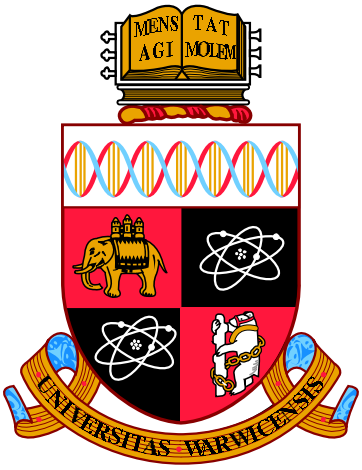


An Algorithm for Computing Short-Range Forces in Molecular Dynamics Simulations with Non-Uniform Particle Densities

by

Timothy R. Law

A thesis submitted to the University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

September 2017

Contents

Abstract

We develop the *projection sorting* algorithm, used to compute pairwise short-range interaction forces between particles in molecular dynamics simulations. We contrast this algorithm to the state of the art and discuss situations where it may be particularly effective.

We then explore the efficient implementation of the projection sorting algorithm in both on-node (shared memory parallel) and off-node (distributed memory parallel) environments. We provide AVX, AVX2, KNC and AVX-512 intrinsic implementations of the force calculation kernel. We use the modern multi- and many-core architectures: Intel Haswell, Broadwell Knights Corner (KNC) and Knights Landing (KNL), as representative slice of modern High Performance Computing (HPC) installations.

In the course of implementation we use our algorithm as a means of optimising a contemporary biophysical molecular dynamics simulation of chromosome condensation. We compare state-of-the-art Molecular Dynamics (MD) algorithms and projection sorting, and experimentally demonstrate the performance gains possible with our algorithm. These experiments are carried out in single- and multi-node configurations. We observe speedups of up to 5× when comparing our algorithm to the state of the art, and up to 10× when compared to the original unoptimised simulation. These optimisations have directly affected the ability of domain scientists to carry out their work.

Acknowledgments

First and foremost I would like to thank my supervisor, Prof. Stephen Jarvis for securing my funding, giving me the opportunity to undertake a PhD, and nurturing the research group that made this thesis possible.

Second I extend my sincere gratitude to colleagues at Intel, in particular Jonny Hancox, Gaurav Kaul and John Pennycook—thank you for involving me in so many projects, enabling access to code, data and hardware, and making time for me in your busy schedules.

Within the Department of Computer Science Faiz Sayyid, Steven Wright, James Davis and Richard Bunt in particular (amongst innumerable others) have all contributed either directly or indirectly to this thesis, and to making my time at Warwick enjoyable.

Finally I would like to thank my other friends and family for the many varied and entertaining distractions from work over the past four years, and Jess, for the ceaseless moral and gastronomic support.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree. The work presented (including data generated and data analysis) was all carried out by the author.

Parts of this thesis have been previously published by the author in:

- Timothy R. Law, Jonny Hancox, Tammy M. K. Cheng, Raphaël A. G. Chaleil, Steven A. Wright, Paul A. Bates, and Stephen A. Jarvis, Optimisation of a molecular dynamics simulation of chromosome condensation, Proceedings of the 28th International Symposium on Computer Architecture and High Performance Computing 2016, Los Angeles, USA, October 2016 [53]

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- The University of Warwick, United Kingdom:
  Engineering and Physical Sciences Research Council Studentship (1365607)
- Intel Corporation

Abbreviations

AoS Array-of-Structs

API Application Programming Interface

ASIC Application-Specific Integrated Chip

AVX Advanced Vector eXtensions

AVX2 Advanced Vector eXtensions 2

AVX-512 Advanced Vector eXtensions 512-bit

CFD Computational Fluid Dynamics

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DRAM Dynamic Random Access Memory

FPGA Field-Programmable Gate Array

FMM Fast Multipole Method

FLOP Floating-Point Operation

GPU Graphics Processing Unit

GPGPU General Purpose Graphics Processing Unit

HPC High Performance Computing

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

KNC Knights Corner

KNL Knights Landing

LAMMPS Large-scale Atomic/Molecular Massively Parallel Simulator

LINPACK Linear Algebra Package

LLC Last Level Cache

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MCDRAM Multi-Channel DRAM

MD Molecular Dynamics

MPI the Message Passing Interface standard

N3 Newton's Third Law

NAMD Nanoscale Molecular Dynamics

NAS NASA Advanced Supercomputing Division

NPB NAS Parallel Benchmarks

NUMA Non-Uniform Memory Access

OoO Out-of-Order

OpenMP the Open Multiprocessing standard

PACE Performance Analysis and Characterisation Environment

PAPI Performance Application Programming Interface

PCIe Peripheral Component Interconnect Express

POSIX Portable Operating Systems Interface

PRNG Pseudo Random Number Generator

PS Projection Sorting

RAM Random Access Memory

SDE Software Development Emulator

SIMD Single Instruction Multiple Data

SMT Simultaneous Multithreading

SNL Sandia National Laboratories

SoA Structure-of-Arrays

SRAM Static Random Access Memory

SSE Streaming SIMD Extensions

SST Structural Simulation Toolkit

TAU Tuning and Analysis Utilities

TPU Tensor Processing Unit

VL Verlet Lists

WARPP WARwick Performance Prediction Toolkit

List of Tables

List of Figures

Comfort as a philosophy of life! The least possible commotion, nothing shocking. Those who so love comfort will never seek where there is not definitely something to find.

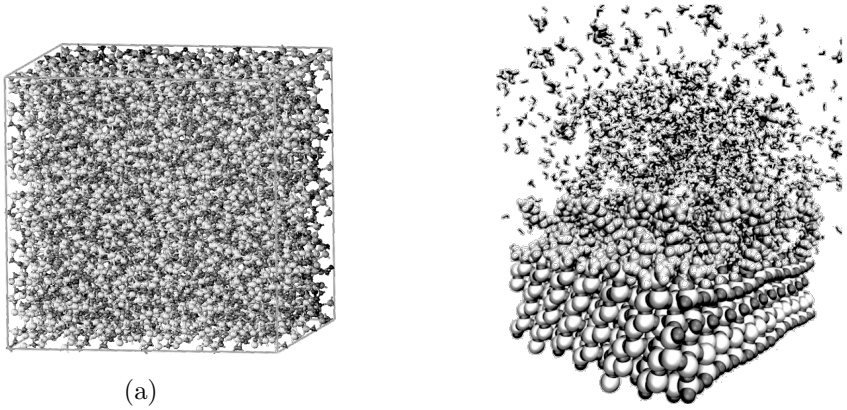
— Arnold Schoenberg

Chapter 1

Introduction

Computational simulations are widely used across many scientific disciplines, spanning a variety of domains of investigation from crystalline atomic structures to cell pathways, with numerous software packages available. Of these, Molecular Dynamics (MD) simulations are some of the most well known. Two prominent examples are the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [73], developed by Sandia National Laboratories to simulate materials under the influence of various physical potentials, and Nanoscale Molecular Dynamics (NAMD) [71], which focuses on biological applications and has been used solve important recent medical problems, such as resolving the structure of the HIV-1 virus responsible for AIDS [91].

Simulations on this scale require enormous computational resources and the time is long since past where a single machine could provide the necessary power. In our current age of falling clock-speeds (due to thermodynamic constraints), exploiting the increasing amounts of available parallelism at all levels—from large networked clusters, down to optimal ordering of microarchitecture instructions—has become crucially important. Maturing many-core architectures such as Intel's Knights Landing (KNL) and NVIDIA's Pascal and Volta Graphics Processing Unit (GPU) architectures epitomise this philosophy of "going wide", but in turn demand much



(a) (b)

Figure 1.1: Comparison between MD simulations exhibiting uniform and non-uniform particle density. (a) shows a regular lattice-like structure, whereas in (b) a much more irregular structure can be seen.

more from implementations in order to extract maximum performance. Significant work has gone into optimising MD applications for such architectures [9, 35, 70].

1.1 Motivations

The initial motivation for the work contained in this thesis was the desire to improve the performance of a particular biophysical MD simulation from Cheng et al. [20], described in detail in Chapter 5. As the work progressed however, it became apparent that the root cause of many of the performance issues encountered was the use of classic MD algorithms designed decades ago with specific types of simulation in mind. Whilst these algorithms worked, in that they produced the correct answer, they were not the most appropriate choices for achieving good performance.

Specifically, this MD simulation exhibits what we will term *non-uniform particle density*. Many MD simulations are designed to handle materials such as atomic lattices, or gases, which conversely exhibit approximately constant particle density throughout the simulation domain. See Figure 1.1 for a comparison between these two types of structure. Algorithms designed for simulations like Figure 1.1a

can exhibit inefficiencies when applied in situations as shown in Figure 1.1b, and vice versa.

1.2 Aims & Objectives

Our primary focus in this thesis is the development and optimisation of an algorithm suitable for pairwise short-range force calculations (that is, forces between every pair of particles that are within a certain small distance of each other, see Chapter 3 for a full discussion) within MD applications which exhibit non-uniform particle density. We call this algorithm *projection sorting*. Owing to their computational expense and widespread applicability, pairwise short-range force calculations have received significant attention in the MD literature.

> **Conjecture**: It is possible to accelerate pairwise short-range force calculations relative to classic MD methods by taking closer account of the domain-specific properties of the simulation (such as non-uniform particle density).

The three research questions (referred to henceforth as RQ1, RQ2 and RQ3) we propose to answer are as follows:

- **RQ1:** How might an algorithm designed to accelerate an MD simulation in accordance with the above conjecture look?
- **RQ2:** How might such an algorithm perform (and how might it be best implemented) in a shared-memory parallel environment?
- **RQ3:** How might such an algorithm perform (and how might it be best implemented) in a distributed-memory parallel environment?

1.3 Research Methodology

The research we undertake is primarily *quantitative*. We will use the methods and techniques which make up *performance engineering* in order to improve the effectiveness of our new algorithm, and to compare its effectiveness against existing algorithms. Performance engineering is discussed in detail in Chapter 2 (see Section 2.4), however we will briefly discuss the principles here.

Performance engineering is an iterative process consisting of 4 phases: benchmarking, profiling, optimisation and analysis. Benchmarks assess a system to determine peak feasible performance, profiles assess an application to determine actual performance, optimisation improves actual performance (moving it closer to peak feasible performance), and analysis determines the effectiveness of optimisations, and may suggest new ones. The results of this cycle then feed back into a new iteration.

In this thesis we seek to improve established MD methods when applied to certain types of simulation. The standard means of calculating pairwise short-range forces is by means of Verlet (or neighbour) lists [84] (discussed in Chapter 3). Therefore the analysis phase of the performance engineering cycle will consist primarily of quantitative comparisons between the algorithm we will develop, and Verlet lists. Further details of benchmarking, profiling and optimisation techniques are discussed in Chapter 2.

1.4 Thesis Contributions

The research presented in this thesis makes the following primary contributions:

- We develop the *projection sorting* algorithm, used to compute pairwise short-range interaction forces between particles in molecular dynamics simulations. We compare this algorithm qualitatively to the state of the art and discuss situations where it may be particularly effective. This pertains to RQ1.

- We explore the efficient implementation of the projection sorting algorithm in both on-node (shared memory parallel) and off-node (distributed memory parallel) environments. We provide Advanced Vector eXtensions (AVX), Advanced Vector eXtensions 2 (AVX2), Knights Corner (KNC) and Advanced Vector eXtensions 512-bit (AVX-512) intrinsic implementations of the force calculation kernel. We use the modern multi- and many-core architectures: Intel Haswell, Broadwell KNC and KNL, as a representative slice of modern High Performance Computing (HPC) installations. This pertains to RQ2 and RQ3.

- We use our algorithm as a means of optimising a contemporary biophysical molecular dynamics simulation of chromosome condensation [20]. We compare state-of-the-art MD algorithms and projection sorting, and experimentally demonstrate the performance gains possible with our algorithm. These experiments are carried out in single- and multi-node configurations. We observe speedups of up to 5× when comparing our algorithm to the state of the art, and up to 10× when compared to the original unoptimised simulation. These optimisations have directly affected the ability of domain scientists to carry out their work. This pertains to RQ2 and RQ3.

1.5 Thesis Overview

The remainder of this thesis is structured as follows:

- **Chapter 2** provides an overview of the fundamentals and terminology related to the field of HPC, including the development and current state of parallel hardware and the science of performance engineering, which is at the core of this thesis.

- **Chapter 3** presents a similar treatment of computational molecular dynamics

simulations, broadly covering the aspects relevant to this thesis.

- **Chapter 4** presents our algorithm: *projection sorting*. We provide an overview and broadly compare it against Verlet lists, the state of the art for pairwise short-range force calculations. This chapter covers RQ1.

- **Chapter 5** describes a molecular dynamics simulation of chromosome condensation from Cheng et al. [20]. This simulation is used in Chapters 6 and 7 to compare the projection sorting algorithm from Chapter 4 against state-of-the-art MD algorithms. We also detail some optimisations performed unrelated to the projection sorting algorithm. This chapter covers RQ2.

- **Chapters 6 and 7** implement projection sorting in the context of the simulation from Chapter 5. In Chapter 6 we explore shared memory settings, using the Intel Haswell and KNC architectures, and deal with the data layout and vectorisation of the algorithm. Chapter 7 extends the implementation to distributed memory settings using the Message Passing Interface standard (MPI), and uses the Intel Broadwell and KNL architectures. This chapter covers RQ3.

- **Chapter 8** concludes the thesis and discusses the limitations of our work, as well as future work.

Chapter 2

Parallel Hardware and Performance Engineering

We begin with an overview of the current state of parallel hardware, and then discuss the principles of performance engineering which underpin this thesis.

2.1 Types of Parallelism

Flynn's taxonomy [28] (see Table 2.1) is a broad classification of computer architectures based on whether or not they can handle multiple concurrent instruction and/or data streams. The vast majority of modern architectures are capable of executing in any of these four configurations; multiple processes can run concurrently, all using separate data streams.

Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD) can be thought of as employing "task parallelism", where different

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1: Flynn's taxonomy

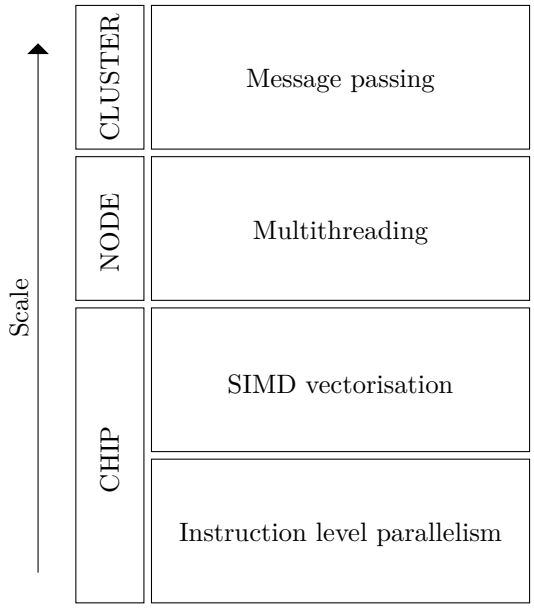


Figure 2.1: Layers of parallelism.

instruction streams execute concurrently, whereas Single Instruction Multiple Data (SIMD) characterises "data parallelism": multiple similar (or identical) instruction streams operating on different parts of a dataset.

In modern hardware, parallelism exists at several different layers, shown in Figure 2.1. Higher layers make use of parallelism at the lower layers. At the lowest layer, individual instructions executing on a processor are run in parallel. Above that, so called *vector* instructions operate on multiple pieces of data at once. One step higher, multiple cores can operate simultaneously on a single node, each running a different *thread* of execution. Finally at the top layer, multiple processes cooperate in a *cluster* environment, exchanging messages to coordinate their activities.

2.1.1 Instruction Level Parallelism

Instruction Level Parallelism (ILP) exists in several forms in modern microarchitectures. Pipelined processors work on multiple instructions simultaneous at different stages of the *fetch-decode-execute* cycle. While one instruction is executing, others can be fetched and decoded. Superscalar processors employ multiple pipelines which

operate concurrently. Out-of-Order (OoO) execution is possible—given a queue of instructions, a processor may execute any instruction whose dependencies are satisfied, regardless of its position in the queue. Instructions can also be executed speculatively, in anticipation of a branch in execution. Significant work has gone into developing accurate dynamic branch prediction schemes to minimise the number of speculatively executed instructions that must be discarded due to an incorrect branch prediction [90].

Writing code with ILP in mind is difficult, and rarely worthwhile in today's age of modern optimising compilers, which more often than not are capable of generating code optimised for ILP themselves.

2.1.2 SIMD Vectorisation

Modern microarchitectures often provide SIMD instruction sets alongside the standard serial operations. These *vector* instructions operate on larger-than-usual vector registers, containing multiple data items. For example, if one wished to add 100 pairs of numbers, SIMD instructions would allow adding two, four, eight or even more pairs in a single instruction. GPUs employ the widest SIMD of all, usually 2048-bits (or 64 32-bit integers).

Vector instruction sets include Intel's Streaming SIMD Extensions (SSE) and AVX and Arm's NEON. Developers may employ these directly using assembly code, or indirectly, either by *intrinsic* wrappers in high-level programming languages, through auto-vectorising compilers, or by making use of libraries with built in SIMD support.

2.1.3 Multithreading

As clock speeds rose rapidly throughout the 90's and early 2000's, it became clear to chip manufacturers that the associated increase in thermal output was unsustainable. Rather than continue to push clock speeds, they moved to a multi-core

model, where more than one chip running at a lower clock speed runs in parallel. Applications make use of multiple cores simultaneously through the use of more than one *thread* of execution. Threads can make use of both task- and data-parallelism, whereas SIMD vectorisation is better suited to data-parallelism.

Modern architectures also make use of Simultaneous Multithreading (SMT), where a single core can concurrently execute more than one thread [83]. 2-way SMT is common in today's multi-core processors, and GPUs reach much higher numbers.

Common methods of using multiple threads in applications include direct implementation with Portable Operating Systems Interface (POSIX) threads or a higher level approach with tools like the Open Multiprocessing standard (OpenMP) [69] or Intel Cilk [13].

2.1.4 Message Passing

When reaching the level of a cluster of networked nodes, processors no longer share a single memory address space, and must instead communicate explicitly over an interconnect. The advantages of this model are that applications can be scaled to much larger numbers of cores than is possible on a single node. As the nodes are entirely independent, it also obviates barriers to parallel efficiency like cache coherency protocols. However, applications must be carefully designed to minimise the amount of inter-node communication required.

In HPC, MPI is the *de facto* standard for implementing message passing applications [2].

2.2 The Memory Hierarchy

All computer architectures require some mechanism to store and retrieve data. Many different technologies exist to achieve this, ranging widely in speed of access, capacity and cost. During computation the Central Processing Unit (CPU) operates

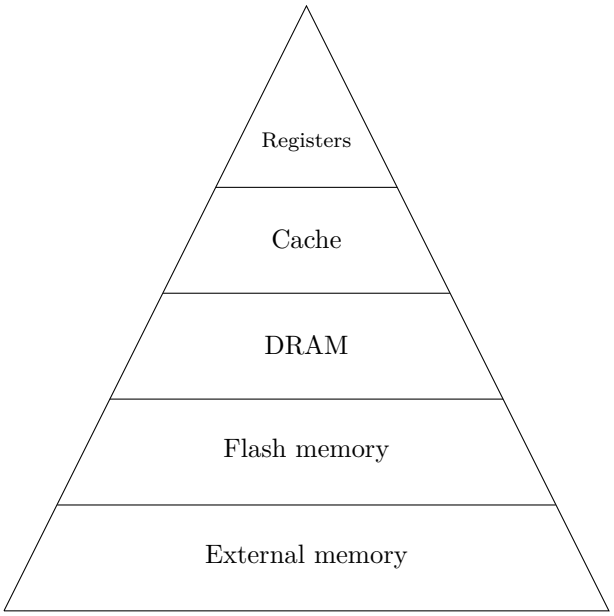


Figure 2.2: The memory hierarchy: the fastest and lowest capacity at the top (on-chip memory), the slowest and largest capacity at the bottom (external mechanical drives).

extremely rapidly on small fragments of data. Here, minimising latency is the primary concern and the capacity required is low, expensive technologies such as Static Random Access Memory (SRAM) are used for processor registers and cache. Conversely, storing large amounts of data (as in backups or data warehouses) has very different requirements. Capacity is key, speed is secondary. Using SRAM would be economically infeasible, other technologies such as mechanical drives or magnetic tape are more appropriate.

