2024

# Improving the Performance of HiRep Lattice Simulations Software by Exploiting the CPU Hardware Architecture Details and Algorithm Characteristics

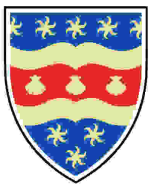
Rahman, Md Shidur

# COPYRIGHT STATEMENT



[1] E.g. in the example of third-party copyright materials reused in the thesis.

[2] In accordance with best practice principles such as *Marking/Creators/Marking third party content* (2013). Available from: https://wiki.creativecommons.org/wiki/Marking/Creators/Marking_third_party_content [accessed 28th February 2022].



UNIVERSITY OF
PLYMOUTH

# Improving the Performance of HiRep Lattice Simulations Software by Exploiting the CPU Hardware Architecture Details and Algorithm Characteristics

by

Md Shidur Rahman

A thesis submitted to the University of Plymouth
in partial fulfilment for the degree of

**DOCTOR OF PHILOSOPHY**

School of Engineering, Computing and Mathematics

August 2024

# Acknowledgements

FIRST AND FOREMOST, I would like to express my profound gratitude to Almighty Allah for granting me the health, strength, and perseverance to embark on and complete this arduous journey of my PhD. His countless blessings have been instrumental in this achievement, and without His guidance, this work would not have been possible.

I am deeply indebted to my supervisory team for their unwavering guidance and encouragement. I extend my heartfelt thanks to my Director of Studies, **Dr. Vasilios Kelefouras**, for his prompt and supportive advice, which continually inspired me to pursue this research with passion and determination. I am equally grateful to my second supervisor, **Professor Antonio Rago**, for his patience, support, and invaluable assistance throughout the course of my study. Special thanks go to my collaborator, **Professor Claudio Pica** from the University of Southern Denmark, for his insightful and valuable advice during my research.

I owe a profound debt of gratitude to my beloved wife, Shanag Parvin, and my son, Habib Rahman, for their unwavering love, encouragement, and support. Their optimism and belief in me have been a constant source of motivation.

I also wish to extend my sincere thanks to my family—my mother, Parula Begum, my father, Md Sherazul Islam, as well as my brother and sister—for their unconditional support and encouragement.

Finally, I acknowledge the use of the Foseres and HIPPO High Performance Computing Facility and the associated support services at the University of Plymouth and the University of Southern Denmark. I am also thankful for the financial support provided by the University of Plymouth Studentship (URS).

# Author's Declaration

AT NO TIME during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Doctoral College Quality Sub-Committee. Work submitted for this research degree at the University of Plymouth has not formed part of any other degree either at the University of Plymouth or at another establishment. This study was financed from the University of Plymouth Studentship (URS). Relevant scientific seminars and conferences were attended at which work was often presented. Two papers are currently under review for publication in refereed journals.

Word count for the main body of this thesis: **19751**

**Signed:** 

**Date:** 7/11/2024

### Publications:


[1] Rahman, M.S., Kelefouras, V., Pica, C. and Rago, A.: Accelerating HiRep Lattice Simulations on CPU-based Computer Clusters. Proceedings of XXXV IUPAP Conference on Computational Physics (CCP2024). Springers Proceedings in Physics.

[2] Rahman, M.S., Kelefouras, V., Pica, C. and Rago, A. Optimizing HiRep lattice simulations on CPU-based computer clusters. *Computer Physics Communications* (Under review).

# Abstract


## Improving the Performance of HiRep Lattice Simulations Software by Exploiting the CPU Hardware Architecture Details and Algorithm Characteristics

**Md Shidur Rahman**

IN THE SCIENTIFIC exploration of Quantum Chromodynamics (QCD)—the theory governing the strong interaction among quarks and gluons—large-scale numerical simulations are performed using the framework of lattice gauge theories. Lattice Gauge Theory (LGT) simulations involve the formulation of gauge field theories on a space-time lattice.

HiRep is a simulation suite designed for running lattice simulations, leveraging high-performance computing platforms. HiRep is designed to be flexible enough to study a wide range of strongly interacting systems, particularly those pertinent to novel physics investigations at CERN's Large Hadron Collider (LHC). However, improving the execution time of HiRep is a challenging and non-trivial task. Even marginal improvements in HiRep's execution time can have a significant impact on paving the way to new discoveries in the field of particle physics.

However, a detailed study, analysis, and profiling of the HiRep application revealed that the implementation of the Dirac operator is one of the most computationally intensive routines, serving as the main performance bottleneck. Consequently, this routine was optimized for CPU-based distributed-memory hardware platforms. The main performance inefficiencies include communication overhead due to extensive data exchanges between MPI processes, workload imbalances in OpenMP regions, inefficient data reuse of lattice sites, and ineffective auto-vectorization.

To this end, both algorithmic and hardware-dependent optimization strategies are employed. These strategies include efficient hybrid parallelization (using both MPI and OpenMP parallel programming frameworks), optimizing OpenMP parallelism through loop collapsing, memory access patterns optimization, and vectorization (using both AVX2 and Clang compiler's vector intrinsics).

Based on experimental results obtained from two distinct High-Performance Computing (HPC) platforms, the proposed optimizations boost the performance of HiRep, achieving an overall speedup of up to $\times 1.80$ compared to the baseline MPI version.

# Contents

# List of Figures

# Chapter 1

# Introduction

***In this chapter**, the research background and aim are delineated, followed by the identification of the objectives. The methods for achieving these objectives are also highlighted. In addition, the contributions of this PhD research are demonstrated. An outline of the thesis structure is presented, offering a comprehensive overview of its contents.*

LATTICE SIMULATIONS are essential for probing strongly interacting theories, particularly those fundamental to the Standard Model of elementary particles. These simulations demand significant computational resources, necessitating the use of costly and energy-intensive High-Performance Computing (HPC) platforms. They often run for months to obtain predictions on physical phenomena relevant to elementary particles research and generate datasets of considerable memory size (Wilson 1974*a*, 1980; Creutz 1979). Such a parallel application is `HiRep`[1].

`HiRep` is designed to be flexible enough to study a wide range of strongly interacting systems, particularly those pertinent to novel physics investigations at CERN's Large Hadron Collider (LHC). HiRep has garnered global recognition, currently used by hundreds of physicists worldwide. Enhancing the efficiency of HiRep is a tremendous challenge with the potential for groundbreaking discoveries in particle physics. Even marginal improvements in HiRep's execution time can have a significant impact.

The aim of this PhD project is to improve the performance of `HiRep` lattice simulations software by exploiting the CPU hardware architecture details and algorithm characteristics. This aim is pursued through two main objectives: first, addressing and analyzing the performance

[1] https://github.com/claudiopica/HiRep

inefficiencies of HiRep; and second, delivering an optimized version of HiRep for CPU-based clusters. HiRep is open source and can be found in HiRep GitHub Repository[2].

To achieve these objectives, various optimizations are implemented. These include hybrid MPI-OpenMP parallelization, optimizing OpenMP parallelism through loop collapsing, and improving data access patterns. Additionally, hardware-dependent compiler optimizations are employed, such as manually vectorizing the Matrix Vector Multiplication (MVM) operation using AVX2 and the Clang compiler's vector intrinsics.

Based on our experimental results which include two hardware clusters, Foseres and HIPPO, the proposed optimizations boost HiRep performance resulting in an up to $\times 1.80$ overall speedup compared to the baseline MPI-only version.

The major contributions of this PhD research are as follows:

1. An in-depth performance analysis of HiRep, entailing the identification of performance inefficiencies (Chapter 4).

2. An optimization methodology for HiRep that comprises various optimization strategies (Chapter 5).

3. An experimental procedure demonstrating that the proposed approach yields substantial performance improvements on two distinct computer clusters (Chapter 6).

The rest of this thesis is structured as follows:

**Chapter 2: Background Knowledge - High Performance Computing** A brief introduction in HPC is provided. These are parallel computer hardware architectures, parallel programming frameworks such as MPI, OpenMP, hybrid programming, and Vectorization. Moreover, the HPC performance profiling and analysis tools used in this research are explained.

**Chapter 3: Background Knowledge - HiRep and Dirac Operator** Some of the general concepts and calculations involved in lattice regularization, HiRep and Dirac operator are presented. Then, the details of the HiRep coding conventions, data structures, problem paralleliza-

[2] https://github.com/claudiopica/HiRep

tion, and data movement are discussed.

**Chapter 4: HiRep Performance Inefficiencies** The inefficiencies of `Dphi` routine, which implements the Dirac operator in `HiRep`, are identified. The possible solutions to overcome these inefficiencies are also mentioned in brief.

**Chapter 5: Optimization Methodology** The code optimization methodology performed on HiRep is discussed in detail. This methodology entails Hybrid MPI-OpenMP Parallelization, Optimizing OpenMP Parallelism Through Loop Collapsing, Data Access Patterns Optimization (MPI+Path-blocking), and Vectorization.

**Chapter 6: Experimental Results** This chapter evaluated the experimented results. The experimental setup and performance metrics are described. Then the results are visualized and evaluated the performance.

**Chapter 7: Related Work** The related works are listed in this chapter and compared them with `HiRep` implementation.

**Chapter 8: Summary and discussion** In this chapter, the research work is summarised and discussed the implications of this study.

**Chapter 9: Conclusions and Further Work** Finally, in this chapter, the research work is concluded and pointed out its possible improvement in the future.

# Chapter 2

# Background Knowledge: High-Performance Computing

***This chapter*** *provides a brief overview of the fundamentals of High-Performance Computing (HPC). It begins by defining HPC and outlining the current challenges faced in the field. Subsequently, it delves into the discussion of parallel computer hardware architectures, covering aspects such as memory hierarchy, registers, cache memories, DRAM memories, vectorization engines, as well as shared and distributed memory architectures. Furthermore, the chapter explores various HPC programming models, including MPI (Message Passing Interface), OpenMP (Open Multi-Processing), and vectorization techniques. Additionally, it discusses the importance of HPC application profiling and performance analysis tools in optimizing code for efficient execution on HPC systems.*

HIGH-PERFORMANCE computing (HPC) encompasses a broad spectrum of technologies, methodologies, and applications aimed at achieving the utmost computing capability available at any given time and with current technology (Sterling et al. 2017). It serves not only specific research needs in fields like physics, chemistry, and biology, but also addresses broader societal demands (Navaux et al. 2023).

HPC systems typically consist of compute and storage resources interconnected by high-speed networks. These systems may comprise thousands of compute nodes, utilized by users to execute complex software applications or "jobs". Access to these resources is managed by a job scheduler, which assigns compute nodes to jobs in a queue based on user-defined criteria and resource availability. The scheduler also monitors job progress and manages contention for shared resources (Silva et al. 2024).

In the context of High-Performance Computing (HPC), a node is a basic unit of a supercomputer or a cluster. It is essentially an independent computing unit that contains its own processors, memory, and network connections, and it runs its own instance of an operating system. Nodes can work independently or as part of a larger distributed system, communicating with each other to perform complex computations.

A socket is a physical slot on the motherboard that holds a CPU (Central Processing Unit) (Intel Corporation 2024*f*). Each socket can host one multi-core processor. Sockets provide the physical and electrical interface through which the CPU communicates with the system memory and other components. High-performance nodes often have multiple sockets to increase processing power.

A core is an individual processing unit within a CPU. Modern CPUs are multi-core, meaning they contain multiple cores within a single processor (Reinders et al. 2017). Each core can execute multiple threads independently, allowing for parallel processing within a single CPU. This improves the performance and efficiency of computational tasks.

Random Access Memory (RAM) is the primary memory used by a node to store data that is actively being used or processed by the CPU (Reinders et al. 2017). RAM allows for fast read and write access to data, which is crucial for high-performance computations. The amount of RAM in a node determines the size of the datasets that can be processed efficiently.

The network interface is a hardware component that allows a node to connect to other nodes within a cluster or to external networks (Asanović and Patterson 2014). High-speed network interfaces, such as InfiniBand or Ethernet, enable fast data transfer between nodes, which is essential for distributed computing and parallel processing tasks.

Porting a serial software application to a High-Performance Computing (HPC) system is a challenging task. There are no tools available that can efficiently and automatically generate optimized parallel code. As a result, the programmer must manually parallelize the code, which is a complex process that requires careful consideration of various factors such as hardware architecture, parallel programming models, and the utilization of tools and profilers. These aspects are briefly explained below.

## 2.1 Parallel Computer Hardware Architectures

### 2.1.1 Memory Hierarchy

Memory hierarchy is a performance-critical component in modern computer systems. Given the disparity in speed between the main memory and the CPU, engineers have introduced faster memory layers between them to reduce access time (Hager and Wellein 2010). Memory hierarchy, as illustrated in Figure 2.1, is a fundamental concept in computing, organizing various types of storage based on their speed of access. At the summit of this hierarchy are CPU registers, offering the fastest read and write speeds. Directly below registers lie cache memory, followed by conventional DRAM memory, and further down the hierarchy, disk storage including SSDs, optical, and magnetic disk drives. The significance of memory hierarchy lies in its role in bridging the performance gap between the processor and memory.

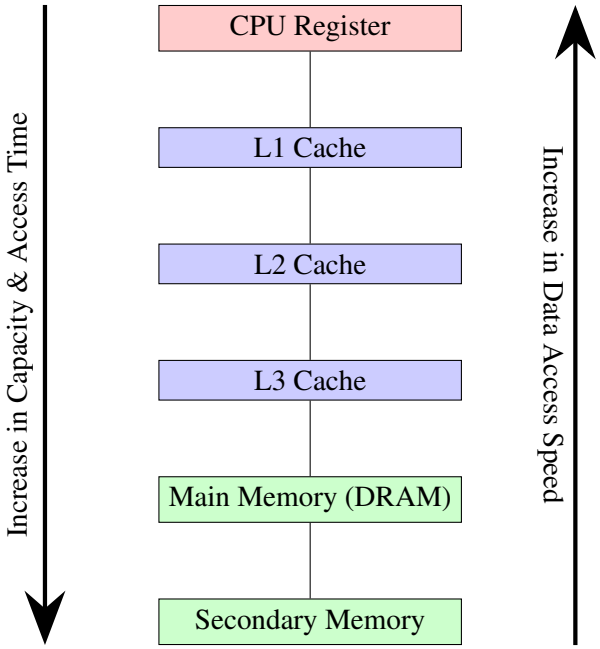



*Figure 2.1:* Memory Hierarchy

To bridge this performance gap, programmers aim to maximize data reuse within the valuable cache memories, thereby reducing the number of main memory accesses and improving overall computational efficiency.

Registers, integral components of the CPU, constitute small yet high-speed memory units pri-

marily designated to retain frequently accessed data and instructions. Characterized by their exceptionally fast access time, registers possess a minimal storage capacity, nowadays spanning from 16 to 512 bits (AVX-512). Divided into L1, L2, and L3 levels, cache memory operates based on speed and cache size, as depicted in Figure 2.1.

Each core within a CPU used in High-Performance Computing (HPC) systems is equipped with its own L1 and L2 caches. The L1 cache is divided into two distinct sections: the instruction cache (L1I) and the data cache (L1D), which are typically 32KB each, although this size can vary depending on the processor. Often referred to as the Primary Cache, this component is the fastest form of memory in the computer system, designed to prioritize access to data likely to be needed by the CPU during task execution (Phillips 2023). If the processor fails to locate the required data within the L1 cache, it proceeds to search for them within the L2 and L3 cache levels (Russ Ware 2023).

The L2 cache, typically 256KB in size though varying with different processors, is a unified cache that stores both instructions and data, operating as an exclusive cache. This means it does not necessarily contain all the data present in the L1 cache. However, due to its larger size, it typically holds more data and instructions, minimizing the need for evictions (Frank Denneman 2016).

Data fetched from memory populates all cache levels on its path to the core $(L3 \rightarrow L2 \rightarrow L1)$. The L3 cache serves as an inclusive cache designed to function as shared memory, ensuring it holds all data present in the L2 or L1 caches. Both the L1 and L2 caches are private to each core, storing data that is read, written, or modified. In contrast, the L3 cache is shared among multiple cores (Frank Denneman 2016).

Modern CPUs can feature substantial L3 caches, with high-end consumer CPUs boasting capacities of up to 32MB, while advanced server CPUs may even exceed 128MB, such as those found in AMD's Ryzen 7 5800X3D series (Phillips 2023; Russ Ware 2023).

In Non-Uniform Memory Access (NUMA) systems, memory access time varies depending on the memory location relative to the CPU. Each CPU has its own local memory, and accessing this memory is faster than accessing memory located on another CPU. This architecture aims to

improve scalability and performance in multi-CPU systems by ensuring that each CPU core can efficiently access its local memory while also providing mechanisms to access remote memory when necessary. NUMA systems benefit from the hierarchical cache structure described above, as the L1, L2, and L3 caches help mitigate the latency associated with remote memory access by caching frequently used data locally (NUMA).

When the processor does not find a data item it needs in the cache, a cache miss occurs (Hennessy and Patterson 2011). This situation leads to delays in execution as the program or application must fetch the data from other cache levels or the main memory. For example, in the event of an L1 cache miss, L2 cache is accessed. On the other hand, a cache hit occurs when the data requested for processing by a component or application is found in the cache. This results in faster data delivery to the processor, as the cache already holds the requested data (Margaret Rouse 2013; Hennessy and Patterson 2011).

### 2.1.2 Vectorization Engine

A vectorization engine in modern CPUs is a specialized hardware unit designed to efficiently process vector operations. These engines, also known as SIMD (Single Instruction, Multiple Data) units, enable the CPU to perform the same operation on multiple data points simultaneously, greatly enhancing performance for certain types of computations.

An illustrative example of an SIMD-enabled operation is depicted in Figure 2.2. The comparison illustrated in Figure 2.2 underscores the distinction between scalar and vectorized programming approaches. In scalar code, each instruction manages the loading and processing of a single array element. Conversely, the vectorized code utilizes the AVX vaddps instruction, enabling the simultaneous loading and addition of eight single-precision array elements. This capability allows a single instruction to process multiple array elements concurrently, resulting in improved computational efficiency.

These advancements have led to wider SIMD registers, which aim to accommodate a greater number of smaller cores per chip, thereby increasing computational throughput and efficiency (Polychroniou and Ross 2019).

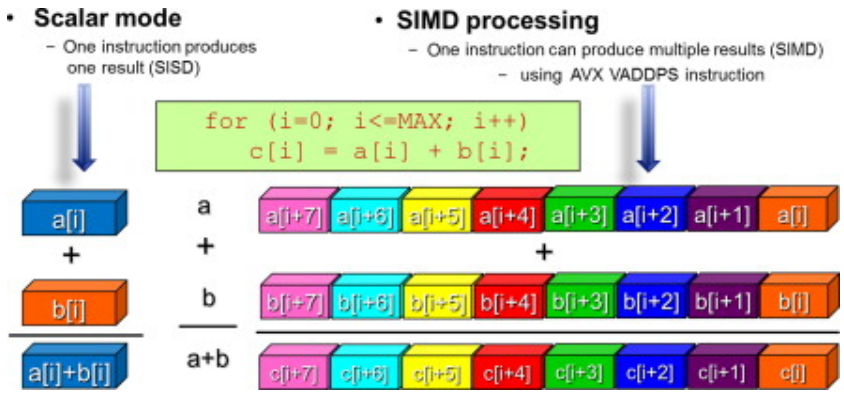



*Figure 2.2:* An illustration of Intel's AVX vector operation (SIMD) is provided in the reference by Jeffers and Reinders (2015).

Modern CPUs support vector engines which are designed to efficiently execute vectorized instructions, which can result in significant performance gains for tasks that involve large amounts of data parallelism, such as numerical computations, multimedia processing, and scientific simulations.

To utilize these hardware vector engines, modern compilers support auto-vectorization that automatically transforms scalar code into vectorized code. This process involves identifying opportunities for parallelism within the code and generating instructions that can be executed in parallel on vector units of the processor.

Overall, vectorization engines play a crucial role in accelerating computational tasks by exploiting data parallelism and leveraging the capabilities of modern processor architectures.

Different CPU vendors employ different vector instructions, vector lengths and technologies. For example, ARM supports NEON (ARM Developer 2024), while x64 architectures support SSE/AVX/AVX2/AVX-512 (Intel Corporation 2024*b*),

The x86 architecture typically supports 16 vector registers, whereas the x64 architecture extends this capability to 32 registers. An increased number of registers allows for greater data reuse at the register level, thereby reducing the frequency of memory operations. This reduction in memory operations can enhance overall performance by minimizing the need to access slower memory hierarchies and optimizing the utilization of the faster register storage.

### 2.1.3 Shared Memory Architectures

Shared memory architectures enable multiple processors or cores to access a single, unified address space, facilitating direct communication and data sharing. They represent a foundational paradigm in parallel computing (Diaz et al. 2012).

Shared memory architectures encompass Symmetric Multiprocessor (SMP) systems, where each processor enjoys equal access to memory resources - Unified Memory Access (UMA), and Non-Uniform Memory Access (NUMA) systems, characterized by varying memory access times depending on processor proximity (Hennessy and Patterson 2011). In a NUMA architecture, the L3 cache is shared among all cores within a CPU socket. In a multi-socket system, memory is divided between sockets, and accessing L3 cache local to the socket is faster due to lower latency, but accessing L3 cache belongs to the other socket experiences higher latency, and NUMA violation occurs.

Programming models such as OpenMP (OMP ARB 2018), Pthreads (PHP Documentation Team 2024), and Cilk (Wikipedia contributors 2023)/Cilk Plus (Cilk Plus Team 2021) empower developers to craft parallel programs that harness shared memory parallelism. These models furnish constructs for creating and managing parallel threads or tasks (Chandra et al. 2001).

Performance optimization techniques for shared memory architectures encompass strategies such as data locality optimization, thread scheduling, and load balancing. These techniques are designed to mitigate overheads and enhance parallelism, thereby enhancing programs' performance.

Shared memory architectures face various challenges, including scalability issues, contention for shared resources, and synchronization overheads. Research endeavors are directed towards mitigating these challenges to enhance performance and scalability (Nagarajan et al. 2020). Despite these challenges, shared cache (L3) plays a crucial role in applications' performance improvement. Modern processors, including those evaluated in this PhD thesis, feature L3 shared memory. This shared memory is leveraged using a variety of techniques, which are elaborated upon in Chapters 5 and 6. These techniques aim to optimize the utilization of L3

cache, improving the overall efficiency and performance of the processors under study.

### 2.1.4 Distributed Memory Architectures

Distributed memory architectures find application across diverse domains including scientific computing, big data analytics, and distributed simulations. They furnish scalable computing resources for tackling large-scale computational challenges mandating extensive parallelism and robust data processing capabilities. Distributed memory architectures form a foundational aspect of parallel computing, characterized by independent processing units equipped with individual local memory units. These units interact via explicit message passing mechanisms (Gropp et al. 1999). Examples of distributed memory systems encompass clusters, grid computing infrastructures, and massively parallel supercomputers.

