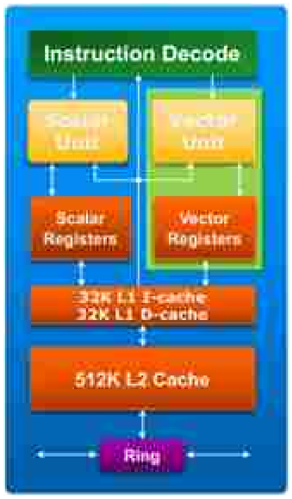


Figure 2.8: Intel Xeon Phi core architecture. There are two execution units per core, the scalar and vector units (shown in yellow). Each unit has access several memory regions (shown in orange) (Figure from Intel 2013).

by round-robin scheduling of multiple threads on a core. Therefore, the effective pipeline throughput is one-per-clock [Jeffers and Reinders, 2013, Intel, 2013]. Since the instruction decoding unit has a 2-cycle latency, no back-to-back cycle issue is possible from the same thread; therefore, at least two hardware threads are needed to fully utilize each core [Jeffers and Reinders, 2013, Intel, 2013]. Unlike Intel's previous vector instruction sets such as the 64-bit MMX which could only operate on integers, the 128-bit Streaming SIMD Extensions (SSE), and the 256-bit Advanced Vector Extensions (AVX) instruction sets, the Intel MIC has a new vector processing unit with 512-bit registers (see Fig. 2.9). As a result, for every cycle, the Xeon Phi can execute an instruction on twice or more data than the CPU.

There are a few differences between the Intel Xeon Phi processor and traditional CPU. First, because the clock rate of the Xeon Phi is lower than that of the CPU, it is not optimized to run serial code as it designed for parallelized/vectorized code. Second, the co-processor usually ships with lower memory than the host and less is

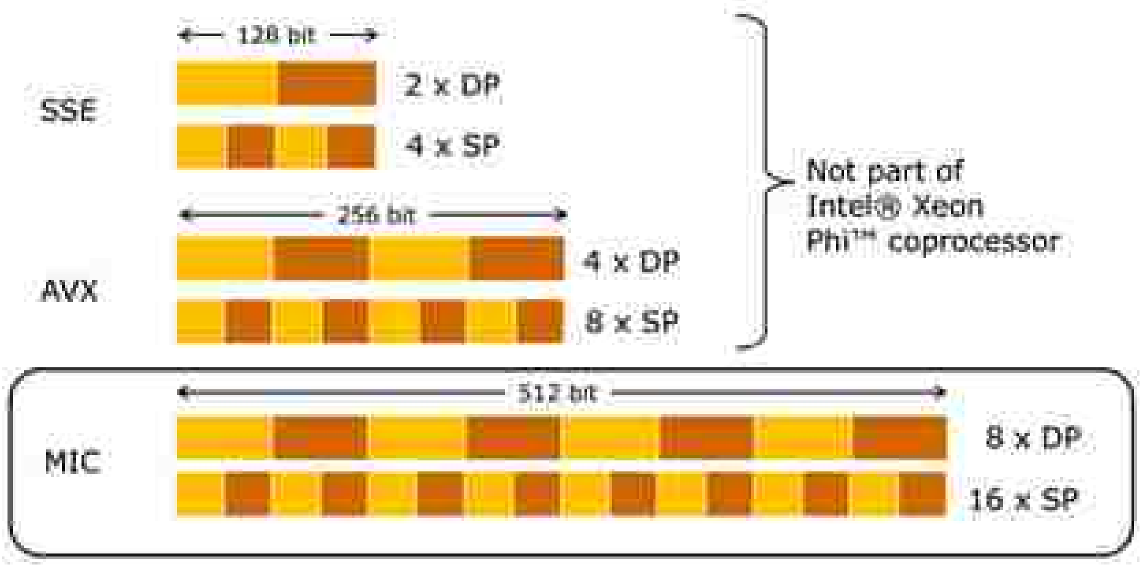


Figure 2.9: Intel Xeon Phi vector processing unit can operate on wider registers compare to traditional CPU's vector units. It can issue instructions on 8 double precision (DP) or 16 single precision (SP) numbers simultaneously (Figure from Intel 2013).

available to programs [Intel, 2013]; thus, programs that require a larger amount of data may not be able to run on the co-processor. Third, compared to the out-of-order CPU cores, the MIC cores are simpler and cannot efficiently handle complicated code paths with many out-of-order jumps or nested conditional blocks.

There are several ways to utilize the vetorization unit on the Intel MIC: Intel Cilk Plus, Intel MKL, compiler auto-vectorization, and SIMD compiler directives [Jeffers and Reinders, 2013]. Intel Cilk Plus is an extension of the C/C++ language which facilitates programming on the Xeon Phi for developers. However, it is still new, not portable, and is only available on Intel compilers. Intel MKL is Intel's math library that can be called within codes. It is optimized and easy to call: no code changes are required. However, MKL provides a math library only and thus cannot be employed for general purpose use. In addition, it has been shown that the Intel MKL is not reliable on non-Intel x86 CPUs (*i.e.* AMD) and causes a drop in the performance of codes [Fog, 2015]. Although the auto-vectorization technique only requires minor changes in codes, programmers need to make loops as simple as possible so that the compiler can vectorize them well [Jeffers and Reinders, 2013]. Furthermore, functionality is limited by language and compiler technology and is not portable. Vectorization with SIMD directives is a reliable technique and is standardized in OpenMP 4.0 [OpenMP4.0, 2015]. The SIMD directives are powerful, yet need to be used carefully in order to avoid changing the meaning of the program [Jeffers and Reinders, 2013].

Similarly, various techniques can be employed for parallelizing code on the Intel MIC. These include Intel Cilk Plus, Intel TBB, and OpenMP. As mentioned earlier, Intel Cilk Plus extends C/C++ for parallelism. The parallelism can be achieved by employing special keywords, attributes, directives, and through the runtime libraries [Jeffers and Reinders, 2013]. Intel Threading Building Blocks (TBB) is a compiler-independent C++ library for rich parallelism via template programming [Jeffers and Reinders, 2013]. Another technique is employing OpenMP for parallelization. OpenMP works with compiler directives and a runtime library which supports parallel loops, the use of a tasking model, and portable locks. Due to the standard nature of OpenMP and its portability, in XFLAT, OpenMP was employed for intra-processor vectorization and parallelization.

There are considerable differences between the MIC and GPU accelerators from the perspective of architectural design. First, in contrast to the GPU, Intel MIC is not optimized for concurrent out-of-cache random memory access by large numbers of threads [Intel, 2013]. Furthermore, the Intel MIC architecture has a traditional coherent-cache architecture, whereas GPUs have a memory architecture specialized for localized shared memory processing [Intel, 2013]. Moreover, “threads” and “cores” mean something very different, as GPU versions are limited and lighterweight [Intel, 2013]. Furthermore, on the GPU, cores are limited to performing simple floating point operations, and a number of threads can act as a single SIMD unit. Since the Intel MIC architecture is similar to traditional CPUs, it is a better choice for accelerating highly vectorized and parallelized codes such that they can be run on both CPU and the MIC without any further modification simultaneously.

Typically, programs run on accelerators such as GPUs via *offload* mode in which the main program still continues its run on the CPU. The computational portions of the code and the required data are only transferred to the GPU. After completing the calculations, the results are transferred back to the main program on the CPU. Hence the non-computational parts such as input/output, MPI calls and other network communication remain on the CPU. However, on the Intel MIC there are two methods for running programs on the co-processor. The first method is to run codes in offload mode in a way that is similar to GPU. This method is compatible with the GPU programming style so that codes written for the GPU can run on the MIC with only minor modifications. Fig. 2.10 illustrates the offload programming model. The other approach is to run codes in *symmetric* mode, which implies an entirely independent program executing on the MIC co-processor [Jeffers and Reinders, 2013]. In this mode, programs run on the Xeon Phi precisely as they run on the CPU. Consequently, the Intel MIC can be seen as an MPI node at runtime, so that all the communication and I/O tasks are performed without interrupting the CPU. Hence, it is possible to have another MPI process running on the CPU simultaneously. Fig. 2.11 depicts different programming models on the Xeon Phi.

In addition, from the perspective of the software stack, there are a number of important differences between GPUs and the Xeon Phi: the former is controlled by driver software, while the latter is controlled by a micro-kernel Linux operating system [Jeffers and Reinders, 2013, Intel, 2013]. As a result, the Linux Standard Base core libraries are available on the Xeon Phi (see Table 2.1). On accelerators such as the GPU, a portion of the driver resides on the CPU (host) and controls the

```
#pragma offload target(mic : target_id) \
                in(data_in : length(SIZE) \
                inout(data_inout: length(SIZE)
for (int i = 0; i < SIZE; ++i)
{
    /// calculations here!
    data_inout[i] += data_in[i];
}
```

Figure 2.10: Offloading a loop to an accelerator. The offload programming model of the Xeon Phi is similar to that of the GPU programming model.

accelerator via the PCI Express bus. The driver is responsible for managing accelerators, providing a low-level software stack for data transfer, and running kernel codes by providing the runtime environment for programs. By having a Linux operating system running on the Xeon Phi, there is no need to have the co-processor controlled from the host. As the co-processor is Internet Protocol (IP) addressable,

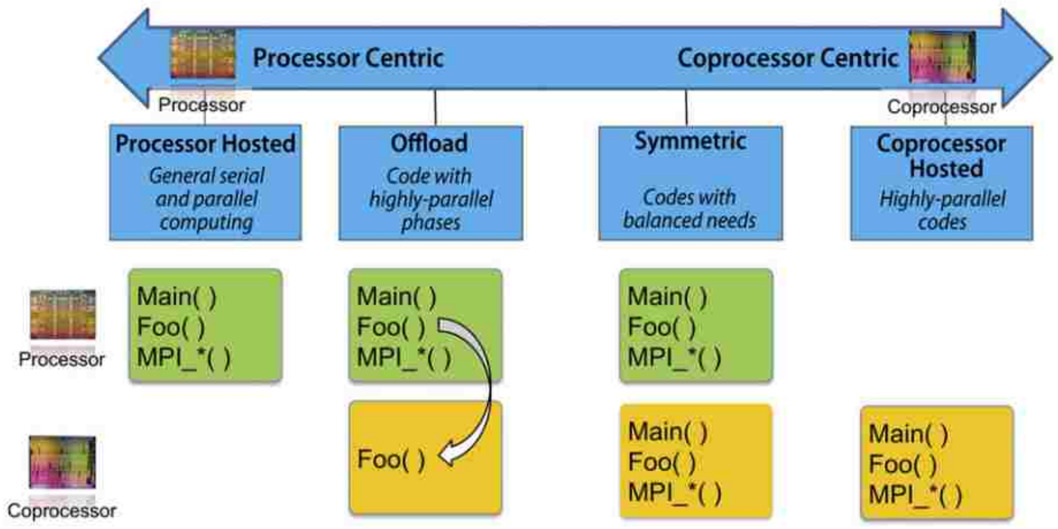


Figure 2.11: Xeon Phi programming models. A code can run on either the CPU (shown in green) or the MIC (shown in yellow) or both [Jeffers and Reinders, 2013].

Table 2.1: Linux Standard Base (LSB) core libraries on the Intel MIC. From [Jeffers and Reinders, 2013].

Component	Description
glibc	the GNU C standard library
libc	the C standard library
libm	the math library
libdl	programmatic interface to the dynamic linking loader
librt	POSIX real-time library
libcrypt	password and data encryption library
libutil	library of utility functions
libstdc++	the GNU C++ standard library
libgcc_s	a low-level runtime library
libz	a lossless data compression library
libcurses	a terminal-independent method of updating character screens
libpam	the Pluggable Authentication Module (PAM) interfaces

communicating with the MIC can be directly completed without routing via the host. Additionally, because there is a Linux OS on the co-processor, various I/O, MPI and other third-party libraries can be compiled, built, and run on the Xeon Phi as well. Therefore, in contrast to the GPU, there is no need to write specialized libraries for the Xeon Phi from scratch.

Since both the Xeon Phi and CPU share the same underlying architecture (x86-64 instruction set), it has been demonstrated that optimizing code for the Intel Xeon Phi can result in more optimized code for the Xeon CPU as well [Calvin et al., 2013, Jeffers and Reinders, 2013]. Consequently, programmers need to not maintain two separate code bases, each with different data structures, algorithms, and optimization paths. Understandably, this is a huge advantage over GPU programming for developers, where they must maintain multiple programs for multiple targets.

2.2.2 Raw performance of the MIC

Although the MIC is equipped with more cores and lengthier vector registers, its performance advantages over the CPU depends strongly on the specific code executing on it. Therefore, prior to any real-world benchmarks, the Intel Xeon Phi should be benchmarked for basic arithmetic operations (*i.e.* additions, multiplications and transcendental functions) in order to assess the processor's capabilities and limitations for the basic and most-utilized mathematical functions ultimately utilized in full-scale applications code such as XFLAT.

The raw performance of the CPU and Xeon Phi was first benchmarked on the Texas Advanced Computing Center (TACC) Stampede supercomputer using a code adapted from [Jeffers and Reinders, 2013], with the structure illustrated in Fig. 2.12. Characteristics of the Stampede machine are shown in Table 1 of [Noormofidi et al., 2015]. In the innermost loop of this code, simple floating point operations (*i.e.* additions and multiplications) were performed on an array or vector of double-precision (DP) floating-point numbers. The widths of the vectors were taken to be a multiple of that of the SIMD registers of the computing component (256 bits or 4 DP for the Xeon CPU, and 512 bits or 8 DP for the Xeon Phi). The same vector operations were repeated 10 million times in the middle loop to maintain data locality. In the outermost loop, all of the hardware threads were utilized to achieve the best performance. The results of these benchmarks with different vector widths in the innermost loop are shown in Fig. 2.13. These results show that the floating point performance of the Xeon Phi is highly sensitive to the width of the vector, while the performance of the CPU is relatively stable. The floating point performance of the Xeon Phi is best when the width of the DP vectors is 64; in that test, the Xeon Phi ran 10 times as fast as the CPU. However, the performance of the Xeon Phi degrades substantially as vector width increases.

```
/// 8/244 OpenMP threads for CPU/MIC.
#pragma omp parallel for
for (t: NUM_THREADS)
{
  /// Repeat 10 million times.
  for (itr: LOOP_COUNT)
  {
    /// VECTOR_WIDTH is a multiple of 4/8
    /// for CPU/MIC.
    #pragma omp simd
    for (s: VECTOR_WIDTH)
    {
      /// floating point operations
      ...
    }
  }
}
```

Figure 2.12: High-level structure of the benchmark code for floating point performance.

Since XFLAT employs transcendental functions such as `sin()` and `exp()`, the transcendental function performance of the CPU and Xeon Phi was benchmarked using a code similar to that of Fig. 2.12, with the simple floating point operations replaced by a pair of `sin()` and `cos()` functions in one series of tests, and `exp()` in the other (see Fig. 2.13). These results suggest that transcendental function performance on the Xeon Phi is relatively stable against the width of the vectors. For the tests on `sin()` and `cos()` the Xeon Phi ran 6–8 times as fast as the CPU, but for `exp()` the Xeon Phi was only 3–4 times faster.

The double-precision (DP) floating-point benchmarks showed that the Xeon Phi can outperform the CPU by ten times only when the vector of double-precision numbers is short. Increasing vector length beyond 64 double-precision numbers causes the Xeon Phi's performance to drops dramatically and become comparable to that

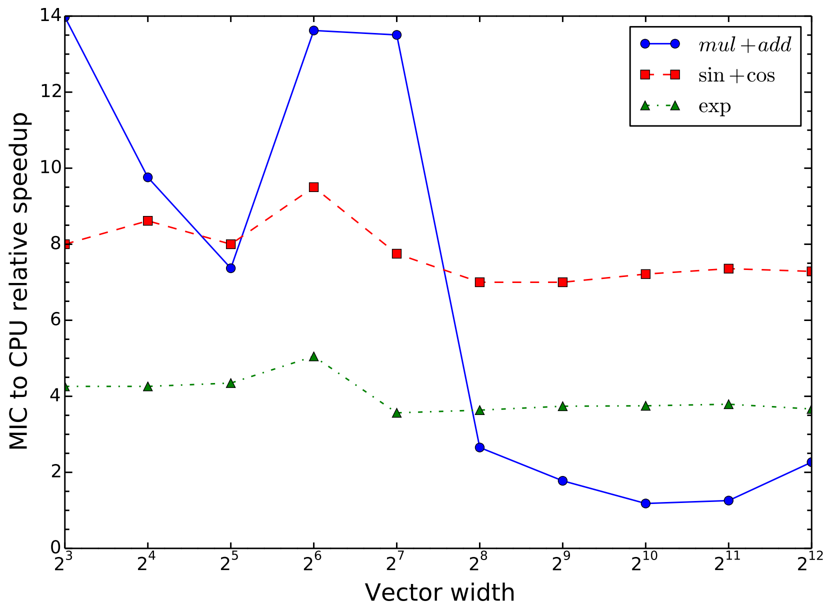


Figure 2.13: The relative speedup of the floating point and transcendental functions for the CPU and Xeon Phi on the TACC Stampede Xeon Phi-based supercomputer (single node only). The width of the vector is the length of the innermost loop (vectorized loop) in the kernel.

of the CPU. From the transcendental functions benchmarks it can be concluded that the performance of transcendental functions on the $\tilde{6}$0-core MIC is a few times better than that of the 8-core CPU. For short vector lengths, the `sin()/cos()` and `exp()` on Xeon Phi ran 8 times and 4 times as fast as the CPU, respectively. However, by increasing the vector length beyond 64 double-precision numbers, the `sin()/cos()` on Xeon Phi ran only 6 times as fast as the CPU, and the `exp()` on 244 threads of the Xeon Phi was slightly more than 3 times as fast as on the CPU's 8 threads.

Based on these results, it is clear that performance of the Xeon Phi will be dependent on application loop structures and the length of vectorized loops. It can vary dramatically with only slight changes in the size of the data. It also directly depends on the type of the employed operations and functions. Therefore, the decision to utilize the Xeon Phi in real world applications depends on the application type and may or may not be economically beneficial.

2.2.3 Previous work on the Intel MIC

Following the introduction of the Xeon Phi in 2012, many researchers started to explore the capabilities of the new co-processor. There has been a considerable amount of research on the Xeon Phi via benchmarking kernels in order to study the advantages and disadvantages of the co-processor. Examples includes LINPACK [Heinecke et al., 2013] and matrix multiplication [Saule et al., 2014]. In addition, real-world applications have been developed or partially rewritten to harness the power of the Xeon Phi for scientific research. Here we provide a brief survey of the most prominent efforts to date, and an overview of typical performance improvements overall compared to CPU-only code.

[Pennycook et al., 2013] [Pe] employed Xeon Phi for molecular dynamics. They employed intrinsic functions, which are mapped by the compiler directly onto the corresponding assembly instruction, as well as the compiler `#pragma` in order to exploit SIMD registers. Their code achieved $1 - 1.4\times$ performance improvement on a Xeon Phi over a single 8-core Xeon E5-2660 CPU.

[Crimi et al., 2013] [Cr] ported a Lattice Boltzmann code on the Xeon Phi. They used the offload programming approach. For vectorization they employed the intrinsic functions. Their code on the Xeon Phi achieved $1.12\times$ higher performance than a 16-core system.

[Leang et al., 2014] [Le] utilized the Xeon Phi for quantum chemical calculations. They compared the Xeon Phi with a system that was equipped with two 8-core Xeon E5-2650. For all Phi runs, the `MIC_KMP_AFFINITY` environment variable for the co-processor was set to `compact`, as this was found to offer better overall performance relative to the default affinity setting of `scatter`. To improve data transfer performance on the Phi, the environment variable `MIC_USE_2MB_BUFFERS=2M` was set to turn on the use of 2 MB page buffers. They achieved $1.5 - 2\times$ improvement over the host.

[Aprà et al., 2014] [Ap] reported an implementation of many-body quantum chemical methods on the Xeon Phi. The auto-vectorization and offload model were chosen for vectorization and design strategies, respectively. Moreover, performance benefit was observed by setting the environment variable `MIC_USE_2MB_BUFFERS=16K`. They reported $2.1\times$ improvement over a Xeon E5-2670 CPU.

[Teodoro et al., 2014] [Te] employed the Intel MIC for microscopy image analysis (digital pathology). Digital pathology involves the analysis of images obtained from whole slide tissue specimens using microscopy image scanners. They employed the offload programming approach and utilized the compiler `#pragma` in order to vectorize their code. Their Xeon Phi implementation achieved $0.4 - 1.9\times$ performance over a system with two 8-core Xeon E5-2680 processors.

[Wende and Steinke, 2013] [We] exploited the Intel MIC in a Swendsen-Wang multi-cluster algorithm for the 2D/3D Ising model (ferromagnetism in statistical mechanics). They utilized the intrinsic functions for vectorizations and the environment variable `KMP_AFFINITY=balanced,granularity=fine` for thread pinning. Speedups up to a factor 3 over a single 8-core Xeon E5-2680 were observed; however, for the smaller problem sizes the performance of the CPU vs. MIC was comparable.

[Liu and Schmidt, 2014] [Li] utilized the Intel MIC for biological sequence database search. Their code was written specifically for the Xeon Phi with a set of SIMD intrinsics. They investigated the offload model in order to coordinate multiple coprocessors to perform sequence alignments. Their Xeon Phi code achieved $1.19 - 2.49\times$ higher performance than implementations on a single 8-core CPU.

[Park et al., 2013] [Pa] developed a code for efficient backprojection-based synthetic aperture radar computation using the Intel MIC. They reported a 1.9× improvement over two 8-core Xeon E5-2670 CPUs.

[Halyo et al., 2014] [Ha] employed the Intel MIC for real time particle tracking based on the Hough transform at the Large Hadron Collider (LHC). In order to utilize the co-processor, data was moved from the CPU to the co-processor using the offload functionality. In addition, the auto-vectorization functionality of the compiler was employed for utilizing the SIMD units. The reported Xeon Phi performance was 0.3× that of two 12-core E5-2697v2 CPUs (Xeon Phi was three times slower).

[Knoll et al., 2013] [Kn] exploited the Xeon Phi for an application involving ray tracing and volume rendering of large molecular data. They employed SPMD language as a special language generating vectorized code. They reported about 2× performance improvement on the Xeon Phi over a single 8-core Xeon E5-2680 CPU.

[Kulikov et al., 2015] [Ku] developed a new astrophysics code specifically for the Xeon Phi and they reported over 10× improvement over a single-core of an E5-2690 CPU (around 1.2× improvement over an 8-core CPU).

The implementation methods used in the above applications are summarized in Table 2.2.

All of the reported speedups can be normalized to a reference hardware. Since several works (including the current work) employed the Stampede supercomputer, all the hardware were normalized to the Stampede's MIC and CPU:

$$\frac{\frac{MIC\#\ of\ cores}{Stampede's\ MIC\#\ of\ cores} \times \frac{MIC's\ freq.}{Stampede's\ MIC's\ freq.}}{\frac{CPU\#\ of\ cores}{Stampede's\ CPU\#\ of\ cores} \times \frac{CPU's\ freq.}{Stampede's\ CPU's\ freq.}} \tag{2.1}$$

Fig. 2.14, depicts the relative speedup of each work normalized to Stampede's hardware. As seen from these results, the expected speedup for a full-fledged appli-

Table 2.2: Summary of implementation approaches on the Intel MIC for real-world applications. The – indicates that the approach was not discussed in the paper. For the Multi-node column, N/A means that the code was run on a single node; no multi-node support.

Reference	Exec. mode	Multi-node	Multi-thread	SIMD
[Pennycook et al., 2013]	–	MPI	–	Intrinsic
[Crimi et al., 2013]	Offload	N/A	–	Intrinsic
[Leang et al., 2014]	Offload	N/A	OpenMP	–
[Aprà et al., 2014]	Offload	–	OpenMP	Auto-vec.
[Teodoro et al., 2014]	Offload	–	OpenMP	Auto-vec.
[Wende and Steinke, 2013]	Native	MPI	OpenMP	Intrinsic
[Liu and Schmidt, 2014]	Offload	–	OpenMP	Intrinsic
[Park et al., 2013]	Offload	MPI	OpenMP	Intrinsic
[Halyo et al., 2014]	Offload	N/A	OpenMP	Intrinsic
[Knoll et al., 2013]	–	MPI	OpenMP	SPMD
[Kulikov et al., 2015]	Offload	MPI	OpenMP	–

cation is at most around four times on a lower-end CPU and by increasing the CPU power it decreases and to be around $1 - 1.5\times$ for the current generation of the Xeon Phi. XFLAT could achieve up to three times speedup on the Stampede's MICs and up to four times speed up on Bahcall machine. Note: the specifications for Bahcall can be found in Table5.2 and Table5.3.

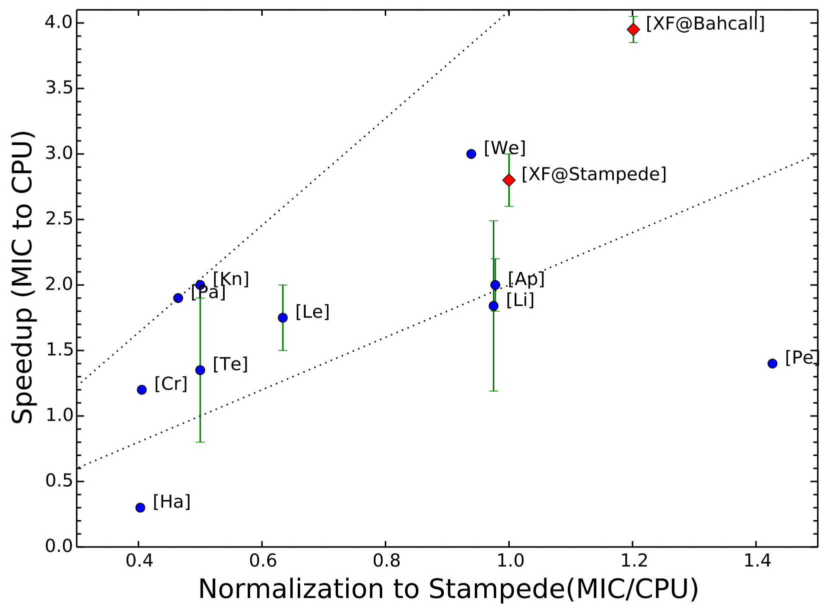


Figure 2.14: Illustration of the relative MIC to CPU speedup of different applications. The x-axis is the $MIC's\ cores \times MIC's\ freq.\ /\ CPU's\ cores \times CPU's\ freq.$ normalized to Stampede's hardware.

Chapter 3

Neutrino Oscillations

3.1 Physics of Neutrino Oscillations in Supernovae

As discussed earlier, neutrino oscillations in which neutrinos can change their flavor state are a quantum mechanical phenomenon [Barger et al., 2012]. A neutrino created with a specific flavor (ν_e, ν_μ or ν_τ) can later change state to a different flavor. In vacuum neutrino oscillations the probability of measuring a particular flavor for a neutrino varies periodically as the neutrino propagates through space. The existence of neutrino oscillations implies that the neutrino has a non-zero mass, which was not included as part of the original Standard Model of particle physics [Barger et al., 2012].

Experimental evidence for neutrino oscillations comes from various sources: solar neutrino oscillations [Davis et al., 1968] (in which neutrinos produced in the Sun are studied); atmospheric neutrino oscillations [Fukuda et al., 1998] (when a cosmic ray hits a nucleus in the upper atmosphere, neutrinos may be created during the process); reactor neutrino oscillations [An et al., 2012] (neutrinos may be created in a nuclear reactor's core), and beam neutrino oscillations [Agafonova et al., 2010] (neutrinos may be created in particle accelerators).

In a supernova, it has been shown that, because there are so many neutrinos emitted from the neutron star within such a short time, flavor oscillations of neutrinos with different energies that are propagating along different trajectories are not independent of each other but are correlated [Duan et al., 2010]. As illustrated in Fig. 3.1, following the flavor evolution of neutrino ν_p would entail knowing the flavor states of all neutrinos on which it forward scatters (only the trajectories of neutrinos along the escaping directions are included), such as ν_q and ν_k propagating on trajectories which intersect ν_p's trajectory at points P and Q, respectively. Note that the flavor histories of ν_q and ν_k are not independent, as they have undergone a forward scattering at point K in the past. This phenomenon is known as *collective neutrino oscillation*. Since it is impossible to solve the supernova neutrino oscillation equations analytically, and it is very challenging to follow collective oscillations of supernova neutrinos numerically, the only model in which collective neutrino oscillations have been solved numerically so far is the *neutrino bulb model* in which the physical conditions are assumed to be exactly spherically symmetric [Duan et al., 2006].

Even in this seemingly simplistic model of neutrino oscillation, as calculations of the oscillations of the neutrinos of different energies and emission angles are highly correlated, millions of coupled, nonlinear differential equations must be solved simultaneously. Hence, with a huge number of neutrinos emitting from the surface, tracing the neutrino-neutrino evolution history quickly become an intractable simulation problem.

Not only does solving each equation requires many computationally expensive functions (*i.e.* sin, cos, and exp), the entire model still only calculates the neutrino-neutrino forward scattering interactions. Thus, it does not even include the interaction of neutrinos that scatter along the backward direction. Since for each step

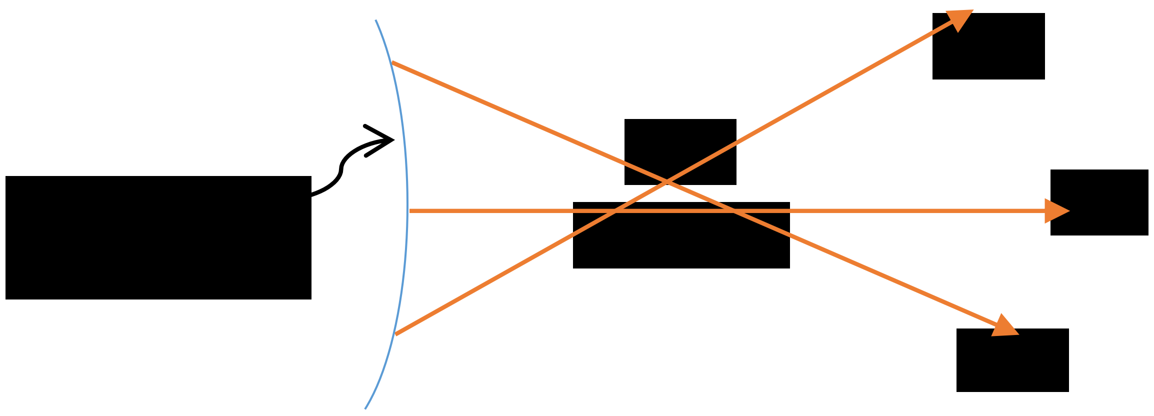


Figure 3.1: Illustration of nonlinearity problem for intersecting neutrino trajectories emerging from a proto-neutron star (a star which is cooling and contracting to become a neutron star). Points P, Q, and K are intersections of the world lines for three neutrino beams, ν_p, ν_q, and ν_k. Hence, the flavor evolution histories of these neutrinos will be quantum-mechanically coupled.

and each particle's beam many quantum mechanical equations have to be solved, and there might be millions of steps and millions of neutrino beams in the system, the entire simulation requires vast computational power which is currently available only on very high-end supercomputers.

As discussed in the Introduction, one of the most common ways to simplify the neutirno oscillation simulation problem is to reduce its dimensionality by studying simpler models such as the spherically symmetric neutrino bulb model. The assumed spherical symmetry in the bulb model reduces the required calculations dramatically, as neutrino trajectories with the same emission angle become equivalent. Thus, neutrinos emitted in the same flavor and energy state and with the same emission angle have identical flavor evolution history [Duan et al., 2006].

As mentioned earlier there are three kinds of neutrinos—the ν_e, ν_μ, and ν_τ—in the particle physics Standard Model. Each particle has a corresponding anti-particle ($\bar{\nu}_e$, $\bar{\nu}_\mu$, $\bar{\nu}_\tau$). The fluxes and energy spectra may be different for each neutrino and anti-neutrino. As a result, the flavor content of the neutrino field above the proto-neutron star surface and its evolution history in time and space can be important, since many environments associated with compact objects and the very early universe are dominated by neutrinos and their interactions at some point. The objective of nonlinear neutrino oscillation simulations in supernovae is to study and understand the flavor evolution of the neutrino field when ν_e and $\bar{\nu}_e$ mix with neutrinos and anti-neutrinos of other active flavors. Hence, the quantum state of the system must be studied. The discussion of neutrino physics that follows is closely based on [Duan et al., 2006] and [Duan et al., 2008].

The quantum state of a particle can be described by the *wavefunction*. The wavefunction of the flavor of a neutrino in the two-flavor system is written as:

$$\psi_\nu \equiv \begin{pmatrix} a \\ b \end{pmatrix}, \tag{3.1}$$

where a, and b are the complex amplitudes for neutrino to be in the ν_e, $\bar{\nu}_e$, and ν_x ($\bar{\nu}_x$) (x represents either the μ or τ neutrino or a linear combination of these) flavor states, respectively.

The development in time and space of the complex amplitudes is followed in order to describe the flavor states of the neutrinos, and the full quantum kinetic equations reduce to a Schrödinger equation. The Schrödinger equation determines how the wave function evolves over time; i.e., the wavefunction is the solution of the Schrödinger equation.

For a given neutrino trajectory, with parameter l, which is the propagation length of a neutrino, the Schrödinger equation can be written as:

$$\mathrm{i}\frac{\mathrm{d}}{\mathrm{d}l}\psi_\alpha\left(\hat{\mathbf{v}}, E; l\right) = \left(\mathsf{H}_0 + \mathsf{H}_m + \mathsf{H}_{\nu\nu}\right) \cdot \psi_\alpha\left(\hat{\mathbf{v}}, E; l\right), \tag{3.2}$$

where α, $\hat{\mathbf{v}}$ and E are the initial flavor, (the unit vector of) the propagation direction, and the energy of the neutrino, respectively. H_0 is the Hamiltonian in vacuum, H_m is

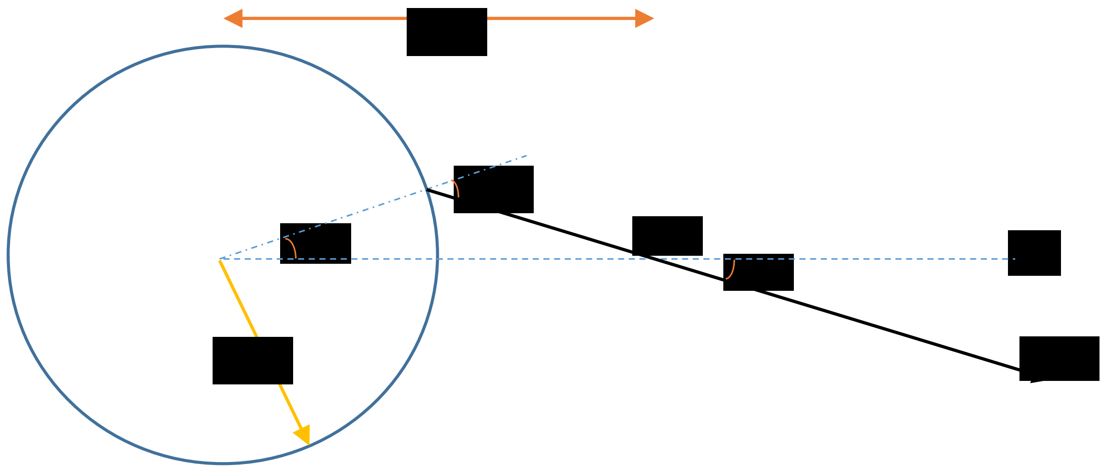


Figure 3.2: Geometry of the neutrino bulb model. The neutrino beam ν_p is emitted from a point on the proto-neutron sphere (radius R), with polar angle Θ. The beam intersects the z-axis at point P with angle ϑ. All geometric properties can be calculated from the radius r (the distance from the center of the proto-neutron star) and the emission angle ϑ_0, which is defined with respect to the normal direction at the point of emission on the star ($\vartheta_0 = \Theta + \vartheta$).

the matter potential, and $\mathsf{H}_{\nu\nu}$ is the neutrino potential due to the ambient neutrinos. The propagation direction $\hat{\mathbf{v}}$ is fully described by the polar angle ϑ between $\hat{\mathbf{v}}$ and the radial direction when axial symmetry about the radial direction is imposed. In the neutrino bulb model $\hat{\mathbf{v}}$ is determined by ϑ. As depicted in Fig. 3.2, at any given radius r all the geometric properties of a neutrino beam can be calculated using only r and ϑ_0. Hence, ϑ and Θ are related to ϑ_0 through:

$$\frac{\sin\vartheta}{R_\nu} \equiv \frac{\sin\Theta}{l - l_0} \equiv \frac{\sin\vartheta_0}{r}, \tag{3.3}$$

where $l \equiv r\cos\vartheta$ and $l_0 \equiv R_\nu \cos\vartheta_0$.

However, in other models such as the *extended bulb model* [Mirizzi, 2013] where axial symmetry is not imposed, $\hat{\mathbf{v}}$ is determined by both ϑ and the azimuthal angle (φ) about the radial axis. Thus, $\hat{\mathbf{v}}$ is directly related to the geometry of the model. For n neutrino flavors, ψ is a vector of n complex variables, and H_0 and $\mathsf{H}_{\nu\nu}$ are both $n \times n$ Hermitian matrices. Since the neutrino potential $\mathsf{H}_{\nu\nu}$ governing ambient neutrino forward-scattering contributes to the Hamiltonian, a serious complication arises and renders the problem nonlinear, as the interactions which dictate flavor transformation amplitudes are themselves dependent on the neutrino flavor states. Neutrino modes are binned in each flavor by their energy and emission angle.

As a result, the flavor evolution of the neutrino wavefunction can be represented as:

$$\begin{aligned} \mathrm{i}\frac{\mathrm{d}}{\mathrm{d}l}\psi_\alpha(\hat{\mathbf{v}}, E; l) &= (\mathsf{H}_0 + \mathsf{H}_m + \mathsf{H}_{\nu\nu}) \cdot \psi_\alpha(\hat{\mathbf{v}}, E; l) \\ &= \frac{1}{2}\begin{pmatrix} -\Delta\cos 2\theta + A + B & \Delta\sin 2\theta + B_{e\tau} \\ \Delta\sin 2\theta + B_{e\tau}^* & \Delta\cos 2\theta - A - B \end{pmatrix} \psi_\alpha(\hat{\mathbf{v}}, E; l), \end{aligned} \tag{3.4}$$

where θ is the vacuum mixing angle, and Δ, A, and $B_{e\tau}$ are the vacuum potentials induced by neutrino mass difference, the matter, and the background neutrinos, respectively. The appropriate Hamiltonian for anti-neutrinos is obtained by making the transformation:

$$A \to -A, \quad B \to -B, \quad B_{e\tau} \to -B_{e\tau}^*$$

In Eq. 3.4, the H_0 is the Hamiltonian related to vacuum oscillations that causes the neutrinos change their flavors by propagating through vacuum. The vacuum oscillation term is:

$$\mathrm{i}\frac{\mathrm{d}}{\mathrm{d}l}\psi_\alpha(\hat{\mathbf{v}}, E; l) = \frac{1}{2}\begin{pmatrix} -\Delta\cos 2\theta & \Delta\sin 2\theta \\ \Delta\sin 2\theta & \Delta\cos 2\theta \end{pmatrix} \psi_\alpha(\hat{\mathbf{v}}, E; l), \tag{3.5}$$

where Δ is the vacuum oscillation frequency and is defined as:

$$\Delta \equiv \frac{\delta m^2}{2E_\nu}, \tag{3.6}$$

where δm^2 is the difference between mass-squared of neutrinos, and E_ν is the energy of the neutrino. Because the sign of one of the neutrino mass-squared difference

(δm^2_{23}) has yet to be measured, numerical simulations in the two-flavor mixing scheme are run with mass-squared difference δm^2 with both the positive and negative values.

Electrons interact with neutrinos via the weak nuclear force. As a result, the matter density above the neutron star plays an important role as well, since neutrinos interact with matter around the neutron star. The matter potential (H_m) in Eq. 3.4 is defined as:

$$\mathsf{H}_m = \frac{1}{2}\begin{pmatrix} A & 0 \\ 0 & -A \end{pmatrix} \tag{3.7}$$

where $A = \sqrt{2}G_F n_e$. G_F is the Fermi coupling constant, n_e is the net electron number density that is related to baryon density n_b with $n_e = Y_e n_b$, where Y_e is the electron fraction (electron to baryon ratio).

The difficulty in solving Eq. 3.4 stems from the neutrino potential $\mathsf{H}_{\nu\nu}$ which is given by:

$$\mathsf{H}_{\nu\nu} = \sum_{\alpha'} \int \mathrm{d}E' \int \mathrm{d}\hat{\mathbf{v}}' \; (1 - \hat{\mathbf{v}} \cdot \hat{\mathbf{v}}') \left[\varrho'_{\nu_\alpha} f_{\nu_\alpha}(E') \frac{L_{\nu_\alpha}}{\langle E_{\nu_\alpha} \rangle} - \varrho'^{*}_{\bar{\nu}_\alpha} f_{\bar{\nu}_\alpha}(E') \frac{L_{\bar{\nu}_\alpha}}{\langle E_{\bar{\nu}_\alpha} \rangle} \right], \tag{3.8}$$

where the quantities with primes are associated with the ambient neutrinos, and the quantities with bars are associated with anti-neutrinos. L_{ν_α}, $\langle E_{\nu_\alpha} \rangle$, and $f_{\nu_\alpha}(E')$ are the energy luminosity, average energy and normalized energy distribution function of ν_α, respectively.

ϱ' in Eq. 3.8 is called the *density matrix* and can be calculated directly from $\psi_{\alpha'}(\hat{\mathbf{v}}', E'; l)$ as follow:

$$\varrho \equiv \begin{pmatrix} |a|^2 - |b|^2 & 2ab^* \\ 2a^*b & -|a|^2 + |b| \end{pmatrix}, \tag{3.9}$$

where a and b are the components of the wavefunction (ψ_α) in Eq. 3.1.

Eq. 3.4 shows that all the neutrino beams propagating along different trajectories and having different energies are coupled. Therefore, it makes Eq.3.4 nonlinear. For a small distance, Eq. 3.4 can be solved analytically as:

$$\psi_\nu\left(l+\delta l\right) \simeq \exp\left(-i\mathsf{H}\delta l\right)\psi_\nu\left(l\right) \tag{3.10}$$

$$= \frac{1}{\lambda}\begin{pmatrix} \lambda\cos(\lambda\delta l) - ih_{11}\sin(\lambda\delta l) & -ih_{12}\sin(\lambda\delta l) \\ -ih_{12}^{*}\sin(\lambda\delta l) & \lambda\cos(\lambda\delta l) + ih_{11}\sin(\lambda\delta l) \end{pmatrix}\psi_\nu\left(l\right), \tag{3.11}$$

where h_{11} and h_{12} are the diagonal and off-diagonal elements of the Hamiltonian H, and λ is defined as:

$$\lambda \equiv \sqrt{h_{11}^2 + \left|h_{12}\right|^2}. \tag{3.12}$$

3.2 Implemented Physics Models

XFLAT is designed to support a variety of geometries; the “neutrino bulb model” is just one of them. In this section, we describe the physics models currently implemented in XFLAT: the single-angle bulb model; extended bulb model; plane/point model; and cylinder/line model.

3.2.1 Bulb Model

In the neutrino bulb model, there are multiple neutrino trajectories along different zenith directions (the ϑ angles). The emission from each point on the star’s surface is assumed to be identical; consequently the study of a single emission point is sufficient. The bulb model assumes azimuthal symmetry, meaning that all trajectories along a particular zenith angle but along different azimuthal directions (the φ angles) are assumed to be identical. Fig. 3.3 illustrates the neutrino bulb model. There are different trajectories along the zenith direction emitted from a given point on the surface, and every point on the surface is assumed to be equivalent.

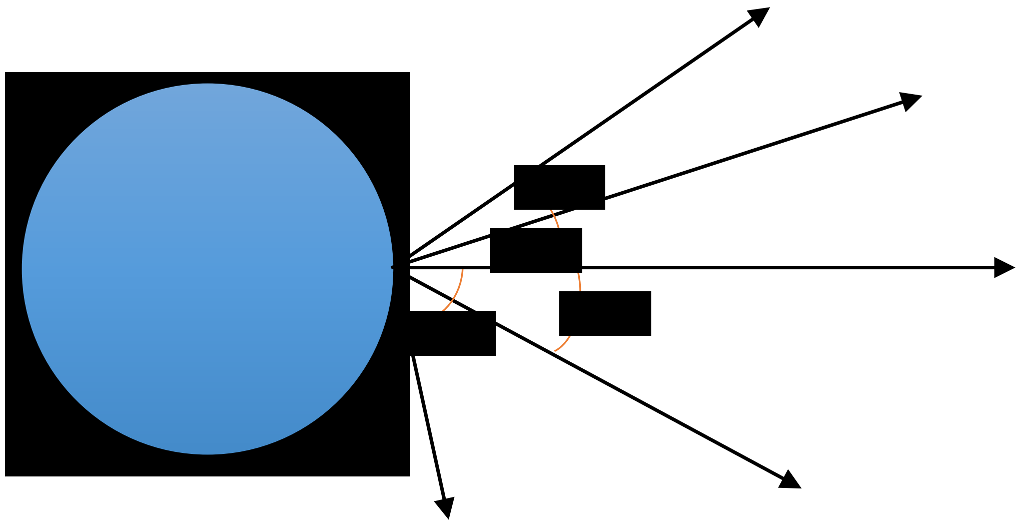


Figure 3.3: Illustration of multi-zenith angle model. There are different trajectories along zenith direction (ϑ_1, ϑ_2, ϑ_3, ϑ_4) emitted from a single point on the star's surface (shown in blue), and every point on the surface is assumed to be equivalent.

Eq. 3.8 can be rewritten and simplified for the bulb model to read:

$$\mathsf{H}_{\nu\nu} = \sum_{\alpha'} \int \mathrm{d}E' \int (1 - \cos\vartheta \cos\vartheta') \left[\varrho'_{\nu_\alpha} f_{\nu_\alpha}(E') \frac{L_{\nu_\alpha}}{\langle E_{\nu_\alpha} \rangle} - \varrho'^{*}_{\bar{\nu}_\alpha} f_{\bar{\nu}_\alpha}(E') \frac{L_{\bar{\nu}_\alpha}}{\langle E_{\bar{\nu}_\alpha} \rangle} \right] \mathrm{d}\left(\cos\vartheta'\right), \tag{3.13}$$

where ϑ is the polar (zenith) angle of the neutrino of interest and ϑ' is the polar (zenith) angle of the background neutrino. At each radial point along trajectories, the $\cos\vartheta'$ can be calculated as:

$$\cos\vartheta' = \sqrt{1 - \left(\frac{R}{r}\right)^2 (1 - \cos^2\vartheta'_0)}, \tag{3.14}$$

where R is the proto-neutron star's radius, r is the distance from the center of the proto-neutron star to the current point of interest and ϑ'_0 are the initial angles at the surface (see Fig. 3.2) for which the angle bins (which are the result of discretizing the zenith angle range) are calculated. From Eq. 3.14, $d\cos\vartheta'$ can be computed as:

$$d\cos\vartheta' = \frac{1}{2}\left(\frac{R}{r}\right)^2 \frac{d\cos^2\vartheta'_0}{\cos\vartheta'}, \tag{3.15}$$

where the ϑ_0 angles only need to be calculated once at the proto-neutron star's surface.

3.2.2 Single-angle Model

This model is a simplified version of the bulb model. In fact, it is the simplest geometry implemented in XFLAT. Similar to the bulb model, every point on the surface of the proto-neutron star emits neutrinos identically, however, there is a single identical emitting angle for all points on the surface. Therefore, the required amount of computation is several orders of magnitude less than for the bulb model since there are no multi-zenith angle trajectories needed. As a result, Eq. 3.13 can be simplified further to read:

$$\mathsf{H}_{\nu\nu} = D(r/R_\nu)\sum_{\alpha'}\int\left[\varrho'_{\nu_\alpha} f_{\nu_\alpha}(E')\frac{L_{\nu_\alpha}}{\langle E_{\nu_\alpha}\rangle} - \varrho'^{*}_{\bar{\nu}_\alpha} f_{\bar{\nu}_\alpha}(E')\frac{L_{\bar{\nu}_\alpha}}{\langle E_{\bar{\nu}_\alpha}\rangle}\right]\mathrm{d}E', \tag{3.16}$$

where r is the current radius, R_ν is the proto-neutron star's radius, and the geometric factor $D(r/R_\nu)$ is defined as:

$$D(r/R_\nu) \equiv \frac{1}{2}\left[1-\sqrt{1-\left(\frac{R_\nu}{r}\right)^2}\right]^2. \tag{3.17}$$

Due to the simplicity of this model and the presence of just a single trajectory, only the vector registers within a single core can be employed for computations. Therefore, multi-core and multi-node execution is not supported for this model in XFLAT, and the Xeon Phi cannot be used to accelerate the computations.

3.2.3 Extended Bulb Model

The extended bulb model is essentially the same as the neutrino bulb model with the inclusion of azimuthal trajectories (the φ angles). Fig 3.4 illustrates the extended bulb model including the azimuthal angles. For each zenithal angle (ϑ), there is a cone of azimuth angle trajectories. r denotes the distance from the center of the proto-neutron star to the current calculation point, and R is the neutron star's radius. While the resulting model is not truly a representation of neutrino supernova physics (breaking the azimuthal symmetry by adding beams along the azimuth direction causes breaking of the spherical symmetry), the multi-azimuth model can nevertheless be a useful tool for studying neutrino behavior in order to predict a neutrino's evolution in more complicated environments.

For the multi-azimuth model, Eq. 3.8 can be rewritten as:

$$\mathsf{H}_{\nu\nu} = \sum_{\alpha'} \int \mathrm{d}E' \int (1 - (\cos\vartheta \cos\vartheta' + \sin\vartheta \sin\vartheta' \cos(\varphi - \varphi'))) \left[\varrho'_{\nu_\alpha} f_{\nu_\alpha}(E') \frac{L_{\nu_\alpha}}{\langle E_{\nu_\alpha} \rangle} - \varrho'^{*}_{\bar{\nu}_\alpha} f_{\bar{\nu}_\alpha}(E') \frac{L_{\bar{\nu}_\alpha}}{\langle E_{\bar{\nu}_\alpha} \rangle} \right] \mathrm{d}\varphi' \mathrm{d}\left(\cos\vartheta'\right). \tag{3.18}$$

This model requires orders of magnitude greater computational effort than the simple bulb model, since for each zenith angle trajectory, there may be hundreds to thousands of azimuth angles. As a result, employing multi-node supercomputers is necessary and the rise of accelerators becomes important in order to achieve higher on-node performance. Multi-node distributed-memory supercomputers are essential since memory requirements are at least two order of magnitudes greater for this model.

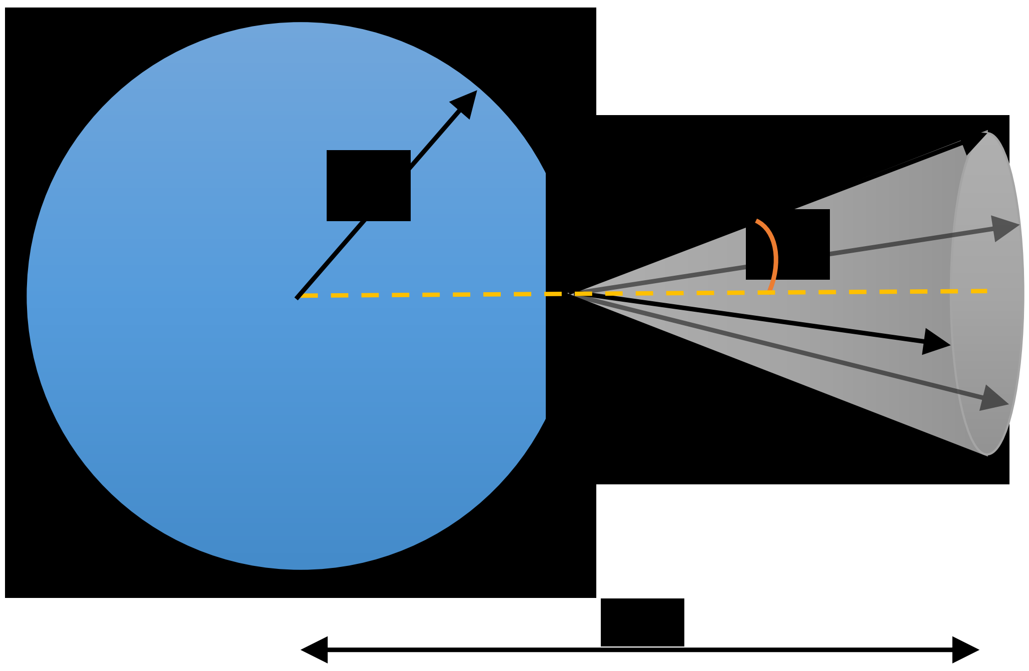


Figure 3.4: Illustration of the extended bulb model including the azimuthal angles. For each zenith angle (ϑ), there is a cone (shown in grey) of azimuth angle trajectories. The variable r is the distance from the center of the proto-neutron star (shown in blue) to the current calculation point; R is the neutron star's radius.

3.2.4 Plane/Point Model

One of the geometries used to study neutrino oscillations is the plane geometry. Recent progress in instability calculations for neutrino oscillations [Mirizzi, 2013] has shown that if the azimuthal asymmetries are taken into account, in linearized stability calculations, a new kind of instability, the multi-azimuth angle instability, appears. This occurs at smaller radii. Since adding the azimuth angles to the bulb model breaks the spherical symmetry, this model tries to restore the broken symmetry in order to make the model self-consistent. As a result, all the neutrino emitting points are assumed to be on a plane surface, therefore there is no breaking in the symmetry anymore. As depicted in Fig. 3.5, all points on the plane surface are assumed to be the equivalent, hence a particular neutrino trajectory emitting along the $(\vartheta,\ \varphi)$ direction from point p_0 would have the identical properties and history as another neutrino trajectory emitting along the $(\vartheta,\ \varphi)$ direction from another point p_1. For this reason, all neutrino trajectories alongside the same direction $(\vartheta,\ \varphi)$, and emitted from different points on the plane surface, would have precisely the equivalent evolution history.

Hence, Eq. 3.8 can be recast as:

$$\begin{aligned}\mathsf{H}_{\nu\nu} =& \sum_{\alpha'} \int \mathrm{d}E' \int (1 - (\cos\vartheta \cos\vartheta' + \sin\vartheta \sin\vartheta' \cos(\varphi - \varphi')))\\ & \left[\varrho'_{\nu_\alpha} f_{\nu_\alpha}(E') j_{\nu_\alpha}(\varphi', E') - \varrho'^{*}_{\bar\nu_\alpha} f_{\bar\nu_\alpha}(E') j^{*}_{\bar\nu_\alpha}(\varphi', E')\right] \mathrm{d}\varphi' \mathrm{d}\left(\cos\vartheta'\right), \qquad (3.19)\end{aligned}$$

where $j_{\nu_\alpha}(\varphi', E')$ is the number flux of ν_α with energy E' emitted in the φ' direction.

3.2.5 Cylinder/Line Model

It is of interest to understand the behavior of neutrinos when there are multiple emitting points from a surface. However, there is still no complete model available for the multiple emitting point simulation of supernovae. As an alternative to the neutrino bulb model, in the cylinder model an infinitely long cylinder emits neutrinos at multiple points. To make the model more straightforward, neutrino trajectories are all perpendicular to the axis of the cylinder. As illustrated in Fig. 3.6, neutrinos that are emitted from points p_1, p_2, p_3, and p_4 are different. However, the points that located along the z axis are assumed to all be identical. Therefore, neutrino oscillation instability comes from different points on a slice of the cylinder, emitting different neutrino beams at the same time.

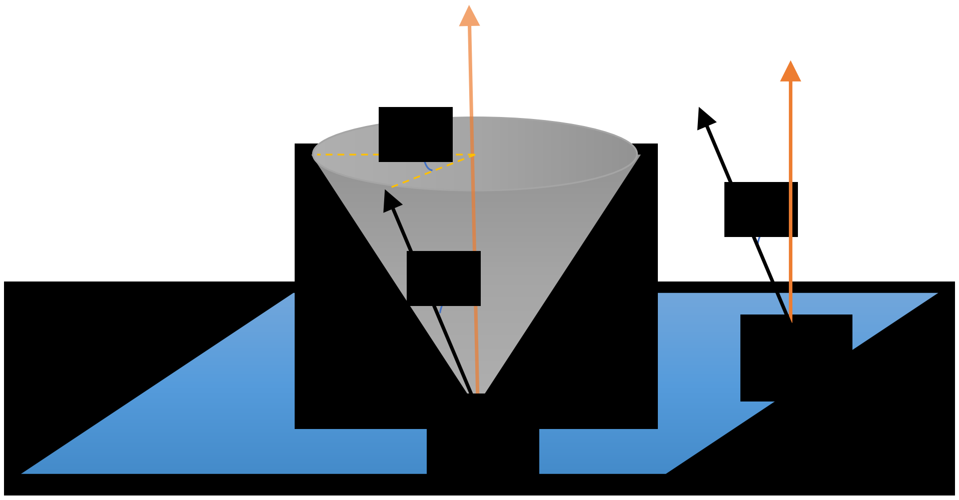


Figure 3.5: Illustration of the plane geometry. The point p_0 and p_1 have identical neutrino emission. For every zenith angle ϑ, there are multiple neutrino trajectories along different φ angles (shown as the grey cone).

In this model, Eq. 3.8 is recast as:

$$H_{\Phi\vartheta} = \int \mathrm{d}E' \int_{-\pi/2}^{\pi/2} (\varrho'_{\nu_\alpha}(\Phi', \vartheta') - \varrho'^{*}_{\bar{\nu}_\alpha}(\Phi', \vartheta')) \frac{\cos\vartheta'}{\sqrt{1 - \left(\frac{R_\nu}{r}\right)^2 (1 - \cos^2\vartheta')}}$$
$$\left[1 - \cos\left(\arcsin\left(\left(\frac{R_\nu}{r}\right)\sin\vartheta\right) - \arcsin\left(\left(\frac{R_\nu}{r}\right)\sin\vartheta'\right)\right)\right]\mathrm{d}\vartheta', \qquad (3.20)$$

where Φ is the latitude angle of the neutrino of interest and Φ' is the latitude angle of the background neutrino, and $\Phi' = \Phi + \vartheta' - \vartheta + \arcsin\left(\left(\frac{R_\nu}{r}\right)\sin\vartheta\right) - \arcsin\left(\left(\frac{R_\nu}{r}\right)\sin\vartheta'\right)$.

Complex as they are, even these models are still far from a complete supernova model. The goal of XFLAT is to enable the simulation of significantly more complex physical models for the neutrino oscillation problem by harnessing the power of modern heterogeneous supercomputers, including current, state-of-the-art microprocessors and next-generation many-core architectures. In the following chapters,

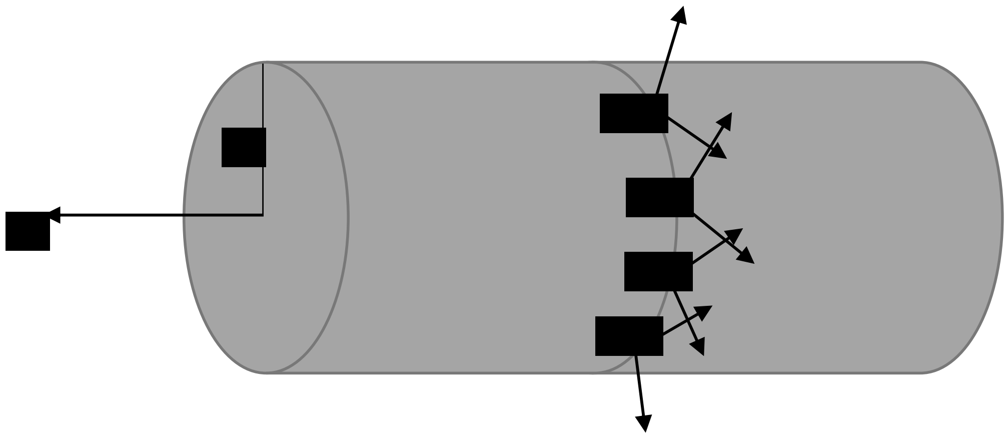


Figure 3.6: Illustration of the cylinder model and its emitting points' trajectories. Neutrinos emitting from points p_1, p_2, p_3, and p_4 are different; however, since azimuthal symmetry is assumed, points located along the z axis are all identical. r is the radius of the cylinder.

we describe the design and implementation of XFLAT, and its validation for some of the current generation of physics models that have been described here.

Chapter 4

Design, Implementation, and Validation of the XFLAT code

4.1 Numerical Implementation

It has been demonstrated that the neutrino oscillations in the bulb model can be modeled and computed numerically using the FLAT code previously developed by [Duan et al., 2008]. In FLAT the quantum flavor states of neutrinos, $\psi(\vartheta, E; r)$, at a given radius r are described by a multi-dimensional array `psi[theta, E]`. At each radius r, a summation over all the elements of this array is performed in order to obtain the neutrino potential $\mathsf{H}_{\nu\nu}$ (Eq. 3.13). A numerical algorithm similar to the midpoint method is employed to solve the Schrödinger equation (Eq. 3.2) and to evolve $\psi(\vartheta, E; r)$ in successive radial steps. In a typical run, $100 - 1,000$ discrete energy bins may be required in order to achieve the desired resolution for the energy spectral function. In addition, over $1,000$ polar angle (ϑ) bins may be required to achieve numerical convergence. In other words, millions of nonlinear differential equations (see Eq. 3.2) must be solved simultaneously in order to compute neutrinos flavor evolution in the bulb model. As discussed in the previous chapter, realistic models, in which the geometry is even more complex, can result in significantly larger computational problem sizes. For example, the inclusion of the azimuth (φ) dimension in the extended bulb model increases the problem size by at least a couple of orders of magnitude, depending on the number of φ beams. As a result, in order to accelerate computations and explore new physics models, the use of modern supercomputers is essential.

One of the most computationally-demanding parts of the model is the calculation of the $\mathsf{H}_{\nu\nu}$. In order to accelerate the computations, it is desirable that those calculations run on more than one compute node. At the same time, the number of nodes should be kept to a minimum in order to minimize the overhead of inter-node communication. Furthermore, the larger the problem size (for more complex supernova models), the more computing power is required. A balance between these two factors can be achieved by employing accelerators or co-processors such as Graphics Processing Units (GPU) or the Xeon Phi (Intel MIC). In principle the computational power of accelerators is higher than that of traditional CPUs.

As discussed previously, in order to maintain the same code for both accelerators and CPUs, the Intel Xeon Phi co-processor (which is based on the Intel x86 architecture) was chosen for XFLAT framework development.