Figure 2.2 summarises the different tiers of storage commonly used, known as the memory hierarchy. The fastest, smallest, costliest technologies used are at the top, and the slowest, largest, cheapest at the bottom.

2.3 Many-core and Heterogeneous Computing

A relatively recent development is the advent of "many-core" processors. Whereas standard multi-core processors are designed to be efficient for both serial and parallel

code, many-core processors are designed exclusively for highly parallel code and do away with single-thread optimisations that may limit scaling. GPUs fall under the umbrella of many-core processors.

At the time of writing, the number one supercomputer on the Top500 ranking [61] is Sunway TaihuLight, based at the National Supercomputing Centre in Wuxi, China. This machine achieves over 93 PFLOP/s from 40,960 Sunway SW26010 many-core processors, each with 260 cores.

Intel's Xeon Phi product line is based on many-core architectures, specifically Knights Corner (KNC) and Knights Landing (KNL). Both offer in the region of 60 cores. We use both of these in this thesis and compare results to traditional multi-core architectures.

A related concept is heterogeneous computing, where single nodes contain various *accelerator* hardware alongside traditional CPUs. Accelerators have long been used for a variety of purposes with GPUs of course originally being used to accelerate the graphics pipeline in multimedia computers, and ASICs commonly being used to relieve the burden on the CPU during tasks such as video encoding. In modern HPC, General Purpose Graphics Processing Unit (GPGPU) computing is flourishing, with many scientific codes offloading all or part of their computational workload to GPUs. Recently the so-called Tensor Processing Unit (TPU) has emerged in response to the proliferation of deep-learning applications, designed specifically to accelerate the kinds of computation done in, for example, convolutional neural networks (CNNs). The Field-Programmable Gate Array (FPGA) has also seen a resurgence—many applications are being experimented with on such hardware.

In short, for many applications it is increasingly important to be able to take advantage of accelerators, and a variety of software solutions have emerged to make writing code that will run on different accelerator architectures easier. OpenMP [69] and OpenACC [68] both offer "directive-driven" ways to offload code to accelera-

tors, where developers decorate their code with special statements which instruct the compiler to generate device-specific code, and transparently handle moving data between host and device memory. The CUDA library has continued to evolve, making offloading to NVIDIA GPUs easier than ever. Performance portability libraries such as Kokkos [24] and RAJA [42] have also begun to emerge, which enable developers to write an application once, and then run it on many different architectures with minimal overhead.

2.4 Performance Engineering

In HPC, performance engineering is the set of techniques and methodologies that scientists and engineers use to predict and improve the *performance*—here referring to a variety of metrics including wall-clock time taken, storage space used and power expended—of computation. The fundamental basis of performance engineering is the cycle of hardware benchmarking, application profiling, code optimisation and performance modelling shown in Figure 2.3.

2.4.1 Benchmarking

Benchmarking is often the first task performed when presented with a new computer architecture or set of hardware. A *benchmark* is a piece of software designed to assess performance in some way. Perhaps the most well known examples in the field of HPC are the Linear Algebra Package (LINPACK) benchmarks, based on the Fortran library of the same name [23]. These benchmarks test a system's ability to rapidly solve dense systems of linear equations, a common task in many engineering problems. LINPACK scores are used to build the Top500, a biannual ranking of supercomputers across the world [61].

Other benchmarks include STREAM, designed to assess sustainable memory bandwidth [59], SKaMPI, which measures network communication times in a

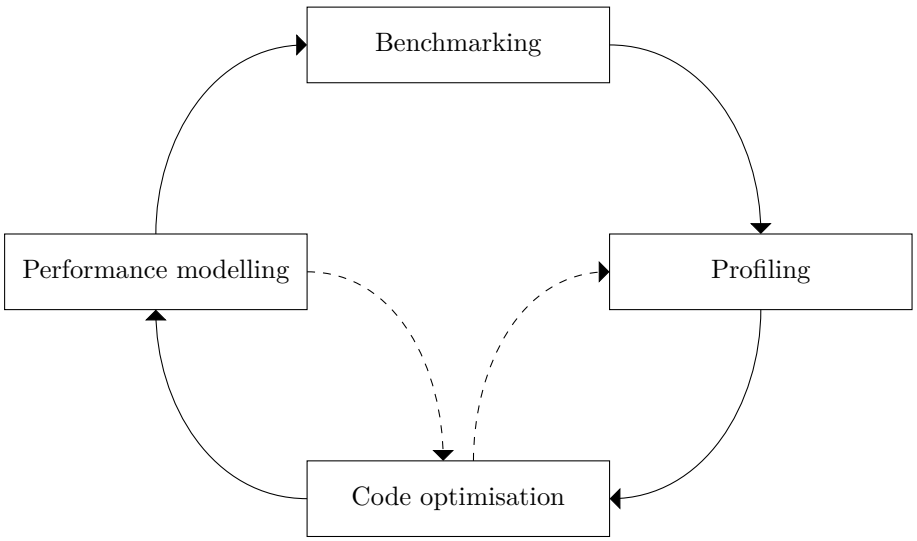


Figure 2.3: Performance engineering: (1) Benchmark the hardware to determine attainable performance, (2) profile the application to determine areas where performance is not up to standard, (3) optimise the application code to bring performance closer to the ideal, (4) analyse achieved performance, which can feed back into further optimisation. When new hardware is acquired, the cycle begins again.

cluster environment [76], and the NAS Parallel Benchmarks (NPB) from the NASA Advanced Supercomputing Division (NAS), a suite of benchmarks derived from Computational Fluid Dynamics (CFD) applications.

2.4.2 Profiling

Application profiling refers to the use of a tool—a *profiler*—to analyse the execution of an application. Profilers can provide information such as which sections of code consume the majority of the runtime, memory usage throughout execution, parallel inefficiencies and I/O wait times for network of disk access. They can also shed light on low-level metrics collected by hardware performance counters, such as efficiency of cache usage and instruction level parallelism.

Examples of profiling tools include:

- GNU `gprof` [26]

  Uses a combination of *static automated instrumentation* (where a compiler

inserts calls to profiler functionality within the compiled application itself) and hardware performance counter sampling to generate a *call graph* (showing which source code functions call which other ones) and a list of functions ordered by cumulative execution time.

- Intel VTune Amplifier [43]

  Uses *dynamic automated instrumentation* (where the code is analysed and rewritten at runtime to call profiler functions), based on the Intel Pin tool [55], and hardware performance counter sampling. Can generate detailed call graphs and cumulative execution time per individual line of code, in addition to a myriad of low level metrics.

- Linux `perf` [1]

  Also capable of performance counter sampling, and dynamic automated instrumentation using the Linux `kprobe` and `uprobe` frameworks for kernel and userspace instrumentation respectively. Deeply integrated into the Linux kernel providing rich profiling functionality.

- Tuning and Analysis Utilities (TAU) Performance System [79]

  Provides static and dynamic automated instrumentation, and event-based sampling. Features many graphical analyses of the collected profile.

- Performance Application Programming Interface (PAPI) [17]

  Provides an Application Programming Interface (API) for programmers to *manually instrument* their code with access to hardware performance counters. PAPI doesn't profile code itself, but provides the tools for programmers to add profiling to their applications manually.

2.4.3 Code Optimisation

When the application hotspots have been identified through profiling, code optimisation activities can begin. Such optimisations can take a variety of forms ranging

```
// before unrolling
for (i = 0; i < N-1; i++) {
    A[i] = 0.5 * (B[i] + B[i+1]);
}

// after unrolling by a factor of 4
int M = (N-1) / 4;
for (i = 0; i < M; i++) {
    int j = i*4;
    A[j+0] = 0.5 * (B[j+0] + B[j+1]);
    A[j+1] = 0.5 * (B[j+1] + B[j+2]);
    A[j+2] = 0.5 * (B[j+2] + B[j+3]);
    A[j+3] = 0.5 * (B[j+3] + B[j+4]);
}

// ... perform remaining iterations
if ((N-1) % 4 != 0) {
    for (i = M*4; i < N-1; i++) {
        A[i] = 0.5 * (B[i] + B[i+1]);
    }
}
```

Listing 1: An example of unrolling a 1D stencil by a factor of 4. This optimisation improves the ratio of compute to loop-control overhead.

from simple code transformations to total rewrites based on new algorithms or alternative hardware solutions (such as accelerator devices like GPUs and FPGAs) which radically alter the performance characteristics of the application.

Commonly used code transformations include *loop unrolling* [67], where the body of a simple loop may be duplicated n times and the number of iterations reduced correspondingly by a factor of n. An example of this is shown in Listing 1. The principal advantages of loop unrolling are that it reduces the number of branches required to complete a loop, and allows other hardware mechanisms such as prefetching and pipelining to operate more effectively. The downsides are an increase in the size of compiled code, and worsened readability (although as modern optimising compilers are perfectly capable of performing loop unrolling without programmer intervention, the latter of these is moot).

Another example of a common code transformation is *loop tiling* [88]. As discussed in Section 2.2 (the Memory Hierarchy), modern architectures make use of multiple levels of data storage, differing in speed and size. Moving data from one level to another takes a non-zero amount of time, so achieving optimal performance

```
// before tiling
for (i = 0; i < N; i++) {
    for (j = 0; j < P; j++) {
        C[i][j] = 0.0;
        for (k = 0; k < M; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

// after tiling (assuming N and P are even)
for (i = 0; i < N; i += 2) {
    for (j = 0; j < P; j += 2) {
        C[i+0][j+0] = 0.0;
        C[i+0][j+1] = 0.0;
        C[i+1][j+0] = 0.0;
        C[i+1][j+1] = 0.0;
        for (k = 0; k < M; k++) {
            C[i+0][j+0] += A[i+0][k] * B[k][j+0];
            C[i+0][j+1] += A[i+0][k] * B[k][j+1];
            C[i+1][j+0] += A[i+1][k] * B[k][j+0];
            C[i+1][j+1] += A[i+1][k] * B[k][j+1];
        }
    }
}
```

Listing 2: An example of tiling a matrix multiplication $\mathbf{C} = \mathbf{AB}$ where $\mathbf{A}$ is an $N \times M$ matrix and $\mathbf{B}$ is an $M \times P$ matrix. In the un-tiled version, calculating each row of $\mathbf{C}$ requires loading the i-th row of $\mathbf{A}$ and *all* of $\mathbf{B}$. In the tiled version, entries in 2×2 blocks of $\mathbf{C}$ are calculated simultaneously, which reuses each loaded value from $\mathbf{A}$ and $\mathbf{B}$ twice.

is contingent on minimising data movement. Loop tiling is a technique that can be used to reorder iterations in nested-loop structures to increase *data reuse*—where data is moved to the top level of the memory hierarchy as few times as possible, and the amount of computation performed on the data while it is at that top level is maximised. Listing 2 shows an example of simple loop tiling applied to matrix multiplication. Whilst loop tiling can improve performance, optimally tiling a loop usually requires knowledge of the particular hardware the code is to be run on, in particular the cache sizes, and is therefore not a portable optimisation. This highlights the distinction between *cache-aware* optimisations (such as loop tiling), and *cache-oblivious* optimisations. Cache-oblivious algorithms are ones that make effective use of hierarchical memory in general without being particularly suited to any specific hardware implementation.

Other types of code transformation include loop fission (where a single loop

is broken into multiple loops over the same iteration space, each taking a portion of the original loops body), loop fusion (the opposite), loop interchange (changing the order of nested loops to improve data access patterns), code hoisting (moving loop invariant code outside a loop), strip mining (a kind of loop tiling where the iteration space is partitioned to allow for vectorised implementations) and countless others [8, 85]. Many of these optimisations can be performed transparently by modern optimising compilers.

After optimising an application, the natural thing to do is to compare it with the original unoptimised version as a baseline. If the optimisation is successful in improving performance one may then profile the code again to find secondary areas to improve upon. This process is repeated until the level of performance reached is satisfactory, relative to the benchmarks previously obtained.

2.4.4 Performance Modelling

Constructing an accurate model of an application's performance has many benefits. It allows prediction of performance on other hardware [60, 41, 48, 50], which is useful in procurement (supercomputers are very expensive, and being able to speculatively "try before you buy" is valuable when one knows what workloads one intends to run on a purchased machine). Performance models can also inform further optimisations on the existing hardware [22, 64], and reveal potential misconfiguration [18].

Conducting performance modelling usually involves a combination of analytical methods and simulation [7]. Simpler aspects may be modelled using a series of mathematical equations, while deeper complexities may resist analytical approaches and require simulation. The results from simulation can be plugged into analytical models to tune their accuracy. An important consideration when using simulation is the computational resources required; if a simulation takes longer to perform than simply running the application in the desired configuration, then it is not a particularly useful model. This also applies to the time required to develop the model in

the first place.

The characteristics of the hardware on which an applications runs form an integral part of a performance model, as different hardware will greatly influence the application's runtime. It is important to separate the hardware-related aspects of the performance model from the application-related aspects, so that different applications can be modelled given a single hardware model, and likewise so that a single application's performance model can be used in the context of multiple machine models [49].

Performance modelling tools include SNL's Structural Simulation Toolkit (SST), which provides a framework for simulating parallel scientific applications at both the macro and micro levels [65]. SST supports both offline and online simulation. Offline simulation is where a real application is run on a real machine, instrumented so as to produce a "trace" of various events of interest, such as MPI calls. This trace can be played back and further data can be extrapolated from it. Online simulation is where an application skeleton is constructed, which in part mimics the real application. Rather than actually running expensive sections of compute, they are replaced with an estimation of how much time they would have taken based on application and system data. Online simulation is more flexible than offline simulation, and is the recommended way to use SST.

Intel's Software Development Emulator (SDE) allows developers to emulate the execution of code under architectures other than that which available hardware implements. For example, an application that has been compiled with future x86 instructions that are not available in currently released hardware. To use SDE, the developer inserts special start and end markers around a loop body in their code. The SDE application looks through a compiled binary for these markers, and performs analysis of the machine code between them. By using knowledge of instruction latency and throughput, and of the target architecture, SDE is able to produce various statistics about the expected performance of the code. This allows

application developers to prepare their code for future architectures before they arrive *in silico*.

The WARwick Performance Prediction Toolkit (WARPP) [33, 34] is a set of tools designed to automate parts of the performance modelling workflow. It combines automated source code instrumentation, benchmarking and discrete event simulation to produce estimated performance figures. The input to a WARPP simulation is instrumented application source code (capturing the control flow and the compute), MPI and IO benchmarks, and actual timings produced by the instrumented code from a real run. WARPP builds on the earlier Performance Analysis and Characterisation Environment (PACE) [19] tool.

A survey of other available performance modelling tools has previously been undertaken by Allan [7].

2.5 Mini-applications

Undertaking performance engineering endeavours on a large production application is often an arduous process. It's not unusual for modern scientific codes to reach half a million lines of code. Making significant changes to such an application can take months of development and re-validation effort. Mini-applications, or *mini-apps*, can alleviate some of this burden.

A mini-application is a small program, usually around 10 to 20 thousand lines of code, which is designed to mimic the performance characteristics of a particular production application. For example, the Mantevo suite from Sandia National Laboratories (SNL) [38, 39] consists of several mini-apps covering a variety of application domains including numerical solutions of partial differential equations (both implicit and explicit, structured and unstructured), hydrodynamics and molecular dynamics. CloverLeaf [56] is around 15 thousand lines and represents an Eulerian solution of the compressible Euler equations. miniMD is around the same size,

and consists of the basic computational kernels from the LAMMPS molecular dynamics package. The small size of these mini-apps has enabled the development of a great many variants, both using different implementations and combinations of parallelism, and targeting different machines and processor types. The results of this work have then fed back into the development of the corresponding production applications.

2.6 Summary

Modern hardware offers a great many ways for developers to leverage parallelism in their applications. Indeed, doing so is crucial to getting the most out of the hardware. Performance engineering provides a framework for taking underperforming applications and identifying and rectifying their weaknesses. Throughout the rest of this thesis, we apply this framework to studying and improving the performance of a class of MD applications.

Chapter 3

Molecular Dynamics

Molecular Dynamics (MD) is a computational method that uses simulation to study the dynamical behaviour of systems of particles. The trajectories of N particles are computed by integrating Newton's classical equations of motion. MD was developed in the 1950s by theoretical physicists looking to study such systems but lacking suitable analytical methods to do so [4, 6, 15, 27]. Today MD is used across a wide range of scientific fields, including chemical physics, materials science and biophysics. Well known molecular dynamics codes include LAMMPS [73], NAMD [12, 45, 71, 80], DL_POLY [82] and GROMACS [5].

To accurately capture the motion of atomic-scale particles, the timescale over which Newton's equations of motion are integrated must be extremely small. A single "timestep" is measured in femtoseconds for such simulations. Many thousands of timesteps must therefore be computed to obtain results over a representative duration in "real world" time. Considerable research effort has been expended to make such simulations tractable.

In this chapter, we will explain concepts in 2-dimensional space, for simplicity of exposition. Everything presented generalises straightforwardly to 3-dimensions.

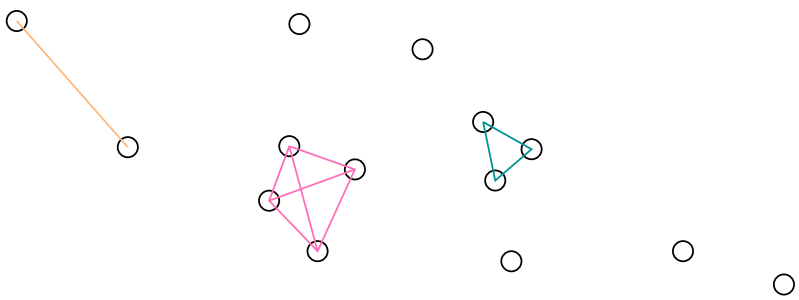


Figure 3.1: An example of the forces acting between particles in an MD simulation. The yellow line indicates a pair potential between two particles. The blue lines indicate a three-body potential, and the pink lines indicate a four-body potential. All the particles exert a non-zero force on all other particles for each such degree. In practice, these interactions must be approximated. In all simulations we consider, all higher-order potentials are approximated—only the pair-potentials are calculated explicitly.

3.1 Computational Aspects of Molecular Dynamics

At a fundamental level, MD simulations must solve the differential equations given by Newton's equations of motion [52]:

$$\begin{aligned} m_i \frac{d\boldsymbol{v}_i}{dt} &= \sum_{j_1} \boldsymbol{F}_{i,j_1} + \sum_{j_1}\sum_{j_2} \boldsymbol{F}_{i,j_1,j_2} + \cdots + \sum_{j_1}\sum_{j_2}\cdots\sum_{j_{N-1}} \boldsymbol{F}_{i,j_1,j_2,\ldots,j_{N-1}} \\ \frac{d\boldsymbol{x}_i}{dt} &= \boldsymbol{v}_i \end{aligned} \tag{3.1}$$

The first of these, is the well-known $F = ma$ (the acceleration induced in an object by a net force is in the same direction as the force, proportional to the magnitude of the force, and inversely proportional to the mass of the object). The second relates position to velocity. The trajectory of each particle i is calculated based on the sum of all "pair potentials"—forces $\boldsymbol{F}_{i,j}$ between i and all other particles j—plus all three-body potentials, four-body potentials, and so on (illustrated in Figure 3.1). Such a sum is extremely expensive to calculate. In practice, the interaction energy is usually largely concentrated in the pair term. Many MD simulations truncate the series after the pair term and calculate an "effective" pair potential which assumes

pairwise additivity and includes the average effects of the many-body terms (and quantum effects), based on experimental data.

The force terms in Equation 3.1 are typically non-linear functions of the distances between the particles involved. They fall into two categories: *short-* and *long-range*. Long-range forces are those that do not become negligible over a finite distance, and therefore require inspection of all particles during calculation. Calculating long-range forces directly is practically infeasible in most cases, and as such a variety of approximate methods have been developed such as the Fast Multipole Method (FMM) [77]. Despite advances in these methods, many MD simulations do not directly include long-range force terms due to their expense.