Programming models tailored for distributed memory architectures predominantly employ Message Passing Interfaces (MPI) as the primary mode of communication and synchronization between processes (Snir et al. 1998). MPI provides a standardized suite of functions for data exchange and coordination across distributed systems, empowering developers to write portable and scalable parallel applications.

Challenges inherent in distributed memory architectures encompass latency and bandwidth constraints in communication networks, load imbalances, and scalability concerns (Dongarra et al. 2003). As the scale of distributed systems amplifies, managing communication and synchronizing computation escalates in complexity, necessitating sophisticated algorithms and system architectures.

In this thesis, clusters comprising several nodes are employed. Various techniques aimed at minimizing communication overhead are discussed in detail in Chapters 5 and 6.

## 2.2 Parallel Programming Models

### 2.2.1 MPI (Message-Passing Interface)

The MPI is a crucial application programmer interface (API) for parallel computing, introduced in 1992, which has significantly influenced scientific parallel computing. MPI facilitates

parallel computation across high performance computers by utilizing a proprietary, scalable, high-bandwidth, low-latency interconnect, enabling seamless communication among processors and scaling to thousands of processors while minimizing overhead communication time. This is particularly important in distributed memory environments or processes lacking shared memory address spaces. As a primary choice for harnessing parallelism, MPI's development has kept pace with evolving HPC system technologies, addressing challenges such as balancing communication and computation across nodes, efficiently using asynchronous communication, and ensuring robust fault tolerance mechanisms (Gropp 2024; Navaux et al. 2023).

A plethora of MPI implementations are currently available, each focusing on distinct facets of high-performance computing. At the core of MPI lies a standardized library facilitating communication between processes in distributed processor environments. MPICH, Open MPI, and Intel MPI Library are three prominent implementations of the MPI standard, each with unique strengths. MPICH is a freely accessible, portable MPI implementation encompassing MPI-1 to MPI-3. Known for its efficiency and adaptability, MPICH supports a wide range of platforms from standard clusters to specialized HPC systems, and its modular framework allows for seamless extension and customization. It is open-source and extensively tested on multiple platforms, including Linux, Mac OS/X, Solaris, and Windows. Open MPI, also open-source, is widely used for parallel applications on distributed memory systems, enabling efficient communication between multiple processes and supporting various communication protocols. It is highly portable and valuable in fields requiring HPC, such as scientific research and engineering. Open MPI scales well to large numbers of processors and supports diverse hardware and operating systems. The Intel MPI Library, based on the MPICH specification, is designed for HPC clusters using Intel and compatible processors. It enables the creation of high-performance applications optimized for various cluster interconnects, featuring automatic tuning mechanisms for latency, bandwidth, and scalability. It supports the OpenFabrics Interface (OFI) and ensures peak performance despite interconnect changes while prioritizing thread safety, start scalability, and cloud compatibility.

Here is an example of MPI code:

```
1  int main(int argc, char** argv) {
2      // Initialize the MPI environment
3      MPI_Init(&argc, &argv);
4
5      int world_size, world_rank;
6      // Get the number of processes
7      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
8      // Get the rank of the process
9      MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
10
11     // Print a message with the rank of the process
12     printf("Hello from process %d out of %d
           processes\n", world_rank, world_size);
13
14     // Finalize the MPI environment
15     MPI_Finalize();
16     return 0;
17 }
```

*Listing 2.1:* Example of MPI Code

This code snippet in Listing 2.1 initializes the MPI environment (line 3), determines the number of processes and the rank of each process (lines 7 and 9), prints a message from each process (line 12), and then finalizes the MPI environment (line 15). This simple example demonstrates basic MPI functionalities such as initialization, querying the environment, and finalization.

### 2.2.2 OpenMP (Open Multi-Processing)

With the proliferation of multi-core architectures, there has been a surge in the adoption of libraries designed to leverage parallelism in shared memory environments, such as OpenMP

parallel programming framework. OpenMP stands as a standard parallel programming API, compatible with C, C++, or FORTRAN, and comprises a suite of compiler directives with a user-friendly syntax, alongside library routines and environment variables that influence runtime behavior, simplifying thread management within the code (Chandra et al. 2008).

To exploit parallelism, developers integrate directives into their code to instruct the compiler on which parts of the application should execute in parallel. As the number of cores increases and the memory hierarchy in shared memory architectures becomes more complex, utilizing OpenMP poses challenges such as (i) determining the optimal number of threads for each parallel region; (ii) devising strategies for thread and data placement to mitigate cache contention and L3 cache misses; and (iii) effectively employing directives for heterogeneous computing.

Here is an example of loop-based OpenMP parallel code snippet:

```
1 int main(int argc, char** argv) {
2     // Parallelize the loop using OpenMP
3     #pragma omp parallel for reduction(+:sum)
4     for (int i = 0; i < size; i++) {
5         sum += array[i];
6     }
7     return 0;
8 }
```

*Listing 2.2:* Example of OpenMP Code

The OpenMP code snippet in Listing 2.2 demonstrates a simple approach to parallelizing a loop using OpenMP directives. By including the `#pragma omp parallel for` directive in line 3, the loop in line 4 is executed simultaneously by multiple threads, which allows the program to utilize multiple CPU cores effectively. The `reduction(+:sum)` clause ensures the correct aggregation of the `sum` variable across all threads, preventing race conditions. This example illustrates the fundamental concepts of OpenMP parallelism, which can enhance computational performance in applications.

Core and thread affinity in OpenMP refers to the practice of binding specific threads to specific processor cores. By setting affinity policies, data locality is enhanced, thread migration is minimized, and predictable performance can be achieved. In the context of the above OpenMP code snippet, applying core and thread affinity ensures that each thread remains bound to a designated core, thus maximizing the efficiency of parallel computation. This technique is particularly beneficial in applications requiring intensive computational resources and frequent data access, as it leverages the full potential of the underlying hardware architecture.

### 2.2.3 Hybrid Parallelization (MPI-OpenMP)

The hybrid parallel programming model combines MPI with the OpenMP programming model. Figure 2.3 illustrates this hybrid programming model. In the MPI model (left box), data is distributed among MPI processes, with each process handling its own data and exchanging data via MPI send-receive calls, which introduces communication overhead.

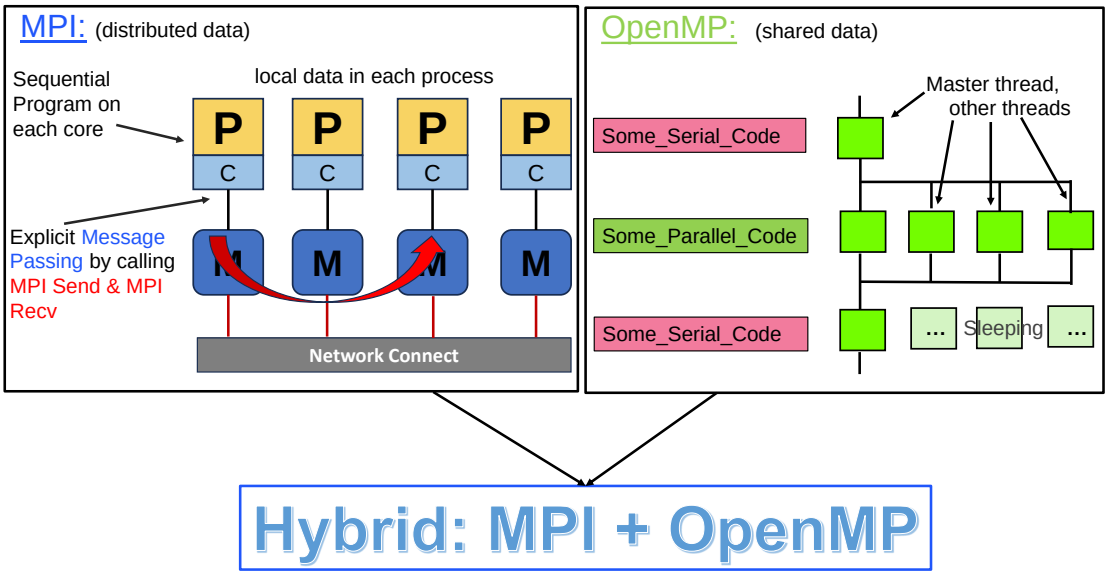



*Figure 2.3:* Hybridization: Combining MPI with OpenMP.

Conversely, in the OpenMP model (right box), the master thread spawns a team of threads that share data through the shared L3 cache memory, thereby avoiding communication overhead. By integrating these two models, the hybrid MPI-OpenMP approach can effectively reduce communication overhead. This hybrid model will be employed in this research work.

## 2.3 Vectorization

As already discussed in section 2.3, modern processors are equipped with vectorization capabilities, featuring specialized hardware components such as wide registers and processing units tailored for vector processing tasks. Software applications can be vectorized in four main ways:

- Automatically by the flags/options such as '-O3', '-ftree-slp-vectorize', and '-ftree-vectorize'. While this approach is straightforward to apply, it may not always yield the best performance.
- Utilizing OpenMP C/C++ pragmas.
- Using C/C++ intrinsics (assembly-coded functions). While this approach can achieve better performance compared to compiler-based vectorization, it might require a high programming effort and expertise.
- Directly writing assembly code. While potentially offering the highest performance, this method is highly intricate and challenging to utilize effectively.

The automatic application of vectorization by compilers, known as auto-vectorization, significantly enhances code performance. However, successful auto-vectorization is not guaranteed at all times. Note that all modern compilers are capable of autovectorizing blocks of code or loop kernels.

OpenMP provides a variety of directives, such as "reduction" and "simdlen," which enable efficient and straightforward vectorization, even for complex loop kernels. For example, the following code snippet in Listing 2.3 utilizes the `"omp simd"` directive to vectorize the loop operation:

```
1 #pragma omp simd
2 for (int n=0; n<N; ++n)
3    a[n] += b[n]; // Compiler may vectorize
```

*Listing 2.3:* Example of #pragma omp simd

Another effective method for vectorizing the code is to manually rewrite the serial code using SIMD intrinsics. The serial version of the vector addition in Listing 2.4 iterates through the elements of the input arrays “a” and “b” one by one, performing each addition operation sequentially and storing the results to array “c”. This approach is straightforward but does not leverage the capabilities of modern CPUs for parallel processing, which can limit its performance, especially for large datasets.

```
1 for (int i = 0; i < n; i++) {
2     c[i] = a[i] + b[i];
3 }
```

*Listing 2.4:* Example of a serial code that adds two vectors element-wise

On the other hand, the AVX2 version in Listing 2.5 utilizes AVX2 intrinsics to perform vectorized operations. Specifically, four double values or 256 bits of data are loaded simultaneously into AVX2 registers (lines 3 and 4) with a single load instruction, allowing for four parallel additions (line 7), thereby significantly accelerating the computation. Finally, the processed data is stored back with a single store instruction (line 10). The AVX2 code achieves faster performance due to two key factors: (1) it loads and stores 256 bits of data within a single instruction, as opposed to 32-bit, and (2) it processes 256 bits of data in parallel. This parallel processing capability enhances the efficiency of handling large vectors compared to serial code. However, optimal performance requires CPU support for vectorization and proper data alignment. Although the AVX2 code is more complex due to the use of intrinsics and handling edge cases, it can yield substantial performance improvements in suitable environments.

```
1  for (i = 0; i <= n - 4; i += 4) {
2      // Load 4 double elements of a[] and b[] into
          AVX2 registers
3      __m256 vec_a = _mm256_loadu_pd(&a[i]);
4      __m256 vec_b = _mm256_loadu_pd(&b[i]);
5
6      // Perform vector addition
7      __m256 vec_c = _mm256_add_pd(vec_a, vec_b);
8
9      // Store the result back to array c
10     _mm256_storeu_pd(&c[i], vec_c);
11 }
```

*Listing 2.5:* Example of an AVX2 vectorized code for parallel computation

Moreover, Clang compiler vector intrinsics are specialized functions provided by the Clang compiler that map directly to low-level SIMD hardware instructions. These intrinsics enable fine-grained control over vector operations and facilitate the explicit use of vector registers and SIMD execution units within the CPU. Clang vector intrinsics can target multiple architectures, not limited to Intel or AMD processors, thereby enhancing portability. In contrast, AVX2 intrinsics are specifically designed for Intel's 256-bit x86 vector instructions.

Clang supports, for instance, GCC vector types which can be created using the vector_size(N) attribute, where N specifies the number of bytes allocated for an object of this type. The size must be a multiple of the vector element type size. The following example (line 1) demonstrates the creation of a vector type `suNg_vector_V`, representing a vector of six 'double' elements, totaling 48 bytes.

```
1 typedef double suNg_vector_V __attribute__((vector_size(6 *
    sizeof(double))));
2 __builtin_shufflevector(vec1, vec2, index1, index2, ...)
```

The `__builtin_shufflevector` function in line 2 allows for generic vector permutation, shuffle, or swizzle operations. This function can be used within constant expressions. The first two arguments to `__builtin_shufflevector` are vectors with the same element type. The remaining arguments are integers specifying the indices of elements from the first two vectors to be extracted and returned in a new vector. These element indices are sequentially numbered, starting with the first vector and continuing into the second vector. In my PhD research, Matrix-Vector Multiplication (MVM) operations will be manually vectorized using AVX2 and Clang SIMD intrinsics. The performance of these manually vectorized implementations will be tested and compared.

## 2.4 Profiling and Performance Analysis Tools

The experiments incorporate a comprehensive set of profiling tools. Cachegrind, a cache profiling tool (Carnegie Mellon University 2009), will be employed to scrutinize memory access patterns, allowing us to quantify data reuse and instruction counts. Additionally, the Intel OneAPI HPC Toolkit (Intel Corporation 2024*g*) will be utilized, including VTune Profiler (Intel Corporation 2024*e*) for pinpointing performance bottlenecks, deciphering memory access behaviors, and scrutinizing threading intricacies such as locks and synchronizations between threads. The Intel Advisor tool (Intel Corporation 2024*c*) will be employed to identify code sections that could benefit from threading and vectorization, while Intel Inspector will be used to detect any threading-related issues and dynamic memory errors (Intel Corporation 2023).

**Cachegrind** which is one of Valgrind tools (Van Dung et al. 2014). It is a cache profiling tool that provides information for L1 (Level 1) and LL (Last Level: can either be L2 or L3) caches, and, specifically provides CPU cache hits and misses in general for the whole program, as well

as per line of code. Cachegrind is able to provide that kind of information by simulating the CPU caches of a computer either by auto-detecting their characteristics, or by letting the user specify them manually. Moreover, most profiling tools provide only global statistics, such as the number of cache hits and misses, which are of limited help to the programmer. Cachegrind provides cache information tied to particular parts of a program (Kaparelos 2014).

**Intel® VTune™** Profiler serves as a performance analysis tool designed for both serial and multithreaded applications. Leveraging VTune Profiler allows for the analysis of a chosen algorithm, enabling the identification of potential advantages for the application by leveraging available hardware resources.

VTune Profiler is used to accomplish the following tasks:

- Identify the most time-consuming (hot) functions in `HiRep`.
- Determine the best sections of code to optimize for both sequential and threaded performance.
- Analyze synchronization objects that impact application performance.
- Measure the performance impact of different synchronization methods, various numbers of threads, or different algorithms.
- Monitor thread activity.
- Detect hardware-related issues in the code, such as data sharing problems, cache misses, and other performance bottlenecks.

Intel® VTune™ Profiler deals with various analysises such as hotspots analysis, threading analysis, HPC Performance Characterization analysis, Microarchitecture Exploration, Memory Access analysis, Memory Consumption analysis, OpenMP* analysis, MPI analysis, and so on (Intel Corporation 2024*e*).

The **Intel® Advisor** is a tool designed to aid in the development and optimization of high-performing code for modern computer architectures. It supports a variety of programming

languages, including C, C++, Fortran, SYCL*, OpenMP*, OpenCL™, and Python* (Intel Corporation 2024*c*). Intel® Advisor facilitates the following tasks in `HiRep` application Roofline analysis for CPUs: collecting Roofline data, understanding a Roofline chart, interpreting the Roofline data.

By running the CPU/Memory Roofline Insights perspective, actual performance against hardware-imposed performance ceilings was visualized. This perspective assists in finding whether the bottleneck is DDR, L3 or L2 cache or compute capacity, and provides an ideal roadmap of potential optimization steps (Intel Corporation 2024*a*).

Moreover, the use of Roofline Analysis is explained for CPUs in optimizing code by considering both memory and compute aspects. This analysis provides a visual representation of application performance against hardware limitations, enabling developers to prioritize optimization efforts and identify critical areas for improvement. During the performance measurement and optimization phases of this project, Roofline Analysis is to be employed for assessing how closely the optimized code approaches the hardware's peak performance. This analysis aids in identifying the critical areas that require further optimization to enhance overall performance.

**Intel® Inspector** locates and debugs threading, memory, and persistent memory errors early in the design cycle to avoid costly errors later. Memory errors and nondeterministic threading errors are difficult to find without the right tool. Intel Inspector is designed to find these errors. It is a dynamic memory and threading error debugger for C, C++, and Fortran applications that run on Windows* and Linux* (Intel Corporation 2024*d*). Inspector helped me ensure that no race conditions (heap and stack races), deadlocks and cross-thread stack access occur during the runtime.

**LIKWID** is a tool suite for performance-oriented programmers and administrators. The term LIKWID stands for "Like I know what I do". LIKWID provides a set of helpful tools for analysis of systems and applications. For example, I used `likwid-topology` to show system topology ranging from thread topology to cache and finally to NUMA topology, as well as `likwid-mpirun` for MPI wrapper for likwid-pin and likwid-perfctr.

This chapter has explored various aspects of optimizing code for HPC. I delved into the sig-

nificance of leveraging parallel hardware and programming models such as MPI and OpenMP to exploit shared and distributed memory architectures effectively. Additionally, I discussed the importance of vectorization techniques, such as SIMD instructions, in enhancing computational efficiency of a program. Given that HPC clusters are employed in the performance experiments, both MPI and OpenMP parallel programming frameworks are essential. This necessitates a hybrid MPI-OpenMP approach combined with vectorization techniques to achieve higher performance.

Furthermore, I examined the role of performance analysis tools like Intel® VTune™ Profiler and Intel® Advisor in identifying performance bottlenecks, optimizing code for modern computer architectures, and visualizing potential optimization strategies. These tools offer valuable insights into application performance, thread activity, hardware-related issues, and optimization opportunities, empowering developers to maximize the utilization of available hardware resources and improve overall application performance.

In my PhD project, Intel® VTune™ will be utilized to identify the primary performance bottlenecks in the routines. Additionally, Intel® Advisor will be employed to detect threading issues, such as data sharing conflicts and excessive synchronization, and to identify threading opportunities, such as determining which loops are the best candidates for parallelization. Intel® Inspector will be used to locate and debug threading and memory errors. LIKWID will be used for visualizing and understanding the thread, cache and NUMA topology.

Overall, by integrating parallel programming models, vectorization techniques, and performance analysis tools into the development workflow, I want to design and optimize `HiRep` code for diverse computing environments, ranging from single-core systems to large-scale HPC clusters. This holistic approach to code optimization is essential for meeting the increasing demands of modern computing applications while efficiently utilizing available hardware resources.

# Chapter 3

# Background Knowledge - HiRep and Dirac operator

***This chapter*** *establishes the foundational knowledge upon which the subsequent sections of this thesis are built. Initially, it introduces lattice regularization, followed by an exploration of the definition of the Dirac operator and an introduction to HiRep. The chapter further delves into the coding conventions, data structures, parallelization of the problem, elucidates the intricacies of data movement within the Dirac operator.*

ALL presently known elementary particle physics phenomena are described by four forces; these are the strong, electromagnetic, weak and gravitational forces. Particles that participate directly in the strong interactions are called hadrons, such are the proton, neutron the pions and many others. The fundamental theory that describes the strong interactions is called Quantum Chromodynamics (QCD). QCD is a quantum field theory in four dimensions (three dimensions for space and one for time), which is formally specified by a Lagrangian describing the interactions of so-called quarks (elementary constituent particles) and gluons (the carriers of the strong force).

## 3.1 Lattice Regularization

Defining the theory on a space-time lattice, as first proposed by Wilson (1974*b*), provides a non-perturbative gauge invariant UV regularization of QCD. In this formulation the lattice spacing `"a"` acts as an (inverse) UV cutoff, i.e. it allows for the theory to be properly defined up to some energy scale identified by the cutoff value.

Mostly (but not exclusively) regular hypercubic lattices are considered for simplicity, as for the case of interest in this presentation.

I will refer to the discretized spacetime as a four-dimensional hypercubic box spaced with an identical lattice spacing in each of the four directions, and I will refer to sites as the points belonging to the lattice and to links as to the bonds connecting two neighbouring sites. All lattice sites undergo evaluation, a process detailed in Algorithm 1 below. This evaluation involves loading and processing the neighboring sites and links associated with each lattice site.

For illustrative purposes, it is often referred to the $2\mathscr{D}$/$3\mathscr{D}$ case to represent the code in a simplified manner, however the `HiRep` code is always formulated in $4\mathscr{D}$.

For instance, let us consider the $2\mathscr{D}$ case illustrated in Figure 3.1. The lattice site (either red or blue) designated for evaluation is connected to two neighboring sites via two links in each dimension. This $2\mathscr{D}$ toroidal lattice space exhibits a unique property where any exit direction eventually leads back to the same starting point.

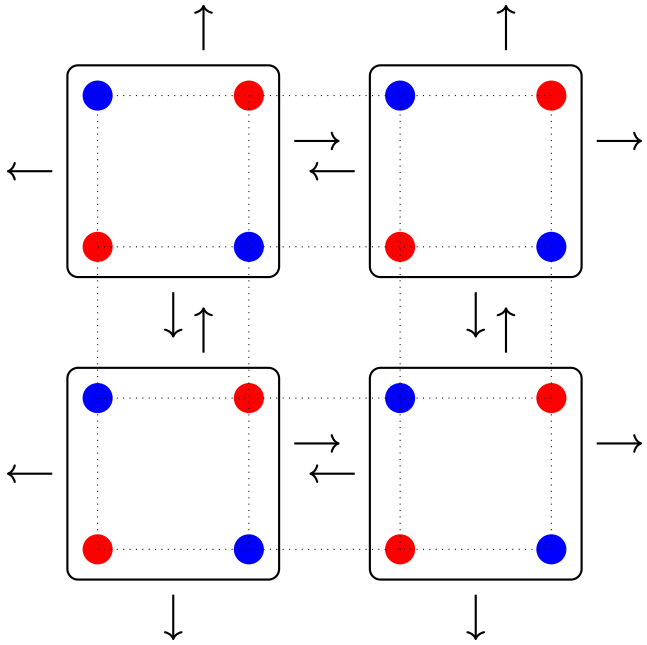


*Figure 3.1:* A $2\mathscr{D}$ toroidal lattice is shown; the red and blue dots represent sites, whereas the dotted links are bonds. In this lattice, the quark fields live on the sites, while the gluon fields reside on the bonds. Boxes enclose groups of four lattice sites. Arrows indicate the flow of sites in both the x-direction and y-direction at the top, bottom, and between the boxes.