XFLAT is written in C++ and contains about ten thousand lines of code. It employs an algorithm similar to FLAT for solving the Schrödinger equations. However, unlike FLAT, which exploited only MPI, XFLAT implements and utilizes three levels of parallelism. In Fig. 4.1 the high level code structure of XFLAT is illustrated for the extended neutrino bulb model. At the top level, the ϑ angle bins are distributed among compute nodes (either a CPU or a Xeon Phi), and inter-node communication is handled via MPI. At the middle level (*i.e.*, on a CPU or Xeon Phi) ϑ angle bins are further dispatched to processor cores via OpenMP. At the bottom level (within

```
/// Each node receives array of neutrino beams
init_NBeam(psi[]);
/// The main neutrino evolution loop
while (!termination_conditions)
{ ...
  /// Partition local polar bins among OpenMP threads
  #pragma omp parallel for
  for (theta: POLAR_ANGLE_COUNT)
  { ...
    for (phi: AZIMUTH_ANGLE_COUNT)
    { ...
      /// Distribute energy bins across vector registers (SIMD)
      #pragma omp simd
      for (E: ENERGY_BINS)
      {
        /// wavefunction computations of neutrino beams(theta,phi,E)
        psi[theta,phi,E].calc(...);
      }
    }
  }
  /// Inter-node MPI communication; nodes exchange partial results
  MPI_AllReduce(...);
}
```

Figure 4.1: High-level structure of XFLAT parallelism for the extended bulb model. The outer loop is over the polar angles (ϑ); the next (middle) loop is over the azimuth angles (φ); the innermost loop is over energy bins.

a thread), the energy bins' loops are executed via the Single Instruction with Multiple Data (SIMD) paradigm. SIMD units cannot directly compute on C++ complex numbers. Therefore, in order to use the SIMD units efficiently, double-precision floating-point arrays (*i.e.*, `ar[E]`, `ai[E]`, *etc.*) are used to represent the real and imaginary components of the variables in complex vectors $\psi(\vartheta, \varphi, E; r)$ for a single φ and ϑ, and at a given r. These arrays are then grouped into an object element of neutrino beam array `NBeam[theta,phi]` for a given φ and ϑ.

4.2 Architecture

As discussed in Chapter 2, there are two methods for executing applications on the Xeon Phi: offload mode and symmetric mode. Offload mode entails several compromises. For instance, programs have to use one less core, since the idle core is responsible for code and data transfers to/from the MIC [Jeffers and Reinders, 2013]. Thus, there is always less processing power available in offload mode. Furthermore, offload mode is not suitable for very short tasks, as the overhead of data transfers as well as thread creation may suppress computational segment performance. On the other hand, in symmetric mode, there is no code or data transfer during run time except for inter-node MPI communication. Therefore, all of the hardware resources such as cores and hardware threads are available for computation. In addition, in symmetric mode the CPU does not have to be attached to the codes that run on the MIC, hence it is feasible to launch another process on the CPU as an additional MPI task. As a result, in this work the symmetric mode was chosen for XFLAT implementation.

There are several factors that affected the overall design of the code. First, the architecture of XFLAT needed to be as flexible as possible so that the physics is easily extensible and it can perform the simulation of diverse geometries and scenarios. Furthermore, since XFLAT is a high-performance code, it should be able to exploit all available parallelism levels and hardware resources (see Fig. 4.2).

In order to increase the flexibility of the code, XFLAT was designed to be modular. This modularity can save scientists' and developers' time and effort by allowing them to change a single physics module without any requirement of knowledge about the internal structure of other modules. Since various geometries, different particle types in the system, diverse energy spectra functions, ... might be required, a module can simply be switched with another module; thus only a code rebuilding

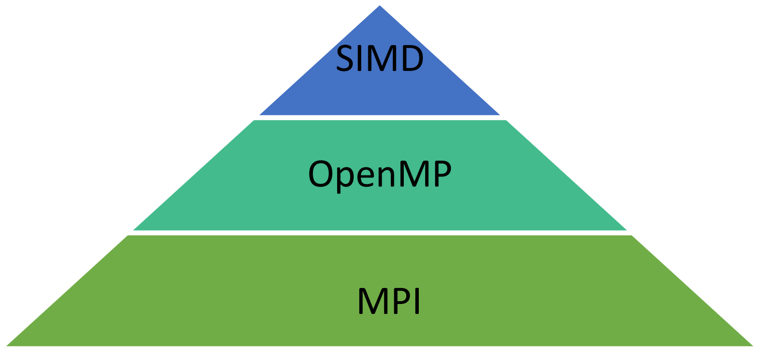


Figure 4.2: Parallelism hierarchy of XFLAT. MPI (bottom level) is employed for inter-node parallelism, OpenMP (shown in the middle) is exploited for multi-thread parallelism on each processor, and SIMD vectorization (shown at the top) is utilized for the finest-grained parallelism within each core.

(re-compilation), is necessary before running XFLAT.

As further illustrated in Fig. 4.3, the software framework architecture is a dual-layer design. The upper layer is responsible for general portions of the code including the numerical, physics, and I/O modules. The lower layer is responsible for per particle calculations such as calculating Hamiltonians and computing neutrino flavor evolution. In this approach, if a particle in the simulation needs to be replaced with another particle type, only the lower module needs to be swapped with a replacement module. In addition, if the geometry and physics of the simulation needs to be changed, the only affected module will be the physics module in the upper layer. The modularity of the code dictates that each module encapsulates all of its internal data, structures, and functionalities within the module, and only communicates with other modules via their exposed Application Programming Interface (API) functions. This requirement guarantees that by swapping modules the functionality of the rest of the framework will not be affected.

Together with this modular data structure, the code was designed so that overhead due to inter-node communications, data movement within memory, and I/O

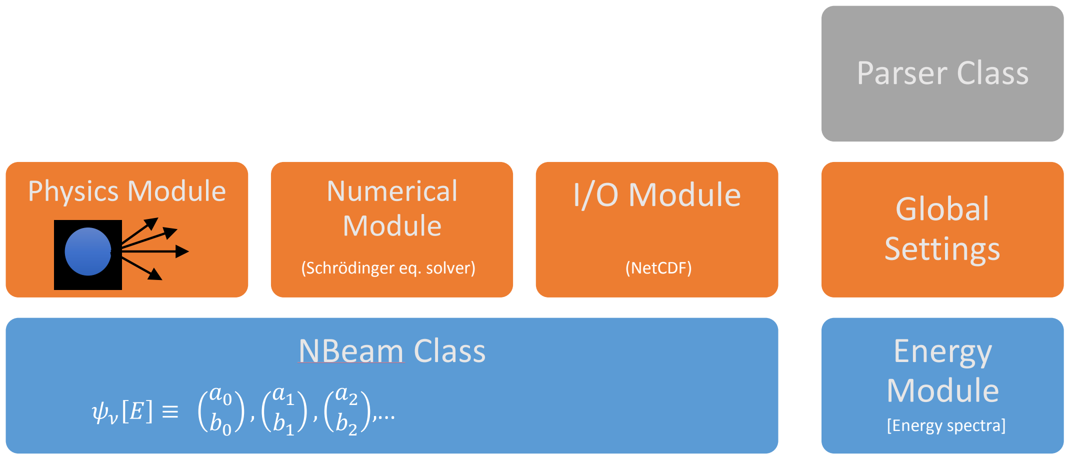


Figure 4.3: High-level hierarchical illustration of XFLAT modules. The bottom layer (shown in blue) is responsible for per particle calculations. The upper layer (shown in orange) is responsible for the general functionalities such as geometrical and I/O methods. Other modules (shown in grey) function as helper modules.

operations are minimized. As illustrated in Fig. 4.4, the application is initiated from the Parser class, which receives a configuration file from the command prompt console. The Parser extracts the variables from the configuration file and stores them in the Utility module. Next, the Parser starts reading the configuration file line by line. If it encounters the character '#', the line is commented out and is skipped. Otherwise, the Parser scans each line from left to right. The configuration file's tokens start at the beginning of each line and are in `TokenID= <value>` format. The Parser class maintains a list of valid tokens (i.e. `'TokenID='`), and tries to match the read token with one of the pre-defined tokens. Whenever a match is found, the `<value>` next to the token is read and stored in the corresponding token's variable. The rest of each line may contain a comment, which starts with a '#' as well. The Parser class continues scanning all of the lines of the configuration file and stores their values in the Utility module for later use.

Subsequently, the code continues within the top-layer modules by calling the initialization functions of every module in a precise order. The global module calls the `init()` functions of the NBeam module, the Physics module, the I/O module, and the Matter profile module, in that order.

The following subsections describe the key modules implemented in XFLAT.

4.2.1 NBeam module

The NBeam module is at the lowest level of the hierarchy that implements the NBeam class. Since each instance of the NBeam class represents a single neutrino beam, it contains an array of $\psi(\vec{v}, E)$ in which each element represents a neutrino with a particular direction and energy. As a result, the NBeam `init()` function is responsible for memory allocation of the energy bins as well as invoking the `init()` function of the Energy module in which the energy spectral function is initialized. For each component of the wavefunction, there is an array of double precision numbers inside the NBeam class. For instance, the wavefunction of the two-flavor system contains two complex numbers (a, b) and each complex number can be represented by two double precision floating point numbers, one for the real part and the other for the imaginary part (see Fig. 4.5). As a result, four double precision numbers can describe the flavor state of a neutrino at a given energy E (*i.e.*, $\psi(a_r, a_i, b_r, b_i; E)$). Consequently, for a given range of energies, there can be four arrays $a_r[E], a_i[E], b_r[E], b_i[E]$, and each element of those arrays corresponds to a flavor state for a particular energy bin of width ΔE $[E - \frac{1}{2}\Delta E, E + \frac{1}{2}\Delta E]$.

The NBeam class encapsulates several private (internal) methods. Those methods include the density method in which the density matrix calculations are performed (see Eq. 3.9), and the neutrino evolution method in which the corresponding flavor state $\psi_\alpha(\vec{v}, E)$ evolves one step further for a given Hamiltonian $\hat{H}$ (see

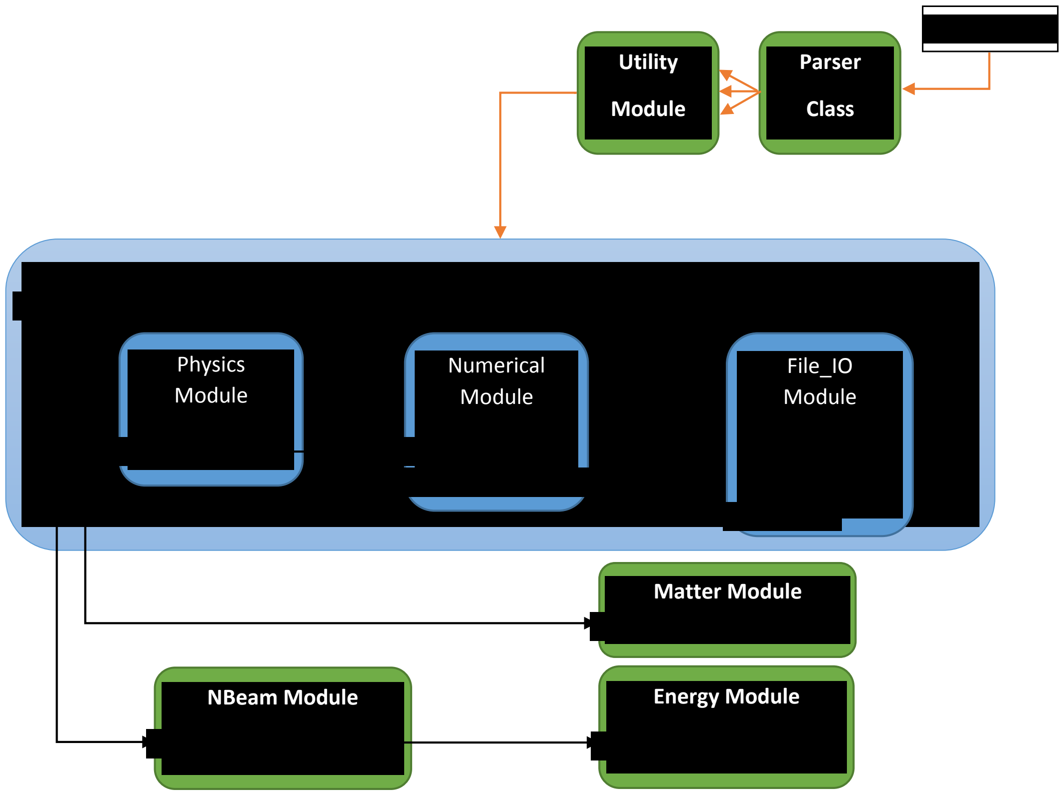


Figure 4.4: High-level illustration of XFLAT initialization. The Global module (shown in light blue) in which the upper layer's modules are encapsulated, begins initialization of each module in a particular order.

Eq. 3.11). In addition, the NBeam class contains several public (external) methods that are accessible from other modules. One of the important interface functions is `getESum()`, in which numerical integration over all energy bins is performed (the first integral in Eq. 3.8). The other important function is the `evolveBins(...)` function that loops over all energy bins and evolves each corresponding flavor state for a given Hamiltonian (see Eq. 3.11). Another of NBeam's public methods is the `addAvg(NBeam&)` method in which the average of the flavor states between two NBeam objects is calculated (one object is passed by reference and the other is the `this` pointer). Additionally, in order to calculate the error between two neutrino

```
class NBeam
{
private:
  // Each array's range is [E_min:E_max]
  double ar[], ai[], br[], bi[];
  // density matrix method
  void density(ar[E], ai[E], br[E], bi[E], ...);
  // single energy flavor state evolution
  void U(Hamiltonian H[], ...);
  ...
public:
  // calculates the integral over energy bins
  void getESum(...);
  // loops over energy range and evolve all of the flavor states
  void evolveBins(Hamiltonian H[], ...);
  // computes the average of the flavor states of two NBeam objects
  void addAvg(NBeam& beam);
  // calculates the error of flavor states
  double calcErr(NBeam& beam);
  ...
};
```

Figure 4.5: NBeam class overall structure and its private and public functions, for the two flavor system.

beams, the `calcErr(NBeam&)` method is provided. This method receives an object of NBeam as an argument and loops over energy bins to calculate the error between the flavor states of the current object, passed via `this`, and the received object. All of the public functions in the NBeam class exploit SIMD instructions to compute the $a_r[E], a_i[E], b_r[E], b_i[E]$ components for $\psi_\alpha(\vec{v}, E)$. Since NBeam objects were distributed over processors' cores, and the NBeam module is located on the bottom layer of the XFLAT architecture, one NBeam instance can entirely exploit the first layer of parallelism within each core (SIMD) for the flavor state calculations (see Fig. 4.6).

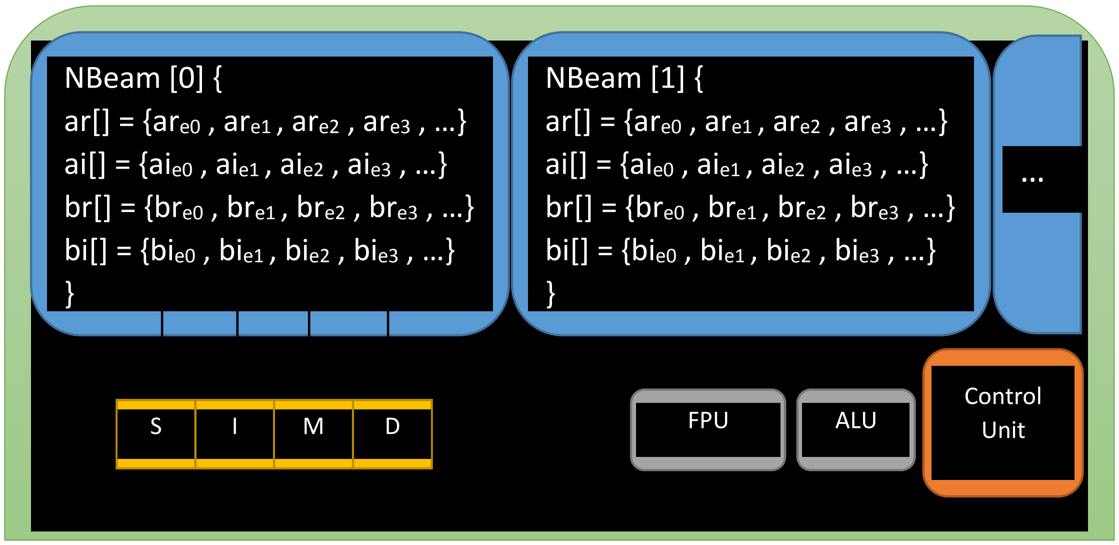

NBeam [0] {
ar[] = {are0 , are1 , are2 , are3 , ...}
ai[] = {aie0 , aie1 , aie2 , aie3 , ...}
br[] = {bre0 , bre1 , bre2 , bre3 , ...}
bi[] = {bie0 , bie1 , bie2 , bie3 , ...}
}
NBeam [1] {
ar[] = {are0 , are1 , are2 , are3 , ...}
ai[] = {aie0 , aie1 , aie2 , aie3 , ...}
br[] = {bre0 , bre1 , bre2 , bre3 , ...}
bi[] = {bie0 , bie1 , bie2 , bie3 , ...}
}
...
S
I
M
D
FPU
ALU
Control Unit

Figure 4.6: Within a processor core (green), NBeam objects can employ the lowest level of parallelism, SIMD unit (shown in yellow). Each NBeam instance (shown in blue) issues vector instructions in order to utilize SIMD unit.

4.2.2 Physics module

The Physics module is the next module from which the `init()` function is called from the global module. The Physics module is responsible for the implementation of geometrical functions as well as the computation of the Hamiltonian. The module encapsulates functions in which computations on arrays of NBeam objects are performed via OpenMP threads (see Fig. 4.7). The instructions within those functions are based on the plugged-in geometry module and may differ between modules. The Physics module's `init()` function has two major responsibilities. First, it is responsible for allocating the angle-dependent arrays (such as $\cos(\vartheta)$ bins). Furthermore, it calls the `init(int)` method of the Numerical module to which an integer as the length parameter is passed. The length parameter is the number of all trajectories along all different directions, such as zenith direction (ϑ) and azimuth (φ) angles. Since the Physics module maintains information about the number of dimensions and

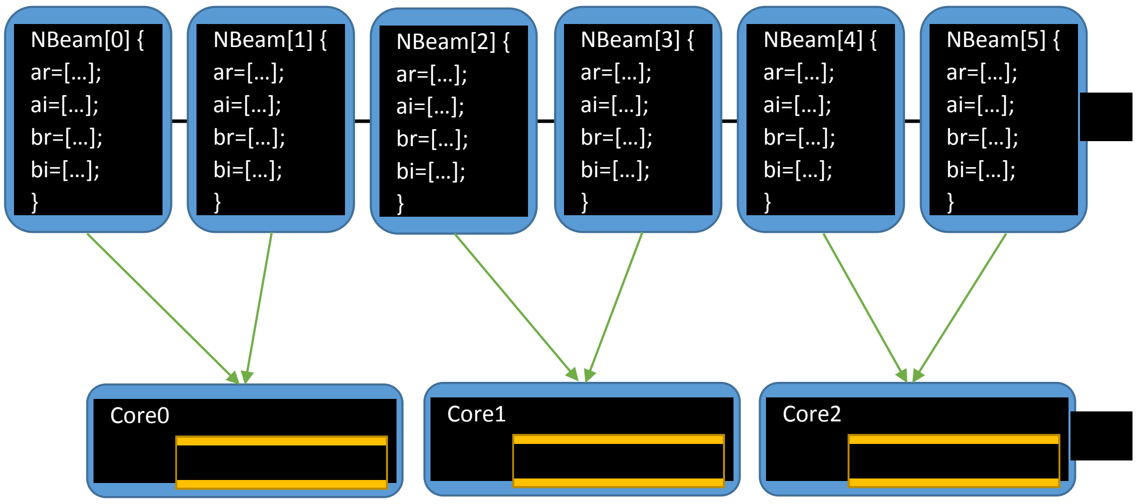


Figure 4.7: Arrays of NBeam objects are distributed over processor cores via OpenMP. Thus, each core may be responsible for the calculations of several NBeam class instances.

the length of each dimension, the Numerical module can receive the length parameter only from the Physics module.

Within the Physics module there are several methods that are accessible and utilized from the Numerical module. One of these methods is `initBeam(NBeam*)`, which receives an array of NBeam objects, loops over each element of the array, and calls the constructor of each element. This function must be called prior to any function calls from the NBeam class in order to initialize and allocate memory for the wavefunction's components within each object. Thus, for each NBeam class array, the Numerical module calls the `initBeam(NBeam*)` method to initialize each array's elements. There are two other important methods within the Physics module that are utilized from the Numerical module. The first one is the `newHvv(double*&)` method in which the memory for an array of Hamiltonians is allocated (each particular neutrino beam trajectory has a particular, corresponding Hamiltonian). The last method is the `deleteHvv(double*&)` method in which previously allocated memory for the Hamiltonians is freed.

4.2.3 Numerical module

Upon calling the `init(int)` method of the Numerical module, depending on the implemented numerical algorithm, multiple arrays of NBeam instances may be required to be allocated and maintained in main memory. For each allocated NBeam array, the `initBeam(NBeam*)` method of the Physics module must be called to initialize the array. Hence, the `evolutionLoop()` method in the Numerical module is called from the global module to begin the neutrino flavor evolution calculations. `evolutionLoop()` continues the neutrinos evolution calculations until one of the *application-ending* criteria reaches its maximum value. The application-ending criteria parameters are: the maximum radius, the maximum allowable run time, or the maximum number of radial iterations. Before starting the evolution loop, the first step within the `evolutionLoop()` method is to perform memory allocation for the arrays of Hamiltonians. This is done by calling the `newHvv(double*&)` method from the Physics module. Afterwards, the neutrino evolution function is called from the global module.

The flowchart of the neutrino evolution loop is illustrated in Fig. 4.8. The neutrino evolution function is responsible for solving the the Schrödinger equation (Eq. 3.2) and evolving the flavor state of each neutrino beam accordingly. Fig. 4.9 illustrates the algorithm steps in order to solve the Schrödinger equation within the `evolutionLoop()` function. The blue squares represent wavefunction states of neutrinos, and the green squares represent the calculated Hamiltonian from the previous state functions. The green arrows represent the spacial length and the direction of neutrino evolution by applying the Hamiltonian, the 'Op(Avg)' box represents averaging between two state functions, and the '?' box shows the comparison of two

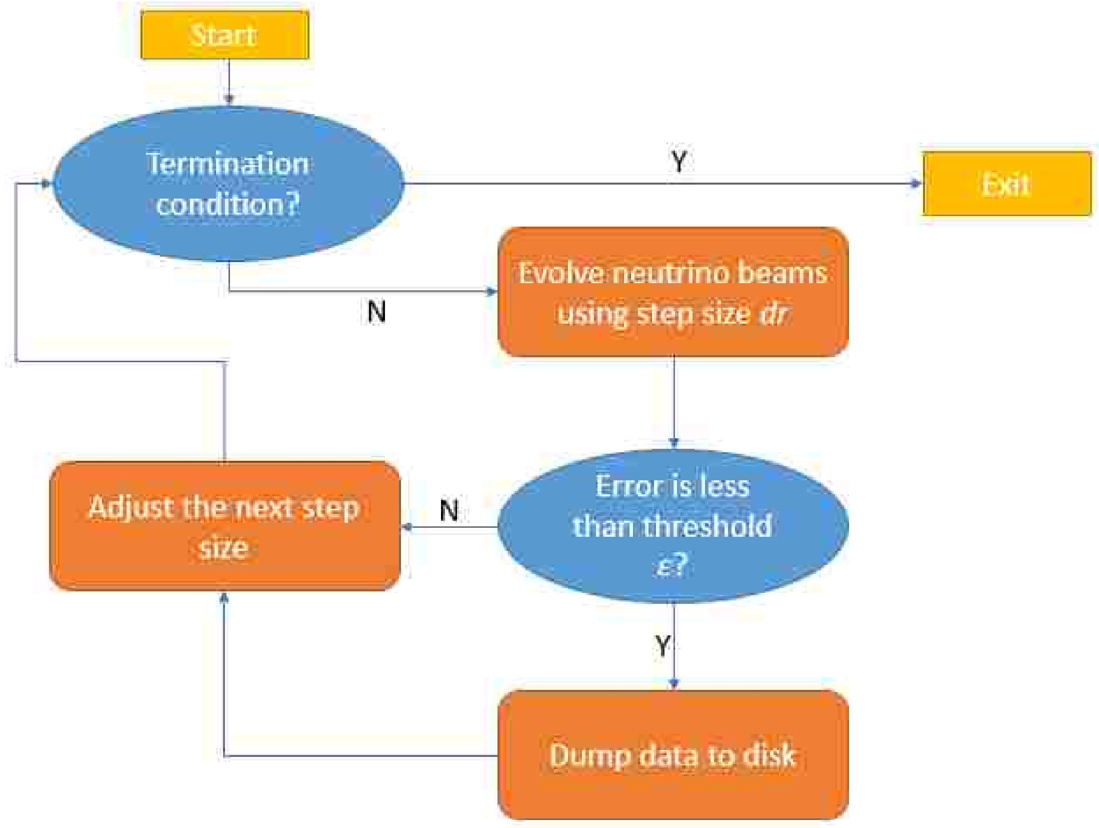

Start
Termination condition?
Y
Exit
Evolve neutrino beams using step size *dr*
N
Adjust the next step size
N
Error is less than threshold ε?
Y
Dump data to disk

Figure 4.8: High-level neutrino evolution flowchart. The loop always starts by checking the termination conditions. The three major phases: the evolving of neutrino beams phase, the dumping data phase, and the adjusting step size phase (shown in orange) are performed accordingly.

state functions. If the error is less than a predefined threshold, the results are saved.

As can be seen in Fig. 4.9, within the Numerical module at several points, the neutrino-neutrino Hamiltonian matrix (Eq. 3.8) must be calculated based on the previously-calculated neutrinos' wavefunctions. Each matrix calculation can be performed via a function call within the Physics module. The `calc_Hvv(...)` method is responsible for computing the neutrino self-coupling Hamiltonian integral for which the loops and instructions within the function depend on the implemented geometry. Once this method is called, it receives the current NBeam array's pointer

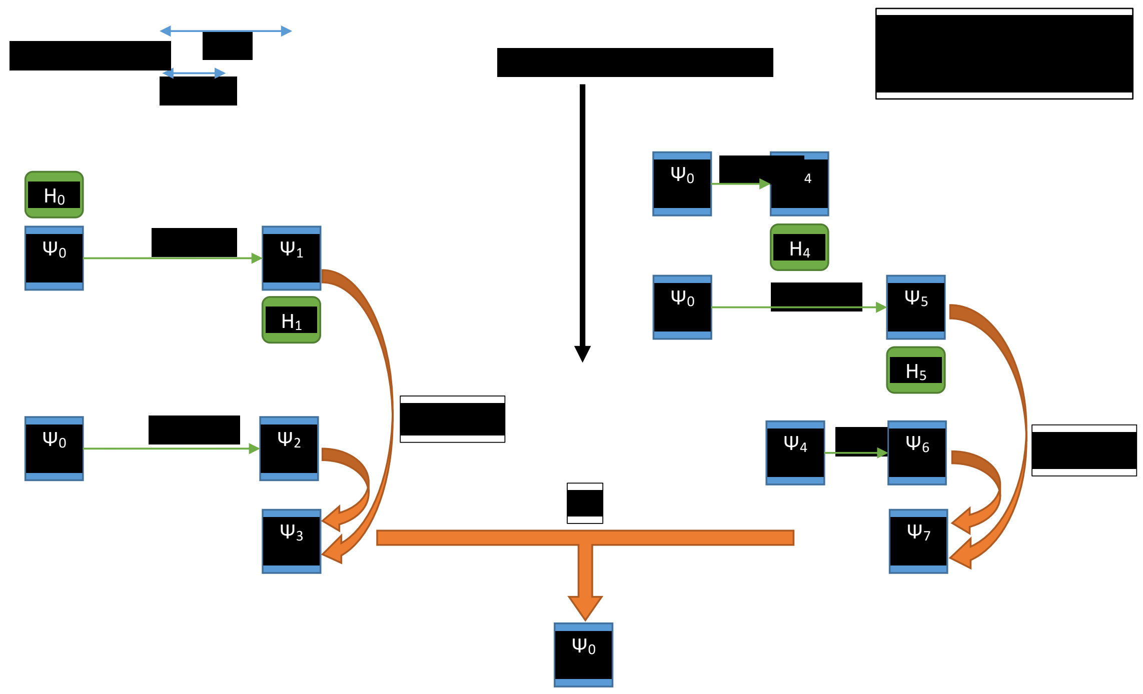


Figure 4.9: Illustration of XFLAT's numerical algorithm for solving the Schrödinger equation (Eq. 3.2). The algorithm starts at the top left, and moves from top to bottom, and left to right. The blue squares represent flavor states of neutrinos; the green squares represent the calculated Hamiltonian from the previous state functions; the green arrows represent the direction and step size of neutrino evolution using Hamiltonians; the 'Op(Avg)' boxes represent the averaging between two flavor states, and the '?' box shows the comparison of two flavor states (if the error is less than a predefined threshold then results are saved).

as an argument and performs a summation over the array's elements via OpenMP threads. This is done by computing the partial Hamiltonian integral summation (the pseudocode for the neutrino bulb model corresponding to Eq. 3.8 is illustrated in Fig. 4.10). Within the loop, by calling each NBeam object's `NBeam::getESum()` method, in which the SIMD unit is employed, the summation of wavefunctions over energy bins is calculated internally. If the code runs in a multi-node configuration, there will be several data exchange points within the algorithm. MPI functions must then be used, since the NBeam arrays are distributed over multiple compute nodes

```
// parallel loop via OpenMP
#pragma omp parallel for
for (int ang = 0; ang < THETA_BINS; ++ang)
{
  // loop over neutrino flavors
  for (int n = 0; n < FLAVOURS; ++n)
  {
    ...
    // calculates Sum(E) of NBeam objects internally (SIMD)
    beam[neutrino_idx     ].getESum(res_neu[] );
    beam[anti_neutrino_idx].getESum(res_aneu[]);
    ...
  }
  // apply cos(t) and dcos(t) to the result
  angle_calc(result[], ...);
}
```

Figure 4.10: Illustration of the loop structure that is responsible for computing the partial Hamiltonian integral summation.

(see Fig. 4.11). Consequently, to evaluate the final Hamiltonian integral summation's result, each MPI task exchanges its partially-calculated integral result with all other tasks by employing the MPI reduction method. Afterwards, each node will have the final Hamiltonian integral value.

After the calculations of the Hamiltonian, the corresponding neutrino flavor state residing at the parameter value r on a given trajectory must be evolved one step further for the given Hamiltonian H and step size Δr. Therefore, the `evolve(...)` method within the Physics module is called in which it receives an array of NBeam objects and performs a loop over them via OpenMP threads. Within the loop, the NBeam's `NBeam::evolveBins(...)` method is called for each object. Within the `NBeam::evolveBins(...)` method, the SIMD unit is exploited in order to perform the neutrinos' flavor state evolution for all energy bins. The neutrinos' flavor state evolution can be performed by applying the calculated neutrino self-coupling Hamil-

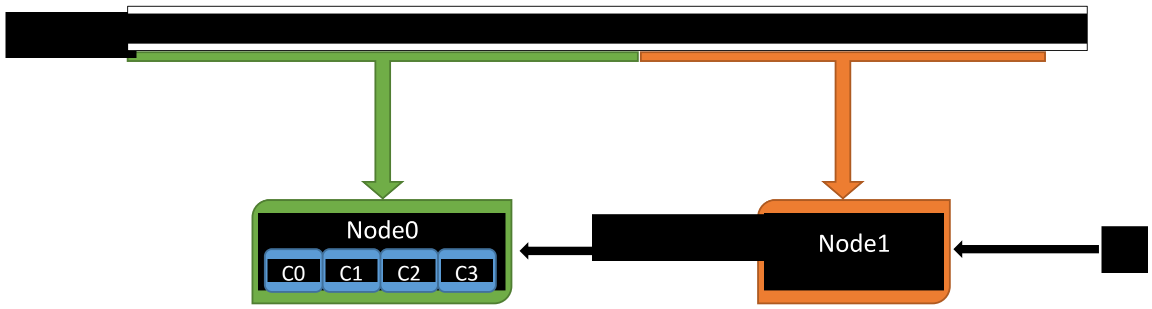


Figure 4.11: Distribution of NBeam objects over multiple nodes. Each node, based on its computational capacity, will receive a particular load. For instance, Node0 (shown in green) is more powerful than Node1 (shown in orange), thus it received more NBeam objects [NBeam0:Nbeam4] to process while Node1 received only four objects [NBeam5:Nbeam8].

tonian H (the pseudocode corresponding to Eq. 3.11 is illustrated in Fig. 4.12) and updating the flavor states accordingly. The result of the evolved flavor state is saved in another one-dimensional array of NBeam objects.

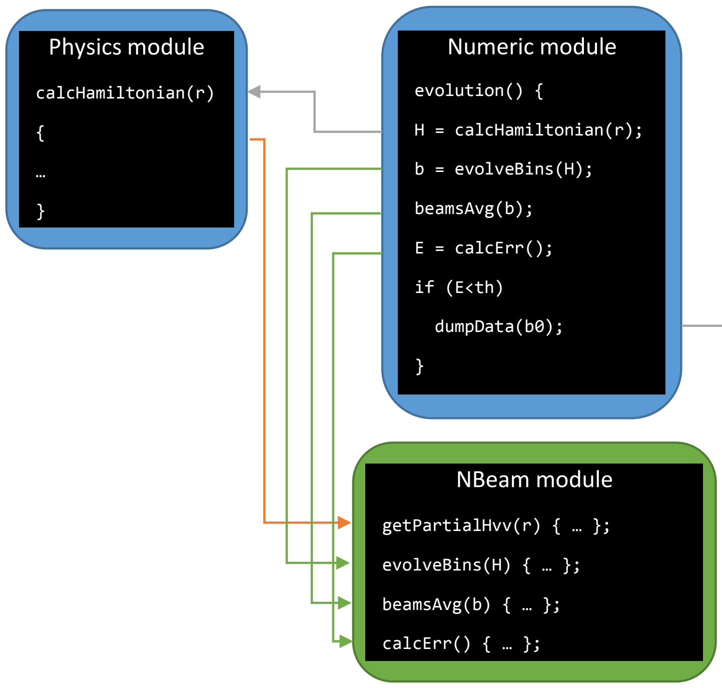
After evolving the neutrinos' flavor states, the next step is to compute the average of the two calculated flavor states. The two flavor states were evolved separately using two different Hamiltonians and different step sizes. Therefore, the `avgBeam(...)` function from the Physics module is called to perform this task. This function receives two NBeam object arrays as the argument to calculate the average between their wavefunctions. The function employs OpenMP threads for looping over NBeam objects. OpenMP threads are responsible for calling each NBeam object's `NBeam::addAvg(NBeam&)` method. The `NBeam::addAvg(NBeam&)` method is responsible for computing the average of neutrino flavor states of all energy bins between two NBeam objects via SIMD (see Fig. 4.13). Since the evolving and averaging steps can be completed separately on each node, no MPI message-passing calls are required for those tasks, so there is no network communication overhead.

The last step of the neutrino evolution loop is to find the global maximum differ-

```
#pragma omp simd
for (int e = 0; e < NUM_of_EBINs; ++e)
{
  ...
  // calculates the vacuum Hamiltonian
  getH0(e, h0);
  // add vaccum hamiltonian to hamilt[]
  ...
  double lambda = sqrt( hamilt00^2 + hamilt01^2 );
  double ldr = lambda * dr;
  double cosCoef = cos(ldr);
  double sinCoef = sin(ldr) / lambda;
  ...
  // complex numbers multiplications for result[]
  ...
  // save the final values to neuBeam's components
  neuBeam2->ar[e] = result00_real + result01_real;
  neuBeam2->ai[e] = result00_imag + result01_imag;
  neuBeam2->br[e] = result10_real + result11_real;
  neuBeam2->bi[e] = result10_imag + result11_imag;
}
```

Figure 4.12: Pseudocode illustrating evolution of the neutrinos' flavor state for all energy bins within the NBeam class. Since the computations of different energy bins are all identical, the loop is vectorized via SIMD.

ence (error) between the flavor states of the two final evolved neutrino beams (the ψ_3 and ψ_7 in Fig. 4.9). Hence, there is one more OpenMP-managed loop over the NBeam objects to find the maximum difference between the two arrays of the neutrinos' flavor state. Within the loop, for each element within the NBeam object's array, the `NBeam::calcErr(NBeam&)` method is called. This method calculates the error across all the energy bins via vector registers. Afterwards, the global maximum error may be found by performing a global reduction with respect to all local maximum errors on each node, via MPI. Consequently, if the computed global error is less than a predefined threshold, the results of the evolved wavefunctions are accepted. Then,



```
calcHamiltonian(r)
{
…
}
```

```
evolution() {
H = calcHamiltonian(r);
b = evolveBins(H);
beamsAvg(b);
E = calcErr();
if (E<th)
  dumpData(b0);
}
```

```
dumpData(b0)
{
…
}
```

```
getPartialHvv(r) { … };
evolveBins(H) { … };
beamsAvg(b) { … };
calcErr() { … };
```

Figure 4.13: Function dependency between modules and the order of the function calls. Upper layer modules are shown in blue, and bottom layer modules are shown in green.

depending on the I/O module configurations, the wavefunctions may also be required to be saved to a file. In the current version of XFLAT, the Network Common Data Form (NetCDF) library [NetCDF, 2015] is employed to perform all I/O tasks. (Note: alternative I/O APIs can be supported in the future by implementing appropriate I/O modules in XFLAT.) The next iteration continues with the final evolved neutrino flavor states. However, if the global error is greater than the threshold, the computed evolved flavor states are discarded, the step size is adjusted accordingly, and the next iteration resumes with the previous flavor states.

4.2.4 I/O module

The I/O module is the next module whose `init()` function is called from the global initialization process. This function is responsible for initializing data files in which the results of computations are stored. The NetCDF file format is used to write data to disk. Several dump modes are provided via the configuration file's parameters. Thus, XFLAT may create several data files in which different calculations' results with different formats can be stored. The first implemented dump mode is responsible for taking a snapshot of the neutrino flavor states by writing them to a file: without any further processing, the raw values of the neutrinos' wavefunctions are written to disk. Later, this file may be used to resume computations from a particular radial point. However, if many snapshots are required during runtime, the size of the data file may grow very large. For instance, for the problem size of $1000(\vartheta) \times 100(\varphi) \times 100(energy\ bins)$, the size of a single snapshot would be $1000 \times 100 \times 100 \times 4(particles) \times 4(wavefunction\ components) \times 8(size\ of\ double) = 1.28$ GB. Thus, if the code is run out to $r = 100$ km, and even if only one snapshot is taken per kilometer, the total file size will be 128 GB! As a result, there may be issues in opening and manipulating the data file due to its size. On the other hand, if each snapshot is saved as a single file, the overhead of opening and closing files may affect the overall performance. To provide control over these potential I/O issues, XFLAT can be provided a parameter via the configuration file that defines the maximum number of saved snapshots per file. When the maximum number of allowable snapshots is reached, XFLAT closes the current data file and creates a new file. As a result, the overhead of opening and closing a file per snapshot is reduced and if the code crashes during runtime, previous output is preserved in the already-closed snapshot files.

The second implemented dump mode stores less data by preprocessing data before saving it. Thus, instead of dumping the wavefunction's components for every beam, it first calculates the average flavor states of all energy bins for a single neutrino's trajectory, then stores the results in a file. As a result of this averaging process, there is no need to write flavor state results per energy bin individually, and the data files become significantly smaller. The drawback of this approach is that the generated files cannot be used as XFLAT restart files since the information related to each individual energy bins has been lost.

Similar to the other parts of XFLAT, the dump mode functionality is extensible as well, and alternative dump modes can be added in the future based on requirements. XFLAT can simultaneously work with a combination of dump modes since the provided dump mode's code in the configuration file is encoded as a bit pattern, where each bit of the dump code in the configuration file corresponds to a separate dump code. In the configuration file, the dump code 0 means no I/O tasks. The dump code 1 (001) represents the first mode and the dump code 2 (010) represents the second mode. The third dump mode is represented by binary number 100, *etc.* The binary codes can be combined to enable multiple dump modes simultaneously. As an example, since the binary representation of dump code 3 is 011, this code will enable both the first and the second dump modes simultaneously. Dump code 5 enables the first and the third dump modes (101 in binary); 6 enables the second and the third dump modes (110 in binary), and 7 enables all of three dump modes (111 in binary). Additional dump modes can be added to XFLAT using this approach.

As discussed below in the performance analysis section, the I/O performance of the the Xeon Phi is very poor, thus heavy I/O tasks should be avoided on the Xeon Phi, otherwise the co-processor's poor I/O performance will hurt the overall performance of the application. To address this issue, in XFLAT it is possible to redirect the Xeon Phi's I/O traffic to the corresponding CPU. Thus, instead of dumping data from the Xeon Phis directly onto disk, their processed data is written onto disk indirectly via the CPUs. Prior to redirecting the I/O traffic, each instance of XFLAT has to know if it is running on the Xeon Phi or on the CPU. This is detected within the application as follows. By default, on the MIC the `hostname` is the same as its host (CPU), with the addition of `'-mic'` at the end. For instance, a host name might be `hostid1`, thus its first MIC's `hostname` will be `hostid1-mic0` and its second MIC's `hostname`, if installed, will be `hostid1-mic1`. As a result, each instance of XFLAT can query its host name by calling the `MPI_Get_processor_name()` method and checking whether or not the host name includes the `'-mic'` suffix. Afterwards, all XFLAT instances exchange their `hostname` by calling the `MPI_Allgather()` method, in order to find their mate. For instance, on a compute node with two CPUs and two MICs, the first CPU rank is 0, the second CPU rank is 1, the first MIC rank (connected to CPU 0) is 2, and the second MIC rank (connected to CPU 1) is 3. Therefore, the first MIC (rank=2) must send its data to the first CPU (rank=0), and the second MIC (rank=3) must send its data to the second CPU (rank=1). If the XFLAT task runs on the MIC, it tries to find the CPU id (the MPI rank) to which it is connected. Since every process maintains the list of processors ranks and host names, the XFLAT task that runs on the MIC starts from $rank - 1$ and checks the host names. If it encounters another MIC immediately, it means that the current MIC is the second installed MIC, and should thus connect to the second CPU. If by decreasing the rank a CPU is encountered first, that means that the current MIC is the first installed MIC, so it continues rank decrements until it finds the first CPU of the current node. (At this point by one more decrements the compute node may change and the host names change completely, *e.g.* `hostid1` to `hostid0`, thus it indicates that we found the first CPU rank on the current node, *i.e.*, on the `hostid1`).

As mentioned earlier, XFLAT can resume computation from a previous run. In order to complete the resumption task, XFLAT must first check whether or not an input data file was provided to the application. If an input file was provided to XFLAT, the neutrinos' flavor state of `NBeam` objects must be initialized using the neutrinos' flavor state from the last snapshot that was stored in the file. Therefore, in contrast to the `NBeam` object initialization from within the `phys::initBeam(NBeam*)` function, the objects are initialized with the final flavor states which were saved to the file. As a result, the `phys::initBeam(NBeam*)` function may call the `fillInitData(NBeam*)` method from the I/O module in order to open the provided data file, read the stored neutrinos' flavor states, and initialize the `NBeam` objects with them.

In order to act properly, the I/O module requires information about the data that will be written to disk. This information is the length and the number of dimensions as well as the name of each dimension. The I/O module can query this information from the Physics module, since the Physics module maintains the geometry-related information. The first method is `phys::beamLen()` returning the length of the `NBeam` arrays. The second method is `phys::getDim()` which returns the number of dimensions in the geometry. The next method is `getDimInfo(string str[])` which returns the name of each dimension so as to distinguish them in the file. The I/O module may use this information for formatting the saved data. The next two methods are `phys::startDim()` and `phys::countDim()` which are utilized by the I/O module in order to receive an array of starting points and an array containing the length of each data dimension, respectively. The `phys::startBeamIdx()` and `phys::endBeamIdx()` methods return the index of the beginning and ending beam of the first dimension. (Since the neutrino beams are distributed over compute nodes by the first dimension, the starting and ending beam indices are required for the I/O module on the nodes to function properly). For instance, if there are 1000 zenith angle beams and are distributed over ten identical nodes, the first node receives `NBeam[0:99]`, the second node receives `NBeam[100:199]`, *etc.* and there will be no conflict between node data. The last public method is the `phys::firstDimLen()` that returns the total length of the first dimensions on which the data is distributed over nodes (depending on the implemented geometry module, the first dimension length may be the size of the ϑ or φ dimensions).

4.2.5 Matter module

In addition to the previous initialization function, another module in XFLAT from which the `init()` function is called by the global module is the Matter module. This module is responsible for implementing the matter profile.

4.2.6 Energy module

This module is responsible for the implementation of the energy spectra function for neutrinos. The NBeam module employs the functions of this module.

4.2.7 Utility module

The Utility module fetches and saves global variables from the Parser. The Utility module also implements miscellaneous helper functions that are employed by multiple modules. For instance, the complex number multiplication `mul_cmplx()` and the `norm2()` methods are two important methods that are utilized by several other modules. Since the other modules have access to the Utility module's data, it contains the global state of the application that is used to indicate whether or not the application is in a specific state such as the benchmark state. Initialization is different in *benchmark mode* where there is no requirement for loading and initializing the I/O module.

4.2.8 XFLAT termination

XFLAT's final step after completing the neutrino evolution loop and storing the results is the memory deallocation by calling the `freemem()` functions. The global module calls the I/O module's `freemem()` in which the NetCDF open files are closed.

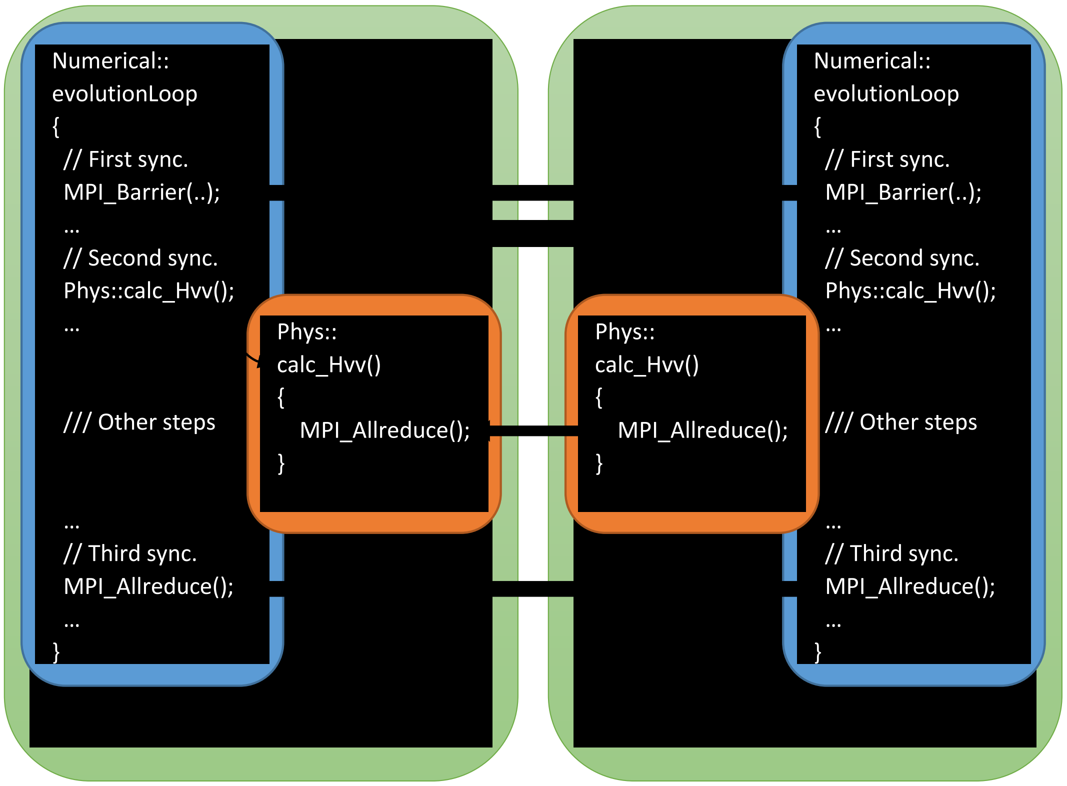
The next call is the Physics module's `freemem()` method in which the deconstructor of NBeam objects are called from within a loop. Hence, all the assigned memory is deallocated sequentially prior to the application termination.

As discussed earlier, multiple copies of the NBeam arrays are maintained in memory in which the intermediate calculation results are maintained. Those arrays are among the data that has to be transferred between XFLAT modules. Since they are all represented by one-dimensional arrays, modules can transfer them via their pointers (a single variable) without the requirement to pass further data. For instance, the number of required NBeam arrays is defined in the Numerical module, then their pointers are directed to the Physics module for memory allocation, and at the end of each iteration the pointer to the final result is forwarded to the I/O module for I/O tasks. As a result, the number of NBeam arrays required by the solver does not place extra overhead on the inter-module communications, and thus does not produce performance bottleneck.

4.3 High-level Parallelization

As previously mentioned, XFLAT initially allocates one-dimensional NBeam object arrays of neutrino's angle beams within main memory. Since the number of neutrino beams may be quite large for complicated geometries, domain decomposition of NBeam arrays over multiple nodes may be needed in order to avoid exceeding the capacity of local on-node RAM. For inter-node communication, the MPI library is used (*de facto* standard on supercomputer systems). One MPI process on each processor is used. The processor can be either a CPU installed on a socket or the Xeon Phi card installed on a PCIe slot. Creating more than one MPI process per processor (as an alternative to multithreaded use of the multicore processors) is supported as well, but this mode of execution did not provide significant performance improvement in tests. This is because increasing the number of MPI tasks per processor also increases the MPI communications overhead. In addition, the number of communication synchronization points in the code should ideally be minimized in order to reduce the processors' idle time as well as the overhead. Consequently, in XFLAT there are only three major MPI synchronization points in the main loop. At the first synchronization point, the root node broadcasts global variables such as the next calculated radius or the computed termination conditions to all of the nodes. At the second communication point, nodes exchange the results of the background neutrino-neutrino partial Hamiltonian in order to compute the final integral summation for every node. Finally, at the third communication point, nodes exchange their local maximum error in order to find the global maximum error among all neutrino beams. Fig. 4.14 depicts the location of these synchronization points within the modules.

Modern CPUs as well as the Intel MIC contain multiple cores. Thus, within each MPI node there may be an additional level of parallelism. To take advantage of this, the NBeam object arrays are distributed across all available cores and hardware threads via OpenMP. Since inside a processor or co-processor the memory model is shared, *i.e.*, the entire memory is accessible by all cores, OpenMP is the appropriate software parallelization approach to use. In this way, the threads' communications and synchronizations remain entirely within processor. The advantage of keeping threads' communications within a processor is that the communication between CPUs or between the CPU and MIC is slow in comparison to intra-processor communications. The CPU-CPU communication is performed via the QuickPath Interconnect (QPI) bus, and the CPU-MIC communication is performed via the high latency PCIe bus. OpenMP is preferred over MPI for intra-processor computations since cores are able to perform context switching between its lighter-weight threads more efficiently. Placing multiple processes on a core decreases performance due to process context switching overhead. In XFLAT, all available hardware threads on



```
Numerical::
evolutionLoop
{
  // First sync.
  MPI_Barrier(..);
  ...
  // Second sync.
  Phys::calc_Hvv();
  ...

  /// Other steps

  ...
  // Third sync.
  MPI_Allreduce();
  ...
}
```

```
Phys::
calc_Hvv()
{
    MPI_Allreduce();
}
```

```
Phys::
calc_Hvv()
{
    MPI_Allreduce();
}
```

```
Numerical::
evolutionLoop
{
  // First sync.
  MPI_Barrier(..);
  ...
  // Second sync.
  Phys::calc_Hvv();
  ...

  /// Other steps

  ...
  // Third sync.
  MPI_Allreduce();
  ...
}
```

Figure 4.14: MPI communication points between nodes (green). There are several MPI calls within the Numerical module (blue) as well as MPI calls within the Physics module (orange).

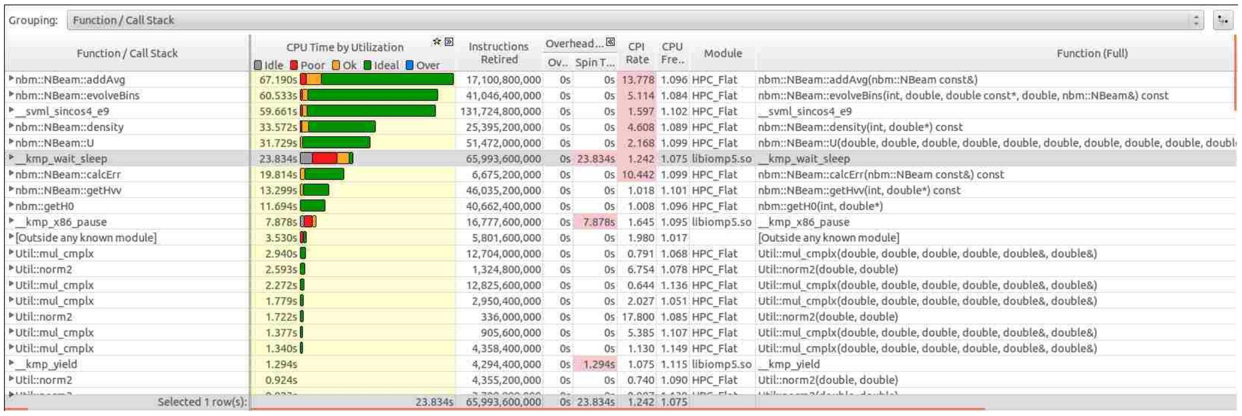
the CPU and on the Xeon Phi (the number depends on the MIC's model, and ranges from 228 to 244 threads), are exploited via OpenMP.

The last level of parallelism is the SIMD unit within each processor core. Neutrinos propagating along a given direction have many different energies. Since the calculations on energy bins are identical, multiple energy bins' data can be packed together and the calculations performed simultaneously via SIMD units on the cores. Hence, each OpenMP thread can exploit SIMD instructions to perform calculations on the NBeam object's inner arrays in parallel. As a result, XFLAT can utilize all of the available levels of parallelism on modern multinode supercomputers.

4.4 Optimization

The final design challenge in XFLAT was to optimize it to run efficiently on the Intel MIC co-processors as well as the latest generation of CPUs. The first step of this process is to identify potential performance bottlenecks. Since bottlenecks can arise due to various factors including thread overhead, memory latency, cache line evictions, pipeline evictions, *etc.*, understanding root causes upfront is essential in guiding initial code design. Once the initial code has been written, the next step, performance analysis, is accomplished by studying the behavior of the code manually, or by employing analysis tools. There are various tools available on the market to help programmers find code bottlenecks quickly. The analyzer that was utilized for a part of the XFLAT performance analysis is Intel VTune [Intel, 2015c]. VTune can be used to perform code analysis on both Xeon CPUs and the Xeon Phi. The tool provides useful information on memory latency, cache lines eviction rate, processor pipeline occupancy, processor cycles per instruction ratio, SIMD instruction unit occupancy, *etc.* categorized by function calls (see the example in Fig. 4.15). Consequently, it is easy to see which procedure on which processor's unit may be a potential cause of a bottleneck.

In approaching the problem of performance optimization it should be noted that conventional Object-Oriented Programming (OOP) approaches present challenges for high-performance computing code design since their first goal is to achieve *Rapid Application Development* (RAD) cycles. Maintaining clear class hierarchies and utilizing virtual and multiple inheritances takes higher priority than performance. On the other hand, typical high-performance implementation techniques result in having less complex data structures, and thus less code flexibility, in order to optimize per-



Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization	Instructions Retired	Overhead...		CPI Rate	CPU Fre..	Module	Function (Full)
	▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over		Ov..	Spin T...				
nbm::NBeam::addAvg	67.190s	17,100,800,000	0s	0s	13.778	1.096	HPC_Flat	nbm::NBeam::addAvg(nbm::NBeam const&)
nbm::NBeam::evolveBins	60.533s	41,046,400,000	0s	0s	5.114	1.084	HPC_Flat	nbm::NBeam::evolveBins(int, double, double const*, double, nbm::NBeam&) const
__svml_sincos4_e9	59.661s	131,724,800,000	0s	0s	1.597	1.102	HPC_Flat	__svml_sincos4_e9
nbm::NBeam::density	33.572s	25,395,200,000	0s	0s	4.608	1.089	HPC_Flat	nbm::NBeam::density(int, double*) const
nbm::NBeam::U	31.729s	51,472,000,000	0s	0s	2.168	1.099	HPC_Flat	nbm::NBeam::U(double, double, double, double, double, double, double, double, double, doubl
__kmp_wait_sleep	23.834s	65,993,600,000	0s	23.834s	1.242	1.075	libiomp5.so	__kmp_wait_sleep
nbm::NBeam::calcErr	19.814s	6,675,200,000	0s	0s	10.442	1.099	HPC_Flat	nbm::NBeam::calcErr(nbm::NBeam const&) const
nbm::NBeam::getHvv	13.299s	46,035,200,000	0s	0s	1.018	1.101	HPC_Flat	nbm::NBeam::getHvv(int, double*) const
nbm::getH0	11.694s	40,662,400,000	0s	0s	1.008	1.096	HPC_Flat	nbm::getH0(int, double*)
__kmp_x86_pause	7.878s	16,777,600,000	0s	7.878s	1.645	1.095	libiomp5.so	__kmp_x86_pause
[Outside any known module]	3.530s	5,801,600,000	0s	0s	1.980	1.017		[Outside any known module]
Util::mul_cmplx	2.940s	12,704,000,000	0s	0s	0.791	1.068	HPC_Flat	Util::mul_cmplx(double, double, double, double, double&, double&)
Util::norm2	2.593s	1,324,800,000	0s	0s	6.754	1.078	HPC_Flat	Util::norm2(double, double)
Util::mul_cmplx	2.272s	12,825,600,000	0s	0s	0.644	1.136	HPC_Flat	Util::mul_cmplx(double, double, double, double, double&, double&)
Util::mul_cmplx	1.779s	2,950,400,000	0s	0s	2.027	1.051	HPC_Flat	Util::mul_cmplx(double, double, double, double, double&, double&)
Util::norm2	1.722s	336,000,000	0s	0s	17.800	1.085	HPC_Flat	Util::norm2(double, double)
Util::mul_cmplx	1.377s	905,600,000	0s	0s	5.385	1.107	HPC_Flat	Util::mul_cmplx(double, double, double, double, double&, double&)
Util::mul_cmplx	1.340s	4,358,400,000	0s	0s	1.130	1.149	HPC_Flat	Util::mul_cmplx(double, double, double, double, double&, double&)
__kmp_yield	1.294s	4,294,400,000	0s	1.294s	1.075	1.115	libiomp5.so	__kmp_yield
Util::norm2	0.924s	4,355,200,000	0s	0s	0.740	1.090	HPC_Flat	Util::norm2(double, double)
Selected 1 row(s):	23.834s	65,993,600,000	0s	23.834s	1.242	1.075		

Figure 4.15: Intel vTune report section screenshot, for XFLAT.

formance. The implementation approach used for XFLAT attempts to find the best balance between flexibility and performance. The code architecture was designed to be dual-layer and modular as much as possible. The priority of the internal structure of the lower layer is to gain high-performance computing capability on both the CPU and Xeon Phi, while the design goal of the upper layer is flexibility to support future expansion and new modules. The following subsections describe the principal techniques and methods used in XFLAT optimization.

4.4.1 Changing Array of Structures (AoS) to Structure of Arrays (SoA)

There are two high-level and general approaches for designing and implementing data structures. The simpler approach is to maintain an array of structures in which each structure encapsulates variables for a single particle. The other approach is to maintain a single structure in which arrays of data are allocated and maintained.

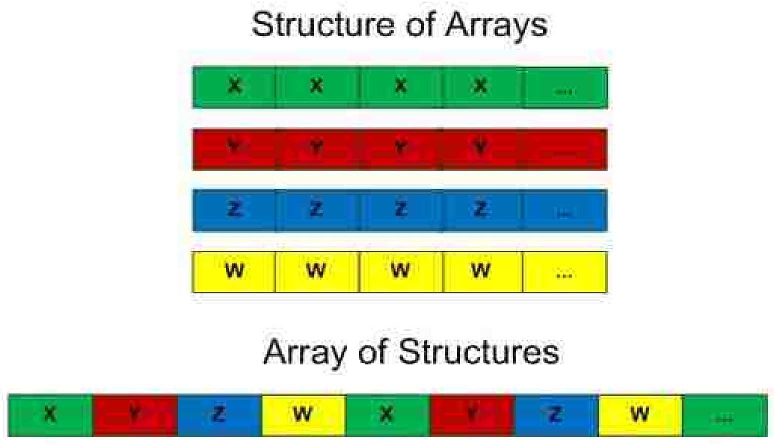


Figure 4.16: Array of Structures (AoS) and Structure of Arrays (SoA) representation inside memory [Intel, 2015b]. In AoS, elements related to a single data structure reside continuously in memory. In the SoA approach, for each field, there is an array in which data fields related to different structures reside continuously.

Hence, each element in an array belongs to a single particle (see Fig. 4.16).

The AoS approach is easier to use for development and expansion since each particle is represented by an independent object. In addition, when an object is accessed, all related data is accessible. Nevertheless, the AoS structures approach requires gather/scatter methods in order to get/set similar fields of various objects. It may introduce extra latency for memory accesses as well, since in order to access data elements, multiple jumps within memory space are unavoidable. Furthermore, non-continuous memory access may hurt SIMD performance, since several memory loads may be required in order to fill up a SIMD register.

The SoA approach can address performance issues related to the AoS approach. By maintaining separate arrays, each containing multiple data element, it is possible to preserve contiguous memory access so that accessing the data elements in memory and streaming memory to SIMD units remains continuous. As a result, with a single load it is possible to fill up a cache line or a SIMD register. Moreover, the problem of accessing an identical field for all objects is now the problem of accessing neighbor elements within a continuous array. As a result, the performance of the SoA approach can be significantly higher than the performance of the AoS approach. On the other hand, the SoA approach may introduce another issue as well. For instance, choosing the SoA approach may reduce locality for accessing multiple fields of the original structure instance which results in less flexible code. However, the significantly better performance of SoA vs. AOS make it the clear choice for HPC applications.

In XFLAT, all of the lower modules' data are structural as double precision arrays, where each element represents the wavefunction of a single neutrino. For instance, for the two-flavor system, since the neutrino's wavefunction has two complex components (Eq. 3.1), there are four double precision arrays representing the four wavefunction components. Therefore, there are two double precision arrays for the first complex number a, designated by `ar[], ai[]` (one for the real part, one for the imaginary part), and two double precision arrays for the second complex number b, designated by `br[], bi[]`. Accordingly, as opposed to creating one object per energy bin holding the double precision components, which would result in memory fragmentation and non-aligned memory access, only one object is created for the entire set of energy bins. Thus, contiguous aligned double precision data arrays are maintained for the wavefunctions' components (see Fig. 4.17), which results in performance improvement.