Short-range forces on the other hand, do become negligible over a finite distance. Only particles that fall within a limited "cut-off radius" (r_c) need be considered when calculating short-range forces. Particles outside of this radius make only a negligible contribution to the total force and do not need to be considered. The cut-off radius is typically taken as a constant parameter in molecular dynamics simulations, and a suitable value is determined through knowledge of the physical processes being simulated.

Short-range forces often dominate the execution time of MD simulations, therefore the cut-off radius and the particle density (the inverse of the average separation between particles within the simulation domain, which depends on the physical processes under simulation) are two quantities that can greatly affect the overall performance. A larger cut-off radius includes more neighbour particles in the calculation of each particle's short-range forces, which is more computationally expensive. A lower particle density implies that there will be fewer neighbour particles falling within a given cut-off radius of a particle. A simulation of a gas implies a lower particle density than a simulation of a solid. This thesis is concerned with the calculation of pairwise short-range forces, and we now review common approaches in the literature.

3.1.1 Short-Range Force Algorithms

The naïve approach to calculating pairwise short-range forces on a given particle is to iterate through all other particles, calculate the distances between each pair, and only apply forces for those within the cut-off radius. This takes quadratic time in the number of particles and is therefore not scalable to larger systems.

This can be accelerated by decomposing the simulation domain into disjoint uniform axis-aligned square or rectangular "cells". A list is maintained for each cell containing the particles resident. Particle-pair lookups are then restricted to a small neighbourhood of cells within the cut-off radius. This is called using "cell lists", or the "link-cell" approach [74, 40, 75, 25, 58, 37].

As cells are square or rectangular as opposed to the sphere indicated by the cut-off radius, cell lists still carry an inefficiency in the number of particle-pair checks required. All particles associated with each cell that partially intersects the circle defined by the cut-off radius must be inspected, despite the likelihood that not all the particles are truly within the cut-off radius. This is illustrated in Figure 3.2. We refer to these failing checks as being *spurious*.

The inefficiency increases as the cell size is increased, as larger cells imply a greater spatial area that does not intersect the cut-off circle. For example, when the cells are defined as squares of side r_c, the immediate neighbourhood of 9 cells must be inspected for each particle. This has an area of $(3r_c)^2$, which is over 2.8 times greater than circle of area πr_c^2 defined by the cut-off radius. Assuming a uniform particle density across these 9 cells, 65% of the particle-pair checks are unnecessary. This increases to 84% in 3-dimensions. While the inefficiency tends to zero with the cell size (see Figure 3.3), programming overheads will erase the benefit past a certain point as the number of cells to be inspected increases rapidly as the size is decreased. Gonnet [32] discusses reducing the inefficiency by sorting cell particles according to their scalar projection onto the vector connecting the cell centre and the position of the particle for which forces are being calculated. This strategy

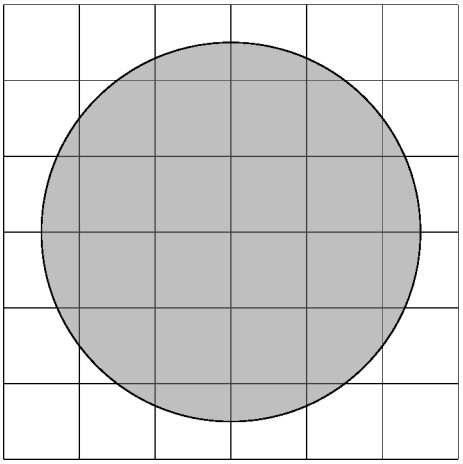


Figure 3.2: Cell lists are inefficient in the sense that particles are checked that cannot possibly fall within the cut-off radius. Here the shaded circle represents a particle's cut-off radius, and the grid represents a decomposition of the space into cells. All particles are checked within each cell that intersects the cut-off radius. As some squares only partially intersect the circle, particles that fall within those cells, but outside of the intersection, are checked despite not being within the cut-off radius.

allows the algorithm to stop checking distances after encountering the first particle that falls outside of the cut-off radius, although it introduces its own overheads.

The situation can often be improved by utilising Verlet, or "neighbour", lists [84]. This approach was first developed by Loup Verlet in 1967. A list is maintained per particle, storing only the particles within a certain "Verlet radius" r_v ($r_v \geq r_c$). Rather than building this list every timestep, each list is reused a number of times (k). Doing so requires knowledge of the maximum distance a particle can move in a single timestep, called the "skin distance" of the simulation (r_s). This quantity is used to ensure that we do not neglect contributions from particles within the cut-off radius of a particular reference particle. Particles that are outside the cut-off radius of the reference particle at timestep n, but which could possibly enter it before timestep $n+k$ should be included within the reference particle's Verlet list. Therefore, the Verlet radius is set $r_v = r_c + 2kr_s$ (with the factor two accounting for the situation where two particles are moving directly towards each other at maximum velocity). It should be noted that Verlet lists and

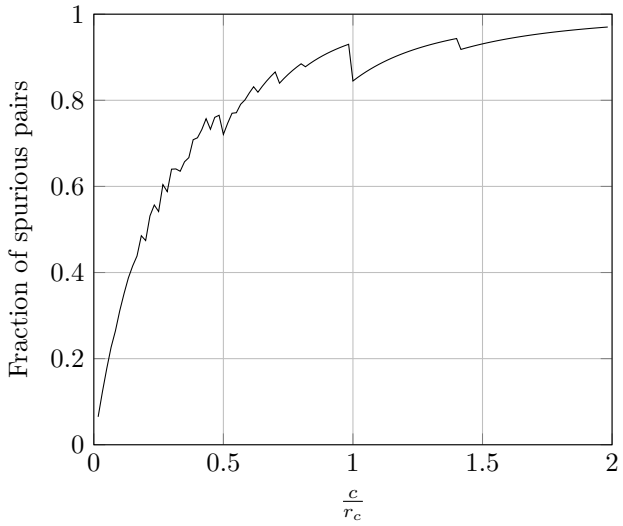


Figure 3.3: Cell lists—fraction of unnecessary particle-pair checks as a function of cell size for a fixed r_c (assuming 3 spatial dimensions and uniform particle density).

cell lists are not mutually exclusive; cell lists are often an efficient way of building Verlet lists. Using this strategy, the cost of traversing the cell lists is amortised over k timesteps.

One disadvantage of Verlet lists is that the number of particle-pair checks is now tied to the speed at which particles move, which can be undesirable when this is a large, or highly variable quantity. The rebuild period k should be chosen to strike a balance between the frequency of expensive list rebuilds, and the increased size of the lists. It is also clear that the viability of using Verlet lists depends heavily on the skin distance; it must be small relative to the cut-off radius or the number of unnecessary particle-pair checks will be too high. Figure 3.4 shows how the fraction of unnecessary checks increases rapidly with both k and r_s. Nevertheless, many modern MD codes, including NAMD [71] and LAMMPS [73], use Verlet lists.

Newton's Third Law

As the force terms are based on relative particle positions, it should be noted that significant computational redundancy exists. Following Newton's Third Law (N3) [52],

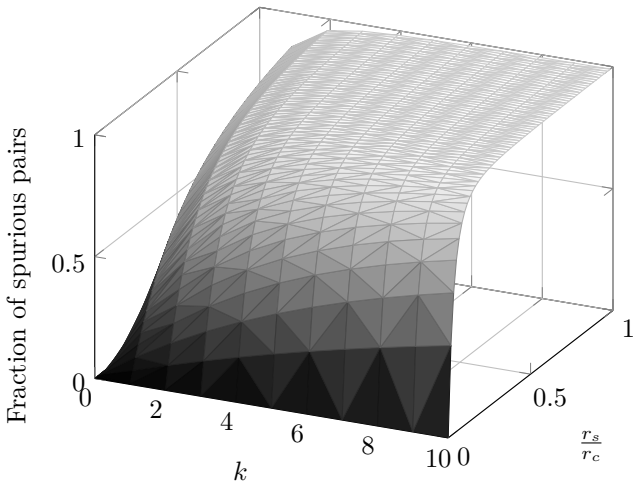


Figure 3.4: Verlet lists—fraction of unnecessary particle-pair checks as a function of rebuild period k and skin distance r_s for a fixed r_c (assuming a uniform particle density).

the pairwise force of particle i on particle j will be the negative of the pairwise force of particle j on particle i ("every action has an equal and opposite reaction"). This fact can be exploited to cut the number of force calculations in half, and is used extensively throughout many MD algorithms.

Spatial Locality

Another type of optimisation involves reordering stored particles such that those local to each other in two-or-three dimensional simulation space are also local in the computer's memory (a one dimensional space). Spatial locality is important in contemporary computer architectures, whose performance often depends on being able to work around memory latency by means of reuse within multiple layers of cache, and the ability to predict and prefetch data likely to be needed in the near future. Yao et al. discuss sorting particles along an axis of the simulation domain [89]. Gonnet discusses decreasing spurious distance checks through sorting [32]. Anderson et al. demonstrate a successful application of a more sophisticated approach, whereby particles are ordered according to their distance along a space-filling Hilbert curve [9]. The Hilbert curve is chosen due to its locality preserving properties [63].

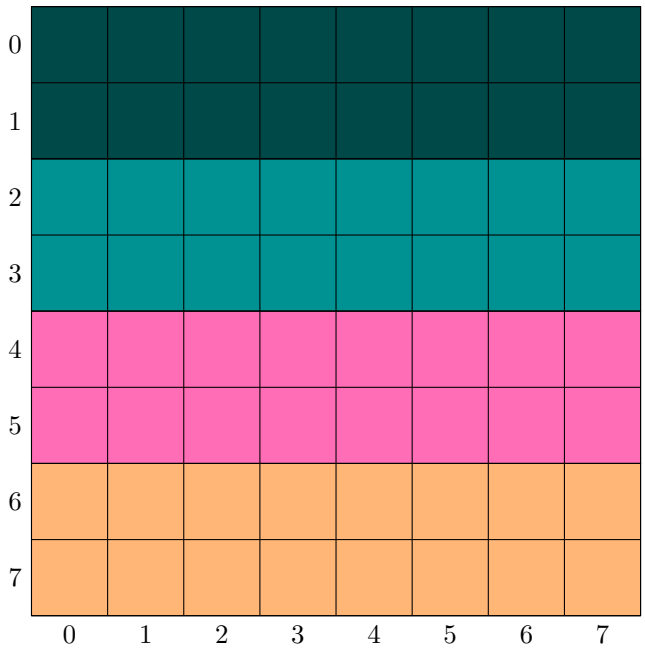
3.2 Parallelisation of Molecular Dynamics

MD exhibits significant inherent parallelism [14, 29]. Given a set of positions, calculating the acting forces can be done in parallel for every individual particle. The same is true for updating velocities and positions in accordance with Newton's equations of motion. Plimpton [73] describes the three main ways of extracting this parallelism: *particle* decomposition, *force* decomposition and *spatial* decomposition.

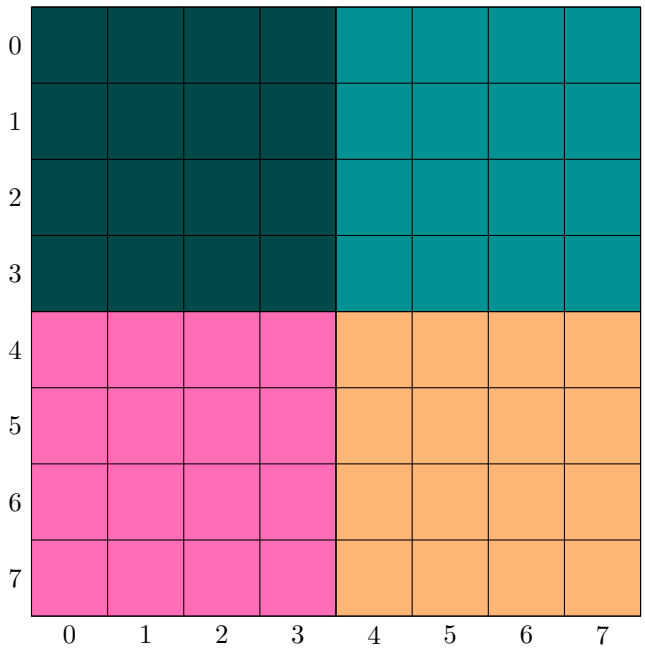
In a particle decomposition, each parallel processor takes a subset of the particles in the simulation and is responsible for computation of all forces on these particles, and for updating their trajectories. Even if particles move away from their initial positions, the same processor retains control for the duration of the simulation. The advantages of this scheme are that it is simple to implement and assigns an even amount of work to each processor. The main problem is that in a distributed memory setting it requires an *all-to-all* communication operation after each timestep: each processor needs to know the current position of all particles in the simulation, and must therefore send all its newly computed particle positions to every other processor, and receive a copy of every other processors' in turn. This is an expensive operation which can seriously inhibit scaling.

This deficiency can be somewhat remedied by using a force decomposition, where each processor is responsible for a block of interactions in the matrix of particle-pairs (see Figure 3.5 which contrasts this with the particle decomposition). Under this scheme processors do not need to know all particle positions, only those belonging to one of their pairs. This reduces the communication requirements.

Spatial decompositions are entirely different. Rather than taking an arbitrary subset of either particles or particle-pairs, each parallel processor is responsible for a sub-region of the physical simulation space. If a particle moves such that it no longer resides within a given processor's sub-region it is handed off to the new processor. This decomposition is the most complicated to implement but has the



(a)



(b)

Figure 3.5: The difference between the particle decomposition and force decomposition strategies for $N = 8$ particles. The two grids show all particle-pairs, and the colours indicate which of four parallel processors is responsible for each pair. (a) shows the particle decomposition and (b) shows the force decomposition.

Figure 3.6: An example of an MD dataset exhibiting significant non-uniformity in the particle density.

lowest communication requirements of all. Each processor only needs to exchange particles with processors handling adjacent sub-regions. The price of this efficiency comes in the form of load balancing: if one processor's sub-region contains more particles than the others it will take longer to complete its calculations and hold up progression to the next timestep. In many simulations this is not a significant issue, where particles are densely packed and have an approximately uniform density across the entire simulation domain. Other simulations, particularly biological ones, can exhibit significant non-uniformity here (see Figure 3.6 for an example that we will return to in Chapter 5).

3.3 Summary

We summarise the computational MD literature reviewed in this chapter in Table 3.1. There is a great deal of work on this subject and this chapter aims to provide a representative sample of various approaches rather than an exhaustive list. In this thesis we now move to add another tool to the computational scientist's toolbox in the same vein as many of the authors listed above—the work of many of whom is applicable only to certain kinds of MD simulation. In particular, we take inspiration from Yao et al. and Gonnets' ideas on particle sorting [89, 32] and develop them with respect to simulations with certain characteristics of geometry and particle mobility (as discussed in Chapter 1), where the approaches surveyed

Verlet (1967) [84]	Develops **Verlet lists** to study the thermodynamical properties of a system of 864 particles interacting through a Lennard-Jones potential.
Quentrec and Brot (1973) [74]	Proposes an algorithm similar to **cell lists** where each cell contains at most one particle. Identifies situations where this method can give speedups over Verlet lists.
Hockney et al. (1974) [40]	Proposes a linked-list based formulation of **cell lists** for use in plasma simulations.
Rapaport (1988) [75]	Studies the effective vectorisation of **cell lists**.
Everaers and Kremer (1994) [25]	Extends the ideas of Quentrec and Brot regarding **cell lists** where each cell contains at most one particle.
Plimpton (1995) [73]	Surveys different ways to parallelise molecular dynamics simulations.
Mattson and Rice (1999) [58]	Modifies **cell lists** by the use of cells of side smaller than r_c to decrease the number of spurious checks.
Heinz and Hünenberger (2004) [37]	Proposes an alternate **cell list** formulation which can efficiently handle periodic boundary conditions, which operates by grouping adjacent non-empty cells into stripes according to a mask.
Yao et al. (2004) [89]	Proposes optimisations to the combination of **cell lists** and **Verlet lists** through partial Verlet list updates and sorting data to improve cache performance.
Gonnet (2007) [32]	Proposes a way to accelerate searching **cell lists** by sorting particles within cells by their projection onto a vector connecting two cell centres.
Anderson et al. (2008) [9]	Gives a complete implementation of MD algorithms on GPU, paying particular mind to spatial locality.
Pennycook et al. (2013) [70]	Explores the implementation of MD algorithms on Intel's Xeon Phi coprocessor, paying particular attention to effective vectorisation.
This work	Proposes an alternative to cell lists and Verlet lists based on sorting particles across the whole simulation domain by their projection onto a particular vector.

Table 3.1: Summary of the developments in computational MD covered in this chapter, and how this thesis fits in.

above may not yield good performance.

Chapter 4

Projection Sorting

In this chapter we propose the *projection sorting* algorithm, an alternative approach to computing pairwise short-range forces as discussed in Section 3.1.1, as an answer to Research Question 1 (RQ1):

> **Conjecture**: It is possible to accelerate pairwise short-range force calculations relative to classic MD methods by taking closer account of the domain-specific properties of the simulation (such as non-uniform particle density).
>
> **RQ1:** How might an algorithm designed to accelerate an MD simulation in accordance with the above conjecture look?

Section 4.1 contains a description of the projection sorting algorithm and explanation of the steps involved. We then provide an illustrative example in Section 4.2. In Section 4.3 we discuss the computation complexity of projection sorting, and its inherent scope for optimisation. Finally, in Section 4.4, we compare and contrast the algorithm to Verlet lists, the state of the art in pairwise short-range force calculation.

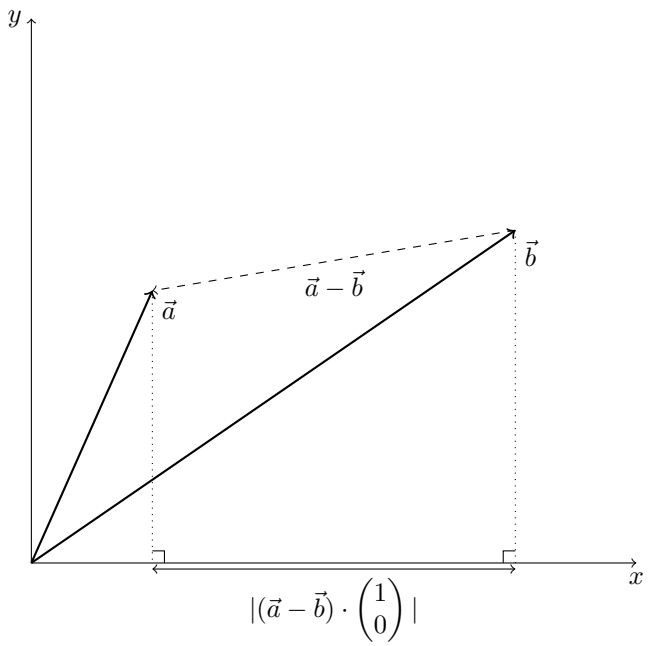


Figure 4.1: 2-dimensional illustration of projecting two position vectors onto the x-axis. Equation 4.1 clearly follows from Pythagoras' theorem.