The fundamental structures living in this spacetime are quark and gluon fields. The quark fields are structures located at lattice points and the dynamical gauge fields live on the links.

More in detail, each link joining sites $x$ to $x+\mathbf{a}\mu$ is associated with an $SU(N)$ matrix ($N \times N$) $U_\mu(x)$ encoding the gauge field, where $\mu$ is the unit vector in the $\mu$-direction, and N is an integer-valued tunable parameter of the theory. In the case of QCD, N = 3. Analogously, on each site, a (pseudo) quark vector of size $4N$ is placed.

| **Term** | **Definition** |
|---|---|
| Sites | The points belonging to the lattice. |
| Spinor | The field that associate to each site a set of 12 (4N) double precision complex numbers. |
| Bonds | The element connecting every pair of neighbouring sites. |
| Gauge Link | The field that associate to each bond a set of 9 ($N^2$) double precision complex numbers. |
| Piece | A contiguous chunk of memory comprising of an array of sites. |
| Bulk Piece | The inner segment of a lattice which is most contiguous in the memory. Computational operations executed on bulk elements do not require communication, as the sites within the bulk solely depend on those at the boundary (see Figure 3.5). |
| Boundary Piece | The next-to-bulk segment of a local lattice comprises the boundary, in which the MPI send buffers are located. The computation of boundary sites depends on the buffer sites received from neighboring processes (see Figure 3.5). |
| Send Buffers | A contiguous array of memory containing the elements which are geometrically displaced along one boundary surface. |
| Receive Buffers | An extended lattice segment which is used to store incoming portion of fields (e.g. spinors or gauge links) from neighboring processes (see Figure 3.5). |

*Table 3.1:* List of Terms and Definitions

Once a lattice action has been defined, expectation values of observables can be evaluated through the numerical integration over the dynamical fields. Their computation amounts to solving the Dirac equation several times,

$$D\psi(x) = \eta(x) \tag{3.1}$$

where $D$ denotes the massive lattice Dirac operator (depending on the gluon fields), $\eta(x)$ a given quark field (the source field) and $\psi(x)$ the desired solution.

The Dirac equation represents a large linear system, with sizes ranging from $96 \times 48^3 \times 12$ (127,401,984 rows) to $192 \times 192^3 \times 12$ (16,307,453,952 rows). In this notation, the value 96 or 192 denotes the temporal direction, the values $48^3$ or $192^3$ represent the spatial dimensions, and the final value corresponds to the 12 components per spinor. Solving such a large system requires iterative methods, which involve recursive procedures that generate sequences of increasingly accurate approximate solutions through the application of the Dirac operator.

This implies that a primary request for any simulation code of QCD is an efficient implementation of the Dirac operator to a quark field. The specific implementation of the Dirac operator that will be focused on in this work is known as the Wilson Dirac Operator.

## 3.2 Introduction to HiRep and Dirac operator

HiRep is a software package (Del Debbio et al. 2010; Pica 2023) widely used in lattice quantum chromodynamics (QCD) calculations. HiRep offers a powerful tool for simulating the fermions on a discrete lattice.

As already mentioned in section 3.1, there are two key data structures in Dirac operator: the spinor (quark) field and the gauge (gluon) field. Each element of a spinor field is associated to a site which holds one data item, and each of these items contains 4N double precision complex numbers. On the other hand, in the gauge field, each bond holds a matrix of size $N \times N$ filled with double precision complex numbers.

The sites are labeled in a checkerboard fashion as even or odd (or red and blue as depicted in our figures, such as Figures 3.1, 3.2, 3.5, and 5.2). When evaluating a spinor from the input data, whether for an even (red) or odd (blue) lattice site as the output, I need to load all the neighboring sites of the opposite parity, along with their corresponding gauge links, in the computation of the "hopping terms" (Figure 3.2).

The "hopping term" is a derivative part of the Dirac operator which exhibits its distinct pattern of operations. This term receives input of one parity (either even or odd) and yields output of the opposite parity (odd or even). The specific access pattern influenced by the even-odd structure results in a sparse matrix representation of the Dirac operator. Consequently, the operator is

accurately described as highly sparse due to the specific characteristics of its hopping term.

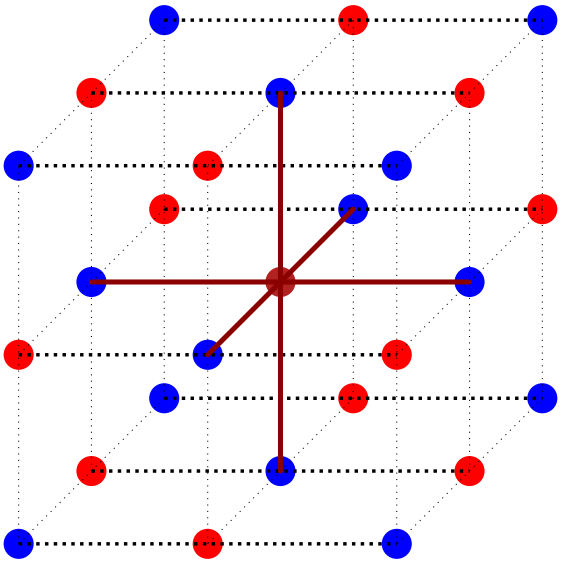


*Figure 3.2:* An example of a $3\mathscr{D}$ lattice is shown; the red dots are the even sites while the blue dots are the odd sites. For evaluating the central dark red site, the six blue adjacent (odd) sites and their corresponding gauge links are needed.

The order in which the sites are stored in main memory is determined by the lattice geometry and the even-odd structure as discussed. The even index sites are stored first in consecutive memory locations, while the odd index sites are stored next in consecutive memory locations too. The even and the odd sites are processed separately. The $4\mathscr{D}$ lattice $(t,x,y,z)$ is indexed by using a unique array index. This mapping forms a bijective relationship between the Cartesian coordinates and the array index, ensuring each point is uniquely identified.

**Algorithm 1:** Calculating the central red (even) point in the Dirac operator, as illustrated in Figure 3.2

1: **Load** six neighboring blue (odd) sites and their associated gauge links connected to the dark red central site (Figure 3.2).
2: **Process** the six blue sites and the six gauge links and accumulate the results.
3: **Update** the output dark red (even) site with the output values.

Figure 3.2 and Algorithm 1 serve as illustrative examples, portraying a $3\mathscr{D}$ lattice configuration. These examples highlight a specific data access pattern wherein the update of a red (even) site as an output requires the loading of all six neighbouring blue (odd) sites as inputs.

The arrangement of the data storage in memory is designed to be efficient, taking advantage of

the prefetching and caching capabilities of the CPU processors. In applying the Dirac operator, the evaluation of a single site output requires information from only neighboring sites of opposite parity as input. This implies that to evaluate all the sites of a given parity, input will only be needed from the part of the field of opposite parity, with potential data reuse due to the pattern of access of the Dirac operator. The strategy for data arrangement follows from these observations. Two independent allocations are stored, one for each parity, with a geometrical setup within the single array that maximizes data reuse.

As the Dirac operator operates on the lattice (specifically on its individual sites), in the computation process associated with the Dirac operator, the number of floating-point operations per site is

$$Flop\_site = 128N^2 + 56N \tag{3.2}$$

For the calculation of Dirac operator the amount of memory in bytes per site is

$$Memory\_site = 128N^2 + 640N \tag{3.3}$$

where 640N denotes 640 bytes scaled with N = 3 for the loading and storing of sites.

Furthermore, the Dirac operator's sparsity is visually represented in Figure 3.3 for a lattice size of $6 \times 4^3$, resulting in 384 sites. The operator is applied to all the spinors; a spinor is an array of elements of length 384 sites multiplied by the elements per site (4N). The matrix comprises of rows and columns/vectors, each containing 4608 complex elements. The first half of these elements is even, while the second half is odd, resulting in 2304 even and 2304 odd elements.

## 3.3 Coding Conventions

The HiRep simulation code is predominantly implemented in C programming language. Adhering to specific coding conventions is imperative within the HiRep implementation. Importantly, these conventions dictate the naming structure of functions and the usage of Macros, ensuring clarity and coherence throughout the codebase.

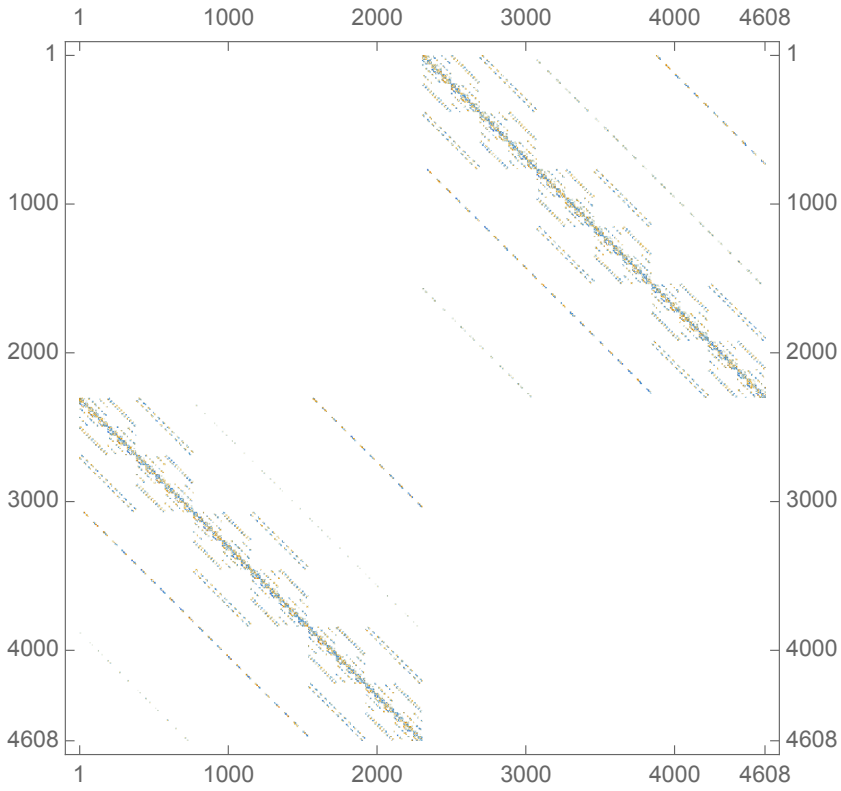



*Figure 3.3:* The visual representation of Dirac operator as a sparse operator. The colour defines the amplitude of the complex number in the entry.

### 3.3.1 Function Names

In the context of HiRep, function names follow a structured format to allow for better usability, extendability and maintenance. This format is exemplified in Figure 3.4, where each component of the function name serves a distinct purpose in conveying essential information about the operation being performed (Pica 2008*a*).

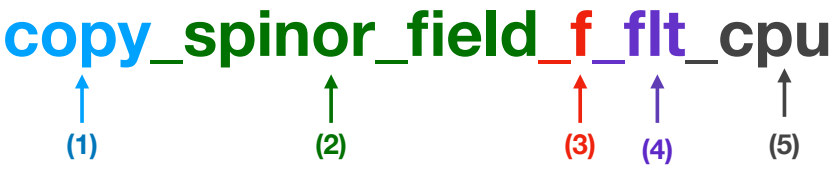



*Figure 3.4:* Naming Convention Example is provided in the reference by (Pica 2008*a*).

Firstly, part (1) of the function name signifies the specific operation being executed. This descriptive component provides a clear indication of the purpose or function of the code segment. Subsequently, in part (2) the type of field structure or elementary site structure on which the

operation is performed is denoted. This categorization aids in identifying the context or domain of the operation within the simulation framework.

Furthermore, the representation of the structure is denoted in part (3) of the function name. Specifically, the letter 'f' indicates the fundamental representation of the structure. Additionally, part (4) of the function name specifies whether the function operates with single or double precision. Notably, the suffix 'flt' corresponds to single precision, while the absence of this suffix implies double precision.

Finally, the suffixes 'cpu' and 'gpu' differentiate between functions operating on objects within the memory of the host CPU or the device GPU, respectively. This distinction is crucial for optimizing performance and resource utilization within the simulation environment.

By adhering to these naming conventions, the HiRep simulation code maintains clarity, consistency, and comprehensibility, facilitating efficient development, debugging, and maintenance processes.

### 3.3.2 Usage of Macros

The HiRep code extensively utilizes macros for various purposes. There are three primary reasons why macros are employed within the codebase (Pica 2008*a*):

1. **Lack of Function Overloading Feature in C**: Function overloading allows the same function to be executed with alternative argument sets and return types (Ali et al. 2023). However, the compiler in C programming does not support function overloading. To address this limitation, macros are declared instead of functions. An illustrative example is found in the macros defined in `HiRep`, which facilitate operations on elementary site types irrespective of precision (single or double). Consequently, the same macros can be employed for different data types, such as `suN_vector` and `suN_vector_flt`.

2. **Avoiding Code Replication**: Similar code segments often need to be implemented multiple times for different field types within HiRep. For instance, when implementing memory management, communication protocols, or data conversion operations for various field types that share common properties but differ in specific attributes (e.g., dimen-

sion), macros are utilized to define these functions dynamically. For instance, the function `declare_things` in Listing 3.1 needs to be declared for all field types, not just `spinor_field`.

```
      void declare_things(spinor_field* s) {
        suNf_spinor spinor;
     }
```

*Listing 3.1:* Example of a Non-Macro Function

In this case, a macro can be devised to accommodate different field types and their respective elementary site types as arguments. This macro would allow to invoke the similar functions across diverse data types. For instance, the following macro in Listing 3.2 could be implemented.

```
#define _DECLARE_USELESS_FUNCTION(_field_type,
   _site_type) \
    void declare_things(_field_type* s) { \
        _site_type* site; \
    }
_DECLARE_USELESS_FUNCTION(spinor_field,
   suNf_spinor)
_DECLARE_USELESS_FUNCTION(suNg_field, suNg)
_DECLARE_USELESS_FUNCTION(suNf_field, suNf)
```

*Listing 3.2:* Example of a Macro Function

With this macro, the function `declare_things` can be applied to various field types (e.g., `spinor_field`, `suNg_field`, `suNf_field`) by specifying the respective `_field_type` and elementary `_site_type` as arguments to the macro. This approach enhances code flexibility and reusability, allowing for the consistent application of functions across

different data types.

3. **Ensuring Consistency and Efficiency**: By encapsulating repetitive code segments within macros, HiRep ensures consistency and efficiency in code implementation. Macros facilitate the management of complex functionalities by providing a modular and reusable approach to code development. This not only enhances code readability and maintainability but also streamlines the process of incorporating changes or updates across different parts of the codebase.

In essence, the strategic utilization of macros in the HiRep codebase addresses various challenges inherent in C programming, such as function overloading limitations and code replication requirements. By leveraging macros, HiRep achieves code modularity, scalability, and maintainability, thereby enhancing the robustness and versatility of the simulation code.

## 3.4 Data Structures

Within HiRep, the organization of data revolves around two distinct categories: elementary data types and field data types. These classifications form the foundational structures upon which various computational components and algorithms within HiRep are built. Understanding the characteristics and roles of these data types is essential for comprehending the inner workings and capabilities of the HiRep simulation code (Pica 2008*b*).

### 3.4.1 Elementary Data Types

Within HiRep, elementary data types are fundamental building blocks defined as C structures containing a single array of elements (see Table 3.2). These data types serve as the foundation for creating fields, which are essential components for various computational tasks within HiRep. To facilitate the creation of fields, the macro `_DECLARE_FIELD_STRUCT` in `Include/spinor_field.h` is utilized.

In this context, $N_c$ represents the number of colors, while $D_R$ signifies the dimension of the fermion representation. The data type suNf may vary between real and complex, depending on the nature of the representation being either real or complex. Furthermore, each data type is

| Name | Array of | Size in Elements |
|---|---|---|
| hr_complex | double | 2 |
| suNg_vector | hr_complex | $N_c$ |
| suNg | hr_complex | $N_c \times N_c$ |
| suNg_spinor | suNg_vector | 4 |
| suNg_algebra_vector | double | $N_c \times N_c - 1$ |
| suNf_vector | hr_complex | $D_R$ |
| suNf | hr_complex or double | $D_R \times D_R$ |
| suNfc | hr_complex | $D_R \times D_R$ |
| suNf_spinor | suNf_vector | 4 |
| ldl_t | 2 arrays, hr_complex | each $D_R(2D_R + 1)$ |

*Table 3.2:* List of Elementary Data Types

accompanied by a corresponding single precision variant, denoted by appending the suffix _flt.

Linear algebra operations within the HiRep codebase necessitate looping over the elements in the arrays of the structures. However, to optimize performance, it is imperative to unroll for-loops in bottleneck functions. Consequently, these linear algebra functions are defined as macros, expanding the unrolled code. Given that the number of iterations in the for-loop relies on parameters such as the number of colors and the dimension of fermion representation, which must be known at compile time, the definition of these macros is contingent on compilation parameters.

An autogeneration approach facilitated by a Perl script in Make/Utils is employed to accommodate different macros based on compilation parameters. This mechanism ensures that the appropriate macros are available depending on the compilation parameters specified.

### 3.4.2 Field Data Types

In HiRep, field data is organized within field structs, each containing an array of values situated on sites or links. These arrays are allocated on the CPU, and if compiled with GPU acceleration, a corresponding allocation is made on the GPU. Definitions for various fields are outlined in `LibHR/spinor_field.h`. The available types include (Pica 2008*b*):

New field types can be declared by utilizing the predefined macro provided within the HiRep codebase. This macro simplifies the process of defining custom field types, allowing developers to easily extend the range of available field data structures within the HiRep framework.

| Name | Elementary Data Types |
|---|---|
| suNg_field | suNg |
| suNg_scalar_field | suNg_vector |
| suNf_field | suNf |
| spinor_field | suNf_spinor |
| suNg_av_field | suNg_algebra_vector |
| scalar_field | double |
| ldl_field | ldl_t |
| suNfc_field | suNfc |

*Table 3.3:* List of Field and Elementary Data Types

```
#define _DECLARE_FIELD_STRUCT(_name, _type) \
  typedef struct _name                      \
  {                                           \
    _type *ptr;                               \
    geometry_descriptor *type;                \
    _MPI_FIELD_DATA                           \
    _GPU_FIELD_DATA(_type)                    \
  } _name
```

*Listing 3.3:* A New Field Declaring Using the Macro

The macro `_DECLARE_FIELD_STRUCT` in Listing 3.3 facilitates the declaration of new field types within the HiRep code. When using this macro, the `_name` parameter specifies the name of the new field, providing flexibility in naming conventions. However, the `_type` parameter must correspond to a type previously defined in `suN_types.h`, as outlined in the preceding section. This ensures consistency and compatibility with the existing data structures and operations within the HiRep code-base.

The field value copy allocated on the CPU is represented by the variable `_type *ptr`, which is a one-dimensional array storing the values of the field at each lattice site. This array provides direct access to the field's data, allowing for efficient manipulation and computation within the HiRep simulation code.

In the Dirac operator, there are two key data structures: the spinor field and the gauge field. Each site in the spinor field holds one data item, and each of these items contains 4N double precision complex numbers. On the other hand, in the gauge field, each bond holds a matrix of size $N \times N$ filled with double precision complex numbers. It is imperative to note that a direct correspondence exists between each gauge field and individual lattice sites.

## 3.5 Problem Parallelization

The entire four-dimensional lattice of sites is divided into hyper blocks and each hyper block is associated to a distinct MPI (Message Passing Interface) process. Let us assume that GLB_T, GLB_X, GLB_Y, and GLB_Z represent the number of sites in the global lattice setup; and NP_T, NP_X, NP_Y and NP_Z represent the number of MPI processes in t, x, y and z dimensions (t denotes time and x, y, z denote space) respectively. The hyper block associated to a single MPI process is obtained by diving each direction GLB_T, GLB_X, GLB_Y and GLB_Z in NP_T, NP_X, NP_Y and NP_Z segments such that each block will be of size $\left(\frac{GLB\_T}{NP\_T} \times \frac{GLB\_X}{NP\_X} \times \frac{GLB\_Y}{NP\_Y} \times \frac{GLB\_Z}{NP\_Z}\right)$ (assuming, for the sake of simplicity, that all these sizes are integers). Each resulting hyper block is called a local lattice. Thus, the problem size in the local lattice is determined by dividing the problem size in the global lattice by the number of MPI processes. If the number of processes is increased in any direction for fixed global size, the size of the local lattice will become even smaller.

## 3.6 Data Movement in Dirac operator

The computation of the Dirac operator on a site requires data communication among distinct MPI processes, where each MPI process is responsible for computing its own local lattice. In Figure 3.5, two $2\mathscr{D}$ local lattice blocks are depicted, with sites in the bulk and boundary segments representing the original lattice blocks. Each original block is expanded to accommodate receive buffers with memory capable of storing data from the neighboring block's boundary segment.

However, HiRep's strategy aims to minimize memory copying by incorporating the send buffer directly within the boundary part of the lattice, reminding that, in MPI, the send buffer needs to

be contiguous in memory. To exemplify the implemented solution consider Figure 3.5 where send and receive buffer are highlighted in cyan and purple. To obtain a contiguous array of memory for each send buffer is sufficient to copy the memory of site 2 to the unused site 8. The synchronization of these two sites' memory is only necessary when an MPI send operation is required; therefore, this copying is performed only immediately prior to the MPI call. This approach reduces the need to copy every individual site in the send buffer, limiting it to just one site in the present example.

Communication involves the exchange of boundary sites between neighbouring local lattices. For instance, boundary sites (2, 3, 19, 20) and their associated gauge links from one block are copied to the receive buffer sites (13, 14, 29, 30) of another block. Similarly, boundary sites (5, 6, 22, and 23) and their gauge links are copied to the receive buffer sites (9, 10, 25, and 26).

To compute sites within the boundary block (Figure 3.5) using Algorithm 1, neighboring sites of different colors and their corresponding gauge links (indicated by green arrows) are loaded. Some of these sites and gauge links belong to the receive buffer region and must be obtained from the neighboring block associated with a different MPI process. Conversely, sites and gauge links from the boundary region need to be combined and sent to another neighboring block, which belongs to a different MPI process, for the calculation of corresponding sites.