Since, in XFLAT the upper layer modules commonly have to maintain arrays of `NBeam` objects as well as functions (as they only encompass functions that perform computations on lower layer data), there is no requirement to instantiate the upper layer modules more than once. Consequently, they are not implemented as classes or arrays of classes. Instead, their functionalities are encapsulated in C++ namespaces. Thus the upper modules only maintain arrays of `NBeam` objects similar to the AoS approach, thus flexibility and modularity are preserved and there is no object creation

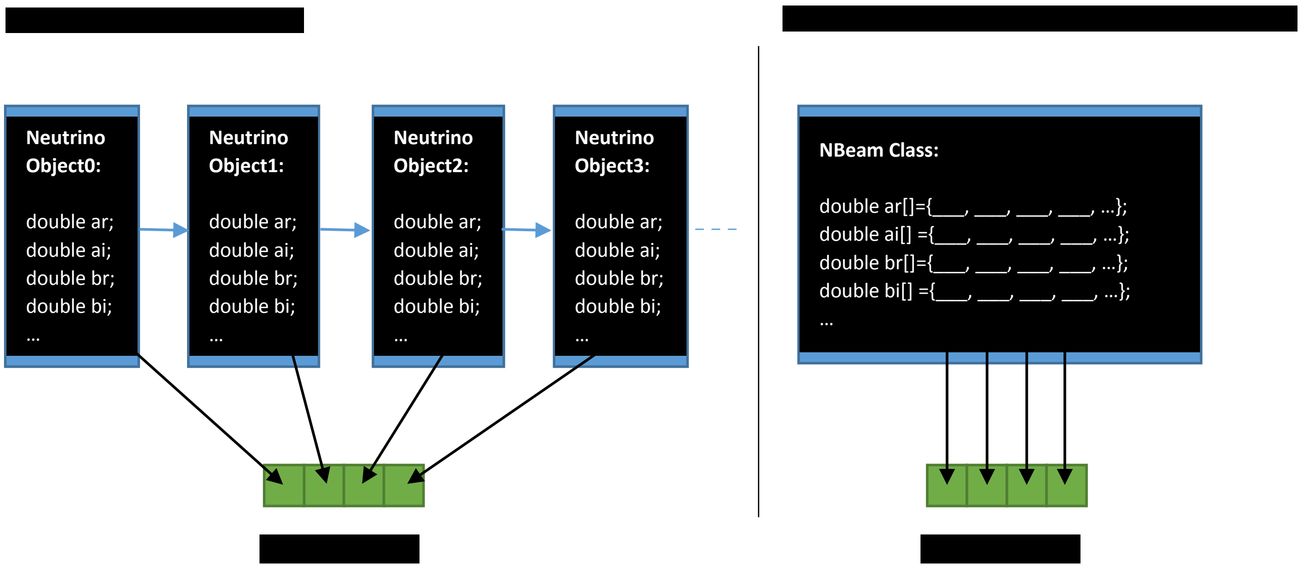


Figure 4.17: XFLAT data access pattern for AoS and SoA approaches. In the AoS approach (shown on the left), in order to fill up a SIMD register (shown in green), fragmented data must be extracted from multiple objects (shown in blue). On the other hand, in the SoA approach (shown on the right), multiple data can be loaded simultaneously into a register with a single continuous fetch.

overhead. Therefore, there is neither memory fragmentation nor non-aligned memory issues for the upper layers. Since XFLAT must utilize SIMD units of processor cores, the SoA approach was chosen for the NBeam class to make sure that the maximum performance can be achieved within each core.

4.4.2 Data Alignment

Data alignment (also known as *memory alignment*) denotes accessing the data at a memory offset equal to some multiple of word size. Memory alignment plays an important role in affecting both the cache hit rate and SIMD instruction performance in most modern processor architectures. The memory cells are read and written word by word. Each word contains a few bytes, typically four bytes (32 bits) or eight bytes (64 bits). Hence, by accessing a memory cell, more than a single byte is accessible. As a result, the number of memory fetches will depend on the way that the data reside in memory. Fig. 4.18 illustrates the difference between non-aligned and aligned memory. When data is aligned to the word size, data is accessible in a single fetch, while more than one fetch is required to access non-aligned data.

The data alignment role is critical inside cache memory and vector units. If the data is non-aligned to the correct alignment length, which implies that the starting point of the data array is not a multiple of a predefined word size (64 byte cache lines and vector registers), it can affect application performance. The reason is that the compiler has to fetch data from memory more than once in order to fill up a cache line. Multiple data fetches from main memory into cache is one of the causes of performance loss in processors due to the slowness of the main memory compared to that of cache memory. Likewise, in order to fill up the SIMD vector registers, if the data is not aligned to the vector registers' length, once more the compiler has to perform multiple fetches from memory in order to fill up the vector registers, which results in a major impact on the vectorization pipeline performance.

For the implementation of XFLAT, 64 byte alignment was chosen on both the Xeon CPUs and the Intel MIC. This was the logical choice since the lengths of the Intel MIC vector registers are 512 bits (64 bytes), and the cache line length on both the Xeon CPUs and the Intel MIC is also 64 bytes. Therefore the choice of 64 byte alignment is the most appropriate on both chips. This helps minimize memory fetch attempts and maximize the performance of the SIMD unit on both platforms.

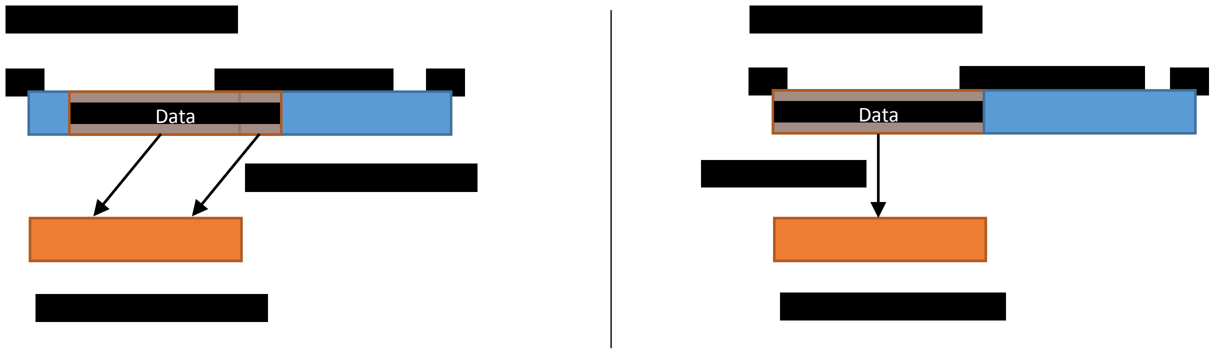


Figure 4.18: Illustration of the difference between non-aligned (left) and aligned memory (right) and impact on performance. If the data is non-aligned to 64-bytes memory boundaries (shown in blue), more than one fetch is required in order to extract the data from memory.

4.4.3 Inter-socket communication on Non-Uniform Memory Access domains

This technique only has an impact on Non-Uniform Memory Access (NUMA) multi-socket systems, as it is only associated with inter-socket memory access. Prior to development of the NUMA architecture, the most common shared memory architecture was Uniform Memory Access (UMA) in which all processors within a single node shared the physical memory uniformly. However, scaling the UMA architecture to a multi-socket configuration was hard and required complex hardware and sophisticated software in order to control and manage the memory access. In contrast, modern multi-processor systems have universally adopted the NUMA architecture in order to simplify the hardware and software architecture (see Fig. 4.19).

Nonetheless, the NUMA architecture has its own challenges. Virtually all modern operating systems (OS) do not allocate memory when allocation methods are called. In fact, operating systems allocate memory as soon as the first-touch happens to the memory, *i.e.* when a thread attempts to touch a part of the memory (read/write)

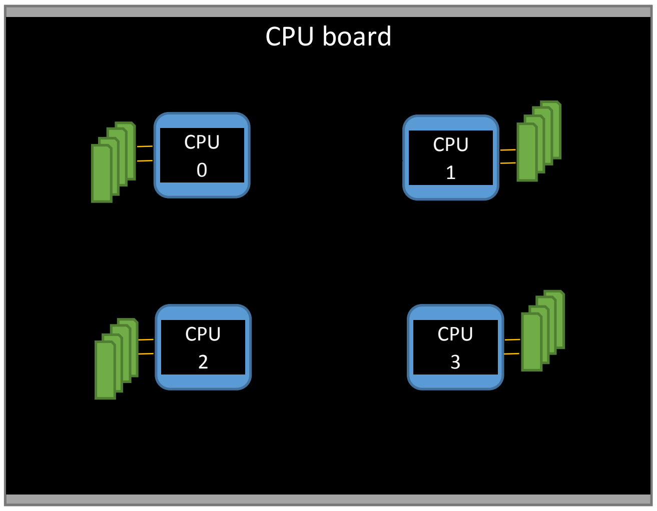


Figure 4.19: Non-Uniform Memory Access architecture. The memory (shown in green) connected to a particular CPU (shown in blue) is visible and accessible by threads on other CPUs through the QPI link (black lines).

for the first time. At that time, the OS allocates the memory such that it resides as close as possible to that thread. Therefore, although in multi-socket machines a single OpenMP process can access and manage all the hardware threads on all CPUs, if the memory allocation or the initialization section is completed by the master thread (which is typically the situation in many applications for simplicity), all memory is allocated in such a way that the access time of the master thread to the allocated memory is minimized. Hence, during execution, the rest of the threads on other processors must access memory through the CPU-CPU QPI bus, resulting in higher memory latency. For instance, in Fig. 4.20, if the thread on CPU0 tries to access to the memory that is controlled by CPU1, it has to access it through the QPI interconnection for which the latency is higher.

There are multiple approaches to resolve this issue. The first method is attempting to parallelize every first-touch memory access in the code by using multi-threaded

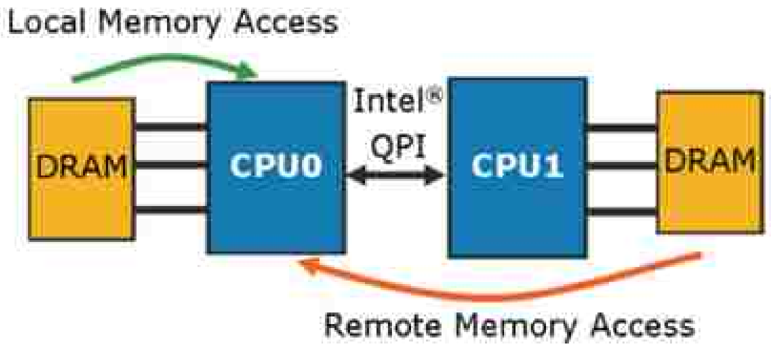


Figure 4.20: Illustration of accessing the memory (shown in yellow) in multi-socket CPU (shown in blue) architecture and the impact on the thread performance due to the route to the memory [SlideShare, 2015].

loops, which may not be feasible in every application and situation due to code dependencies in the initialization functions. The second solution is to eliminate the issue completely by maintaining independent MPI tasks on each CPU socket.

In XFLAT, the latter approach was adopted. Instead of allowing OpenMP to manage an entire multi-socket workstation or CPU board, one MPI process executes per socket. Thus, the communications of the OpenMP threads are internal to each CPU. Accordingly, inter-socket communications consist solely of the MPI messages that occur only at a few points per iteration. As a result, inter-socket OpenMP thread overhead is eliminated completely.

4.4.4 Fusing Functions

In modern processors, the role and performance of cache memory is important in overall application performance. As depicted in Fig. 4.21, the latency of cache memory is several orders of magnitude lower than that of main memory (RAM); however, the capacity of this special memory is limited compared to that of the main memory. Therefore, when a function's instructions, which are being executed on the processor, try to access data, the data latency varies based on the data location. If the required data can be found within the cache memory, the execution performance will be higher due to the higher speed of the cache memory. On the other hand, if the data is not found in cache, the processor must fetch the data from main memory, and due to main memory latency, CPU idle time may increase. Since the amount of available cache memory is limited and much less than that of main memory, it is not possible to fetch all of main memory into cache. Typically, only a limited memory section can be fetched into cache (see Fig. 4.22). The fetched memory is most closely related to the instructions that are being executed on the CPU. Consequently, when the function that is utilizing the CPU returns, its data is evicted from the cache in order to make room for new functions' data. Fetching memory into cache lines is an expensive and time-consuming task for the CPU. The situation can become worse if a subsequent function tries to work on the same set of data. As a result, the CPU must fetch the identical data, which it just evicted, from the main memory and into cache again.

There are numerous sections in a typical application where different functions perform calculations on the same set of data, and the functions are called continuously. Assume that the first function has been called, so that the required data must be fetched from main memory and into the processor's cache. If the data consist of a large array (most of the data in XFLAT is in this format), the data cannot stay inside the cache for a long period of time due to its size. Therefore, when the function completes its tasks and returns, the data may have been evicted from the cache lines. Afterwards, as soon as the second function is called, the same set of data must be fetched from the main memory once more. This extra memory fetch may have a significant impact on the application's performance. In order to eliminate this bottleneck, if possible, functions that are working on the same set of data should be fused together as a single multi-purpose function (see Fig. 4.23). In this manner, the required data for computations is only fetched once and all the function's instructions

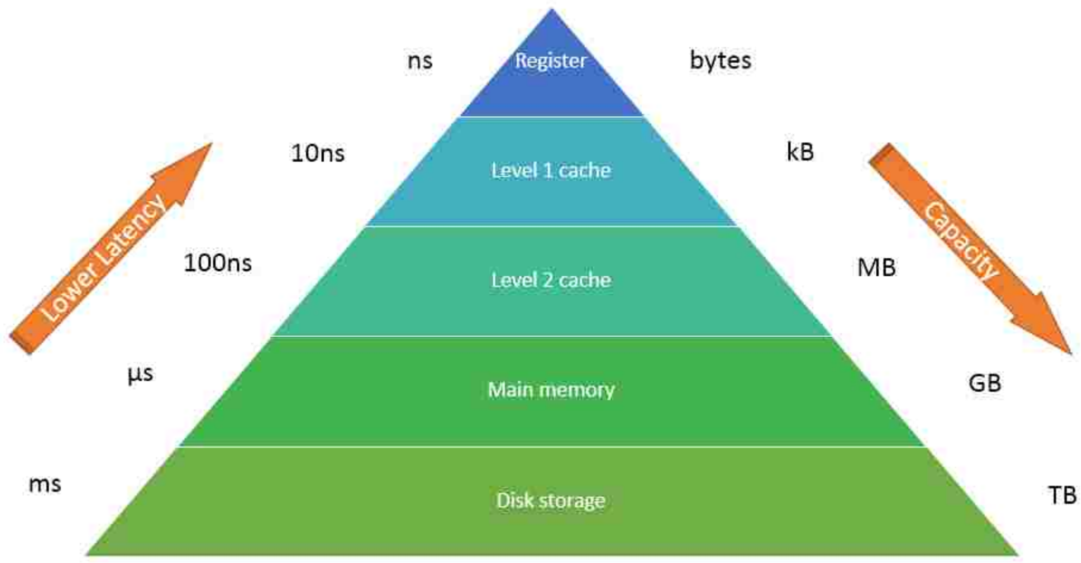


Figure 4.21: Illustration of the memory hierarchy on a modern compute node. CPU internal registers (shown at the top) have the highest speed and lowest capacity. Disk storage (shown at the bottom) has the highest capacity and slowest access time. All other memory levels reside in between.

can perform calculations on the cached data.

In XFLAT, in order to eliminate unnecessary memory fetches, and since several of the functions in the NBeam class work on the wavefunction components (the same data), several procedures of the NBeam class were fused together. Instead of calling several functions sequentially, a single fused function is called to perform the combined tasks. It is important to note that this fusing technique must be used with care, since there may be circumstances where only one of the functionalities of the NBeam class is required. In that situation calling a fused function will result in unnecessary computational overhead. Hence, in XFLAT, multiple API functions are provided in order to support a variety of situations including the fused functions that can be utilized only in the appropriate places.

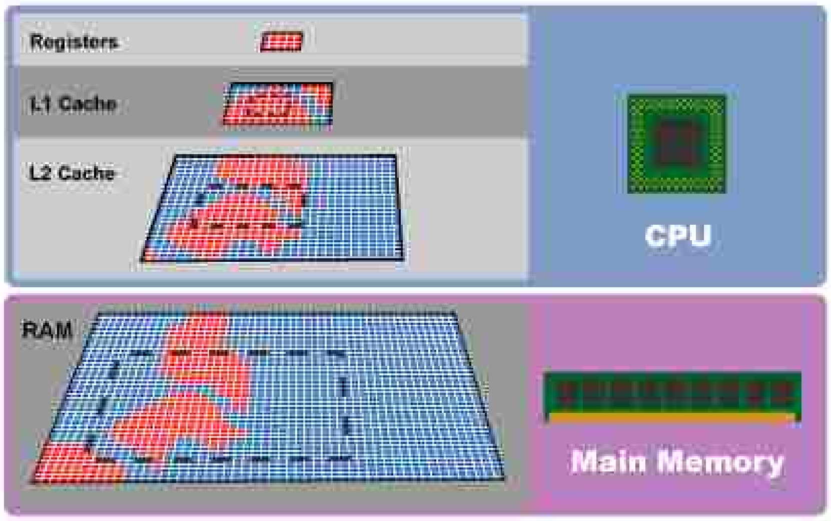
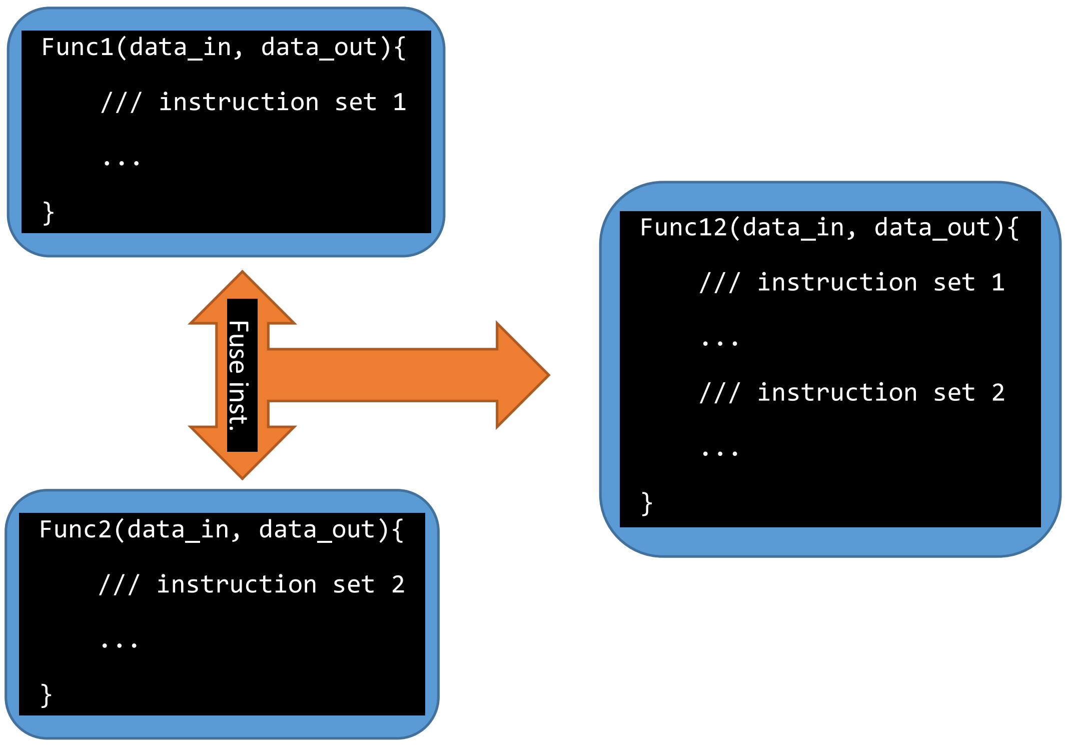


Figure 4.22: Caching pyramid illustration. For each memory fetch, the CPU fetches a region of memory containing the required data. If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future [ArsTechnica, 2015].

4.5 Code Validation

Prior to performance analysis, it is essential to perform code validation to ensure that a complete and correct algorithmic implementation is being analyzed. For XFLAT, this validation is performed against [Duan and Shalgar, 2014]. Although XFLAT supports a wide range of physics and geometries, since the previous code was developed only for the bulb model, this is the physics and geometry scenario used here.

In addition, an internal validation/consistency check at the end of the first step was performed, by computing the results of two of XFLAT's canonical modules: the bulb model and the extended bulb model. It is expected that the results of the multi-zenith supernova run (bulb model) will be very close to the results of the multi-azimuth multi-zenith run (extended bulb model) for the inverted mass hierarchy, *i.e.*,



```
Func1(data_in, data_out){
    /// instruction set 1
    ...
}
```

```
Func2(data_in, data_out){
    /// instruction set 2
    ...
}
```

```
Func12(data_in, data_out){
    /// instruction set 1
    ...
    /// instruction set 2
    ...
}
```

Figure 4.23: Fusing different functions that work on the same data. The instructions of the two functions on the left, in which the same set of data is processed within them, can be fused together to form the fused function on the right side.

$\delta m^2 < 0$. In that limit, the survival probability results for a particular neutrino are expected to be similar (The survival probability measures the percentage of neutrinos that remained in their initial flavors at a particular step). Hence, an experiment can start with a pure beam of known flavor ν_x, and at the end observe how many neutrinos have changed their flavors, and how many neutrinos have remained in their initial flavors.

The results of these two validation procedures (external and internal consistency checks) are described in the following two subsections.

4.5.1 Internal consistency check

XFLAT was run first with the bulb model (`Phy_MA.cpp`) module and then with the extended bulb model (`Phy_MAA.cpp`) module. For both runs the energy distribution function, $f_{\nu_\alpha}(E')$, was taken to be of the Fermi-Dirac form with two parameters (T_ν, η_ν),

$$f_{\nu_\alpha}(E') \equiv \frac{1}{F_2(\eta_\nu)} \frac{1}{T_\nu^3} \frac{E^2}{\exp(E/T_\nu - \eta_\nu + 1)}, \tag{4.1}$$

where η_ν is the degeneracy parameter, T_ν is the neutrino temprature, and

$$F_k(\eta_\nu) \equiv \int_0^\infty \frac{x^k dx}{\exp(x-\eta)+1}. \tag{4.2}$$

For all numerical calculations, unless stated otherwise, $\delta m^2 = -3 \times 10^{-3}$ eV^2, $\theta = 0.1$, $L_\nu = 10^{51}$ erg/s, $n_{b0} = 1.63{\times}10^{36}$ cm^{-3}, $L_{\nu_e} = L_{\nu_{\bar{e}}} = L_{\nu_x} = L_{\nu_{\bar{x}}} = 10^{51}$ erg/s, $\langle E_{\nu_e} \rangle = 11$ MeV, $\langle E_{\nu_{\bar{e}}} \rangle = 16$ MeV, $\langle E_{\nu_x} \rangle = \langle E_{\nu_{\bar{x}}} \rangle = 25$ MeV, $\eta_e = \eta_{\bar{e}} = \eta_x = \eta_{\bar{x}} = 3$. With these choices, $T_{\nu_e} \simeq 2.76$ MeV, $T_{\nu_{\bar{e}}} \simeq 4.01$ MeV, $T_{\nu_x} = T_{\nu_{\bar{x}}} \simeq 6.26$ MeV. The baryon density is:

$$\begin{aligned} n_b &\simeq \frac{2\pi^2}{45} g_s \left(\frac{M_{NS} m_N}{m_{PI}^2} \right)^3 S^{-4} r^{-3} \\ &\simeq \left(4.2 \times 10^{30} cm^{-3}\right) g_s \left(\frac{M_{NS}}{1.4 M_0} \right)^3 \left(\frac{100}{S} \right)^4 \left(\frac{10\text{ km}}{r} \right)^3, \end{aligned} \tag{4.3}$$

where m_N is the mass of a nucleon, $m_{Pl} \simeq 1.221 \times 1022$ MeV is the Plank mass S is the entropy per baryon and is set to $\frac{11}{2}$, r is the distance from the center of the proto-neutron star, $M_\odot$ is the solar mass, and $M_{NS} = 1.4 M_\odot$.

The number of angle beams was set to 800, the energy function range was $0 -$ 80 MeV and the number of energy bins was set to 160. For the extended bulb model, each zenith angle had 100 azimuth beams as well. The computations started at 50 km, and the survival probabilities are shown at 250 km.

Fig. 4.24 depicts the survival probabilities of electron neutrinos for the bulb model and extended bulb model runs. The vertical and horizontal axes correspond to zenith-

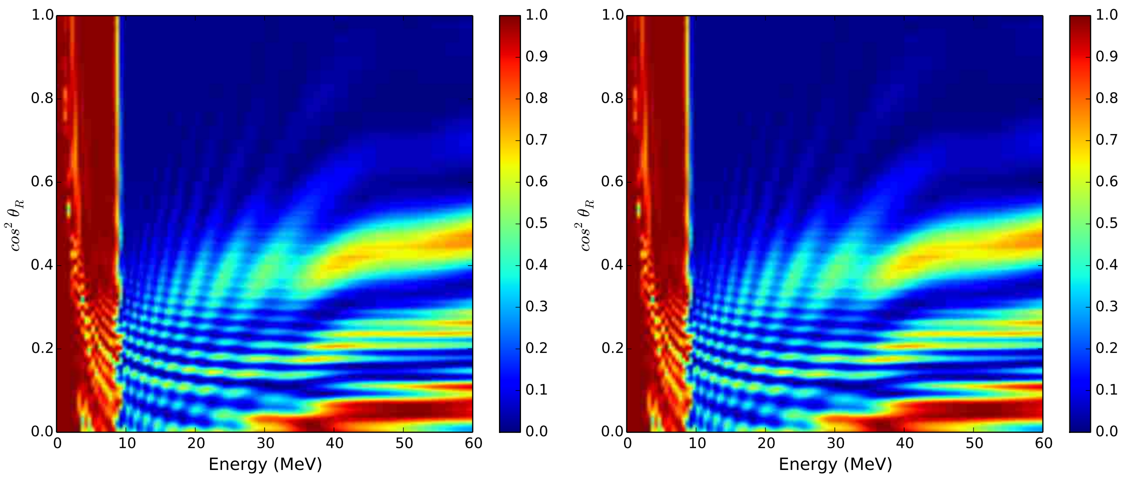


Figure 4.24: Survival probabilities of electron neutrinos for the bulb model (left) and extended bulb model (right), for the inverted mass hierarchy. The vertical and horizontal axes correspond to zenith-angle and energy, respectively. Red indicates a survival probability of 100% and blue, 0%.

angle and energy, respectively. Red corresponds to a survival probability of 100%, and blue indicates 0% survival probability. A plot of tha absolute difference is shown in Fig. 4.25. The maximum absolute difference between the two sets of survival probabilities is only 0.00029 (the values' range is [0, 1]).

Fig. 4.26 depicts the survival probability of an anti-electron neutrino for the bulb model and extended bulb model runs. In Fig. 4.27, the difference between the results of the bulb model and extended bulb model is depicted. Here, the absolute maximum difference is only 0.00049 (the values' range is [0, 1]).

The two runs (and different physics modules) produce visually identical results, with a maximum absolute difference of less than 0.05%.

While these results constitute one form of validation, this comparison can only be performed for the inverted mass hierarchy, since the two geometries only produce similar results in that limit. In contrast, and not surprisingly, for the normal mass

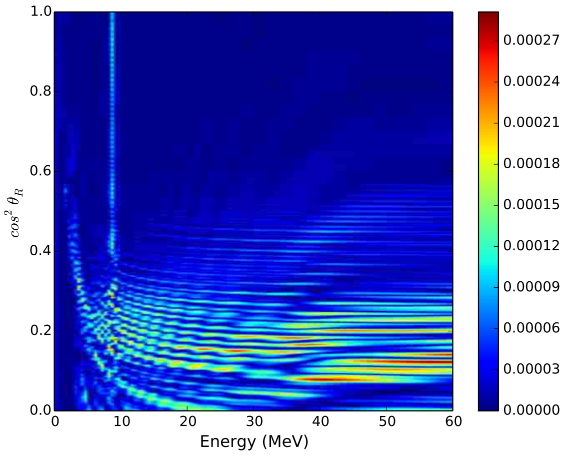


Figure 4.25: Difference between the bulb model and extended bulb model runs of Fig 4.24 for electron neutrino survival probability in the inverted mass hierarchy limit. The vertical and horizontal axes correspond to the zenith-angle and energy, respectively. Color scale representing magnitude is from blue (low) to red (high).

hierarchy the survival probability results for the two geometries, for the electron

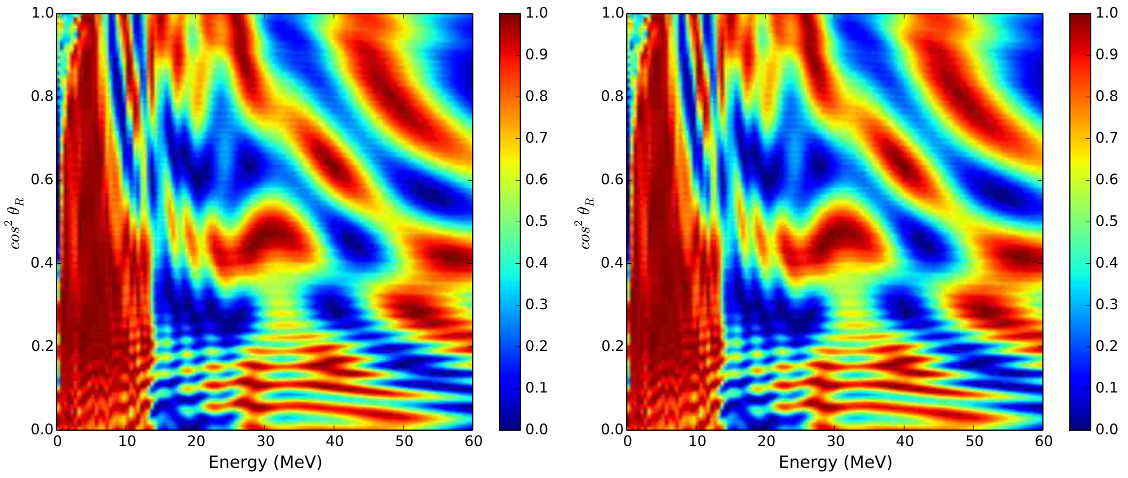


Figure 4.26: Survival probabilities of anti-electron neutrinos for the bulb model (left) and extended bulb model (right), for the inverted mass hierarchy. The vertical and horizontal axes correspond to the zenith-angle and energy, respectively. Red indicates a survival probability of 100% and blue, 0%.

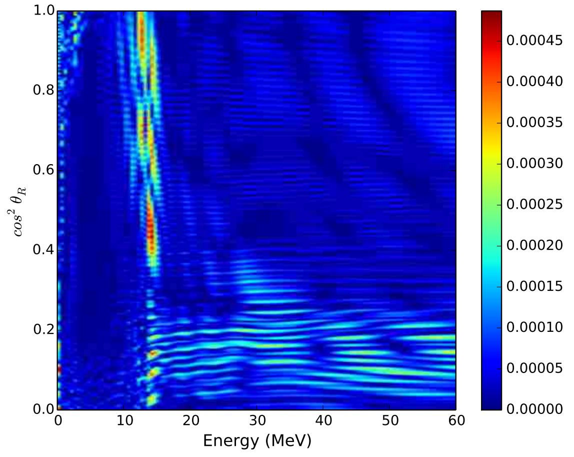


Figure 4.27: Difference between the bulb model and extended bulb model runs of Fig. 4.26 for anti-electron neutrino survival probability in the inverted mass hierarchy limit. Color scale representing magnitude is from blue (low) to red (high).

neutrino, are completely different (see Fig. 4.28).

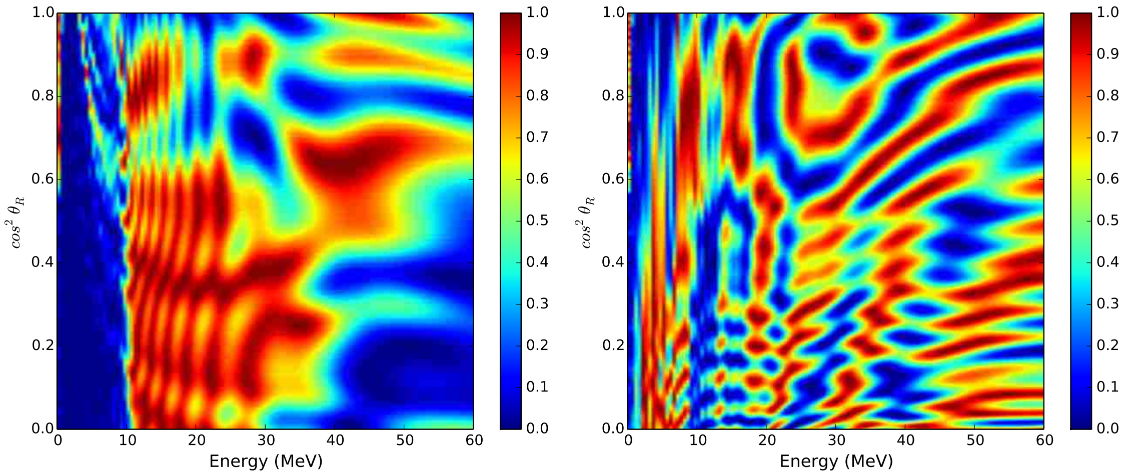


Figure 4.28: Survival probability of electron neutrino for the bulb model (left) and extended bulb model (right) in the normal mass hierarchy regime. The vertical and horizontal axes correspond to the zenith-angle and energy, respectively. Red indicates a survival probability of 100% and blue, 0%.

4.5.2 External code validation

The next step in the validation process was to compare the results of XFLAT against a previously-developed bulb model code [Duan and Shalgar, 2014]. While both codes were based on the formulation of [Duan et al., 2006] the code development paths were isolated from each other. The [Duan and Shalgar, 2014] code was developed previously and independently at Northwestern University. Thus, the algorithms, data structures, and numerical calculations are completely different between XFLAT and [Duan and Shalgar, 2014].

The simulation of the collective neutrino oscillations can be categorized into two classes, *single-split* and *multi-split* simulations [Dasgupta et al., 2009a]. The most important observational consequence of neutrinos collective effects is an exchange of the ν_e ($\bar{\nu}_e$) spectrum with the ν_x ($\bar{\nu}_x$) spectrum in certain energy intervals. Such a flavor exchange is called a "swap", whereas "splits" are sharp boundary features at the edges of each swap interval [Dasgupta et al., 2009b].

First, the results of single-split runs were compared. In order to produce the following results, the previously-mentioned parameters were employed, except for the number of angle beams which was now set at 2046, and the starting radius, which was set to 20 km. The survival probabilities are shown at 250 km. Fig. 4.29 depicts the results of the two codes, and Fig. 4.30 illustrates the absolute difference between the results of the two codes. In Fig. 4.31 the initial and final spectra of the XFLAT run and the absolute difference between the XFLAT and [Duan and Shalgar, 2014] runs are shown. The absolute difference of spectra between the two codes is $\sim 10^{-4}$, thus the two codes yield results that are virtually identical.

The first comparison was for a single-split energy spectrum. However, the single-split energy spectrum simulation is more stable than the multi-split energy spectrum simulation. Therefore, XFLAT and the [Duan and Shalgar, 2014] codes were also

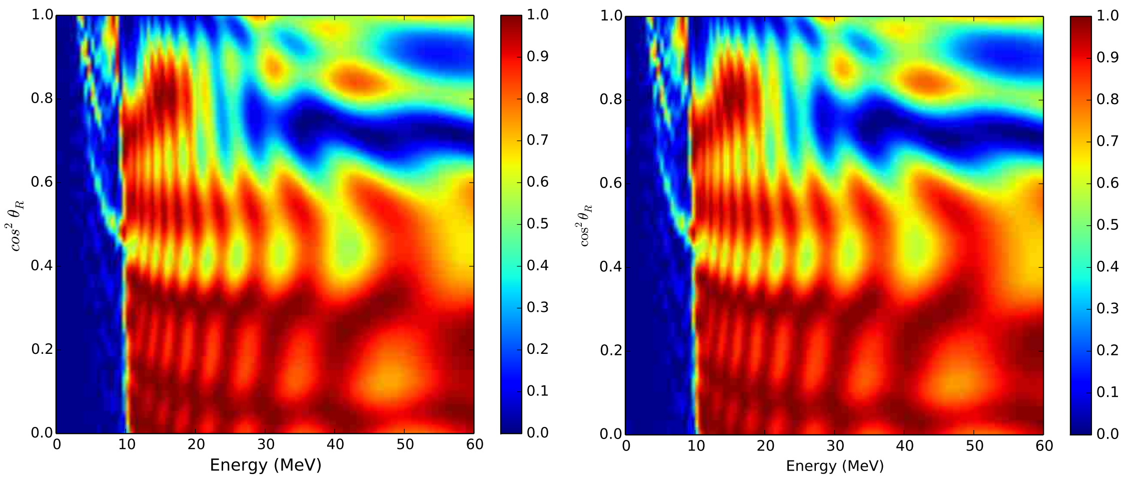


Figure 4.29: Survival probability of electron neutrinos for XFLAT (left) and [Duan and Shalgar, 2014] (right). The vertical and horizontal axes correspond to the zenith-angle and energy, respectively. Red indicates a survival probability of 100% and blue, 0%.

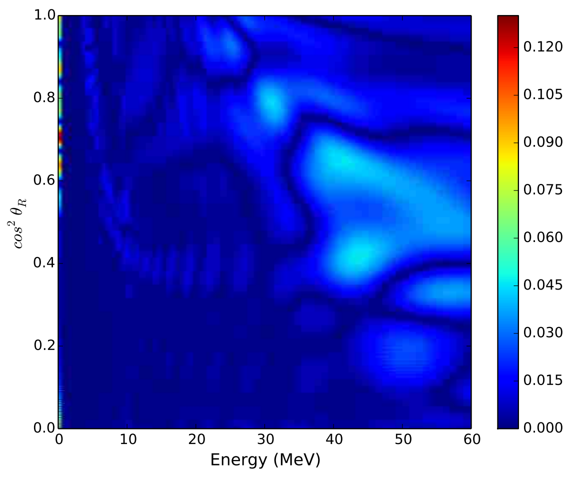


Figure 4.30: Absolute difference of survival probability of electron neutrinos computed in XFLAT and [Duan and Shalgar, 2014] for a single-split simulation. The vertical and horizontal axis show the zenith-angle beams and energy bins, respectively.

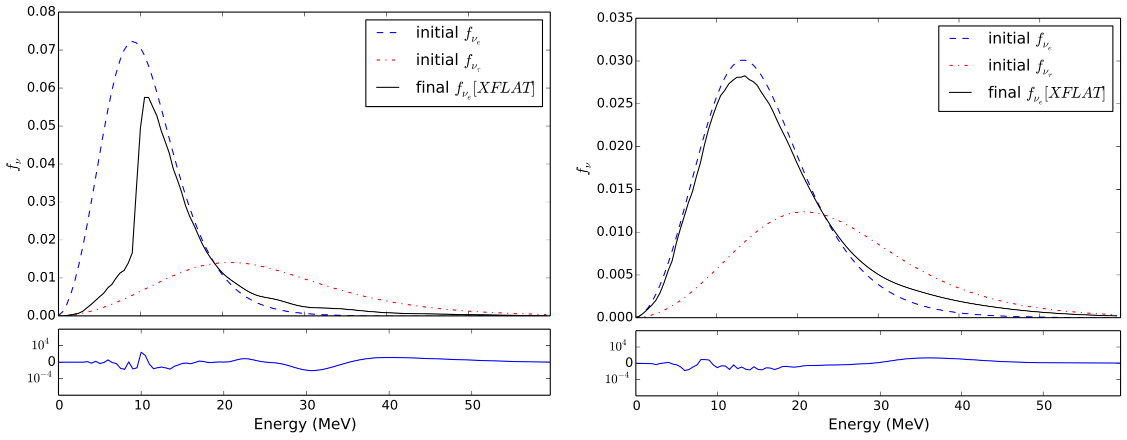


Figure 4.31: Change in neutrino (left) and anti-neutrino (right) energy spectra for single-split spectrum runs. The dashed and dot-dashed lines in upper panels are the initial spectra (f_ν) of the electron and tau neutrinos, respectively. The solid lines are the corresponding spectra at $r = 250$ km computed using XFLAT. The bottom panel shows the absolute differences between the XFLAT and [Duan and Shalgar, 2014] runs for the final spectra.

compared for a multi-split spectrum run as well. For all runs the parameters in Eq. 4.1 were taken as follow: $L_{\nu_e} = 4.1 \times 10^{51}$ erg/s, $L_{\nu_{\bar{e}}} = 4.3 \times 10^{51}$ erg/s, $L_{\nu_x} = L_{\nu_{\bar{x}}} = 4.1 \times 10^{51}$ erg/s, $\eta_e = 3.9$, $\eta_{\bar{e}} = 2.3$, $\eta_x = \eta_{\bar{x}} = 2.1$, $T_{\nu_e} \simeq 2.1$ MeV, $T_{\nu_{\bar{e}}} \simeq 3.4$ MeV, $T_{\nu_x} = T_{\nu_{\bar{x}}} \simeq 4.4$ MeV, the final radius was at 400 km and the number of zenith angles were set at 10000.

As illustrated in Fig. 4.32, there is good agreement between the results of XFLAT and [Duan and Shalgar, 2014] even in the more unstable multi-split run. In Fig. 4.33 the change in energy spectra (initial to final) and absolute difference in spectra for the neutrino (left) and anti-neutrino (right) for the multi-split spectrum runs are depicted. The absolute difference of the spectra in both cases, at the final radius ($r = 400$ km), is on the order of 10^{-4}.

For XFLAT the error tolerance was set to 10^{-10} and for the [Duan and Shalgar, 2014] code, in order to ensure convergence, the error tolerance was set to 10^{-11}. However,

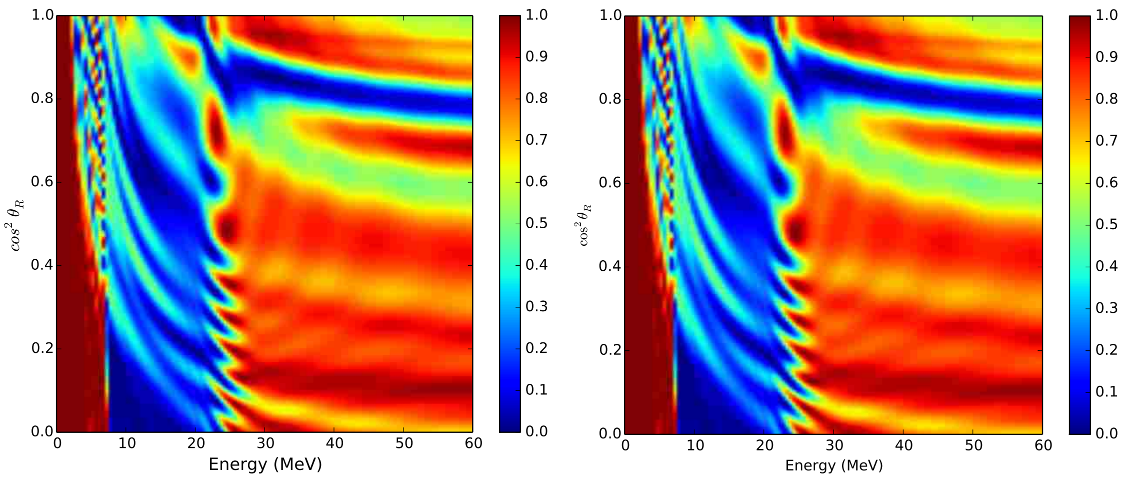


Figure 4.32: Survival probability of electron neutrino for XFLAT (left) and [Duan and Shalgar, 2014] (right) for multi-split spectrum runs. The vertical and horizontal axes correspond to the zenith-angle and energy, respectively. Red indicates a survival probability of 100% and blue, 0%.

the definition of error tolerance between the two codes is completely different. As a result, it is difficult to set the error tolerance to a comparable value for both codes. For XFLAT, choosing a lower error tolerance produced satisfactory results as well (see Fig. 4.34). As seen in Fig. 4.35, the absolute difference between the two XFLAT runs (with 10^{-8} and 10^{-10} error tolerance) was higher than the absolute difference between the XFLAT run (with 10^{-10} error tolerance) and the [Duan and Shalgar, 2014] run (with 10^{-11} error tolerance). The maximum absolute difference between the two XFLAT results is ~ 0.06 and the maximum absolute difference between the XFLAT and [Duan and Shalgar, 2014] results is ~ 0.04. Thus, the difference in results between XFLAT and the validation code is within the "convergence error bars" of XFLAT itself.

Finally, Fig. 4.36 depicts the survival probability for anti-electron neutrinos for XFLAT and [Duan and Shalgar, 2014]. As for the electron neutrino's plots, the results of the two codes are qualitatively similar. In addition, as shown in Fig. 4.37,

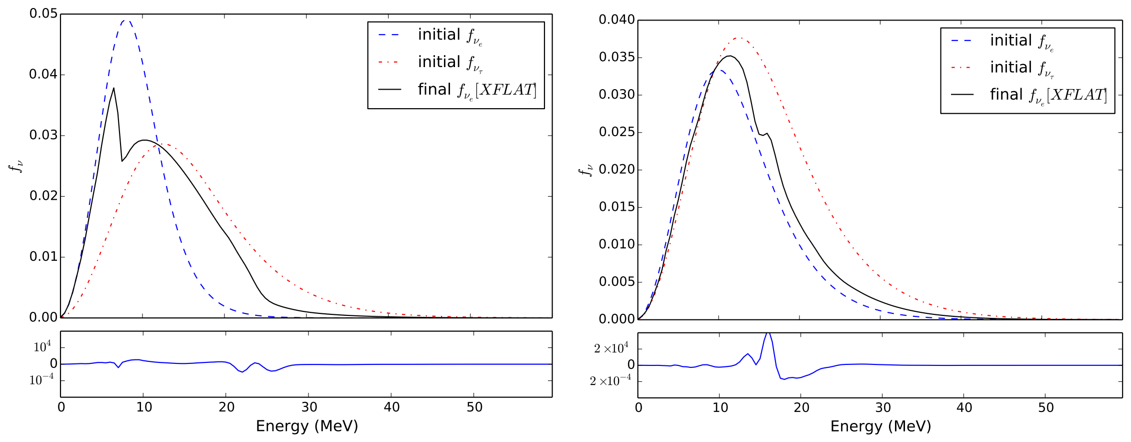


Figure 4.33: Change in neutrino (left) and anti-neutrino (right) energy spectra for multi-split spectrum runs. The dashed and dot-dashed lines in upper panels are the initial spectra (f_ν) of the electron and tau neutrinos, respectively. The solid lines are the corresponding spectra at $r = 400$ km computed using XFLAT. The bottom panel shows the absolute differences between the XFLAT and [Duan and Shalgar, 2014] runs for the final spectra.

the absolute difference between two XFLAT results with 10^{-8} and 10^{-10} error tolerance is similar to the difference between the results of XFLAT with 10^{-10} and [Duan and Shalgar, 2014] with 10^{-11} error tolerance.

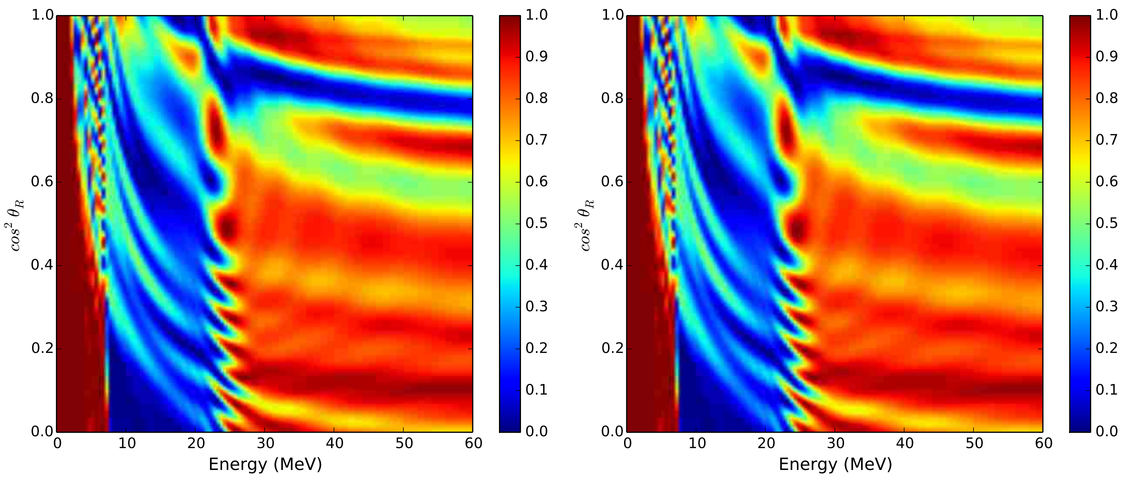


Figure 4.34: Survival probability of electron neutrino for XFLAT from multi-split spectrum runs with 10^{-8} (left) and 10^{-10} (right) error tolerances. The vertical and horizontal axes correspond to the zenith-angle and energy, respectively. Red indicates a survival probability of 100% and blue, 0%.

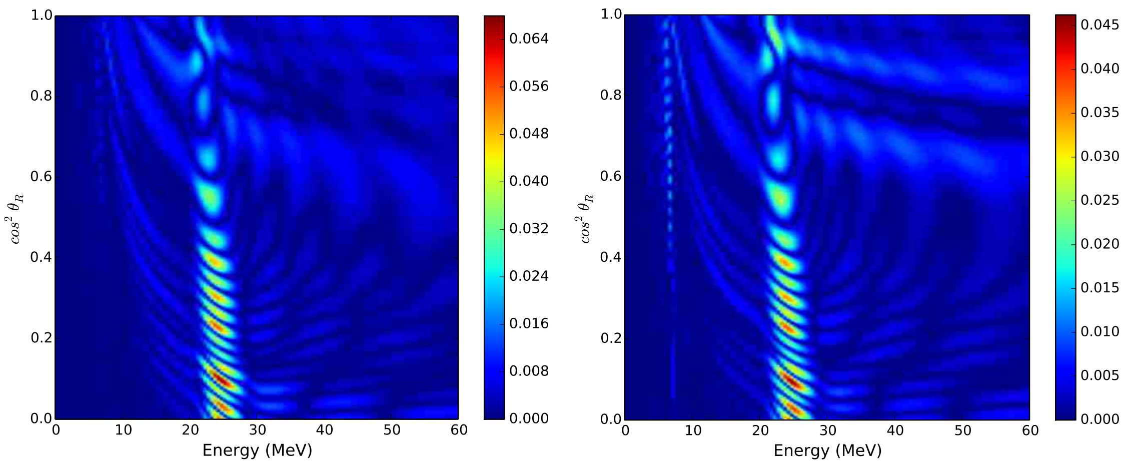


Figure 4.35: The absolute difference of the survival probability of electron neutrino between two XFLAT runs with 10^{-8} and 10^{-10} error tolerances (left), and between XFLAT with 10^{-10} error tolerance and [Duan and Shalgar, 2014] with 10^{-11} error tolerance (right). The vertical and horizontal axis show the zenith-angle beams and energy bins, respectively.

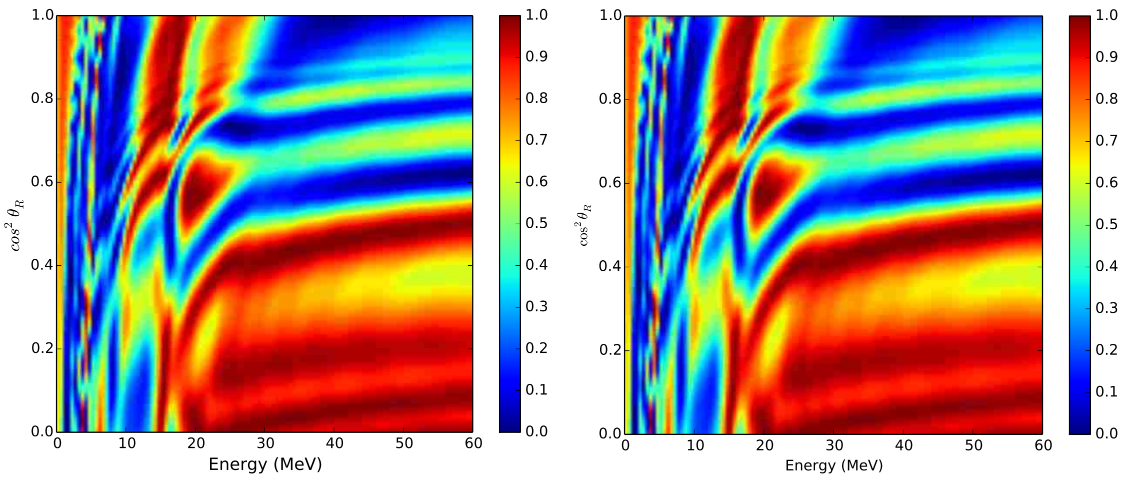


Figure 4.36: Survival probability of anti-electron neutrino for XFLAT from multi-split spectrum runs. The vertical and horizontal axes correspond to the zenith-angle and energy, respectively. Red indicates a survival probability of 100% and blue, 0%.

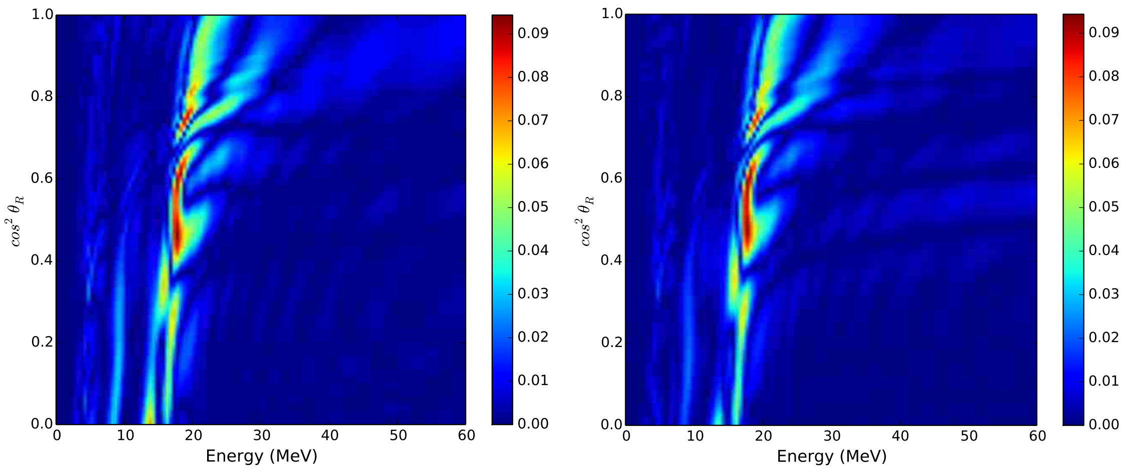


Figure 4.37: The absolute difference of the survival probability of anti-electron neutrino between two XFLAT runs with 10^{-8} and 10^{-10} error tolerances (left), and between XFLAT with 10^{-10} error tolerance and [Duan and Shalgar, 2014] with 10^{-11} error tolerance (right). The vertical and horizontal axes correspond to zenith-angle beams and energy bins, respectively.

Chapter 5

XFLAT Performance

5.1 Benchmarking and Performance Analysis

The performance of a parallel code on a new architecture can be studied via two approaches. The first approach is to create isolated benchmarks (kernels) in order to understand the intrinsic properties of the new platform through small, tightly-defined pieces of code. These kernels are typically chosen to be mathematical functions such as Fast-Fourier Transforms, that are used repeatedly in the code. The second approach is to study the overall performance of the newly developed code (performance analysis). By benchmarking standalone kernels, instead of the entire code, oportunities for performance improvements and potential sources of bottlenecks can be identified up front. Subsequently, benchmarking of the overall code can be used to identify key performance bottlenecks of the internal structures and algorithms of the code. Finding the root cause of performance bottlenecks plays a major role in optimizing a parallel code's efficiency.

A significant factor in parallel code development is the amount of speedup that may be obtained by running the code on parallel systems. In simple terms, *speedup* is defined as the relative performance improvement of a task when comparing two different runs of the identical code. This relative improvement is typically expressed as instructions per cycle (IPC). IPC is the average number of instructions executed per processor clock cycle. Instructions include the set of operation codes (opcodes) for a particular processor. The operation codes are those instructions that identify the operation to be performed on processor (*e.g.* multiplications and additions). Another way to define speedup is in terms of cycles per instruction (CPI), *i.e.*, the length of time between successive instruction completions. Formally, speedup is defined as:

$$S \equiv \frac{T_{old}}{T_{new}}, \tag{5.1}$$

where T_{old} is the old execution time without improvement, and T_{new} is the new execution time with improvement.

Linear speedup or *ideal speedup* is obtained when code speedup S_p on p processors is equal to p, *i.e.*, the scalability is perfect. For example, this implies that by doubling the number of processors, the speedup doubles as well. The other important metric that can be derived from speedup is the *efficiency* E_p which is defined as:

$$E_p \equiv \frac{S_p}{p}. \tag{5.2}$$

The efficiency value falls between zero and one, and reflects how well-utilized a parallel system is in solving a particular problem, compared to how much the time is wasted in communication and synchronization.

Unfortunately, the way that speedup and efficiency are defined make them inappropriate for measuring the efficiency of codes on muti-node systems with heterogeneous nodes. On a system with homogeneous nodes, there is only one type of processor as well as one type of available memory. However, those definitions are not directly applicable on a multi-node heterogeneous system, where the nodes are equipped with both CPUs and MICs. On a heterogeneous system, there are different processor types, each having different clock frequency and cache, various memory levels each with different bandwidth and latency, and multiple buses and I/O routes available. Hence, there may be numerous factors that can affect the overall performance of the system. As shown in this chapter, the performance of XFLAT turns out to depend on the MIC to CPU load ratio as well as the distribution of the load across the MIC's threads. When the optimum number of nodes is not known in advance, predicting expected speedup becomes challenging, and measured results may be difficult to understand and model. For instance, Fig. 5.1 illustrates multi-node XFLAT speedup relative to a single node (CPU only). On each node, there were two CPUs and two MICs. The benchmark measured the number of calculated radial steps for 100 seconds. When the number of nodes was less than 10, the 3:1 MIC to CPU load ratio resulted in the best performance. For instance, by employing 7 nodes and a 3:1 load ratio, the speedup was about 22 times higher than the single node result. However, when the number of nodes was chosen to be 14, the optimum load ratio was 2:1 and the relative speedup was 32 times higher than the single node result, rather than 44 (2×22). Hence, doubling the number of nodes did not result in a doubling of speedup, since on heterogeneous systems the performance depends on many other factors including load distributions and load ratios.

Likewise, the efficiency metric cannot be applied in its traditional definition since multiple types of processors with different capability are available at run time. At with speedup, the efficiency metric definition must be modified accordingly.

In summary, the speedup and efficiency metrics traditionally used to describe the scalibity of a code cannot be employed for performance analysis of hybrid codes on multi-node systems with heterogeneous nodes. On such systems, several factors including bus speeds, memory hierarchy, specifics of code work distribution across hardware compute elements, ... can affect the overall performance of a code, hence the code scalibility prediction cannot be achieved by exploiting traditional metrics.

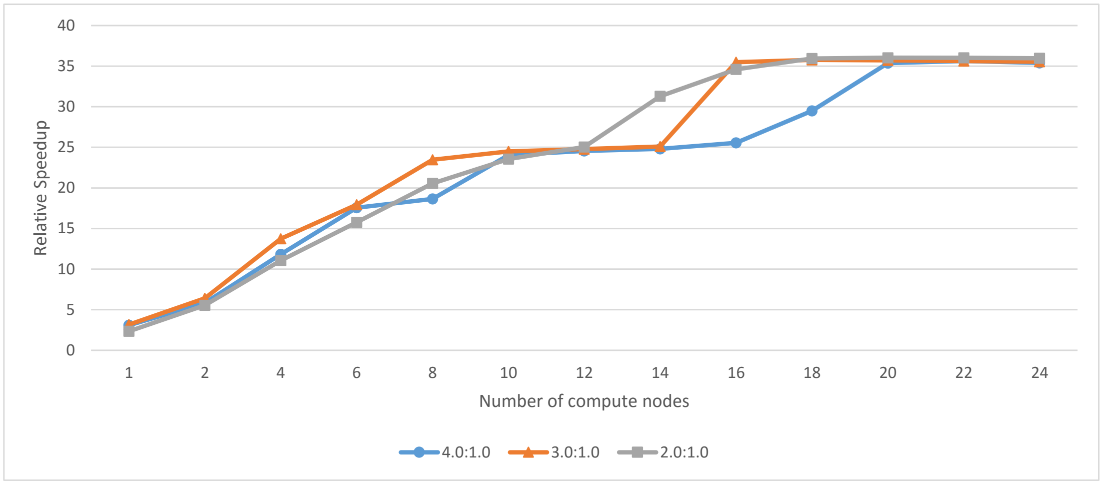


Figure 5.1: XFLAT speedup relative to single node (CPU only) on a multi-node environment. The blue, orange, and grey curves correspond to MIC to CPU load ratios of 4:1, 3:1, and 2:1, respectively.

In this work, new metrics are used in order to describe XFLAT behavior.

5.2 Kernel Benchmarks

Kernels are typically small codes corresponding to a single or limited set of mathematical operations, developed with the purpose of testing and analyzing performance of a given hardware/software platform. The performance analysis of large codes (*e.g.* XFLAT) may be hard, due to their being composed of many modules and parts. Each software module may have its own bottlenecks that can affect overall performance. Kernels are used to analyze each functionality or module by isolating and testing it as a standalone software elements. Hence, employing kernels for performance analysis is essential to the analysis of large codes.

As previously mentioned, there are several techniques to improve the performance of an HPC code such as XFLAT including data alignment, fusing functions, reducing inter-node communication, and changing loop structure. In addition, studying the performance of low-level details such as dereferencing pointers and transcendental functions can be helpful if these are used frequently. In this section, the performance impact of these techniques will be studied as independent kernels.

5.2.1 Floating point and transcendental function performance

One of the goals of kernel benchmarks is to measure the raw performance of each processor. Hence, kernels are needed in order to measure the performance of floating point operations as well as transcendental functions. XFLAT utilizes double precision floating point operations exclusively and it heavily employs transcendental functions as well. As discussed below, all double precision transcendental functions used in XFLAT were implemented in software. They exploit SIMD units for internal calculations, thus understanding their performance plays an important role in the performance analysis of XFLAT. In XFLAT, since the majority of the innermost loops were vectorized, all of the following benchmarks were performed inside vectorized loops (see Fig. 5.2) (Unless stated otherwise, all benchmarks in this chapter were performed on the Stampede system with hardware characteristics as summarized in Table 5.3). The widths of the vectors were taken to be a multiple of the SIMD registers width (256 bits or 4 DP (double precision) numbers for the Xeon CPU, and 512 bits or 8 DP numbers for the Xeon Phi). In order to maintain data locality, the same vector operations were repeated ten million times inside the middle loop. Furthermore, in the outermost loop, all of the hardware threads were utilized in order to achieve the best performance.