4.1 Description of the Algorithm

When two particles are separated by a distance greater than r_c along any vector $\boldsymbol{v}$, the Euclidean distance between them cannot possibly be less than r_c. This fact forms the crux of the projection sorting algorithm, and is formalised for two particle position vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ and an arbitrary unit vector $\hat{\boldsymbol{v}}$ in Equation 4.1. Figure 4.1 demonstrates the statement graphically—where $\hat{\boldsymbol{v}}$ is set to the unit vector in the direction of the x-axis ($\boldsymbol{i}$). It can be seen that the projection of the two position vectors onto $\boldsymbol{i}$ forms a right-triangle with the vector $\boldsymbol{a} - \boldsymbol{b}$. Equation 4.1 thus follows directly from Pythagoras' theorem, which has the corollary that the hypotenuse must be at least as long as the adjacent.

$$\forall \hat{\boldsymbol{v}}, \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^3, |(\boldsymbol{a} - \boldsymbol{b}) \cdot \hat{\boldsymbol{v}}| \leq \|\boldsymbol{a} - \boldsymbol{b}\| \tag{4.1}$$

It follows that if one were to order the set of particles in a MD simulation

by the scalar projections of their position vectors onto a vector $\boldsymbol{v}$, then for each particle there would exist a contiguous block of other particles either side in that ordering, limiting the search space for possible neighbour particles within r_c. Only particles within this block could possibly be within the cut-off radius. Note that this is a necessary condition, but not a sufficient one; a full distance check must still be carried out to ensure that the particle is within r_c for all possible values of $\boldsymbol{v}$. The crucial point is that, outside of this block, all other particles could be disregarded completely. In a sense, this is like a 1-dimensional formulation of cell lists, except the lists are implicit in the particle ordering, rather than explicitly built, stored and traversed at each iteration. This leads to the following three step algorithm for computing pairwise short-range forces:

1. **Selecting $\boldsymbol{v}$:** The first step is to select a suitable vector $\boldsymbol{v}$ to use for calculating the projections. For a simulation with non-uniform particle density, the choice of $\boldsymbol{v}$ can greatly impact the number of particles for which the projections fall within the cut-off radius r_c. Imagine a configuration of particles where all are positioned along a straight line. Choosing $\boldsymbol{v}$ to be perpendicular to this line would result in all projections being the same, whereas choosing the line itself would spread the projections out as much as possible, and consequently minimise the number of distance checks and maximise performance. Figures 4.2b and 4.2c below demonstrate how the choice of $\boldsymbol{v}$ can impact the number of distance checks that need to be performed. In general the choice of $\boldsymbol{v}$ should be informed by knowledge of the specific simulation, and should ideally maximise the difference between any pair of particle projections. Possible ways of determining a reasonable value for $\boldsymbol{v}$ in general include principal component analysis [46] and linear regression [62]. The vector $\boldsymbol{v}$ need not be reselected every timestep (or indeed ever, depending on the simulation), but instead only when the conformation of particles changes in such a way that it is rendered suboptimal.

2. **Particle sort:** The particles are then sorted according to their scalar projections onto $\boldsymbol{v}$. Scalar projections are calculated by taking the dot product of the particle position vector and the vector $\hat{\boldsymbol{v}}$. This creates a 1D spatial ordering within which forces can be efficiently calculated.

3. **Force sweep:** In order to calculate the pairwise short-range forces on particle i due to its neighbours, loop over all particles j, where j is bounded by k_{lo} and k_{hi}, the first particles below and above i respectively for which the difference between the scalar projections of j and i onto $\boldsymbol{v}$ exceeds r_c. It is guaranteed by Equation 4.1 that no particle outside this set will fall within the cut-off radius. For each j, calculate the distance $\|\boldsymbol{a} - \boldsymbol{b}\|$. If this is less than r_c, then j is a neighbour of i and the application-specific force between the two can be calculated and added to the total force on i. The neighbour search space can be further reduced to particles $i < j < k_{\text{hi}}$ by using N3, applying the complementary negative force to each particle j.

It should be noted that the calculation of the scalar projections in Step 2 can be obviated by a change of basis at the start of the simulation. Changing the basis such that $\hat{\boldsymbol{v}}$ corresponds to a basis vector allows simply using that particle coordinate as the sort key. Whether this is appropriate depends on the simulation in question, specifically whether any other parts of the code necessitate use of the original coordinate system (for instance, file I/O using a file format expecting particle positions relative to the standard basis). Computationally speaking, calculating scalar projections is a cheap operation, and only needs to be performed once per timestep, so a change of basis is unlikely to yield significant performance improvements.

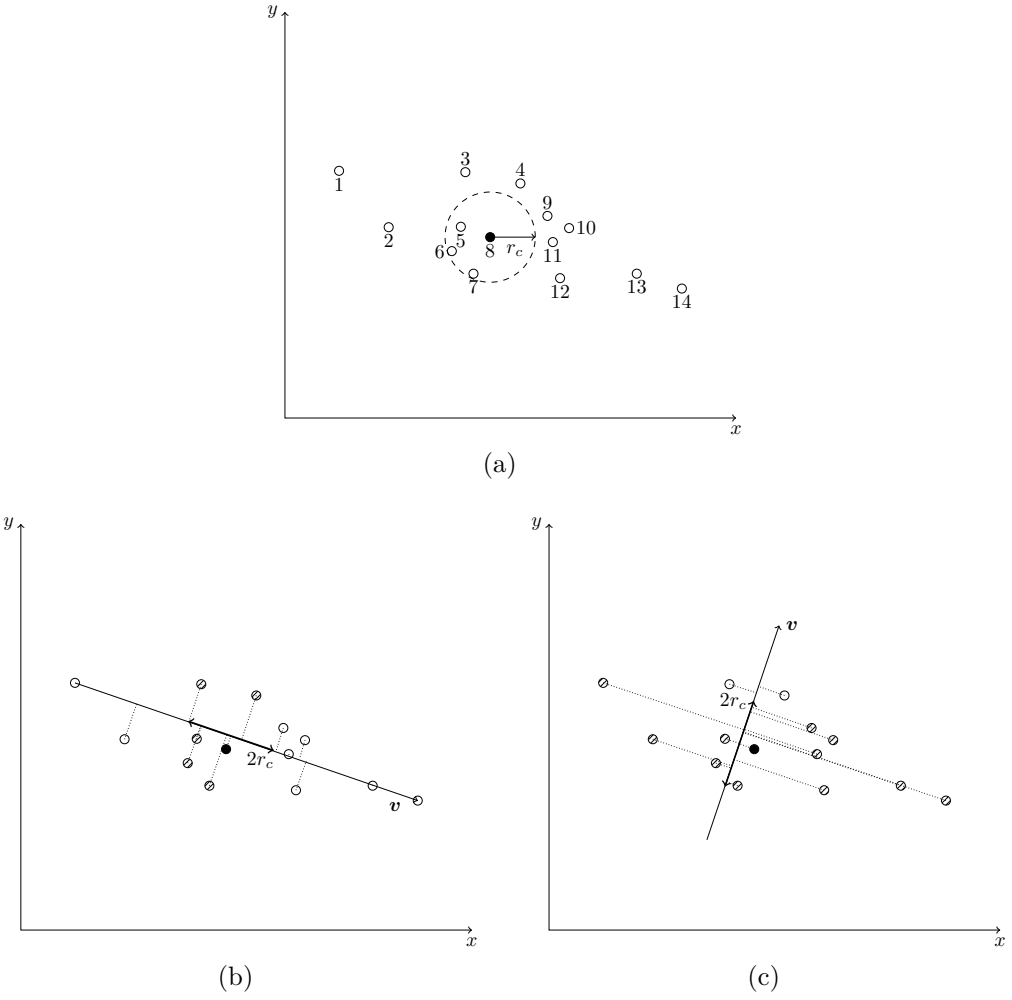


Figure 4.2: 2-dimensional illustration of projection sorting. Imagine we have 14 particles as shown in (a) (numbers indicating some possible in-memory ordering), and that we wish to determine which particles are within the cut-off radius of the filled particle 8 (the cut-off radius being marked by the dashed circle). We can see that there are 3 such particles: 5, 6 and 7, but how would the computer determine this? (b) and (c) show the calculation of the projections onto two possible values for $\boldsymbol{v}$. In (b) we choose $\boldsymbol{v}$ as the vector connecting the two endmost particles, whereas in (c) we use a vector perpendicular to this. The thick arrows show the projection of the cut-off circle onto $\boldsymbol{v}$ in each case, which defines the search area for candidate particles. Particles whose projections fall into this region are hatched. In (b) there are 5 such particles, whereas in (c) there are 11. We must perform full distance checks for all of these particles, so (b) is clearly the better choice for $\boldsymbol{v}$.

```
for (i = 0; i < nb; i++) {
    projs[i] = p_x[i] * vx + p_y[i] * vy + p_z[i] * vz;
}
```

Listing 3: C-like pseudocode demonstrating the calculation of scalar projections onto the unit vector $\hat{\boldsymbol{v}}$ (with components `vx`, `vy` and `vz`) from the `nb` particle positions in 3-dimensional space (stored in `p_x`, `p_y` and `p_z`).

4.2 Illustrative Example

As a simple example, let us consider the case presented in Figure 4.2. Figure 4.2a shows the setup: we have 14 particles arranged in the xy-plane (numbered arbitrarily according to some initial in-memory ordering), and we wish to calculate the pairwise short-range forces on the filled particle number 8. The circle indicates the cut-off radius for this particle, which contains 3 other neighbour particles numbered 5, 6 and 7 (although an implementation could not look at this diagram to see this).

The first step in the projection sorting algorithm is to select some vector $\boldsymbol{v}$ which we will use to calculate the particle projections. Figure 4.2b shows the vector between the two endmost particles. It is clear from this diagrammatic perspective that this choice does a good job of spreading out the particle projections (indicated by the dotted lines). Figure 4.2c shows the projections we would get if we used the line perpendicular; note that more particles fall within the marked range here, indicating that more full distance checks need to be carried out (at additional computational expense). In practice the selection of $\boldsymbol{v}$ would be based on some knowledge of the dataset, and would ideally spread out the set of projections as much as possible. In the experiments in Chapters 6 and 7 we actually do use the strategy of picking the two endmost particles, as we know the dataset is laid out in a long stripe.

The next step in the algorithm is to actually calculate and sort the scalar projections of the particle positions onto $\boldsymbol{v}$ (taking $\boldsymbol{v}$ as shown in Figure 4.2b). Figure 4.3 shows the resulting values of these calculations. Figure 4.3a gives the particle position vectors and $\hat{\boldsymbol{v}}$ and the first row in Figure 4.3b shows the calculated

Particle #	x	y
1	-3.80	0.48
2	-2.70	-0.77
3	-1.00	0.45
4	0.22	0.20
5	-1.10	-0.76
6	-1.30	-1.30
7	-0.82	-1.80
8	-0.45	-0.99
9	0.82	-0.52
10	1.30	-0.79
11	0.94	-1.10
12	1.10	-1.90
13	2.80	-1.80
14	3.80	-2.13
$\hat{\boldsymbol{v}}$	0.95	-0.32

(a)

Projections (unsorted)													
1	2	3	4	5	6	7	8	9	10	11	12	13	14
-3.76	-2.32	-1.09	0.14	-0.80	-0.82	-0.20	-0.11	0.95	1.49	1.24	1.65	3.24	4.29
Projections (sorted)													
1	2	3	6	5	7	8	4	9	11	10	12	13	14
-3.76	-2.32	-1.09	-0.82	-0.80	-0.20	-0.11	0.14	0.95	1.24	1.49	1.65	3.24	4.29
True distances													
1	2	3	6	5	7	8	4	9	11	10	12	13	14
-	-	1.54	0.90	0.69	0.89	-	1.37	-	-	-	-	-	-

(b)

Figure 4.3: Worked example of projection sorting for the scenario in Figure 4.2b. (a) shows the xy-positions for each particle (numbered as in Figure 4.2a), and the components of $\hat{\boldsymbol{v}}$. In this example, we take $r_c = 1$, and wish to calculate the neighbours of particle 8. The first row in (b) shows the projections calculated from this data, ordered by particle number. In the second row these are sorted, and the shaded cells show the range of particle projections within the range $-0.11 \pm r_c$. In the final row, the true distances are calculated for these 5 candidate particles using $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. 3 of these distances are less than r_c, corresponding to particles 5, 6 and 7 as expected.

projections. Listing 3 gives some example code for calculating the projections: a simple dot product.

Next we must use the projections to determine which of the particles are candidate neighbours for particle 8. Although it is clear from Figure 4.2b which projected particles fall within the cut-off radius (the hatched particles), it is not so easy for the computer to efficiently determine. This is where the particle sort comes in. Sorting the particles by their scalar projection means that we can step through the particles either side of the filled particle *in order of projection*, and therefore know that when we reach the first particle that is out of range, there are *no other possible candidate particles*. This sorted order is shown in the second line of Figure 4.3b, and the shaded block of projections indicates the candidate neighbours for particle 8.

Finally we calculate the true neighbours of particle 8, which are guaranteed by Equation 4.1 to be within the set of candidate neighbours identified by the contiguous block of shaded projections. For each candidate neighbour we calculate the true distance between it and particle 8 using the typical formula. The results of these calculations are shown in the last line of Figure 4.3b, and the true neighbours (those for which the distance is less than $r_c = 1$) are shaded green. As expected, these correspond to the particles within the cut-off radius originally shown in Figure 4.2a. This concludes the projection sorting algorithm.

4.3 Complexity & Scope for Optimisation

We can deduce the algorithmic complexity of projection sorting as follows. Assuming N particles, calculating projections is $\mathcal{O}(N)$. Sorting is commonly taken to be $\mathcal{O}(N \log N)$, however this may often be a loose bound for our purposes. The ordering in an implementation of projection sorting will not change much from timestep to timestep, and can therefore use the previous timesteps ordering as a "nearly

sorted" starting point, which brings the complexity closer to $\mathcal{O}(N)$ for some sorting algorithms (e.g. Insertion sort). For each particle we must then step through the set of candidate neighbours (let us call this set $\mathcal{C}(i)$ for particle i). If we let $M = (\sum_{i=1}^{N} |\mathcal{C}(i)|)/N$ (the mean size of $\mathcal{C}(i)$), then this is $\mathcal{O}(NM)$. Therefore the total complexity is $\mathcal{O}(N(1 + \log N + M))$. The algorithmic efficiency therefore hinges on M being "small enough", which is a property of the simulation in question.

Complexity is not the only factor on which performance depends however, and a key advantage of projection sorting is the ability to implement it in an efficient way with respect to modern hardware. The particle sort, in addition to restricting the search space for true neighbours, gives another advantage. We can combine the force calculations with the distance checks into a "stepping" algorithm, which lets us reuse distance information, as well as knowing that memory accesses will be efficient as all data is contiguous in memory. As modern applications are often memory-bound, this memory efficiency is a crucial property of projection sorting. Listing 4 gives pseudocode for a simple implementation of this algorithm (assuming pre-sorted arrays), where for each particle i (e.g. the filled particle in Figure 4.2), the candidate neighbours j (the 5 hatched particles in Figure 4.2b) are identified by stepping through the sorted arrays in both directions, a full distance check is performed for each j, and if that check is passed (i.e. if j is within the original cut-off radius illustrated in Figure 4.2a), then the force on i due to j is calculated and added to the total force on i. We do not illustrate the sorting here, as for the purposes of exposition any sorting algorithm will do.

4.4 Preliminary Comparison to Verlet Lists

When seeking to compare projection sorting to Verlet lists, it is helpful to break each algorithm down into two parts: periodic computation and force computation. Periodic computation refers to work that needs to be done in preparation for the

```
1  for (i = 0; i < nb; i++) {
2      fxi = 0.0;
3      fyi = 0.0;
4      fzi = 0.0;
5
6      // step in one direction
7      for (j = i + 1; j < nb; j++) {
8
9          // are we stepping out of the block of candidate neighbours?
10         if (projs[j] - projs[i] > rc) break;
11
12         // full distance check
13         dx = p_x[j] - p_x[i];
14         dy = p_y[j] - p_y[i];
15         dz = p_z[j] - p_z[i];
16         dsq = dx*dx + dy*dy + dz*dz;
17         if (dsq <= rcsq) {
18
19             // force calculation
20             coeff = /* app-dependent */;
21             fxi += coeff * dx;
22             fyi += coeff * dy;
23             fzi += coeff * dz;
24         }
25     }
26
27     // step in the other direction
28     for (j = i - 1; j >= 0; j--) {
29         if (projs[i] - projs[j] > rc) break;
30         // ...
31         // as above
32         // ...
33     }
34
35     // accumulate forces on particle i due to neighbours
36     f_x[i] += fxi;
37     f_y[i] += fyi;
38     f_z[i] += fzi;
39 }
```

Listing 4: C-like pseudocode demonstrating the final part of the projection sorting algorithm. It is assumed that the `projs` array contains scalar projections of each particle position, and that this array and the position arrays `p_x`, `p_y` and `p_z` are all sorted on these projections. The resulting force arrays `f_x`, `f_y` and `f_z` will also be sorted in this manner. The conditionals with the **`break`** statements on Lines 10 and 29 represent the points where the algorithm checks whether the projections of particles i and j are within range of each other. Immediately afterwards, a full distance check is performed. If this succeeds, the force between the pair of particles is calculated.

force computation: for Verlet lists this is building the lists themselves, including constructing cell lists, which needs to be done every k timesteps. For projection sorting this includes calculating particle projections and sorting the particles by said projections, which needs to be performed before every force computation. We call this "periodic" as, in the case of Verlet lists, it doesn't need to be done every timestep.

Force computation refers to using the precomputed information to calculate forces for a given set of particle positions, which must be done every timestep for both projection sorting and Verlet lists. In this section we discuss some of the trade-offs between projection sorting and Verlet lists.

4.4.1 Periodic Computation

The algorithmic complexity of the construction of cell lists (i.e. binning each particle depending on the cell it is in) particles is $\mathcal{O}(N)$ (calculating the cells and adding each particle to the appropriate list). Let $\mathcal{S}(i)$ be the set of particles in cell i. Construction of each particle's Verlet list depends on the contents of the surrounding $3 \times 3 \times 3$ cells. Let $K = \max_j |\mathcal{S}(j)|$, the size of largest cell. Then constructing all Verlet lists is $\mathcal{O}(NK)$ (omitting the constant factor $3^3 = 27$). Therefore the total complexity of the periodic computation associated with Verlet lists is $\mathcal{O}(N(1 + K))$. Above we derived the complexity for projection sorting, and the part of this pertaining to periodic computation (calculating and sorting projections) is $\mathcal{O}(N(1 + \log N))$. Therefore we are interested in the value of K, and how this compares to $\log N$. The value of K depends on the size of the cells, which depends on the value of r_v, which in turn depends on the parameters to the simulation (r_c, r_s, k etc.). In addition to the dependence on r_v, the number of particles in each cell also depends on the geometry of the simulation; in a simulation with significant non-uniform particle density, we would expect some cells to contain many particles, whereas others would contain none. Bringing in an example from an experiment we run in

Chapter 6, with N =128,000 we have $\log N \approx 17$ and $K = 44$ (the maximum cell list size, although the mean non-empty cell list size is 7.1).

As previously mentioned, the algorithmic complexity is only one factor in performance. If we directly compare the linear ($\mathcal{O}(N)$) terms discussed above (calculating projections and constructing cell lists), we find that computing a collection of N dot products (as is necessary for the projection calculation) is much cheaper on modern hardware than appending N items to many distinct lists (as is necessary for the cell list construction). This is because FLOPs are cheaper than memory accesses. Constructing cell lists is also more difficult to parallelise, as threads need to serialise their access to the shared cell lists to avoid race conditions.

Additionally we must remember that in the case of Verlet lists, the periodic costs are amortised over k timesteps, whereas the projection sorting costs occur every timestep. In short it is not easy to theoretically compare these periodic costs. We explore them empirically in Section 6.3.2.

4.4.2 Force Computation

Let $\mathcal{L}(i)$ be the length of the Verlet list associated with particle i. Then let $Q = (\sum_{i=1}^{N} |\mathcal{L}(i)|)/N$, the average length of all Verlet lists. The algorithmic complexity of the force computation phase when using Verlet lists is then $\mathcal{O}(NQ)$. We have from Section 4.3 that the corresponding complexity for projection sorting is $\mathcal{O}(NM)$, where M is the average number of candidate neighbours per particle. These complexities are therefore directly comparable with empirical values of Q and M for a given simulation. We perform this comparison in Chapter 6.

Complexity aside, a key advantage of projection sorting is the highly contiguous memory access pattern it affords during the force sweeps. As the particles are sorted they are accessed in a linear order, which allows the memory subsystems to work at peak efficiency, and enables highly efficient SIMD vectorisation. Runtime analysis of hardware counters reveals that around 99.8% of memory accesses during

the force sweeps hit L1 cache (see Chapter 6). This stands in stark comparison to Verlet lists, which require sophisticated gather/scatter memory operations on every access.