For example, when calculating site 3 at the boundary of the top block in Figure 3.5, the four neighboring blue (odd) sites (17, 19, 20, 25) and their associated gauge links (indicated by green arrowheads) must be processed. Evidently, sites 17, 19, and 20, along with their gauge links, are loaded faster as they are local to the MPI process to which the block is assigned; while site 25 and its links need to be received from the neighboring block at the bottom which belongs to another MPI process. It is important to highlight that no communication is required to compute sites in the bulk region, as all the require data for these sites are local to the process.

Detailed instructions on how to use the HiRep simulation code—including prerequisites, compilation steps, and execution guidelines—are provided in the documentation available at the GitHub repository (Pica and contributors 2007).

In Chapter 3, various aspects of lattice regularization, an introduction to HiRep, and the Dirac

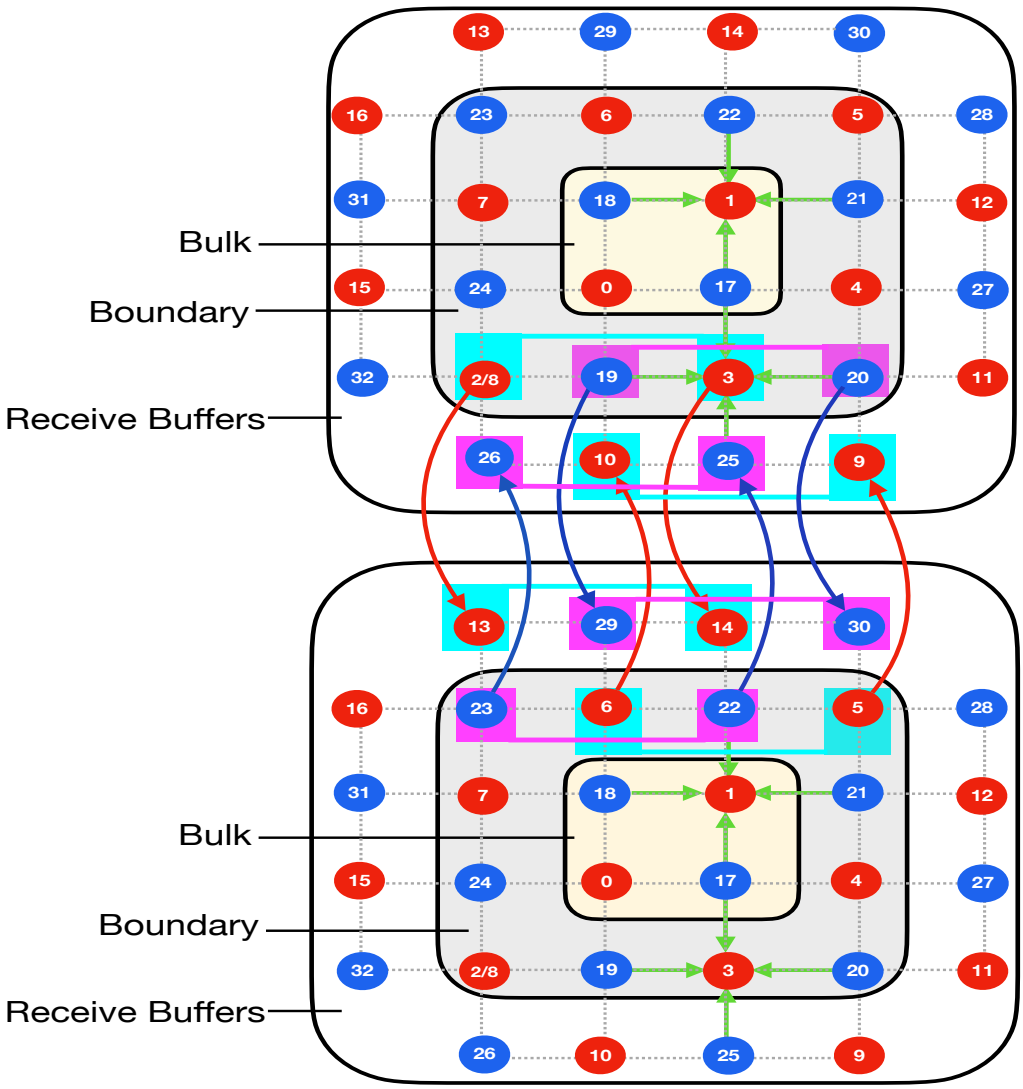



*Figure 3.5:* A visual representation of communication between $2\mathscr{D}$ local lattice blocks. The local lattice consists of the bulk, boundary and buffer pieces. The even/odd sites are shown in red/blue color, respectively, with a lexicographic indexing system. In the even lattice, sites 0-1 represent the bulk lattice sites. The boundary piece, where MPI send buffers are located, facilitates communication with neighboring processes. The boundary sites 2-7 are stored into consecutive memory locations. Since boundary sites 7-2 are non-contiguous in memory, the site at position 2 is copied to position 8, ensuring contiguity.

operator were explored. Coding conventions and data structures essential for efficient implementation were also discussed. Problem parallelization and data movement within the Dirac operator were also discussed in detail. Furthermore, I provided a document link for the comprehensive guide on how to use HiRep, covering prerequisites, compilation steps, and running

the simulation. This chapter lays a solid foundation for understanding the theoretical framework and practical implementation of lattice QCD simulations using HiRep, setting the stage for further exploration and experimentation in subsequent chapters.

# Chapter 4

# HiRep Performance Inefficiencies

***This chapter*** *identifies and addresses performance inefficiencies in the original version of HiRep. Four main inefficiencies are highlighted: communication overhead resulting from excessive data movement via network interconnect, non-uniform workload distribution hindering load balance in OpenMP parallel sections, inefficient data access patterns leading to limited data reuses, and compromised auto-vectorization support in compilers dealing with complex memory access patterns.*

DPHI is a routine which implements the Dirac operator in HiRep, entailing the multiplication of the fermion field by the Dirac operator (Bennett et al. 2016). The main performance inefficiencies of the `Dphi` routine are stemmed from communication overhead, workload imbalance challenges, inefficient data reuse in sites, and ineffective auto-vectorization. Before explaining these inefficiencies further, it is essential to first explain Algorithm 2.

**Algorithm 2:** Pseudo code for original Dphi routine

```
1: Start buffer communication    ▷ buffer sites must be sent and received
2: Load, Compute and Update (Algorithm 1) sites in bulk
3: Wait for buffer communication    ▷ wait for buffer sites to be received
4: for all boundary_pieces do    ▷ piece not equally sized
5:     for all sites_per_piece do
6:         Load, Compute (Algorithm 1) and Update sites
7:     end for
8: end for
```

`Dphi` was parallelized using the MPI programming model. Algorithm 2 presents the corresponding pseudo code. In line 1, each MPI process sends the buffer of fixed parity sites to its neighboring processes, which are received in line 3. While the buffer sites are being sent, the bulk sites are being processed in line 2 to hide the MPI communication. Then the sites in the

boundary piece, where MPI send buffers are located, are processed from lines 4 to 8.

The underlying principle of this algorithm is to overlap the computation of the bulk sites with the communication of the boundary ones, in order to mask the data transfer. The MPI standard offers an interface for non-blocking communication precisely for such requirements.

Each of these inefficiencies listed below is analysed and proposed strategies for optimization.

## 4.1 Communication Overhead

In the MPI-only code of HiRep (Algorithm 2), each core is assigned only one MPI process, and each MPI process handles an equivalent number of local lattice sites. This MPI-only configuration leads to data exchanges between MPI processes through MPI calls, resulting in substantial data movement both within and between nodes and introducing communication overhead.

To tackle this problem, it is efficient to use hybrid parallelization approach that allows the utilization of a limited number of MPI processes per node while leveraging OpenMP threads to exploit additional parallelism within the node. This reduction in the number of MPI processes can mitigate communication overhead associated with MPI calls. Further details are discussed in Section 5.1 of Chapter 5.

## 4.2 OpenMP Workload Imbalance Challenges

Efficiently parallelizing Algorithm 2 using the OpenMP parallel programming framework is challenging due to the non-uniform workload distribution of MPI send buffers in the boundary. This uneven workload distribution among OpenMP parallel regions during boundary site computation leads to inefficiencies in multi-threaded parallelism. Specifically, the number of sites in the send buffers within the boundary (line 4 in Algorithm 2) varies, lacking uniformity. This variability hinders load balance in multi-threaded parallelism with OpenMP. Solutions to address this issue are discussed in Section 5.2 of Chapter 5.

## 4.3 Inefficient Data Reuse in Lattice Sites

As it can be observed in Figure 3.5, certain sites exhibit a higher frequency of loading instances compared to others (data reuse). By changing the order that the sites are accessed, data reuse can be utilized and as a consequence reduce the number of cache misses. To evaluate red (even) sites, it is necessary to load the blue (odd) sites. Conversely, evaluating blue (odd) sites requires the loading of red (even) sites.

In a local $2\mathscr{D}$ lattice setup, for example, Figure 3.5 demonstrates the lexicographic memory access pattern. In the first step, four neighboring blue/odd sites (17, 18, 21, 22) are read from DRAM to write a red/even site (1), and no data reuse occurs in this process. However, in the second step, when writing another red/even site (3), a blue/odd site (17) is reused from the first step, which is already present in the cache. Thus in the subsequent steps only one site is reused. This finding suggests that the amount of data reuse is not efficient, regardless of the expansion in the lattice size. To maximize data reuse, the optimization of the data access pattern is described as an effective strategy in Section 5.3 of Chapter 5.

## 4.4 Ineffective Auto-Vectorization

Lastly, while modern compilers offer auto-vectorization support, its efficiency is often compromised when dealing with complex memory access patterns (Maleki et al. 2011). Such a case is `Dphi` routine where it is experimentally found that auto-vectorization is not efficiently applied. To this end, `Dphi` routine is manually vectorized using AVX2 or the Clang vector extensions and intrinsics (LLVM 2023) (see the implementation details in Section 5.4 of Chapter 5).

In summary, the `Dphi` routine in HiRep presents several performance inefficiencies, including communication overhead, workload imbalance challenges, inefficient data reuse, and ineffective auto-vectorization. By addressing these inefficiencies through optimized parallelization strategies, data access pattern enhancements, and manual vectorization techniques, the overall performance and efficiency of the Dirac operator application in HiRep are sought to be improved.

# Chapter 5

# Optimization Methodology

***This chapter*** *focuses on optimizing the performance of HiRep through various optimization techniques. The initial optimization involves hybridizing the existing MPI-only code by integrating the OpenMP parallel programming framework, thereby leveraging multi-threading capabilities to improve parallelism. Subsequently, a loop collapse optimization strategy is implemented to enhance OpenMP parallelism further, enabling more efficient utilization of computational resources. Additionally, efforts are directed towards optimizing data access patterns within the bulk region of the lattice to minimize cache misses, enhancing data reuse capabilities. Rather than relying solely on compiler auto-vectorization (which is not efficient here), manual vectorization techniques are applied to matrix-vector multiplication (MVM) operations within the* `Dphi` *routine, resulting in improved computational efficiency and overall performance.*

THE OPTIMIZATION methodology encompasses several strategies aimed at improving the performance of the HiRep simulation code. The selection of these strategies is informed by a comprehensive code analysis, as well as the specific performance inefficiencies detailed in Chapter 4. The main optimizations are discussed below:

## 5.1 Hybrid MPI-OpenMP Parallelization

Since HiRep targets distributed memory systems, a hybrid (MPI-OpenMP) parallelization strategy is embraced. MPI is used to distribute the workload on different processor nodes, while OpenMP is used to distribute the workload on the CPU cores of a node.

In the baseline (MPI-only) implementation of the `Dphi` routine, users have the flexibility to determine the number of MPI processes through configurable parameters denoted as NP_T, NP_X, NP_Y, and NP_Z. These parameters correspond to the number of lattice blocks along the

respective directions (T, X, Y, Z). Meaning that there is a regular division into blocks along each direction, with each block belonging to an MPI process.

In the case of $1\mathscr{D}$ MPI process decomposition over a global lattice (Figure 3.5), for example, two MPI processes correspond to two local lattice blocks, each exclusively associated with a distinct MPI process. However, in the scenario of a $4\mathscr{D}$ process decomposition, the overall count of blocks is determined by the product of the parameters, expressed as $(NP\_T) \times (NP\_X) \times (NP\_Y) \times (NP\_Z)$. For instance, if a user specifies $NP\_T = 2$ and $NP\_X = 2$ in the T and X directions respectively, the total number of processes would amount to 4 ($2 \times 2 \times 1 \times 1$). This also signifies that there are 4 local lattice blocks and each of them is computed by a single MPI process. Coupled with MPI processes, the users define the global problem size, and then the local problem size is computed by dividing the global problem size by the number of MPI processes in each direction, as already elaborated in Section 3.5.

**Algorithm 3:** Dphi optimization with MPI-OpenMP + Loop Collapse

```
1: #pragma omp parallel
2: {
3:     #pragma omp master
4:       Master thread starts non-blocking MPI_send of sites to its neighboring processes
5:     #pragma omp for
6:       for bulk_sites do
7:         Load, Compute and Update bulk_sites
8:       end for
9:     #pragma omp master
10:      Master thread completes MPI_recv of sites from neighboring processes
11:      Other threads wait here until the master thread completes MPI_recv
12:    #pragma omp barrier
13:    #pragma omp for
14:      for boundary_sites do
15:        Load, Compute and Update boundary_sites
16:      end for
17: }
```

Algorithm 3 represents an optimized version of Algorithm 2, aimed at improving the performance of `Dphi`. This optimization incorporates OpenMP, transforming the original MPI-only implementation into a hybrid MPI-OpenMP application. In Algorithm 3, the MPI communication and the bulk computation (Algorithm 1) are interleaved for improved performance. When

the MPI communication is finished (line 12) in Algorithm 3), the boundary sites are computed (Algorithm 1).

The line 1 in Algorithm 3 initiates the OpenMP parallel region. It ensures that multiple threads are spawned to execute the code that follows concurrently. In line 3, the master thread starts the `non-blocking MPI_send` operation, which sends site data to neighboring processes. This operation allows the master thread to initiate data transmission without waiting for it to complete, thus enabling overlapping communication with computation.

In line 5, bulk sites are processed with OpenMP For Loop by all threads. The master thread, in line 9, completes receiving site data from neighboring processes. This step synchronizes the receipt of data that was previously sent. In line 12, all other threads wait at a synchronization point until the master thread finishes the receiving data. This synchronization is crucial to ensure that all threads have the required data before proceeding. Finally, in line 13, boundary sites are processed with OpenMP For Loop by all threads.

This hybrid parallelization approach offers two key advantages. Firstly, it enables the utilization of a less number of MPI processes per compute node, leveraging OpenMP threads to further exploit parallelism within the node. As an efficient setup, only one MPI process is mapped to each NUMA domain and/or sub-domain along with its OpenMP threads. This reduction in MPI processes mitigates MPI communication overhead resulting from MPI calls. Secondly, the sharing of data via the L3 cache by OpenMP threads enhances cache locality. Improved cache (data) locality contributes to reduced time spent on memory access, consequently enhancing overall performance. However, achieving efficiency with hybrid parallelization is challenging due to the uneven MPI send buffers in the boundary segment of the lattice.

Furthermore, the loops responsible for processing the boundary sites in Algorithm 2 (line 4 and 5) are collapsed (line 14 in Algorithm 3) and as a consequence only one loop exists now. This optimization addresses the OpenMP workload imbalance challenge mentioned in Section 4.2. It is important to note that loop collapsing is far from trivial in this context and it is explained in Section 5.2.

The optimized algorithm leverages hybrid parallelization through the MPI-OpenMP frame-

works. By collapsing loops and strategically synchronizing threads, the algorithm aims to reduce communication overhead and improve load balancing, ultimately enhancing the overall performance of the `Dphi` routine.

For improved performance and efficient utilization of hardware resources, MPI processes are mapped to specific nodes and their corresponding processors. Simultaneously, OpenMP threads were assigned to specific CPU cores. Given the hybrid parallelization approach, the determination of the number of OpenMP threads is contingent upon the count of CPU cores per socket and the number of sockets within a NUMA architecture. Thus, the configuration of OpenMP threads is tailored to these parameters, ensuring a finely tuned utilization of available resources.

An efficient MPI-OpenMP configuration is typically achieved by assigning one MPI process to each socket or NUMA domain, with the corresponding OpenMP threads distributed across the cores within that domain. This approach ensures efficient use of computational resources and minimizes data movement between sockets. For example, in a hardware architecture with two sockets per node, each containing 16 cores, the optimal configuration would involve 2 MPI processes, each with 16 OpenMP threads.

To ensure that the MPI processes and OpenMP threads utilize the expected hardware resources, two crucial environmental variables need to set: `OMP_PLACES=cores` and `OMP_PROC_BIND=close`. The former binds threads to specific CPU cores, while the latter ensures that thread placement occurs sequentially across the available cores (Arora et al. 2018). The `OMP_PROC_BIND` variable controls thread migration behavior (Li et al. 2023), with options such as `close` and `spread`. Setting `OMP_PROC_BIND=close` directs that thread assignment proceeds successively through the designated cores (Arora et al. 2018).

In summary, managing affinity in hybrid HiRep application is important for optimizing resource utilization, minimizing remote memory accesses, and achieving better performance on parallel computing architectures (see Section 6.1).

## 5.2 Optimizing OpenMP Parallelism Through Loop Collapsing

Algorithm 2 exhibits workload imbalance within the MPI send buffers in the boundary segment of the lattice (line 4). Specifically, when each MPI send buffer is embedded within an OpenMP parallel region, where all OpenMP threads compute the buffer sites, a workload imbalance arises among the different OpenMP regions. This imbalance is due to the varying sizes of send buffers within the boundary piece (line 4 in Algorithm 2), where the sizes progressively decrease, leading to an uneven distribution of work during multi-threaded parallelism.

To tackle the workload imbalance, the algorithm was modified by collapsing multiple OpenMP regions into a single OpenMP region, resulting in a more balanced distribution of work. This modification is implemented in Algorithm 3, which introduces a single OpenMP loop, `"boundary_sites"` (line 14), that combines the *boundary_pieces* and *sites_per_piece* loops from Algorithm 2 (lines 4 and 5). This loop processes all boundary sites within each local lattice.

In order to integrate the bulk and boundary sites of the lattice and handle the entire boundary as a single send buffer, a lookup table is created. This table contains the specific sequence in which the sites are intended to be accessed. For this purpose, a new routine has been created to populate an array with the indices of all memory addresses of lattice sites. The array's size (i.e., the new table's size) is determined by the scope of the problem under consideration. For example, with a problem size of $32 \times 16 \times 16 \times 16$, the table array size is 131,072. When the problem size increases to $64 \times 32 \times 32 \times 32$, the table array size expands to 2,097,152. The first element of the array corresponds to the memory address of the initial site to be accessed, the second element to the subsequent site, and so forth.

Using this lookup table, the indices for bulk sites can be retrieved as needed and utilized for bulk site computations. Similarly, when indices for boundary sites are required, they are used for boundary site computations. For example, in Figure 3.5, the function retrieves elements 0 and 1 from the lookup table when evaluating even (red) bulk sites, and elements 17 and 18 when evaluating odd (blue) bulk sites.

In the context of the `loop collapse` optimization, the lookup table allows for the process-

ing of boundary sites as a single unified send buffer within a single loop, rather than splitting them into multiple send buffer chunks and processing them separately. The lookup table thus plays a pivotal role in enhancing the overall functionality and efficiency of the computation and communication.

Furthermore, to maintain a uniform distribution of iterations among the available threads, the OpenMP `schedule(static)` clause is used where the loop iterations are equally distributed to the threads at compile time.

The aforementioned optimization strategy provides two key advantages. Firstly, an equal number of iterations (chunks) are assigned to each thread within the loop for `"boundary_sites"` (line 14 in Algorithm 3), promoting load balancing and contributing to the effective utilization of computational resources in parallel execution. Secondly, communication efficiency is enhanced. The synchronization points in MPI (line 4 in Algorithm 2) are minimized by consolidating the split MPI send buffers into a single MPI send buffer (line 14 in Algorithm 3). The sites in this unified send buffer are then transmitted to the neighboring processes as a single message chunk. However, a drawback lies in the fact that invoking a function for the lookup table at each step of the calculation introduces additional function call overhead.

## 5.3 Data Access Patterns Optimization (MPI+Path-blocking)

The Dirac operator, functioning as a hopping term, requires access to its neighboring sites within the lattice (Figure 5.1). However, these neighboring sites are not sequentially positioned in memory, leading to non-contiguous memory access patterns when attempting to access them (see Figure 5.1 numbering). In the existing memory access patterns, data reuse is not fully exploited, resulting in a significant number of cache misses.

To further elucidate the issue of inefficient data reuse, an example is provided. When updating a single site as shown in Figure 5.1, four adjacent sites from the up and down directions of the x and y axes are required. In the $4\mathscr{D}$ setup, however, eight neighboring sites are loaded from the x, y, z, and t directions. Each direction contributes two sites, one from up and one from down, relative to the site being updated.

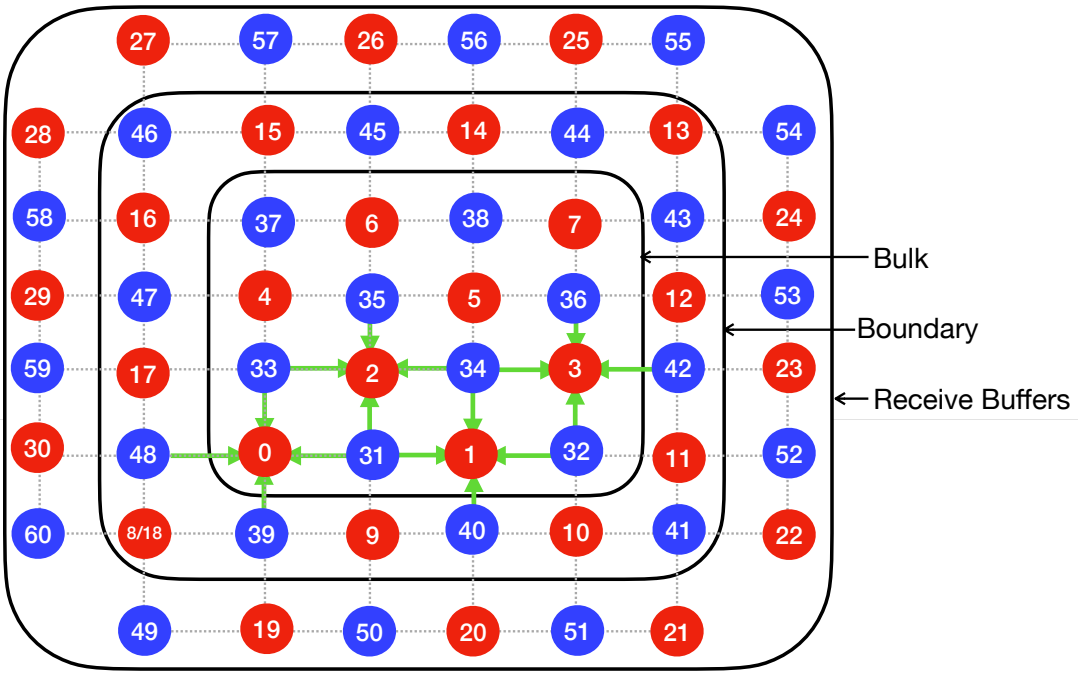



*Figure 5.1:* Non-Path-blocking on a 6-by-6 $2\mathscr{D}$ local lattice. The local lattice consists of the bulk, boundary and buffer parts, with even/odd sites depicted in red/blue and lexicographically indexed. Within the bulk, sites are ordered lexicographically. For an even lattice, boundary elements are non-contiguous, necessitating the copying of the element at position 8 to position 18 to ensure contiguity. Conversely, the odd part of the lattice maintains memory contiguity.