Three similar kernels were prepared, corresponding to the three different mathematical operations to be benchmarked. The only variable in the kernels was the length of the vectorized loop that varied from 8–4096 double precision elements. The first kernel benchmarked simple floating point operations (additions and multiplications), thus one double precision addition and one double precision multiplication were performed per iteration. In the second kernel, one `sin()` and one `cos()` calculation in double precision were performed per loop iteration. The third kernel benchmarked the performance of the double precision version of the `exp()` function.

As illustrated in Fig. 5.3, the Xeon Phi can reach over 1 TeraFlops (1000 GigaFlops) as advertised in [Jeffers and Reinders, 2013]. However, the performance of the Xeon Phi is highly sensitive to the width of the vector. On the other hand, the performance of the CPU is relatively stable as a function of vector length. The floating point performance of the Xeon Phi is best when the width of the DP vectors is 64; in that case, the Xeon Phi ran 10 times as fast as the CPU. However, the performance of the Xeon Phi degraded substantially as the vector width increased.

The results of transcendental function benchmarks are shown in Fig. 5.4. These results suggest that transcendental function performance on the Xeon Phi is relatively stable with respect to the width of the vectors. For the tests on `sin()` and `cos()` the Xeon Phi ran 6–8 times as fast as the CPU, but for `exp()` the Xeon Phi was

```
// Two transcendental operations per iteration

// Outer loop
for (...)
{
    // Vectorized loop
    #pragma omp simd
    for (int i : ARRAY_LENGTH)
    {
        A[i] = sin(A[i]);
        B[i] = cos(B[i]);
    }
}
```

Figure 5.2: Illustration of vectorized loop structure for transcendental function benchmark.

only 3–4 times better.

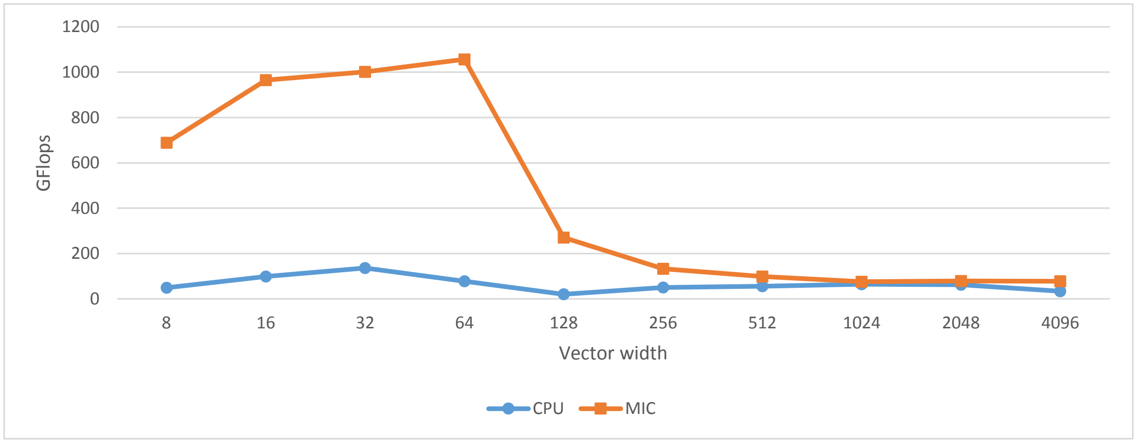


Figure 5.3: Floating point performance of CPU and MIC (Xeon Phi) for various vector widths. The performance of the Xeon Phi (orange curve) degraded significantly where the vector width was increased beyond 64; however, the CPU performance (blue curve) remained constant. Benchmarks performed on Stampede system with hardware characteristics as summarized in Table 5.3.

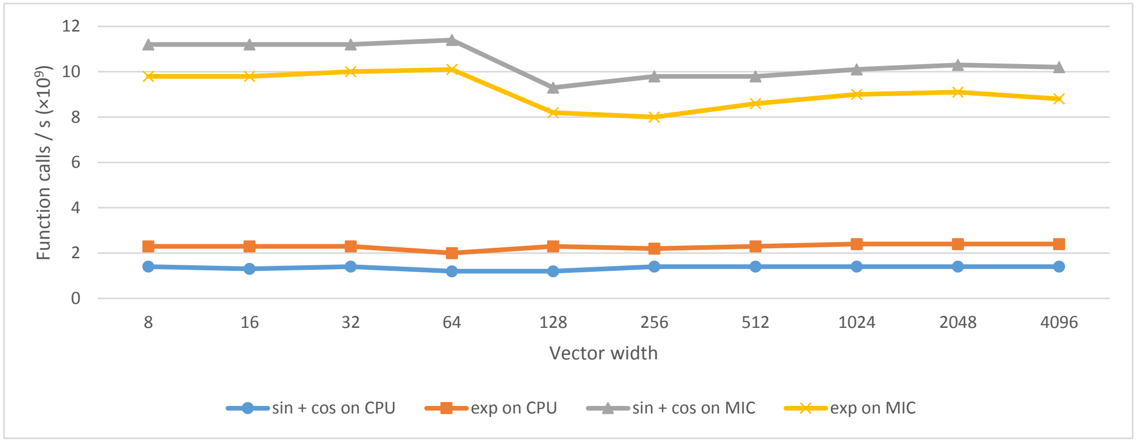


Figure 5.4: Transcendental function performance of CPU and MIC (Xeon Phi) on Stampede (see Table 5.3). For the tests on `sin()` and `cos()` the Xeon Phi (grey curve) ran 6–8 times as fast as the CPU (blue curve), but for `exp()` the Xeon Phi (yellow curve) is only 3–4 times better (orange curve).

Since the observed performance drop may have been due to the loop structure, alternative benchmarks with different loop structures were performed on the Xeon Phi. In addition, the compiler was upgraded from Intel C++ 13.1 to recently installed Intel C++ 15.0 on the TACC Stampede supercomputer (see Table 5.3 for hardware specifications). Fig. 5.6 illustrates the performance of floating point operations (additions and multiplications) on Stampede's Xeon Phi. The difference between the new and old kernels is that unlike the previous kernel, within the new kernel's loop, there were two and four independent array operations per iteration, respectively (see Fig. 5.5). These results confirm that by going beyond a length of the 64 double precision numbers, the performance of the Xeon Phi drops dramatically. This issue appears to be fundamental to the current generation of Xeon Phi's software or hardware architecture.

Fig. 5.7 illustrates the performance of `sin()` and `cos()` when are were four array operations within the vectorized loop. As is evident, by changing the compiler version and loop structure, the overall Xeon Phi performance was still constant with respect

```
// Two transcendental operations per iteration

// Outer loop
for (...)
{
    // Vectorized loop
    #pragma omp simd
    for (int i : ARRAY_LENGTH)
    {
        A[i] += x1 * B[i];
        C[i] += x2 * D[i];
        E[i] += x3 * F[i];
        G[i] += x4 * H[i];
    }
}
```

Figure 5.5: Illustration of vectorized loop structure with four array operations per iteration.

to the previous results. This shows that the observed performance issue was not due to the compiler used.

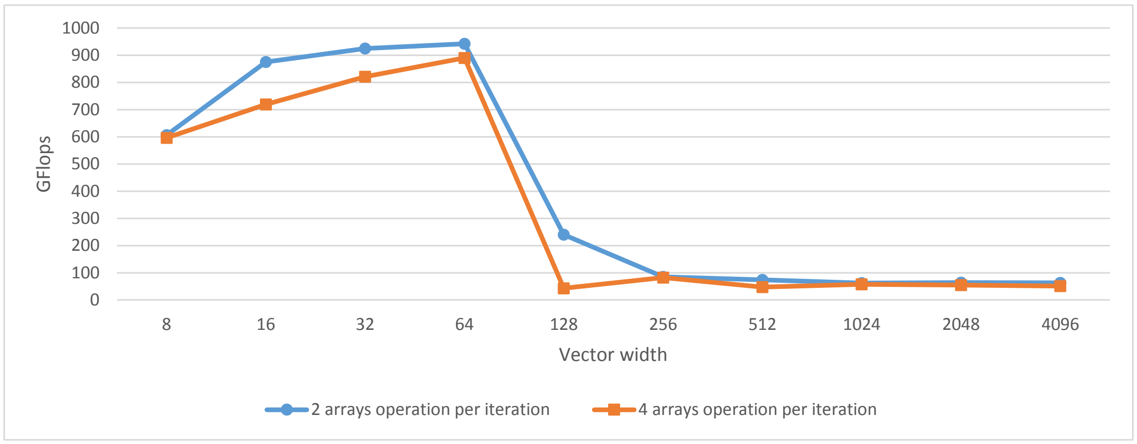


Figure 5.6: Performance of floating point operations on CPU and MIC (Xeon Phi) on Stampede. The blue and orange curves illustrate benchmarks with two and four independent lines of array operations, respectively.

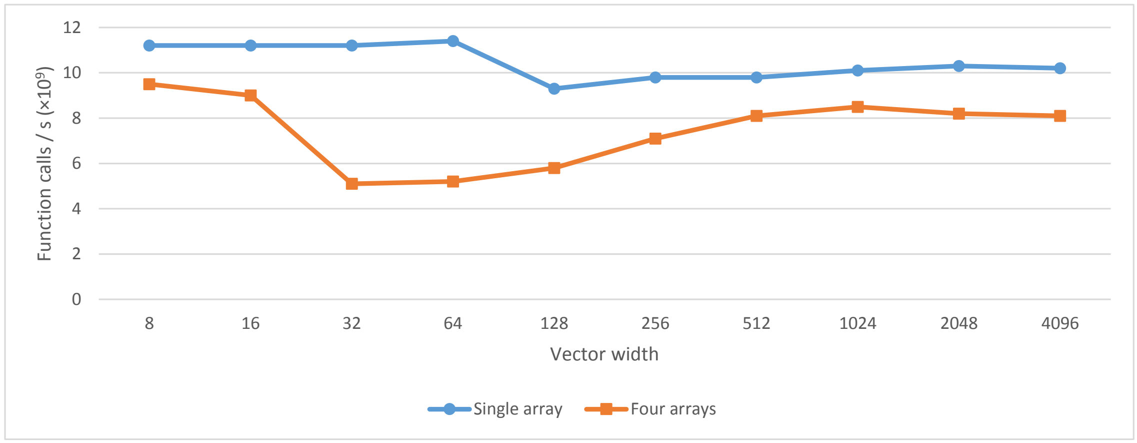


Figure 5.7: Transcendental function performance of CPU and MIC (Xeon Phi) on Stampede. The blue and orange curves illustrate benchmarks with one and four independent lines of array operations, respectively.

The performance of the transcendental functions within parallelized and vectorized loops may depend on another factor as well. As previously mentioned, all of the double precision versions of the transcendental functions were implemented in software, thus the SIMD units are also utilized internally for the underlying transcendental calculations. This means that the benchmarks' loop as well as the functions within those loops utilize the SIMD units. Depending on the employed compiler flags, for transcendental functions the internal employed function calls within the SIMD units may be different. This was confirmed by examining the generated assembly code of a built kernel using different compiler flags. For instance, there can be multiple internal function calls depending on the compiler flags for the kernel that utilized the `sin()` and `cos()` functions. Fig. 5.8, Fig. 5.9, Fig. 5.10, and Fig. 5.11 show the internal called functions that were employed when the code was compiled with `-S -no-vec`, `-S -no-vec -xHOST`, `-S`, and `-S -xHOST` flags respectively. The compiler `-S` flag generates the assembly file output, the `-no-vec` flag prevents loops from being vectorized, and the `-xHOST` flag forces the compiler to generate AVX (see Ch. 2) vector specific-instructions. Unlike SSE (see Ch. 2) instructions (and its later

```
..B1.33:
        movsd     40(%rsp), %xmm0
        movl      %r12d, %r14d
        addsd     .L_2il0floatpacket.88(%rip), %xmm0
        movsd     %xmm0, 40(%rsp)
        call      __libm_sse2_sincos
```

Figure 5.8: Kernel code compiled with `-no-vec` flag. The employed transcendental function is `__libm_sse2_sincos` in which one `sin()` and one `cos()` are calculated.

versions SSE2, SSE3, SSSE3, SSE4, *etc.*) on the previous generations of CPU on which two double precision floating point numbers per instruction can be handled, AVX instructions can perform a single instruction on four double precision floating point numbers simultaneously. As can be seen in the generated assembly code, even when the compiler is forced not to vectorize the loop, it still calls the SSE2 version of a function to calculate `sin()` and `cos()` simultaneously. Even the `-S -xHOST` flag cannot change the utilized function when the compiler is forced not to generate a vectorized loop. When the no-vectorization restriction is lifted, normally the compiler will pack `sin()` and `cos()` functions together and compute them together. Only by forcing the compiler to utilize the AVX registers, can it be seen that the `sin()` and `cos()` functions are employed separately to pack four elements together and issue the similar instructions on them simultaneously.

As a result, in a vectorized loop, the transcendental functions should always be employed with caution, since depending on the loop's structure and compiler flags, the underlying function calls may be different.

```
..B1.33:
        vmovsd    40(%rsp), %xmm0
        movl      %r12d, %r14d
        vaddsd    .L_2il0floatpacket.88(%rip), %xmm0, %xmm0
        vmovsd    %xmm0, 40(%rsp)
        call      __libm_sse2_sincos
```

Figure 5.9: Kernel code compiled with `-no-vec` and `-xHOST` flags. The employed transcendental function is `__libm_sse2_sincos` in which one `sin()` and one `cos()` are calculated.

```
..B1.33:
        xorl      %r12d, %r12d
        addsd     %xmm8, %xmm9
        movaps    %xmm9, %xmm0
        unpcklpd  %xmm0, %xmm0
        call      __svml_sincos2
```

Figure 5.10: Kernel code compiled without using any of `-no-vec` and `-xHOST` flags. The employed transcendental function is `__svml_sincos2` in which two `sin()` and two `cos()` are calculated.

5.2.2 Structure of Arrays (SoA) vs. Array of Structures (AoS)

At mentioned earlier, there are two general approaches for developing codes in which computations are performed on arrays of data. The first and more convenient approach is to build a structure that encapsulates a group of variables, afterwards allocating memory for an array of those structures. For example one can design a neutrino class that encapsulates the flavor state of a single neutrino. Next, for each energy and angle beam, one instance of the object is allocated. This approach is more straightforward to implement and easier to expand. However, for performing the same operations on components of all neutrinos, data must be fetched from non-

```
..B1.33:
        xorl      %ebx, %ebx
        vaddsd    %xmm8, %xmm9, %xmm9
        vmovddup  %xmm9, %xmm1
        vinsertf128 $1, %xmm1, %ymm1, %ymm11
        vmovaps   %ymm11, %ymm0
        call      __svml_cos4

..B1.57:
        vmovapd   %ymm0, %ymm10
        vmovaps   %ymm11, %ymm0
        call      __svml_sin4
```

Figure 5.11: Kernel code compiled with `-xHOST` flag only. The employed transcendental functions are `__svml_cos4` and `__svml_sin4` in which four `sin()` and four `cos()` are calculated independently.

continuous and sparse locations into vector registers (see Fig. 4.17). As a result, the amount of memory fetch and the latency increase per cycle.

On the other hand, if a structure can encapsulate a continuous range of data, the compiler can provide continuous streams of data in order to utilize SIMD registers efficiently. As an example, in XFLAT an instance of the neutrino class encapsulates arrays of neutrino flavor states for which each element represents a particular energy. Therefore, an object encompasses a range of neutrino flavor states. As a result, in order to perform the same computations on the neutrino wavefunction's components, they can be fetched and stored into SIMD registers continuously.

In order to study the performance impact of the SoA and AoS approaches, two kernels were developed. The first one contained a class in which four arrays were defined, thus the design approach was SoA. The length of the arrays was 100 double precision numbers, and 1000 instances of the class were created. For the second kernel, a single class was developed in which four single variables were encapsulated, thus the design approach was AoS. Afterwards, 100k instances of the class were

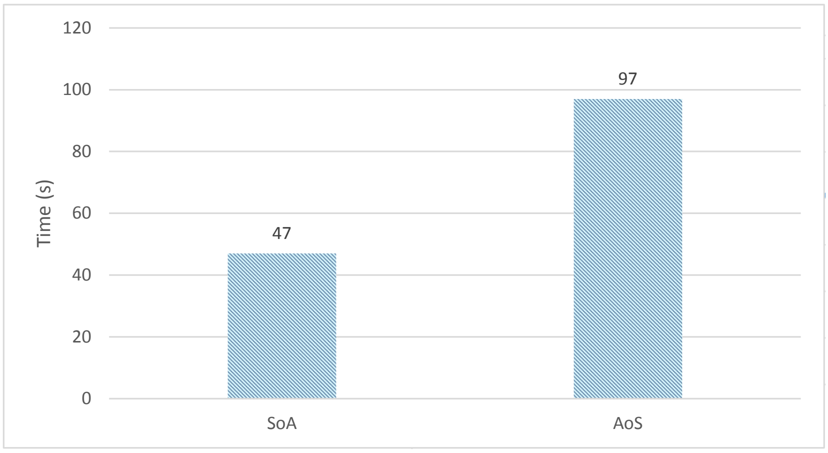


Figure 5.12: Performance comparison of two general designs: Structure of Arrays (SoA) vs. Array of Structures (AoS). Timing on Stampede machine (see Table 5.3).

allocated. Simple floating point calculations were performed on both kernels' data for a 1M iteration count. The results of the benchmark on a single CPU on Stampede machine (see Table 5.3) are shown in Fig. 5.12.

One can see that even for simple floating point operations, the performance of the SoA approach is about twice that of the AoS approach. Although the performance gap may vary based on specifics of the loops' structure and length of arrays, for high-performance applications it is clear that the SoA approach is preferable. For general (non-HPC) applications, AoS is prefered since it is easier to implement.

5.2.3 Dereferencing pointers inside vectorized loops

During the development of XFLAT, several performance fluctuations were observed on the Xeon Phi platform on the Stampede machine. One peculiar issue was related to dereferencing pointers within a vectorized loop and the way that arguments were passed to a function. The NBeam class contains several vectorized loop inside which function calls are performed. Thus, pointers and data are passed to the called functions. One important function, which is called frequently, is the neutrino evolution function that loops over neutrino bins, evolves neutrino flavor states, and saves the results into another neutrino array. Another frequently called function is the density function that receives an array of neutrino flavor states and computes the density matrix for each wavefunction. Both functions are called within vectorized loops, and the loops perform reduction operations on arrays.

In order to identify the root cause of the performance issue, two kernels with similar behavior to relevant sections of XFLAT were developed. The first kernel contained a class that implements a function in which the same instructions as the evolution function in XFLAT were utilized. The second kernel's class had a function similar to the density method in XFLAT with the same set of instructions. There were five ways to call, pass arguments, and utilize the evolution function as shown in Fig. 5.13. Likewise, there were four ways to call, pass arguments, and utilize the density function as illustrated in Fig. 5.14.

Fig. 5.15 illustrates the performance of the five evolution functions on the CPU as well as on the MIC. As can be seen, the performance on the MIC varied significantly across the five methods. The performance of the first method on the MIC was more than twice that of the performance of the fifth method. In addition, Fig. 5.16 depicts the performance of the density functions on the CPU and the MIC. Once more, the performance on the MIC fluctuated by a large margin.

The best performance was achieved when the pointers were dereferenced before each vectorized loop. In that case, the compiler can generate better optimized code. However, on the CPU, this behavior was not observed. In XFLAT, in order to achieve the highest possible performance, all pointers were therefore dereferenced before vectorized loops.

```
// The input is passed to the class by object reference
// The output is saved to the class array member
for (index: ARRAY_LENGTH)
{
    //1) both input and output are explicitly passed as arguments
    evolve1(input[index], output[index], ...);

    //2) the output is an array's member
    // thus, it is accessible within the function
    evolve2(index, input[index], ...);

    //3) the input is passed via a reference to the object's function
    // within the function the input arrays are accessed directly
    evolve3(&object_input, index, ...);

    //4) similar to 3) but first the input array is dereferenced;
    // thus, the input data access are performed via simple pointers
    evolve4(&object_input, index, ...);

    //5) similar to 4) but first the output arrays are dereferenced
    // so member array are accessed via simple pointers
    evolve5(&object_input, index, ...);
}
```

Figure 5.13: Different approaches for passing arguments and calling the evolution function. Within the function elements are accessed from inside a vectorized loop.

5.2.4 Calling functions and their arguments

In the previous section, the body of the functions contained complex instructions (similar to those appearing in the XFLAT code). Performing complex instructions can affect overall timings. Therefore, a simpler kernel was designed in order to isolate and study the effects of different ways of sending arguments to functions. Within the kernel a single simple function was called from inside a vectorized loop using two approaches. Fig. 5.17 illustrates these two approaches. The first method passes the element's index to the function. Within the functions, the received index is employed

```
// The arrays are member of the class
for (index: ARRAY_LENGTH)
{
    //1) the data arrays are dereferenced and used within the method
    density1(index, ...);

    //2) the arrays' elements are accessed directly for calculations
    // without any prior dereferencing
    density2(index, ...);

    //3) the data arrays are passed directly to the function
    density3(output[index], ...);

    //4) the data arrays are dereferenced before calling the function
    density4(output[index], ...);
}
```

Figure 5.14: Different approaches for passing arguments and calling the density function. Within the function elements are accessed from inside a vectorized reduction loop.

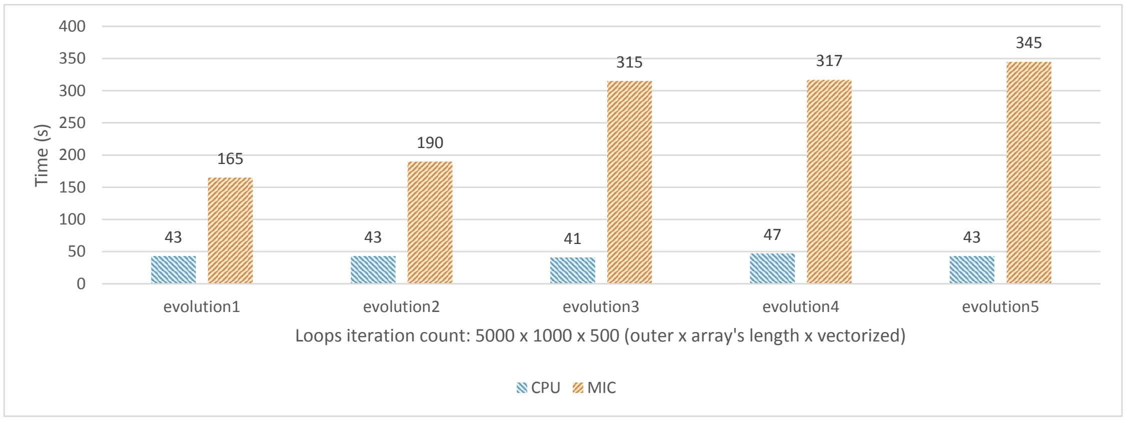


Figure 5.15: Performance comparison of the evolution functions on the CPU and the MIC on Stampede. On the MIC, the results (orange bars) fluctuated, whereas on the CPU, the results (blue bars) were steady.

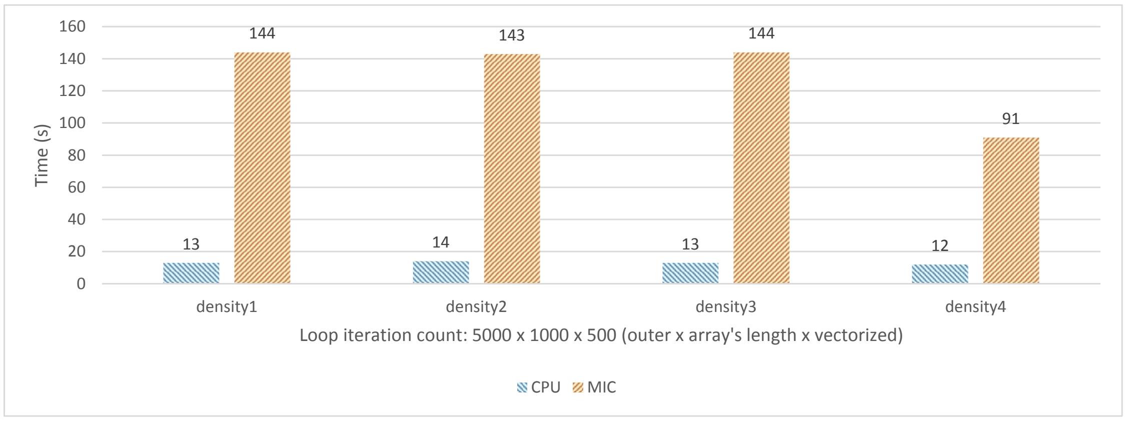


Figure 5.16: Performance comparison of the density functions on the CPU and the MIC on Stampede. On the MIC, the results (orange bars) fluctuated, whereas on the CPU, the results (blue bars) were steady.

to look up array elements. The second method passes each element as a separate argument. In principle, the performance of the two methods should not differ much. Nevertheless, the compiler could conceivably generate more optimized code for the second approach.

The kernel had two nested loops: an outer loop which simply repeated the inner loop, and an inner loop that was responsible for looping over array elements and

```
#pragma omp simd
for (int index : ARRAY_LENGTH)
{
    /// Only one of the following is called each time!
    func1(index, &results);
    func2(ar[index], ai[index], br[index], bi[index], &results);
}
```

Figure 5.17: Two different approaches of passing arrays' elements to a function. In the first approach, the function receives only the index of the elements, whereas in the second approach each element is passed to the function independently.

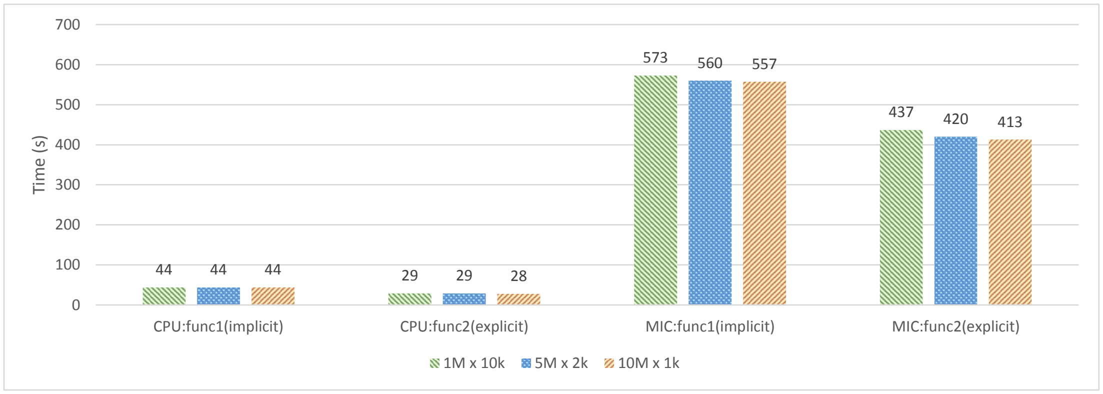


Figure 5.18: Performance of the CPU and MIC in two scenarios (implicit and explicit) on Stampede for different loop lengths (`outer_loop_len` $\times$ `inner_loop_len`). The kernel was benchmarked for three different loop combinations (scenarios): $1M \times 10k$ (green bars), $5M \times 2k$ (blue bars), and $10M \times 1k$ (orange bars).

calling functions with proper arguments. The first function received only an index as an argument (implicit method) and the second function received each array's element as an argument (explicit method). Fig. 5.18 illustrates the performance of the kernels on the CPU and MIC. The length of arrays were 10000, 2000, and 1000 elements for the first, second, and third scenarios, respectively, and the combined number of elements accessed in each scenario was 10^9. The outer loop iteration count was 1 million, 5 million, and 10 million for the first, second, and third approaches, respectively.

As illustrated in Fig. 5.18, the performance of the explicit method on the CPU and the MIC was 35% and 25% higher than the implicit method, respectively. Another noticeable difference is the performance difference between the CPU and the Xeon Phi. This difference, which is about $12-15\times$, shows that calling a function on the Xeon Phi is an expensive task compared to the CPU. As of this writing, it is unclear that whether this issue is due to a bug in the curent generation of Intel compiler or an intrinsic limitation in the current generation of hardware.

5.2.5 OpenMP loop parallelization

There are several approaches for parallelizing a serial section of code via OpenMP. Implementing parallel regions and nested loops together can cause performance bottlenecks if programmers do not take into account the effects of nested parallel loops. The situation can become even more complicated inside hybrid (MPI+OpenMP) codes such as XFLAT, where, depending on the location of code synchronization points the performance of parallel regions may be different. XFLAT has an outer loop that persists throughout the lifetime of the application. Within the loop, there are several regions that were parallelized using OpenMP. There are four possible approaches through which a parallel region and a parallel loop can be implemented in XFLAT. Fig. 5.19 illustrates the first approach, in which a parallel region encloses the outer loop as well as multiple parallel `for` loops. Note that there are single threaded sections before each `for` loop. The second approach, as illustrated by Fig. 5.20, is to enclose the internal region of the outer loop only. Thus, the single-threaded regions as well as the inner `for` loops are enclosed with an OpenMP pragma. Fig. 5.21 depicts a third approach, which is to have the parallel region enclose everything, and the `nowait` pragma is added to the inner `for` loops. In this way, every time that a thread completes its `for` loop computations, it does not stay idle at the end of the loop and continues outside the `for` loop. The last method is to parallelize only the inner `for` loops using an OpenMP pragma. Hence, there is no need to define the outer parallel region or to have single-threaded regions inside the outer loop (see Fig. 5.22).

Several kernels were constructed in order to benchmark the performance of the four approaches. In all of the benchmarks, the outer loop contained four parallel regions and four single-threaded regions. Each single-threaded region resided exactly before one `for` loop. The computations within each region depend on the previous region's result to make sure that the compiler did not remove and optimize out any

```
#pragma omp parallel
{
    Loop(termination_conditions)
    {
        #pragma omp single
        {
            /// single-threaded code
        }
        #pragma omp for
        for (int i : index)
        {
            /// multi-threaded code
        }
        ...
    }
}
```

Figure 5.19: First approach for parallelizing a region via a parallel region that encloses everything, and implements a single region within the loop.

part of the kernels.

Since the amount of computation did not vary between kernels, any performance difference was due to the different parallelism approaches. The kernels were benchmarked using three different `for` loop lengths. For the first run, the outer loop iteration count was set to 100k and every inner loop iteration count was set to 10k. The results for the MIC and CPU for the four different approaches are illustrated in Fig. 5.23. On the CPU there was no visible performance difference between the four methods, and on the MIC the maximum difference was about 10 seconds for 100k iteration count. For the next run, the outer loop iteration count was set to 500k and every inner loop iteration count was set to 2k. As shown in Fig. 5.24, on the CPU there was almost no performance difference between the different approaches; however, on the MIC, the maximum performance gap was about 40 seconds. For the last run, the outer loop iteration count was set to 1M and each inner loop ieration

```
Loop(termination_conditions)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            /// single-threaded code
        }
        #pragma omp for
        for (int i : index)
        {
            /// multi-threaded code
        }
        ...
    }
}
```

Figure 5.20: Second approach for parallelizing a region via a parallel region inside the main loop that encloses everything.

count was set to 1000. This time, as depicted in Fig. 5.25, the CPU performance fluctuation was about 7 seconds; however, on the MIC the maximum performance gap increased to about 100 seconds. The performance gap on the MIC may be due to MIC's simpler core architecture and lower clock rate. Over 1 million iterations the performance difference between the four approaches was negligible on the CPU. On the MIC the performance was less than 100 seconds. Note that in real applications 1 million iterations of the outer loop may take hours or days to complete, therefore in absolute terms, the 100 seconds of difference among the four approaches will be negligible.

For XFLAT, the fourth method was chosen. There were two main reasons for this. First of all, the performance of the third and fourth methods were always the best. The second and more important factor was simplicity. The simplicity of the fourth method comes from the fact that there is no need to define the OpenMP parallel

```
#pragma omp parallel
{
    Loop(termination_conditions)
    {
        #pragma omp single
        {
            /// single-threaded code
        }
        #pragma omp for nowait
        for (int i : index)
        {
            /// multi-threaded code
        }
        ...
    }
}
```

Figure 5.21: Third approach for parallelizing a region via a parallel region that encloses everything, with a single region within the loop. Threads at the end of parallel `for` loop do not wait for the other threads.

```
Loop(termination_conditions)
{
    /// single-threaded code
    ...
    #pragma omp parallel for
    for (int i : index)
    {
        /// multi-threaded code
    }
    ...
}
```

Figure 5.22: Fourth approach for parallelizing a region via separated parallel `for` regions.

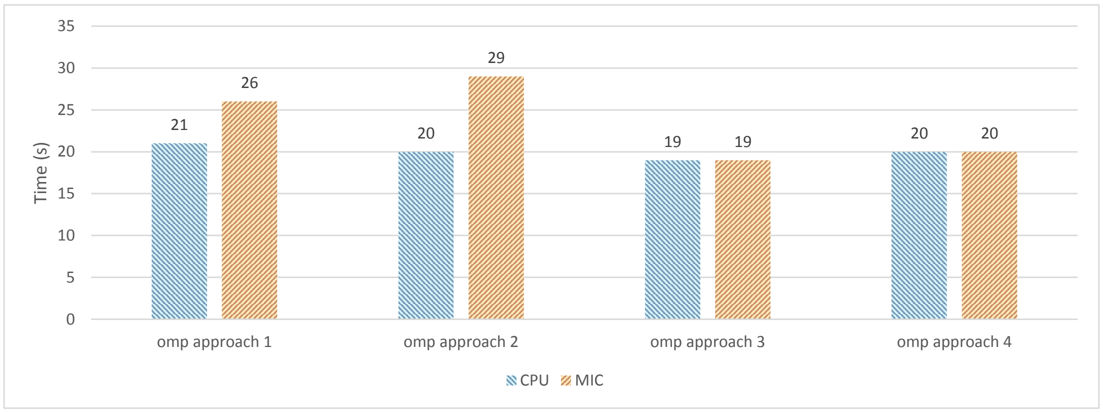


Figure 5.23: Illustration of CPU (blue bars) and MIC (orange bars) OpenMP performance on Stampede for 100k × 10k loop configuration (`outer_loop_len` × `inner_loop_len`). Numbers on top of bars correspond to total time in seconds.

region to enclose the inner parallel `for` loops. Furthermore, defining the single-threaded regions was not required. Consequently, MPI functions can be put after each `for` loop without requiring to treat them as special lines of code inside OpenMP parallel regions. As a result, the implementation becomes simpler, maintenance

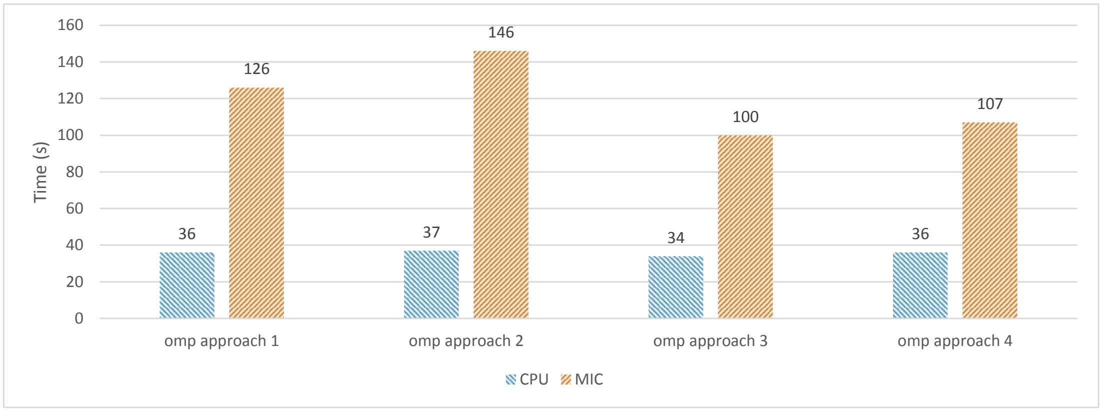


Figure 5.24: Illustration of CPU (blue bars) and MIC (orange bars) OpenMP performance on Stampede for 500k × 2k loop iteration counts (`outer_loop_len` × `inner_loop_len`). Numbers on top of bars correspond to total time in seconds.

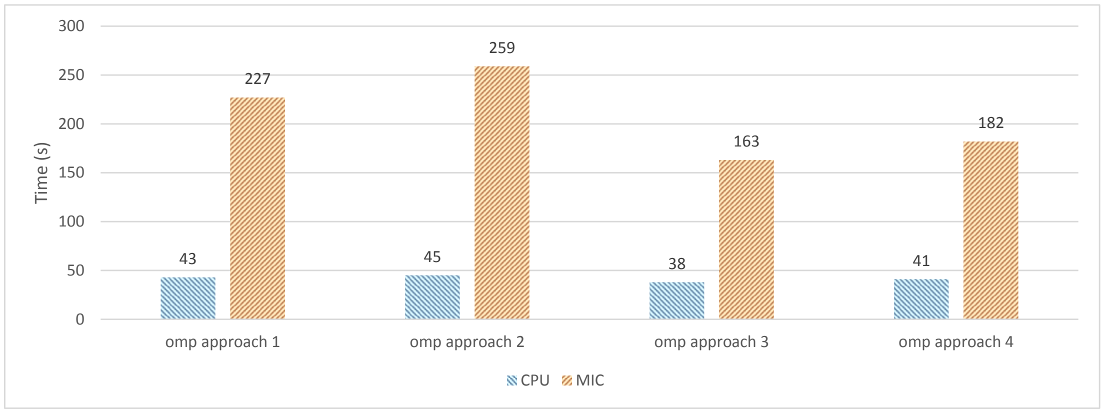


Figure 5.25: Illustration of CPU (blue bars) and MIC (orange bars) OpenMP performance on Stampede for 1M × 1k loop iteration counts (`outer_loop_len` × `inner_loop_len`). Numbers on top of bars correspond to total time in seconds.

becomes easier, and the debugging phase is less complicated.

5.2.6 The effects of NUMA on multi-socket code performance

On multi-socket systems, multiple processors may be installed on a single motherboard and managed by a single image of an OS. Therefore, the OS can manage all of the available cores. Furthermore, the OS manages all of the available memory, although each RAM module is directly connected to only one CPU socket. As a result, one single instance of an application can employ every accessible core as well as the entire amount of RAM. For multi-threaded codes, depending on the application objective, at some points every thread may require access to a specific memory block. Since each block of memory resides on a unique location on physical RAM, threads that do not reside on the corresponding socket can access the memory block only by going through extra buses. For example, refering back to Fig. 4.19, if a thread on CPU1 attempts to access a memory block that resides on the RAM module connected to CPU2, it has to go through one more bus (QPI) in order to access the memory. It is not hard to imagine the performance impact of a situation where many threads on different CPUs are attempting to access scattered memory blocks on a multi-socket system. The situation will be even worse if the application repeats the memory access within a loop for every iteration.

Two possible approaches can be used to address this issue. The first approach is to make the memory initialization code multi-threaded, in the same way that the rest of the code is multi-threaded. Modern OSs do not allocate the requested memory before the first touch, *i.e.* the first attempt to read from or write to the memory. Therefore, the allocated memory will reside as close as possible to the thread that first attempts to touch it. If the memory initialization code is multi-threaded in the same way as the rest of the code, it is highly probable that the thread's initialized memory and its working memory remain identical (see Fig. 5.26). The second method is to eliminate the issue completely by running two separate instances of the code, each on a single socket. However, the second method requires multi-node support by employing MPI communication.

XFLAT's functions fetch large amount of data per iteration. Therefore, this issue could potentially cause a seious performance bottleneck for XFLAT running on a multi-socket system. In order to measure the performance impact of the first touch issue, XFLAT was benchmarked for three different scenarios on Stampede. For the first run, a single instance of XFLAT utilized the entire dual-socket compute node. For the second run, two separate instances of XFLAT were run, each employing one socket of a dual-node compute node. The communication between the two instances was performed via MPI (software) and on the hardware side via the QPI bus. For the third run, a single instance of XFLAT utilized an entire dual-socket compute node similar to the first run, however, the memory initialization section was parallelized via OpenMP. All runs were executed for $\sim$ 100 seconds, with a

```
// By adding the OpenMP parallel for pragma, memory initialization
// become multi-threaded. As a result, the allocated memory resides
// near the thread that initialized it

#pragma omp parallel for
for (index: ARRAY_LENGTH)
    Beam[i].init();

// The computational part is always multi-threaded
#pragma omp parallel for
for (index: ARRAY_LENGTH)
    Beam[i].calc(...);
```

Figure 5.26: Parallel initialization of memory as one of the solutions for the first-touch memory issue.

problem size of $1200 \times 100 \times 80$ ($\theta \times \phi \times\ EBins$) neutrino beams. At the end of each run, the total (integer) number of computed radial steps was measured. As depicted in Fig. 5.27, the performance of the first run was half that of the other runs, indicating that the inter-socket communications can significantly impact on the performance of high-performance applications such as XFLAT. As a result, the amount of inter-socket communications should be minimized in order to achieve maximum application performance.

5.2.7 Fusing functions

XFLAT contains several functions in which different instructions are performed on identical data. By calling each function, the required data must be fetched from main memory. The fetched data is stored within the processor's fast memory, *i.e.* the cache memory. After the function returns to the calling point, its data that resides in the processor's cache memory may be evicted in order to make space for new function data. However, the new function may perform different calculations on

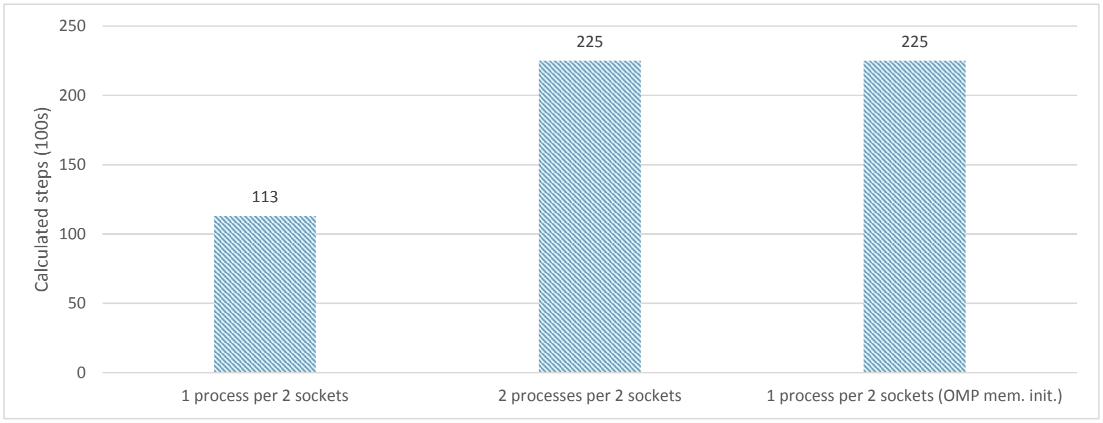


Figure 5.27: Performance of three alternative methods of running XFLAT on a NUMA system (Stampede). The calculated radial steps were measured for 100-second runs. From left to right, one XFLAT instance employed the entire dual-socket node, two XFLAT instances each employed a single CPU socket, and one XFLAT instance initialized memory in parallel and employed the entire dual-socket node. Net computed radial steps (listed at the top of the bars) were measured for ~ 100 second runs.

the same set of data and there is no guarantee that the old data, within the cache, can be reused. Since the fetching time is on the order of microseconds and the execution time is on the order of nanoseconds (the clock frequency of the modern processors is typically above 1 GHz), reducing the amount of memory fetch can boost the overall performance dramatically.

In order to improve performance, XFLAT provides fused functions, which were created by merging two functions that belong to a single module. The functions can be fused if and only if they process on the same set of data. The method of fusing functions is only applicable when the data remain unchanged between the two function calls.

As an example of implementing fused functions, within the NBeam class, the function that calculates the neutrinos' evolution can be fused with the function that calculates the partial summation over the neutrinos' wavefunction, since both functions may be called back to back and require the same set of data. In NBeam class, there is another, similar fused function that receives an array of neutrino beams as an extra argument. Thus, it can calculate the average of the neutrinos' flavor states between the computed neutrino beams and the passed neutrino beam array (*i.e.*, three functionalities fused together).

As another example, at the end of the evolution loop, within the numerical module, there is a function that computes the maximum global error. Moreover, there is a function at the beginning of the evolution loop in which partial summation over neutrino's energy bins is performed. Since those functions perform calculations on the same set of data, and the data remain unchanged between the two calls, they can be safely fused together. Nevertheless, the original version of those functions is available as well, since depending on the algorithm the fused versions might not be applicable everywhere.

In order to study the effect of function fusing, XFLAT was benchmarked on Stampede using the two alternative approaches. The first run employed the plain non-fused functions, and the second run exploited the fused functions for calculations. Fig. 5.28 illustrates the performance of XFLAT for the problem size of $1200 \times 100 \times 80$ ($\theta \times \phi \times$ $Ebins$) on both the CPU and MIC. Each run continued for about 100 seconds and the number of calculated radial steps were measured. The XFLAT performance on the MIC improved by 15% when fused functions were used. On the CPU, the improvement was around 30% due to its larger cache size, in which more data can be maintained, thus the required memory fetches were reduced. This is a notable result since the fusing function optimization was intended for the Xeon Phi. However, the CPU can gain even more benefit from the method since the cache size on the CPU is larger.

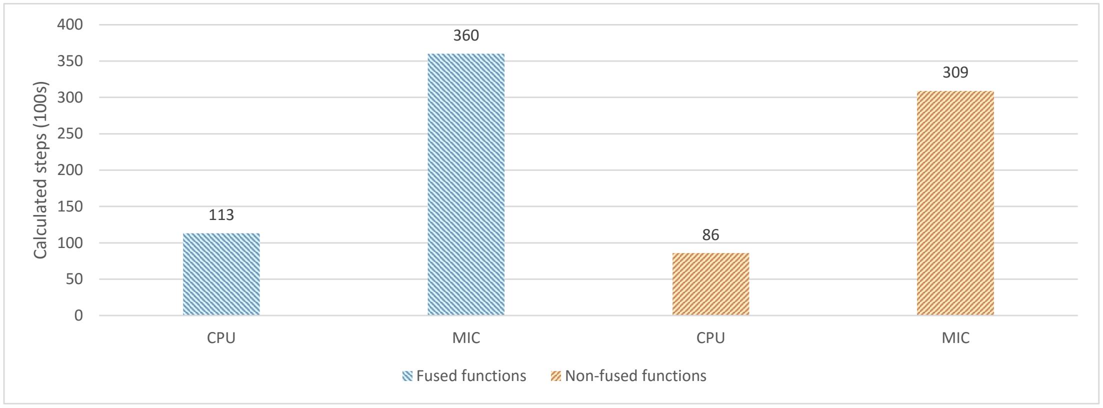


Figure 5.28: Performance of fused and non-fused functions on the CPU and the MIC on Stampede. The net computed radial steps (listed at the top of the bars) were measured for $\sim$ 100 second runs. XFLAT performance was 15% and 30% higher on the MIC and CPU, respectively, when the fused functions were employed (blue bars), compared to when the non-fused functions were employed (orange bars).

5.2.8 I/O performance

Every high-performance computing application requires saving the results of computations by writing data before termination. Since I/O tasks are slower than the rest of an application, there may be bottlenecks in the I/O section of an HPC code. On heterogeneous-node systems the performance hit may be even greater since there are different types of processors as well as multiple data buses involved. Depending on the run configuration, XFLAT may need to write gigabytes of data to disk at each radial point. Since XFLAT employs NetCDF for saving data, the performance of NetCDF plays critical role in XFLAT performance. There are several approaches for writing multi-dimensional arrays of data using NetCDF, thus several kernels with similar structure to XFLAT's I/O module were developed in order to determine the optimum approach.

In XFLAT, there are three methods for saving multi-dimensional data to disk.

```
for (int theta : Theta_Angles)
    for (int phi : Phi_Angles)
        for (int p : Particle_Num)
            for (int c : WF_Components)
                for (int e : Energy_Bins)
                    nc_put_var...(..., data[theta][phi][p][c][e]);
```

Figure 5.29: Saving data via NetCDF within nested loops.

```
for (int i : Neu_Beams_TotLen)
{
    int index = calc_index(i);
    nc_put_var...(..., start[index], count[index], data);
}
```

Figure 5.30: Saving data via NetCDF within a single loop.

The first method is to write data within nested loops as illustrated in Fig. 5.29. Nested loops may have more overhead than a single loop. Therefore, the second method is to implement a single loop in which a 1D data array is written to disk (see Fig. 5.30). The third approach is similar to the second approach; however, instead of calling the NetCDF function within the loop, it is only called once after the loop is completed. Thus within the loop, data is extracted from neutrino beams and saved to a buffer. After completion of the loop, the entire buffer is passed to the NetCDF function to be written to disk.

In order to study the efficiencies of these three approaches, the I/O module of XFLAT was isolated and benchmarked as a separate kernel. As a result, the performance of the other calculations in other modules did not interfere with the performance of the I/O methods. The kernel was benchmarked on Stampede for both the CPU and the MIC. Each kernel's snapshot was 128 MB and a total of 100 snapshots (12.8 GB) were written on Stampede's `$SCRATCH` disk. Fig. 5.32 shows the

```
for (int i : Neu_Beams_TotLen)
{
    int index = calc_index(i);
    buffer[index] = data[index];
}
nc_put_var...(..., buffer);
```

Figure 5.31: Saving data via NetCDF after completion of a single loop. Within the loop data may be extracted from NBeam objects and stored onto a buffer.

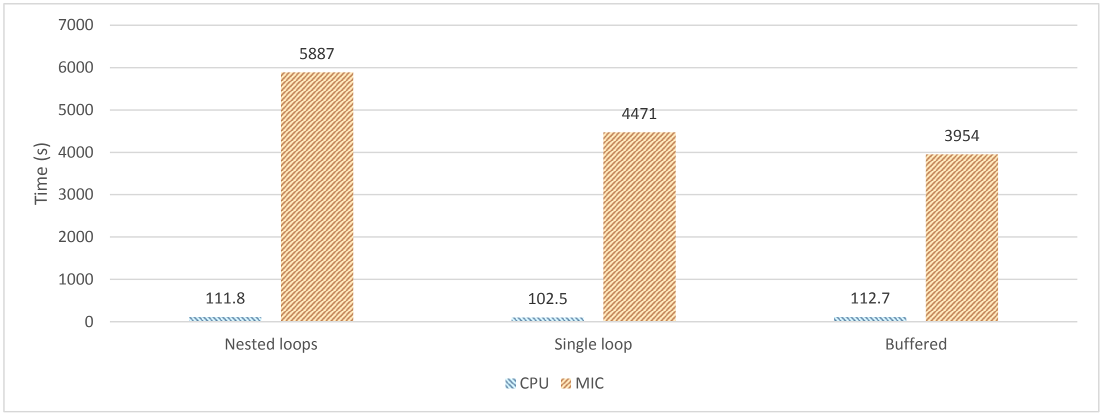


Figure 5.32: XFLAT's I/O performance on the CPU and MIC for three different approaches on Stampede. The performance of the MIC (orange bars) was 35 to 53 times worse than the performance of CPU (blue bars). Numbers at top of bars correspond to time in seconds.

performance of each approach on the CPU and MIC.

As seen in Fig. 5.32, the Xeon Phi has very poor I/O performance that may result in a major bottleneck at run time. Therefore, direct I/O tasks should be avoided on the Xeon Phi. Since I/O tasks are inevitable in production environments, another I/O module was added to XFLAT. In the new module, since the I/O performance of the CPU was satisfactory, the Xeon Phi sends its own data to its CPU mate instead of writing the data to disk directly. Afterwards, the CPU is responsible for

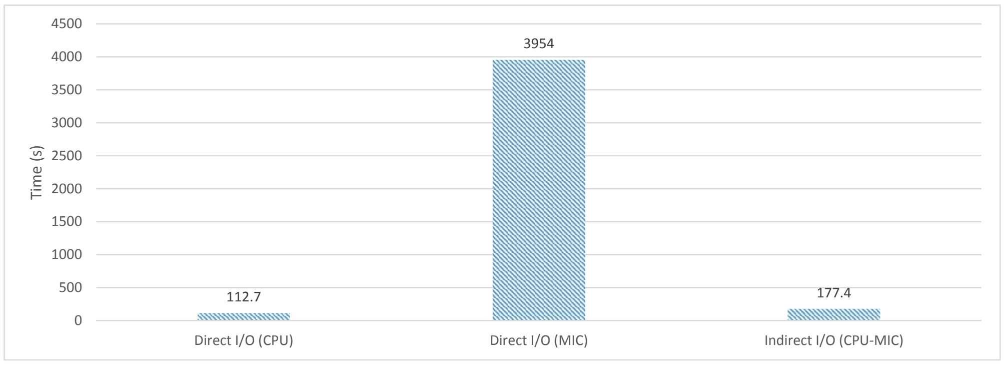


Figure 5.33: XFLAT's I/O performance on the CPU and MIC on Stampede. From left to right: direct I/O on the CPU, direct I/O on the MIC, and indirect I/O from MIC to CPU. Numbers at top of bars correspond to time in seconds.

saving its own data as well as that of the Xeon Phi. As a result, in this new indirect module, the Xeon Phi extracts data from NBeam objects and saves them to a buffer which later sends the buffer data to the CPU. Consequently, the CPU is the only processor type that is responsible for performing I/O tasks and writing data to disk. Fig. 5.33 illustrates the performance of the direct module in which each processor is responsible for writing its own data to disk, and the indirect module in which the MIC sends its data to CPU for dumping to disk.

Table 5.1: Bethe workstation specifications.

CPU	Intel Xeon CPU 6Core/12Thread E5-2620 @ 2.0 GHz
Mem	8x 4096 MB DDR3 @ 1333 MHz
MIC	Intel Xeon Phi 5110P 60Core/240Thread @ 1.053 GHz - 8 GB GDDR5

Table 5.2: Bahcall workstation specifications.

CPU	2x Intel Xeon CPU 6Core/12Thread E5-2620 v2 @ 2.1 GHz
Mem	4x 16 GB DDR3 @ 1600 MHz
MIC	2x Intel Xeon Phi 3120A 57Core/228Thread @ 1.1 GHz - 6 GB GDDR5

5.3 Performance Analysis

In order to fully analyze the performance of XFLAT, the code should be benchmarked on single-node and multi-node as well as homogeneous and heterogeneous environments. It is expected that the performance of the code will scale linearly or near-linearly with processor load (*i.e.*, as problem size increases). To test this, the code was benchmarked on three different available machines: Bethe (Table 5.1), installed at the Center for Advanced Research Computing (CARC) at the University of New Mexico; Bahcall (Table 5.2), located at the Physics and Astronomy Department of the University of New Mexico; and the Stampede supercomputer (Table 5.3), installed at the Texas Advanced Computing Center (TACC). The TACC Stampede system is a 10 PFLOPS (PF) Dell Linux Cluster based on 6400+ Dell PowerEdge server nodes, each outfitted with 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi coprocessor (Intel MIC). The aggregate peak performance of the Xeon E5 processors is 2+PF, while the Xeon Phi processors deliver an additional aggregate peak performance of 7+PF [TACC, 2015].

Based on Stampede's specifications, Hyper-Threading (HT) was apparently disabled on Stampede's compute nodes. This was confirmed in an email dated 11/26/14 by TACC's system administrators. They stated:

Table 5.3: Stampede Dell PowerEdge C8220z compute node specifications.

CPU	2x Intel Xeon CPU 8Core/8Thread E5-2680 @ 2.7 GHz
Mem	8x 4096 MB DDR3 @ 1600 MHz
MIC	2x Intel Xeon Phi SE10P 61Core/244Thread @ 1.1 GHz - 8 GB GDDR5

"HPC workloads are typically parallelized in a homogeneous fashion, and this is not optimal for HyperThreading – if all the threads are going after the same resources, then performance is not likely to improve much, and will often decrease due to contention for cache resources and DRAM banks. The primary reason we have disabled HyperThreading on all of our production systems is that the performance degradation due to incorrect assignment of processes/threads to logical cores can easily outweigh the (modest) benefits that HyperThreading might provide. The variability in performance due to incorrect assignment of processes/threads to logical cores would almost certainly confuse users and significantly increase our support workload."

Interestingly, however, and as depicted in Fig. 5.34, when XFLAT was benchmarked on Bahcall using Hyper-Threading a small performance gain was observed.

In the remaining plots presented in this section, an instance of XFLAT (extended supernova module), was run on each compute node. Unless stated otherwise, in all of the following benchmarks the number of energy bins was set to 100, the number of azimuth angles (φ bins) was set to 100, and only the number of zenith angles (ϑ bins) was varied. There may be thousands of neutrino zenith angles in a typical problem size, and the number of zenith angles can be increased until it hits the maximum available memory on each node.

The starting point of problem size for the following benchmarks was $1000 \times 100 \times 100$ ($\theta\ angles \times \phi\ angles \times\ Energy\ bins$) beams and the number of zenith angles was increased by 500 bins until it hit $T \times 100 \times 100$ neutrino beams in which the parameter

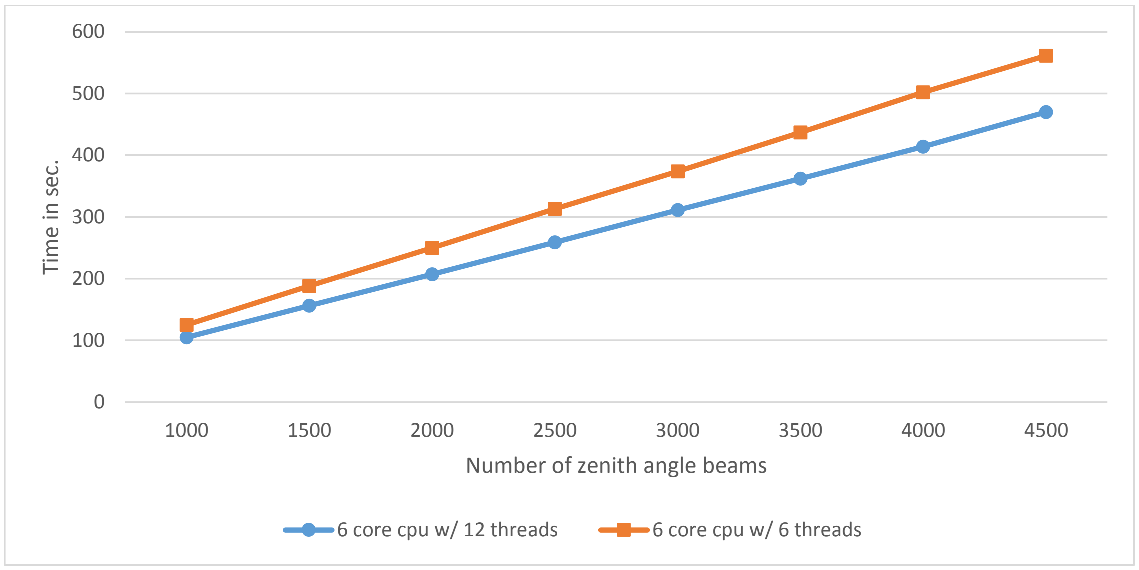


Figure 5.34: CPU benchmark with and without Hyper-Threading on Bahcall. The blue line shows a run utilizing all threads on the processors, while the orange line shows a run utilizing single thread per core on the CPU.

T depended on the available memory for the CPU or MIC. All the benchmarks were executed for a total of 100 radial steps.

5.3.1 XFLAT single processor scalability

The first set of benchmarks was designed to test XFLAT scalability as the load increased on a single processor. As shown in Fig. 5.35, when the load was increased on the CPU the timing increased linearly. This shows that the code can utilize the entire processor, since by increasing the load the performance did not decrease. However, surprisingly, on the dual-socket Bahcall, at $\sim$ 4500 zenith beams the timing started to improve!

This behavior was unexpected since typically by increasing the amount of load on a node, the performance behaves linearly or decreases after reaching a particular

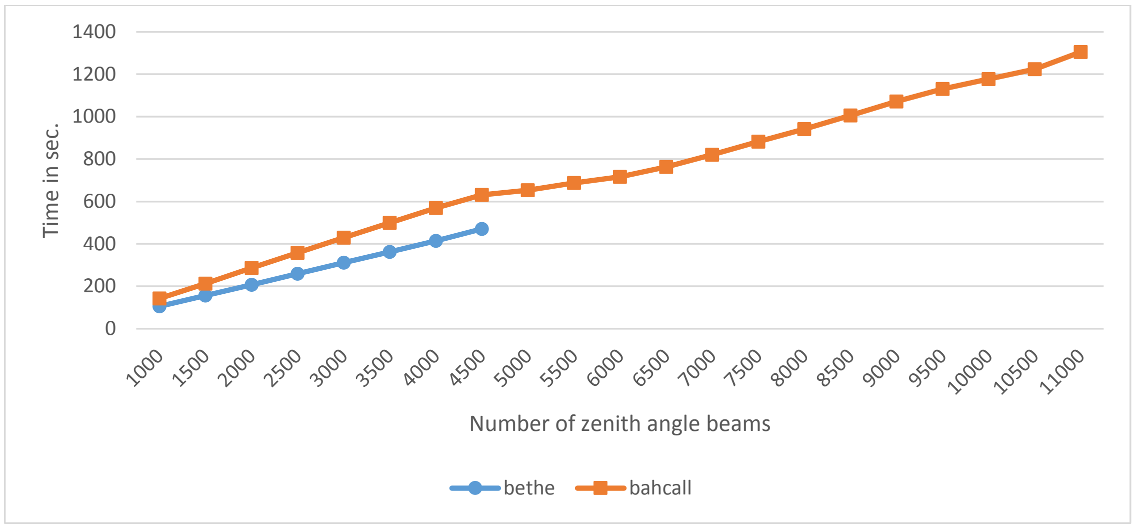


Figure 5.35: XFLAT performance on Bethe and Bahcall machines using a single CPU. The orange curve shows the run time on Bahcall and the blue line shows the run time on Bethe, both as a function of the number of zenith angles. Due to memory limitations runs with greater than 4500 zenith angle beams could not be performed on Bethe.

point. The first step to investigate more about this result was to take into account the hardware differences between the two machines. Bahcall equipped with more memory, thus it was possible to continue the benchmark using a larger number of neutrino beams. However, the performance of the Bethe's CPU was about 30%-40% better than Bahcall's, although both CPUs are the same model and Bahcall's is a later version (Bahcall's CPU is E5-2620v2 and Bethe's CPU is E5-2620). Hence, two puzzling behaviors can be seen in Fig. 5.35: the improvement of XFLAT timing when the load increased on Bahcall, and second, Bethe's better CPU performance, even though Bahcall is equipped with a newer generation of CPU.