Another key metric is the number of particle-pair checks performed by each algorithm (previously discussed in Section 3.1.1). The number of checks is a good predictor of an algorithm's performance [86]. Whether projection sorting or Verlet lists require more spurious checks depends on the skin distance, rebuild period, and overall particle conformation. Typically, projection sorting will require fewer checks along the axis of projection, but more along orthogonal axes. This is because, along the axis of projection, projection sorting only searches for neighbours up to r_c, whereas Verlet lists searches up to $r_v > r_c$. Along orthogonal axes however, projection sorting must check all particles in the domain, whilst Verlet lists still only needs to search up to r_v.

4.4.3 Communication Costs

As MD simulations are almost always executed on clusters nowadays, it is instructive to look at the effect an algorithm might have on the communication costs of particle data between nodes. When implementing a distributed-memory parallelisation of MD using a spatial decomposition (note that here we are referring to a decomposition of the simulation domain between *nodes*, distinct from the spatial decomposition performed as a part of the cell list algorithm, see Section 3.2), adjacent processors need to share particle data so that particles on the edge of their subdomains have access to particles within their cut-off radius, but owned by another processor. As Verlet lists are constructed based on $r_v > r_c$, they require more data from adjacent processors than projection sorting does. The amount of communication increases cubically with an increased cut-off radius, so can be significant. Conversely, projection sorting needs to communicate more frequently, so again we have a trade-off dependent on skin distance.

4.4.4 Comparing Projection Sorting and Verlet Lists in Practice

In Chapters 6 and 7 we will seek to compare projection sorting and Verlet lists empirically. As these two methods operate in a fairly distinct manner, it is important that these comparisons are carefully considered to avoid an "apples-to-oranges" situation. At a high level, we are seeking to determine the fastest available way to execute a given simulation. We assume that this runtime is principally bound by the time taken to compute short-range pairwise interaction forces (as is often the case in MD simulations). Projection sorting and Verlet lists are both methods for computing these forces, and we assume that substituting one method for the other does not change either the results of the simulation or any aspect of the application performance under consideration other than the time taken to compute these forces, and the initialisation time (which is discounted as negligible in the context of a long running simulation). This is a reasonable assumption as most applications are broken up into independent parts which do not interact; this is standard software engineering practice.

As a consequence of the above, we can focus our attention on the individual implementations and parameterisations of the two algorithms. As Verlet lists are the *de facto* standard for computing short-range pairwise interaction forces in MD simulations (discussed in Chapter 3), the optimised implementation thereof is well studied. We will draw from the literature to ensure that our implementation of Verlet lists is as performant as possible on the architectures we will consider.

As is also discussed in Chapter 3, Verlet lists require selection of two principal parameters dependent on the particular application in question: the skin-distance r_s and rebuild period k. In order to ensure that we choose the values giving the best possible performance, we will perform a full parameter sweep and select the best performing values.

Given the above considerations, we believe that the empirical comparisons we present in the remainder of the thesis are not biased in favour of either projection

sorting or Verlet lists, and purely reflect which of the two gives the best performance in the context of the application in question (which we discuss in the next chapter).

4.5 Summary

Projection sorting is an algorithm for computing pairwise short-range forces in MD simulations. It is designed to work in simulations where the particle density is non-uniform: some areas have many particles and others have none. Projection sorting works by creating a linear ordering of particles dependent on each particle's scalar projection onto a certain unit vector $\hat{\boldsymbol{v}}$. The remainder of this thesis seeks to show that choosing a suitable vector and ordering particles in this manner can yield speedups over traditional methods.

Chapter 5

A Simulation of Chromosome Condensation

In this chapter, we introduce a modern MD simulation of chromosome condensation from Cheng et al. [20], which we use as a vehicle for the implementation of our ideas described in Chapter 4 and elaborated throughout the remainder of this thesis.

In the search for a suitable candidate application for the evaluation of a new algorithm, the primary relevant characteristics are as follows:

- **Domain characteristics**. The intuition behind projection sorting implies that it would be most effective in simulations which exhibit significant non-uniformity in particle density across the simulation domain, so we would like an application that exhibits these characteristics. Projection sorting is also designed to calculate short-range forces, so a suitable application should be heavily dependent on the efficient computation of such forces.

- **Size** (lines of code). A production molecular dynamics package is unsuitable as a testbed for new ideas and low-level optimisation work, as they typically involve hundreds of thousands of lines of code implementing a variety of nuanced physical models which are irrelevant to the algorithmic underpinnings.

Small applications, or mini-applications, require less work while still providing representative performance results.

- **Scientific value**. Ideally the code would be seeing active use, so that our optimisation work has impact.

The simulation from Cheng et al. fits well with all three of these. It exhibits a high degree of non-uniformity in particle density and spends the majority of its execution time in pairwise short-range force calculations. It is also small, consisting of approximately 7000 lines of code, and is actively being used by domain scientists at Cancer Research UK.

Other small MD applications, and MD mini-applications, exist, such as miniMD and CoMD from the Mantevo mini-application suite [39]. However these focus on the simulation of homogenous materials exhibiting unsuitable domain characteristics for effective use of the projection sorting algorithm.

We first provide a description of the simulation from Cheng et al. itself, enumerating the main computational aspects which are relevant to this thesis. We then briefly discuss some general optimisations we made to the simulation that are *not* related to projection sorting. In Chapter 6 we implement projection sorting and evaluate its performance in a shared memory environment, and in Chapter 7 we do the same for a distributed memory environment.

5.1 Chromosome Condensation

Genomes can be large—the DNA comprising the human genome is approximately 2 m long when stretched out. In order to fit inside the nucleus of each cell, the DNA is wrapped around many bundles of proteins to form packaging units called nucleosomes, collectively comprising a structure called *chromatin*. Chromatin's structure varies with the cell cycle; when the time comes for the cell to divide, it moves from the loose conformation of nucleosomes and DNA linkers (often likened to "beads

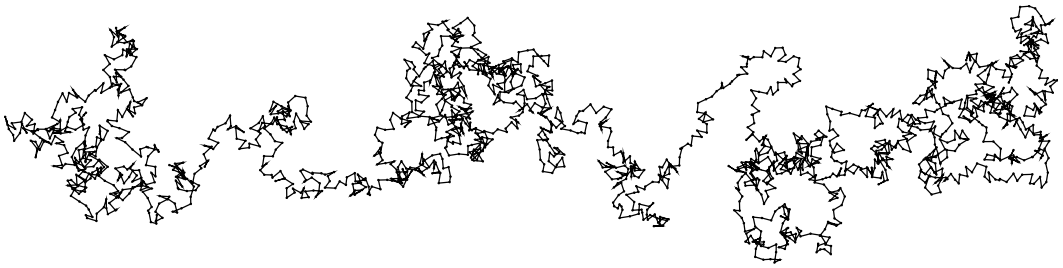


Figure 5.1: A visualisation of one of the datasets used—a conformation of yeast DNA dotted with nucleosomes.

on a string") to a more tightly compacted version typically associated with chromosomes. This process is known as *chromosome condensation*, and it is thought to be affected by protein complexes known as condensins, although exactly how it happens is currently not well understood.

Current research looks to leverage computational techniques to answer this question. The simulation from Cheng et al. is used to study the effects of different models of condensin interaction on the condensation of conformations of nucleosomes and linker DNA determined via *in vitro* methods (see Figure 5.1). The optimisations presented throughout this thesis are instrumental in making it feasible to run the lengthy simulations required.

5.1.1 Simulation Forces

The simulation treats the nucleosomes as uniform particles in an MD simulation, with radius 5 nm. The DNA linkers are modelled as ideal springs following Hooke's law (which states that the force required to extend or compress a spring by a given distance scales linearly with that distance [52]), connecting each nucleosome to the next. Nucleosomes are free to move according to Brownian motion, subject to certain constraints:

- Tension forces exerted by DNA linkers (modelled as ideal springs);
- Repulsion forces between nucleosomes that are intersecting, or very close to

each other;

- Angular tendencies between adjacent DNA linkers;
- Optionally: interactions between "condensin binding sites" spaced along the string.

In the original implementation, the vast majority (≈95%) of the runtime is spent computing the repulsion forces that prevent particles from overlapping. This kernel is an excellent candidate for the projection sorting algorithm.

Repulsion Forces

Two nucleosomes cannot occupy the same area in space at the same time. In order to enforce this in the simulation, a force is included that repels pairs of nucleosomes whose centres come very close to each other. Figure 5.2 illustrates the magnitude of this force as it varies by distance. 15 nm is the cut-off radius r_c outside which no repulsion forces apply between two nucleosomes, and is equal to twice the nucleosome radius of 5 nm plus an extra 5 nm to deter possible collisions on the next timestep. c_3 is an experimentally determined value set to 10 pN.

This is a *spatial force*: it applies between close nucleosomes, independent of their "string order" as connected by DNA linkers. This is analogous to Van der Waals [87] forces in other MD simulations, where a combination of cell lists and Verlet lists are often used to facilitate computation. In the original implementation of this simulation cell lists were used exclusively.

Condensin Interaction Forces

The application supports two distinct modes of simulation, each representing a different phase of the cell cycle: interphase and mitosis. Mitosis simulation differs from interphase simulation in that it includes additional forces between "condensin

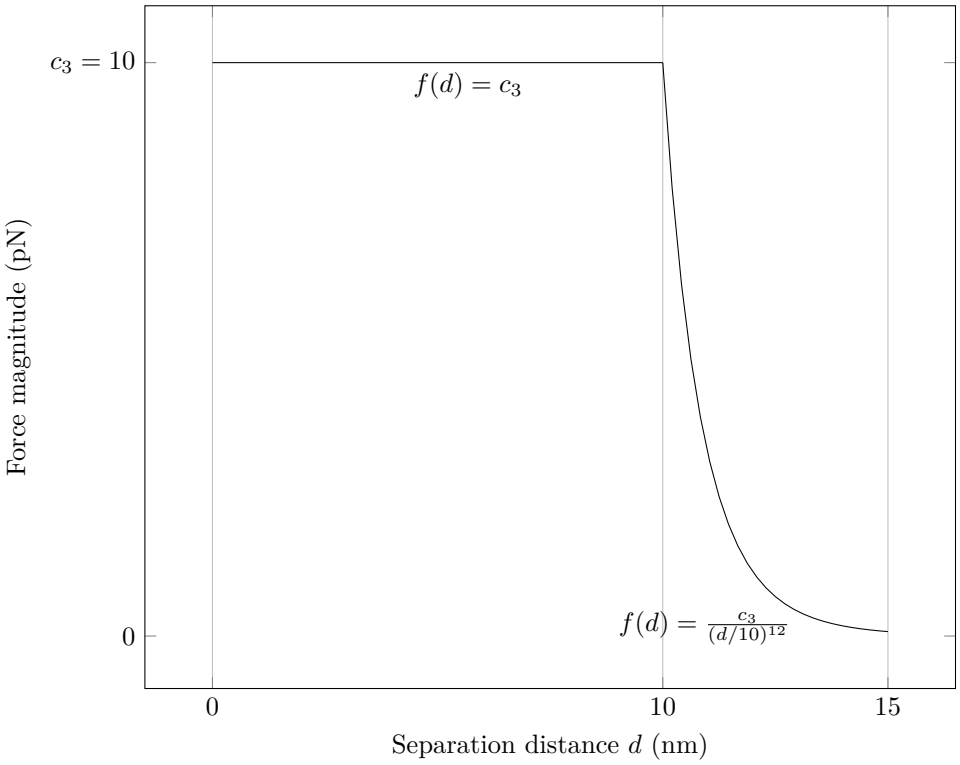


Figure 5.2: Repulsion force on nucleosome pairs within 15 nm. A force of 10 pN repels pairs within 10 nm, which then falls off rapidly to near-zero at 15 nm. After 15 nm, no repulsion forces apply. c_3 is an experimentally determined value, set to 10 pN.

binding sites", which correspond to points on the string which are rich in specific "condensin" proteins.

On average every 48th nucleosome is designated a condensin binding site, When simulating mitosis, these sites may bind together for extended periods, causing the condensation of the chromosome over time. Cheng et al. [20] test various hypothetical models for these interactions and compare the results between simulations where no condensin interaction forces are applied against each model in turn, and see which best fits the experimental data.

These forces are also spatial in nature. For two sites to bind together, they must come within 40 nm. When this occurs, sites remain bound until either they are pulled apart by other forces, or a "dissociation event" occurs, which has a small probability of happening each timestep. When two sites dissociate, they enter a cooldown period of three timesteps, during which they may not rebind. When two sites are bound, forces regulate the distance between them, which averages $c_5 = 30\,\text{nm}$.

Linear Forces

The tension forces and angular tendencies are products of the DNA linkers. They apply between adjacent nucleosomes in string order, hence we refer to them together as *linear forces*. Figure 5.3 demonstrates how these forces are applied in the simulation. Tension forces regulate the distance between adjacent nucleosomes, implemented based on Hooke's Law for ideal springs. This has the interesting side effect of enforcing some level of spatial locality (see Section 3.1.1) as an *inherent property* of the simulation. That is to say, when the nucleosomes are stored in memory based on string order, they are implicitly partially ordered spatially, due to the tension forces preventing adjacent nucleosomes in string order from becoming spatially distant. The spring constant is set $K_s = 50\,\text{pN}\,\text{nm}^{-1}$ and the relaxed spring length $c_2 = 15\,\text{nm}$, both based on experimental data.

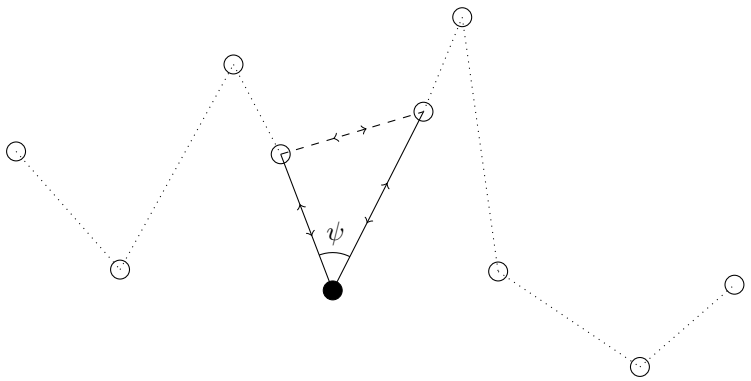


Figure 5.3: Linear forces: the solid lines indicate the tension forces applied by the DNA linkers on the filled nucleosome and its neighbours. The dashed line indicates a weak force designed to regulate the angle ψ between each triplet of consecutive nucleosomes.

The relative angles by which the DNA linkers emanate from each nucleosome are regulated by a weak force between alternating pairs of nucleosomes. The forces are tuned such that the mean value of $\psi = 70°$ based on experimental data.

Entropic Forces

The Brownian motion is implemented by applying an independent, randomly directed force of magnitude $c_1 = 24.5\,\text{pN}$ to each nucleosome. This force simulates the action of diffusion within a cell and produces movement consistent with experimental data.

The simulation uses the midpoint method, a second-order Runge-Kutta method [36], to numerically calculate nucleosome trajectories based on the forces above. This is illustrated in Equation 5.1.

$$\begin{aligned} \vec{x}_{t+\frac{\Delta t}{2}} &= \vec{x}_t + \frac{\Delta t}{2}\vec{f}_t \\ \vec{x}_{t+\Delta t} &= \vec{x}_t + \Delta t \vec{f}_{t+\frac{\Delta t}{2}} \end{aligned} \tag{5.1}$$

Given nucleosome positions at time t, $\vec{x}_t$, the acting forces $\vec{f}_t$ are calcu-

lated. These are used to calculate the "midpoint" positions, $\vec{x}_{t+\frac{\Delta t}{2}}$, where Δt represents an *in silico* timestep. Each *in silico* timestep corresponds to approximately 1.69×10^{-5} s. Forces are then calculated again for these new positions, $\vec{f}_{t+\frac{\Delta t}{2}}$, and finally used to calculate the updated positions at time $t + \Delta t$, $\vec{x}_{t+\Delta t}$. Experiments need to run for up to 40×10^6 *in silico* timesteps.

5.2 Optimisations

Prior to using the simulation in our experiments we performed some optimisations, both to assist the domain scientists, and to ensure that the code was fit for comparing short-range force algorithms.

5.2.1 Parallelisation

The original implementation of the simulation was serial so our first step was to parallelise it. We chose to do this with OpenMP [69], as a widely supported and understood open standard. As this is a shared memory parallelisation we chose to use a particle decomposition (see Section 3.2). Each thread takes a contiguous length of the string and is responsible for all nucleosomes within.

5.2.2 Cell Lists and Verlet Lists

It was mentioned above that the original implementation used cell lists to calculate repulsion forces, and that these calculations consumed around 95% of the overall runtime. Given the small cut-off radius in this simulation, it is desirable to construct cell lists based on a fine decomposition, otherwise there will be many spurious checks. Calculating these naïvely (by allocating a three dimensional array of bins) uses an infeasibly large amount of memory, which increases cubically with the problem size. As we wish to test the simulation over a variety of datasets, we implemented an alternative approach using a lock-free hash map (based on Marçais et al. [57]) to

construct the lists in a much smaller amount of space, without sacrificing high performance when running multiple threads.

We then implemented Verlet lists on top of this optimised cell list construction. Verlet list rebuilds and force computation using Verlet lists were hand-vectorised as described by Pennycook et al. [70].

5.2.3 Linear Forces

The tension and angular forces were originally calculated separately. As they both require scanning through the particles in string order we obtained a speedup by combining their calculation into a single "linear" kernel.

5.2.4 Entropic Forces

The original Pseudo Random Number Generator (PRNG) used is not thread safe. We replaced this with one that is thread safe, and also amenable to compiler auto-vectorisation. This came at a slight performance cost as the substitute generator uses more state (providing higher quality random numbers).

For the results of these optimisations we refer the reader to our comparison of the original and optimised simulation in Section 6.5.

5.3 Summary

In this chapter we have discussed the basic theory and implementation of the simulation which we will use as the basis for the experimental part of the work in Chapters 6 and 7. Specifically we have covered the physical justification for each of the forces included in the simulation, specific parameters in the implementation and some of the computational factors. We briefly surveyed some basic optimisations that we made in preparation for the work in the upcoming chapters. We now move to implementing and evaluating the projection sorting algorithm.

Chapter 6

Implementing Projection Sorting On-node

In Chapter 4 we described the three phases of the projection sorting algorithm, which we list again here:

1. Selecting axis of projection $\boldsymbol{v}$;
2. Calculating and sorting particle projections;
3. Calculating forces based on projections.

In this chapter we propose an answer to Research Question 2 (RQ2):

> **RQ2:** How might [projection sorting] perform (and how might it be best implemented) in a shared-memory parallel environment?

To do this we implement projection sorting in the context of the simulation described in Chapter 5. Specifically we reframe the calculation of the repulsion forces in terms of projection sorting. We then compare the implementation empirically against Verlet lists, the state of the art for pairwise short-range force computations.

In terms of hardware, we implement two versions: Intel Haswell and Intel KNC. We chose these architectures so as to represent both the multi- and many-core architectures which dominate the HPC landscape today. An alternate popular choice for many-core implementations is NVIDIA's GPU architectures. We decided on Xeon Phi on the basis that MD on GPU is already well studied: the explosion in popularity of GPGPU programming has left few scientific fields untouched. Xeon Phi is a much newer architecture whose characteristics, advantages and disadvantages are less well understood.

6.1 Experimental Setup

Throughout this chapter and the next we present experimental results based on our implementations. Below we summarise the setup for these[1].

6.1.1 Datasets

The initial dataset described by Cheng et al. [20] was derived from a budding yeast cell and contains 2000 particles. For three main reasons, this is not large enough for the experiments that we conduct in this chapter and in Chapter 7. Firstly, the cited work from Cheng et al. represents an initial foray into this area, with the intent to continue with larger datasets in the future. Therefore we need to ensure that our algorithm is still effective as the dataset size increases. Secondly, we wish to stress the memory subsystems of the hardware involved, and to do so we need large enough datasets that do not fit into the higher levels of the memory hierarchy. Thirdly and finally, the scaling studies that we conduct require larger datasets. For example, if we were to strong scale a simulation of the 2000 particle dataset to 512 cores, each core would only be responsible for approximately 4 particles, at which point scaling would obviously be limited by programming overheads.