In the non-path-blocking case, the Figure 5.1 shows that when updating the site (0), site 48 is loaded from down and site 31 from up in the x direction. Likewise, site 39 is loaded from down and site 33 from up in the y direction. In this initial step of computation, there lacks data reuse as all four neighboring sites are loaded from the main memory (and none from the cache). Subsequently, when advancing to update the second site (1), only site 31 is reused. Thus, only one site is reused per computation step. The remaining sites (32, 34, and 40) continue to be fetched from the main memory. In the subsequent steps of computation, almost similar pattern persists, with minimal data reuse.

However, to allow for more consecutive memory accesses, the way data are stored into memory has been changed and implemented an approach in which the sites are accessed in blocks in the bulk portion of the lattice. This optimization has been named path-blocking.

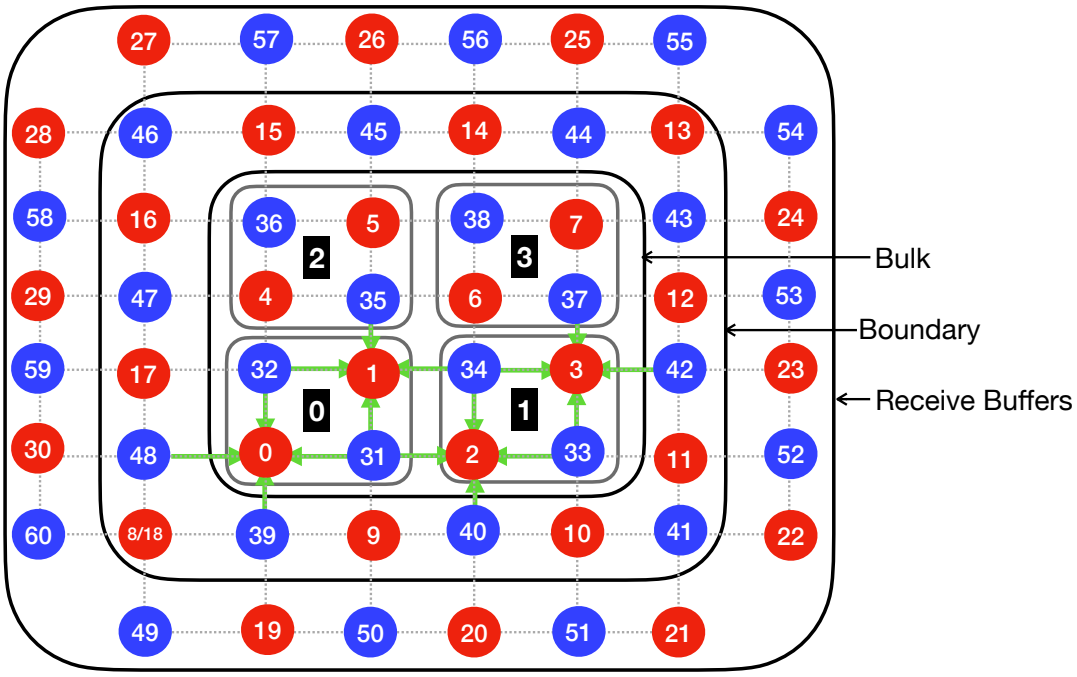



*Figure 5.2:* Path-blocking on a 6-by-6 $2\mathscr{D}$ local lattice with sub-block dimensions X=2 and Y=2. The local lattice consists of the bulk, boundary and buffer parts, with even/odd sites depicted in red/blue and lexicographically indexed. Within the bulk, sites are organized into blocks, ordered lexicographically. For an even lattice, boundary elements are non-contiguous, necessitating the copying of the element at position 8 to position 18 to ensure contiguity. Conversely, the odd part of the lattice maintains memory contiguity.

In order to optimize data reuses, the `path-blocking` strategy (as depicted in Figure 5.2) is employed where the bulk sites within a local lattice are sub-divided into smaller site blocks. In this $2\mathscr{D}$ configuration, each block has dimensions of $2 \times 2$ (x = 2, y = 2). The block size has been determined experimentally. The process involves selecting a block and computing the sites associated with the block before proceeding to the next block and repeating the computation for the sites within that block. This approach increases contiguous memory accesses, thereby improving memory access efficiency.

An example of a $2\mathscr{D}$ setup, as depicted in Figure 5.2, illustrates how data reuse is achieved through path-blocking optimization. To update the site 0 located within the block 0, 4 corresponding neighbouring blue (odd) sites (31, 32, 39 and 48) and their associated gauge links (marked as green) are needed to load from main memory. The memory operations in this $2\mathscr{D}$

set up entail 1 site store, 4 site loads, and 4 gauge link loads. However, when updating the site 1 within the same block (0), data reuse can be capitalized. The reuse of sites 31 and 32 from the cache is possible as they are already available and contiguous within the block. This minimizes the necessity for data loads from the slower and more energy-demanding main memory. Consequently, caching proves to be optimal when the bulk elements are subdivided into smaller block elements.

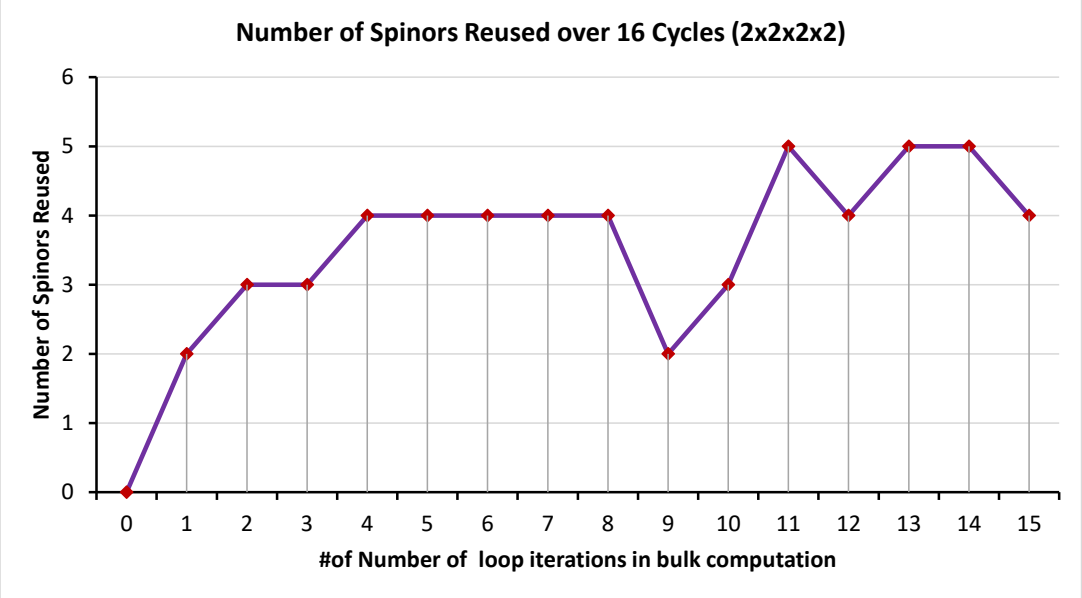



*Figure 5.3:* After implementing path-blocking optimization, a theoretical analysis was conducted to count the number of sites reused across all 16 computation steps while computing the bulk part of the lattice. Each loop iteration evaluates one lattice site.

After implementing path-blocking optimization, a theoretical analysis was conducted to evaluate data reuse during different loop iterations (line 6 in Algorithm 3), maintaining a manual counting of site reuses and measuring site hits when keeping a 40MB of L3 cache. The results of theoretical counting of site reuses in Figure 5.3 show that the reuses grow as the evaluation steps increase. The highest number of site reuses observed in the path-blocking case is 5, whereas in the non-path-blocking case, only one site was reused. This suggests that the implementation of path-blocking resulted in an increase in data reuse.

## 5.4 Vectorization

The core computational part in `Dphi` involves the application of gauge matrices to spinors through a Matrix-Vector Multiplication (MVM) operation. This MVM operation is performance critical and compilers struggle to optimize it effectively, particularly concerning auto-vectorization. To address this issue, the code was manually vectorized to improve performance.

Algorithm 4 demonstrates the double MVM implementation. Initially, two complex number vectors and a $3 \times 3$ matrix are loaded (line 1). Subsequently, the matrix is multiplied twice—first by the first vector and then by the second vector (line 2). Finally, the two resulting vectors from the MVM operations are stored (line 3). However, it was found that this MVM operation is slow, as none of the compilers employed succeeded in auto-vectorizing the MVM code in `Dphi` due to the inclusion of complex multiplication operations (line 2).

**Algorithm 4:** Pseudo Code - Dphi_hop for each site (double MVM)

1: **Load** two complex number vectors of size 3 and a complex $3 \times 3$ matrix
2: **Multiply** the same matrix by two complex number vectors
3: **Store** two resulting complex number vectors of size 3

Instead of relying on the compiler's auto-vectorization option, the double MVM implementation (Algorithm 4) was manually optimized using AVX2 intrinsics. However, the optimization can also be used to the hardware architectures which support AVX-512 [1] or Arm NEON [2]. The AVX2 optimized MVM operation encompasses a range of SIMD AVX2 instructions, including load, multiplication, addition, subtraction, and the `fmaddsub`. Nevertheless, in this context of complex numbers as operands, additional instructions such as duplicate, shuffle, cast, blend, or permute become essential for manipulating elements within vector registers (Popovici et al. 2017; Van Zee and Smith 2017).

In Figures 5.4, 5.5, and 5.6, the process begins with the loading of complex numbers. Subsequently, the data undergoes reordering through permutations to position them in their appropriate locations. Ultimately, following the reorganization, arithmetic operations are executed and

[1] https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX_512

[2] https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiessimdisa=[Neon]

the data is stored back in memory.

It is important to note that the shuffle operations must be executed prior to the commencement of floating-point computations. This sequencing ensures the proper alignment and arrangement of data elements necessary for accurate arithmetic operations.

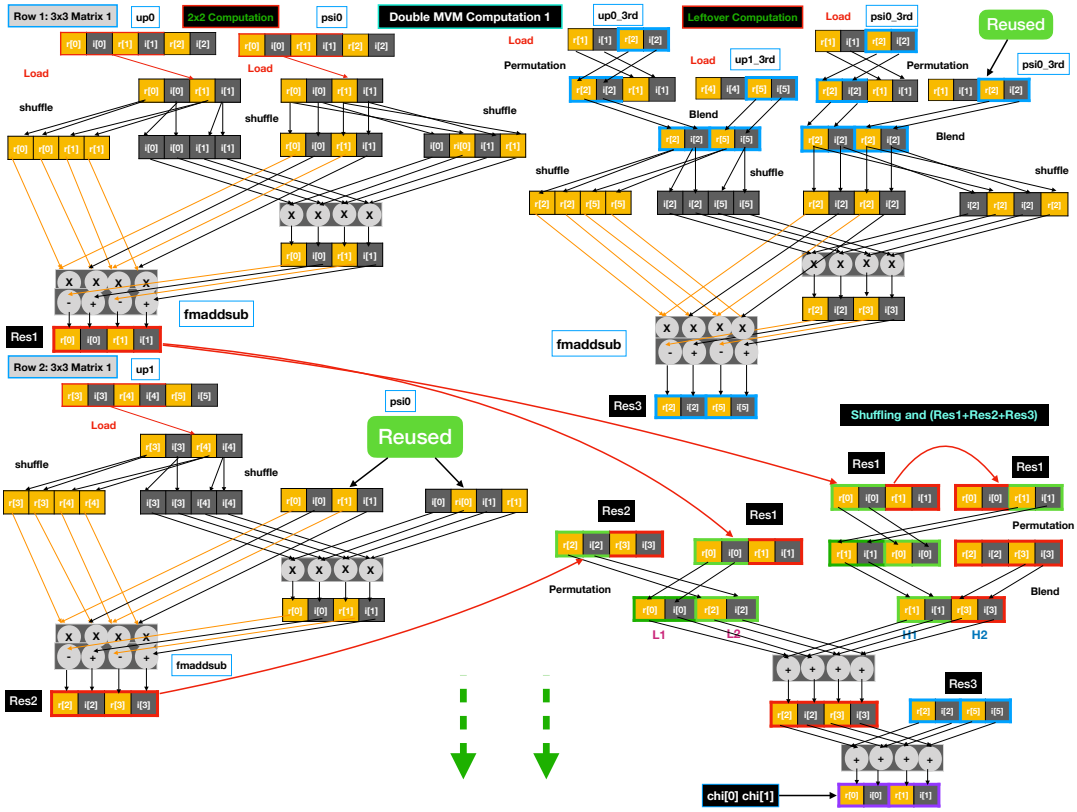



*Figure 5.4:* An AVX2 Implementation of double MVM routine PART-1. A matrix of 9 complex elements ($3 \times 3$) is multiplied by a vector of 3 complex elements. In this figure only the multiplication of the first two rows by the vector is shown. The multiplication of the third row is similar. Yellow boxes signify the real part, while grey boxes represent the imaginary part of the complex numbers.

In Figure 5.4, each vector register can hold two complex numbers (four double precision values). Each row and column/vector comprises of three double complex numbers, corresponding to six double precision numbers. Thus, to multiply the first row by the vector, the first two complex numbers need to be loaded and processed separately from the third one. The first two complex numbers are loaded and processed first (top left block). The complex multiplication has been efficiently implemented by using shuffle and `fmaddsub` instructions. This procedure is repeated for the multiplication of the first two complex numbers of the second row by the

vector (bottom left load block).

Subsequently, the third complex number of the first row is loaded and multiplied by the third complex number of the vector; since the third complex number of the second row is multiplied by the same vector value, this operation is optimized (top right block). Then, the results are unpacked and accumulated (bottom right block). Lastly, the two resulting complex numbers are stored into memory.

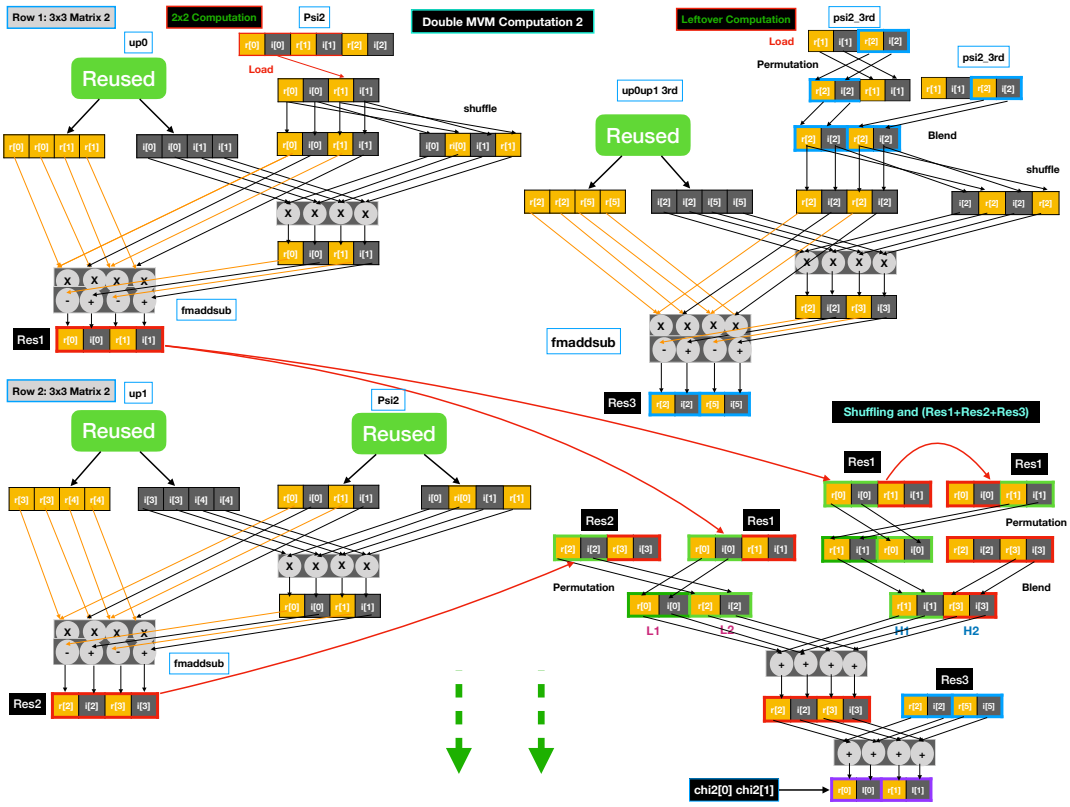



*Figure 5.5:* An AVX2 Implementation of double MVM routine PART-2. A matrix of 9 complex elements ($3 \times 3$) is multiplied by a vector of 3 complex elements. In this figure only the multiplication of the first two rows ($2 \times 3$) by the vector is shown. The multiplication of the third row is similar. Yellow boxes signify the real part, while grey boxes represent the imaginary part of the complex numbers.

In the implementation of the double Matrix-Vector Multiplication (MVM), illustrated in Figure 5.5, the process to include the multiplication of the first and second rows of the same matrix is extended, as the matrix is reused in the MVM computation 2, utilizing a new vector. This optimization aims to enhance computational efficiency by minimizing redundant operations.

Specifically, to perform the multiplication of the first row by the new vector, the first two com-

plex numbers are reused and processed initially (top left block). This approach optimizes the arithmetic and shuffle operations.

This process is then repeated for the second row and the same vector multiplication. Here, the operations are further optimized, as both the row and column complex elements are reused (bottom left block). Subsequently, the third complex number of the first and second rows are reused when multiplied by the third complex number of the column (vector). Finally, the resulting two complex numbers are stored into memory.

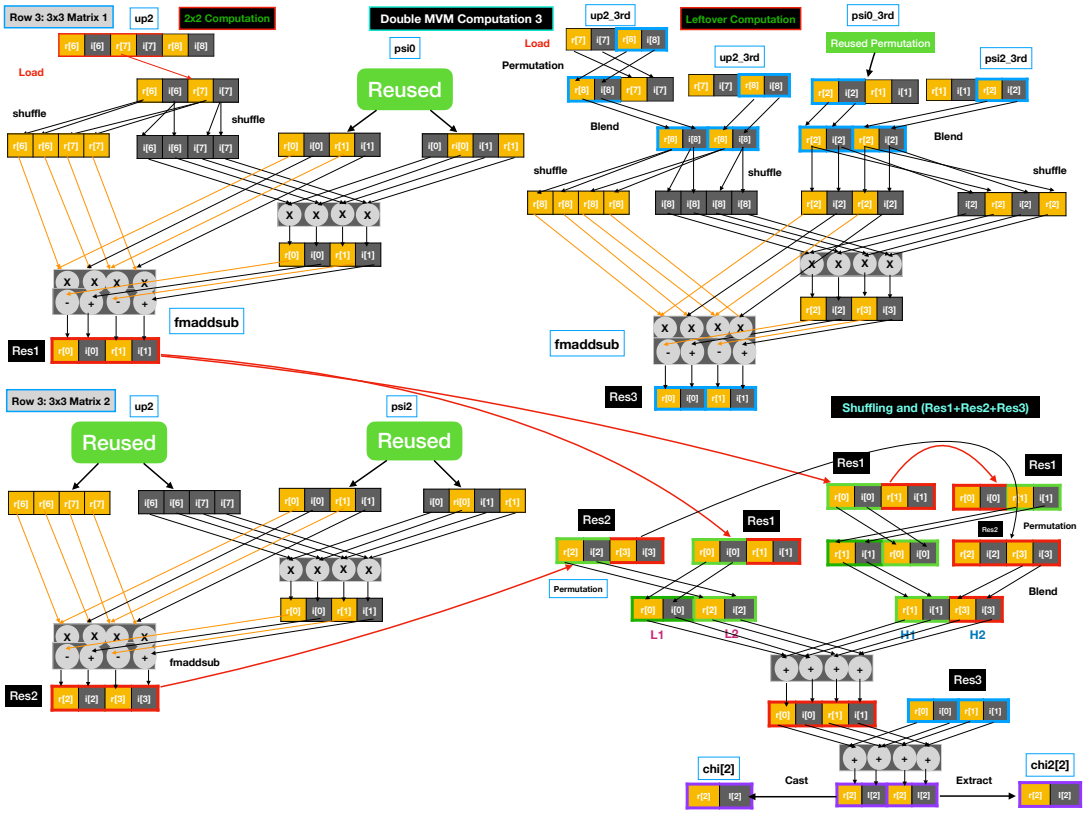



*Figure 5.6:* An AVX2 Implementation of double MVM routine PART-3. The input matrix is 3x3 and contains complex double precision values. AVX2 (SIMD vector length = 4) Implementation of Linear Algebra double MVM Routine in which a matrix of 9 complex elements (3x3) and a vector of 3 complex elements. This figure deals with the row 3 of the matrix 1 and 2 (the same matrix) with a new column vector.

The multiplication process for the third row of the matrix follows a similar pattern. In the top left block, the two complex numbers corresponding to the third row of the matrix are loaded into registers and subsequently shuffled. Meanwhile, the vector (column) is already present in the register, allowing for reuse during arithmetic operations (specifically, fmaddsub instructions).

Moving to the bottom left block, both the row and the column (vector) are shuffled and reused during arithmetic operations. In the top right block, the operation involves utilizing the row comprising the third elements of the third rows from both matrices, along with the corresponding elements from the vector (as depicted in Figure 5.6).

Alongside the AVX2 implementations, Clang compiler's vector intrinsics are also utilized for the MVM operations (LLVM 2023) (see Section 2.3 for details). Compared to AVX2 intrinsics that target 256-bit x86 vector instructions only, Clang compiler's vector intrinsics are hardware independent and thus they provide improved portability. However, since AVX2 intrinsics are tied to the x86 architectures only, they normally achieve higher performance on x86 architectures.

To sum up, Chapter 6 will show that the utilization of manual vectorization as an optimization strategy on double MVM operations enhances the performance of `Dphi`. This approach, leveraging SIMD AVX2 vector extensions, not only enables efficient utilization of registers but also minimizes the need for redundant data loading and reshuffling. Consequently, these enhancements collectively contribute to a significant improvement in computational efficiency.