Both of these behaviors can be explained by noting an important additional hardware configuration difference between Bahcall and Bethe. On Bethe, the RAM modules are all connected directly to the single available CPU, while on Bahcall half

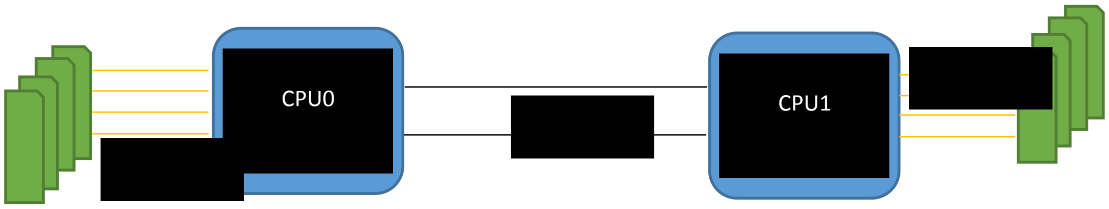
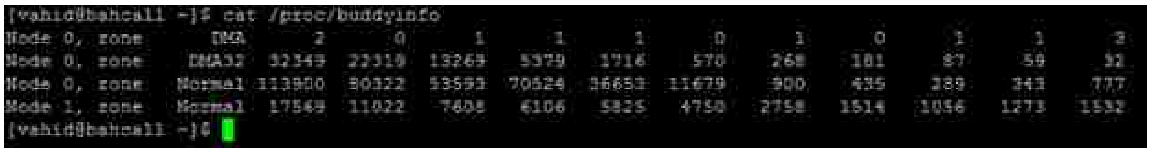


Figure 5.36: Illustration of the memory configuration in NUMA architecture machines, such as Bahcall. CPUs are connected via Intel QPI link and DDR3 is the link between the memory and processor.

of the RAM modules are installed on the other zone, *i.e.*, connected to the other CPU socket. On Bahcall the CPU accesses the second half of the RAM via the QPI bus (Fig. 5.36). Consequently, the timing improvement was conjectured to be due to the fact that on Bahcall, when too many beams are allocated, some of them are actually allocated on the other memory zone, which is connected to the second CPU socket. Therefore, since accessing the RAM modules on the second zone is provided via QPI bus, the first CPU gained extra memory bandwidth. In other words, the CPU can access to the first part of the memory via its direct memory bus, and it can simultaneously access the second part via the QPI bus.

The first step to confirm this hypothesis was to find out the physical location of the allocated memory by consulting the operating system. On most of the Linux systems this can be confirmed by probing the `/proc/buddyinfo` file. Bahcall's OS is RedHat-based and according to RedHat:

"This file is used primarily for diagnosing memory fragmentation issues. Using the buddy algorithm, each column represents the number of pages of a certain order (a certain size) that are available at any given time. For example, for zone DMA (direct memory access), there are 90 of $2^{0*\texttt{PAGE\_SIZE}}$ chunks of memory. Similarly, there are 6 of $2^{1*\texttt{PAGE\_SIZE}}$ chunks, and 2 of $2^{2*\texttt{PAGE\_SIZE}}$ chunks of memory available. The **DMA** row references the first 16 MB on a system, the **HighMem** row references all



```
[vahid@bahcall ~]$ cat /proc/buddyinfo
Node 0, zone      DMA      2      0      1      1      1      0      1      0      1      1      3
Node 0, zone    DMA32  32349  22319  13269   5379   1716    570    268    181     87     59     32
Node 0, zone   Normal 113900  80322  53593  70524  36653  11679    900    435    289    343    777
Node 1, zone   Normal  17569  11022   7608   6106   5825   4750   2758   1514   1056   1273   1532
[vahid@bahcall ~]$
```

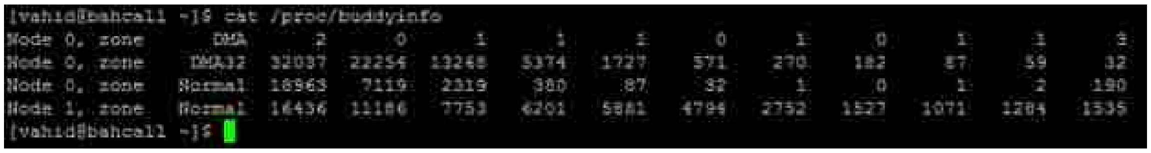
Figure 5.37: `/proc/buddyinfo` file content prior running XFLAT on Bahcall.

memory greater than 4 GB on a system, and the **Normal** row references all memory in between."

`PAGE_SIZE` is a fixed-length continuous block of memory that is the smallest unit of data for memory management on an operating system. Normally, `PAGE_SIZE` is 4096 bytes. In the course of large memory allocation, the number of available pages decreases accordingly. Therefore, it is possible that if the memory, which is directly connected to the first CPU socket, is exhausted, then the rest of the allocation is performed on the second memory zone that is connected to the second socket. In Fig. 5.35, Bahcall's performance was essentially linear up to 4500 zenith angle beams; afterwards there is a sudden change when the number of allocated zenith angle beams was set to 5000 and larger. By probing the `/proc/buddyinfo` file during the allocation, it was possible to figure out if the performance gain was due to the memory allocations on both sockets that resulted in providing more memory bandwidth to the processor.

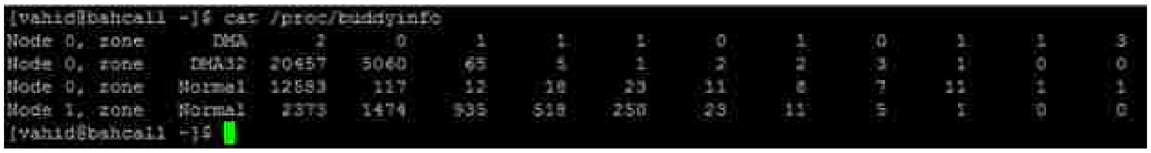
Fig. 5.37 illustrates the `/proc/buddyinfo` contents prior to XFLAT run on Bahcall, showing that there were plenty of pages available on both sockets (Node 0 and Node 1).

For a problem size below the 5000 zenith angle beams, the required memory for XFLAT could fit onto one memory zone, therefore the code did not utilize more memory bandwidth. As a result, it scaled linearly. For example, as shown in Fig. 5.38 for a run with 2000 zenith angles, the entire required memory was allocated onto



```
[vahid@bahcall ~]$ cat /proc/buddyinfo
Node 0, zone      DMA      2      0      1      1      1      0      1      0      1      1      3
Node 0, zone    DMA32  32037  22254  13248   5374   1727    571    270    182     87     59     32
Node 0, zone   Normal  18963   7119   2319    380     87     32      1      0      1      2    190
Node 1, zone   Normal  16436  11186   7753   6201   5881   4794   2752   1527   1071   1284   1535
[vahid@bahcall ~]$
```

Figure 5.38: `/proc/buddyinfo` file content when XFLAT instance fit onto one memory zone on Bahcall.



```
[vahid@bahcall ~]$ cat /proc/buddyinfo
Node 0, zone      DMA      2      0      1      1      1      0      1      0      1      1      3
Node 0, zone    DMA32  20457   5060     65      5      1      2      2      3      1      0      0
Node 0, zone   Normal  12583    117     12     18     23     11      8      7     11      1      1
Node 1, zone   Normal   2373   1474    935    518    250     23     11      5      1      0      0
[vahid@bahcall ~]$
```

Figure 5.39: `/proc/buddyinfo` file content when XFLAT instance does not fit onto one zone on Bahcall.

Node 0 only.

However, as soon as the number of zenith angle beams exceeded ~ 5000, the entire allocation could not fit onto one zone, thus several memory pages were allocated on the other memory zone. In this way, extra buses became available for transferring memory from/to CPU via QPI. Fig. 5.39 illustrates a run with 5000 zenith angle beams. In contrast to the previous screenshots, the available pages (especially large chunk size pages) were depleted on both zones (sockets), which confirms the memory distribution over both sockets.

Moreover, in order to confirm this result from a software point of view, XFLAT was benchmarked for two different scenarios. One benchmark was designed so that in the first run, the neutrino's beams were distributed evenly on both memory zones using all available threads on both CPU sockets, but the calculations were performed by employing only the first CPU's threads. In this way, similar to the previous benchmarks, only one CPU was employed to process the data, yet half of the memory was connected directly to the CPU and the rest of the memory was connected to the CPU via QPI. This resulted in providing more aggregate memory bandwidth.

```
#pragma omp parallel for num_threads(24)
for (int i = 0; i < neutrino_beams; ++i)
    new (&beam[i]) NBeam();
```

Figure 5.40: Parallel neutrino beam memory allocation using all available threads on Bahcall.

```
#pragma omp parallel for num_threads(12)
for (int i = 0; i < neutrino_beams; ++i)
    exec(beam[i]);
```

Figure 5.41: Parallel execution of neutrino beams using half of the available threads on Bahcall.

In order to reach this goal, OpenMP threads were utilized. First, for the memory initialization all the available threads were used on Bahcall (12 threads per CPU), as illustrated in Fig. 5.40. Hence, memory usage was uniformly distributed over two sockets.

After performing memory initialization, the rest of the computational loops were performed using only the OpenMP threads on the first CPU. Therefore, the number of threads were adjusted to 12 (Fig. 5.41), since each CPU on Bahcall has 12 hardware threads.

Nevertheless, enabling only half of the available threads on Bahcall was not enough, since one has to make sure that only threads on the first CPU were participating in the computational loops. This can be achieved by invoking OpenMP functions within the program or by setting environment variables from the OS shell. The settings and commands may vary between compilers and shells. For all of the benchmarks the Intel compiler was used and OpenMP threads were located as close as possible to each other (`export KMP_AFFINITY=compact`). As a result, utilizing the first CPU with 6 cores/12 threads was possible at run time.

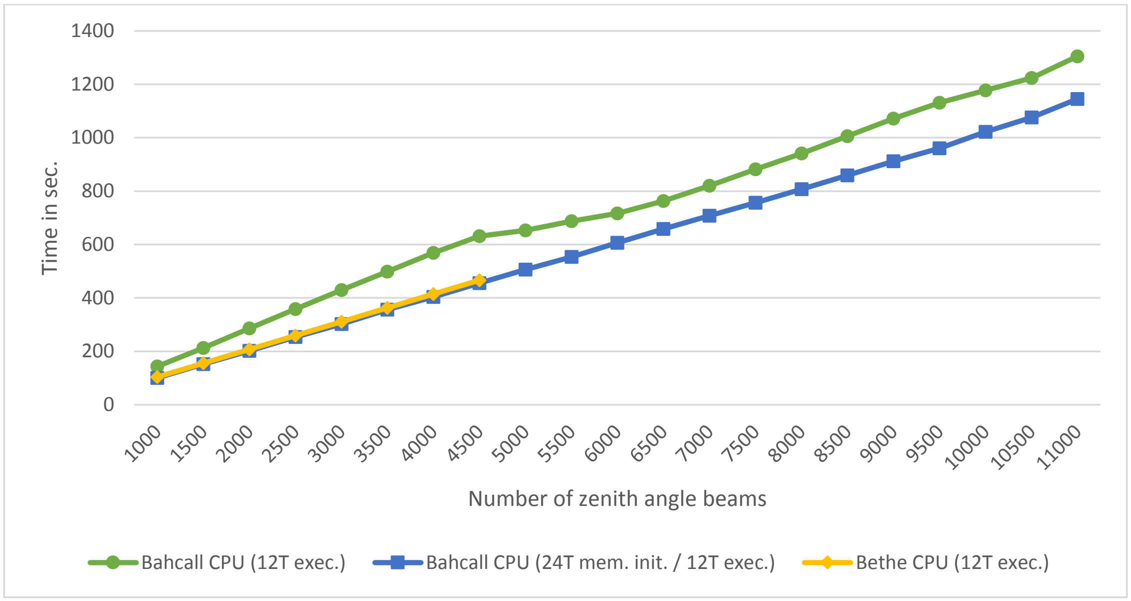


Figure 5.42: XFLAT performance on Bahcall when different numbers of threads (indicated by 12T or 24T) are utilized for memory initialization (init.) and execution (exec.). Green curve illustrates performance when a single Bahcall CPU was utilized for memory initialization and execution. Blue curve illustrates performance when both Bahcall CPUs was utilized for memory initialization but only a single CPU were employed for execution. Yellow curve illustrates the performance of the single-CPU Bethe machine.

Fig. 5.42 illustrates that by applying these changes, the code now scaled linearly on Bahcall and the timing results were similar to those on Bethe. Furthermore, the performance of Bahcall's CPU was very similar to Bethe's CPU due to providing more memory channels.

The performance difference initially observed between the two machines was this clearly related to the memory bottleneck on Bahcall. Two RAM modules (each 16 GB) were connected to each CPU on Bahcall, while on Bethe, eight RAM modules (each 4 GB) were connected to the CPU. Since Bahcall's CPU could not reach full capability due to this bandwidth bottleneck, the RAM configuration on the machine was subsequently modified in order to provide higher memory bandwidth.

Previously, Bahcall was equipped with four 16 GB RAM modules, two installed on each zone (connected to each CPU). However, the Bahcall's CPU (Xeon E5-2620v2) can support a quadruple-channel architecture, which increases data transfer rate by adding four communication channels between the CPU and RAM modules.

The multi-channel technology effectively multiplies data throughput from RAM to the CPU's memory controller by providing multiple channels. In the single-channel configuration there is only one 64-bit memory channel to transfer data between the RAM and CPU (see Fig. 5.43). In the dual-channel configurations, two channels are available between the selected RAM modules and the memory controller, thus the effective data channel is equal to a 128-bit channel (see Fig. 5.44). The same is true for triple and quadruple channel configurations that are only available on high-end CPUs such as Bahcall's CPUs. The multi-channel technology requires identical RAM modules. For instance, for the quadruple-channel configuration four identical RAM modules should be installed on specific slots on motherboard. The RAM modules should be identical in capacity, speed, latency, number of memory chips, and matching size of rows and columns of memory cells.

Consequently, the RAM configuration on Bahcall's machine was changed and eight 8 GB RAM modules were installed on it; four modules on each CPU zone. The rest of the memory's specifications remained identical as before. Fig. 5.45 depicts the improvement for the single-CPU benchmark and Fig. 5.46 illustrates the improvement for the dual-CPU benchmark on Bahcall.

The next performance benchmark for the scaling of XFLAT was the performance benchmark on multi-socket systems. Bahcall and a single Stampede compute node were utilized for that purpose. On Bahcall, Hyper-Threading was enabled; thus, up to 24 threads could be employed for the benchmark. Since each node of Stampede contains two 8-core CPUs, and Hyper-Threading was disabled as noted above, it was possible to utilize up to 16 threads per Stampede node. Two separate XFLAT

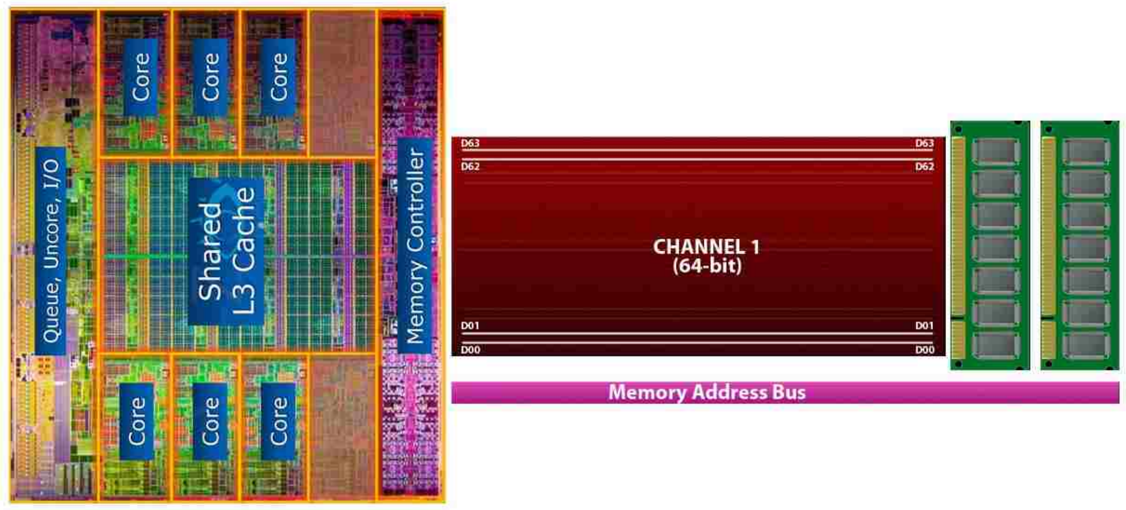


Figure 5.43: Illustration of single-channel memory configuration between Intel CPU and DDR3 memory modules [www.gamersnexus.net, 2015]. One 64-bit channel is available for transferring data to/from the CPU.

benchmarks were performed on a single Stampede node. The first one launched one MPI task per socket (illustrated in Fig. 5.47), and the other launched only one MPI task per compute node (one MPI task per two CPUs) as shown in Fig. 5.48.

Since XFLAT can uniformly allocate memory on multi-socket nodes, and since the QPI buses between CPUs on a Stampede node have enough bandwidth (dual QPI, 8 GT/s (GigaTransfers per second) per QPI bus), the performance difference between the two benchmarks is negligible as depicted in Fig. 5.49.

The technical reasons for the performance similarity between the two Stampede benchmarks is due to the way that the code is implemented. There are two main features that keep performance satisfactory on the multi-socket benchmarks. First of all, during memory initialization (see discussion of first touch earlier in this section), all of the threads contribute in the initialization section. This indicates that the memory section related to each thread resides as close as possible to the thread. Therefore, threads on one CPU do not have to try to access the other CPU's memory

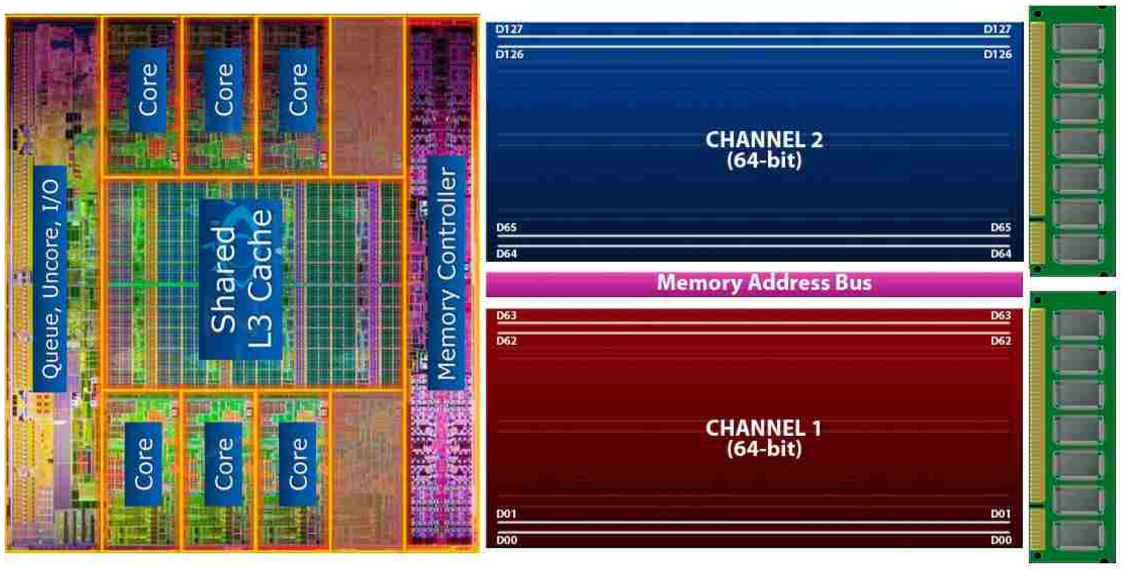


Figure 5.44: Illustration of dual-channel memory configuration between Intel CPU and DDR3 memory modules [www.gamersnexus.net, 2015]. Two 64-bit channels (one 128-bit aggregate channel) are available for transferring data to/from the CPU.

zone, thus the communication overhead between sockets is reduced.

Another reason is because of the selected approach for implementing the Hamiltonian in XFLAT. In order to compute the total Hamiltonian, which depends on the selected physics module, one or more nested summation loops have to be computed. According to Eq. 3.8, the value of the integral can be calculated by performing a loop over all of the neutrino's angle beams. Since the neutrino beams are distributed over compute nodes, a bad implementation would transfer a lot of data between nodes in order to calculate the integral. However, in XFLAT, each node can go through an independent loop for calculating the partial integral over their own set of beams. At the end of the loops, only the final value is exchanged between nodes via the `MPI_Allreduce()` function. Therefore, every node at the end of the Hamiltonian function has the final value of the Hamiltonian integral. In this way, only a few bytes are required to be exchanged after the completion of the nested loops on each

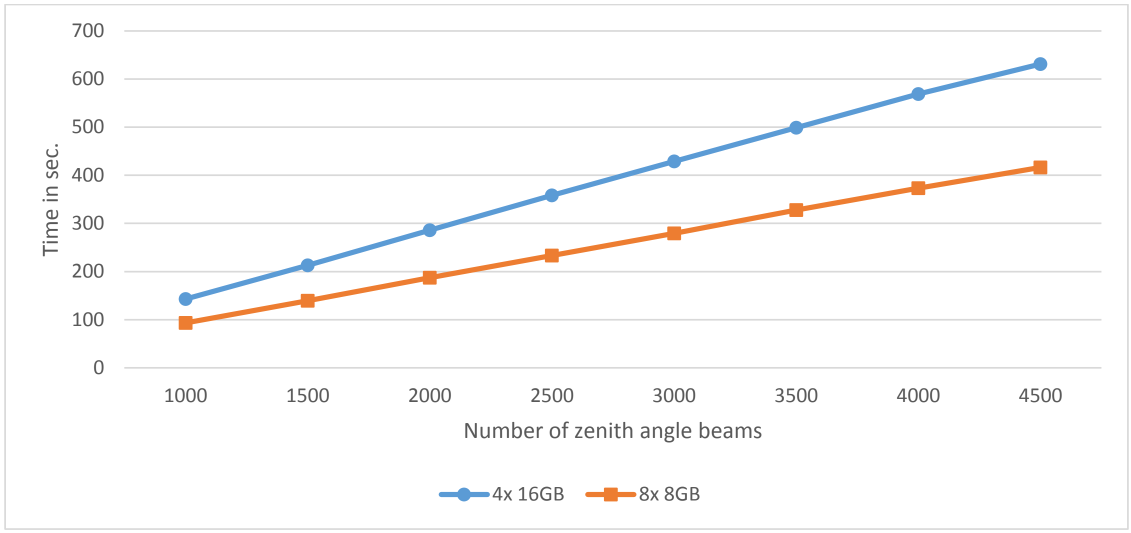


Figure 5.45: Single CPU timings on Bahcall when 4 RAM modules were installed (blue) and when 8 RAM modules were installed (orange).

node. Within the neutrino evolution loop there are not many exchange points, thus there is no need for large communication bandwidth. As a result, it is expected that by increasing the number of nodes, the performance of XFLAT should not drop and will scale linearly.

Prior to studying the multi-node benchmarks, XFLAT was benchmarked on single CPU, dual CPU, single MIC, and dual MIC configurations on a single node. As shown in Fig. 5.50, XFLAT was benchmarked on a Stampede node for 1000 radial steps. The number of zenith angles started at 200 and scaled up to 1200. It was not possible to fit additional beams onto the MIC's memory.

As can be seen from the figure, MIC performance was about three times higher than that of the CPU on Stampede. In addition, the performance of the dual-processor benchmarks was twice that of the single-processor benchmarks.

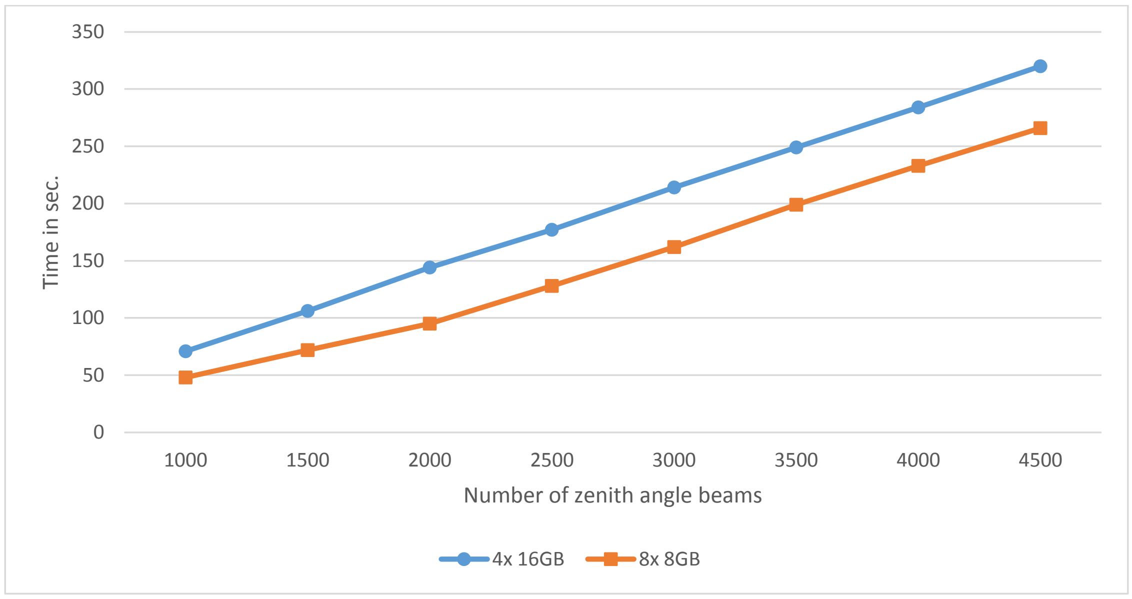


Figure 5.46: Dual CPU timings on Bahcall when 4 RAM modules were installed (blue) and when 8 RAM modules were installed (orange).

5.3.2 XFLAT perofrmance on heterogeneous environments

Since XFLAT should be able to utilize both the MIC and CPU simultaneously, the next stage of XFLAT performance analysis was to study code performance on heterogeneous (MIC and CPU together) environments. As the first step on a heterogeneous

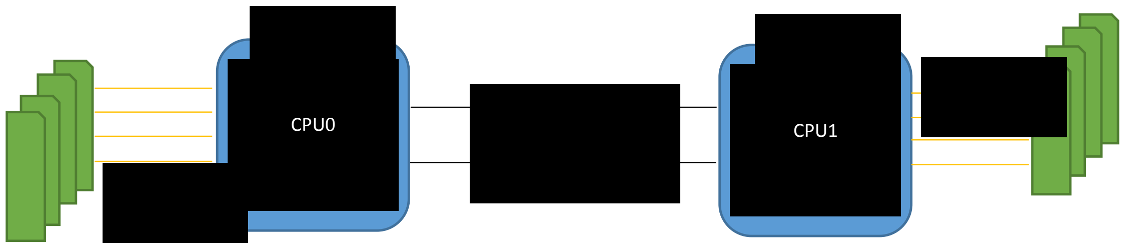


Figure 5.47: A single MPI task per socket that only employs a single CPU on the NUMA architecture. Blue squares are CPUs and green recangles are RAM modules.

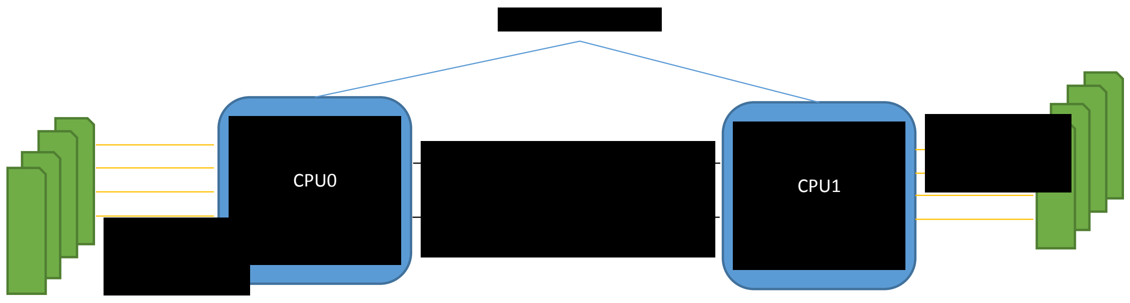


Figure 5.48: A single MPI task per node that employs the entire system including all available processors on the NUMA architecture. Blue squares are CPUs and green recangles are RAM modules.

system, such as the Stampede supercomputer, appropriate data distribution across different processors must be determined. Since in a heterogeneous environment the

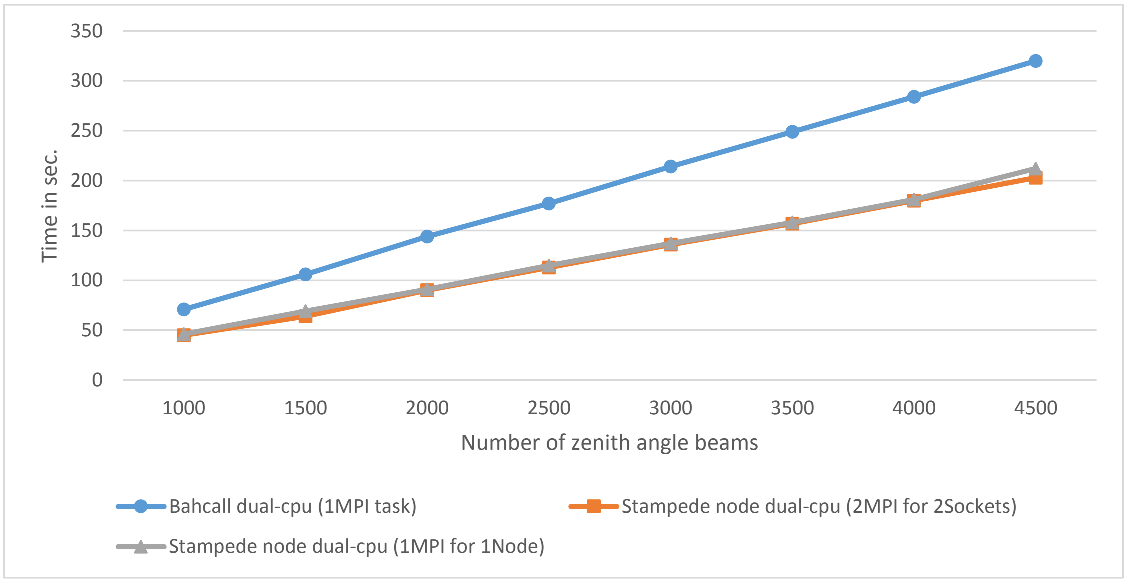


Figure 5.49: XFLAT benchmark utilizing different process configurations. Blue curve illustrates performance on the dual-socket Bahcall machine by employing only one MPI task. Orange curve illustrates performance on dual-socket Stampede node by employing one MPI task per socket. Grey curve illustrates the performance on Stampede node using one MPI task per entire node.

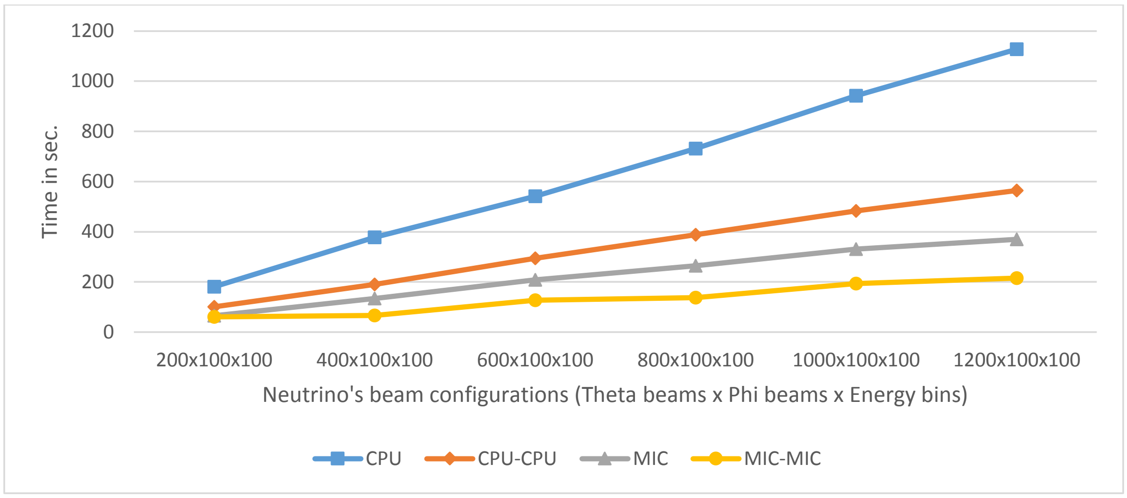


Figure 5.50: Illustration of XFLAT performance on a single Stampede node. Blue line illustrates single CPU performance. Orange line illustrates dual CPU performance. Grey line illustrates single MIC performance. Yellow line illustrates dual MIC performance. The timings were measured for $\sim$ 1000 radial steps.

computational capability of each processor may differ even within a node, there needs to be a precise mechanism to decide what portion of data should be placed on each processor based on its computational capabilities. If the code runs on a heterogeneous cluster with a load-imbalanced configuration, the MPI communication cost will be high. Due to load imbalance on nodes, the waiting time at synchronization points increases. This means that a few tasks may complete earlier and remain idle. They will then have to wait for the other tasks to reach the same synchronization point. In addition to the idle nodes, there may also be oversubscribed nodes.

The first heterogeneous XFLAT benchmark was designed to identify the "sweet spot" for the CPU and MIC on Stampede. The goal was to identify the optimal MIC to CPU load ratio. The motivation for this benchmark was that the raw performance of a processor, its FLOPS capability, and actual performance, which may be different based on the application's instructions, are not necessarily identical. Therefore, even when the raw performance of the CPU and MIC are known in advance, actual XFLAT performance may vary depending on the details of the specific physics modules and problem size. In addition, due to low-level differences in the underlying hardware, the performance of a processor might be satisfactory for many instructions but not satisfactory for other instructions. Therefore, depending on the instructions used in a code, the relative performance (CPU vs. MIC) may vary. Although the instruction set is similar for the CPU and MIC (both are x86-based), there are still several factors that can affect the final performance of the code. Hardware design factors such as the size and number of levels of cache memory, core frequency, memory bus frequency, number of hardware threads per core, memory bandwidth, and length of the vector registers, can all affect final performance dramatically. Therefore, both CPU and MIC must be benchmarked using the actual XFLAT code in order to find the true sweet spot.

For the first timing benchmark, the number of neutrino beams was manually chosen to be $1000 \times 100 \times 100$ (θ *angles* $\times$ ϕ *angles* $\times$ *Energy bins*), and the number of total radial steps for the calculation was fixed at 100. The number of neutrino beams assigned to the MIC and CPU varied unless the optimal load ratio for the overall node was found. As depicted in Fig. 5.51, the optimum MIC:CPU load ratio was in the [2.7–3.0] range. The benchmark was also repeated for another problem size with a different number of beams. Fig. 5.52 illustrates the benchmark with a different problem configuration. This second benchmark was performed with $10000 \times 10 \times 100$ (Θ *angles* $\times$ Φ *angles* $\times$ *Energy bins*) beams. The optimum point remained in the same range as for the previous benchmark.

In Fig. 5.53 and Fig. 5.54, the maximum time spent within MPI functions is shown. The maximum MPI time is the maximum time that either the CPU or MIC spent in MPI functions. Since communication timings fluctuated on the processors, only the maximum MPI times were recorded. When communication increases, overall code performance decreases, since time spent on communication is not be-

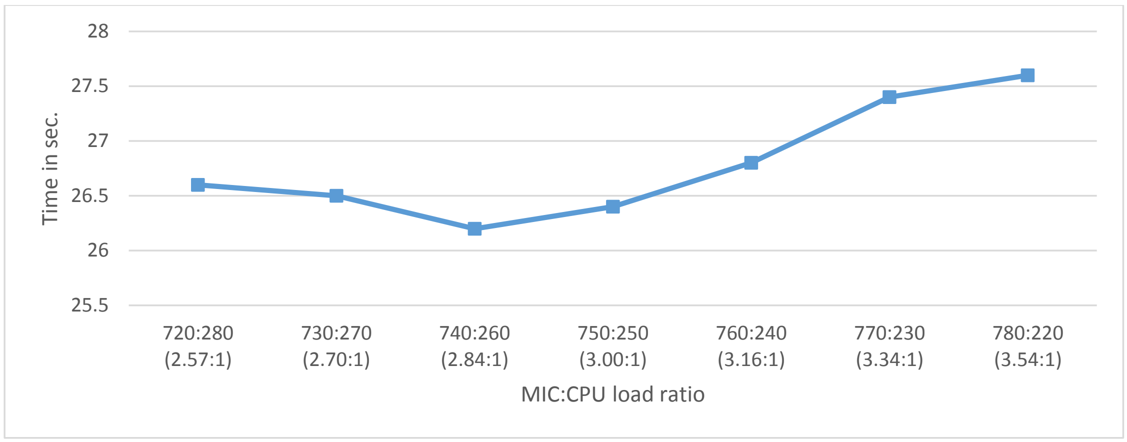


Figure 5.51: XFLAT timing benchmark on a single-node heterogeneous environment. Execution time shown for different MIC:CPU load ratios using 1000 zenith trajectories. Each point represents [number_of_theta_angles_on_MIC : number_of_theta_angles_on_CPU] (the normalized load ratio is shown in parenthesis).

ing performed. The benchmark was repeated for various MIC:CPU load ratios. A minimum (optimum) region is observed around a similar range as in Fig. 5.51 and

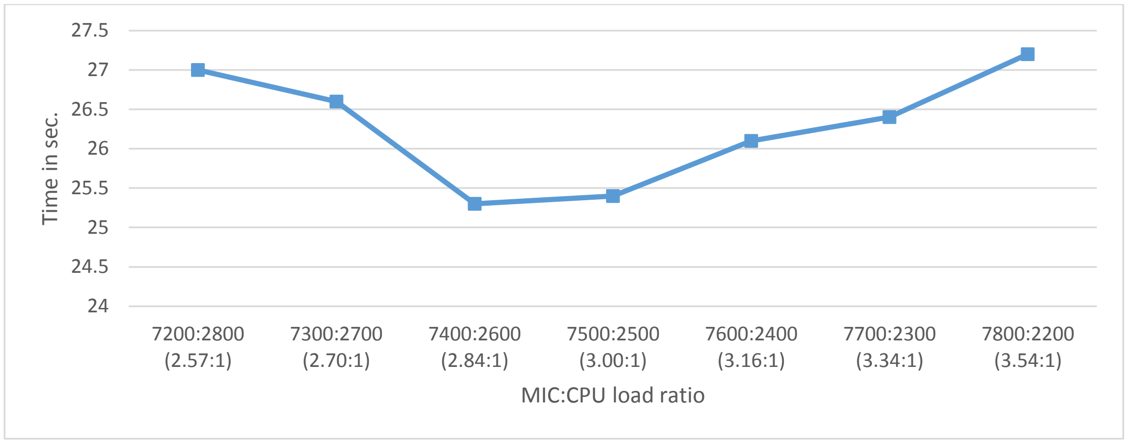


Figure 5.52: XFLAT timing benchmark on a single-node heterogeneous environment. Execution time shown for different MIC:CPU load ratios using 10k zenith trajectories. Each point represents [number_of_theta_angles_on_MIC : number_of_theta_angles_on_CPU] (the normalized load ratio is shown in parenthesis).

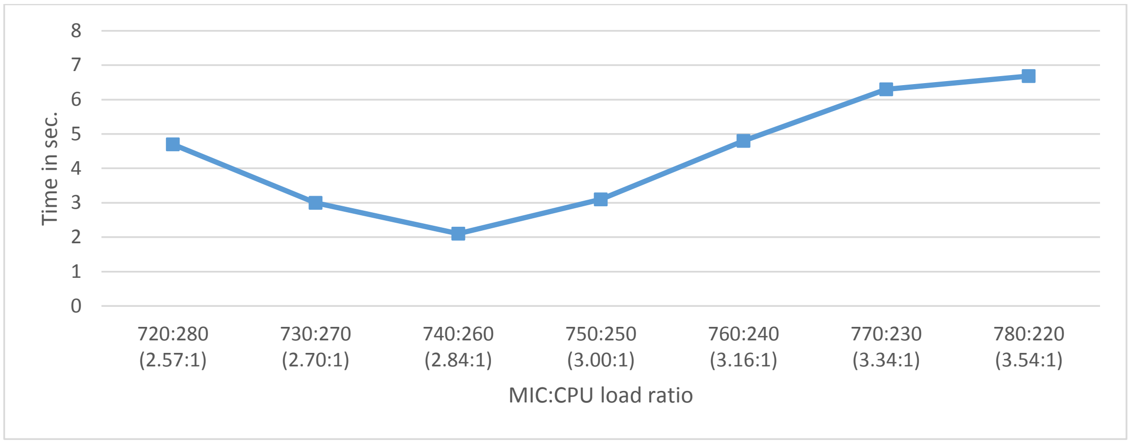


Figure 5.53: XFLAT timing benchmark for a single-node heterogeneous environment. The maximum time spent on MPI functions on either CPU or MIC is shown for different MIC:CPU load ratios and runs employing 1000 theta beams. Each point represents [number_of_theta_angles_on_MIC : number_of_theta_angles_on_CPU] (the normalized load ratio is shown in parenthesis).

Fig. 5.52. On both sides of the minimum region the time increases. This increase in timing is due to the fact that at least one processor is remaining idle and waiting for other tasks to reach the synchronization point (which is due to load imbalance between processors).

The best results were achieved when the load was distributed across both MIC and CPU based on their respective computational capabilities. Consequently, the MPI wait time was also minimized at the same point. The benchmarks show that on the Stampede nodes, the optimal MIC:CPU load ratio range is in the range of 2.8:1 to 3.0:1. With this optimum load ratio known the next step can be benchmarking the code on heterogeneous multi-node environments.

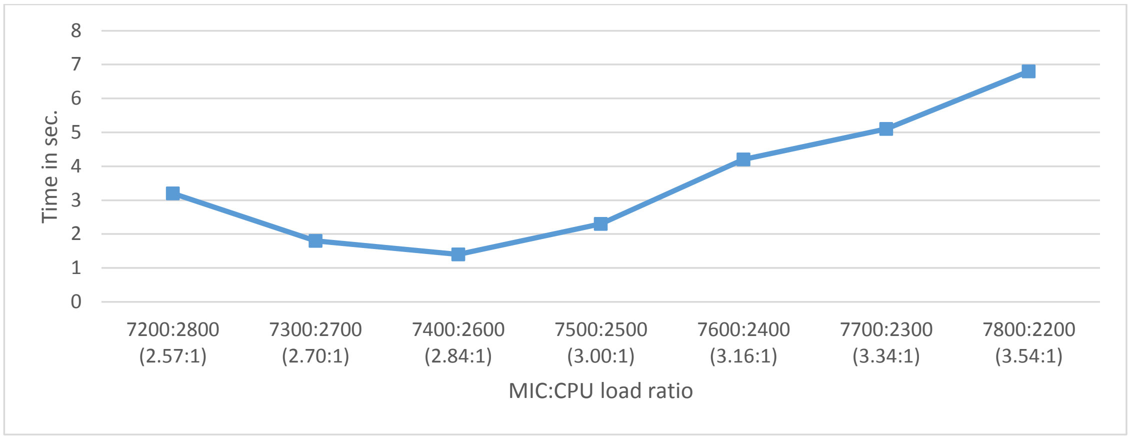


Figure 5.54: XFLAT timing benchmark for a single-node heterogeneous environment. The maximum time spent on MPI functions on either CPU or MIC is shown for different MIC:CPU load ratios and runs employing 10k theta beams. Each point represents [number_of_theta_angles_on_MIC : number_of_theta_angles_on_CPU] (the normalized load ratio is shown in parenthesis).

5.3.3 XFLAT performance on multi-node systems

In light of the previous benchmarks presented in the previous section the on-node MIC:CPU was set to 2.9:1 for all multi-node benchmarks, in order to minimize processor idle time.

The first benchmark employed one CPU and one MIC per Stampede node. The problem size was set to $10000 \times 10 \times 100$ beams and the number of processed radial steps was measured over a 100 second period of run time. The number of nodes was varied from 1 to 16 nodes. As can be seen in Fig. 5.55, behavior was virtually linear from one node up to 16 nodes. The 16-node benchmark utilized 32 MPI tasks overall. 16 of them employed 16 Xeon CPUs and the other 16 tasks employed the 16 Xeon Phis. The maximum time spent in MPI functions among all nodes was recorded, and the results are illustrated in Fig. 5.56.

Similar benchmarks were then performed on Stampede for the dual-CPU and

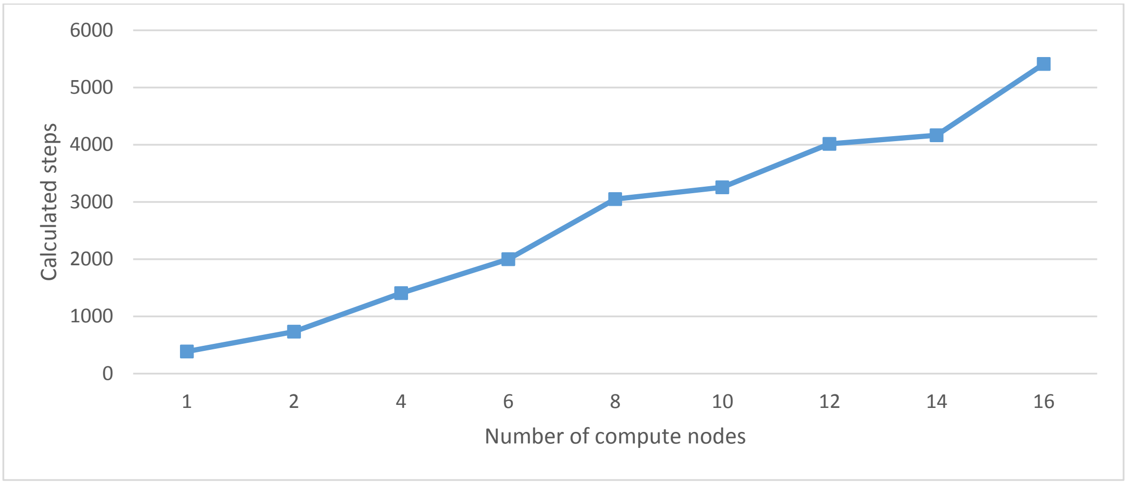


Figure 5.55: XFLAT heterogeneous multi-node performance on Stampede (one CPU and one MIC per node) for various numbers of compute nodes. The calculated radial steps were measured for a ~ 100 second run.

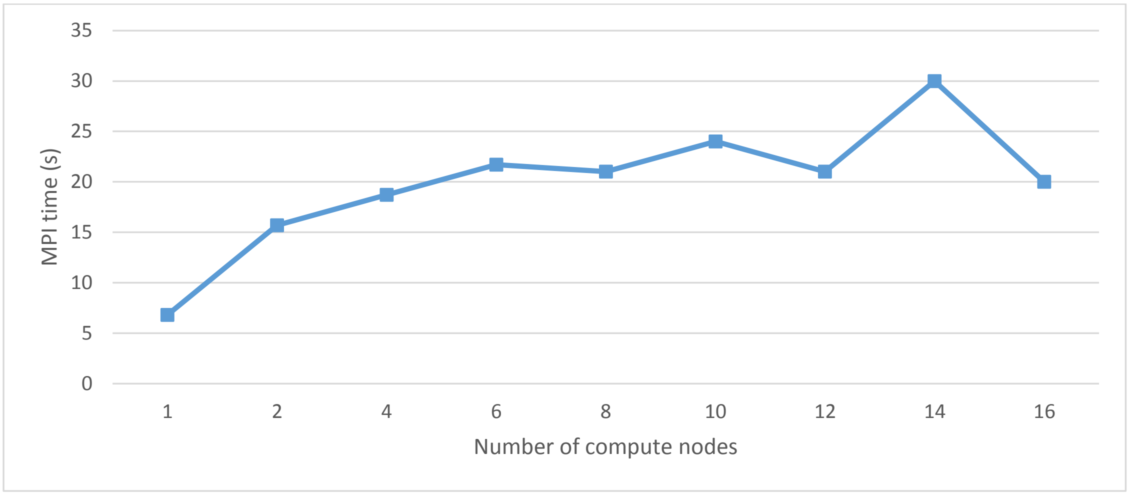


Figure 5.56: Maximum spent time within MPI functions for various numbers of nodes on Stampede (using one CPU and one MIC per node). The problem size was set to $10000 \times 10 \times 100$ beams and the number of processed radial steps was measured over ~ 100 second period of run time.

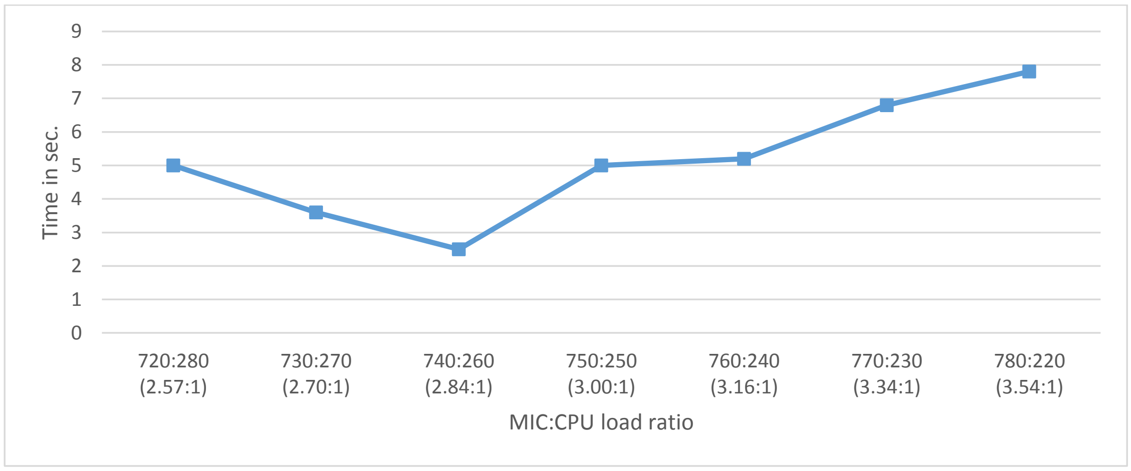


Figure 5.57: XFLAT timing benchmark on single-node (dual-CPU and dual-MIC) heterogeneous environment. The maximum time spent on MPI functions either CPU or MIC is shown for different MIC:CPU load ratios and runs employing 1000 theta beams. Each point represents [number_of_theta_angles_on_MIC : number_of_theta_angles_on_CPU] (the normalized load ratio is shown in parenthesis).

dual-MIC node configurations. On each node there were two MPI tasks on the two CPUs (one per CPU) and two MPI tasks on the two MICs (one per MIC).

As for the single-processor benchmarks, the MIC:CPU ratio was set to 2.9:1. Fig. 5.57 and Fig. 5.58 illustrate the maximum time of MPI functions for the $1000 \times 100 \times 100$ and $10000 \times 10 \times 100$ beam configurations, respectively.

Next, benchmarks were performed using the identical problem size as for the single processor benchmarks, but for varying numbers of nodes (1 to 16 compute nodes). If the single-node benchmark utilized four MPI tasks (two on the CPUs and two on the MICs), the 16-node benchmark utilized 64 MPI tasks. The results are presented in Fig. 5.59.

Surprisingly, for runs employing more than 8 nodes, performance did not improve. In fact, there were no significant improvement until the number of nodes was increased to 16. By employing 16 nodes, an unexpected jump in performance is

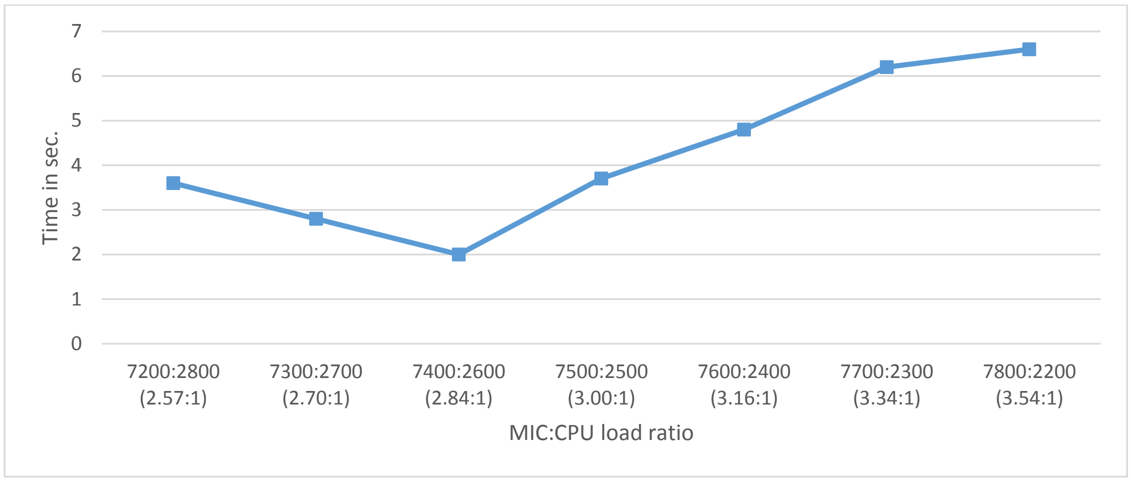


Figure 5.58: XFLAT timing benchmark on single-node (dual-CPU and dual-MIC) heterogeneous environment. The maximum time spent on MPI functions either CPU or MIC is shown for different MIC:CPU load ratios and runs employing 10k theta beams. Each point represents [number_of_theta_angles_on_MIC : number_of_theta_angles_on_CPU] (the normalized load ratio is shown in parenthesis).

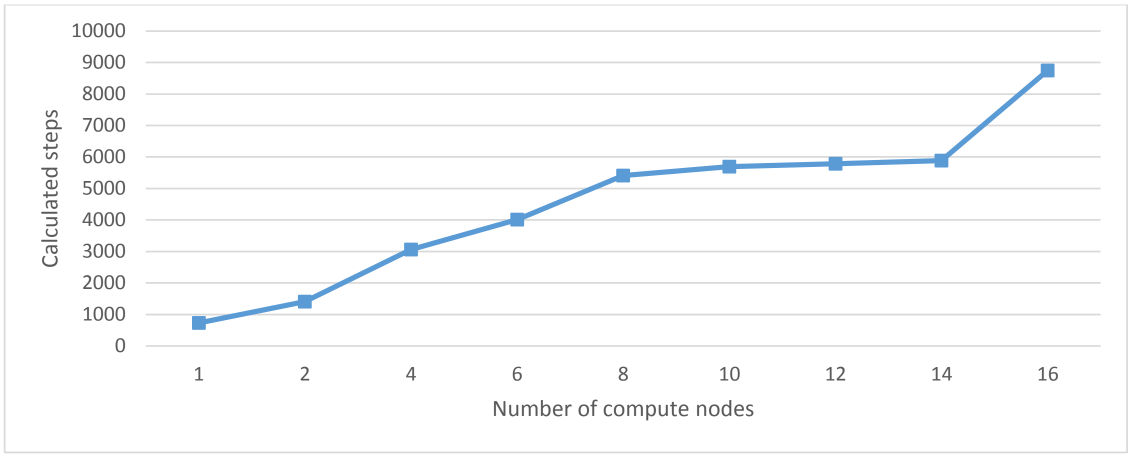


Figure 5.59: XFLAT heterogeneous multi-node performance on Stampede (two CPUs and two MICs per node) for various numbers of compute nodes. The calculated radial steps were measured for ~ 100 second run.

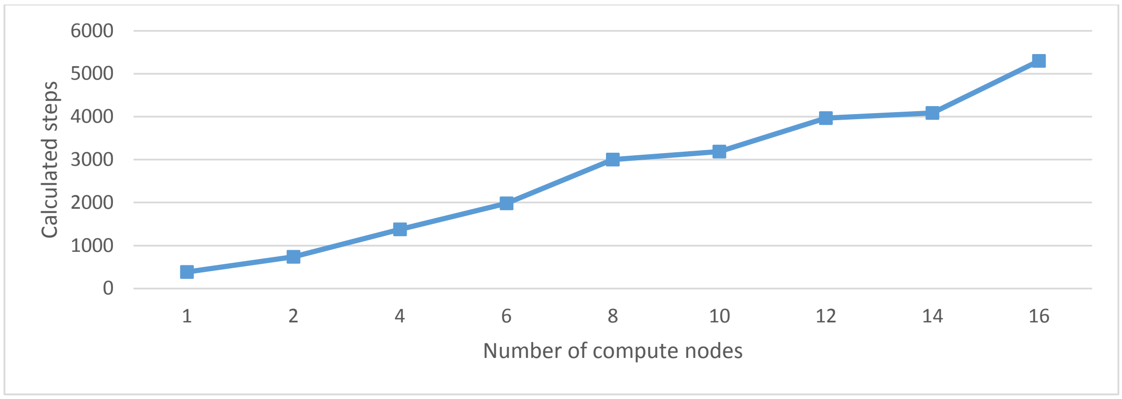


Figure 5.60: XFLAT heterogeneous multi-node performance on Stampede (two CPUs and two MICs per node) by employing a double amount of load on each processor. The calculated radial steps were measured for ~ 100 second run.

observed.

It was conjectured that the lack of improvement in performance for 8–16 nodes might be due to insufficient load on the Xeon Phi processors. In order to test this, a benchmark was designed in which a double amount of load were placed on each processor. As seen in Fig. 5.60, after placing twice more load on each component, the performance of the dual-processor benchmark appeared to be similar to that of the single-processor benchmark.

In addition, as another test, the single-processor benchmark was continued beyond 16 compute nodes. Consequently, the load on each processor eventually became similar to the load on the dual-processor benchmark (for instance, for the 32-node single-processsor benchmark, the load on a CPU was identical to the load on a CPU in the dual-processor benchmark using 16 nodes). As illustrated in Fig. 5.61, the same flat region that was observable in the dual-processor benchmark appeared again in going beyond 16 nodes.

As a result, it can be concluded that the flat region in the performance result

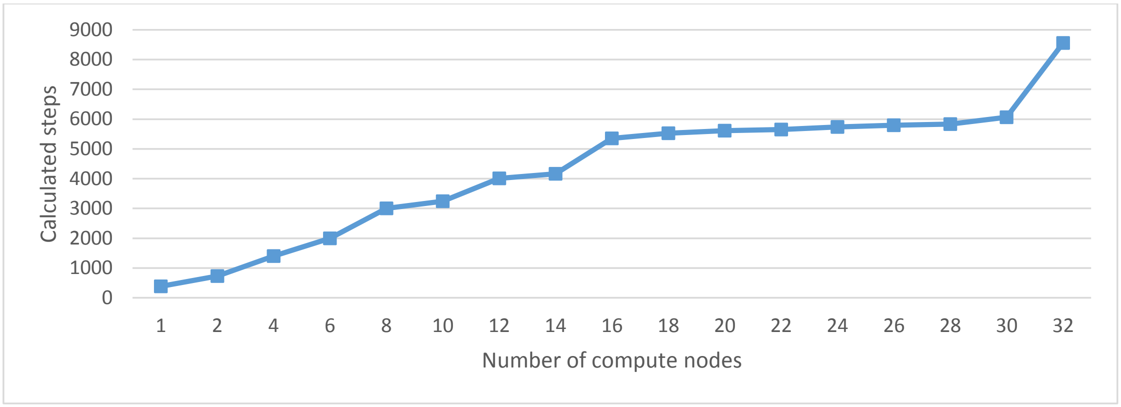


Figure 5.61: XFLAT heterogeneous multi-node performance on Stampede (one CPU and one MIC per node) employing up to 32 nodes. The calculated radial steps were measured for about $\sim$ 100 second run.

was not due to the number of employed nodes in the environment. One possible explanation for this behavior was the fact that by increasing the number of nodes, the overhead of sending and receiving MPI messages may affect overall performance. Consequently, in order to validate this hypothesis, the net amount of MPI messages' overhead should be extracted from the processors' idle time to figure out whether or not the overhead can affect overall XFLAT performance. Thus, detailed studies on MPI tasks were undertaken in order to understand the role of MPI overhead in XFLAT performance.

5.3.4 MPI communication overhead

In order to extract and measure the overhead of MPI communications from the idle (waiting) time at synchronization points, two sets of benchmarks were designed. In the first set, barriers were placed prior to each MPI synchronization point, and the timer was started only after the barrier line. Hence, only the time of the MPI communication section (the MPI overhead) was measured (see Fig. 5.62). As a result,

```
while (conditions)
{
    ... // Computations!

    // All of the process are synchronized at this point
    MPI_barrier(...);

    timer.Start();
    MPI_Send(...);
    timer.Stop();
    ...
}
```

Figure 5.62: Pseudocode illustrating measurement of MPI communication time excluding idle time by placing a barrier prior to the communication line.

all of the processes reached the synchronization point virtually at the same time.

In the second set of benchmarks the barrier line was removed, thus the idle time of each process was included in the overall time (see Fig. 5.63).

Consequently, following removal of the barrier, due to the load imbalance between the MIC and CPU, one of them reached the synchronization point earlier.

```
while (conditions)
{
    ... // Computations!

    timer.Start();
    MPI_Send(...);
    timer.Stop();
    ...
}
```

Figure 5.63: Pseudocode illustrating measurement of total MPI communication time including idle time of processes.

Therefore, it resulted in increased MPI time for that process. However, when there was a barrier exactly prior to the synchronization point, since both the MIC and the CPU reached the synchronization point virtually at the same time, the overhead of the waiting time was eliminated from the MPI function timing. Therefore, during applications run time when processes on different nodes reached the same synchronization point simultaneously, the processes' idle time remained minimized. As a result, by minimizing the idle time more time could be dedicated for computations.

The designed benchmarks were performed for both blocking and non-blocking MPI calls. Blocking functions are those for which all processes must reach the same point before leaving the function. For non-blocking functions, they can return immediately. In all of the following benchmarks the message size was equal to 8 double precision numbers, and the loop iteration count was 10k. In order to break the symmetry and to make sure that one process always arrived later, the computational section prior to the synchronization point was about three times higher on the Xeon Phi, hence the process on the CPU always reached the communication points earlier during the benchmarks without barriers (by at least tens of seconds). The results of the MPI send/receive timing are shown in Table 5.4 in which a barrier was presented before MPI was invoked and in Table 5.5 in which there was no barrier prior to MPI being invoked. Furthermore, a ping-pong benchmark was performed in which the first process sent data to the second process, and the second process then sent data to the first one. Table 5.6 and Table 5.7 show the results for the ping-pong benchmarks where barriers were presented and where barriers were removed before MPI invocation, respectively.

There are several interesting observations to be made about these results. First, it can be seen that the overhead of the MPI methods when initiated on the MIC side was higher (*i.e.*, when the CPU attempted to send data to the MIC, the overhead of the MPI method was lower than in the case where the MIC attempted to send data

Table 5.4: Timing measurement for MPI Send/Receive between two nodes when a barrier was presented prior to the MPI invocation. The first column shows the direction of the message flow. The second and third columns show the communication time in seconds for the MIC and CPU, respectively.

Send/Recv	MIC comm. time	CPU comm. time
Blocking Sends		
CPU→MIC	2.2	0.05
MIC→CPU	0.7	0.8
Non-blocking Sends		
CPU→MIC	2.0	0.06
MIC→CPU	0.8	1.1

Table 5.5: Timing measurement for MPI Send/Receive between two nodes when there was no barrier prior to the MPI invocation. The first column shows the direction of the message flow. The second and third columns show the communication time in seconds for the MIC and CPU, respectively.