[1]Note that there are some differences in the setup for the experiments in Chapter 7, which we cover in Section 7.1.

As no larger real datasets were available while this work was being undertaken, we generate extended versions of the original using statistical methods. Synthetic datasets were generated for $N_{ext} = 2^k \cdot 10^3$, $2 \leq k \leq 11$. The largest of these, with $2^{11} \cdot 10^3 = 2{,}048{,}000$ particles, satisfies the above criteria. Below we describe the process used to create these datasets.

We define three normal distributions, each parameterised using the sample mean and sample standard deviation of the positional differences between successive particles in string-order along each axis, x, y and z in the original dataset. There are N particles, and therefore $N - 1$ differences. Equations 6.1, 6.2 and 6.3 show how these quantities are calculated. We represent the difference between the positions of particles i and $i + 1$ by $\boldsymbol{\Delta}_i$, the component-wise sample means of these differences by $\boldsymbol{\mu}$, and the component-wise sample standard deviations by $\boldsymbol{\sigma}$.

$$\boldsymbol{\Delta}_i = \boldsymbol{x}_{i+1} - \boldsymbol{x}_i \tag{6.1}$$

$$\boldsymbol{\mu} = \frac{1}{N-1} \sum_{i=1}^{N-1} \boldsymbol{\Delta}_i \tag{6.2}$$

$$\boldsymbol{\sigma} = \sqrt{\frac{\sum_{i=1}^{N-1} (\boldsymbol{\Delta}_i - \boldsymbol{\mu})^2}{N-2}} \tag{6.3}$$

We then generate new conformations of length N_{ext} particles by sampling these distributions to perform an $N_{ext} - 1$ step random walk. Starting by placing a particle at the origin, the walk proceeds by sampling each normal distribution to construct an offset from the current position, and placing a new particle at that point. The construction of the position of particle $i+1$ from the position of particle

i is shown in Equation 6.4.

$$\begin{aligned} \boldsymbol{x}_1 &= \mathbf{0} \\ \boldsymbol{x}_{i+1} &= \boldsymbol{x}_i + (\mathcal{N}_x(\boldsymbol{\mu}_x, \boldsymbol{\sigma}_x), \mathcal{N}_y(\boldsymbol{\mu}_y, \boldsymbol{\sigma}_y), \mathcal{N}_z(\boldsymbol{\mu}_z, \boldsymbol{\sigma}_z)) \end{aligned} \tag{6.4}$$

Finally condensin binding sites are placed randomly on average every 48 particles. After generation we simulate the conformation in interphase (see Section 5.1.1 for discussion of the difference between interphase simulation and mitosis simulation) for 100,000 timesteps to reach a stable state, free of artefacts caused by the random walk process. These artefacts can include particles being placed within repulsion range of each other, far enough apart that tension forces pull them together, or in configurations requiring angular correction (see Section 5.1.1 for discussion of these forces).

6.1.2 Machine Specifications & Compilation

To run the experiments, two machines were used. Firstly, Tinis, a cluster at the University of Warwick consisting of 203 nodes, each with 2 Intel Xeon E5-2630 v3 processors, for 3488 cores in total. Each node is equipped with 64 GB of RAM. Single Tinis nodes were used to run the Xeon experiments.

Secondly we used Chiron, also located at the University of Warwick. This machine is host to several accelerator nodes, 2 of which are equipped with 2 Intel Xeon Phi 7120P coprocessors. Single coprocessors were used to run the Xeon Phi experiments. We refer the reader to Table 6.1 for more details.

All code was compiled using the Intel C++ compiler, v15.0.4 with the following performance flags: `-O3 -ansi-alias -qopt-assume-safe-padding -ipo -qopenmp`. In addition, `-qopt-threads-per-core=T` was used, with `T` set to the number of hardware threads in use per core (usually 1 on the CPU and 4 on the coprocessor for best performance). CPU AVX2 vectorisation was forced with `-xCORE-`

	Xeon E5-2630 v3	Xeon Phi 7120P
Sockets×Cores×Threads	2×8×2	1×61×4
Clock (GHz)	2.40	1.24
L1{i,d} / L2 cache (KB)	{32,32} / 256	{32,32} / 512
Memory (GB)	64	16
SIMD ISA	AVX2	KNC

Table 6.1: Summary of node hardware configurations on Tinis and Chiron.

`AVX2`. Threads were pinned sequentially at runtime using a combination of the `KMP_PLACE_THREADS` and `KMP_AFFINITY` environment variables.

6.1.3 Validation

Simulation Validation

Our optimised version of the simulation has been independently validated by the domain scientists that authored the original. The simulation is deterministic (given a fixed random seed), and we can therefore ensure that the results produced are accurate (and be confident that the time to solution is meaningful) by checking that they do not deviate from run to run.

Results Validation

Detailed timing information was collected across all areas of the code using the `rdtsc` hardware counter in order to achieve high accuracy with minimal overhead. This counter reports the number of CPU clock cycles which have passed since some fixed time in the past. We can convert these measurements to seconds using knowledge of the CPU's clock speed. In practice we do this conversion by measuring the total simulation time using the OpenMP `omp_get_wtime()` function which returns the number of seconds since some fixed time in the past as a double-precision value. We take quotient of the number of clock cycles counted over the same period and the measured wall time, and multiply each counter by this value to obtain the time in

Timer ID	Comments
VL_MISC	VL only. Miscellaneous.
VL_BIN	VL only. Cell list construction.
VL_BUILD	VL only. Verlet list construction.
VL_WAIT	VL only. Thread barriers.
PS_MISC	PS only. Miscellaneous.
PS_SORT	PS only. Per-thread particle sort.
PS_MERGE	PS only. Parallel particle merge.
PS_WAIT	PS only. Thread barriers.
ENTROPIC	Entropic forces.
LINEAR	Linear forces.
COND_MISC	Miscellaneous condensin forces.
COND_BIN	Binding site binning.
COND_INT	Binding site interactions.
COND_WAIT	Condensin thread barriers.
REPULSION	Repulsion forces.
REPULSION_WAIT	Repulsion thread barriers.
INTEGRATION	Timestep integration.
CHECKPOINT	Output checkpointing.
WAIT	Miscellaneous thread barriers.
COMM	MPI comms (see Chapter 7).
OFFLOAD_TX	KNC offload data transfer (outbound).
OFFLOAD_RX	KNC offload data transfer (inbound).

Table 6.2: List of timers used for data collection.

seconds. We take this approach rather than measuring in seconds directly to ensure that there is no contention between the many threads trying to measure the time simultaneously.

The results reported do not include time spent in initialisation or shutdown, as this becomes a negligible fraction of the total runtime at the scale of full simulations. It should be noted that the initialisation phase performs some "dummy calculations" in order to warm up the caches prior to starting the simulation proper. Table 6.2 lists each individual timer used for data collection in the experiments (we report aggregate results based on the measurements from these timers).

Each experiment was allocated a full node (nodes in Chapter 7), to ensure that there was no contention for hardware from other users. Each experiment was repeated five times. As the simulation is deterministic and performs exactly the same

calculations on each run with the same inputs, we can be sure that the variation between these results is due to background noise on the node (i.e. operating system processes). Because this variation is orthogonal to what we are actually testing (i.e. the runtime of the simulation), we select the lowest runtime out of these five. The purpose of doing five runs is therefore to establish the impact of system noise on the results, however in all cases we observed very little variation.

6.1.4 Parameter Selection

All our experiments have a duration of 1000 timesteps. In theory, as each timestep does exactly the same thing, it does not matter how many we choose to execute, as long as we are consistent. In practice however, there are two interacting constraints that led us to choose 1000 as the length for our experiments. Firstly it is necessary to run for long enough that the minor variations observed due to background system noise on the node discussed in the previous chapter are negligible in comparison. Secondly, in Chapter 7 we run strong scaling studies, which necessitates simulating large datasets at a low core count. This takes a very long time, and the machines on which we run have upper limits on the amount of time which may be requested. We selected 1000 timesteps as a number which is feasible to use when simulating large datasets with a small number of cores, and provides sufficient total runtime when simulating those datasets with a large number of cores.

When using Verlet lists, we set the skin distance to the empirical minimum $r_s = 40\,\text{nm}$, and $k = 2$, which yields the best performance for the given value of r_s.

6.2 Repulsion Forces

We now discuss the implementation of the two performance-critical components of the projection sorting algorithm—the force sweep and the global particle sort. Projection sorting is used in the context of the simulation described in Chapter 5 to

provide a means for the calculation of repulsion forces which prevent particles from overlapping each other (see Section 5.1.1 for a more detailed discussion).

The Structure-of-Arrays (SoA) data layout (where each particle facet is laid out independently and contiguously in memory, cf. Array-of-Structs (AoS)) is used throughout the code for position and force arrays to facilitate vectorisation. As a result, the compiler is able to auto-vectorise the simpler kernels (the entropic, tension and angular forces, and the integration), with a small amount of help in the form of `#pragma` directives. Those that do not auto-vectorise, including the projection sorting implementation, have been hand-vectorised using both AVX2 and KNC intrinsics (depending on whether the code is being built for CPU or the coprocessor respectively).

6.2.1 Force Sweep

Due to the nature of vectorisation, all instructions within a branch must be executed if any of the lanes trigger the condition. In this case, if any particles in the vector pass the cut-off check at Line 11 in Listing 5, all the instructions within that branch must be executed for all vector lanes. This introduces inefficiency, as the scalar version only executes the inner branch on a per-particle basis as necessary. The actual inefficiency depends on the proportion of particles that are within the cut-off distance. For a SIMD width of W, up to $W - 1$ of the force computations carried out inside the branch could be unnecessary.

Inefficiency also comes from padding to a multiple of the vector width at the end of each force sweep, and from redundant computation due to alignment requirements at the start of each force sweep. Each sweep must continue until all particles in the vector fail the projection cut-off check, which implies a maximum wastage of $2W - 2$. The worst case for wastage due to alignment is $W - 1$, so for a bidirectional pair of sweeps, the maximum wastage is $6W - 6$. The longer the sweep, the smaller a fraction of the total number of particles processed this will

```
for (i = 0; i < nb; i++) {
    fxi = 0.0;
    fyi = 0.0;
    fzi = 0.0;

    // step in one direction
    for (j = i + 1; j < nb; j++) {

        // are we stepping out of the block of candidate neighbours?
        if (projs[j] - projs[i] > rc) break;

        // full distance check
        dx = p_x[j] - p_x[i];
        dy = p_y[j] - p_y[i];
        dz = p_z[j] - p_z[i];
        dsq = dx*dx + dy*dy + dz*dz;
        if (dsq <= rcsq) {

            // force calculation
            coeff = /* app-dependent */;
            fxi += coeff * dx;
            fyi += coeff * dy;
            fzi += coeff * dz;
        }
    }

    // step in the other direction
    for (j = i - 1; j >= 0; j--) {
        if (projs[i] - projs[j] > rc) break;
        // ...
        // as above
        // ...
    }

    // accumulate forces on particle i due to neighbours
    f_x[i] += fxi;
    f_y[i] += fyi;
    f_z[i] += fzi;
}
```

Listing 5: Skeleton implementation of the force sweep.

account for, leading to better vector efficiency. With real datasets, the sweeps are quite short so the inefficiency can be significant. We explore the empirical values for these inefficiencies in detail in Section 6.3.

In addition to inefficiency arising from simply performing unnecessary computation, it is also necessary to ensure that these superfluous calculations do not affect the results. This is achieved by using blend operations to set the out-of-range values to zero, so that when the vector is written to memory only the valid values are updated. This requires 2 extra comparison operations on both the CPU and the coprocessor, 3 extra blend operations on the CPU, and the addition of masking to the final triplet of fused multiply-add instructions used to accumulate forces on the coprocessor.

The force sweep is very cache friendly as all accesses are contiguous. Hardware counter analysis for a representative run reveals that 99.8% of loads issued hit L1 cache. This minimises delays in getting data into the vector registers.

Appendix A gives full listings for our intrinsic implementations.

6.2.2 Sorting

The other computationally intensive component of the projection sorting approach is the particle sort. Each thread uses a tuned in-place Quicksort to sort the particles under its control. We can exploit the partially ordered conformation at each step to accelerate the sort. Pairs of sorted blocks are then merged iteratively using the balanced asynchronous parallel merging algorithm described by Francis and Mathieson [30]. This ensures that each thread merges an even portion of the input sequences. For P threads, $\lceil \log_2 P \rceil$ layers of merging are required.

Each particle consists of five pieces of information—the value of its scalar projection, its (x, y, z) coordinates in space, and its status as a condensin binding site. We sort an array of indices first, keyed by the projection values. These indices are then used to reorder the other four arrays in linear time.

	SSE 4.2 (128-bit)	AVX2 (256-bit)	KNC (512-bit)
# Lanes (W)	2	4	8
Transpose	3	8	24
Sort	6 (3+3)	28 (20+8)	100 (76+24)
Bitonic merge	15	26	36

Table 6.3: Number of instructions as it scales with W for various in-register sorting operations.

	Speedup
SSE 4.2	1.30×
AVX2	2.02×
KNC	1.31×

Table 6.4: Relative performance for vectorised sorting (versus scalar sorting) per SIMD ISA. The scalar implementations use a comparison based sort rather than bitonic networks.

Using the SoA format enables the use of vectorised in-register sorting techniques. Bitonic sorting networks [10] are frequently applied here in the literature, as they fit well with existing SIMD ISAs. We use the in-register sorting/merging scheme described by Chhugani et al. [21], implemented with SSE, AVX2 and KNC intrinsics. Although the size of these networks scales poorly with the SIMD width W (see Table 6.3) we see reasonable speedups (see Table 6.4).

We also explored several alternate sorting strategies, including the non-comparative radix sorting family of algorithms [78, 81]. As we are sorting based on 64-bit key values (projections are stored in double precision floating point format) these are at a disadvantage, as they require splitting each key up into chunks and performing several passes, checking each chunk in turn. Larger key sizes require more passes, which takes longer. We found the optimal chunk size (or *radix*) to be 16 bits, but the Quicksort + asynchronous merge algorithm described above still yielded better performance.

There exist hybrid strategies, for example: doing one pass of radix-style binning to split the keys up, and then sorting bins in parallel with multiple threads.

We found load balancing to be the main issue with this; the IEEE 754 floating point standard [3] makes it difficult to partition many close values finely enough to achieve similarly sized bins.

6.3 Projection Sorting vs. Verlet Lists

As discussed in Chapter 3, Verlet lists are the *de facto* standard approach to pairwise short-range force calculations. In this section we investigate how the performance of the projection sorting approach compares.

To fairly compare the two, we need to choose values for the skin distance r_s and rebuild period k that maximise the performance of the Verlet list approach while still computing correct results. The values $r_s = 40\,\text{nm}$ and $k = 2$ were determined by tracking the maximum distance moved by any particle over an experiment using the projection sorting method, and setting r_s to just greater than that, ensuring that the results are correct. k was then chosen to maximise performance.

6.3.1 Distance Check Counts

A "distance check" is a calculation of the distance between a pair of particles, necessary to determine whether we need to calculate the force between them. The number of distance checks performed by an algorithm is a good predictor of its performance [86]. Table 6.5 shows the average number of distance checks performed using each method, with N3 on and off, as well as the SIMD inefficiency for AVX2 and KNC intrinsic implementations (i.e. the number of distance checks that were unnecessary, and only performed as a result of SIMD limitations).

Projection sorting performs fewer distance checks overall, but is affected more by SIMD inefficiencies. As discussed above, when N3 is not used projection sorting requires two force sweeps. There is SIMD inefficiency at the end of both of these sweeps, and also at the beginning of the sweep due to alignment requirements. Verlet

Algorithm	N3?	# checks	AVX2 ineff. (#/%)	KNC ineff. (#/%)
PS	N	64.73	10.15 (13.55%)	22.03 (25.39%)
	Y	32.41	6.00 (15.54%)	13.98 (30.13%)
VL	N	91.74	1.42 (1.52%)	3.54 (3.72%)
	Y	45.83	1.57 (3.31%)	3.60 (7.27%)

Table 6.5: Mean number of distance checks performed per particle, and the number of unnecessary checks performed as a result of SIMD inefficiencies for AVX2 (4-wide) and KNC (8-wide) implementations. The dataset used contained 128,000 nucleosomes, with $r_s = 40$ and $k = 2$.

lists only require one sweep regardless of N3, and have no alignment requirements as they are allocated on a cache line boundary. As we scale up to wider SIMD, we see the projection sorting technique approaching the operation of Verlet lists in terms of the number of distance checks performed. At current SIMD widths however, projection sorting still requires the fewest checks in all cases.

6.3.2 Periodic Costs

A key part of the Verlet list algorithm is the use of cell lists to accelerate the list build phase. The simulation space is discretised into cubes of side r_v (the Verlet radius, $r_v = r_c + kr_s$) and particles are binned accordingly. While this step is necessary (construction of the Verlet lists takes time quadratic in the number of particles otherwise), using a naïve three dimensional array method means the cell lists consume a very large amount of memory. This is especially true in our case due to the non-uniform particle density. Many of the cell lists are empty, but space for at least a pointer must still be pre-allocated.

As the conformation is concentrated in a small portion of simulation space, we instead choose to implement the construction using a lock-free hash table, where cell lists are only allocated when a particle actually needs to be added. Once an allocation has occurred we do not free the memory until the end of the simulation. This is done to avoid the large overhead of continually freeing and reallocating

Algorithm	N3?	# calcs.	AVX2 ineff. (#/%)	KNC ineff. (#/%)
PS	N	1.68	4.74 (73.83%)	10.51 (86.21%)
	Y	0.84	2.38 (73.91%)	5.47 (86.69%)
VL	N	1.68	3.53 (67.75%)	7.80 (82.28%)
	Y	0.84	2.05 (70.93%)	4.69 (84.81%)

Table 6.6: Mean number of full neighbour force calculations performed per particle, and the number of unnecessary calculations performed as a result of SIMD inefficiencies for AVX2 (4-wide) and KNC (8-wide) implementations. The dataset used contained 128,000 nucleosomes, with $r_s = 40$ and $k = 2$.

memory that is likely to be reused anyway. Using atomic operations rather than mutexes ensures internal consistency with minimal performance penalties. This method is slower than simply allocating all bins at the start, but uses orders of magnitude less memory, and as such is feasible for larger datasets.

Figures 6.1a and 6.2a compare the costs of Verlet list rebuilds using this scheme and the particle sort required by the projection sorting algorithm. The sort is clearly cheaper than the Verlet list rebuild, even though it is performed 4 times as often (with $k = 2$).

6.3.3 Force Sweep Costs

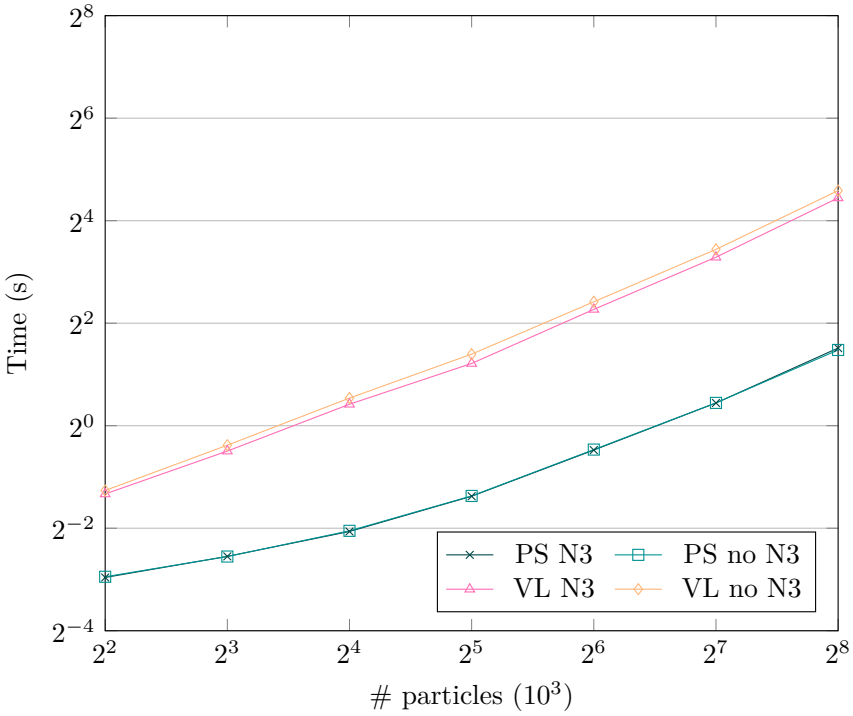
Finally, we compare the cost of the force sweeps. Vectorisation is a major consideration, and Table 6.6 shows the empirical values for the inefficiency arising from wasted computation inside the force calculation branch. As discussed in Section 6.2.1, we see very high fractions approaching $\frac{W-1}{W}$ here due to the low rate of interactions between particles (brought on by the small cut-off distance). This impacts the overall SIMD speedup as the width increases. Verlet lists have slightly lower inefficiencies as they preserve the order of particles better than projection sorting.