# Chapter 6

# Experimental Results

***This chapter*** *delves into three primary sub-sections: experimental setup, performance metrics, and performance evaluation. The results, obtained from experiments conducted on both Foseres and HIPPO HPC platforms, distinctly illustrate the performance improvements over Foseres. Based on the experimental results from these two hardware platforms, all optimizations boost HiRep performance resulting in an up to* $\times 1.80$ *overall speedup. Additionally, code scalability is observed to be superior in HIPPO compared to Foseres due to HIPPO's high performance interconnection network.*

THIS CHAPTER is organized into three main sub-sections. First, the hardware resources of the two HPC platforms, Foseres and HIPPO, where the optimized code was tested, are detailed. Second, the performance measurement metrics utilized to evaluate the optimizations are explained. Third, the experimental results are presented and analyzed, demonstrating the performance improvements achieved through various optimizations. The comparative analysis between Foseres and HIPPO highlights the superior scalability and efficiency gains on the HIPPO platform.

## 6.1 Experimental Setup

The experimental results are performed on two x64 platforms, namely Foseres and HIPPO which use 32 and 128 CPU cores per node, respectively (Figures 6.1 and 6.2).

As illustrated in the Figure 6.1, a Foseres compute node is equipped with two sockets, where each socket is a 16-core Intel Xeon E5-2683v4 (Broadwell) processor, resulting in a total of 32 cores per node. Each core possesses a 32KB private L1 data cache, 32KB private L1 instruction cache, and a 256KB private L2 cache, while the socket features a 40MB shared L3 cache.

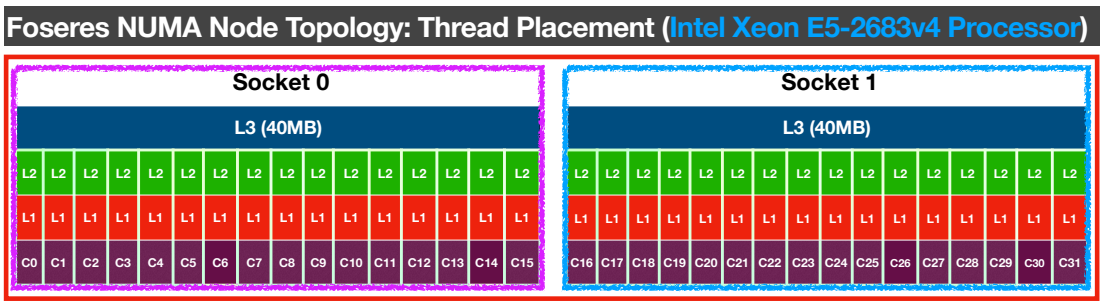



*Figure 6.1:* Foseres NUMA node architecture: The diagram illustrates the organization of the NUMA (Non-Uniform Memory Access) node within the Foseres system. The notation L1, L2, and L3 denotes the first, second, and third levels of cache memory, respectively. The designation C represents individual physical CPU cores, with C0-C31 indicating a total of 32 physical cores distributed across two sockets within the node.

Consequently, the combined L3 cache capacity across both sockets amounts to 80MB (Intel Corporation 2022*b*). Each CPU core supports AVX2 Instruction Set Extensions (ISEs); this is further explained in Chapter 2.

The compute nodes in Foseres cluster are connected with Intel® Omni-Path Architecture, providing high bandwidth of 100 GB/s and low-latency communication between nodes in the cluster, as elucidated in UoP (2023) and UoP (2022).

On the other hand, the Figure 6.2 shows that a HIPPO cluster node is also equipped with two sockets/NUMA-Nodes, where each socket is a 64-Core AMD EPYC 7713 processor which is decomposed into 8 sub-domains. Each sub-domain is equipped with 8 cores, resulting in a total of 128 cores per compute node. The core architecture of the AMD EPYC 7713 processor includes a 32KB private L1 data cache, 32KB private L1 instruction cache, and a 512KB private L2 caches per core and a 32MB shared L3 cache per sub-domain. Thus, the total shared L3 cache available is 512MB (32MB * 16 sub-domains) (AMD 2021). Each CPU core supports a broad array of SSE, AVX2, and AVX-512 Instruction Set Extensions (ISEs). The system utilizes a high-speed 100 Gbps InfiniBand EDR interconnect (UoSD 2023) to maintain the low-latency connection network.

Three distinct compilers were employed: Intel Compiler (icc) version 2021.2.0, gcc-12.2.0, and clang-15.0.7. While all three compilers were utilized in Foseres, only gcc and clang were

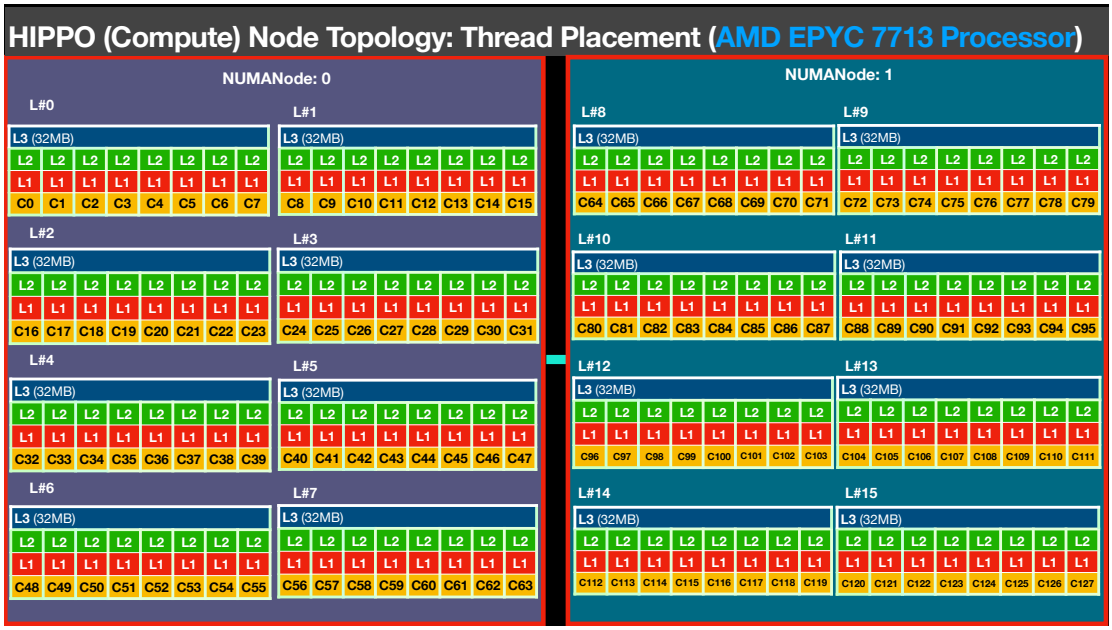



*Figure 6.2:* HIPPO NUMA node architecture: The diagram depicts the arrangement of the NUMA (Non-Uniform Memory Access) node within the HIPPO system. The labels L1, L2, and L3 correspond to the first, second, and third levels of cache memory, respectively. L#0 - L#15 corresponds to the collective sub-domains of two NUMA-Nodes. The designation C denotes individual CPU cores, with C0-C127 signifying a total of 128 cores distributed across two sockets/NUMA-Nodes within the node.

employed in HIPPO. Furthermore, three distinct MPI implementations were utilized: MPICH, OpenMPI, and Intel® MPI.

In the context of theoretical peak performance and bandwidth, Foseres and HIPPO exhibit distinct characteristics. A node in Foseres has a theoretical peak performance of 537.6 GFLOPs (Intel Corporation 2022*a*) and main memory bandwidth of 93 GB/s. Conversely, a node in HIPPO features a significantly higher theoretical peak performance of 2048 GFLOPs (Microway 2021), and memory bandwidth of 409.6 GB/s (204.8 GB/s per channel) (AMD 2021). These distinctions in theoretical performance metrics underscore the architectural disparities between the two systems.

## 6.2 Performance Metrics

In order to comprehensively assess the outcomes of experiments, a set of performance metrics was employed, including FLOPS, bandwidth, execution time and speedup.

The formulas for calculating FLOPS and bandwidth (GB/s) were defined as:

$$\text{FLOPS} = \frac{\text{VOLUME} \cdot \text{Flop\_site} \cdot \text{MPI\_RankSize}}{\text{elapsed time}} \tag{6.1}$$

$$\text{Bandwidth} = \frac{\text{VOLUME} \cdot \text{Memory\_site} \cdot \text{MPI\_RankSize}}{\text{elapsed time}} \tag{6.2}$$

where `VOLUME` represents the number of sites (bulk and boundary) on a local lattice block, `Flop_site` and `Memory_site` are calculated using the respective formulas defined in Eq. (3.2) and Eq. (3.3). `MPI_RankSize` represents the number of MPI tasks in a given run, and `elapsed time` is the run-time difference between end and start of a program which is measured in seconds. To accurately measure the `elapsed time`, the routines run for a minimum of two seconds. Therefore, the Dirac operator was run a number of times such that the elapsed time exceeds the threshold of 2 seconds.

Last, the speedup is defined based on execution time as:

$$\text{Speedup} = \left( \frac{\text{Elapsed Time of Baseline Routine}}{\text{Elapsed Time of the Optimised Routine}} \right) \tag{6.3}$$

## 6.3 Performance Evaluation

Due to the diverse factors affecting performance, such as problem sizes, MPI/OpenMP configurations, optimization strategies, hardware architectures, and hardware resources (including nodes, sockets, and CPU cores), a brief performance evaluation on Foseres and HIPPO is initially presented, followed by a more detailed evaluation.

### 6.3.1 Brief Evaluation on Foseres and HIPPO

In this section, a first/brief evaluation of the proposed optimizations studied in Chapter 5 is provided (Figures 6.3 - 6.4). The following cases were investigated:

- `ALL-OPTS` - the implementation with all optimization strategies; vectorization is implemented using AVX2 intrinsics,

- ALL-OPTS-CL - the implementation with all optimization strategies; vectorization is implemented with Clang vector intrinsics,
- NVEC-HLCPB - the hybrid implementation with loop collapse (Algorithm 3) and path-blocking (Figure 5.2) without vectorization, and
- ORIG - the original (MPI only) implementation (Algorithm 2): one MPI process is assigned to each physical core.

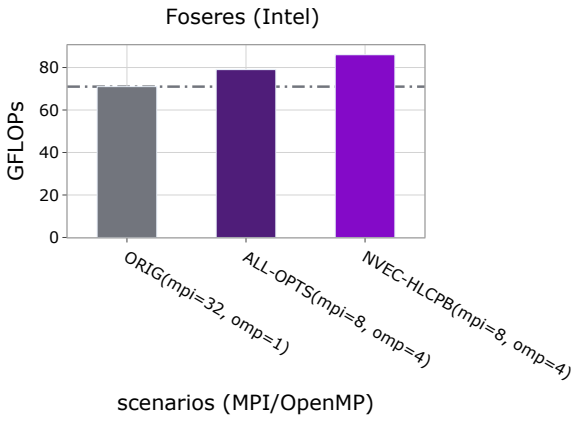



*Figure 6.3:* Evaluation on a single node within Foseres.

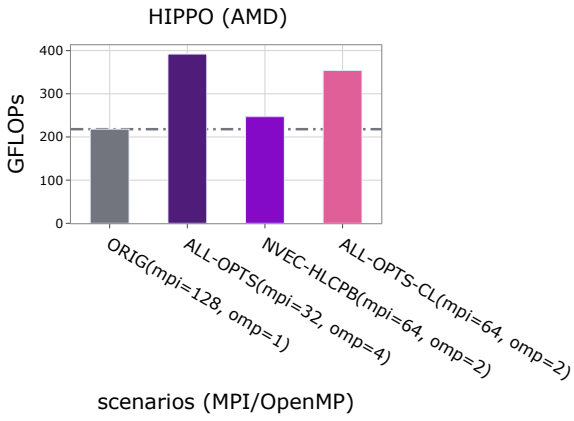



*Figure 6.4:* Evaluation on a single node within HIPPO.

On Foseres, as shown in Figure 6.3, the ALL-OPTS achieves a performance of 79 GFLOPs, surpassing the ORIG counterpart, which attains 71 GFLOPs. The speedup of ALL-OPTS over ORIG is calculated at approximately $\times 1.09$; attributed to the incorporation of hybridization, loop collapsed, data access pattern (path-blocking), and vectorization optimization strategies. This denotes that ALL-OPTS completes the computation $\times 1.09$ times faster than ORIG, indicating enhanced computational efficiency.

While the ALL-OPTS implementation of Dphi achieves better speedup compared to ORIG, it still performs slower than the NVEC-HLCPB implementation.

A comparable analysis was also carried out on HIPPO, with the results presented in Figure 6.4. The performance figures for ORIG, ALL-OPTS, and NVEC-HLCPB stand at 218 GFLOPs, 392 GFLOPs, and 247 GFLOPs, respectively. Consequently, a $\times 1.80$ speedup is achieved from ORIG to ALL-OPTS, whereas the speedup from ORIG to NVEC-HLCPB is only $\times 1.11$. Thus, ALL-OPTS outperforms both ORIG and NVEC-HLCPB. This indicates that the AVX2

vectorized implementation is faster on HIPPO (details are discussed in Section 6.3.5).

Considering the peak theoretical performance on HIPPO is 2048 GFLOPs, the `ALL-OPTS` implementation operates at approximately $\times 5.22$ slower than the peak performance.

Regarding the performance comparison between Clang vector and AVX2 implementations in Figure 6.4, `ALL-OPTS-CL` obtained 354 GFLOPs. Therefore, `ALL-OPTS-CL` achieves $\times 1.63$ speedup over `ORIG`, whereas `ALL-OPTS` gains $\times 1.80$. Thus, while both `ALL-OPTS` and `ALL-OPTS-CL` outperform `ORIG`, `ALL-OPTS-CL` exhibits a smaller speedup improvement than `ALL-OPTS` (discussed in detail in Section 6.3.5).

### 6.3.2 Evaluation of Optimizations in Sections 5.1 and 5.2 on Foseres

In this Section, the optimizations proposed in Sections 5.1 and 5.2 are evaluated in detail (Figure 6.5). To this end, the following two implementations are compared for various MPI/OpenMP configurations:

- `ORIG` - the original (MPI only) implementation (Algorithm 2): one MPI process is assigned to each physical core, and
- `HLC` - the implementation optimized with MPI+OpenMP (Hybrid) and Loop Collapse (Algorithm 3) strategies: configured with varying MPI/OpenMP settings.

For a fair comparison, it is important to note that all MPI/OpenMP configurations utilize precisely 64 threads distributed over two nodes. This ensures that exactly one thread runs on each CPU core at all times.

For a fixed problem size of $64 \times 16^3$, a maximum $\times 1.35$ speedup is achieved by `HLC(mpi=4, omp=16)` over `ORIG(mpi=64, omp=1)`. This improvement is attributed to the efficient use of shared memory by OpenMP threads, which facilitates faster local memory access. The threads utilize their local L1, L2, and unified shared L3 caches within their respective NUMA (Non-Uniform Memory Access) domains, eliminating any NUMA violations and ensuring faster memory access. Additionally, this configuration reduces MPI communication overhead due to the smaller number of MPI processes.

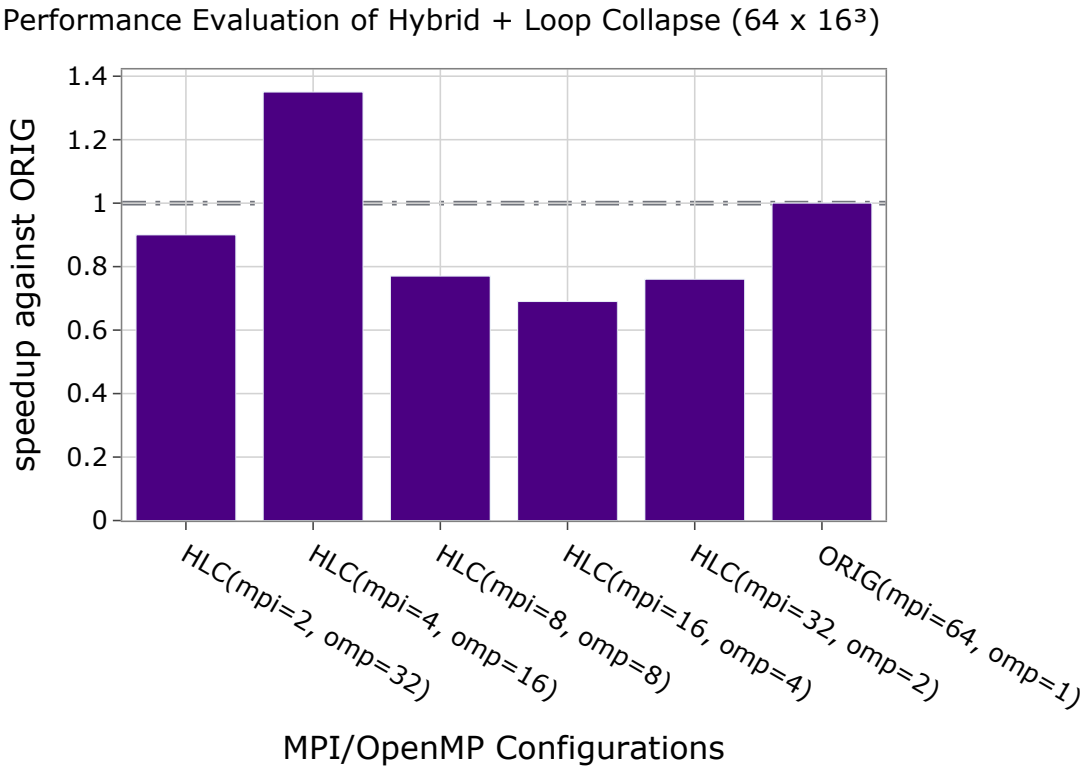



*Figure 6.5:* Performance evaluation of `Dphi` routine over two nodes in Foseres using various MPI/OpenMP configurations and a fixed problem size of $64 \times 16^3$, when applying the optimizations in Sections 5.1 and 5.2. Therefore, two nodes collectively have 64 cores.

However, depending on the hardware architecture, different MPI/OpenMP configurations exhibit varying levels of performance. HiRep code is designed to run multiple configurations, enabling us to identify the most suitable configuration for a given machine. On the Foseres platform, `HLC(mpi=4, omp=16)` has been found as the optimal configuration for achieving the best performance.

Nonetheless, the `ORIG(mpi=64, omp=1)` configuration, where each MPI process runs on a single CPU core, experiences slow data access to the main memory. This configuration incurs communication overhead due to the high volume of data exchange among the 64 MPI processes.

### 6.3.3 Evaluation of Optimizations in Section 5.3 on Foseres

To optimize memory access patterns, the path-blocking optimization strategy was applied to both the baseline and hybrid (MPI+OpenMP) implementations with loop collapse, as detailed

in Section 5.3. Then, a detailed investigation was conducted into three distinct cases:

- `ORIG` - the original (MPI only) implementation (Algorithm 2): one MPI process is assigned to each physical core,
- `HLCPB` - the implementation optimized with MPI+OpenMP (hybrid), loop collapse (Algorithm 3) and memory access patterns (path-blocking) (Figure 5.2), and
- `ORIGPB` - the original implementation (ORIG) (Algorithm 2) with the addition of path-blocking (Figure 5.2).

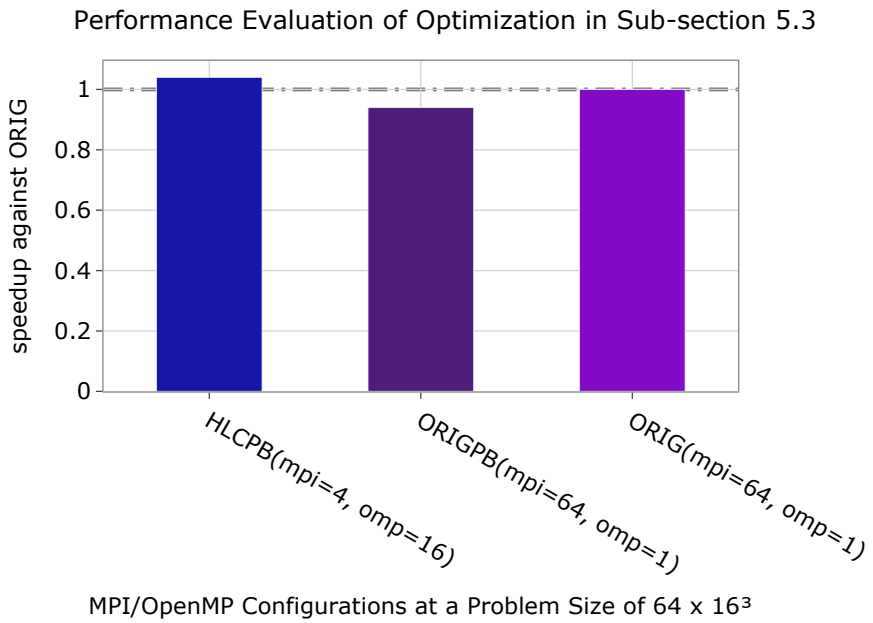



*Figure 6.6:* Performance evaluation of `Dphi` routine over two nodes on Foseres at a problem size of $64 \times 16^3$, when applying the optimizations in Section 5.3. Two nodes collectively possess 64 cores.

Similar to Section 6.3.2, in Figure 6.6, three MPI/OpenMP configurations utilize precisely 64 threads distributed over two nodes. I executed each configuration nine times on a problem size of $64 \times 16^3$, measuring elapsed time in seconds on a full node. Then I calculated the mean of elapsed time for each case and subsequently computed the speedup. `HLCPB(mpi=4, omp=16)` achieves a speedup of $\times 1.04$ over `ORIG(mpi=64, omp=1)` due to the change in memory access patterns and better cache utilization by OpenMP threads in an MPI process. Each MPI process can use multiple threads to execute the sub-blocks concurrently, taking ad-

vantage of shared memory parallelism. This can lead to improved cache utilization and reduced communication overhead.

However, the speedup of `ORIGPB(mpi=64, omp=1)` compared to `ORIG(mpi=64, omp=1)` decreased to $\times 0.94$. This reduction could be attributed to the compiler's auto-optimization options working more effectively with `ORIG` than with `ORIGPB`. Nonetheless, `ORIGPB` may perform better when combined with other optimizations, such as vectorization. This observation warrants further investigation in future studies.

### 6.3.4 Detailed Evaluation of All Optimizations on Foseres

To identify the best input size per MPI process, a thorough experimental evaluation/analysis is performed; To this end, the problem size is varied in different MPI/OpenMP configurations for each of the cases mentioned in Section 6.3.1, ensuring full utilization of all cores on a node. The measurements are performed on 1, 2, and 4 nodes for comprehensive evaluation, however, instead of providing the overall GFLOPs and bandwidth value, the GFLOPs and bandwidth per node in Figures 6.7-6.8 are provided.