Send/Recv	MIC comm. time	CPU comm. time
Blocking Sends		
CPU→MIC	0.9	37
MIC→CPU	2.1	39
Non-blocking Sends		
CPU→MIC	0.4	0.03
MIC→CPU	2.2	39

to the CPU). Furthermore, the task to which the data was sent (the receiver task) via the PCI Express bus had more wasted (idle) time (see Table 5.4 and Table 5.6).

Similar to the send and receive benchmarks, the next benchmarks measured MPI communication time for the MPI broadcast messages. The broadcast benchmarks showed that for the blocking calls, the node on which the message was initialized wasted slightly less time for sending messages (Table 5.8). However, for the benchmark without barrier, when MIC was the receiver, its time at the synchronization point was less than the time when the MIC was a sender (Table 5.9). Note that the non-blocking broadcast data are not usable on the receiver side without making sure

Table 5.6: Timing measurement for double MPI Send/Receive (ping-pong) between two nodes when a barrier was presented prior to the MPI invocation. The first column shows the direction of the message flow. The second and third columns show the communication time in seconds for the MIC and CPU, respectively.

Send/Recv	MIC comm. time	CPU comm. time
Blocking Sends		
CPU→MIC+MIC→CPU	2.8	2.7
MIC→CPU+CPU→MIC	3.6	0.8
Non-blocking Sends		
CPU→MIC+MIC→CPU	2.6	2.9
MIC→CPU+CPU→MIC	3.6	1.1

Table 5.7: Timing measurement for double MPI Send/Receive (ping-pong) between two nodes when there was no barrier prior to the MPI invocation. The first column shows the direction of the message flow. The second and third columns show the communication time in seconds for the MIC and CPU, respectively.

Send/Recv	MIC comm. time	CPU comm. time
Blocking Sends		
CPU→MIC+MIC→CPU	3	40.7
MIC→CPU+CPU→MIC	5.5	41
Non-blocking Sends		
CPU→MIC+MIC→CPU	3.1	40.2
MIC→CPU+CPU→MIC	5.5	41.1

that the message has arrived completely before trying to touch the buffer.

For the MPI reduction functions there was no non-blocking benchmark, since the nature of MPI reduction function is blocking. As shown in Table 5.10 and Table 5.11, for the MPI all-reduce functions, there was no significant difference between the reduction with summation and reduction with maximum. Similar to the previous results, removing the barrier caused a slight increase in the timing on the MIC side.

In all of the kernel benchmarks, the MPI invocation overhead was subtle. Consequently, in addition to benchmarks the actual XFLAT code, with the actual in-

Table 5.8: Timing measurement for MPI broadcast between two nodes when a barrier was presented prior to the MPI invocation. The first column shows the direction of the message flow. The second and third columns show the communication time in seconds for the MIC and CPU, respectively.

Broadcast	MIC comm. time	CPU comm. time
Blocking Broadcasts		
CPU→MIC	3.2	0.05
MIC→CPU	0.8	0.8
Non-blocking Broadcasts		
CPU→MIC	0.2	0.00
MIC→CPU	0.9	0.01

Table 5.9: Timing measurement for MPI broadcast between two nodes when there was no barrier prior to the MPI invocation. The first column shows the direction of the message flow. The second and third columns show the communication time in seconds for the MIC and CPU, respectively.

Broadcast	MIC comm. time	CPU comm. time
Blocking Broadcasts		
CPU→MIC	1.0	37.2
MIC→CPU	2.1	38.1
Non-blocking Broadcasts		
CPU→MIC	0.4	0.04
MIC→CPU	1.6	0.02

structions and messages, was benchmarked on Bahcall. The problem size was set to $555 \times 10 \times 100$ ($\vartheta \times \varphi \times EBins$) neutrino beams. The time for completing 10k radial steps was measured and the benchmark was repeated for various MIC:CPU load ratios. The results of the benchmarks in which barriers were presented before each MPI synchronization and when the barriers were removed, are shown in Table 5.12 and Table 5.13, respectively.

As shown in Table 5.12, the time of the MPI communications in XFLAT was not minimized at any point and fluctuated by varying the load ratio. That result was not expected since a barrier was placed before each MPI communication point. Thus, the

Table 5.10: Timing measurement for MPI reduction between two nodes when a barrier was presented prior to the MPI invocation. The first column shows the direction of the message flow. The second and third columns show the communication time in seconds for the MIC and CPU, respectively.

All reduction	MIC comm. time	CPU comm. time
All_reduce(SUM)	2.5	2.0
All_reduce(MAX)	2.3	2.2

Table 5.11: Timing measurement for MPI reduction between two nodes when there was no barrier prior to the MPI invocation. The first column shows the direction of the message flow. The second and third columns show the communication time in seconds for the MIC and CPU, respectively.

All reduction	MIC comm. time	CPU comm. time
All_reduce(SUM)	3.2	39
All_reduce(MAX)	3.4	40.1

MPI communication time should not be the function of load ratio. According to the MPI standard, the `MPI_Barrier()` function only guarantees that at some point all the processes are within the barrier function; however, it does not guarantee that all of them exit the barrier function at the same time [LLNL, 2015]. Therefore, the way that the barrier function is implemented may affect the result. Hence, the observed fluctuations may be due to the utilized MPI library on the Bahcall machine, which was the MPICH library. Although the MPICH library [MPICH, 2015] is one of the most popular and widely-used implementations of the MPI standard, an HPC design was not the highest priority for its implementation, unlike other MPI implementations such as the MVAPICH [MVAPICH, 2015] or Intel MPI [Intel, 2015a] libraries (another widely-used MPI library is Open MPI [MPI, 2015] which was not available at the time of analysis for the Intel MIC architecture).

Since the Intel MPI library is not available as free software and MVAPICH is the MPI library used over InfiniBand or 10GigE links on multi-node systems, it was not possible to use these libraries on the Bahcall machine (Bahcall was equipped

Table 5.12: XFLAT performance benchmark on Bahcall (using MPICH library) when barriers were placed prior to each MPI synchronization. The first column shows the load ratios of MIC to CPU (number of beams on the MIC and CPU). The second and third columns show the communication time in seconds on the MIC and CPU, respectively. The fourth column lists overall run time in seconds.

MIC:CPU (beams#)	MIC comm. time	CPU comm. time	Overall time
5.0:1 (463:092)	17.7	13.0	254.3
4.5:1 (454:101)	16.8	12.5	217.6
4.0:1 (444:111)	16.5	13.3	215.3
3.5:1 (432:123)	16.7	15.5	214.7
3.0:1 (416:139)	14.1	14.1	209.0
2.5:1 (397:158)	12.0	18.3	205.8
2.0:1 (370:185)	11.3	20.9	245.2
1.5:1 (333:222)	11.3	23.8	262.6
1.0:1 (278:277)	11.4	21.3	332.1

with neither the InfiniBand cards nor the 10GigE cards). Fortunately, the Stampede supercomputer utilizes the Intel MPI library. Consequently, the XFLAT code and the Intel MPI library were benchmarked on one of the Stampede compute nodes. The benchmarks were repeated with the same neutrino beam configuration and the same iteration count. As can be seen in Table 5.14 and Table 5.15, unlike the previous results, the MPI communications time for benchmarks with barriers remained constant. Therefore, the previously observed fluctuations in the benchmark results (with barriers) were due only to the MPICH library design and implementation. By switching to the Intel MPI library the expected results were achieved. In addition, the optimum MIC:CPU load ratio appeared to be around 3:1, which was in agreement with the results of the previous benchmarks.

Nevertheless, if the number of beams on the MIC is not sufficient, the distribution of beams over threads may affect results. The previous benchmarks were therefore repeated using ten times more load on each processor (see Table 5.16 and Table 5.17).

In all of the previous benchmarks, the optimum MIC:CPU load ratio was around

Table 5.13: XFLAT performance benchmark on Bahcall (using MPICH library) when barriers were remove prior to each MPI synchronization. The first column shows the load ratios of MIC to CPU (number of beams on the MIC and CPU). The second and third columns show the communication time in seconds on the MIC and CPU, respectively. The fourth column lists overall run time in seconds.

MIC:CPU (beams#)	MIC comm. time	CPU comm. time	Overall time
5.0:1 (463:092)	19.8	134.1	232.8
4.5:1 (454:101)	19.4	90.1	199
4.0:1 (444:111)	19.1	77.1	197.3
3.5:1 (432:123)	19.3	65.5	195.7
3.0:1 (416:139)	18.0	48.5	191.4
2.5:1 (397:158)	26.3	24	187.1
2.0:1 (370:185)	63.9	14.8	219.8
1.5:1 (333:222)	89.3	12.7	236.7
1.0:1 (278:277)	171.8	13	310

3:1. This fact indicates that on Stampede, the optimum load on the MIC should be about three times higher than the CPU load in order to achieve optimum performance. These results are consistent with the previous sweet spot identification on Stampede.

Nevertheless, in order to find a more precise number for the sweet spot, a more refined range for the load ratios should be searched. From the previous results, it could be concluded that the range of 2.5–3.5 load ratios should be searched so as to find the sweet spot. Therefore, two benchmarks were performed on the refined range to find the precise sweet spot. The first benchmark utilized $555 \times 10 \times 100$ ($\vartheta \times \varphi \times EBins$) number of neutrino beams similar to the previous benchmark (see Table 5.18), and the second benchmark employed a different neutrino beam number ($12000 \times 10 \times 100$). Consequently, the number of neutrino beams on each processor differed from the first benchmark (see Table 5.19). The timings were measured for 1000 radial steps. The result shows that the more accurate optimum MIC:CPU load ratio was around 2.9:1.0. Fig. 5.64 depicts the separately scaled XFLAT performance for two bench-

Table 5.14: XFLAT performance benchmark on Stampede (using Intel MPI library) when barriers were placed prior to each MPI synchronization. The first column shows the load ratios of MIC to CPU (number of beams on each side). The second and third columns show the communication time in seconds on the MIC, CPU, respectively. The fourth column lists overall run time in seconds.

MIC:CPU (beams#)	MIC comm. time	CPU comm. time	Overall time
5.0:1 (463:092)	3	3.5	168.9
4.5:1 (454:101)	2.9	3.3	166.2
4.0:1 (444:111)	2.9	3.3	164.1
3.5:1 (432:123)	2.9	3.3	163.7
3.0:1 (416:139)	3	3.3	163
2.5:1 (397:158)	3	3.2	165.1
2.0:1 (370:185)	3	3	186
1.5:1 (333:222)	2.9	2.8	215.9
1.0:1 (278:277)	2.9	2.9	265.2

marks. Similar to the previous findings, if enough load is placed on the MIC and CPU, the single-node MIC:CPU ratio sweet spot is about 2.9:1 on Stampede.

In all of the benchmarks with a barrier, $\sim 3-4$ seconds of overall MPI overhead was always observed. That overhead was due to invoking MPI functions alone, and it was not possible to exclude it from XFLAT runs on multi-node environments. The overhead was still small enough that it does not cause any major performance hit. Therefore, none of the benchmarks could explain the reason for previously-observed flat region on the heterogeneous multi-node benchmarks.

On a single node the MPI overhead was not problematic, however, for the multi-node benchmarks MPI communication was heavily used. In the next section, XFLAT multi-node performance benchmarks are simulated so as to eliminate this MPI overhead.

Table 5.15: XFLAT performance benchmark on Stampede (using Intel MPI library library) when barriers were removed prior to each MPI synchronization. The first column shows the load ratios of MIC to CPU (number of beams on each side). The second and third columns show the communication time in seconds on the MIC, CPU, respectively. The fourth column lists overall run time in seconds.

MIC:CPU (beams#)	MIC comm. time	CPU comm. time	Overall time
5.0:1 (463:092)	6	80.8	166.9
4.5:1 (454:101)	5.8	71	165
4.0:1 (444:111)	5.7	60.9	163.6
3.5:1 (432:123)	5.7	39.7	162.6
3.0:1 (416:139)	6.1	26.7	160.8
2.5:1 (397:158)	10.1	15.3	162.4
2.0:1 (370:185)	30	6.8	181
1.5:1 (333:222)	64.1	6.6	211.4
1.0:1 (278:277)	118.2	6.1	260.4

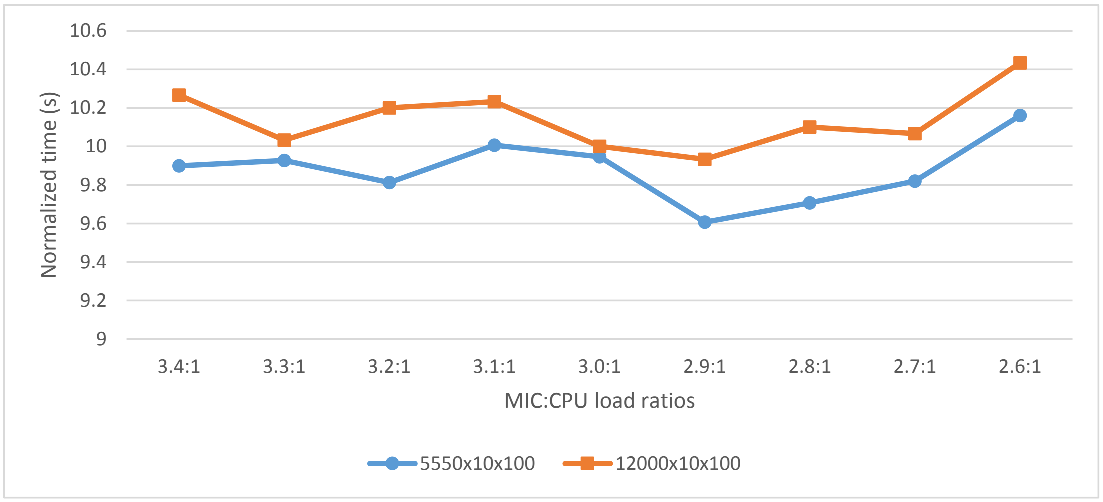


Figure 5.64: Stampede single-node benchmark for two different neutrino beam configurations, $5550 \times 10 \times 100$ and $12000 \times 10 \times 100$, for various MIC:CPU load ratios. In order to compare the two curves on the same scale, the timings for them were scaled (normalized) (actual times in Table 5.18 and Table 5.19).

Table 5.16: XFLAT performance benchmark, by employing ten times more load on each processor, on Stampede (using Intel MPI library) when barriers were placed prior to each MPI synchronization. The first column shows the load ratios of MIC to CPU (number of beams on each side). The second and third columns show the communication time in seconds on the MIC, CPU, respectively. The fourth column lists overall run time in seconds.

MIC:CPU (beams#)	MIC comm. time	CPU comm. time	Overall time
5.0:1 (4630:0920)	3.5	3.9	1576
4.5:1 (4540:1010)	3.5	3.9	1584
4.0:1 (4440:1110)	3.4	3.9	1544
3.5:1 (4320:1230)	3.6	4.1	1488
3.0:1 (4160:1390)	3.6	3.8	1435
2.5:1 (3970:1580)	3.6	3.7	1514
2.0:1 (3700:1850)	3.4	3.5	1742
1.5:1 (3330:2220)	3.5	3.5	2103
1.0:1 (2780:2770)	3.4	3.5	2562

5.3.5 Eliminating inter-node message passing

It is possible to design a benchmark to simulate a multi-node load run on a single node without including any inter-node communication. This can be achieved by placing the amount of load on a single machine equal to the load on a node in a multi-node environment. For fixed problem size, in a multi-node environment, by increasing the number of compute nodes, the amount of load on a single node decreases. A multi-node simulation on a single node measures computational effects only (no inter-node communication can be performed) by adjusting the load on the single-CPU and single-MIC configurations, accordingly.

To perform these tests, the initial problem size was set to $10000 \times 10 \times 100$ neutrino beams for a single node. As the number of nodes was increased, the load on a single compute node was adjusted to be equal as the load of a node in the analogous multi-node configuration. The benchmark was run for ~ 100 seconds and the calculated integer number of radial steps was measured. As illustrated in Fig. 5.65, the flat

Table 5.17: XFLAT performance benchmark, by employing ten times more load on each processor, on Stampede (using Intel MPI library) (using ten times more load on each processor) when barriers were removed prior to each MPI synchronization. The first column shows the load ratios of MIC to CPU (number of beams on each side). The second and third columns show the communication time in seconds on the MIC, CPU, respectively. The fourth column lists overall run time in seconds.

MIC:CPU (beams#)	MIC comm. time	CPU comm. time	Overall time
5.0:1 (4630:0920)	6	722	1587
4.5:1 (4540:1010)	6	671	1584
4.0:1 (4440:1110)	6	542	1543
3.5:1 (4320:1230)	7	313	1480
3.0:1 (4160:1390)	10	222	1471
2.5:1 (3970:1580)	103	77	1502
2.0:1 (3700:1850)	394	58	1724
1.5:1 (3330:2220)	878	54	2055
1.0:1 (2780:2770)	1500	46	2531

region in XFLAT performance is still observed.

Since the flat region appeared only when the number of nodes increased beyond

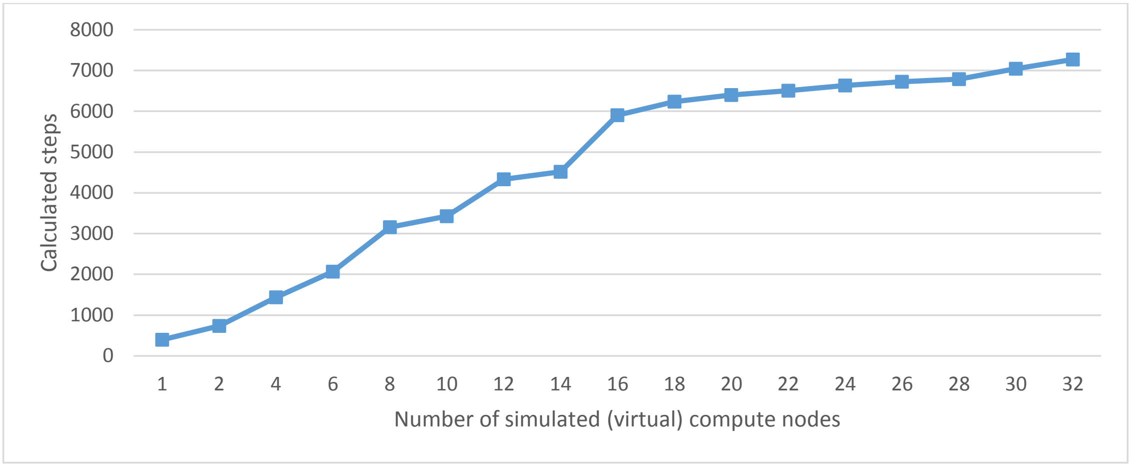


Figure 5.65: Stampede single-node (one CPU and one MIC) multi-node simulation benchmark. The x-axis corresponds to the effective number of compute nodes based on load. The number of calculated radial steps was measured for ~ 100 seconds.

Table 5.18: XFLAT performance benchmark on Stampede (using $5550 \times 10 \times 100$ neutrino beams) to find more refined MIC:CPU load ratio range. The first column shows the load ratios of MIC to CPU (number of beams on each side). The second and third columns show the communication time in seconds on the MIC, CPU, respectively. The fourth column lists overall run time in seconds.

MIC:CPU (beams#)	MIC comm. time	CPU comm. time	Overall time
3.4:1 (4289:1261)	6	347	1485
3.3:1 (4260:1290)	7	322	1489
3.2:1 (4229:1321)	6	281	1472
3.1:1 (4196:1354)	7	280	1499
3.0:1 (4163:1387)	8	242	1492
2.9:1 (4127:1423)	23	163	1441
2.8:1 (4090:1460)	34	143	1456
2.7:1 (4050:1500)	53	121	1473
2.6:1 (4008:1542)	108	67	1524

16 nodes (*i.e.* performance did not improve by increasing the number of nodes to 18, 20, 22, *etc.*). A load equal to the load of a single node in the 18-node, 20-node, 22-node configurations was placed in order to study the performance in this region. In addition, the MIC:CPU load ratio also varied in order to study the effect of the processor idle time on overall performance. Each benchmark was run for $\sim$ 100 seconds and the time spent at synchronization points was also recorded. (Although no inter-node MPI communications were involved, either the CPU or the MIC could reach the synchronization points sooner, resulting in measurable idle time).

Fig. 5.66 is a plot of MPI idle time as a function of MIC:CPU load ratio for a fixed problem size (simulated 18-node load). The times for the MPI synchronization points were recorded separately for the MIC and CPU. The total load was set to $555 \times 10 \times 100$ neutrino beams, which was the load analogous to the load on a node in the 18-node configuration. Likewise, Fig. 5.67 and Fig. 5.68 illustrate the benchmark results for a single node using $500 \times 10 \times 100$ and $455 \times 10 \times 100$ neutrino beams (equal to the load of a single node in the 20-node and 22-node environments), respectively.

Table 5.19: XFLAT performance benchmark on Stampede (using $12000 \times 10 \times 100$ neutrino beams) to find more refined MIC:CPU load ratio range. The first column shows the load ratios of MIC to CPU (number of beams on each side). The second and third columns show the communication time in seconds on the MIC, CPU, respectively. The fourth column lists overall run time in seconds.

MIC:CPU (beams#)	MIC comm. time	CPU comm. time	Overall time
3.4:1 (9273:2727)	0.08	6.28	30.8
3.3:1 (9209:2791)	0.08	5	30.1
3.2:1 (9143:2857)	0.08	4.9	30.6
3.1:1 (9073:2927)	0.09	4.5	30.7
3.0:1 (9000:3000)	.3	3.2	30
2.9:1 (8923:3077)	.7	2.2	29.8
2.8:1 (8842:3158)	1	2	30.3
2.7:1 (8757:3243)	2	1.1	30.2
2.6:1 (8667:3333)	3	1.1	31.3

These results all show normal behavior, *i.e.*, by changing the load ratio, the timing on one side increases and on the other side decreases. Therefore, MPI overhead does not explain the previously observed flattening behavior in the performance trend as the problem size is kept fixed and number of nodes is increased.

The other factor that can potentially affect the XFLAT performance is the processors' internal load imbalance. Modern processors, especially the Xeon Phi, are equipped with many cores and threads. Thus, the way that the neutrino beams are distributed across threads may affect overall performance of a processor. Hence, further benchmarks on multi-node environment were required, and for a controlled single-processor-type (homogeneous node) environment.

5.3.6 The distribution of neutrino beams across threads

Since the number of available hardware threads on each Stampede Xeon Phi is 244, and the number of threads on its CPU is 8, an issue related to load distribution

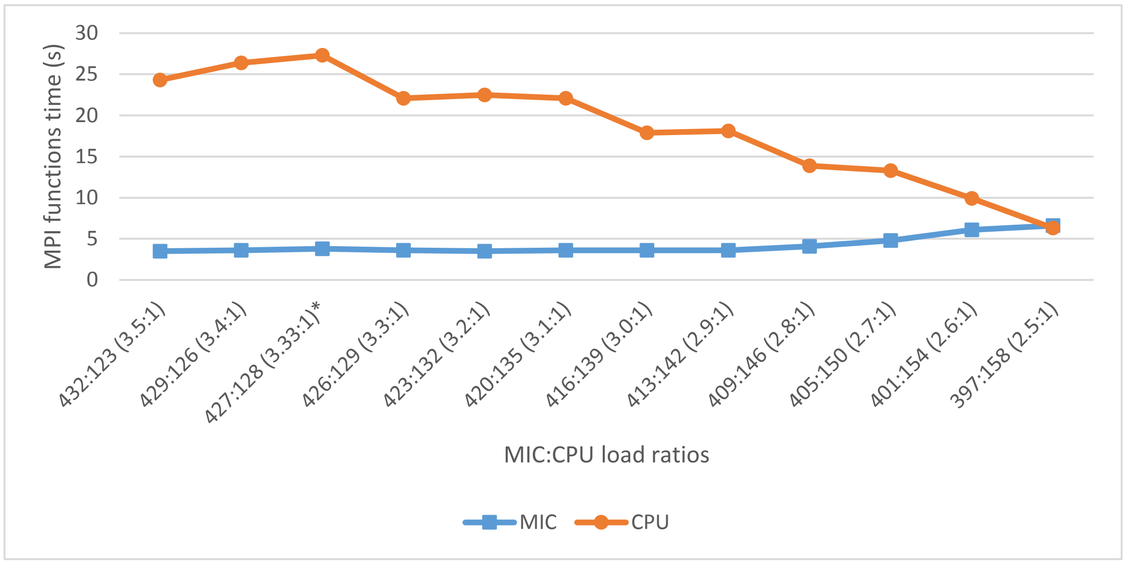


Figure 5.66: Performance of a single node for the simulated 18-node run (single CPU and single MIC) for various MIC:CPU load ratios. The wasted time at MPI synchronization functions were measured separately for the CPU and MIC. The calculated number of radial steps was measured for a $\sim$ 100 second run. The load ratio marked with $*$ indicates where the number of beams divided by the number of threads on the MIC is an integer.

can arise when the load is imbalanced across the Xeon Phi's threads. When the number of neutrino beams is not divisible by the number of available threads, a few threads may end up receiving more tasks for processing, thus causing load imbalance. Furthermore, the load imbalance on threads may be related to the observed flat region in the XFLAT heterogeneous benchmarks. In order to further investigate this issue, CPU-only and MIC-only benchmarks are required. XFLAT should be able to scale well on both homogeneous and heterogeneous environments, including environments equipped with only one type of processor.

Other important benchmarks reported here include the multi-node CPU-only and MIC-only benchmarks. These benchmarks utilize two CPUs or two MICs on Stampede's node. For these runs, the problem size was set to $1000 \times 10 \times 100$ and

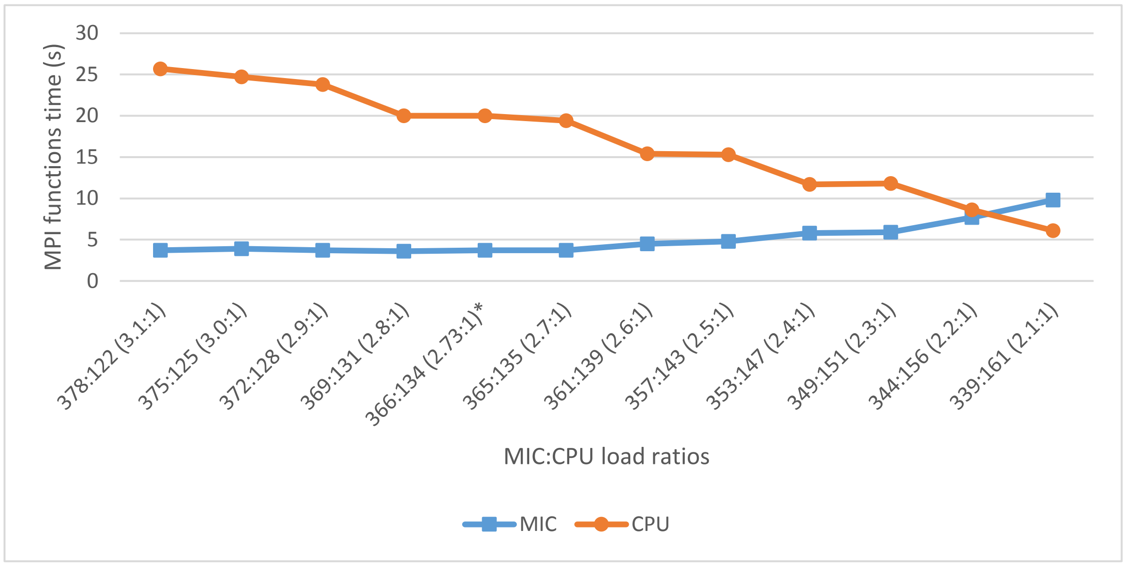


Figure 5.67: Performance of a single node for the simulated 20-node run (single CPU and single MIC) for various MIC:CPU load ratios. The wasted time at MPI synchronization functions were measured separately for the CPU and MIC. The calculated number of radial steps was measured for a $\sim$ 100 second run. The load ratio marked with $*$ indicates where the number of beams divided by the number of threads on the MIC is an integer.

the calculated number of radial steps for $\sim$ 100 seconds was measured. The results of the CPU and MIC benchmarks are depicted in Fig. 5.69. For the CPU-only benchmark, the code scaled linearly; however, for the MIC-only benchmark the flat region was still noticeable.

The flat region appeared only when the number of compute nodes increased beyond a particular value. Since, for a particular problem size, increasing the number of nodes causes the number of neutrino beams per processor to decrease, the issue may be related to the number of available neutrino beams per hardware thread. In fact, with a simple calculation it can be shown that the reason for the flat region in the Xeon Phi's performance is due to load imbalance. For example, as depicted in Fig. 5.69, when the number of compute nodes was 20, the load on a single MIC

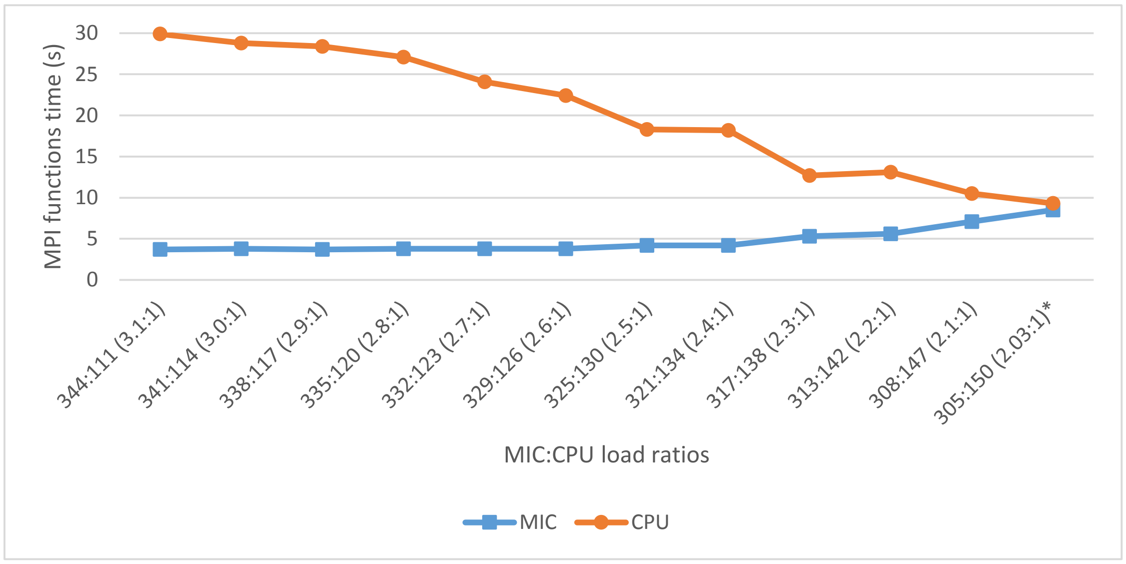


Figure 5.68: Performance of a single node for the simulated 22-node run (single CPU and single MIC) for various MIC:CPU load ratios. The wasted time at MPI synchronization functions were measured separately for the CPU and MIC. The calculated number of radial steps was measured for a ∼ 100 second run. The load ratio marked with ∗ indicates where the number of beams divided by the number of threads on the MIC is an integer.

was equal to 250 zenith angle beams. Since this number of beams needed to be distributed over 244 hardware threads, all of the threads performed calculations on 1 beam except for 6 threads to which 2 beams were assigned (250 mod 244 = 6). Hence, at the end of every iteration, all of the threads except 6 had to remain idle waiting for the last 6 threads to complete. The resulted in a performance hit, and the outset of the flat performance region. On the other hand, when the number of compute nodes was 22, the load on each MIC was equal to 227 beams, thus the computational load on all of the threads were nearly the same. There was no need for threads to wait for only a few threads to complete their tasks.

If the reason behind the flat region is truly due to the load distribution, this behavior should be reflected in other benchmarks with different configurations and

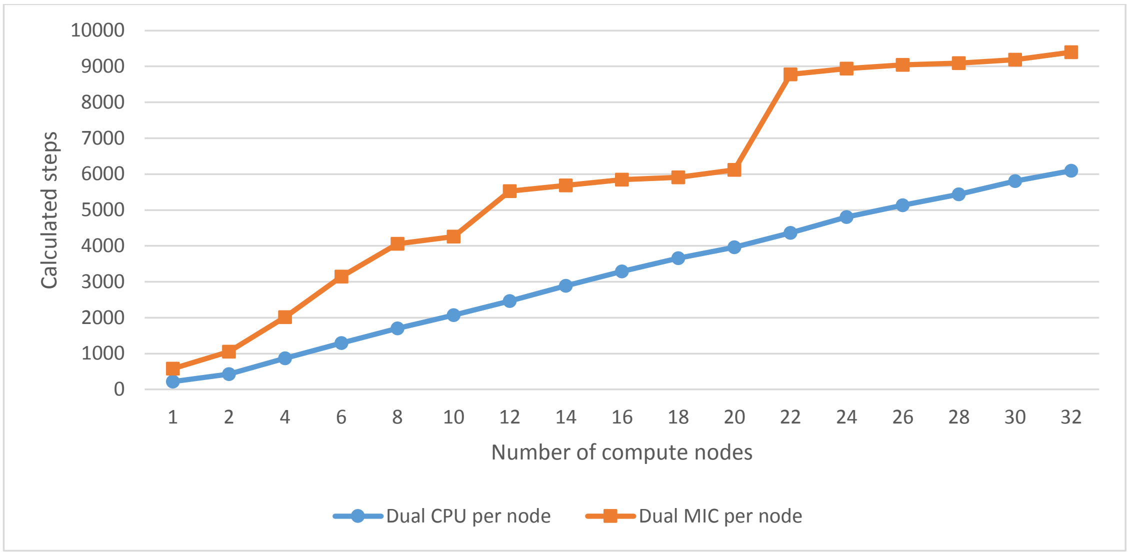


Figure 5.69: Multi-node homogeneous (CPU-only and MIC-only) benchmarks on Stampede. The blue line (CPU) shows that the performance of the CPU scaled virtually linearly by increasing the number of nodes. However, the orange line (MIC), illustrates that the performance of the MIC scaled non-linearly. The problem size was set to $1000 \times 10 \times 100$ and the calculated number of radial steps for about ~ 100 seconds was measured. Each node was equipped with two CPUs and two MICs.

loads as well. Therefore, another benchmark on the single-CPU and single-MIC configurations was performed in which the multi-node benchmarks were simulated by adjusting the load ratio on processors based on the load of a node in the multi-node environment, as described previously. As a result, there was no inter-node communications. In addition, the new benchmark employed a different problem size ($3000 \times 30 \times 100$ beams) with the 2.9:1 MIC to CPU load ratio. The number of calculated radial steps was measured for a ~ 100 second run.

As shown in Fig. 5.70, the flat behavior is observable for benchmarks with 6 to 10 nodes. For instance, for the 8-node run, there were 375 zenith beams on one node for which 279 were allocated on the Xeon Phi. Consequently, a very low number of threads (35 out of 244 threads) were allocated twice as many beams, so that all

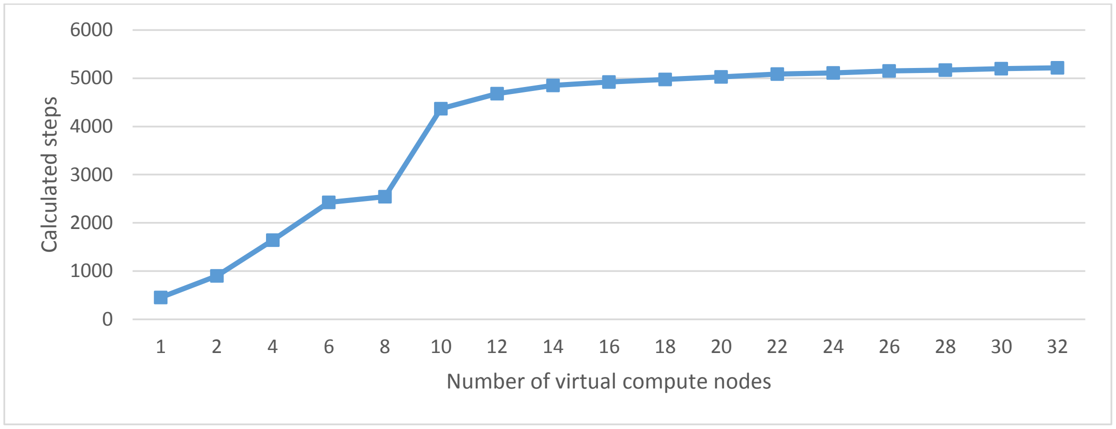


Figure 5.70: Simulated multi-node benchmark on single node (one CPU and one MIC) of Stampede. At each point, the load on the node was adjusted to be equal to the load on a node in a multi-node configuration. The benchmark employed different problem sizes ($3000 \times 30 \times 100$ beams) and the 2.9:1 MIC to CPU load ratio. The number of calculated radial steps was measured for about ~ 100 seconds.

other threads had to wait for them to complete their tasks for every iteration. As a result, there was no improvement in performance for increase number of nodes since the benchmark's result remained virtually identical that of the 6-node run. On the other hand, for the 10-node run there were 223 zenith beam allocated on the Xeon Phi. Nearly all of the available threads had the identical number of beams for calculation, so that there were no waiting time for majority of threads per loop iteration. Consequently, all threads completed their tasks simultaneously resulting in a significant performance gain in comparison to the 8-node run. As mentioned earlier, by going beyond 10 nodes, the number of available beams on Xeon Phi decreases and becomes too low, resulting in not utilizing the entire processor. Therefore, since the number of beams was less than the number of available threads on the Xeon Phi, the flat region for number of nodes greater than 10 was due to insufficient load on the co-processor. The insufficient load resulted in permanent idle time for many threads of the co-processor.

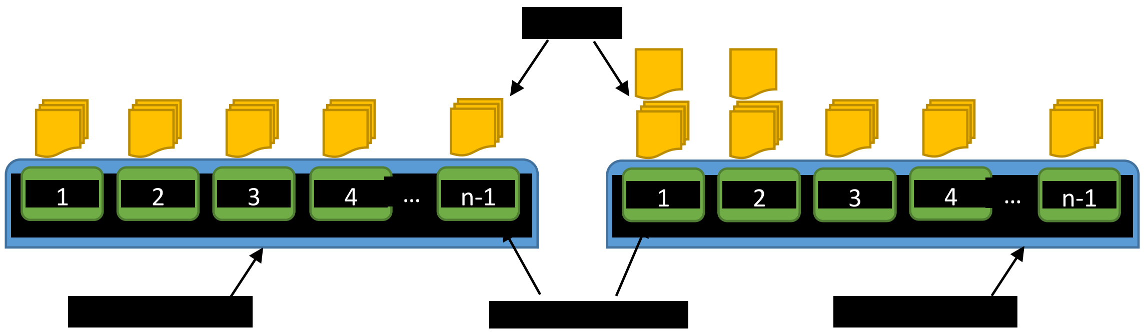


Figure 5.71: The effect of Xeon Phi thread load imbalance on overall performance. On the left side, the load on the Xeon Phi (shown in blue) is balanced, thus each core/thread (shown in green) receives equal number of tasks (shown in yellow). They complete their tasks virtually at the same time. On the right side, only a few threads (the first two threads) received more tasks, thus for every iteration the rest of the threads must remain idle and wait for the first two threads to complete their tasks.

The neutrino beams distribution issue can occur on both the CPU and the MIC. However, since there are large number of cores and threads on the MIC, the effect of load imbalance may affect a large number of threads. Therefore, even one thread can keep hundreds of threads in the waiting state, by forcing them to remain idle at every synchronization point. For instance, as depicted in Fig. 5.71, on the left side all of the hardware threads have an equal number of tasks, *i.e.* three tasks per thread. If each task takes t seconds to complete, the Xeon Phi completes all the tasks in $3 * t$ seconds. Alternatively, if only a few threads have one more task (the right side), the Xeon Phi completes all of its tasks in $4 * t$ seconds per iteration, no matter how many threads have incomplete tasks. Thus, the rest of the threads must remain idle and wait for the first two threads to complete their tasks.

The conclusion is that the reason for the observed flat regions in the results of the homogeneous runs on the MIC, as well as for the heterogeneous runs, is due to the load imbalance issue. This occurs when the load cannot be distributed equally across all threads. Hence, it is recommended that for a given XFLAT problem size, the number of nodes be chosen in the way that all of the threads on every MIC receive a nearly equal number of tasks.

5.3.7 Sweet spot location and MPI timings

The other performance aspect that was required to be analyzed was the location of the sweet spot. On heterogeneous environments, the location of the sweet spot (the optimum performance point) is directly related to the load ratio between processors. On both sides of the sweet spot, the timing increases for one processor and decreases for the other processor. The sweet spot is the location (load ratio) where the two sides change their trends, *i.e.*, the crossing point in the performance plots. For example, on the MIC side, after decreasing, the timing starts to increase, and on the CPU side, the timing starts to decrease after increasing. As illustrated in Fig. 5.66, Fig. 5.67, and Fig. 5.68, the location of the crossing points for the MIC and CPU is not fixed and varies with problem size.

It was expected that the location of the sweet spot, would be at the same point as the MPI timing crossing point (the point when the waiting time on both sides is minimized). However, this was not always the case. For example, the XFLAT overall run time for the $5550 \times 10 \times 100$ configuration and for various MIC:CPU load ratios is depicted in Fig. 5.72 and the MPI timing of each processor is depicted in Fig. 5.73. It anticipated that by changing the load ratio and placing less load on the MIC, the MPI timing on the MIC will decrease until it reaches the crossing point, timing will remain steady at its minimum. The behavior is expected since at the MPI timing crossing point, processes on both the CPU and MIC reach the synchronization point virtually at the same time. Therefore, after passing the sweet spot, due to load imbalance, one of the processes should always arrive at the synchronization point earlier than the other process. Consequently, the late process will not remain idle,

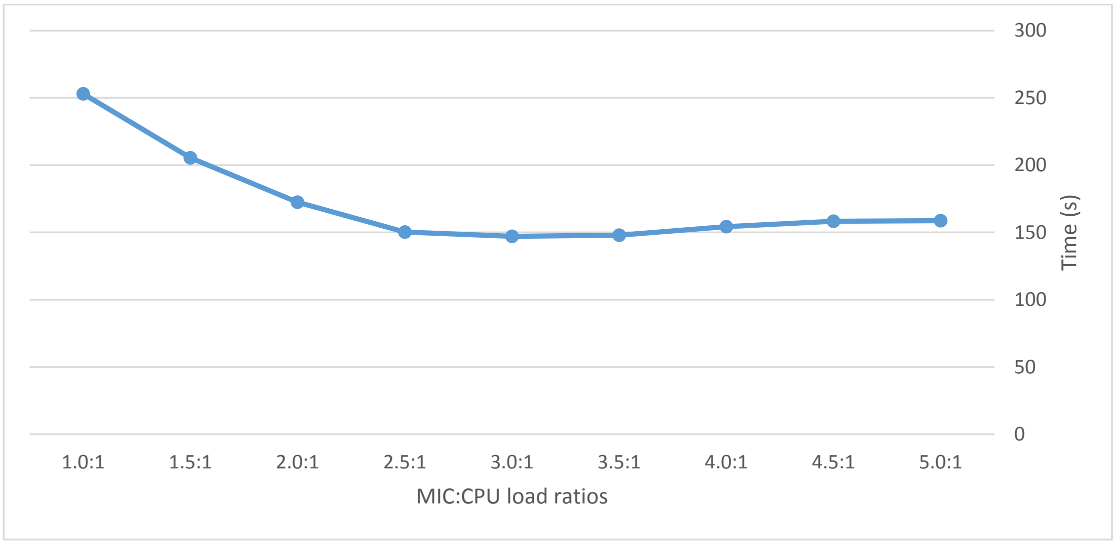


Figure 5.72: XFLAT overall run time on Stampede as the function of MIC to CPU load ratio. The problem size was set to 5550 × 10 × 100, the code was run for 1k radial steps, and run time was measured in seconds.

thus the MPI timing for the process is expected to remain minimized.

However, this expected behavior was not always observed (see Fig. 5.73). One of the reasons may be due to the fact that in XFLAT the outer evolution loop contains several communication points per iteration, and each communication point employs different MPI functions. Although the message size is fixed and not a function of the problem size, invoking different MPI synchronizations per iteration may affect the location of the sweet spot. Within the XFLAT evolution loop, there is one MPI broadcast, four MPI summation reductions, and one MPI maximization reduction per loop iteration. In order to study the effect of each MPI invocation on performance, the previous benchmark was repeated with fewer MPI calls (by eliminating some of the MPI invocations).

For the first benchmark, all MPI calls were eliminated except for the first MPI broadcast call. Thus, the only remaining communication point was the single broad-

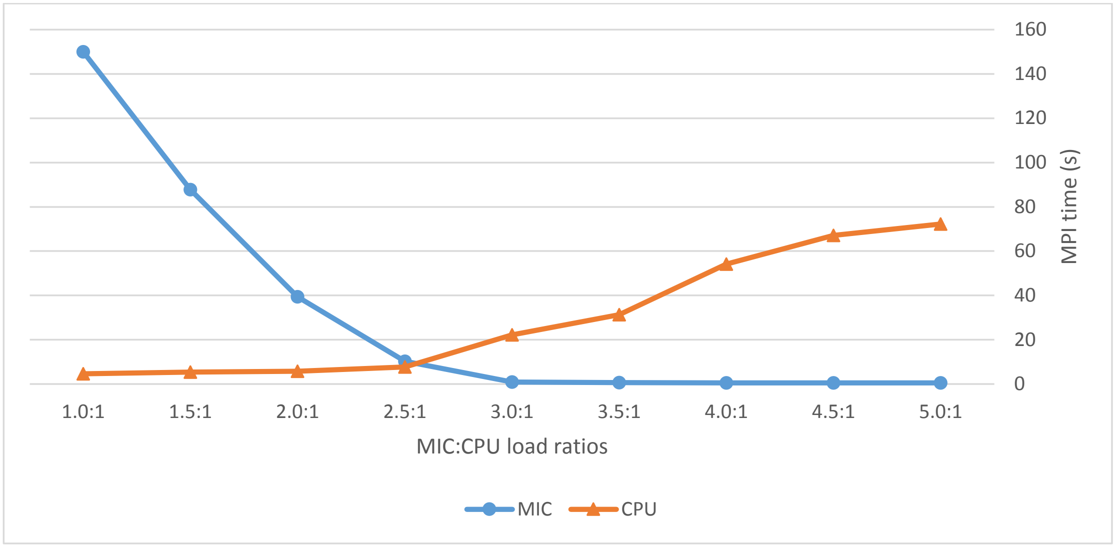


Figure 5.73: Measured MPI communications time for XFLAT on Stampede as the function of MIC to CPU load ratio. The problem size was set to 5550 × 10 × 100, the code was run for 1k radial steps, and the run time were measured in seconds.

cast from the root process to all processes. The results are shown in Table 5.20. The first row shows the MIC:CPU load ratios, the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. The location of the MPI crossing point was in the [2.5–3.0] load ratio range. Clearly, by increasing the load ratio, the MPI timing on the MIC reached its minimum and did not change afterwards, thus indicating that there was no idle time on the MIC side. On the other hand, by decreasing the load ratio, the MPI timing on the CPU was minimized, thus showing that the MIC had reached the communication point sooner and there was no idle time on the CPU side. In order to find the most precise sweet spot, the benchmark was repeated for the [2.6–3.4] range. As seen in Table 5.21, the sweet spot was located at the load ratio crossing point between 2.6:1 and 2.7:1.

In the next set of benchmarks, all MPI invocations were excluded except for the MPI maximum reduction function. The results are shown in Table 5.22, and

Table 5.20: XFLAT overall time for various MIC:CPU load ratios when a single MPI broadcast is presented per loop iteration. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. The highlighted cells show fixed time for each processor. All times in seconds.

MIC:CPU	1.0:1	1.5:1	2.0:1	2.5:1	3.0:1	3.5:1	4.0:1	4.5:1	5.0:1
MIC	147.6	93.3	35.5	4.3	0.1	0.1	0.1	0.1	0.1
CPU	0.02	0.02	0.02	0.02	19.9	32.6	49.3	63.4	67.5
Overall	250.3	201.1	167.4	143.4	145.7	144.4	149.7	154.8	153.2

Table 5.21: XFLAT overall time for various MIC:CPU load ratios (refined range) when a single MPI broadcast is presented per loop iteration. The first row shows the MIC:CPU load ratios, the second; third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. The highlighted cells show fixed time for each processor. All times in seconds.

MIC:CPU	2.6:1	2.7:1	2.8:1	2.9:1	3.0:1	3.1:1	3.2:1	3.3:1	3.4:1
MIC	0.4	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
CPU	0.02	4.1	4.8	11.1	22	23.5	23.6	27.8	31.4
Overall	140.7	140	141.4	140.1	147.5	148.5	143	145.3	145.6

in Table 5.23 for a more refined load ratio span. For the single MPI reduction, the sweet spot location was in the [2.6:1–2.7:1] range. Once again, after reaching the MPI timing crossing point, the timing remained constant on one side at its minimum, similar to the previous results.

For the next set of benchmarks, since there are four identical MPI reductions (using the sum operator) in the evolution loop, only those invocations were presented in the evolution loop. The results are shown in Table 5.24 and Table 5.25. As is apparent, unlike the previous results, the location of the MPI crossing point was shifted to the [2.7:1–2.8:1] range. In addition, the optimum point was shifted and this time located around the 2.9:1 point. The only differences between the previous benchmarks and this benchmark was that multiple identical MPI invocations per iteration were presented. Therefore, by presenting multiple synchronization points

Table 5.22: XFLAT overall time for various MIC:CPU load ratios when a single MPI reduction is presented per loop iteration. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. The highlighted cells show fixed time for each processor. All times in seconds.

MIC:CPU	1.0:1	1.5:1	2.0:1	2.5:1	3.0:1	3.5:1	4.0:1	4.5:1	5.0:1
MIC	153.3	88.9	37.9	5	0.1	0.1	0.1	0.1	0.1
CPU	0.25	0.32	0.37	0.4	19.8	35.7	51.4	66.5	68.3
Overall	252.2	206.2	168.9	144.1	146	147.7	153.1	158.7	152

Table 5.23: XFLAT overall time for various MIC:CPU load ratios (refined range) when a single MPI reduction is presented per loop iteration. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. The highlighted cells show fixed time for each processor. All times in seconds.

MIC:CPU	2.6:1	2.7:1	2.8:1	2.9:1	3.0:1	3.1:1	3.2:1	3.3:1	3.4:1
MIC	0.8	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15
CPU	2.7	5.5	12.4	11.4	16.8	23.5	26.1	29.5	33.8
Overall	143	142	145.2	142	146.6	147.1	146.5	146.7	148.2

(even identical) per iteration, the location of the MPI timing crossing point as well as the sweet spot may be changed.

To explain this, it is noted that although there were four identical MPI invocations per iteration, the code sections between two of the MPI invocations is not identical, thus the performance of each section on the MIC and CPU may vary. For instance, some of the code sections may be single-threaded, thus the CPU may perform faster in those regions, while in other sections the MIC may be faster, resulting in greater idle time for the MIC at the end that section. The combination of those different code sections and different MPI communication, may result in shifting the sweet spot as well as the location of the MPI crossing point.

In order to investigate the behavior of the code between the two MPI communica-

Table 5.24: XFLAT overall time for various MIC:CPU load ratios when four MPI reductions is presented per loop iteration. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. The highlighted cells show fixed time for each processor. All times in seconds.

MIC:CPU	1.0:1	1.5:1	2.0:1	2.5:1	3.0:1	3.5:1	4.0:1	4.5:1	5.0:1
MIC	146.8	93.8	36.6	5.4	0.5	0.5	0.5	0.5	0.5
CPU	0.47	0.48	0.5	2.6	17.9	32.3	52.2	67.8	73.1
Overall	250.9	210.8	168	146.8	143.3	147.9	152.6	159.4	156.3

Table 5.25: XFLAT overall time for various MIC:CPU load ratios (refined range) when four MPI reduction is presented per loop iteration. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. The highlighted cells show fixed time for each processor. All times in seconds.

MIC:CPU	2.6:1	2.7:1	2.8:1	2.9:1	3.0:1	3.1:1	3.2:1	3.3:1	3.4:1
MIC	4.8	2	0.51	0.51	0.51	0.51	0.51	0.51	0.51
CPU	2.4	7.92	13.7	14	26.1	27.2	26	30.1	29.1
Overall	143.8	143.7	145.8	141.4	151.9	150.1	146	147.3	149

tion points, the XFLAT main loop was partitioned such that each MPI function and its preceding code section were isolated and benchmarked separately (see Fig. 5.74). All of the following benchmarks were performed on one of the compute nodes of Stampede, while the total load set to 5550 × 10 × 100 beams. Each of the code sections was benchmarked separately for various MIC:CPU load ratios. In all of the following benchmarks, the code was run for 1k radial steps and the timing results were measured in seconds.

Since the first MPI communication point within the evolution loop is the MPI broadcast function, in the first benchmark the code prior to the broadcast was included in the loop as well as the last section of the code after the last MPI invocations within the loop (see Fig. 5.75). Hence, by altering the MIC:CPU load ratio, it was possible to study the behavior and the performance of this code section on each

```
while (conditions)
{
    ... //Code section before the MPI broadcast
    MPI_Bcast(...);

    ... //Code section before the first MPI reduction
    MPI_Allreduce(..., MPI_SUM, ...);

    ... //Code section before the second MPI reduction
    MPI_Allreduce(..., MPI_SUM, ...);

    ... //Code section before the third MPI reduction
    MPI_Allreduce(..., MPI_SUM, ...);

    ... //Code section before the fourth MPI reduction
    MPI_Allreduce(..., MPI_SUM, ...);

    ... //Code section before the last MPI reduction
    MPI_Allreduce(..., MPI_MAX, ...);

    ... //Rest of the code
}
```

Figure 5.74: Overall structure of XFLAT evolution loop and relative locations of MPI invocations.

processor.

The load ratio range for the MIC:CPU ratio was chosen to be similar to the ranges of the previous benchmarks. The benchmark results for the first isolated section is shown in Table 5.26, where it can be seen that the MPI time on the CPU side was always higher than on the MIC side. This implies that the CPU was faster in performing this section, thus the process on the MIC always arrived afterwards. Therefore, the load on the CPU for the range of the benchmarked load ratios was lower than the optimum load ratio.

```
while (conditions)
{
    ... // Code section before the MPI broadcast

    MPI_Bcast(...);

    /* The rest of the code is excluded from the benchmark */
    /* //Code section before the first MPI reduction       */
    /* MPI_Allreduce(..., MPI_SUM, ...);                    */
    /* //Code section before the second MPI reduction      */
    /* MPI_Allreduce(..., MPI_SUM, ...);                    */
    /* //Code section before the third MPI reduction       */
    /* MPI_Allreduce(..., MPI_SUM, ...);                    */
    /* //Code section before the fourth MPI reduction      */
    /* MPI_Allreduce(..., MPI_SUM, ...);                    */
    /* //Code section before the last MPI reduction        */
    /* MPI_Allreduce(..., MPI_MAX, ...);                    */

    ... // the last part after the last MPI
}
```

Figure 5.75: The evolution loop structure after isolating the sections associated with the first MPI invocation.

Similarly, the next benchmark was performed for the first MPI reduction invocation as well as the code section prior to the invocation. As shown in Table 5.27 and Fig. 5.76, there was no MPI timing crossing point for this range, since the CPU was faster in performing this section. Thus, for this MPI invocation and its associated code section, the MIC always arrived at the communication point after the CPU.

The following results are related to the four similar MPI reduction invocations with their associated code sections. Each section was benchmarked separately in order to study the performance and behavior of each section of code. The results of the second, third, and fourth MPI reduction invocations (and their associated code) are shown in Table 5.28, Table 5.29, and Table 5.30 as well as Fig. 5.77,

Table 5.26: XFLAT timings when the first section of the code including the MPI broadcast was benchmarked. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. All times in seconds.

MIC:CPU	2.2:1	2.3:1	2.4:1	2.5:1	2.6:1	2.7:1	2.8:1	2.9:1	3.0:1
MIC	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
CPU	0.33	0.33	0.33	0.31	0.33	0.33	0.32	0.33	0.32
Overall	0.33	0.33	0.33	0.32	0.34	0.34	0.33	0.34	0.33

Table 5.27: XFLAT timings when the second section of the code including the first MPI reduction was benchmarked. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. All times in seconds.

MIC:CPU	2.1:1	2.2:1	2.3:1	2.4:1	2.5:1	2.6:1	2.7:1	2.8:1	2.9:1
MIC	0.12	0.12	0.12	0.12	0.12	0.12	0.12	0.12	0.12
CPU	5.5	5.5	5.6	5.6	5.7	5.7	5.6	5.4	4.8
Overall	5.7	5.7	5.8	5.8	5.9	5.9	5.8	5.6	4.9

Fig. 5.78, and Fig. 5.79, respectively. As is evident, the crossing point location changed between the benchmarks. In addition, after the crossing point, the timing on one side remained at its minimum. The observed difference among the results indicates that the performance of the code on the MIC and CPU varies between code sections. Although the communications are similar for each MPI invocation, by varying the load ratio, the optimum point for each code section is different. Thus, the performance of the code varies for each section.

Finally, the performance of the last MPI reduction invocation and its associated code was benchmarked. Unlike the previous MPI reductions, the last reduction invocation (using the max operator) calculates the maximum of error values across nodes. The result of this section is shown in Table 5.31 and Fig. 5.80. It can be seen that similar to the previous reduction invocation, prior to reaching the crossing point the MPI time for the MIC was steady, and after passing the crossing point, on

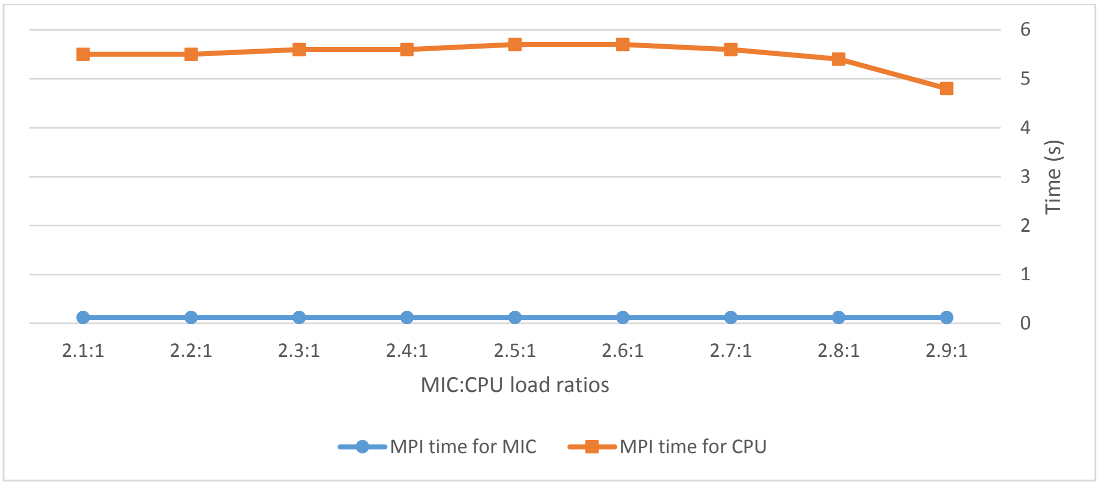


Figure 5.76: XFLAT timing behavior when the second section of the code including the first MPI reduction was benchmarked. The blue curve shows the measured time of MPI invocation on the MIC side, and the orange curve shows the measured time of MPI invocation on the CPU side.

the right side of the point, the MPI time for CPU was stationary. In addition, the location of the crossing point was shifted toward higher load ratios, which indicates that the performance of this section on the MIC and CPU was not identical to that of the previous reduction sections.

Since there are four MPI summation reduction invocations with the same message size within the evolution loop, and since the majority of computations are located within this section, the entire code block containing those invocations was combined together and benchmarked as well. As shown in Fig. 5.81, unlike the previous result, after passing the MPI timing crossing point, the MPI time still decreased on both sides. Therefore, the asymmetry and differences in the instructions of each code section may shift the location of the sweet spot as well as the MPI timing crossing point.

It can be concluded that the location of the sweet spot, which shows the optimum

Table 5.28: XFLAT timings when the third section of the code including the second MPI reduction was benchmarked. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. All times in seconds.

MIC:CPU	2.1:1	2.2:1	2.3:1	2.4:1	2.5:1	2.6:1	2.7:1	2.8:1	2.9:1
MIC	7	8.3	4.8	3.3	1.3	1.2	.12	.12	.12
CPU	.3	.3	.3	.3	.3	.3	5.1	3.22	6.7
Overall	53.7	51.8	50.4	48.7	47.7	47.3	49.8	46.8	51.8

point of the performance, can be located at a different point than the MPI timing crossing point. The reason for this behavior is due to the fact that the performance of the evolution loop is the function of the performance of sections that are separated by MPI synchronization points. On the MIC, the performance of some sections may be higher than that of the CPU; on the other hand, the performance of the other sections may be higher on the CPU than the MIC. Therefore, the combination of

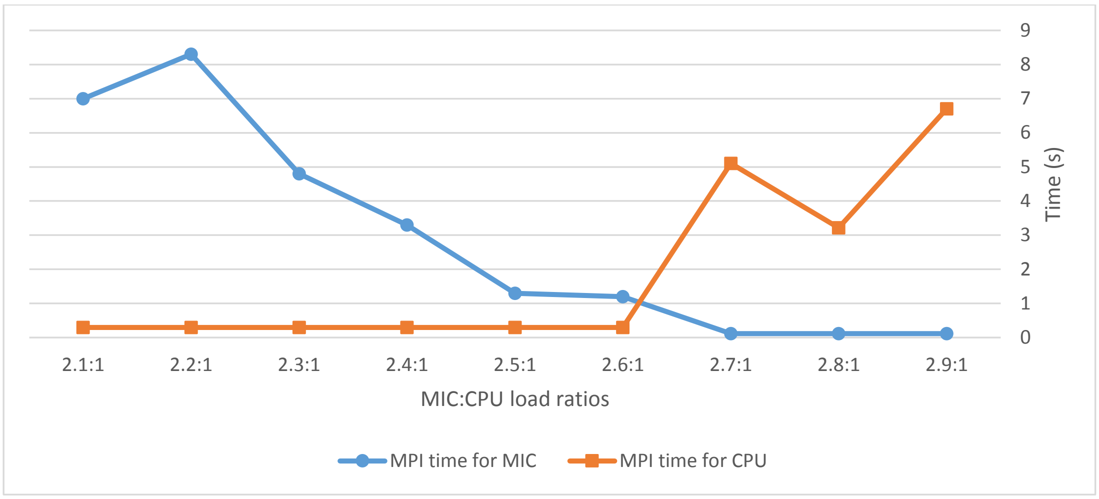


Figure 5.77: XFLAT timing behavior when the third section of the code including the second MPI reduction was benchmarked. The blue curve shows the measured time of MPI invocation on the MIC side, and the orange curve shows the measured time of MPI invocation on the CPU side.

Table 5.29: XFLAT timings when the fourth section of the code including the third MPI reduction was benchmarked. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. All times in seconds.