Figures 6.1b and 6.2b show the full sweep comparison. Interestingly, the fastest option here is projection sorting with N3 *disabled*. N3 cuts the number of distance checks in half, so one would assume that it should be faster. However as

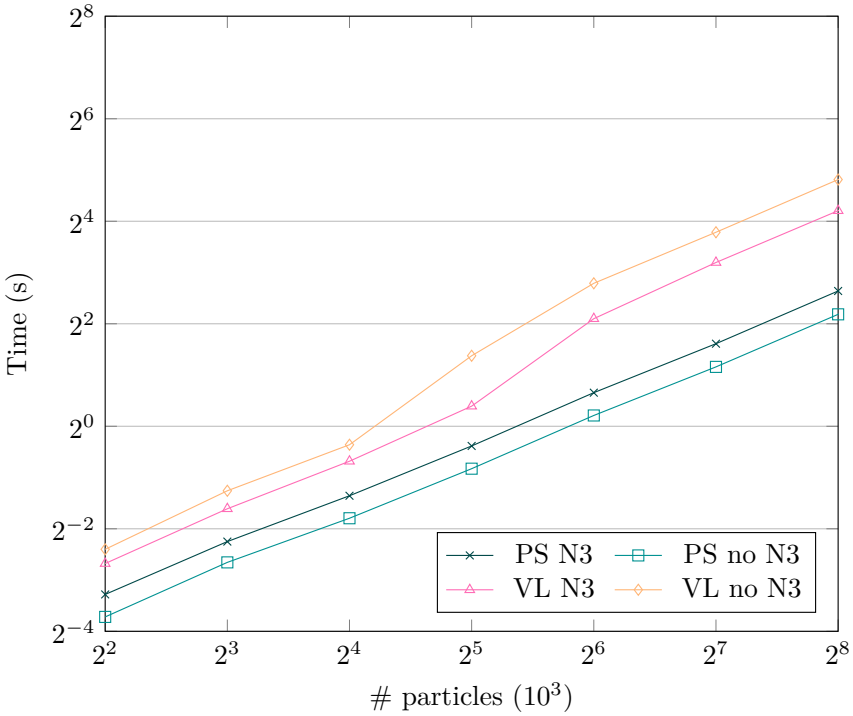
iterations of the outer loop over particles occur concurrently, and N3 introduces memory stores to particles other than the one indexed by the outer loop, using this strategy requires that we introduce a measure to prevent data-races in updating the force array. There are two main ways of doing this, firstly by means of thread-local storage, where each thread writes its forces to a separate location in memory, and each thread's contributions are added up at the end. The second is to make force updates atomic at the hardware level, so that one thread cannot read an invalid value during another thread's store. Both approaches were tried, and it was determined that atomics were more efficient on the present hardware (likely due to the low number of particles that pass the distance-checks). Despite this, the additional cost of atomic operations outweighs the benefits provided by N3. The gap is especially pronounced on Xeon Phi, as it is running 15× as many threads. As is well established, N3 improves performance for Verlet lists.

In conclusion, projection sorting wins on all fronts in these tests, exhibiting the lowest number of distance checks, the cheapest periodic costs, and the fastest force sweeps. The non-uniformity in the particle density and the fast moving particles (necessitating a high value for r_s) are the primary reasons that Verlet lists are ineffective for this simulation. It is clear that projection sorting can be an effective alternative. The primary factors to consider when choosing an algorithm are (in order of importance):

- Uniformity of the particle densities;
- Average movement of particles per timestep (lower allows for a smaller r_s, and therefore better Verlet list performance);
- Projection sorting uses memory bandwidth more effectively;
- Higher SIMD width favours Verlet lists.

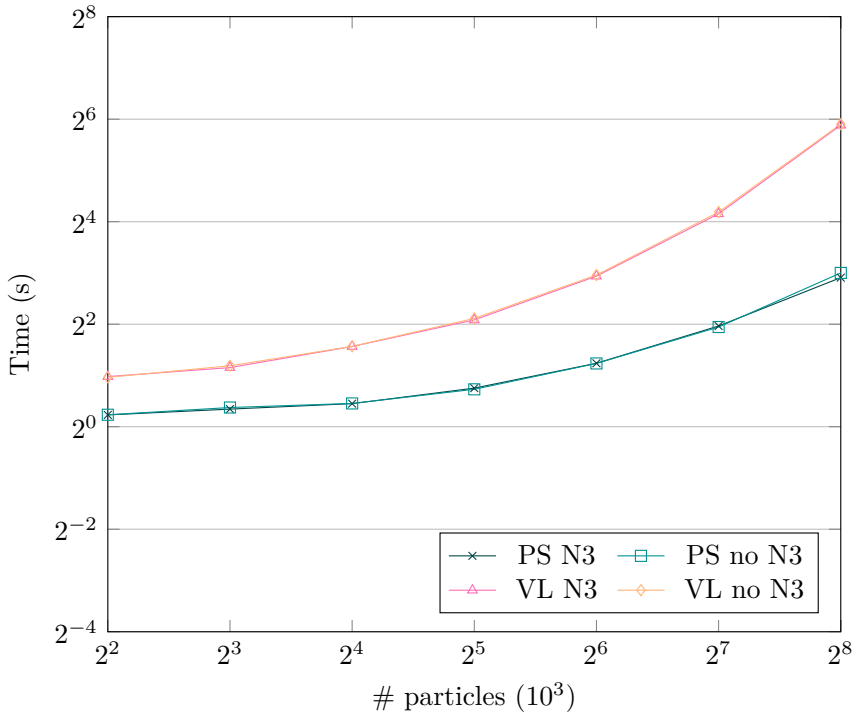


(a)

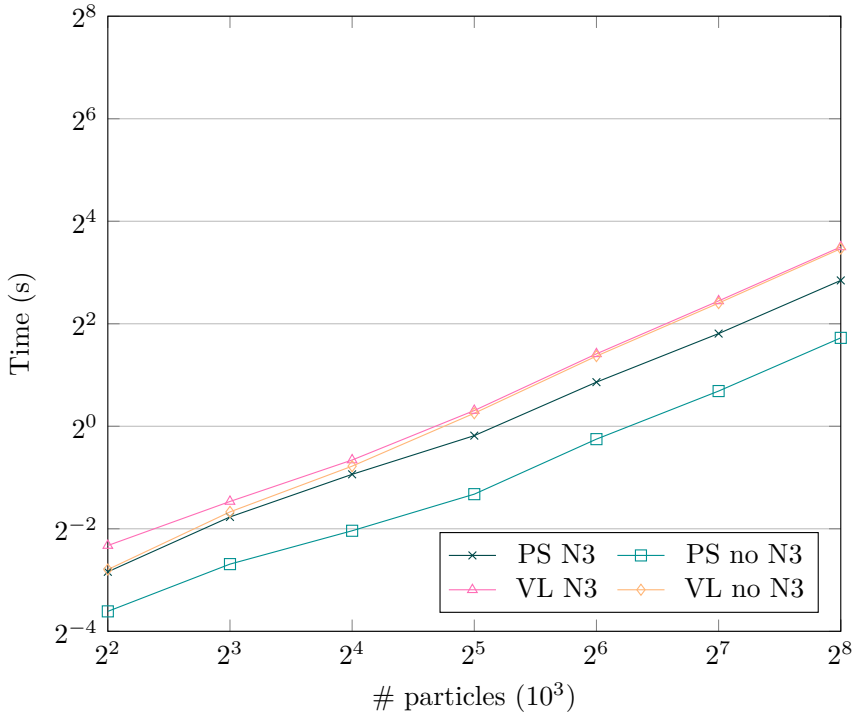


(b)

Figure 6.1: Performance figures for PS kernels and VL kernels on a Tinis node running 16 threads, from 4000 up to 256,000 particles. (a) shows periodic costs and (b) shows force calculation costs. Lower time is better.



(a)



(b)

Figure 6.2: Performance figures for PS kernels and VL kernels on a Chiron coprocessor running 244 threads, from 4000 up to 256,000 particles. (a) shows periodic costs and (b) shows force calculation costs. Lower time is better.

6.4 Condensin Interaction Forces

The other computationally intensive force calculation pertains to the interactions between "condensin binding sites"—modelled as special nucleosomes occurring along the length of the string at irregular intervals, with an average separation of 48 nucleosomes. These sites can interact when they come close, and become stuck together for extended periods, prompting the condensation of the string over time.

Sites whose centres come within 40 nm of each other experience attractive forces, up to a limited number of interactions per site, per timestep (typically capped at 1 or 2). There is also a stochastic component: for each interaction, and each timestep there is a small configurable probability that interacting sites will dissociate from each other. When this happens they enter a cooldown period of 3 timesteps during which they cannot form any bonds, giving them time to move apart.

There are two primary steps to computing the forces on each site, referred to henceforth as the **binning** step and the **interaction** step respectively:

1. List other sites within the 40 nm cut-off radius,
2. Determine whether to apply forces, dissociate, or advance cooldown period, depending on the number of close pairs and their interaction history.

This is another pairwise short-range interaction, although with different properties to the repulsion force. The cut-off radius is larger, 40 nm compared to 15 nm. The force between a pair of sites is more expensive to calculate, but the number of sites is an order of magnitude smaller than the number of nucleosomes. For a given site, we must determine all other close sites before we can compute any forces, rather than accumulating them per interaction as in the repulsion kernel. The cooldown mechanic also introduces additional state between timesteps, which complicates matters. Projection sorting can be used during the binning step, although careful attention must be paid to correctly mapping from sorted binding sites to the inter-timestep state.

6.4.1 Storage

It is necessary to store the cooldown status for every pair of binding sites—a flag indicating whether a site's interaction with another site is currently in cooldown mode, and the number of timesteps remaining before it is free to interact again. While the two can be combined into a single field (with 0 representing no cooldown mode, and any other number representing the remaining count), the naïve storage requirement is still quadratic in the number of sites. This becomes a problem with larger datasets.

As dissociation events are uncommon, the matrix of cooldown state is very sparse. Taking advantage of this fact, we implement the same technique used to reduce the storage requirement for cell lists, and replace the matrix with a lock-free hash table. For a large number of binding sites, say 100,000, this approach requires over 2300× less space, 4.1 MB instead of 9.3 GB.

6.4.2 Vectorisation

Meaningful vectorisation is infeasible for both the binning and interaction steps. The binning step requires access to the cooldown status of each site. As these are stored non-contiguously regardless of the storage strategy used, we are faced with an expensive gather operation. In the case of the hash table, current SIMD ISAs do not support atomic gathers [51], necessitating performing the memory accesses and register insertions manually. More crucially though, the average binding site sweep length is slightly under 2, which negates any benefit due to the large overhead. For the interaction step the algorithm dictates that each site is processed individually based on the contents of a very short list of neighbour sites.

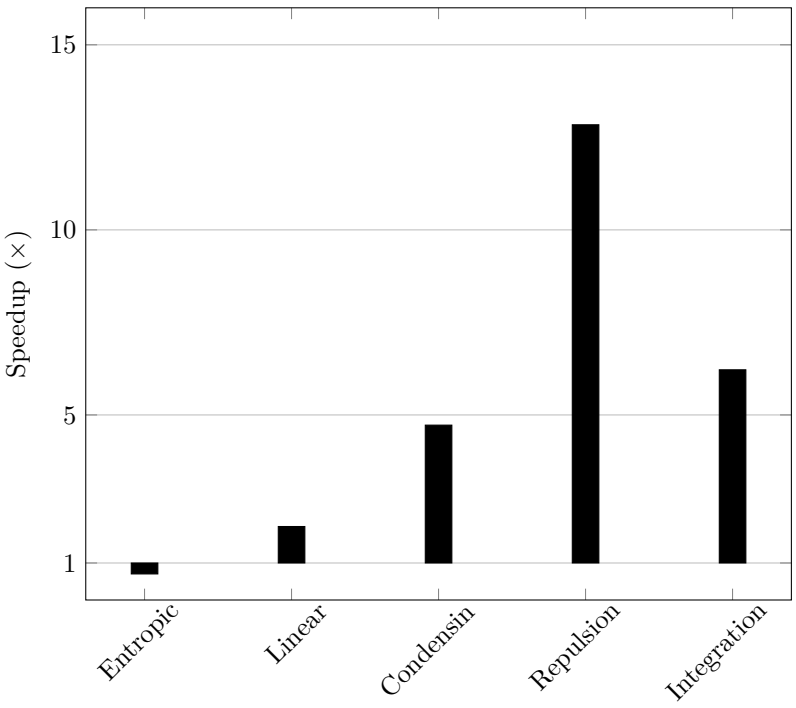


Figure 6.3: Per-kernel single-threaded speedup observed for the optimised implementation relative to the original simulation for the 2000 particle dataset. Higher speedup is better.

6.5 Overall Performance

Relative to the original code, we see single-threaded speedups starting at over 10× on the CPU for 2000 particles (see Figure 6.3 for a per-kernel breakdown), and increasing as the dataset size goes up due to better algorithmic scaling. We would emphasise that this is not a fair comparison of algorithmic approaches (which we previously presented in Section 6.3), as the original implementation is not heavily optimised.

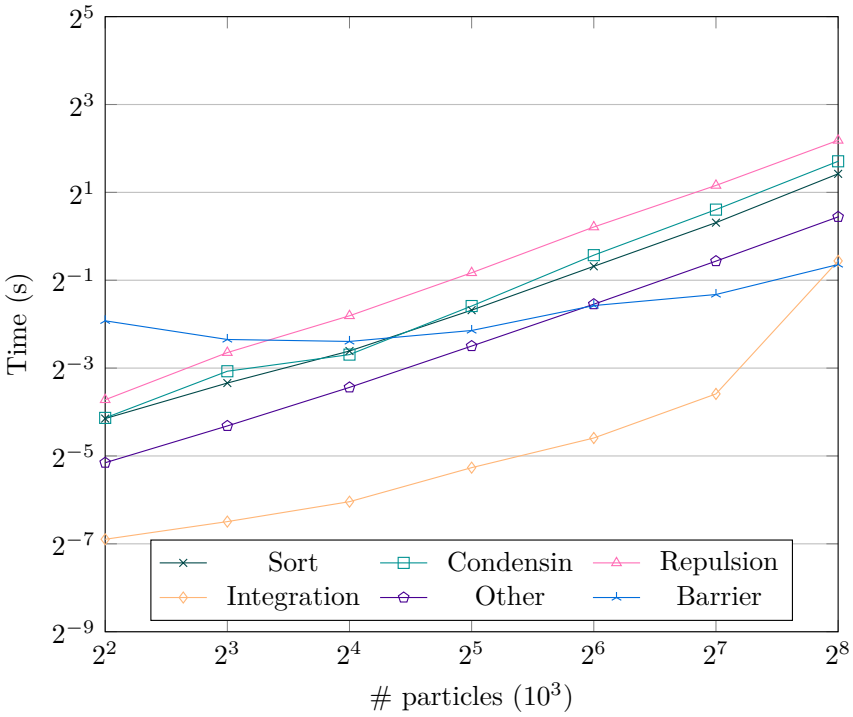
The slowdown to the entropic kernel is due to switching to a random number generator which is both thread-safe and provides a better source of randomness. We observe speedups in all other kernels. This decreases the time taken to perform a typical experimental run, consisting of 40 million timesteps, from ≈90 hours to ≈9 hours on our hardware. When factoring in the effects of parallelisation the

improvement is much greater.

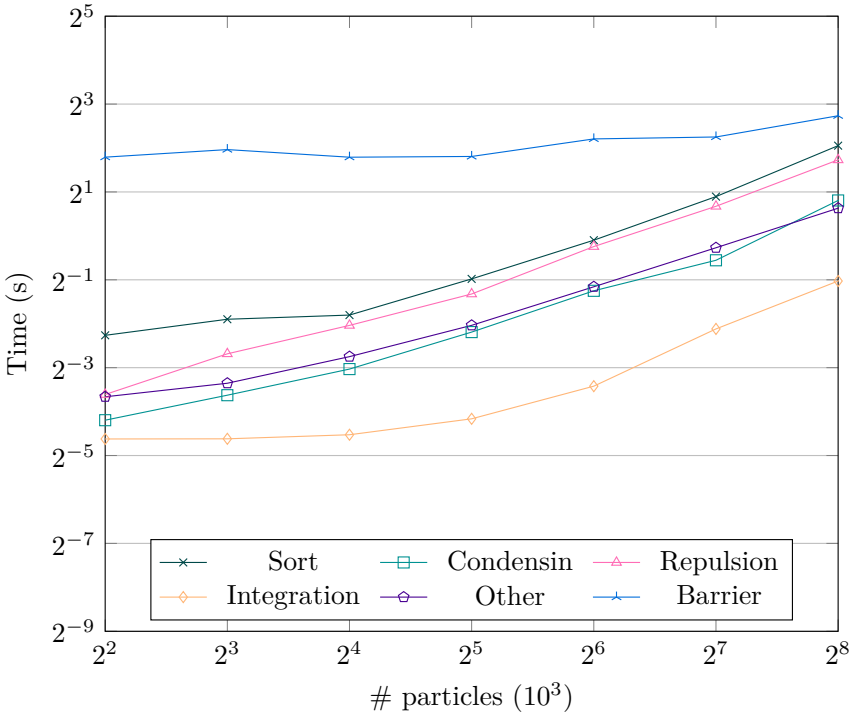
Figure 6.4 shows a breakdown of each optimised kernel's performance over a range of dataset sizes for both the CPU and coprocessor. On the CPU, the repulsion sweep is the most expensive, followed closely by the condensin interactions and the sort. The entropic, tension and attraction forces (grouped under "other") are comparatively cheap. The point where the integration falls out of Last Level Cache (LLC) is clearly visible between 128,000 and 256,000 particles. The barrier costs are fairly low throughout, but increase sharply for the largest dataset, possibly due to Non-Uniform Memory Access (NUMA) problems.

On the coprocessor, the sort is most expensive, primarily as sorting is difficult to vectorise effectively (see discussion in Section 6.2.2). Vectorisation is more crucial to performance on the Xeon Phi than on the CPU so this is expected. The repulsion sweep is cheaper on the Xeon Phi, as it vectorises very well and does not require any barriers. Interestingly, the condensin interactions are also cheaper to compute, despite not being vectorised either, likely because each binding site is largely independent leading to good scaling to a larger number of threads. The integration also scales better with dataset size, as the coprocessor has roughly 3× the memory bandwidth as the CPU (153 GB/s per NUMA region as opposed to 48 GB/s, as reported by the STREAM benchmark [59]).

The main issue we see on the coprocessor is significantly higher barrier costs. On some level this is unavoidable, a higher number of threads is going to mean slower blocking operations and a greater sensitivity to load imbalance, and we cannot remove any barriers as they are necessary to ensure correctness. We can aim to reduce the number of barriers via algorithmic changes however—the midpoint integration scheme used is the main culprit here, requiring twice as many barriers per timestep as would otherwise be needed.



(a)



(b)

Figure 6.4: Performance figures for each kernel when running on Tinis and Chiron, from 4000 up to 256,000 particles. (a) shows the timings for 16 threads on Tinis, (b) shows 244 threads running on Chiron. Lower time is better.