Results are plotted with the y-axis representing GFLOPs or Bandwidth per node and the x-axis indicating the total data movement during the computation within the Dirac operator for various local problem sizes (lattice sizes). The local problem sizes are calculated by multiplying the value of `Memory_site` (Eq. 3.3) by the volume, which represents the total number of input and output sites.

The computational efficiency results of `Dphi` code are presented in Figure 6.7 which shows that `NVEC-HLCPB(mpi=8, omp=4)` configuration achieved the highest performance. This configuration utilized 8 MPI processes and 4 OpenMP threads on a problem size of $4 \times 8^3$ per MPI process, resulting in approximately 85 GFLOPs. In contrast, the `ALL-OPTS(mpi=8, omp=4)` with 8 MPI processes and 4 OpenMP threads attained around 79 GFLOPs, and `ORIG(mpi=32, omp=1)` achieved nearly 71 GFLOPs with 32 MPI processes and 1 OpenMP thread per MPI process. This configuration involves assigning exactly 1 MPI process with 1 OpenMP thread to each CPU core.

Compared to the case of ORIG(mpi=32, omp=1), ALL-OPTS(mpi=8, omp=4) and NVEC-HLCPB(mpi=8, omp=4) demonstrated speedups of $\times$1.09 and $\times$1.18, respectively (Figure 6.3). In this context, the ALL-OPTS exhibits faster performance compared to ORIG, but $\times$1.08 slower than NVEC-HLCPB.

While the utilization of AVX2 does not yield substantial performance improvement on Foseres, it is evident that this implementation leads to a considerable speedup on the new processor architecture in HIPPO, as substantiated in subsequent discussions.

In tandem with GFLOPs, the bandwidth depicted in Figure 6.8 exhibits analogous trends. Specifically, the NVEC-HLCPB(mpi=8, omp=4) scenario displayed superior performance, hovering at approximately 200 GB/s. In contrast, the ALL-OPTS(mpi=8, omp=4) configuration exhibited comparatively slower performance, measured at 184 GB/s. The ORIG(mpi=32, omp=1) shows 163 GB/s.

Based on these bandwidth measurements, ALL-OPTS(mpi=8, omp=4) and NVEC-HLCPB (mpi=8, omp=4) obtained $\times$1.12 and $\times$1.23 speedups respectively over the original version of Dphi code (ORIG). Similarly, in bandwidth measurements, ALL-OPTS exhibited a speedup $\times$1.11 slower than NVEC-HLCPB.

Figures 6.7 and 6.8 reveal a gradual decline in GFLOPs and bandwidth as data movement or lattice size increases. This indicates that while performance remains satisfactory for smaller data movements, it diminishes with larger data volumes. This performance reduction is primarily due to the inability of larger data sets to fit into caches, resulting in the reliance on the slower DDR bandwidth for data transfers.

Furthermore, it is found that the weak scalability of the ALL-OPTS version of Dphi is suboptimal on Foseres. Performance declines as the number of nodes, MPI processes, and problem size increase. This issue will be discussed in greater detail in Section 6.3.5.

In summary, ALL-OPTS outperforms the ORIG, achieving a speedup of $\times$1.09 in the Foseres environment. Moreover, selecting smaller problem size per MPI process demonstrates efficient utilization of the L1, L2 and shared L3 caches.

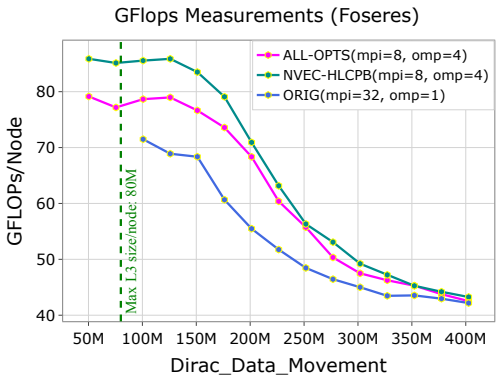



*Figure 6.7:* GFLOPS comparison with ORIG on Foseres. "M" on the x-axis denotes megabytes (MB).

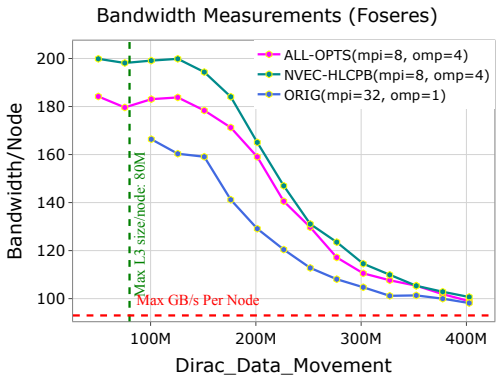



*Figure 6.8:* GB/s comparison with ORIG on Foseres. "M" on the x-axis denotes megabytes (MB).

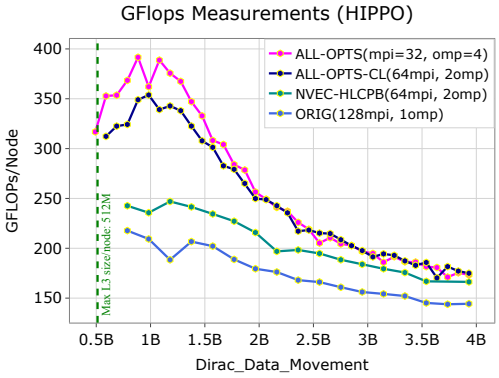



*Figure 6.9:* GFlops comparison with ORIG on HIPPO. "B" on the x-axis denotes gigabytes (GB).

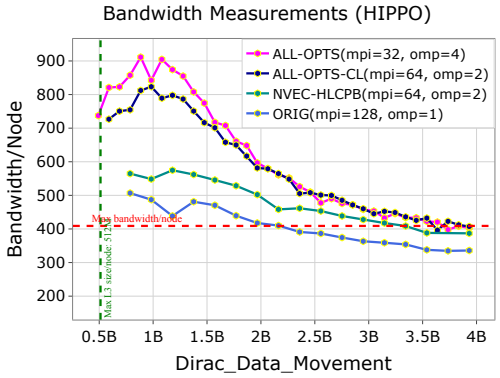



*Figure 6.10:* GB/s comparison with ORIG on HIPPO. "B" on the x-axis denotes gigabytes (GB).

*Figure 6.11:* Comparing computational capability and bandwidth of ALL-OPTS and ALL-OPTS-CL with ORIG and NVEC-HLCPB on Foseres and HIPPO.

### 6.3.5 Detailed Evaluation of All Optimizations on HIPPO

On HIPPO platform, selected results obtained on 1, 2, 4, and 8 nodes are presented in Figures 6.9-6.10 to highlight the performance difference among various cases. `ALL-OPTS` is the fastest among all the cases on HIPPO. Similar to Foseres, the results in HIPPO were collected from a single MPI process on a single node and normalized by multiplying with the number of MPI processes and dividing by the number of nodes to ensure fair performance comparison across different configurations.

In Figures 6.9-6.10, the x-axes denote data movement, while the y-axes represent computational efficiency and data transfer speed of `Dphi`. The performance trends illustrate initial high performance levels across all scenarios. However, as the local problem size escalates, a plateau in performance emerges, indicating that the problem becomes DDR bound. Consequently, the data exceeds the capacity of the L1, L2, and L3 caches, impeding further performance gains.

In terms of GFLOPs measurements, Figure 6.9 shows that `ALL-OPTS(mpi=32, omp=4)` (utilizing 32 MPI processes and 4 OpenMP threads) attains the highest performance at around 391 GFLOPs with a problem size of $(9 \times 10^3)$ per process. In comparison, `NVEC-HLCPB(mpi=64, omp=2)` (employing 64 MPI processes with 2 OpenMP threads) demonstrates performance, reaching around 246 GFLOPs on a local problem size of $(4 \times 10^3)$. Meanwhile, on the same problem size, `ORIG(mpi=128, omp=1)` achieves 217 GFLOPs.

According to the results on HIPPO depicted in Figure 6.9, `ALL-OPTS(mpi=32, omp=4)` and `NVEC-HLCPB(mpi=64, omp=2)` showcase speedups of approximately $\times 1.80$ and $\times 1.11$ (Figure 6.4) respectively compared to the speedup of `ORIG(mpi=128, omp=1)`. Significantly, `ALL-OPTS` outperforms `NVEC-HLCPB` by $\times 1.62$.

In HIPPO, `ALL-OPTS` outperforms all other cases, specifically achieving a speedup of $\times 1.80$ compared to `ORIG(mpi=128, omp=1)`. Conversely, in Foseres, `ALL-OPTS` achieves a modest speedup of $\times 1.09$ (Figure 6.3).

The code was optimized not only with AVX2 SIMD intrinsics but also with Clang vector intrinsics. However, the implementation utilizing SIMD AVX2 outperforms the Clang vector intrinsics. In Figure 6.9, it is observed that `ALL-OPTS(mpi=32, omp=4)` achieved the highest performance of 391 GFLOPs, while `ALL-OPTS-CL(mpi=64, omp=4)` obtained 352 GFLOPs. Consequently, `ALL-OPTS` code obtained $\times 1.11$ speedup over Clang vector code. This is because AVX2 intrinsics in `ALL-OPTS` map directly to specific hardware instructions such as Intel or AMD CPUs, allowing for highly efficient execution that fully leverages the capabilities of the CPU (Intel Corporation 2024*f*). Clang vector extensions in `ALL-OPTS-CL`, on the other hand, are a higher-level abstraction that can target various SIMD instruction sets across different architectures (Clang Documentation 2024). While they provide portability

and ease of use, they are not always as finely tuned to the specific hardware as AVX2 intrinsics (Reinders et al. 2017).

In Figure 6.10, the bandwidth of code optimized with Clang vector intrinsics was also measured and found that `ALL-OPTS-CL(mpi=64, omp=4)` achieved the bandwidth of 820 GB/s. However, the `ALL-OPTS(mpi=32, omp=4)` performed the peak bandwidth of 911 GB/s. The `ALL-OPTS` gains $\times 1.11$ speedup over Clang vector intrinsics. It is noted that the bandwidth speedup ratio of `ALL-OPTS` is identical to the GFLOPs speedup.

Given that the FLOPs and memory per site are fixed, it is observed that bandwidth and GFLOPs are closely related by a rescaling factor, resulting in similar trends in performance improvements.

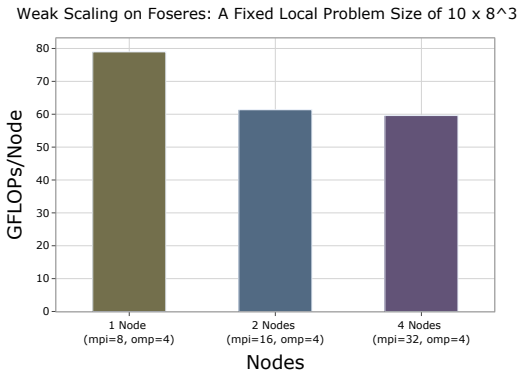



*Figure 6.12:* Scaling Evaluation on Foseres (Intel).

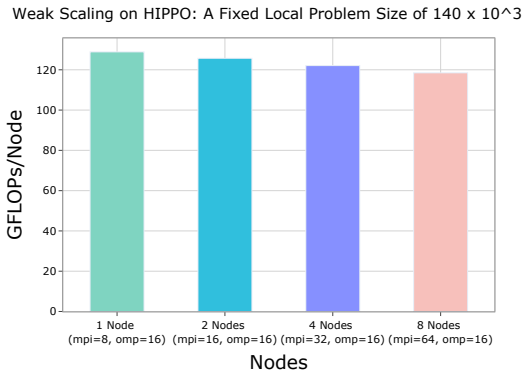



*Figure 6.13:* Scaling Evaluation on HIPPO (AMD).

*Figure 6.14:* Weak scaling evaluation on Foseres (Intel) and HIPPO (AMD), involving an exploration of scalability by increasing the number of nodes, MPI processes, and global problem size while keeping a consistent number of MPI processes per node and a fixed problem size per MPI process. The fixed local problem sizes in Foseres and HIPPO are $10 \times 8^3$ and $140 \times 10^3$ respectively.

The scalability of the `ALL-OPTS` code was compared between the HIPPO and Foseres platforms. The weak scalability of the fully optimized code (`ALL-OPTS`) was tested by increasing the number of nodes, MPI processes, and the global problem size, ensuring that each node had an equal number of MPI processes and OpenMP threads, utilizing all the cores of each node. The results are presented in Figures 6.12 and 6.13. The problem size was also kept identical on each MPI process per node while the global problem size was increased.

It was found that the scalability of the `ALL-OPTS` version of `Dphi` is better in HIPPO than in Foseres. A dramatic performance drop is observed in Foseres as the number of nodes increases, whereas the HIPPO results remain stable despite an increase in the number of nodes. This is because HIPPO's high performance interconnection network - InfiniBand - holds the performance advantage against Foseres's Omni-Path across clusters.

# Chapter 7

# Related work

***In this chapter**, I have provided an overview of simulation suites available alongside `HiRep`, including `Grid`, `openQCD`, `MILC`, and `QUDA`. Comparative analyses of these simulation software packages have been presented. Additionally, I have compared the optimization methodologies applied in HiRep with those employed in other lattice simulations.*

HiRep is not the only simulation suite available; although a comprehensive list is not in the scope of this section, the community has produced a few of other suites. All the implementations are usually characterized by some technical and algorithmic differences, such as the actions implemented or the solver's algorithm. However, overall, the community has always put a huge effort into the efficiency of these software packages. Among the most notable implementations, `Grid` (Boyle et al. 2015), `openQCD` (Lüscher 2024), `MILC` (MILC Collaboration 2016), and `QUDA` (Clark et al. 2010) should be listed.

All of these codes, including `HiRep`, are designed for lattice simulations, suggesting commonalities in their approach to managing data layout on the lattice. This includes the storage of gauge fields, fermion fields, and other pertinent quantities.

`HiRep` and `openQCD` support only simulations employing Wilson fermions, albeit with different enhancements. Conversely, `MILC` offers staggered and clovered fermions. The capabilities of `QUDA` extend to supporting various fermion formulations, including Wilson fermions, Clover-improved Wilson fermions, Twisted mass fermions (both degenerate and non-degenerate variants), Twisted mass fermions with a clover term, Staggered fermions, Improved staggered fermions (such as asqtad or HISQ variants), Domain wall fermions (both 4-dimensional and 5-dimensional preconditioned) and Mobius fermions. Similarly, `GRID` is equipped to support

all major fermion formulations, including Wilson/clover, domain wall, overlap, twisted mass, and staggered (Daniel Richtmann 2018).

While GRID, openQCD, MILC, and HiRep were originally developed for CPU-based computations, efforts have been made to extend their capabilities to accelerate computations on GPU architectures. QUDA, on the other hand, is specifically designed to accelerate computations, particularly on GPU architectures.

In the CPU implementations of openQCD, MILC, and HiRep, the data layout follows a geometrical structure, meaning that the arrangement of data is influenced by the lattice geometry. Conversely, in GRID, the data layout is driven by a thread vectorized approach, which prioritizes data vectorization and job scheduling optimizations to maximize computational efficiency.

In the GPU-based implementations of these codes, the data layout is strategically designed to achieve optimal coalesced memory access, a key aspect in GPU computing performance. This involves organizing data in memory such that consecutive threads access consecutive memory locations, thereby enhancing memory access efficiency and overall performance on GPU architectures.

While GRID, openQCD, MILC, and QUDA were designed for simulations related to QCD-like theories, HiRep offers a broader scope by accommodating simulations not only of QCD-like theories but also a diverse range of theories pertaining to BSM (Beyond Standard Model). This expanded capability of HiRep enables researchers to explore a wider spectrum of theoretical frameworks and phenomena beyond the confines of conventional QCD simulations.

The efficiency of these codes is the subject of continuous discussion in the community, and continuous effort is spent on both implementation and algorithm development. At the yearly International Symposium of Lattice Gauge Theories (Lattice), an entire parallel session is usually devoted to “Algorithms and Machines”. The effort of the community is globally recognized, and the codes are often used as benchmarking applications to test HPC installations and complement the information coming from more traditional benchmark tools.

In the case of HiRep, two benchmarking suites have been derived from it over the years, BSM-

Bench (Bennett et al. 2016), and Sombrero (Mesiti, M. and Bennett E. 2022). One of the key features of `HiRep`, i.e., the possibility of efficiently simulating different gauge groups and fermion representations, makes it capable of changing the relative weights of computational pressure versus data movement, offering a distinctive metric for investigation of HPC hardware.

Furthermore, there are few studies related to code optimization methodology. For example, Pal et al. (2014) have conducted analogous investigations into parallelizing molecular dynamics programs across both shared memory and distributed memory architectures. Utilizing unthreaded MPI processes prove inefficient when compared to threaded MPI processes. The findings indicate that a coupled scheme (MPI/OpenMP) can achieve nearly twice the speed of a pure MPI based implementation for specific system sizes. This advantage arises from circumventing additional overheads present in the latter scheme.

The study conducted by Kunaseth et al. (2013) demonstrates that highly parallel molecular dynamics (MD) benchmarks have yielded substantial performance improvements through hybrid MPI/OpenMP parallelization, particularly for fine-grain large-scale applications. Notably, a speedup $\times 2.58$ was observed for a simulation involving 0.8 million particles on 32,768 cores of BlueGene/P.

Likewise, the research work by Smelyanskiy et al. (2011) on the lattice QCD Wilson-Dslash algorithm illustrates the effectiveness of hybrid programming models. In their implementation, the problem is distributed across nodes using MPI processes, with each MPI process employing OpenMP threads internally to fully utilize the cores and cache. They employ the OpenMP model across sockets within a node and MPI across nodes, achieving a 10% performance improvement over using MPI across sockets within a node.

A critical factor influencing performance is the nature of memory accesses and the memory layout of the data structures. The path-blocking memory access pattern optimization in HiRep has increased L3 cache reuse. Smelyanskiy et al. (2011) implemented 3.5D and 4.5D blocking techniques to maximize the reuse of both spinors and gauges. This study focuses on the Wilson-Dslash operator, whose key operation is a matrix-vector product.

The optimization of Lattice Gauge Theory simulations using Streaming SIMD Extensions (SSE)

was investigated by Srinivasan (2013). Their study demonstrated that SSE provided higher speed-ups for single-precision compared to double-precision floating-point numbers. While the use of SSE significantly reduced simulation time, it did not achieve the theoretical maximum performance. In contrast, my research employs AVX2 to enhance the performance of HiRep.

This chapter has reviewed several notable lattice simulation suites, including `HiRep`, `Grid`, `openQCD`, `MILC`, and `QUDA`, highlighting their technical and algorithmic differences. `HiRep` is characterised by a design that provides built-in support for simulations beyond QCD-like theories. The optimization methodology employed in these suites, particularly the hybrid MPI-OpenMP approach and advanced memory access patterns, demonstrate significant performance enhancements. Additionally, this research utilizes AVX2 to improve the performance of `HiRep`, building on the foundational work of SIMD extensions like SSE, which have shown notable but limited speed-ups in previous studies.

# Chapter 8

# Summary and Discussion

> ***In this chapter**, a concise overview of the research process is presented. Initially, the steps undertaken in this PhD study are outlined: identifying performance inefficiencies in the `Dphi` routine, developing methodologies to address these inefficiencies, and conducting experiments to evaluate and present the results. Following this, the chapter delves into the challenges encountered during the implementation of these optimization methodologies and provides an interpretation of some experimental results.*

IN THIS PhD research work, the performance optimization efforts are presented for the Dirac operator in HiRep simulation software. This was achieved by optimizing `Dphi` routine through implementing different optimization strategies against various performance inefficiencies. The following sections provide a summary of the entire research, followed by a discussion of key aspects and findings.

## 8.1 Summary

`HiRep` was initially parallelized using the MPI programming framework, distributing a large lattice equally among MPI processes. Each MPI process managed its local lattice segment. However, upon studying, analyzing, and profiling the `HiRep` application, it was identified that the baseline implementation suffered from significant communication overhead due to extensive data exchanges between MPI processes. To reduce this overhead, a hybrid (MPI-OpenMP) parallel programming model was applied, utilizing fewer MPI processes and leveraging the shared L3 cache memory among OpenMP threads.

The next inefficiency identified was within the OpenMP regions, specifically when computing

boundary sites located in the send buffers, due to the uneven distribution of data among OpenMP regions in the boundary. To resolve this issue, OpenMP parallelism in hybrid code was optimized by changing the loop structure, collapsing two loops into a single loop. Boundary sites were processed in a single OpenMP region as a unified send buffer instead of multiple send buffers or OpenMP regions. This hybrid loop collapse optimization yielded a $\times 1.35$ speedup improvement compared to the MPI-only code.

Subsequently, non-contiguous memory access patterns were identified as causing numerous L3 cache misses. To increase L3 cache hits and enhance data reuse, the bulk segment of the local lattice was sub-divided, and memory was accessed in a block-wise manner. This optimization achieved a $\times 1.18$ speedup improvement over the MPI-only implementation.

Furthermore, it was experimentally found that auto-vectorization was inefficiently applied in MVM operations. Instead of relying on inefficient compiler auto-vectorization, MVM operations were manually vectorized using AVX2 intrinsics and Clang vector intrinsics. The combination of all these optimizations resulted in a $\times 1.80$ speedup over the MPI-only version of the implementation. It is worth noting that the vectorization using AVX2 intrinsics outperformed the implementation using Clang intrinsics.

## 8.2 Discussion

Both the engineering and research efforts in this PhD study were significant and non-trivial. Multiple aspects of these efforts and the significance of the experimental results are discussed below:

In the case of hybridizing the code in `HiRep`, research was conducted to determine the most appropriate MPI thread support options. The `MPI_THREAD_FUNNELED` style, where all MPI calls are made by the OpenMP master thread (Aoyama et al. 2023), was found to be the best option for `HiRep`. This approach allows for the synchronization of threads before and after message transfers without the need to repeatedly close and open parallel regions. One advantage of this method is that other threads can perform useful computations while the master thread executes MPI calls.

A study was also conducted to identify which loops should be parallelized and which OpenMP clauses would be most efficient. Additionally, optimal and efficient AVX2 and Clang vector intrinsics (SIMD instructions) were studied and implemented. Significant effort was invested in building scripts to generate thousands of test cases, running the executable, and selecting the best cases after analyzing the results with Plotly and other Python packages.

In the original `Dphi`, thread creation overhead was observed, as threads were not persistent and were switched on and off as frequently as the loop kernel ran inside the `Dphi` routine. To eliminate this overhead, `Dphi` was invoked from within the parallel region, rather than launching a parallel region inside `Dphi`.