MIC:CPU	2.1:1	2.2:1	2.3:1	2.4:1	2.5:1	2.6:1	2.7:1	2.8:1	2.9:1
MIC	8.3	6.8	5.9	2.1	2.4	1.5	.4	.12	.12
CPU	.3	.3	.3	.3	.3	.3	.4	2.2	2.1
Overall	35.9	34.5	34.1	32.6	31.8	31.0	30.1	31.0	30.2

those sections on both sides can significantly affect the final position of the sweet spots as well as the MPI timing crossing point. This explains of the difference between the location of the MPI timing crossing point and the overall optimum point. In addition, it explains why the MPI timing did not reach its minimum immediately after passing the crossing point on the MIC or CPU side.

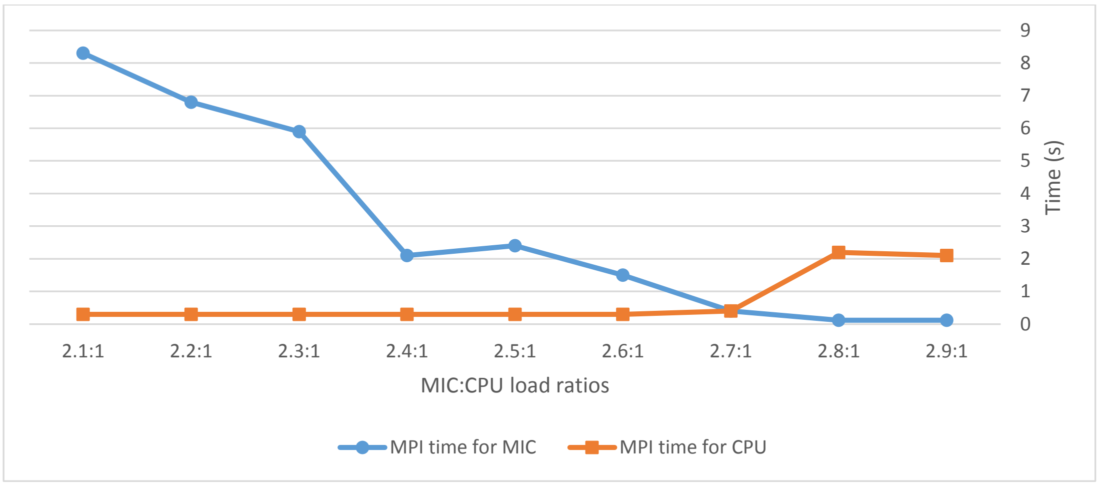


Figure 5.78: XFLAT timing behavior when the fourth section of the code including the third MPI reduction was benchmarked. The blue curve shows the measured time of MPI invocation on the MIC side, and the orange curve shows the measured time of MPI invocation on the CPU side.

Table 5.30: XFLAT timings when the fifth section of the code including the fourth MPI reduction was benchmarked. The first row shows the MIC:CPU load ratios; the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively. All times in seconds.

MIC:CPU	2.1:1	2.2:1	2.3:1	2.4:1	2.5:1	2.6:1	2.7:1	2.8:1	2.9:1
MIC	2.	.13	.5	.12	.12	.12	.12	.12	.12
CPU	.3	.9	.4	2.6	2.7	4.0	4.1	5.2	5.7
Overall	26.9	26.2	25.1	26.5	26.1	26.6	26.3	26.6	26.6

5.3.8 Merging MPI functions

From the previous results, it can be concluded that if the number of MPI synchronization points within the evolution loop can be reduced, there may be improvement in the overall performance of the loop. In order to minimize the number of communication points as well as the waiting time between compute nodes for each processor, the

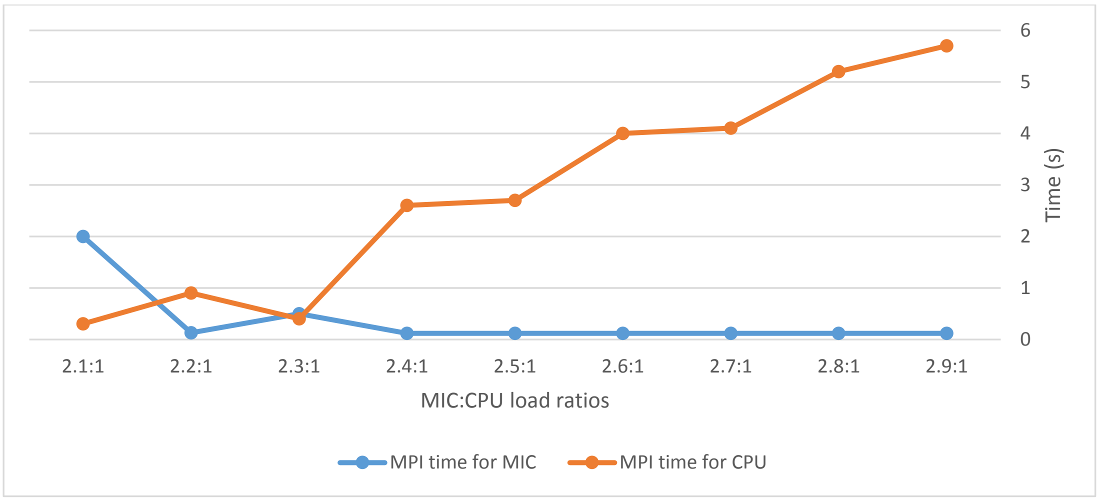


Figure 5.79: XFLAT timing behavior when the fifth section of the code including the fourth MPI reduction was benchmarked. The blue curve shows the measured time of MPI invocation on the MIC side, and the orange curve shows the measured time of MPI invocation on the CPU side.

Table 5.31: XFLAT timing when the last section of the code including the last MPI reduction was benchmarked. The first row shows the MIC:CPU load ratios, the second, third, and fourth rows show the MIC time, the CPU time, and the overall time, respectively.

MIC:CPU	2.4:1	2.5:1	2.6:1	2.7:1	2.8:1	2.9:1	3.0:1	3.1:1	3.2:1
MIC	8.61	8.11	6.22	4.54	3.15	2.59	.95	.15	.15
CPU	0.3	0.3	0.3	0.3	0.3	0.3	0.3	.64	1.67
Overall	39.5	39.1	37.4	36.2	35.1	34.4	33.6	33.0	33.2

main evolution loop can be reordered. By reducing the waiting time at synchronization points, the processing time and the efficiency may be improved. One approach to achieving this goal is to reduce the number of MPI synchronization points by merging two of the similar `MPI_Allreduce()` invocations together. By merging two MPI invocations together, instead of having six MPI invocations per loop iteration (one `MPI_Bcast()`, four `MPI_Allreduce(SUM)`, and one `MPI_Allreduce(MAX)`) there

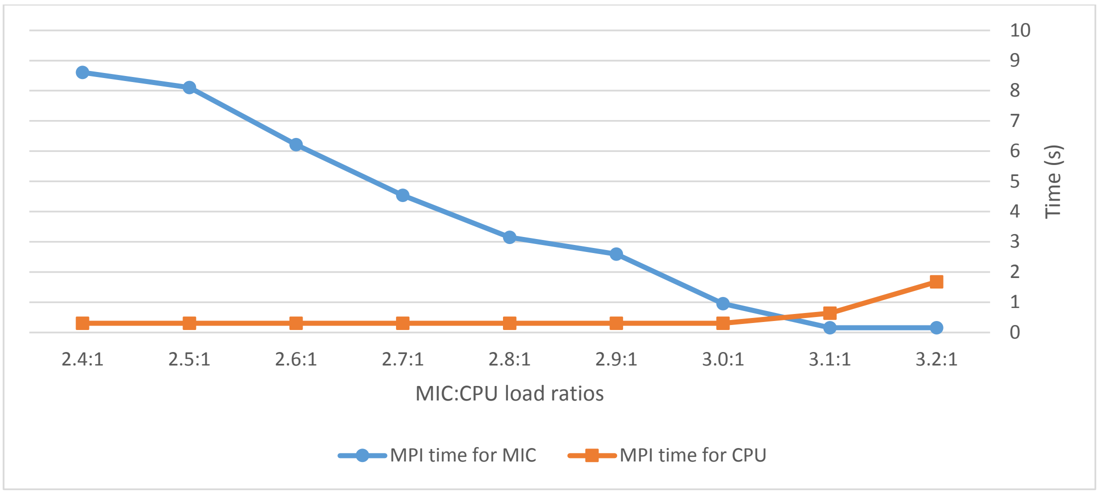


Figure 5.80: XFLAT timing behavior when the last section of the code including the last MPI reduction was benchmarked. The blue curve shows the measured time of MPI invocations on the MIC side, and the orange curve shows the measured time of MPI invocations on the CPU side.

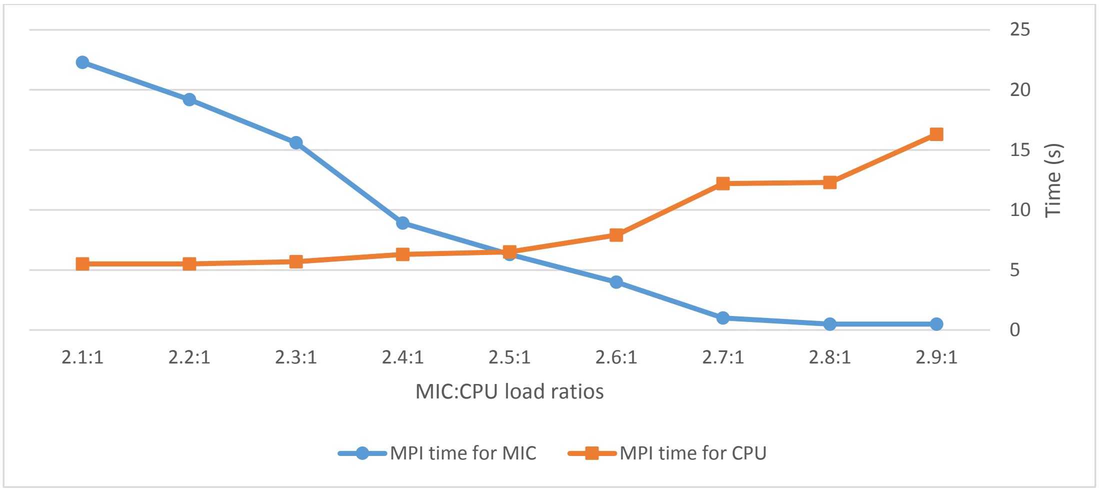


Figure 5.81: Performance for the combined code section of the four MPI summation reductions for various MIC:CPU load ratios.

can be five MPI invocations per iteration. Nevertheless, not every MPI reduction can be merged together, since the code section before each one invocations may depend on the result of the previous reduction invocations. The original neutrino evolution loop arrangement is shown in Listing 5.1.

Listing 5.1: Illustration of the first version of the evolution's loop algorithm

```
while (termination_conditions)
{
    // setting some flags
    // ...
    // broadcasting the termination conditions from the root
    MPI_Bcast(...);

    //========================== S1 ============================
    // calculating angle bins
    // derivative of length for each angle bins, etc.
    // ...
    // calculating the Hamiltonian H0 and
    // exchanging the partial result of H0 between nodes
    MPI_Allreduce(..., H0, MPI_SUM, ...);
    //------------------------- End of S1 ---------------------------

    //========================= S2 ==============================
    // evolving neutrino beams from the current radial point
    // to the middle point (half step size) using the H0
    // ...
    // evolving neutrino beams from the current radial point
    // to the next point (full step) using the calculated H0
    // ...
    // calculating the Hamiltonian H1 and
    // exchanging the partial result of H1 between nodes
    MPI_Allreduce(..., H1, MPI_SUM, ...);
    //------------------------- End of S2 ---------------------------

    //========================= S3 ==============================
    // evolving neutrino beams from the current radial point
    // to the next point (full step size) using the H1
    // ...
    // calculating the average of the two previously evolved
    // neutrino beams
    // ...
    // calculating the Hamiltonian H2 and
    // exchanging the partial result of H2 between nodes
    MPI_Allreduce(..., H2, MPI_SUM, ...);
```

```
    //———————————————————— End of S3 ————————————————————

    //==================== S4 ============================
    // evolving neutrino beams from the middle point
    // to the next point (half step size) using the H2
    // ...
    // calculating the Hamiltonian H3
    // exchanging the partial result of H3 between nodes
    MPI_Allreduce(..., H3, MPI_SUM, ...);
    //———————————————————— End of S4 ————————————————————

    //==================== S5 ============================
    // evolving neutrino beams from the current radial point
    // to the next point (full step size) using the H3
    // ...
    // calculating the average of the two previously evolved
    // neutrino beams
    // ...
    // calculating the maximum error as well as finding
    // the global maximum error among all the beams
    MPI_Allreduce(..., MPI_MAX, ...);
    //———————————————————— End of S5 ————————————————————

    // checking for any necessary I/O
    // ...
    // adjusting the radius for the next iteration
    // ...
}
```

It can be observed that the third MPI reduction section does not depend on the code between the second and third MPI reductions; thus the computation part for the Hamiltonian $H2$ can be moved to the line after the computation of the Hamiltonian $H1$. Therefore, the neutrino evolution loop was reordered in a new way as shown in Listing 5.2.

Listing 5.2: Illustration of the second version of the evolution's loop algorithm

```
while (termination_conditions)
{
    // setting some flags
    // ...
    // broadcasting the termination conditions from the root
    MPI_Bcast(...);

    //======================== S1 ==========================
    // calculating angle bins
    // derivative of length for each angle bins, etc.
    // ...
    // calculating the Hamiltonian H0 and
    // exchanging the partial result of H0 between nodes
    MPI_Allreduce(..., H0, MPI_SUM, ...);
    //-------------------- End of S1 ----------------------

    //======================== S2 ==========================
    // evolving neutrino beams from the current radial point
    // to the middle point (half step size) using the H0
    // ...
    // evolving neutrino beams from the current radial point
    // to the next point (full step) using the H0
    // ...
    // calculating the Hamiltonian H1 and
    // exchanging the partial result of H1 between nodes
    MPI_Allreduce(..., H1, MPI_SUM, ...);
    // calculating the Hamiltonian H2 and
    // exchanging the partial result of H2 between nodes
    MPI_Allreduce(..., H2, MPI_SUM, ...);
    //------------------------- End of S2 ---------------------------

    //============================== S3 ===============================
    // evolving neutrino beams from the current radial point
    // to the next point (full step size) using the H1
    // ...
    // calculating the average of the two previously evolved
    // neutrino beams
    // ...
    // evolving neutrino beams from the middle point
    // to the next point (half step size) using the H2
    // ...
    // calculating the Hamiltonian H3
    // exchanging the partial result of H3 between nodes
    MPI_Allreduce(..., H3, MPI_SUM, ...);
    //------------------------- End of S3 ---------------------------

    //============================== S4 ===============================
    // evolving neutrino beams from the current radial point
    // to the next point (full step size) using the H3
    // ...
```

```
    // calculating the average of the two previously evolved
    // neutrino beams
    // ...
    // calculating the maximum error as well as finding
    // the global maximum error among all the beams
    MPI_Allreduce(..., MPI_MAX, ...);
    //------------------- End of S4 -------------------------

    // checking for any necessary I/O
    // ...
    // adjusting the radius for the next iteration
    // ...
}
```

The last step to restructuring the evolution algorithm was to merge the procedures that were responsible for computing the Hamiltonian $H1$ and the Hamiltonian $H2$ as a single fused method. The two MPI reductions were fused together as a single MPI function invocation. Although the message size in the fused reduction invocation was twice as long as the previous invocation (since the partial results for the Hamiltonian $H1$ and the Hamiltonian $H2$ were exchanged in one message), the number of data exchanging points in the loop was one less than in the previous algorithm. Hence, the performance of the code was expected to improve in the heterogeneous environment, since the wasted time at the MPI exchange point, which may be due to the imperfect load balancing, was reduced. The new, restructured evolution loop is shown in Listing 5.3.

Listing 5.3: Illustration of the third version of the evolution's loop algorithm

```
while (termination_conditions)
{
```

```
// setting some flags
// ...
// broadcasting the termination conditions from the root
MPI_Bcast(...);

//=========================== S1 ==============================
// calculating angle bins
// derivative of length for each angle bins, etc.
// ...
// calculating the Hamiltonian H0 and
// exchanging the partial result of H0 between nodes
MPI_Allreduce(..., H0, MPI_SUM, ...);
//------------------------- End of S1 --------------------------

//=========================== S2 ==============================
// evolving neutrino beams from the current radial point
// to the middle point (half step size) using the H0
// ...
// evolving neutrino beams from the current radial point
// to the next point (full step) using the calculated H0
// ...
// calculating the Hamiltonian H1 and H2 and
// exchanging the partial result of H1 and H2 together
// between nodes
MPI_Allreduce(..., H1_H2, MPI_SUM, ...);
//------------------------- End of S2 --------------------------

//=========================== S3 ==============================
// evolving neutrino beams from the current radial point
// to the next point (full step size) using the H1
// ...
// calculating the average of the two previously evolved
// neutrino beams
// ...
// evolving neutrino beams from the middle point
// to the next point (half step size) using the H2
// ...
// calculating the Hamiltonian H3
// exchanging the partial result of H3 between nodes
MPI_Allreduce(..., H3, MPI_SUM, ...);
//------------------------ End of S3 ------------------------

//========================== S4 ===========================
// evolving neutrino beams from the current radial point
// to the next point (full step size) using the H3
// ...
// calculating the average of the two previously evolved
// neutrino beams
// ...
// calculating the maximum error as well as finding
// the global maximum error among all the beams
MPI_Allreduce(..., MPI_MAX, ...);
//------------------------ End of S4 ------------------------

// checking for any necessary I/O
// ...
```

```
    // adjusting the radius for the next iteration
    // ...
}
```

Several multi-node benchmarks were required in order to study and analyze the performance of the code in the old and the new algorithms. In all of the following benchmarks on Stampede, the MIC:CPU load ratio was set at 3:1, thus the load on the MIC was fixed at $600 \times 10 \times 100$ (θ *angles* $\times$ ϕ *angles* $\times$ *Energy bins*) and the load on the CPU was fixed at $200 \times 10 \times 100$ (θ *angles* $\times$ ϕ *angles* $\times$ *Energy bins*). The benchmarks were repeated for various number of nodes and continued for 10k radial steps. The maximum MPI time of MICs, maximum MPI time of CPUs and the overall timing were measured.

The first benchmark employed one CPU and one MIC per compute node, and the second benchmark utilized two CPUs and two MICs per compute node. Each set of benchmarks was repeated on 16 and 32 nodes. As a result, the minimum number of MPI tasks was 32 (16 tasks on 16 CPUs and 16 tasks on 16 MICs) and the maximum number of MPI tasks was 128 MPI (64 tasks on 32 dual-CPU and 64 tasks on 32 dual-MIC nodes).

In Fig. 5.82 (top), the performance of the old style algorithm (version 1) is depicted. As is visible, by increasing the number of compute nodes the overall timing increased slightly. This was inter-node communication overhead, since by increasing the number of nodes, the amount of inter-node messages increases.

The timing of the reordered algorithm (version 2) was virtually identical to that of the version 1 algorithm, as shown in Fig. 5.82 (middle). Therefore, the reordering of the MPI functions did not boost the performance of the evolution algorithm.

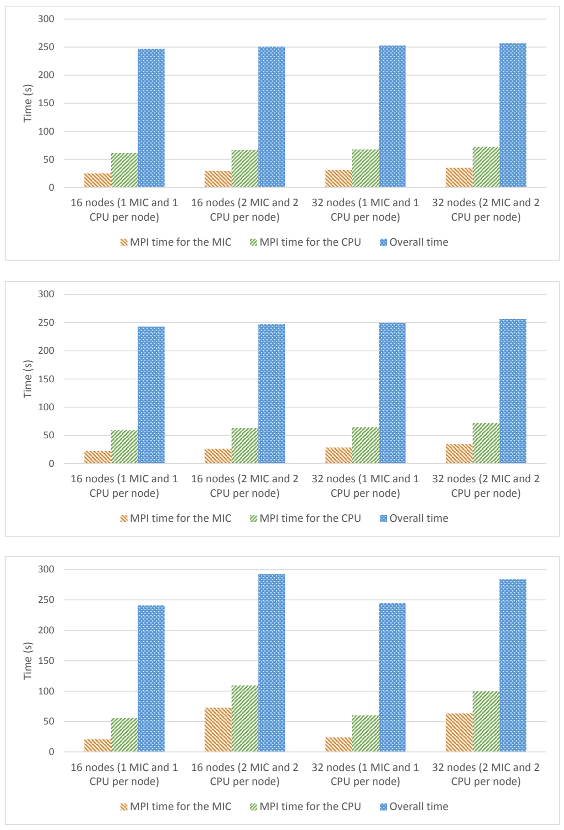


Figure 5.82: Performance of the old style algorithm (top panel), the reordered algorithm (middle panel), and the new style algorithm (bottom panel). There were six MPI function calls per loop iteration. The orange bar shows the maximum MPI time among the MICs' processes, the green bar shows the maximum MPI time among the CPUs' processes, and the blue bar shows the overall run time in seconds.

Surprisingly, as shown in Fig. 5.82 (bottom), when two of the MPI reductions were fused together in the new version of the algorithm (version 3), the overall timing jumped significantly only for the dual processor per node benchmarks. The issue with the performance may be related to the restructuring of the code. Moreover, it may be due to the message size in the version 3 algorithm for which the MPI message size was doubled. Thus, MPI communications might be slower due to the lengthier messages. In order to examine and understand whether or not the behavior was related to the loop structure, the neutrino evolution loop was decomposed into different sections

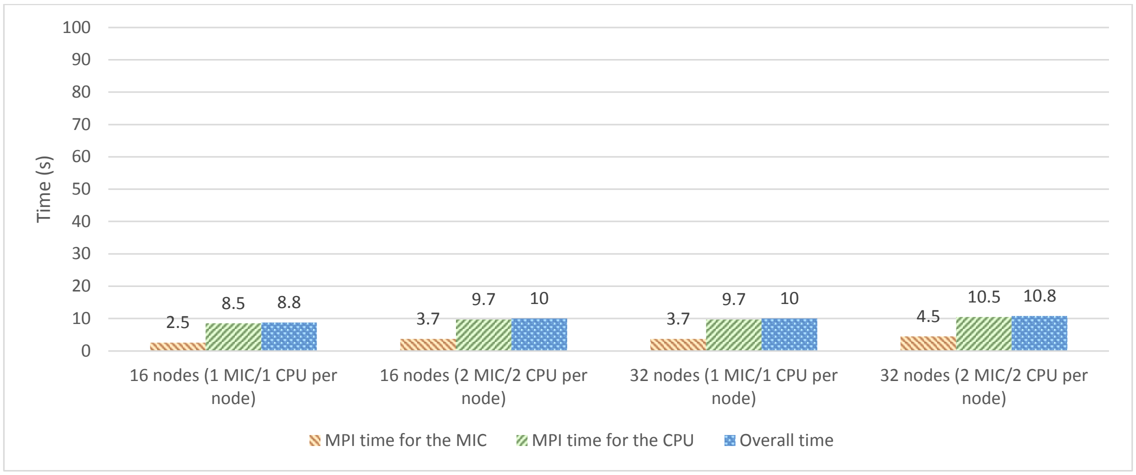


Figure 5.83: Performance of section S1 of all algorithms. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

and each one was benchmarked separately. Hence, any of the differences between benchmarks could be traced to exact line of the code.

The neutrino evolution loop is decomposed into multiple sections by MPI synchronization points, similar to previous benchmarks. Each section ending with a synchronization point was isolated and benchmarked separately. The following plots depict the performance of the each isolated section of the neutrino evolution loop.

Since each section ended with a synchronization point via MPI, only the timing of the codes associated before each MPI invocation was measured. Fig. 5.83 illustrates the performance of section S1, which was identical for all of the three versions of the evolution algorithm. The amount of computation for this section was low, and there was no significant difference between the results.

The benchmarks for the next sections were performed in a similar manner to the first section. Fig. 5.84 shows the performance of the S2 section for the algorithms

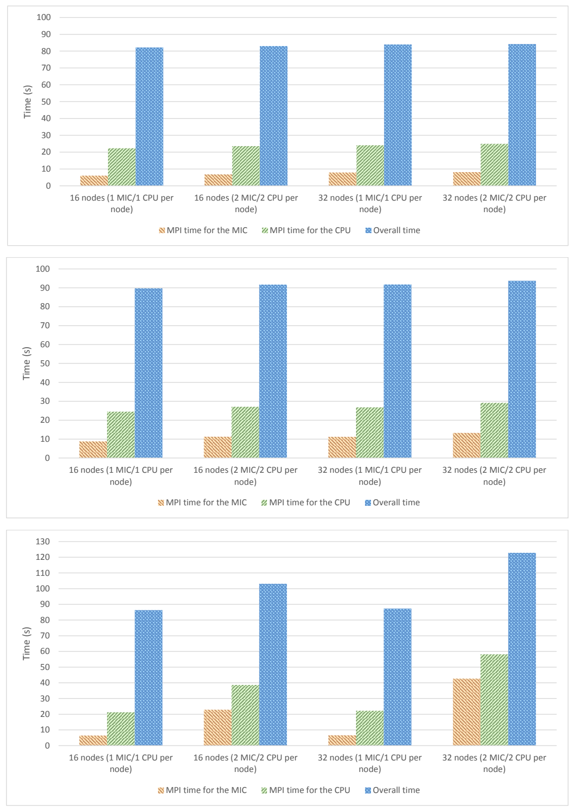


Figure 5.84: Performance of section S2 of algorithms version 1 (top), version 2 (middle), and version 3 (bottom). The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processe; and the blue bar shows the overall run time in seconds.

version 1 (top), 2 (middle), and 3 (bottom), respectively.

As illustrated in Fig. 5.84 (bottom panel), the MPI times of the dual-MIC benchmarks jumped by a significant amount resulting in an increase in the overall run time. This section may be the root cause of the performance difference; however, the rest of the code sections had to be benchmarked as well in order to verify that the issue is only due to this section of the loop. Continuing the benchmarks for the next sections, as depicted in Fig. 5.85 and Fig. 5.86, the performance of the S2 and S3 sections for the old algorithm (version 1) are shown. No significant performance differences were

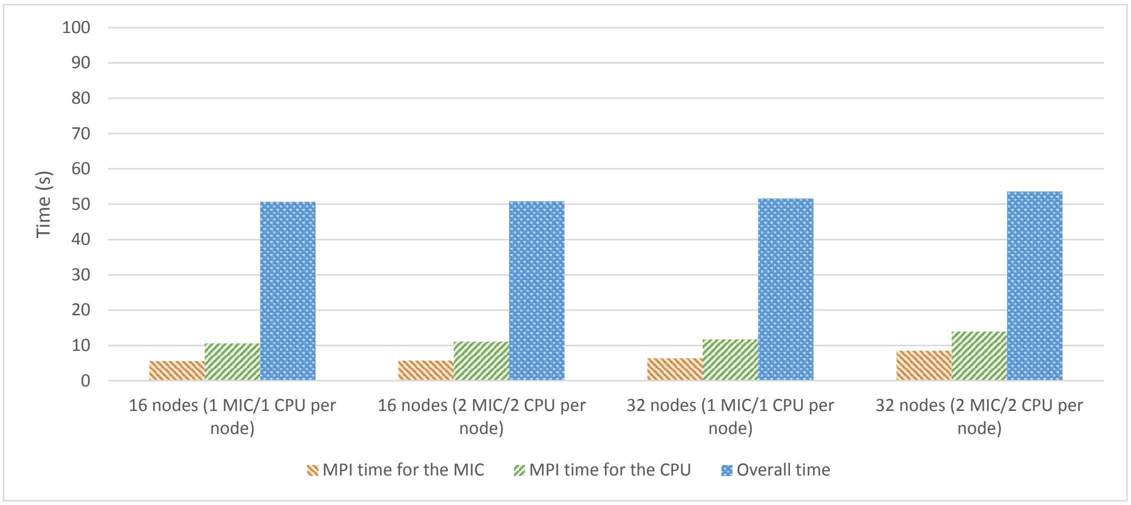


Figure 5.85: Performance of section S3 of algorithms version 1. The red orange shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

observed for these sections.

Algorithms version 2 and 3 had one more code section that needed to be benchmarked. The performance of the code section before the last MPI reduction (S3) the algorithms version 2 and version 3 is illustrated in Fig. 5.87. Since the code inside this section was identical between the two algorithms, the results were equivalent for both approaches. No significant performance dissimilarity was observed for the last code section.

Finally, the performance of section S4 is depicted by Fig. 5.88. All of the algorithms shared the same code for this section, thus the performances were identical for them. Once more, no significant differences between the results were observed.

In the previous plots, the performance gap only appeared when the section in which the fused MPI calls was exploited, *i.e.*, algorithm version 3 (see Fig. 5.84). In addition, it was only observed for the benchmarks that utilized two co-processors

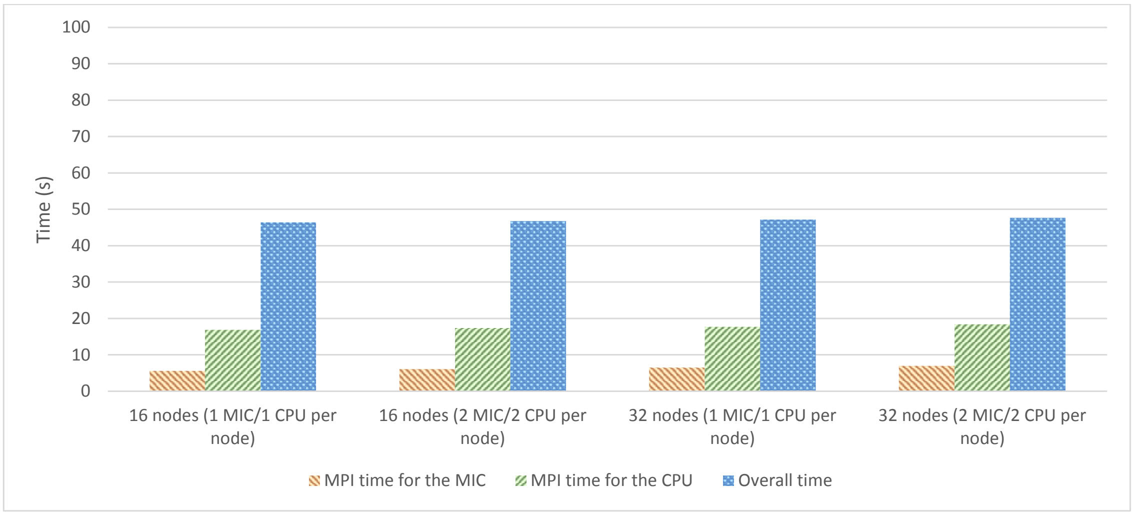


Figure 5.86: Performance of section S4 of algorithm version 1. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

per node. In that section of the code the only major difference between the three algorithms was the MPI message size for which the message size for the algorithm version 3 was doubled in size.

Since the performance hit only occurred when the size of the MPI messages increased, there may be an issue related to the message size. In order to further investigate this issue, the MPI message size was artificially doubled and quadrupled for both algorithm version 1 and version 3 (the performance of versions 1 and 2 was identical), using dummy arrays. The MPI messages were padded with data on which the calculations did not depend, however, this increased the overall MPI message size. Since the only difference between the algorithms at the communication point was the inter-node message size, the results of the new benchmarks were expected to differ when the message size was changed. The benchmarks were repeated on 2, 4, 8, 16, and 32 compute nodes. As shown in Fig. 5.89, by doubling the message size, the

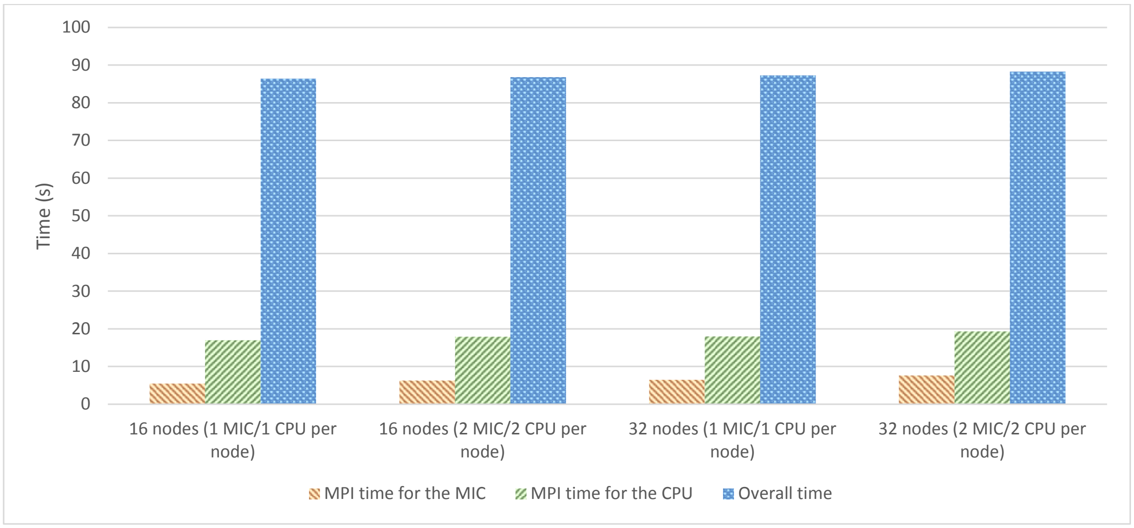


Figure 5.87: Performance of section S3 of algorithms version 2 and version 3. The orange bar shows the maximum MPI time among the MICs' processe; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

performance of the algorithm version 3 remained steady, with no observable drop in performance. Quadrupling the message size did not change the performance trend either (Fig. 5.90).

Similar benchmarks were repeated for algorithm version 1, first by doubling the MPI message size and next by quadrupling the message size. The performance of the benchmarks using the doubled message size is illustrated in Fig. 5.91. By increasing the number of nodes, the overall timing started to surge by at least 50%, similar to the benchmark of algorithm version 3. The communication MPI message size was identical to that of the algorithm version 3 (its regular message size), for which the same behavior were observed. The results of the benchmarks using the quadrupled message size are depicted in Fig. 5.92. Quadrupling the message size caused the overall timing to be restored to its expected normal behavior. Hence, the issue only happened with a particular MPI message size for both algorithms.

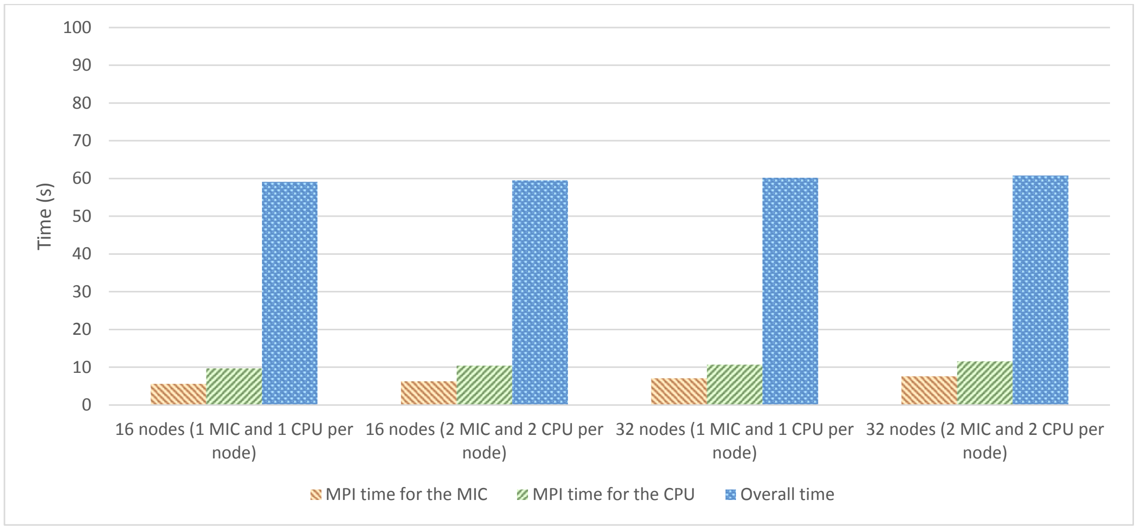


Figure 5.88: Performance of the last section before the last reduction for all algorithms. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

At this point it can be concluded that the poor performance of the code on the dual-MIC nodes only occurred for a specific MPI message size, and it was not a function of the utilized instructions before the MPI invocations, nor the number of employed compute nodes. Furthermore, from the previous plots it is clear that the issue was not due to the code structure or a bug in XFLAT. Since the issue only occurred for a specific MPI message size and the behavior returned to normal when the message size changed, it may be due to the other factors.

One possibility is that it may be due to a bug in the Intel C++ compiler or Intel MPI library. For all previous benchmarks the Intel C++ compiler's version was 13.1.1.163 and the Intel MPI library version was 4.1.0.030. After upgrading the toolkit to packages toward the end of this work (compiler version 14.0.1 and MPI library version 4.1.3.049), the benchmarks were repeated by utilizing similar loads for algorithm version 1 and 3 (the load per each CPU and MIC was fixed). The

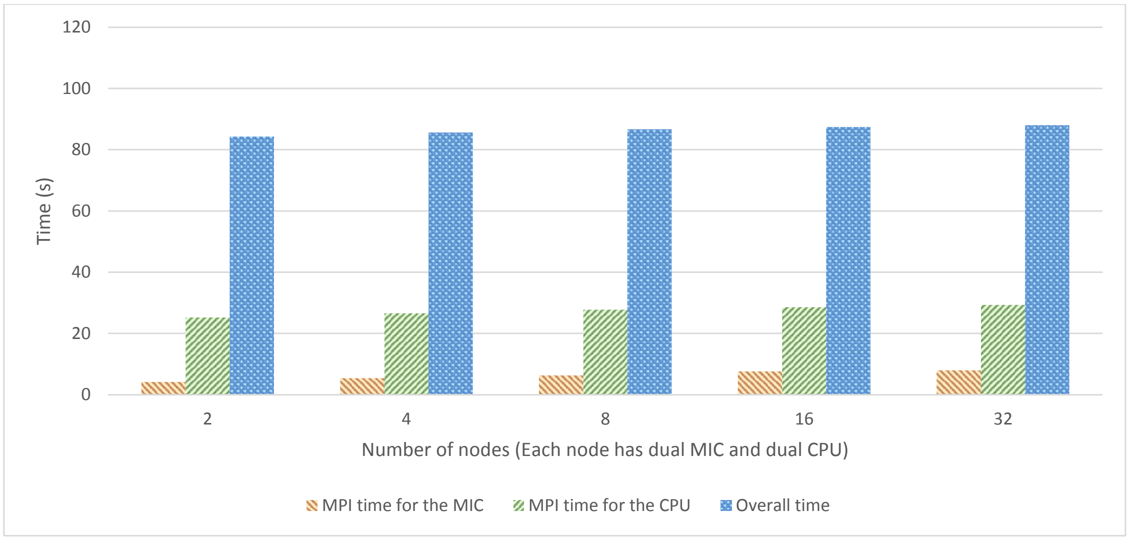


Figure 5.89: Performance of section S2 of algorithm version 3 using doubled MPI message size. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

number of nodes was varied from 16 to 128 compute nodes. The performance of algorithm version 1 and version 3 (one CPU and one MIC per node) is depicted in Fig. 5.93 and Fig. 5.94, respectively. Moreover, Fig. 5.95 and Fig. 5.96 illustrate the performance of algorithm version 1 and version 3 when two CPUs and two MICs were employed per node, respectively.

As is apparent from the results, there was no unexpected jump in MPI timing and the behavior of the single-processor and dual-processor benchmarks were similar. Consequently, the issue cleared up completely after upgrading the libraries. It turned out that the issue was related to a bug in the Intel MPI library 4.1.0.030; thus, by upgrading the library to version 4.1.3.049 the issue was resolved.

From the previous results, it can be concluded the XFLAT performance scales satisfactorily with number of nodes. Even when employing 512 total tasks (128 compute nodes, each equipped with two CPUs and two MICs, thus four tasks per

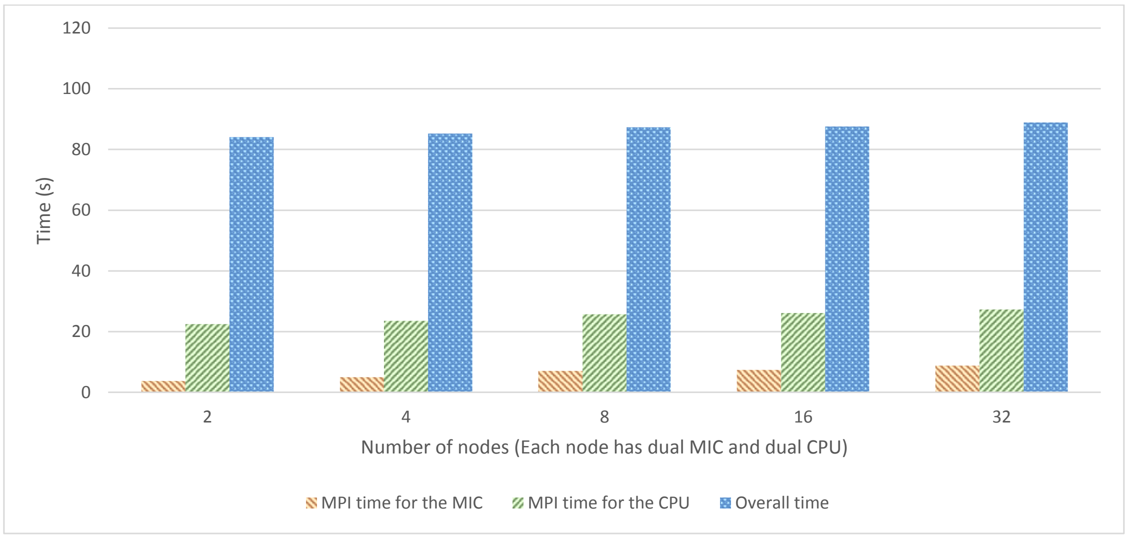


Figure 5.90: Performance of section S2 of algorithm version 3 using quadrupled MPI message size. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

node), XFLAT performance did degrade significantly. In addition, merging two MPI messages together improved overall performance by a few percent. Therefore, XFLAT is not limited by inter-node communication nor by the number of employed nodes and can scales well with hundreds of tasks.

5.3.9 XFLAT auto-benchmarking code

Based the previous benchmarks, is is clear that the prediction of the sweet spot for XFLAT on heterogeneous multi-node environments is not straightforward. On a heterogeneous environment, different processor types with different clock frequencies and multiple inter-node and intra-node buses each with different speeds and latencies, can make the prediction of the location of the sweet spot complex. In addition, different memory hierarchies with different architectures and speeds, utilizing differ-

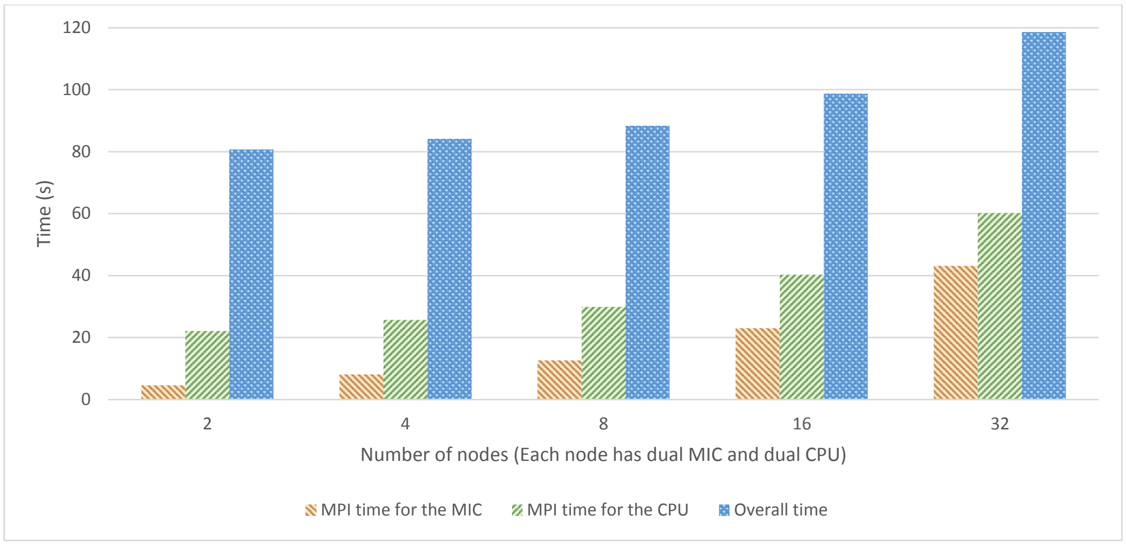


Figure 5.91: Performance of section S2 of algorithm version 1 using doubled MPI message size. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

ent I/O routes for different processors, and overall code structures and algorithms, can make the prediction of the sweet spot even more complicated. Thus, on a heterogeneous environment, without knowledge of the optimum point for the load ratios among processors, it is not possible to launch the optimum configuration that will result in the best performance. Furthermore, that problem can become more complicated when the number of optimum nodes is unknown. For instance, Fig. 5.97 illustrates the performance of XFLAT on dual-CPU and dual-MIC nodes for three MIC:CPU load ratios when the number of nodes is varied from 1 to 24 nodes. The problem size was set to $10000 \times 10 \times 100$, the run time continued for ~ 100 seconds and the number of computed radial steps was measured. As is evident, from 8 to 12 nodes, there was no significant performance gain, thus utilizing only 8 nodes was the best choice. On the other hand, when the number of nodes was 14, the best load ratio for which the highest performance was achievable was 2:1. Going beyond 16

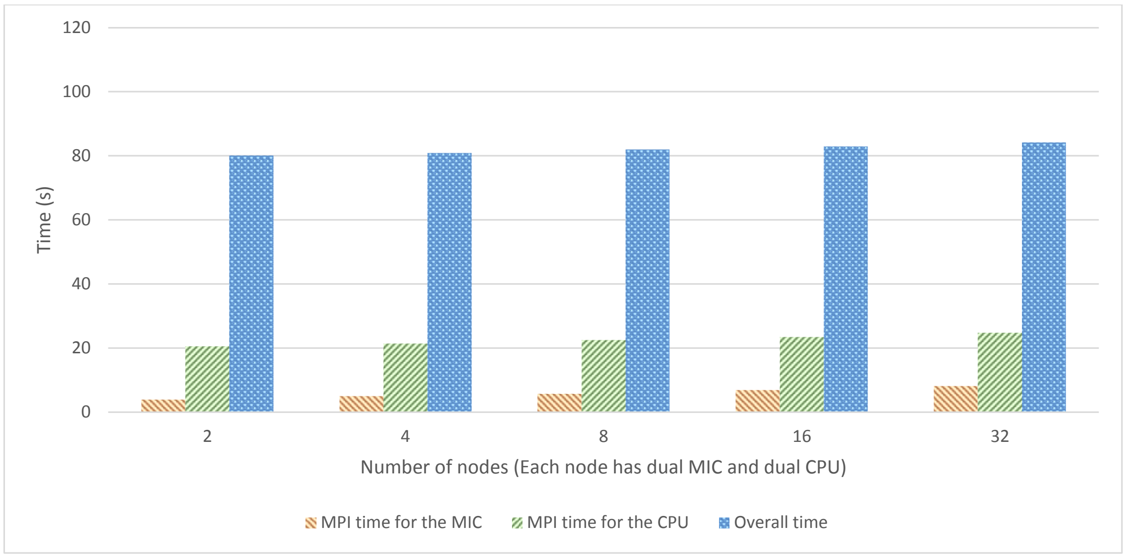


Figure 5.92: Performance of section S2 of algorithm version 1 using quadrupled MPI message size. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

nodes, there was almost no performance gain; thus, utilizing more nodes would only waste resources and yield no noticeable performance improvement.

As a result, the problem of finding the sweet spot can be described as the search in a 2D space of the load ratios and number of nodes, in order to find the optimum point of configuration. Since the optimum sweet spot varies with problem size (as well as with choice of physics module), a capability was added to XFLAT to search this 2D space on a single compute node and to report the near-optimum regions before launching a production run. This code contains two sections. The first section attempts to guess the near optimum computational capability of the CPU and MIC by launching a pre-defined problem size on each processor. After finding the near-optimum load ratios, XFLAT starts to simulate the load on a single node in multi-node configurations by varying the load on a CPU and MIC. XFLAT receives a range for the number of compute nodes (the minimum and maximum number of feasible

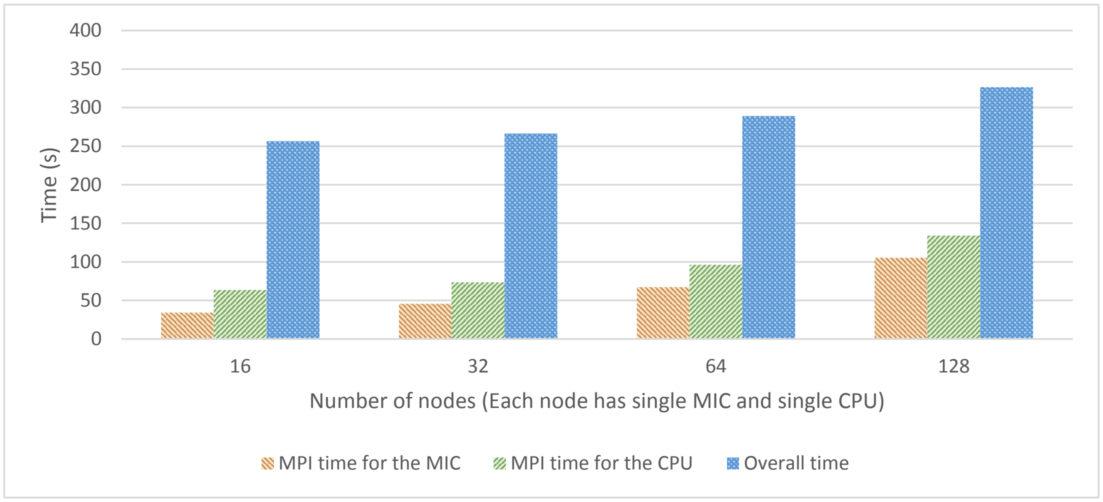


Figure 5.93: Performance the old algorithm (version 1) utilizing newer MPI library (one CPU and one MIC per node) on different numbers of nodes. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

compute nodes) from the configuration file. Afterwards, XFLAT calculates the load on a CPU and MIC for a given number of compute nodes, then starts the benchmark using the actual neutrino evolution code. For each particular load on the benchmark node, XFLAT varies the MIC to CPU load ratio and repeats the benchmark by running the evolution code (see 5.3.9). Hence, XFLAT can iteratively search the compute nodes range and the load ratio space in order to find the sweet spot for a given problem size and for a particular physics module.

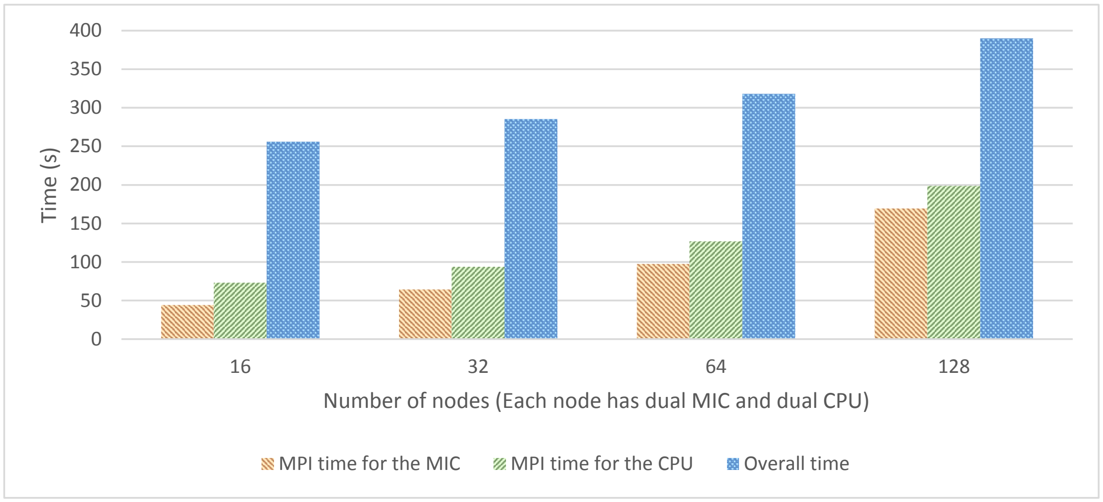


Figure 5.94: Performance the old algorithm (version 1) using newer MPI library (two CPUs and two MICs per node) on different numbers of nodes. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

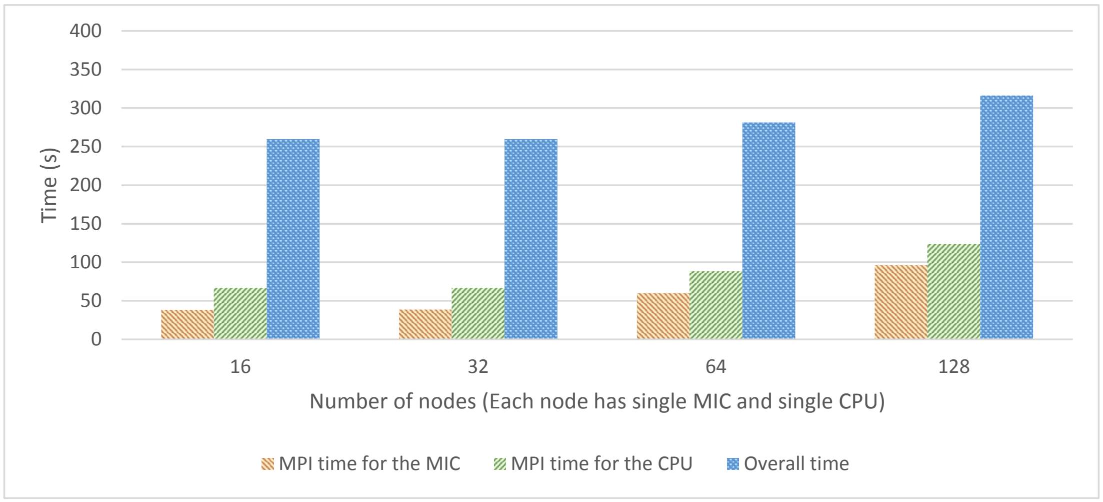


Figure 5.95: Performance the new algorithm (version 3) using newer MPI library (one CPU and one MIC per node) on different numbers of nodes. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

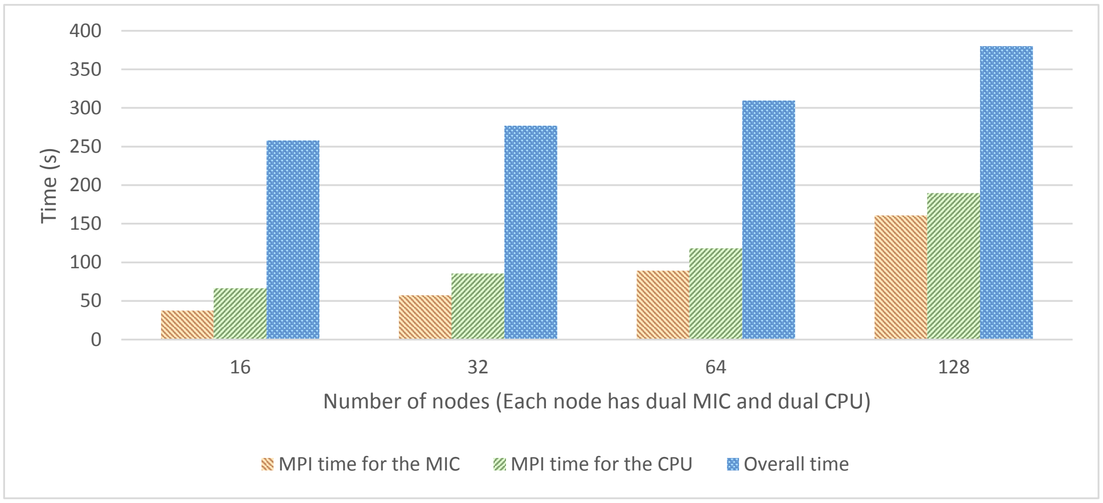


Figure 5.96: Performance the new algorithm (version 3) using newer MPI library (two CPUs and two MICs per node) on different numbers of nodes. The orange bar shows the maximum MPI time among the MICs' processes; the green bar shows the maximum MPI time among the CPUs' processes; and the blue bar shows the overall run time in seconds.

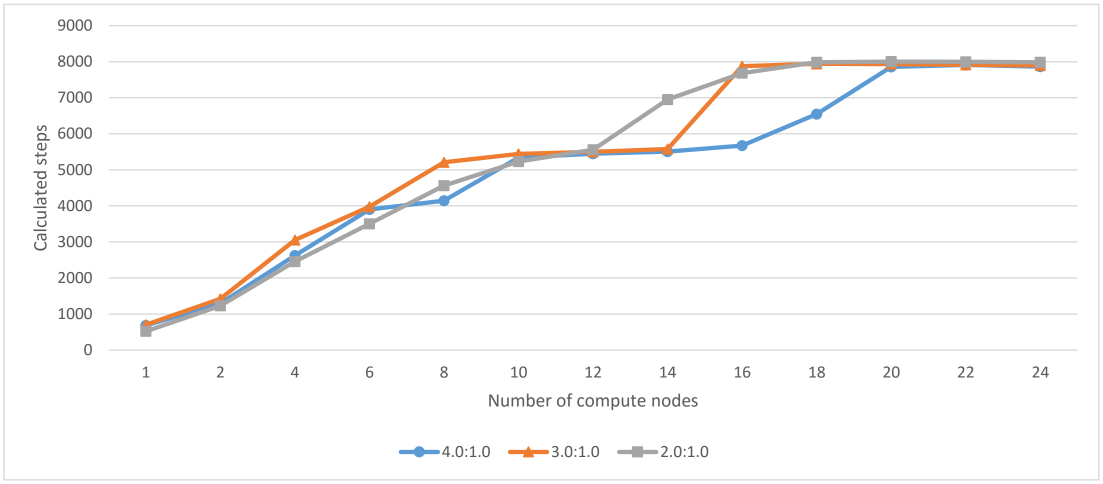


Figure 5.97: XFLAT performance over a range of compute nodes. The problem size was $10000 \times 10 \times 100$. The run time was continued for 100 seconds and the number of computed radial steps was measured. The blue curve shows XFLAT performance when the MIC:CPU ratio is 4; the orange curve shows XFLAT performance when the MIC:CPU ratio is 3; and the grey curve shows XFLAT performance when the MIC:CPU ratio is 2.

```
// Benchmark each processor using a pre-defined problem size
result = Benchmark();

// Search the 2D space iteratively
for (i = minNumNode : maxNumNode)
{
    for (j = minLoadRatios : maxLoadRatios)
    {
        results[i][j] = Benchmark();
    }
}
// Report the results
cout << results
```

Figure 5.98: The benchmark for searching the 2D space of load ratios and compute nodes in order to find the sweet spot for a particular problem size.

Chapter 6

Summary and future work

In this work we have developed an astrophysical simulation code, XFLAT, to study neutrino oscillations in supernovae. Because of its modular design, XFLAT can be easily expanded to investigate neutrino oscillations in various geometries and physical environments.

We have designed XFLAT to utilize all three major levels of parallelism which are available to modern supercomputers, *i.e.* multi-node/device parallelism at the top level, multi-core parallelism within a single compute node/device, and vectorization or SIMD within a single core. In implementing the three levels of parallelism, we have chosen to use open standards to make the code portable. Specifically, we have used MPI for multi-node/device parallelism and OpenMP for both multi-node/device parallelism and vectorization within a single core. In order to efficiently utilize the SIMD hardware in the CPU and Xeon Phi, we have adopted the Structure-of-Arrays scheme for the low-level module. For upper level modules we have used the Array-of-Structure scheme to make the code more modular and thus easy to maintain. The design of XFLAT makes it suitable to run on both the CPU and the Intel Xeon Phi accelerator, the latter of which is based on the Intel Many Core Architecture (MIC).

We have studied the performance of XFLAT with various configurations and in many scenarios on the Stampede supercomputer, as well as two local testbed systems. We find that, in the best scenarios, XFLAT can perform about 3 times as fast on the first-generation the Xeon Phi on an 8-core Xeon CPU, and about 4$\times$ speedup can be achieved on Xeon Phi as compared to a 6-core Xeon CPU. Because the Xeon Phi can be installed as a PCIe extension card to a compatible workstation, our work suggests that the Xeon Phi can be a low-cost choice to dramatically increase the performance of existing computers or compute clusters.

In this work we have found that it can be a great challenge to maintain the load balance in a heterogeneous environment where both the CPU and Xeon Phi are employed. This is because the many-core architecture of the Xeon Phi can support up to about 240 threads per device. There can be a significant drop in the performance of the Xeon Phi when the number of jobs on the device is slightly more than a multiple of the number of its hardware threads.

We have also found that the I/O performance of the first-generation Xeon Phi was very poor. To avoid this I/O penalty we have implemented an indirect I/O module for the Xeon Phi in which output data from the Xeon Phi is redirected to the CPU. This I/O bottleneck may be fixed in the next-generation Xeon Phi.

The recent work by [Raffelt et al., 2013, Duan and Shalgar, 2015, Abbar and Duan, 2015] has shown that the directional, spatial and time symmetries employed in the 3-dimensional neutrino bulb model (with 1 spatial and 2 momentum dimensions) could be broken spontaneously. As a result, simulations of neutrino oscillations in full 7-dimensional supernova models (with 1 temporal, 3 spatial and 3 momentum dimensions) must be performed in order to discover the real impacts of neutrino oscillations on supernova physics. This paradigm shift implies an increase of several orders of magnitude in computational intensity. The development of XFLAT is a first attempt in this direction. In the coming years the XFLAT project will be expanded to include more realistic physical models to simulate the fascinating phenomenon of neutrino oscillations in various physical environments.

Appendix A

XFLAT Documentation

A.1 Compilation and Building Instructions

XFLAT is a command-line application intended to study neutrino flavor oscillation in supernovae environments. The code is a C++ implementation with a hybrid architecture that exploits SIMD, OpenMP and MPI for performance acceleration. It is capable of running supercomputers with heterogeneous nodes including both traditional CPUs and the newer Intel Many Integrated Core Architecture (MIC) or Xeon Phi.

The code contains several modules that can be swapped in or out from the build using provided switches via the Makefile. In addition to the modules, other features can be switched on or off from the Makefile as well. Features such as the use of SIMD, OpenMP, and MPI are controllable from the Makefile. Turning off optimizations can help the debugging process. The following code features can be switched on or off from the second line of the provided Makefile starting with `CXXFLAGS`, and using the `-D` switch:

```
SIMD # when defined the code will use SIMD instructions
OMP # when defined the code will use OpenMP threads
MMPI # when defined the code will use MPI
```

In addition there are multiple modules that can be changed from the same line of the Makefile. These modules can be categorized as follow: Geometry related modules:

```
SA # Single Angle supernova module
MA # Multi Angle supernova module
MAA # Multi Azimutal Angle extended supernova module
CLN # Cylindrical module
PNT # Plane module
LIN # Multi-source line module
```

IO related modules:

```
IOF # performs file IO for each node
IOFI # performs indirect IO in which MIC sends its data to CPU first
```

An example of the line with the usage of IOF and MAA modules which utilizes SIMD, OpenMP and MPI is shown as below:

```
CXXFLAGS = -O3 -openmp -DIOF -DMAA -DSIMD -DOMP -DMMPI
```

The code employs the NetCDF(either version 3 or 4) library for its current IO modules, however, if the NetCDF4 is used, HDF5 library is required as well.

In order to build binary for CPU the following commands must be issued from console:

```
$ cp Makefile.cpu Makefile
$ make all
```

OR

```
$ make all -f Makefile.cpu
```

Likewise, in order to build the code for the Xeon Phi the following commands must be issued:

```
$ cp Makefile.mic Makefile
$ make all
```

OR

```
$ make all -f Makefile.mic
```

Consequently, the CPU binary will be called XFLAT.cpu and the Xeon Phi binary is XFLAT.mic. Please note, if the OpenMP feature is switched on, the `-openmp` flag also must be added to the compiler flags and if the MPI feature is switched on inside the Makefile, a few MPI scripts are required so as to run it on multiple nodes.

In order to optimize the code for any Intel CPU, one has to add `-xHOST` to the compiler flags set in the Makefile. Likewise, in order to build the code for MIC the `-mmic` flag should be added to the compiler flags set.

A.2 Source Code Directories and Files

There are two directories that contains the header files (`include/`) and the source files (`src/`). The header directory contains the following header files:

Fenergy.h: Contains function declarations which are responsible for energy spectra calculations. These functions are called from NBeam module.

Fio.h: Holds function declarations which are responsible for dumping data onto files as well as other I/O related tasks that depends on the settings, different source files can implement those functions.

Global.h: Encloses global settings and constants. These settings affects the general behavior of XFLAT.

Matt.h: Encapsulates the matter profile functions and its related settings. A run may or may not utilize this module.

NBeam.h: Encloses Neutrino Beam class declaration. This class contains all the variables and functions for calculating neutrino beam interaction and evolution. Each neutrino beam in the system must instantiate this class. This module serves as the lower layer module for other upper layer modules.

NBGroup.h: This module encapsulates all the upper layer modules such as the numerical, Physics, and I/O modules.

Nmr.h: Contains numerical related function declaration. Those functions are responsible for solving the numerical algorithm.

Parser.h: Encloses the parser class declaration. This class is responsible for taking a `config.txt` file, parse it, and put the extracted values to the corresponding variables.

Phy.h: Holds physics and geometry related function declaration. Depends on the geometry, different source files can implement those functions.

Util.h: Holds auxiliary functions and variables declarations. These functions may be utilized from the other modules.

The source directory contains the following C++ files:

main.cpp: Holds the main function of the application (the entry point of the application).

Fenergy.cpp: Implements the energy spectra functions and its related procedures.

IO_f.cpp: One of the implementation of `Fio.h` header. In this source file, every node dumps its data directly to its own file.

IO_fi.cpp: One of the implementation of `Fio.h` header. Due to limited I/O capability of the current generation of Intel MIC, Xeon Phi nodes send their data to corresponding CPUs first, the CPU dumps both their own data and the Phis data to file.

Matt.cpp: Implements matter profile functions. Depends on the configuration, the functions may or may not be called in a run.

NBeam.cpp: Implements Neutrino Beam class. The class encapsulates the wavefunctions of a neutrino's beam.

NBGroup.cpp: Encapsulates upper modules. Contains functions for initializing and finalizing other modules.

Nmr.cpp: One of the upper layer modules. Implements the neutrino evolution numerical algorithm.

Parser.cpp: Implements the Parser class. It is responsible for parsing the configuration file.

Phy_CLN.cpp: This is one of the several Physics modules. This module implements the cylindrical geometry.

Phy_MA.cpp: This module implements supernova bulb model (multi-zenith angle geometry).

Phy_MAA.cpp: This module contains the extended supernova physics. It implements the multi-zenith and multi-azimuthal angle geometry.

Phy_PLN.cpp: This module implements the multi-zenith and multi-azimuthal plain geometry.

Phy_SA.cpp: Contains the implementation of the single-angle supernova model. Due to the geometry limitations this module cannot utilize more than a single processor's core.

Util.cpp: This modules holds many general variables and are used in several other modules. It also implements several auxiliary functions.

A.3 The configuration file

In order to run XFLAT, a configuration file has to pass via command line argument to the program. This file contains several settings related to the behavior of the application and many other values for initializing program's variables.

Each keyword in the file must be starts in a new line. Keywords are constant and cannot be change unless the corresponding keyword in the `Parser.cpp` class implementation is changed accordingly. Thus, each keyword contains a constant character string and ends in a `=` and after a white space its value is stored:

```
Keyword1= value1
```

Comments can be added to the configuration file as well. They starts with `#` and continues until reaching the new line character. Therefore, they can be added after the value of a keyword, or can be added in a separate line:

```
# comment1
Keyword1= value1 # extra comment
```

Please note that if an expected variable is not initialized in the configuration file, its initial value will be undefined.

The current keywords can be categorized in two different categories: Those that control the general behavior of the application (mostly related to the I/O tasks) and those from which the physics related variables are initialized.

General keywords are listed as follow:

`dumpMode=` This keyword expects an integer from which the way data dump onto a file is set. The values must be read in binary mode. Putting 0 (also 0 in binary mode) for the value means no dumping data. Putting 1 (also 1 in binary mode) is the first mode which means dumping a whole snapshot in which all the wave-function values are written onto the file. The value 2 (10 in binary mode) means dumping only the weighted average over energy bins onto file. The next independent value can be 4 (100 in binary mode) and so on. Note that in this way values can be combined together as binary flags. Therefore, putting 3 (11 in binary mode) means performing I/O in both the first mode (01 in binary) and second mode (10 in binary).

`filePrefix=` This keyword takes a string as the general name for files. In the application some other strings may attach to it as well. For example when the data dump mode is 1, the "Snapshot" string is also attach to it. Moreover, if multiple files is generated, a counter number starts from 0 is also attached to the file's name afterwards. Finally, for the MPI runtime, each node will attach its own MPI id at the end of the file (separated from the rest of string with '_').

`newFile_step=` Takes an integer value indicating that after a [articular number of I/O task, a new file has to be generated. In this way it is possible to prevent creating a huge file.

`sync_step=` Indicates that after a specific number of data dumping iterations, the file has to be synchronized with the disk. In this way it is possible to prevent data loss due to application crash.

`r_step1=` If for each radial iteration in the evolution loop, data are to be dumped on file, the performance will drop dramatically. In addition, there is no need to perform I/O task for every iteration in the loop as the difference between the two consistent radial steps is normally negligible. Therefore, the value (float) of this keyword indicates that only after advancing a particular distance in Km one snapshot is dumped to file. Hence, there is a radial distance equals to this keyword's value between each snapshot. This keyword only controls the I/O mode of 1, for the second I/O mode the keyword is `r_step2`. As a result, the frequency of saving data can be controlled independently for each I/O mode.

`t_step1=` This keyword is similar to the previous keyword. The I/O is only performed after a particular seconds. Similar to the previous keyword, it only controls the first I/O mode. For the second I/O mode the keyword is `t_step2`.

`itr_step1=` Similar to the previous keywords, the I/O task is allowed to be performed after a particular radial iterations. It only affects the behavior of the first I/O mode. For the second I/O mode the keyword is `itr_step2`.

`start_beam=`, `end_beam=` These two keywords indicates the starting and ending indices of neutrino beams that the current node has to perform computations on. For instance, the distribution of 1000 beams over two identical nodes can be done in this way: for the first node the `start_beam= 0` and `end_beam= 500` and for the second node `start_beam= 500` and `end_beam= 1000`. Thus the 500th beam is the first beam on the second node. The distribution of beams over nodes depends on the Physics module. For instance, for the bulb model, the neutrino beams along the zenith angle are distributed over nodes, however, for the other modules, depends on the geometry the distribution can be different. If the value of these two keywords is a negative number, in order to find distribution range of neutrino beams on each compute node, prior to distributing the load over nodes a benchmark code is performed on each node and based on the computational capability of each node, the starting and ending beam's indices are defined.

`multiNodeBench=` If the value is non zero, it indicates that the multi-node benchmarking should be on. The multi-node benchmarking is the benchmark to find the optimum MIC to CPU load ratios among multiple nodes.

`minNodes=`, `maxNodes=` These two values indicate the minimum and maximum number of desired nodes on which the search for the optimum number of nodes should be performed.

`hasMatter=` Indicates whether or not the matter profile is included into the run. Zero means the matter profile is excluded, and non-zero values indicates the matter profile is included into the run.

`Tn=` The total execution time of the application in second. Note that the time for memory allocations and initilizations is not included, thus this time is only the allowable time for executing the main neutrino evolution loop.

`Ts=` The total number of iteration for the main evolution loop. The program will be finished after reaching the `Tn` seconds or after performing `Ts` radial iteration.

Keywords related to physics are listed as follow:

`eps0=` The error tolerance that indicates the maximum allowable error between two computed wave-functions.

`ksi=` A float value between 0 and 1 for controlling the adaptive step size behavior. It is a safety factor to ensure success on the next try.

`dm2=` The value of the neutrino mass-squared difference. It is positive for the normal mass hierarchy and negative for the inverted mass hierarchy.

`theta=` The vacuum mixing angle.

`R0=` The starting radius in Km.

`Rn=` The final radius in Km.

`dr=` The initial Δr value in Km. Normally less than 1 Km.

`max_dr=` The maximum possible value for 'dr'. The higher values will be trimmed to this value.

`E0=` The starting point of the energy spectra in MeV.

`E1=` The ending point of the energy spectra in MeV.

`Abins=` The number of angle bins along zenith direction. The value is always greater or equal to one.

`Pbins=` The number of azimuth angle bins. The value is always greater or equal to one.

`Ebins=` The number of energy bins over the range of the energy spectra. The value is always greater or equal to one.

`SPoints=` The number of emission surface points, for the multi emission points systems.

`Flvs=` The number of neutrino flavors in the system.

`Ye=` The electron fraction or the net number of electrons per baryon.

`nb0=` The baryon density at the neutrino sphere.

`Rv=` The neutrino sphere radius.

`Mns=` The mass of the neutron star in solar mass unit.

`gs=` The statistical weight in relativistic particles.

`S=` The entropy per baryon.

`hNS=` The scale height.

`Lve=`, `Lv_e=`, `Lvt=`, `Lv_t=` The energy luminosity for electron, anti-electron, tau, and anti-tau neutrinos in erg/s.

`Tve=`, `Tv_e=`, `Tvt=`, `Tv_t=` The neutrino temperature for electron, anti-electron, tau, and anti-tau neutrinos in MeV.

`eta_ve=`, `eta_v_e=`, `eta_vt=`, `eta_v_t=` The degeneracy parameter for electron, anti-electron, tau, and anti-tau neutrinos.

A.4 Methods and Variables of Modules

Here the role of each method and variable within modules is described.

A.4.1 Neutrino Beam module (NBeam.h/NBeam.cpp)

This module is one of the lower layer modules. It contains the NBeam class, which holds arrays of wave-function. Each element of those arrays represent a neutrino in a particular energy bin. The NBeam module has the following functions and class:

`void init(int flavors, int ebins)` First, this function calls the energy spectra module's initialization method. Next, depends on the number of flavors, it allocated several arrays for storing energy bins' values. The size of each array is determined by the number of energy bins. Afterwards, it fills up the energy arrays based on the normalized value of the energy spectra. In addition, it calculates and stores the vacuum Hamiltonian for each bin.

`void freemem()` This method frees up all the arrays that were allocated in the initialization function.

`inline void upd_nu_coef(const double *restrict nu, const double *restrict anu, const double n_cf, const double an_cf, double *restrict ret) throw()`

This inlined function calculates the difference of the multiplication of the energy spectra functions and the density matrices between a neutrino particle and its anti-particle. In addition, each density matrix is weighted by its pre-computed coefficient as follow: $\varrho'_{\nu_\alpha} f_{\nu_\alpha}(E') \frac{L_{\nu_\alpha}}{\langle E_{\nu_\alpha} \rangle} - \varrho'^{*}_{\bar{\nu}_\alpha} f_{\bar{\nu}_\alpha}(E') \frac{L_{\bar{\nu}_\alpha}}{\langle E_{\bar{\nu}_\alpha} \rangle}$

`class NBeam` This class represent a single neutrino beam that contains arrays of wavefunctions expanding over a range of energy spectra. There are several variables and methods inside this class:

- The constructor: The argument is the index that represent the type of the particle (electron neutrino, anti-electron neutrino, etc.) for the beam:

  `NBeam(int prtc);`

- Energy spectra function's setter and getter: Two functions are available to receive the value of energy spectra function. The first method receives an index of energy bin and returns the value of the energy function based on the received index. The second method returns the pointer to the array of energy bins. The pointer can be used to set or get each energy bins' value separately:

  `inline double Fv(int e) const;`

  `inline double* Fv() const;`

- Wave-function's components setter and getter: Sometimes it requires that the wavefunction's components be accessible based on an index (*i.e.* 0 for the real part of the first number, 1 for the imaginary part of the first number, 2 for the real part of the second number, 3 for the imaginary part of the second number, etc.), therefore the following methods are provided to make wave-function's components accessible based on an index number: `inline const double* psi(int cmpn) const throw();`

```
inline double* psi(int cmpn) throw();
```

- Energy bins' setters and getters: These set of functions return the pointer to the components of wavefunction arrays. The pointer can be used to set or get each energy bins' value. The following functions are available for the current version of NBeam class:

```
inline double* Ar();

inline double* Ai();

inline double* Br();

inline double* Bi();

inline const double* Ar() const;

inline const double* Ai() const;

inline const double* Br() const;

inline const double* Bi() const;
```

- General wavefunction's setters and getters: These set of functions can set the value of each energy bin or return the current value of each energy bin. For each wave-function component, there has to be at least one setter and getter functions. The argument of each method is the index of the queried energy bin. The following functions are available for the current version of NBeam class:

```
inline double& Ar(const int e);
inline double& Ai(const int e);

inline double& Br(const int e);

inline double& Bi(const int e);

inline const double& Ar(const int e);

inline const double& Ai(const int e);

inline const double& Br(const int e);

inline const double& Bi(const int e);
```

- Density matrix: The density matrix is calculated from the wavefunction. For instance, a wavefunction with two complex components has a two by two complex density matrix. Yet, since the second row in the matrix can be constructed from the first row, this method only returns the computed first row of the density matrix. The returned value is in a four-element array. There are two possible ways to compute the density matrix from a wavefunction. The first one is by passing an energy bin index, and the other is by passing the wavefunction components as argument:

  ```
  inline void density(const int ebin, Res_t ret) const throw();

  inline void density(const double ar, const double ai, const double
  br, const double bi, Res_t ret) const throw();
  ```

- Neutrinos' evolution: After computing the Hamiltonian, a method is required to evolve the current wavefunctions using the Hamiltonian. There are two methods that can be used for the neutrinos' evolution. The first function, takes an energy bin index, delta-radius, the Hamiltonian, and returns the components of the evolved wavefunction. The second method takes delta-radius, the Hamiltonian, the components of the current wavefunction, and returns the components of the new wave-function:

```
inline void U( const int e, const double dr, const double h_r0, const
double h_i0, const double h_r1, const double h_i1, const double n_ar,
const double n_ai, const double n_br, const double n_bi) throw();
```

```
inline void U( const double dr, const double h_r0, const double h_i0,
const double h_r1, const double h_i1, const double a_r, const double
a_i, const double b_r, const double b_i, double& n_ar, double& n_ai,
double& n_br, double& n_bi) const throw();
```

- Summation over energy bins: The summation over all energy bins is necessary in order to calculate the Hamiltonian, thus a function is provided to calculate the sum and store inside the class:

  ```
  void calcHSum() throw();
  ```

- Energy bins summation: This method returns the previously computed summation over energy bins. The returned value is in a four-element array that is the first row of the summation matrix:

  ```
  void getHSum(Res_t ret) const throw();
  ```

- Neutrino's Beam Evolution: These set of functions receive another neutrino's beam and based on the neutrino-neutrino background Hamiltonian, matter potential, and neutrinos' mass difference term, evolves the wavefunctions:

  ```
  void evolveBinsAvgErr( const NBeam& beam, const int ptc_idx, const
  double dr, const double *restrict hvv, const double hmatt, NBeam&
  beamAvg, NBeam& beamErr ) throw();
  ```

  ```
  void evolveBinsHvvAvg( const NBeam& beam, const int ptc_idx, const
  double dr, const double *restrict hvv, const double hmatt, NBeam&
  beamAvg ) throw();
  ```

  ```
  void evolveBinsAvg( const NBeam& beam, const int ptc_idx, const double
  dr, const double *restrict hvv, const double hmatt, NBeam& beamAvg
  ```

```
) throw();
```

```
void evolveBinsAvg( const int ptc_idx, const double dr, const double
*restrict hvv, const double hmatt, NBeam& beamAvg ) throw();
```

```
void evolveBinsHvv( const NBeam& beam, const int ptc_idx, const double
dr, const double *restrict hvv, const double hmatt ) throw();
```

```
void evolveBins( const NBeam& beam, const int ptc_idx, const double
dr, const double *restrict hvv, const double hmatt ) throw();
```

```
void evolveBins( const int ptc_idx, const double dr, const double *REST
hvv, const double hmatt ) throw();
```

Other than the neutrinos' evolution, several of the methods perform other task such as calculating the summation of energy bins for the background Hamiltonian or taking the average between two neutrino beams or calculating the maximum error between two neutrino beams. Since function fusing can increase the overall performance, it is recommended to use the fused version of the function to perform more tasks on the same data. The particle index (0 for electron neutrino, 1 for anti-electron neutrino, 2 for mu neutrino, 3 for anti-mu neutrino, *etc.*), the delta-radius, the neutrino-neutrino background and the matter potential are passed as arguments too.

- Average of two beams: The average function takes a neutrino beam's array as an argument, then calculates the average between the argument and the current object's beam, and replace the current beam with the result:

  ```
  void addAvg( const NBeam& beam ) throw();
  ```

- Find the maximum error between two beams' wavefunctions: In order to detect whether or not the current delta-radius is appropriate for calculations, the maximum error between two neutrino beams is computed. Two functions are privuded. The first one that takes a neutrino beam and calculates and returns the maximum error over all energy bins. The second function is similar to the first one but also computes the summation over energy bins which can be used in future for the Hamiltonian computations:

```
double calcErr( const NBeam& beam ) const throw();
```

```
double calcErrHvv( const NBeam& beam ) throw();
```

- Maximum calculate error return: If the maximum error is computed in one of the fused function before, this function only returns it:

  ```
  double& getErr() throw();
  ```

A.4.2 Numeric module (Nmr.h/Nmr.cpp)

This module is responsible for the numeric algorithm, and can be replaced with the other numerical modules using different algorithms. There are several functions in this module that have to be implemented:

`int init(int len)` This function is the first function to be called within this module. It takes an integer argument that is the length of the neutrino beams (number of trajectory beams multiplied by number of particles). Afterwards, it allocates the memory for the entire neutrino beam's arrays. The number of arrays may be varied and depends on the algorithm. Next, it calls the `initBeam()` function of the physics module for each of the allocated arrays, in order to initialize them.

`void freemem()` Calls the `freeBeam()` function of the physics module, afterwards performs the neutrino beams' deallocation.

`int evolutionLoop() throw()` This function contains the main neutrinos' evolution loop. In fact, it implements the numerical algorithm. If MPI is enabled, there are several points that nodes communicate and exchange data. At the first point, within the main evolution loop, the master node sends the termination condition, current radius, and delta-radius to all of the nodes. Next, this function calculates the radial advancement based on the number of middle points. Then, it continues by calculating the matter density profile. Afterwards, the Hamiltonian computations are performed. The current algorithm continues with the neutrino beams evolution, error calculations, and if the program is in the multi-node mode, all the nodes exchange maximum local errors in order to find maximum global error. Next, if the maximum error is less that a predefined error threshold an IO module's function is called to perform any necessary IO operations. Finally, it continues the next iteration.

A.4.3 Physics module (Phy.h/Phy_*.cpp)

Currently, there are several physics related module each having different geometry and physics. All of them have to implement at least all the methods within the header file (Phy.h). These general methods are as the following:

`void init()` The Initilization method of the physics module that performs memory allocations in a specific order and arrays initialization.

`void freemem()` Deallocates all the allocated memories for this module.

`int beamLen()` Returns the length of the current neutrino beam's array. It can be used from other modules to find out about the neutrino beams' length for memory allocation purposes (*e.g.* $\Theta \times \Phi \times$ *num of particles*)

`int getDim()` Returns the number of dimensions for the data related to the plugged-in physics module. This can be useful from the I/O modules in order to format the data for the NetCDF file (*e.g.* 6 = [r, theta, phi, num_of_particles, wavefunction's components, E])

`void getDimInfo(std::string str[])` Returns an array containing strings for the names of each dimension (*e.g.* 6 = ["r" "theta", "phi", "prtcl", "comp", "ebin"]) `size_t*startDim()` Returns an array of starting point of each data dimension. To be used by the I/O module (*e.g.* [current_radius, 0, 0, 0, 0, 0])

`size_t* countDim()` Returns an array containing the length of each data dimension. This can determine the size of the current snapshot. To be used by the I/O module (*e.g.* [1, theta_bins_num, phi_bins_num, num_of_particles, num_of_components, num_of_energyBins])

`int& startBeamIdx()` Returns the starting beam number for the dimension that is distributed over nodes. The value may vary on each node (*e.g.* 0 for the first node, 500 for the second node, 1000 for the third node, 2000 for the fourth node)

`int& endBeamIdx()` Returns the length of the dimension that is distributed over nodes. The value may vary on each node (*e.g.* 500 for the first node, 1000 for the second node, 2000 for the third node, 3000 for the fourth node)

`int firstDimLen()` Returns the size of the dimension on which the problem size is distributed over nodes (*e.g.* depending on the module can be theta_bins, or phi_bins, *etc.*)

`void initBeam(NBeam* beam)` Receives an array of NBeam objects and initializes each object's internal arrays (Particles and Ebins) accordingly by calling each the constructor.

`void freeBeam(NBeam* beam)` Receives an array of NBeam objects and calls the deconstructor of each object.

`void calcAngleBins(const double r, const int step_num)` The The functionality depends on the module but normally calculates and caches the cosine bins at the current radius r and for different step numbers (current point, mid-point, full-point). Depends on the module may calculates other angle bins as well.

`void calcDeltaLs(const double dr, const int cur_pnt, const int s_pnt,` `const int e_pnt)` Calculates and caches δl for each angle bins. The 'dr' parameter is the radius difference, the 'cur_pnt' is the point at which the δl is calculated and the last two points are the points at which the average is calculated. (*i.e.* $dl[cur\_pnt] = dr/(.5*(cos[s\_pnt]+cos[e\_pnt]))$)

`void newHvv(double*& hvv)` Allocates memory for an array of Hamiltonians.

`void deleteHvv(double*& hvv)` Deletes the allocated memory for Hamiltonian array.

`void avgBeam(const NBeam *restrict ibeam, NBeam *restrict obeam)` This function only calculates the average of the two input neutrino beam arrays and store the result into the second beam.

The following methods receive an array of NBeam objects, and evolve and save them into an output array of NBeam objects. They may receive the matter profile's value and Hamiltonian array `hvv`. In Addition, they may perform other tasks including the calculation of partial summation for neutrino-neutrino background over energy bins, or calculating the average of two NBeam arrays and store them into the last parameter:

- `void evolve(const nbm::NBeam *restrict ibeam, const int pnt, const double *restrict hvv, const double hmatt, nbm::NBeam *restrict obeam) throw();`
- `void evolveHvv(const nbm::NBeam *restrict ibeam, const int pnt, const double *restrict hvv, const double hmatt, nbm::NBeam *restrict obeam) throw();`
- `void evolveAvg(const nbm::NBeam *restrict ibeam, const int pnt, const double *restrict hvv, const double hmatt, nbm::NBeam *restrict obeam, nbm::NBeam *restrict obeamAvg) throw();`

- `void evolveHvvAvg(const nbm::NBeam *restrict ibeam, const int pnt, const double *restrict hvv, const double hmatt, nbm::NBeam *restrict obeam, nbm::NBeam *restrict obeamAvg) throw();`
- `void evolveAvgErr(const nbm::NBeam *restrict ibeam, const int pnt, const double *restrict hvv, const double hmatt, nbm::NBeam *restrict obeam, nbm::NBeam *restrict obeamAvg, nbm::NBeam *restrict obeamErr) throw();`
- `void evolveAvg(const int pnt, const double *restrict hvv, const double hmatt, nbm::NBeam *restrict iobeam, nbm::NBeam *restrict obeamAvg) throw();`

A.4.4 I/O module (Fio.h/IO_f.cpp, IO_fi.cpp)

Currently, there are two implemented I/O modules in XFLAT. When the first I/O module (IO_f.cpp) is employed, each node will dump data onto its own file using the NetCDF. However, since the I/O performance is very poor on Xeon Phi and may cause a serious bottleneck on heterogeneous environments, another module is provided (IO_fi.cpp) that indirectly sends the Xeon Phi's data to the corresponding CPU. Therefore, CPU is responsible to write down its own data as well as the Xeon Phi's data onto files. Here are the public methods for the I/O modules:

`void init(int file_counter=0)` Initializes the I/O module. In addition, it receives a counter number which indicates the number of generated files so far. The counter is provided since sometimes it is not possible to store all the data onto a single file. Thus, this function again can be called from the I/O module for initializing another file.

`void freemem()` Deallocates all the allocated memories for this module.

`void fillInitData(NBeam* nubeam)` It is possible to resume the code using the previously generated data file. In that case, this function receives an array of NBeam objects in order to initialize the neutrinos' state function by using the provided data. The data file is the second argument that is passed from the console to the code.

`void dumpToFile( const NBeam *restrict nubeam, const int itr, const double r )` This is the main function for saving data. It receives an array of NBeam objects as well as the current iteration number and radius. If the current iteration or radius have reached a predefined thresholds, it stores the data onto file.

A.5 The Dependency of Functions in XFLAT

This section describes the function call hierarchy in XFLAT.

`NBGroup` Module:

Listing A.1: NBGroup::init()

```
\\*****************************************************
\\ Initialization
void init()
{
\\\ Performs benchmarks for heterogeneous multi-node runs
    node_benchmark();
\\\ Initialize other modules
    nbm::init(Util::Flvs(), Util::Ebins());
    phy::init();
    fio::init();
    matt::init();
}
```

Listing A.2: NBGroup::particleLoop()

```
\\********************************************************
\\ Calls the neutrino evolution loop from the numeric module
void particleLoop()
{
    nmr::evolutionLoop();
}
```

Listing A.3: NBGroup::freeme()

```
\\********************************************************
\\ Finalization
void freemem()
{
\\\ Free up memories of each module
    fio::freemem();
    phy::freemem();
}
```

`Numeric` Module:

Listing A.4: Nmr::evolutionloop()

```
\\********************************************************
\\ The main neutrino evolution loop
int evolutionLoop()
{
\\\ Allocates Hamiltonian arrays
    phy::newHvv(h);
while(...)
{
```

```
\\\ Get the matter values
    matt::getHm(...);
\\\ Calculates angles dependent values per each bin
    phy::calcAngleBins(...);
\\\ Calculates dl from dr
    phy::calcDeltaLs(...);
\\\ Calculates the Hamiltonian at each point
    phy::calc_Hvv(...);
\\\ Evolves neutrino beams with the calculated Hamiltonian
    phy::evolve(...);
\\\ Calculates the average flavor states between two beams
    phy::avgBeam(...);
\\\ Conditionally saves the results onto file
    fio::dumpToFile(...);
}
\\\ Deallocates the memory of Hamiltonian arrays
    phy::deleteHvv(...);
}
```

`NBeam` Module:

Listing A.5: NBeam::evolveBins()

```
\\*******************************************************
\\ Computes the result of the neutrino evolution per bin
\\ To be called from the Physics module
void NBeam::evolveBins(...)
{
...
}
```

Listing A.6: NBeam::calcErr()

```
\\**********************************************************
\\ Calculates the maximum error between two NBeam objects
\\ To be called from the Physics module
double NBeam::calcErr(...)
{
...
}
```

Listing A.7: NBeam::addAvg()

```
\\**********************************************************
\\ Adds two NBeam flavor states together, afterwards saves
\\ the result onto 'this' object
void NBeam::addAvg(...)
{
...
}
```

Listing A.8: NBeam::calcESum()

```
\\**********************************************************
\\ Calculates the summation of flavor states
\\ The loop is over energy bins within the NBeam object
void NBeam::calcESum()
{
...
}
```

`Physics` Module:

Listing A.9: Phy::evolve()

```
\\*******************************************************
\\ Loops over all the neutrino beams
\\ calls the NBeam::evolveBin() function per each angle beam
void evolve(...)
{
    for (angle : ANGLE_BEAMS)
        NBeam::evolveBins(...);
}
```

Listing A.10: Phy::avgBeam()

```
\\*******************************************************
\\ Loops over all the neutrino beams, calls the addAvg()
void avgBeam(...)
{
    for (angle : ANGLE_BEAMS)
        NBeam::addAvg(...);
}
```

Listing A.11: Phy::calc_Hvv()

```
\\*******************************************************
\\ Calculates the Hamiltonian
void calc_Hvv(...)
{
\\\ Calculates the angle dependent results of the partial
\\\ Hamiltonian
    getHvv_partial(...);
\\\ In multi−node env. exchange the results between all nodes
        MPI_Allreduce(...);
\\\ Loops over all angles to compute the final Hamiltonian
    for (angle : ANGLE_BEAMS)
        getHvv(...);
}
```

I/O Module:

Listing A.12: IOf::fillInitData()

```
\\******************************************************
\\ Fills wavefunctions by the flavor states from a previous
\\ run (loaded from a file)
void fillInitData(...)
{
    for (bm : NEUTRINO_BEAMS)
        memcpy(&NBeam[bm].psi, &data);
}
```

Listing A.13: IOf::dumpToFile()

```
\\******************************************************
\\ Based on the verbose mode decides which function to call
\\ for performing different writes
void dumpToFile(...)
{
...
}
```

Appendix B

Kernels

In this appendix the majority of the developed codes for the benchmarked kernels are provided. The codes can be compiled with the Intel C++ compilers, although the gcc compiler should be able to build most of them.

B.1 Raw Performance Benchmarks of the Xeon Phi

Code for the benchmarking the Xeon Phi was based on [Intel, 2013] as a starting point for floating point benchmark and modified for the transcendental function benchmarks. The following code was used to benchmark MADD (multiply and addition) and transcendental function throughput.

The compiler flags for the CPU were as follow:

```
icpc -O3 -openmp bench.c -xHOST
```

The compiler flags for the Xeon Phi were as follow:

```
icpc -O3 -openmp bench.c -mmic
```

Listing B.1: Benchmark for MADD and transcendental functions

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <omp.h>
#include <sys/time.h>
#include <cmath>

// dtime
//
// returns the current wall clock time
//
double dtime()
{
    double tseconds = 0.0;
    struct timeval mytime;
    gettimeofday(&mytime,(struct timezone*)0);
    tseconds = (double)(mytime.tv_sec+mytime.tv_usec*1.0e-6);
    return( tseconds );
}

#define FLOPS_ARRAY_SIZE (1024*1024)
#define MAXFLOPS_ITERS 1000000
#define LOOP_COUNT 4096

// number of float pt ops per calculation
#define FLOPSPERCALC 2
```

```
#define REAL double

// define some arrays -
// make sure they are 64 byte aligned
// for best cache access
REAL fa[FLOPS_ARRAY_SIZE] __attribute__((align(64)));
REAL fb[FLOPS_ARRAY_SIZE] __attribute__((align(64)));

int main(int argc, char *argv[] )
{
    int i,j,k;
    int numthreads;
    double tstart, tstop, ttime;
    double gflops = 0.0;
    REAL a=.05;

    //
    // initialize the compute arrays
    //
    //

#pragma omp parallel
#pragma omp master
    numthreads = omp_get_num_threads();

    printf("Initializing\r\n");
    #pragma omp parallel for
    for(i=0; i<FLOPS_ARRAY_SIZE; i++)
    {
        fa[i] = (REAL)i * 0.1;
        fb[i] = (REAL)i * 0.2;
    }
    printf("Starting Compute on %d threads\r\n",numthreads);

    tstart = dtime();

// scale the calculation across threads requested need to
// set environment variables OMP_NUM_THREADS and KMP_AFFINITY
    #pragma omp parallel for private(j,k)
    for (i=0; i<numthreads; i++)
    {
        // each thread will work it's own array section
        // calc offset into the right section
        int offset = i*LOOP_COUNT;

        // loop many times to get lots of calculations
        for(j=0; j<MAXFLOPS_ITERS; j++)
        {
            // scale 1st array and add in the 2nd array
            #pragma omp simd
            for(k=0; k<LOOP_COUNT; k++)
            {
                    /// FMADD benchmark
                fa[k+offset] = a*fa[k+offset] + fb[k+offset];

                /// sin()/cos() benchmark
                //fa[k+offset] = sin(fa[k+offset]);
                //fb[k+offset] = cos(fb[k+offset]);

                /// exp() benchmark
                //fa[k+offset] = exp(fa[k+offset]*.000001);
                //fb[k+offset] = exp(fb[k+offset]*.000003);

            }
        }
    }
    tstop = dtime();
    // # of gigaflops we just calculated
    gflops = (double)( 1.0e-9*numthreads*LOOP_COUNT*
                        MAXFLOPS_ITERS*FLOPSPERCALC);

    //elasped time
    ttime = tstop - tstart;

    //
    // Print the results
    //
    if ((ttime) > 0.0)
    {
        printf("GFlops=%e, Secs=%e, GFlops_per_sec=%e\r\n",
                gflops, ttime, gflops/ttime);
    }
    return( 0 );
}
```

B.2 SoA and AoS Benchmarks

Listing B.2: Structure of Arrays

```
class Data{
public:
    double ar[], ai[], br[], bi[];
    ...
    void init(double x) {
        #pragma omp simd
        for (int j = 0; j < SIZE; ++j) {
            /// vectorized computations
        }
    }
};

int main(int argc, char** argv) {
    ...
    #pragma omp parallel for
    for (int i = 0; i < LEN; ++i) {
        ...
        for (int k = 0; k < L; ++k) {
            /// call array with LEN length, in parallel

        }
    }
    ...
}
```

Listing B.3: Array of Structures

```
class Data
{
public:
    double ar, ai, br, bi;
    ...
    void init(double x)
    {
        /// serial computations on class' members
    }
};

int main(int argc, char** argv)
{
    ...
    #pragma omp parallel for
    for (int i = 0; i < LEN * SIZE; ++i)
    {
        ...
        for (int k = 0; k < L; ++k)
        {
            /// call array with LEN*SIZE length, in parallel
            data[index].init(...);
        }
    }
}
```

B.3 Function Arguments and Their Performance

Listing B.4: The effect of function arguments within vectorized loops

```
void func2( const double ar, const double ai,
           const double br, const double bi,
           double& res_r, double& res_i ) {
        /// Computations!
}

void func1(const int e, double& res_r, double& res_i) {
        /// Computations!
}

int main(int argc, char** argv)
{
    ...
    for (int z = 0; z < N; ++z)
    {
        #pragma omp simd
        for (int e = 0; e < E; ++e)
        {
            /// Either one of the should be commented out
            func1(e, mm[e], nn[e]);
            func2(ar[e], ai[e], br[e], bi[e], mm[e], nn[e]);
        }
    }
}
```

B.4 OpenMP Parallel Loops

OpenMP loops can be written in various ways that may affect the performance.

Listing B.5: First approach for parallelizing a region via a parallel region that encloses everything and single regions within the loop.

```
int main() {
    ...
    #pragma omp parallel {
        for (int i = 0; i < N; ++i) {

            #pragma omp single {
                p[i/SIZE] += ...
                q[i/SIZE] += ...
            }
            #pragma omp for
            for (int j = 0; j < SIZE; ++j) {
                c[j] += ...
                b[j] += ...
                a[j] += ...
            }

            #pragma omp single {
                r[i/SIZE] -= ...
                s[i/SIZE] -= ...
            }
            #pragma omp for
            for (int j = 0; j < SIZE; ++j) {
                z[j] += ...
                y[j] += ...
                x[j] += ...
            }

            #pragma omp single {
                m[i/SIZE] -= ...
                q[i/SIZE] += ...
            }
            #pragma omp for
            for (int j = 0; j < SIZE; ++j) {
                d[j] -= ...
                e[j] -= ...
                f[j] -= ...
            }

            #pragma omp single {
                n[i/SIZE] += ...
                s[i/SIZE] -= ...
            }
            #pragma omp for
            for (int j = 0; j < SIZE; ++j) {
                u[j] -= ...
                v[j] -= ...
                w[j] -= ...
            }
        }
    }
}
```

Listing B.6: Second approach for parallelizing a region via a parallel region inside the main loop that encloses everything.

```
int main() {
    ...
    for (int i = 0; i < N; ++i) {

        #pragma omp parallel {

            #pragma omp single {
                p[i/SIZE] += ...
                q[i/SIZE] += ...
            }
            #pragma omp for
            for (int j = 0; j < SIZE; ++j) {
                c[j] += ...
                b[j] += ...
                a[j] += ...
            }

            #pragma omp single {
                r[i/SIZE] -= ...
                s[i/SIZE] -= ...
            }
            #pragma omp for
            for (int j = 0; j < SIZE; ++j) {
                z[j] += ...
                y[j] += ...
                x[j] += ...
            }

            #pragma omp single {
                m[i/SIZE] -= ...
                q[i/SIZE] += ...
            }
            #pragma omp for
            for (int j = 0; j < SIZE; ++j) {
                d[j] -= ...
                e[j] -= ...
                f[j] -= ...
            }

            #pragma omp single {
                n[i/SIZE] += ...
                s[i/SIZE] -= ...
            }
            #pragma omp for
            for (int j = 0; j < SIZE; ++j) {
                u[j] -= ...
                v[j] -= ...
                w[j] -= ...
            }
        }
    }
}
```

Listing B.7: Third approach for parallelizing a region via a parallel region that encloses everything and single regions within the loop. Threads at the end of parallel for loop does not wait for the other threads.

```
int main() {
    ...
    #pragma omp parallel {

        for (int i = 0; i < N; ++i) {

            #pragma omp single {
                p[i/SIZE] += ...
                q[i/SIZE] += ...
            }
            #pragma omp for nowait
            for (int j = 0; j < SIZE; ++j) {
                c[j] += ...
                b[j] += ...
                a[j] += ...
            }

            #pragma omp single {
                r[i/SIZE] -= ...
                s[i/SIZE] -= ...
            }
            #pragma omp for nowait
            for (int j = 0; j < SIZE; ++j) {
                z[j] += ...
                y[j] += ...
                x[j] += ...
            }

            #pragma omp single {
                m[i/SIZE] -= ...
                q[i/SIZE] += ...
            }
            #pragma omp for nowait
            for (int j = 0; j < SIZE; ++j) {
                d[j] -= ...
                e[j] -= ...
                f[j] -= ...
            }

            #pragma omp single {
                n[i/SIZE] += ...
                s[i/SIZE] -= ...
            }
            #pragma omp for nowait
            for (int j = 0; j < SIZE; ++j) {
                u[j] -= ...
                v[j] -= ...
                w[j] -= ...
            }
        }
    }
}
```

Listing B.8: Fourth approach for parallelizing a region via separated parallel for regions.

```
int main()
{
    ...
    for (int i = 0; i < N; ++i)
    {

        p[i/SIZE] += ...
        q[i/SIZE] += ...

        #pragma omp parallel for
        for (int j = 0; j < SIZE; ++j)
        {
            c[j] += ...
            b[j] += ...
            a[j] += ...
        }

        r[i/SIZE] -= ...
        s[i/SIZE] -= ...

        #pragma omp parallel for
        for (int j = 0; j < SIZE; ++j)
        {
            z[j] += ...
            y[j] += ...
            x[j] += ...
        }

        m[i/SIZE] -= ...
        q[i/SIZE] += ...

        #pragma omp parallel for
        for (int j = 0; j < SIZE; ++j)
        {
            d[j] -= ...
            e[j] -= ...
            f[j] -= ...
        }

        n[i/SIZE] += ...
        s[i/SIZE] -= ...

        #pragma omp parallel for
        for (int j = 0; j < SIZE; ++j)
        {
            u[j] -= ...
            v[j] -= ...
            w[j] -= ...
        }

    }
}
```

B.5 Benchmarks of the I/O Loops

Listing B.9: Saving data via NetCDF within nested loops.

```
void IO(NBeam* beams)
{
    ...
    for (int tet = 0; tet < thetas; ++tet)
    {
        start[1] = tet;
        for (int phi = 0; phi < phis; ++phi)
        {
            start[2] = phi;
            for (int p = 0; p < P; ++p)
            {
                start[3] = p;
                for (int c = 0; c < C; ++c)
                {
                    start[4] = c;
                    NCRUN( nc_put_vara_double(nc1id, psid,
                        start,count,
                        beams[(tet*phis+phi)*P+p].psi(c)));
                }
            }
        }
    }
    ...
}
```

Listing B.10: Saving data via NetCDF within a single loop.

```
void IO(NBeam* beams)
{
    ...
    for (int i = 0; i < thetas*phis*P; ++i)
    {
        /// Calculates the proper indecis

        size_t i0 = i  / (phis*P);
        size_t j0 = i  % (phis*P);
        size_t i1 = j0 / P;
        size_t i2 = j0 % P;

        start[1] = i0; start[2] = i1; start[3] = i2;

        for (int c = 0; c < C; ++c)
        {
            start[4] = c;
            NCRUN( nc_put_vara_double(nc1id, psid,
                start, count,
                beams[i].psi(c)) );
        }
    }
    ...
}
```

Listing B.11: Saving data via NetCDF after completion of a single loop. Within the loop data may be extracted from NBeam objects and store onto a buffer.

```
void IO(NBeam* beams)
{
    ...
    for (int i = 0; i < thetas*phis*P; ++i)
    {
        size_t i0 = i  / (phis*P);
        size_t j0 = i  % (phis*P);
        size_t i1 = j0 / P;
        size_t i2 = j0 % P;

        for (int c = 0; c < C; ++c)
        {
            for (int e = 0; e < ebins; ++e)
                buffer[(i*C+c)*ebins+e] = beams[i].psi(c)[e];

        }
    }
    NCRUN( nc_put_vara_double(  nc1id, psid,
                                start, count, buffer) );
    ...
}
```

References

[Abbar and Duan, 2015] Abbar, S. and Duan, H. (2015). Neutrino flavor instabilities in a time-dependent supernova model. *Phys. Lett.*, B751:43–47.

[Agafonova et al., 2010] Agafonova, N., Aleksandrov, A., Altinok, O., Ambrosio, M., Anokhina, A., Aoki, S., Ariga, A., Ariga, T., Autiero, D., Badertscher, A., et al. (2010). Observation of a first $\nu\tau$ candidate event in the OPERA experiment in the CNGS beam. *Physics Letters B*, 691(3):138–145.

[An et al., 2012] An, F., Bai, J., Balantekin, A., Band, H., Beavis, D., Beriguete, W., Bishai, M., Blyth, S., Boddy, K., Brown, R., et al. (2012). Observation of electron-antineutrino disappearance at Daya Bay. *Physical Review Letters*, 108(17):171803.

[Aprà et al., 2014] Aprà, E., Klemm, M., and Kowalski, K. (2014). Efficient implementation of many-body quantum chemical methods on the intel&reg; xeon phi&trade; coprocessor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 674–684, Piscataway, NJ, USA. IEEE Press.

[ArsTechnica, 2015] ArsTechnica (2015). ArsTechnica. `http://cdn.arstechnica.net/paedia/c/caching/images/figure04-caching-pyramid.png`. Accessed: 2015-09-03.

[Atanassov et al., 2003] Atanassov, E., Dimitrov, D., and Ivanovska, S. (2003). Efficient Implementation of the Heston Model Using GPGPU. *J. of Monte Carlo Methods and Applications, J. of Monte Carlo Methods and Applications*, pages 21–28.

[Bahcall et al., 2005] Bahcall, J. N., Serenelli, A. M., and Basu, S. (2005). New solar opacities, abundances, helioseismology, and neutrino fluxes. *The Astrophysical Journal Letters*, 621(1):L85.

[Barger et al., 2012] Barger, V., Marfatia, D., and Whisnant, K. (2012). *The Physics of Neutrinos*. Princeton University Press.

[Belleman et al., 2008] Belleman, R. G., Bdorf, J., and Portegies Zwart, S. F. (2008). High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA. *New Astronomy*, 13(2):103–112.

[Bionta et al., 1987] Bionta, R., Blewitt, G., Bratton, C., Casper, D., Ciocio, A., Claus, R., Cortez, B., Crouch, M., Dye, S., Errede, S., et al. (1987). Observation of a neutrino burst in coincidence with supernova 1987A in the Large Magellanic Cloud. *Physical Review Letters*, 58(14):1494.

[Brown et al., 2012] Brown, W. M., Kohlmeyer, A., Plimpton, S. J., and Tharrington, A. N. (2012). Implementing molecular dynamics on hybrid high performance computers–Particle–particle particle-mesh. *Computer Physics Communications*, 183(3):449–459.

[Brown et al., 2011] Brown, W. M., Wang, P., Plimpton, S. J., and Tharrington, A. N. (2011). Implementing molecular dynamics on hybrid high performance computers–short range forces. *Computer Physics Communications*, 182(4):898–911.

[Buck, 2007] Buck, I. (2007). Gpu computing with nvidia cuda. In *SIGGRAPH*, volume 7, page 6.

[Calvin et al., 2013] Calvin, C., Ye, F., and Petiton, S. (2013). The Exploration of Pervasive and Fine-Grained Parallel Model Applied on Intel Xeon Phi Coprocessor. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pages 166–173. IEEE.

[Chrysos and Engineer, 2012] Chrysos, G. and Engineer, S. P. (2012). Intel Xeon Phi Coprocessor (codename Knights Corner). *Proceedings of the 24th Hot Chips Symposium, HC*.

[Costigan and Scott, 2007] Costigan, N. and Scott, M. (2007). Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3. *IACR Cryptology ePrint Archive*, 2007:61.

[Crimi et al., 2013] Crimi, G., Mantovani, F., Pivanti, M., Schifano, S., and Tripiccione", R. (2013). Early Experience on Porting and Running a Lattice Boltzmann Code on the Xeon-phi Co-Processor. *Procedia Computer Science*, 18(0):551 – 560. 2013 International Conference on Computational Science.

[Dasgupta et al., 2009a] Dasgupta, B., Dighe, A., Raffelt, G. G., and Smirnov, A. Y. (2009a). Multiple spectral splits of supernova neutrinos. *Phys. Rev. Lett.*, 103:051105.

[Dasgupta et al., 2009b] Dasgupta, B., Dighe, A., Raffelt, G. G., and Smirnov, A. Yu. (2009b). Multiple Spectral Splits of Supernova Neutrinos. *Phys. Rev. Lett.*, 103:051105.

[Davis et al., 1968] Davis, R., Harmer, D. S., and Hoffman, K. C. (1968). Search for Neutrinos from the Sun. *Physical Review Letters*, 20:1205–1209.

[Duan et al., 2008] Duan, H., Fuller, G. M., and Carlson, J. (2008). Simulating nonlinear neutrino flavor evolution. *Comput. Sci. Disc.*, 1(1):015007.

[Duan et al., 2006] Duan, H., Fuller, G. M., Carlson, J., and Qian, Y.-Z. (2006). Simulation of coherent nonlinear neutrino flavor transformation in the supernova environment: Correlated neutrino trajectories. *Phys. Rev. D*, 74(10):105014.

[Duan et al., 2010] Duan, H., Fuller, G. M., and Qian, Y.-Z. (2010). Collective neutrino oscillations. *Ann. Rev. Nucl. Part. Sci.*, 60:569.

[Duan and Shalgar, 2014] Duan, H. and Shalgar, S. (2014). Multipole expansion method for supernova neutrino oscillations. *JCAP*, 1410(10):084.

[Duan and Shalgar, 2015] Duan, H. and Shalgar, S. (2015). Spontaneous breaking of spatial symmetries in collective neutrino oscillations. *Phys.Lett.*, B747:139.

[Fang et al., 2014] Fang, J., Sips, H., Zhang, L., Xu, C., Che, Y., and Varbanescu, A. L. (2014). Test-driving Intel Xeon Phi. *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, pages 137–148.

[Firasta et al., 2008] Firasta, N., Buxton, M., Jinbo, P., Nasri, K., and Kuo, S. (2008). Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. *Intel white paper.*

[Fog, 2015] Fog, A. (2015). Intel's "cripple AMD" function. `http://www.agner.org/optimize/blog/read.php?i=49#49`. Accessed: 2015-08-21.

[Fukuda et al., 1998] Fukuda, Y., Hayakawa, T., Ichihara, E., Inoue, K., Ishihara, K., Ishino, H., Itow, Y., Kajita, T., Kameda, J., Kasuga, S., et al. (1998). Evidence for oscillation of atmospheric neutrinos. *Physical Review Letters*, 81(8):1562.

[Geer, 2005] Geer, D. (2005). Chip makers turn to multicore processors. *Computer*, 38(5):11–13.

[Giacobbe, 2005] Giacobbe, F. W. (2005). How a Type II Supernova Explodes. *Electronic Journal of Theoretical Physics*, 2(6):30–38.

[Gregory and Miller, 2012] Gregory, K. and Miller, A. (2012). *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®.* " O'Reilly Media, Inc.", Sebastopol, CA, USA.

[Halyo et al., 2014] Halyo, V., LeGresley, P., Lujan, P., Karpusenko, V., and Vladimirov, A. (2014). First evaluation of the cpu, gpgpu and mic architectures for real time particle tracking based on hough transform at the lhc. *Journal of Instrumentation*, 9(04):P04005.

[Hamada et al., 2009] Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K., and Taiji, M. (2009). 42 Tflops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 62. ACM.

[Harish and Narayanan, 2007] Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the GPU using CUDA. In *High performance computing–HiPC 2007*, pages 197–208. Springer.

[Heinecke et al., 2013] Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., Shet, A. G., Chrysos, G., and Dubey, P. (2013). Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 126–137. IEEE.

[Hess et al., 2008] Hess, B., Kutzner, C., Van Der Spoel, D., and Lindahl, E. (2008). GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation*, 4(3):435–447.

[Hirata et al., 1987] Hirata, K., Kajita, T., Koshiba, M., Nakahata, M., Oyama, Y., Sato, N., Suzuki, A., Takita, M., Totsuka, Y., Kifune, T., et al. (1987). Observation of a neutrino burst from the supernova SN1987A. *Physical Review Letters*, 58(14):1490.

[Intel, 2013] Intel (2013). Intel Xeon Phi Coprocessor Architecture Overview. `https://goo.gl/lnVLlT`. Accessed: 2014-07-20.

[Intel, 2015a] Intel (2015a). Intel MPI. `https://software.intel.com/en-us/intel-mpi-library`. Accessed: 2015-07-21.

[Intel, 2015b] Intel (2015b). SoA. `https://software.intel.com/sites/default/files/m/d/4/1/d/8/tickertape2.gif`. Accessed: 2015-09-03.

[Intel, 2015c] Intel (2015c). VTune. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`. Accessed: 2015-07-21.

[Jang et al., 2008] Jang, H., Park, A., and Jung, K. (2008). Neural network implementation using cuda and openmp. *Digital Image Computing: Techniques and Applications (DICTA), 2008*, pages 155–161.

[Jeffers and Reinders, 2013] Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann, San Diego, CA.

[Keeler, 1897] Keeler, J. E. (1897). The importance of astrophysical research and the relation of astrophysics to other physical sciences. *Science*, pages 745–755.

[Kirk, 2007] Kirk, D. (2007). NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104.

[Knoll et al., 2013] Knoll, A., Wald, I., Navrátil, P. A., Papka, M. E., and Gaither, K. P. (2013). Ray tracing and volume rendering large molecular data on multi-core and many-core architectures. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, UltraVis '13, pages 5:1–5:8, New York, NY, USA. ACM.

[Krüger and Westermann, 2003] Krüger, J. and Westermann, R. (2003). Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 908–916. ACM.

[Kulikov et al., 2015] Kulikov, I., Chernykh, I., Snytnikov, A., Glinskiy, B., and Tutukov, A. (2015). "astrophi: A code for complex simulation of the dynamics of astrophysical objects using hybrid supercomputers". *Computer Physics Communications*, 186:71 – 80.

[Leang et al., 2014] Leang, S. S., Rendell, A. P., and Gordon, M. S. (2014). Quantum chemical calculations using accelerators: Migrating matrix operations to the nvidia kepler gpu and the intel xeon phi. *Journal of Chemical Theory and Computation*, 10(3):908–912.

[Ligowski and Rudnicki, 2009] Ligowski, L. and Rudnicki, W. (2009). An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8.

[Lindholm et al., 2001] Lindholm, E., Kilgard, M. J., and Moreton, H. (2001). A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM.

[Liu and Schmidt, 2014] Liu, Y. and Schmidt, B. (2014). SWAPHI: Smith-waterman protein database search on Xeon Phi coprocessors. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 184–185. IEEE.

[LLNL, 2015] LLNL (2015). MPIBarrier. `https://computing.llnl.gov/tutorials/mpi/man/MPI_Barrier.txt`. Accessed: 2015-07-21.

[Magro et al., 2002] Magro, W., Petersen, P., and Shah, S. (2002). Hyper-Threading Technology: Impact on Compute-Intensive Workloads. *Intel Technology Journal*, 6(1).

[Mark et al., 2003] Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J. (2003). Cg: A system for programming graphics hardware in a C-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM.

[Michalakes and Vachharajani, 2008] Michalakes, J. and Vachharajani, M. (2008). GPU acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548.

[Mirizzi, 2013] Mirizzi, A. (2013). Multi-azimuthal-angle effects in self-induced supernova neutrino flavor conversions without axial symmetry. *Phys. Rev. D*, 88(7):073004.

[Moore, 2006] Moore, G. E. (2006). Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Newsletter*, 3(20):33–35.

[Moore et al., 1975] Moore, G. E. et al. (1975). Progress in digital integrated electronics. *IEDM Tech. Digest*, 11.

[Moreland and Angel, 2003] Moreland, K. and Angel, E. (2003). The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119. Eurographics Association.

[MPI, 2015] MPI, O. (2015). Open MPI. `http://www.open-mpi.org/`. Accessed: 2015-07-21.

[MPICH, 2015] MPICH (2015). MPICH. `https://www.mpich.org/`. Accessed: 2015-07-21.

[MVAPICH, 2015] MVAPICH (2015). MVAPICH. `http://mvapich.cse.ohio-state.edu/overview/`. Accessed: 2015-07-21.

[NetCDF, 2015] NetCDF (2015). NetCDF. `http://www.unidata.ucar.edu/software/netcdf/`. Accessed: 2015-12-03.

[Nickolls et al., 2008] Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2):40–53.

[Noormofidi et al., 2015] Noormofidi, V., Atlas, S. R., and Duan, H. (2015). Performance Analysis of an Astrophysical Simulation Code on the Intel Xeon Phi Architecture. *arXiv preprint arXiv:1510.02163*.

[Nukada et al., 2008] Nukada, A., Ogata, Y., Endo, T., and Matsuoka, S. (2008). Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11.

[NVIDIA, 2012a] NVIDIA (2012a). CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide. Accessed: 2015-08-19.

[NVIDIA, 2012b] NVIDIA (2012b). Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. Accessed: 2014-07-24.

[Nyland et al., 2007] Nyland, L., Harris, M., and Prins, J. (2007). Fast n-body simulation with cuda. *GPU gems*, 3(1):677–696.

[Olive et al., 2014] Olive, K. A. et al. (2014). Review of particle physics. *Chin. Phys.*, C38:090001.

[OpenGL, 2015a] OpenGL (2015a). Fixed Function Pipeline. `https://www.opengl.org/wiki/Fixed_Function_Pipeline`. Accessed: 2015-03-03.

[OpenGL, 2015b] OpenGL (2015b). Fragment Shader. `https://www.opengl.org/wiki/Fragment_Shader`. Accessed: 2015-03-03.

[OpenGL, 2015c] OpenGL (2015c). Vertex Shader. `https://www.opengl.org/wiki/Vertex_Shader`. Accessed: 2015-03-03.

[OpenMP4.0, 2015] OpenMP4.0 (2015). OpenMP4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`. Accessed: 2015-08-21.

[Pagliaroli et al., 2009] Pagliaroli, G., Vissani, F., Costantini, M., and Ianni, A. (2009). ”improved analysis of {SN1987A} antineutrino events ”. *Astroparticle Physics*, 31(3):163 – 176.

[Park et al., 2013] Park, J., Tang, P. T. P., Smelyanskiy, M., Kim, D., and Benson, T. (2013). Efficient backprojection-based synthetic aperture radar computation with many-core processors. *Scientific Programming*, 21(3-4):165–179.

[Peleg et al., 1997] Peleg, A., Wilkie, S., and Weiser, U. (1997). Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38.

[Pennycook et al., 2013] Pennycook, S. J., Hughes, C. J., Smelyanskiy, M., and Jarvis, S. A. (2013). Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1085–1097. IEEE.

[Phillips et al., 2008] Phillips, J. C., Stone, J. E., and Schulten, K. (2008). Adapting a message-driven parallel application to GPU-accelerated clusters. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–9. IEEE.

[Purcell et al., 2002] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 703–712. ACM.

[Raffelt et al., 2013] Raffelt, G., Sarikas, S., and de Sousa Seixas, D. (2013). Azimuth-angle flavor instability of supernova neutrino fluxes. *Phys. Rev. Lett. 111,*, 091101:091101.

[Ryoo et al., 2008] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82.

[Saule et al., 2014] Saule, E., Kaya, K., and Çatalyürek, Ü. V. (2014). Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *Parallel Processing and Applied Mathematics*, pages 559–570. Springer.

[Schawinski et al., 2008] Schawinski, K., Justham, S., Wolf, C., Podsiadlowski, P., Sullivan, M., Steenbrugge, K. C., Bell, T., Röser, H.-J., Walker, E. S., Astier, P., Balam, D., Balland, C., Carlberg, R., Conley, A., Fouchez, D., Guy, J., Hardin, D., Hook, I., Howell, D. A., Pain, R., Perrett, K., Pritchet, C., Regnault, N., and Yi, S. K. (2008). Supernova Shock Breakout from a Red Supergiant. *Science*, 321:223–226.

[Seiler et al., 2008] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., et al. (2008). Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics (TOG)*, 27(3):18.

[SlideShare, 2015] SlideShare (2015). NUMA. `http://image.slidesharecdn.com/numaistep2014-140603143243-phpapp02/95/notes-on-numa-architecture-3-638.jpg?cb=1401806479`. Accessed: 2015-09-03.

[Spurzem et al., 2009] Spurzem, R., Berczik, P., Marcus, G., Kugel, A., Lienhart, G., Berentzen, I., Männer, R., Klessen, R., and Banerjee, R. (2009). Accelerating astrophysical particle simulations with programmable hardware (FPGA and GPU). *Computer Science-Research and Development*, 23(3-4):231–239.

[Stone et al., 2010] Stone, J. E., Gohara, D., and Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66.

[Sugita et al., 2003] Sugita, K., Naemura, T., and Harashima, H. (2003). Performance evaluation of programmable graphics hardware for image filtering and stereo matching. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 176–183. ACM.

[Swanson, 1995] Swanson, R. W. (1995). Processing commands and data in a common pipeline path in a high-speed computer graphics system. US Patent 5,421,028.

[TACC, 2015] TACC (2015). Stampede User Guide. https://portal.tacc.utexas.edu/user-guides/stampede. Accessed: 2015-09-03.

[Teodoro et al., 2014] Teodoro, G., Kurc, T., Kong, J., Cooper, L., and Saltz, J. (2014). Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1063–1072. IEEE.

[Thakkur and Huff, 1999] Thakkur, S. and Huff, T. (1999). The Internet Streaming SIMD Extensions. *Computer*, 32(12):26–34.

[Thompson et al., 2002] Thompson, C. J., Hahn, S., and Oskin, M. (2002). Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317. IEEE Computer Society Press.

[Thompson et al., 2001] Thompson, T. A., Burrows, A., and Meyer, B. S. (2001). The physics of proto-neutron star winds: implications for r-process nucleosynthesis. *The Astrophysical Journal*, 562(2):887.

[Vineet and Narayanan, 2008] Vineet, V. and Narayanan, P. (2008). CUDA cuts: Fast graph cuts on the GPU. *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–8.

[Wende and Steinke, 2013] Wende, F. and Steinke, T. (2013). Swendsen-Wang Multi-cluster Algorithm for the 2D/3D Ising Model on Xeon Phi and GPU. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 83:1–83:12, New York, NY, USA. ACM.

[Wikimedia, 2015] Wikimedia (2015). Pipeline. `https://upload.wikimedia.org/wikipedia/commons/6/67/5_Stage_Pipeline.svg`. Accessed: 2015-08-30.

[Wikipedia, 2014] Wikipedia (2014). Multi-core processor. `https://en.wikipedia.org/wiki/Multi-core_processor`. Accessed: 2014-07-23.

[Wikipedia, 2015a] Wikipedia (2015a). ASCI Red. `https://en.wikipedia.org/wiki/ASCI_Red`. Accessed: 2015-07-02.

[Wikipedia, 2015b] Wikipedia (2015b). CDC 6600. `https://en.wikipedia.org/wiki/CDC_6600`. Accessed: 2015-07-02.

[Wikipedia, 2015c] Wikipedia (2015c). CDC 7600. `https://en.wikipedia.org/wiki/CDC_7600`. Accessed: 2015-07-02.

[Wikipedia, 2015d] Wikipedia (2015d). CM-5. `https://en.wikipedia.org/wiki/Connection_Machine`. Accessed: 2015-07-02.

[Wikipedia, 2015e] Wikipedia (2015e). Cray-1. `https://en.wikipedia.org/wiki/Cray-1`. Accessed: 2015-07-02.

[Wikipedia, 2015f] Wikipedia (2015f). Cray X-MP. `https://en.wikipedia.org/wiki/Cray_X-MP`. Accessed: 2015-07-02.

[Wikipedia, 2015g] Wikipedia (2015g). Exascale computing. `https://en.wikipedia.org/wiki/Exascale_computing`. Accessed: 2015-08-04.

[Wikipedia, 2015h] Wikipedia (2015h). IBM Roadrunner. `https://en.wikipedia.org/wiki/IBM_Roadrunner`. Accessed: 2015-07-02.

[Wikipedia, 2015i] Wikipedia (2015i). Moore's law. https://en.wikipedia.org/wiki/Moore%27s_law. Accessed: 2015-09-03.

[Wikipedia, 2015j] Wikipedia (2015j). nCube. https://en.wikipedia.org/wiki/NCUBE. Accessed: 2015-08-30.

[Wikipedia, 2015k] Wikipedia (2015k). Top500. https://en.wikipedia.org/wiki/TOP500. Accessed: 2015-09-03.

[Woosley and Janka, 2005] Woosley, S. and Janka, T. (2005). The physics of core-collapse supernovae. *Nat. Phys.*, 1:147.

[www.gamersnexus.net, 2015] www.gamersnexus.net (2015). RAM Performance Benchmark. http://www.gamersnexus.net/guides/1349-ram-how-dual-channel-works-vs-single-channel. Accessed: 2015-09-16.