Upon examining the code, it was ensured that the iterations within the loops responsible for updating bulk and boundary sites of the lattice were evenly distributed among the OpenMP threads within each MPI process, optimizing performance in the parallel computation of the sites. In order to ensure uniform distribution of equally sized chunks, the OpenMP parallel for loop uses the `static` approach, a scheduling policy that efficiently assigns these iteration chunks to the available threads (RookieHPC 2023), thus balancing workloads.

MPI processes were mapped to different nodes while OpenMP threads were bound to specific cores within a node. MPI process affinity was managed through `hwloc`, which gathers hardware information about processors, caches, memory nodes, and more, and exposes it to applications and runtime systems in an abstracted and portable hierarchical manner (Broquedis et al. 2010). The OpenMP thread affinity is to affix threads to CPU cores in a manner that optimizes memory accesses to shared data among threads while achieving balanced core utilization across all CPU cores, thus avoiding resource over-subscription (Diener et al. 2016; Li et al. 2023).

Alongside the utilization of thread-core affinity, the `"first-touch"` policy was implemented. Memory affinity is not decided by the memory allocation, but by the initialization. This is called the `"first-touch"` policy. Parallel first-touch initialization was employed on ccNUMA nodes, as described in Hager et al. (2009). In this context, the spinor and gauge fields were allocated by the master thread but initialized by all threads. Each thread initialized the memory and subsequently utilized it, contributing to an optimized memory access pattern.

Performance optimization was also studied using various compilers, such as the Intel Compiler (icc), GCC, and Clang compilers. Additionally, different MPI implementations, such as MPICH, OpenMPI, and Intel® MPI, were used to assess any performance differences. The discussion so far indicates that achieving performance improvement was a complex task, involving the management of multiple influential factors.

Throughout this implementation and evaluation process, it was observed that the optimized code's performance and scalability are architecture-dependent. The results indicate that while the `ALL_OPTS` version of the code is less efficient than `ORIG` on the Foseres environment, it performs faster on the HIPPO environment. Similarly, it is evident that `ALL_OPTS` achieved higher scalability in HIPPO than in Foseres. This suggests that the underlying hardware architecture significantly influences the performance of the code. Hardware resources affecting performance include the bandwidth of the MPI network interconnect, memory subsystems, cc-NUMA architectures (i.e., CPU core topology), and the size of different levels of caches.

This chapter summarized the steps taken to optimize `HiRep`, including identifying performance inefficiencies, implementing optimization methodologies, and discussing experimental results. Significant performance gains were achieved through all optimizations (hybrid MPI-OpenMP model, loop collapse optimization, improved memory access patterns, and manual vectorization). The discussion highlighted the challenges and solutions in optimizing `HiRep`, emphasizing the importance of hardware architecture in influencing performance.

# Chapter 9

# Conclusions and Further Work

***This chapter*** *provides a comprehensive discussion of the main contributions to knowledge resulting from this research. It highlights the innovative aspects of the work, showcasing the approaches and methodologies developed and their impact on the field. Additionally, it outlines the directions for future research, suggesting areas where further investigation and development could build on the findings of this study. The discussion emphasizes the significance of the contributions in advancing the state of the art in High-Performance Computing (HPC) and optimizing the HiRep lattice simulations software.*

THIS PROJECT presents two distinct optimization strategies: algorithmic optimizations and hardware-dependent compiler optimizations. Within algorithmic optimizations, the Dirac operator was hybridized by integrating MPI with OpenMP threads, and loop collapse was implemented to improve the efficiency of OpenMP parallelism. Then, the memory access patterns were changed from lexicographic ordering to block-lexicographic ordering, known as path-blocking. These combined optimization strategies enhanced the performance of the HiRep simulation software. Concurrently, in the domain of hardware-dependent compiler optimization, the vectorization of Dirac operator was pursued utilizing the SIMD model, specifically employing AVX2 (256-bit register) vector instructions.

## 9.1 Contributions to Knowledge

Overall, the aim and objectives outlined in Chapter 1 have been successfully achieved through this research. The core contribution of this project lies in the synergistic application of various optimization strategies, resulting in compelling outcomes. Notably, these efforts have led to $\times 1.80$ overall speedup compared to the original implementation of `Dphi`, which relied solely

on MPI parallelization. The project establishes the following main contributions:

- The performance analysis of the HiRep simulation software has been conducted. This entailed a meticulous examination of the `Dphi` to identify performance bottlenecks that hinder efficient computation. The analysis was carried out using advanced profiling tools and techniques to systematically pinpoint areas where computational resources were not being optimally utilized. This in-depth performance analysis laid the foundation for targeted optimization efforts. By thoroughly understanding the performance dynamics of HiRep, the research was able to devise and implement strategies that directly addressed the identified bottlenecks, leading to improvements in computational efficiency. This contribution not only enhanced HiRep's performance but also provided a methodological framework for performance analysis applicable to other high-performance computing applications.

- A comprehensive optimization methodology for HiRep has been established. This methodology integrates multiple optimization strategies to address various performance inefficiencies systematically. Key strategies include the hybridization of MPI with OpenMP to leverage shared memory parallelism and reduce communication overhead, the application of loop collapse to enhance the efficiency of boundary site computations, and the implementation of path-blocking techniques to improve data locality and cache utilization. Additionally, the use of hand-tuned AVX2 and Clang vector intrinsics has been employed to achieve fine-grained control over vector operations, bypassing the limitations of compiler auto-vectorization. These strategies collectively aim to maximize the computational efficiency and scalability of the HiRep simulation software across different hardware architectures, demonstrating an advancement in optimizing complex scientific codes.

- A series of experiments has been modeled and performed to validate the proposed optimization methodology, demonstrating substantial performance improvements on two distinct computer clusters. This contribution involves a meticulous experimental procedure where the optimized HiRep code was rigorously tested on the Foseres and HIPPO

environments. By benchmarking the performance across different configurations, such as varying numbers of MPI processes and OpenMP threads, the experiments showcased the efficacy of the hybrid parallelization strategy, loop collapse optimizations, and path-blocking techniques. The results indicated significant reductions in communication overhead, improved data locality, and enhanced cache utilization, leading to marked speedups compared to the baseline MPI-only implementation. Implementing manual vectorization with AVX2, along with other optimizations, resulted in a significant performance improvement. This experimental validation not only underscores the robustness of the optimization strategies but also provides empirical evidence of their applicability and scalability across diverse high-performance computing architectures.

Furthermore, several papers associated with this research have been presented and published in refereed conferences and journals (Appendix-A). Consequently, the project has made positive contributions to High Performance Computing (HPC), particularly in the optimization of the HiRep lattice simulation software package.

## 9.2 Future Work Suggestions

1. While the present optimization efforts have yielded significant performance gains, further investigation is required to understand the inefficiency of AVX2 on the Foseres platform. This involves conducting a detailed analysis to identify the root causes of the suboptimal performance. Potential areas of focus include assessing the impact of the platform's specific hardware and microarchitecture characteristics on AVX2 instructions. Understanding these factors will be crucial for developing strategies to enhance AVX2 performance and, consequently, the overall computational efficiency on the Foseres platform.

2. As is observed that the speedup of `ORIG` is higher than `ORIGPB` in the MPI-only case. The performance degradation of path-blocking optimization could be attributed to the compiler's auto-optimization options working more effectively with `ORIG` than with `ORIGPB`. This observation warrants further investigation in future studies.

3. The current study evaluated the performance on two different hardware platforms. To bet-

ter understand the influences of underlying hardware, future work will involve evaluating performance on several additional hardware platforms.

4. Although the present optimization efforts have resulted in significant performance gains, exploring the implementation of GPU optimization (Ryoo et al. 2008; Hijma et al. 2023) stands as a viable avenue for achieving additional speedup. This avenue of investigation holds great promise for further elevating the capabilities of HiRep and its applicability in the realm of lattice simulations.

# List of references


Ali, H. M., Hamza, M. Y. and Rashid, T. A. (2023), Exploring polymorphism: Flexibility and code reusability in object-oriented programming, *in* 'Proceedings of the 4th International Conference on Recent Innovation in Engineering', ICRIE, p. 33.

AMD (2021), 'Amd epyc 7713 processor', https://www.amd.com/en/products/cpu/amd-epyc-7713. Accessed on 18-07-2023.

Aoyama, T., Kanamori, I., Kanaya, K., Matsufuru, H. and Namekawa, Y. (2023), 'Bridge++ 2.0: Benchmark results on supercomputer fugaku', *arXiv preprint arXiv:2303.05883* .

ARM Developer (2024), 'Introducing neon', https://developer.arm.com/documentation/dht0002/a/Introducing-NEON. Accessed: 29-06-2024.

Arora, K., Barajas, C. and Gobbert, M. K. (2018), 'Parallel performance studies for an elliptic test problem on the stampede2 cluster and comparison of networks', *UMBC Information Systems Department* .

Asanović, K. and Patterson, D. A. (2014), The landscape of parallel computing research: A view from berkeley, Technical Report UCB/EECS-2006-183, University of California, Berkeley.

Bennett, E., Lucini, B., Del Debbio, L., Jordan, K., Patella, A., Pica, C. and Rago, A. (2016), Bsmbench: A flexible and scalable hpc benchmark from beyond the standard model physics, *in* '2016 International Conference on High Performance Computing & Simulation (HPCS)', IEEE, pp. 834–839.

Boyle, P., Yamaguchi, A., Cossu, G. and Portelli, A. (2015), 'Grid: A next generation data parallel c++ qcd library'.

Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S. and Namyst, R. (2010), hwloc: A generic framework for managing hardware affinities in

hpc applications, *in* '2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing', IEEE, pp. 180–186.

Carnegie Mellon University (2009), 'Callgrind manual', https://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-40/Nice/RuleRefinement/bin/valgrind-3.2.0/docs/html/cg-manual.html.

Chandra, R., Dagum, L., Kohr, D. and Mayden, D. (2008), 'Parallel programming in openmp', *NY.: Morgan Kaufmann Publishers* .

Chandra, R., Menon, R., Dagum, L. and Kohr, D. (2001), 'D. maydan e j. mcdonald: Parallel programming in openmp'.

Cilk Plus Team (2021), 'Cilk plus - an extension to the c and c++ languages for parallel programming', https://web.archive.org/web/20210117031010/http://cilkplus.org/. Accessed: 18-05-2024.

Clang Documentation (2024), 'Vector builtins', https://clang.llvm.org/docs/LanguageExtensions.html#vector-builtin-functions. Clang Official Documentation.

Clark, M., Babich, R., Barros, K., Brower, R. and Rebbi, C. (2010), 'Solving lattice qcd systems of equations using mixed precision solvers on gpus', *Computer Physics Communications* **181**(9), 1517–1528.
**URL:** *http://dx.doi.org/10.1016/j.cpc.2010.05.002*

Creutz, M. (1979), 'Confinement and the Critical Dimensionality of Space-Time', *Phys. Rev. Lett.* **43**, 553–556. [Erratum: Phys.Rev.Lett. 43, 890 (1979)].

Daniel Richtmann, Peter A. Boyle, T. W. (2018), 'Multigrid for wilson clover fermions in grid'. arXiv preprint arXiv:1904.08678 [hep-lat].

Del Debbio, L., Patella, A. and Pica, C. (2010), 'Higher representations on the lattice: Numerical simulations, su (2) with adjoint fermions', *Physical Review D* **81**(9), 094503.

Diaz, J., Munoz-Caro, C. and Nino, A. (2012), 'A survey of parallel programming models and tools in the multi and many-core era', *IEEE Transactions on parallel and distributed systems* **23**(8), 1369–1386.

Diener, M., Cruz, E. H., Alves, M. A., Navaux, P. O. and Koren, I. (2016), 'Affinity-based thread and data mapping in shared memory systems', *ACM Computing Surveys (CSUR)* **49**(4), 1–38.

Dongarra, J. J., Luszczek, P. and Petitet, A. (2003), 'The linpack benchmark: past, present and future', *Concurrency and Computation: practice and experience* **15**(9), 803–820.

Frank Denneman (2016), 'Numa deep dive part 3: Cache coherency', http://www.staroceans.org/cache_coherency.htm. Star Oceans. Accessed on: 29-06-2024.

Gropp, W. (2024), 'Using MPI: Portable Parallel Programming with the Message-Passing Interface', https://wgropp.cs.illinois.edu/usingmpiweb/. Accessed on March 03, 2024.

Gropp, W., Lusk, E. and Skjellum, A. (1999), *Using MPI: portable parallel programming with the message-passing interface*, Vol. 1, MIT press.

Hager, G., Jost, G. and Rabenseifner, R. (2009), Communication characteristics and hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes, *in* 'Proceedings of Cray User Group Conference', Vol. 4, p. 5455.

Hager, G. and Wellein, G. (2010), *Introduction to high performance computing for scientists and engineers*, CRC Press.

Hennessy, J. L. and Patterson, D. A. (2011), *Computer architecture: a quantitative approach*, Elsevier.

Hijma, P., Heldens, S., Sclocco, A., Van Werkhoven, B. and Bal, H. E. (2023), 'Optimization techniques for gpu programming', *ACM Computing Surveys* **55**(11), 1–81.

Intel Corporation (2022*a*), 'Application performance profiling for intel xeon processors', https://www.intel.com/content/dam/support/us/en/documents/

processors/APP-for-Intel-Xeon-Processors.pdf. Accessed on 01-09-2023.

Intel Corporation (2022*b*), 'Intel xeon processor e5-2683 v4 specifications', https://www.intel.com/content/www/us/en/products/sku/91766/intel-xeon-processor-e52683-v4-40m-cache-2-10-ghz/specifications.html. Accessed on 18-07-2023.

Intel Corporation (2023), 'Intel hpc toolkit', https://www.intel.com/content/www/us/en/developer/tools/oneapi/hpc-toolkit.html. Accessed on 21-08-2023.

Intel Corporation (2024*a*), 'Intel Advisor User Guide', https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2024-0/analyze-cpu-roofline.html. Accessed on 25-03-2024.

Intel Corporation (2024*b*), 'Intel intrinsics guide', https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#. Accessed: 29-06-2024.

Intel Corporation (2024*c*), 'Intel oneAPI Advisor', https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html#gs.6jfpdp. Accessed on 25-03-2024.

Intel Corporation (2024*d*), 'Intel oneAPI Inspector', https://www.intel.com/content/www/us/en/developer/tools/oneapi/inspector.html#gs.6jeib5. Accessed on 25-03-2024.

Intel Corporation (2024*e*), 'Intel VTune Profiler User Guide', https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2024-0/introduction.html. Accessed on 25-03-2024.

Intel Corporation (2024*f*), 'Intel® avx2 and avx-512 architectures', https://software.intel.com/content/www/us/en/develop/documentation/

get-started-with-intel-advanced-vector-extensions/top.html. Intel Developer Zone, Accessed on : 25-03-2024.

Intel Corporation (2024*g*), 'Intel® toolkits', https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html. Accessed: 01-07-2024.

Jeffers, J. and Reinders, J. (2015), *High performance parallelism pearls volume two: multicore and many-core programming approaches*, Morgan Kaufmann.

Kaparelos, S. (2014), *Extending Cachegrind: L2 cache inclusion and TLB measuring*, (Department of Computer Science Technical Report Series; No. CSBU-2014-01). Department of Computer Science, University of Bath.

Kunaseth, M., Richards, D. F., Glosli, J. N., Kalia, R. K., Nakano, A. and Vashishta, P. (2013), 'Analysis of scalable data-privatization threading algorithms for hybrid mpi/openmp parallelization of molecular dynamics', *The Journal of Supercomputing* **66**, 406–430.

Li, X. S., Lin, P., Liu, Y. and Sao, P. (2023), 'Newly released capabilities in the distributed-memory superlu sparse direct solver', *ACM Transactions on Mathematical Software* **49**(1), 1–20.

LLVM (2023), 'Clang language extensions - vectors and extended vectors', https://clang.llvm.org/docs/LanguageExtensions.html#langext-vectors. Accessed on 31-08-2023.

Lüscher, M. (2024), 'Openqcd: Simulation programs for lattice qcd', https://luscher.web.cern.ch/luscher/openQCD/. Last updated on 04 January 2024. Accessed on: 13-04-2024.

Maleki, S., Gao, Y., Garzar, M. J., Wong, T., Padua, D. A. et al. (2011), An evaluation of vectorizing compilers, *in* '2011 International Conference on Parallel Architectures and Compilation Techniques', IEEE, pp. 372–382.

Margaret Rouse (2013), 'Cache miss', https://www.techopedia.com/definition/6308/cache-miss. Accessed on: 19-03-2024.

Mesiti, M. and Bennett E. (2022), 'sombrero: A Python package for XYZ', https://github.com/sa2c/sombrero. Accessed: 29/03/2024.

Microway (2021), 'Detailed specifications of the amd epyc milan cpus', https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-amd-epyc-milan-cpus/. Accessed on 2023-09-01.

MILC Collaboration (2016), 'MILC Collaboration: Lattice QCD Code', https://web.physics.utah.edu/~detar/milc/. Accessed on: 29/03/2024.

Nagarajan, V., Sorin, D. J., Hill, M. D. and Wood, D. A. (2020), *A primer on memory consistency and cache coherence*, Springer Nature.

Navaux, P. O. A., Lorenzon, A. F. and da Silva Serpa, M. (2023), 'Challenges in high-performance computing', *Journal of the Brazilian Computer Society* **29**(1), 51–62.

OMP ARB (2018), 'Openmp api specification version 5.0', https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Accessed on 18-07-2023.

Pal, A., Agarwala, A., Raha, S. and Bhattacharya, B. (2014), 'Performance metrics in a hybrid mpi–openmp based molecular dynamics simulation with short-range interactions', *Journal of Parallel and Distributed Computing* **74**(3), 2203–2214.

Phillips, G. (2023), 'What is cpu cache? everything you need to know', https://www.makeuseof.com/tag/what-is-cpu-cache/. Accessed on 19-03-2024.

PHP Documentation Team (2024), 'Php: pthreads - manual', https://www.php.net/manual/en/book.pthreads.php. Accessed: 18-05-2024.

Pica, C. (2008*a*), 'HiRep Developer Guide: Coding Conventions', https://claudiopica.github.io/HiRep/d4/d4d/md_developer_guide_

coding_conventions.html#coding_conventions. Last Accessed on 26/03/2024.

Pica, C. (2008*b*), 'HiRep Developer Guide: Data Structures', https://claudiopica.github.io/HiRep/d1/d16/md_developer_guide_data_structures.html#data_structures. Last Accessed on 27/03/2024.

Pica, C. (2023), 'HiRep repository — github.com', https://github.com/claudiopica/HiRep. Accessed on 21-08-2023.

Pica, C. and contributors (2007), 'Hirep github repository', https://github.com/claudiopica/HiRep/tree/HiRep-CUDA?tab=readme-ov-file. Accessed on 13-06-2024.

Polychroniou, O. and Ross, K. A. (2019), Towards practical vectorized analytical query engines, *in* 'Proceedings of the 15th International Workshop on Data Management on New Hardware', pp. 1–7.

Popovici, D. T., Franchetti, F. and Low, T. M. (2017), Mixed data layout kernels for vectorized complex arithmetic, *in* '2017 IEEE High Performance Extreme Computing Conference (HPEC)', IEEE, pp. 1–7.

Reinders, J., Ashbaugh, J., Gerosa, L. and Pennycook, S. (2017), *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, Morgan Kaufmann.

RookieHPC (2023), 'Openmp basics', https://rookiehpc.org/openmp/docs/static/index.html. Accessed on 09-10-2023.

Russ Ware (2023), 'L1 vs. l2 vs. l3 cache: What's the difference?', https://www.howtogeek.com/891526/l1-vs-l2-vs-l3-cache/. Accessed on 19-03-2024.

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, *in* 'Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming', pp. 73–82.

Silva, C., Vilaça, R., Pereira, A. and Bessa, R. (2024), 'A review on the decarbonization of high-performance computing centers', *Renewable and Sustainable Energy Reviews* **189**, 114019.

Smelyanskiy, M., Vaidyanathan, K., Choi, J., Joó, B., Chhugani, J., Clark, M. A. and Dubey, P. (2011), High-performance lattice qcd for multi-core based parallel systems using a cache-friendly hybrid threaded-mpi approach, *in* 'Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis', pp. 1–11.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J. (1998), *MPI: The complete reference*, MIT Press.

Srinivasan, S. (2013), 'Optimizing the performance of lattice gauge theory simulations with streaming simd extensions', *arXiv preprint arXiv:1309.0551* .

Sterling, T., Brodowicz, M. and Anderson, M. (2017), *High performance computing: modern systems and practices*, Morgan Kaufmann.

UoP (2022), 'Introduction to high-performance computing', https://ecm-research.plymouth.ac.uk/hpc/foseres-uop/user-guide/introduction/#hardware. Accessed: 31-03-2024.

UoP (2023), 'Introduction to high-performance computing', https://ecm-research.plymouth.ac.uk/hpc/foseres-uop/_downloads/d8491190cd2073690de460ebad3098fe/HPC_introduction_1.pdf. Accessed on: 31-03-2024.

UoSD (2023), 'Hpc type 3 documentation', https://docs.hpc-type3.sdu.dk/intro/hardware.html. Accessed on: 19-10-2023.

Van Dung, T., Taniguchi, I. and Tomiyama, H. (2014), Cache simulation for instruction set simulator qemu, *in* '2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing', IEEE, pp. 441–446.

Van Zee, F. G. and Smith, T. M. (2017), 'Implementing high-performance complex matrix multiplication via the 3m and 4m methods', *ACM Transactions on Mathematical Software (TOMS)* **44**(1), 1–36.

Wikipedia contributors (2023), 'Cilk – wikipedia, the free encyclopedia', https://en.wikipedia.org/wiki/Cilk. Accessed: 2024-05-18. Last edited on 30 March 2023, at 08:49 (UTC).

**URL:** *https://en.wikipedia.org/wiki/Cilk*

Wilson, K. G. (1974*a*), 'Confinement of Quarks', *Phys. Rev. D* **10**, 2445–2459.

Wilson, K. G. (1974*b*), 'Confinement of quarks', *Physical review D* **10**(8), 2445–2459.

Wilson, K. G. (1980), 'MONTE CARLO CALCULATIONS FOR THE LATTICE GAUGE THEORY', *NATO Sci. Ser. B* **59**, 363–402.

# Appendix A

# List of Publications

[1] Rahman, M.S., Kelefouras, V., Pica, C. and Rago, A.: Accelerating HiRep Lattice Simulations on CPU-based Computer Clusters. Proceedings of XXXV IUPAP Conference on Computational Physics (CCP2024). Springers Proceedings in Physics.

[2] Rahman, M.S., Kelefouras, V., Pica, C. and Rago, A. Code optimization strategies for HiRep lattice simulations. *Computer Physics Communications* (Under review).

# Index