6.5.1 Offload Computation

We experimented with offloading computation to the coprocessor while running on the CPU. Suitable candidate kernels for offloading should perform well on the coprocessor, be able to run in parallel with other kernels (minimal data dependencies), not require large amounts of data transfer on and off the coprocessor each timestep, and take long enough that the overhead of offload does not dominate. Of the kernels in this simulation, the only one that satisfies most of these conditions is the projection sorting force sweep. It performs better on the coprocessor, and can be run in parallel with any of the other force computation kernels. Despite this, the time saved by running offloaded was roughly equalled by the overhead of doing so, and we did not see any significant change in performance.

6.6 Summary

We have explored the optimised implementation of projection sorting in a shared memory environment, for the Intel Haswell and KNC architectures. We show through detailed comparison of the number of distance checks, time spent on periodic calculations and time spent on force calculations that projection sorting outperforms Verlet lists for this simulation of chromosome condensation. We observe speedups of around 5× for both periodic calculations and force calculations across representative runs of the simulation. Together with our general optimisations from Chapter 5 we observe serial speedups up to 10× relative to the original implementation of this simulation.

Chapter 7

Implementing Projection Sorting Off-node

Up until this point, the simulation we have been using has only been parallelised using OpenMP [69], which limits experiments to a single node. As one of the primary ways of reducing the computation time of lengthy simulations is to run on multiple nodes in a cluster environment, it is crucial that we validate the projection sorting algorithm in this setting. Therefore, in this chapter we reimplement the simulation to leverage distributed memory parallelism using MPI in addition to the existing shared memory parallelism. In doing so we answer Research Question 3 (RQ3):

> **RQ3:** How might [projection sorting] perform (and how might it be best implemented) in a distributed-memory parallel environment?

In Chapter 6 we provided implementations for the Intel Haswell and Intel KNC architectures. Here we upgrade to their successors, Intel Broadwell and Intel KNL respectively. Intel KNL in particular is a significant re-engineering of KNC. Instead of running on a PCIe coprocessor KNL runs as a bootable device (in place of a CPU) which allows for faster access to main memory. KNL also provides on-package fast Multi-Channel DRAM (MCDRAM) memory, with up to 4× the memory bandwidth

of conventional Dynamic Random Access Memory (DRAM) memory [44]. This can be used in a variety of configurations, including caching accesses to main memory, or running applications entirely from it. Access to this memory can be controlled using NUMA software-awareness.

KNL provides a partial implementation of the AVX-512 SIMD ISA in place of the architecture specific KNC SIMD ISA. AVX-512 inherits many of the instructions from KNC (they are both designed around 512-bit vectors), but is designed to be used uniformly across KNL and newer CPU architectures including some Skylake chips. We provide new intrinsic implementations of projection sorting force sweep, Verlet list rebuilds and Verlet list force sweeps using this instruction set.

7.1 Experimental Setup

We use the same datasets previously generated (described in Chapter 6) throughout these experiments. As distributed memory parallelism brings with it the possibility of higher processor counts, we use some of the larger ones here, up to 2,048,000 particles.

In our on-node experiments, we enabled the condensin interaction forces to simulate a condensing chromatin string. We found that when doing so, the minimum valid skin distance was $r_s = 40\,\text{nm}$, and noted that the fast moving particles were a reason for Verlet lists being less effective (see Section 6.3). To provide an alternate perspective, we disable the condensin interaction forces for experiments in this chapter to simulate a relaxed chromatin string. This allows us to set the skin distance lower, $r_s = 15\,\text{nm}$, and consequently increase the rebuild period to $k = 5$.

7.1.1 Machine Specifications, Compilation & Execution

To run the experiments, two machines were used. Firstly, Orac, a cluster at the University of Warwick consisting of 84 nodes, each with 2 Intel Xeon E5-2680 v4

	Xeon E5-2680 v4	Xeon Phi 7210
Sockets×Cores×Threads	2×14×2	1×64×4
Clock (GHz)	2.40	1.30
L1{i,d} / L2 cache (KB)	{32,32} / 256	{32,32} / 512
Memory (GB)	128	96+16
SIMD ISA	AVX2	AVX-512

Table 7.1: Summary of node hardware configurations on Orac and the ARCHER Knights Landing platform.

processors, for 2352 cores in total. Each node is equipped with 128 GB of Random Access Memory (RAM), and connected with an Intel Omni-Path X16 100 Gbit/s interconnect. Orac was used to run the Xeon experiments.

To run the Xeon Phi experiments we used the ARCHER Knights Landing platform at the Edinburgh Parallel Computing Centre. This consists of 12 nodes, each equipped with a Xeon Phi 7210 and 96 GB of RAM. The nodes are connected with a high performance Cray Aries interconnect. Each Xeon Phi also has 16 GB of fast onboard MCDRAM, which in this case is used to cache accesses to the external memory. We refer the reader to Table 7.1 for more details.

All code was compiled using the GNU C++ compiler, v6.3.0[1] with the following performance flags: `-O3 -fopenmp`. Architecture was specified with `-march=broadwell`, and `-march=knl` for Xeon and Xeon Phi respectively.

On Orac, the application was launched using the `mpirun` command. Ranks were distributed evenly among the available resources using the following flags: `--map-by socket:SPAN --bind-to core --rank-by core`. Each rank was assigned to a unique core, and used a single thread.

On ARCHER, the application was launched using the `aprun` command. Each KNL node was assigned 64 ranks, each bound to a unique core. Each rank used two threads out of four available hardware threads per core, which we found to yield

[1]Newer versions of the Intel compiler encounter an internal error when compiling our lock-free hash map implementation, which is why we do not use it here as in Chapter 6.

the best performance. This was achieved using the following flags: `-d 2 -j 2 -cc depth`.

All experiments have a duration of 1000 timesteps. When using Verlet lists, we set the skin distance to the empirical minimum $r_s = 15\,\text{nm}$, and $k = 5$, which yields the best performance for the given value of r_s. Timing information was collected across all areas of the code as discussed in Chapter 6.

7.2 MPI Implementation

To reduce the engineering burden of porting the simulation to MPI, we based the new implementation on the Mantevo suite miniMD mini-app from Sandia National Laboratories (SNL) [38, 39], which is known to exhibit good scaling. We removed the application specific code, keeping parts related to particle storage and communication between MPI ranks. miniMD uses a spatial decomposition which theoretically scales well with processor count, at the price of more complex load balancing in the case of simulations with non-uniform particle densities (like this one).

To address this, we implemented a static load balancing strategy wherein the problem domain is decomposed into small cubic "patches" of uniform size. Note that this patch-decomposition is only used to determine which region of space each MPI rank is responsible for, and is distinct from the cell list decomposition (see Chapter 3). Each patch is then weighted according to how many particles it contains, and patches are distributed among processors by a recursive bisection algorithm, such that all processors end up with a cuboidal subdomain of approximately equal total weight. This strategy is used by NAMD [66] and is based on work from Berger and Bokhari [11].

NAMD, which has similar load balancing requirements, implements a layer of dynamic load balancing on top of this, which distributes pairwise force calculations between neighbouring processors. Our implementation does not currently do this,

although it is a possible extension. The static load balancing performs well enough to show the differences between Verlet lists and projection sorting.

The application is parallelised in a so-called "hybrid" fashion. Within each MPI rank, OpenMP may be used to leverage multiple threads. We use this capability to take advantage of SMT, or hyperthreading, on KNL.

Our implementation has two primary compile-time switches: use of projection sorting or Verlet lists, and enabling or disabling halving of the neighbour space using Newton's third law. We refer to these four variants below as *PS N3*, *PS no N3*, *VL N3* and *VL no N3*.

7.2.1 Projection Sorting

The low level implementation of projection sorting is largely consistent with that explained in Chapter 6. After MPI ranks have communicated with their neighbours to receive ghost particles, they include these when computing projections and sorting. Force computation using the sorted conformation was hand-vectorised using both AVX2 and AVX-512 intrinsics. Appendix A gives full listings for these implementations.

When using N3 it is now important that different ranks are consistent when deciding which particles to directly calculate forces for, and which to implicitly calculate as a result of the equal and opposite action of the explicit force. For projection sorting this is straightforward. As long as all ranks use the same axis of projection $\vec{v}$, the scalar projections will form a total ordering over all ranks. Particles with greater relative scalar projections are calculated explicitly, and those with lower relative scalar projections are calculated implicitly (or vice versa).

7.3 Experiments

In this section we seek to empirically establish the performance differences between our implementations of Verlet lists and projection sorting. In order to do this, we present the following comparisons:

- Raw performance comparisons between the two algorithms for fixed numbers of processors and problem sizes (see Section 7.3.1).
- A scaling study for each implementation, showing how the performance changes with varying numbers of processors and different problem sizes (see Section 7.3.2).
- Qualitative performance comparisons between the Xeon and Xeon Phi platforms (see Section 7.3.3).

7.3.1 Raw Performance Comparisons

Figure 7.1 shows the speedup attained by projection sorting relative to the performance of Verlet lists, for the entire simulation (including kernels unrelated to the repulsion calculations). This is significant for all four configurations, reaching nearly $5\times$ on Orac.

We observe that PS attains a greater relative speedup when N3 is disabled, although enabling N3 results in better absolute performance. On ARCHER, PS sees approximately $2\times$ speedup. We conjecture that this lower speedup is due to better hardware support for gather/scatter and vector compression on KNL, which VL relies on.

Figures 7.2 and 7.3 break these speedups down into periodic calculations, force calculations and communication costs. Notable are the communication costs, which are higher for PS. This is because VL only needs to communicate fully every $k = 5$ timesteps, which fits with the approximate $0.2\times$ speedup we see. We note also that the speedup improves with the number of processors. We conjecture that this is

due to the lower amount of communication required per processor in PS (discussed in Section 4.4.3). However, the higher communication costs are more than offset by the significantly lower periodic costs. On ARCHER these reach speedups of over 10×.

7.3.2 Scaling Studies

Figures 7.4 and 7.5 show strong and weak scaling figures on Orac and ARCHER. Both algorithms scale well, with VL reaching over 150× for 256 processors on Orac. VL generally scales slightly better than PS, likely due to the lower communication costs. We expect that this gap could be closed if PS was adapted to use a skin distance (see the further work in Chapter 8).

Figure 7.5 shows some anomalous super-linear scaling for PS, likely due to the lower amount of data per processor improving cache effectiveness.

7.3.3 Xeon vs. Xeon Phi

It is inherently difficult to compare performance between Xeon and Xeon Phi architectures, as they differ in almost every important aspect. Equal numbers of cores are not comparable, as KNL cores are much slower individually. In Chapter 6 we found that KNC was well suited to the force calculations as it has double the vector width available.

Figure 7.3b shows that disabling N3 nets a greater relative increase to force calculation speedup than on Orac (cf. Figure 7.2b). This is because KNL crucially uses SMT to keep its execution units busy, and multiple threads with N3 enabled necessitates atomic accesses to shared memory to prevent conflicts. KNL has poor support for vectorised atomic writes. On Orac SMT is not used, therefore the atomics are not necessary. This effect is also visible in the VL rebuilds, which use atomics as part of a lock-free hash table (compare Figures 7.3a and 7.2a).

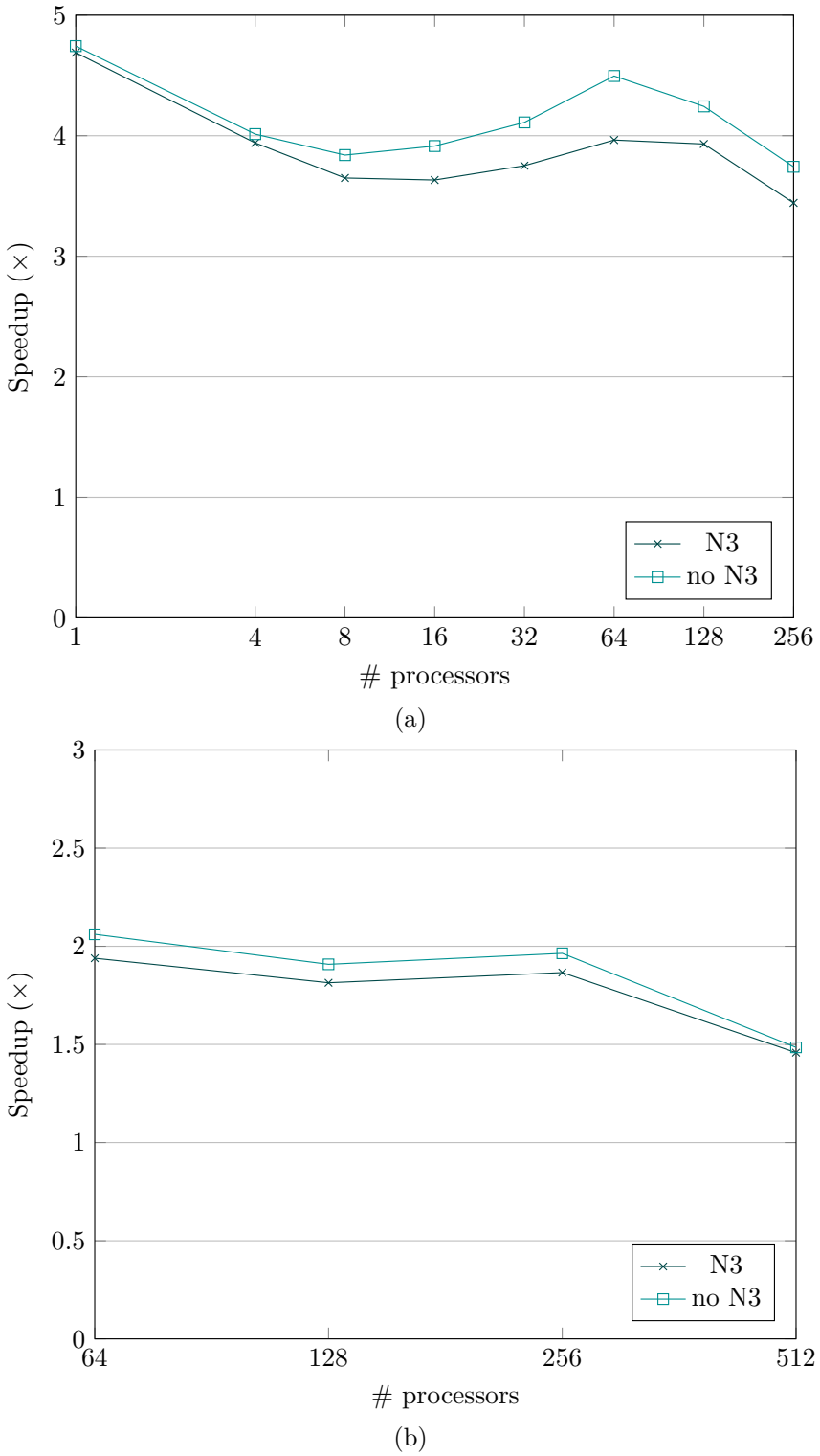
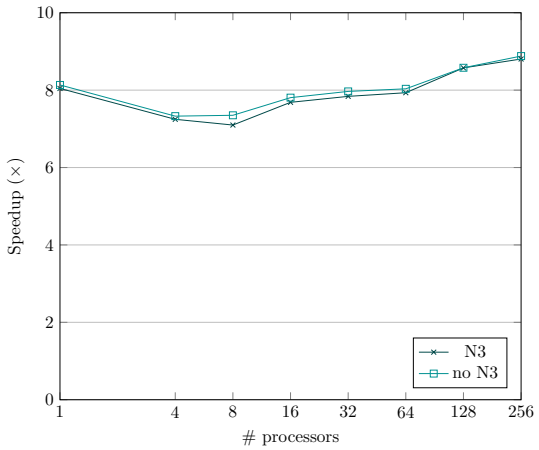
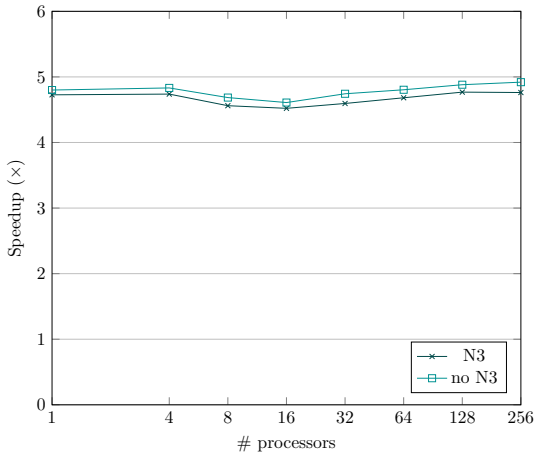


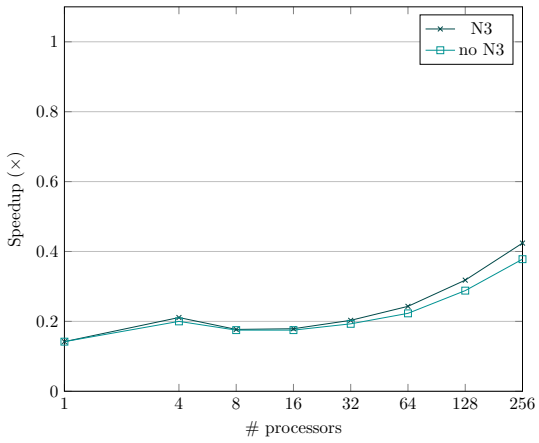
Figure 7.1: Speedup of the full simulation when using PS relative to VL on (a) Orac up to 256 processors and (b) ARCHER up to 512 processors, over 2,048,000 particles. Higher speedup is better.



(a)

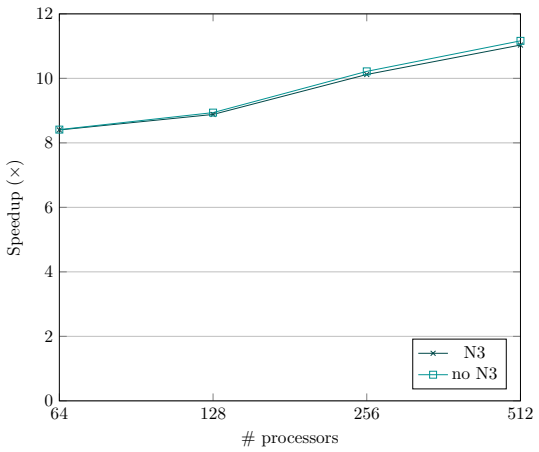


(b)

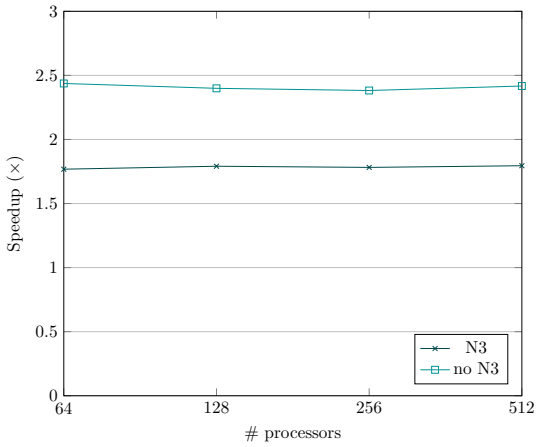


(c)

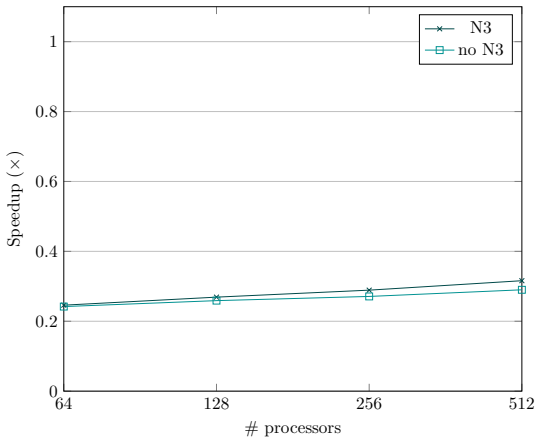
Figure 7.2: Speedup for PS kernel costs relative to VL kernel costs on Orac up to 256 processors, over 2,048,000 particles. (a) shows periodic costs, (b) shows force calculation costs, and (c) shows communication costs. Higher speedup is better.



(a)

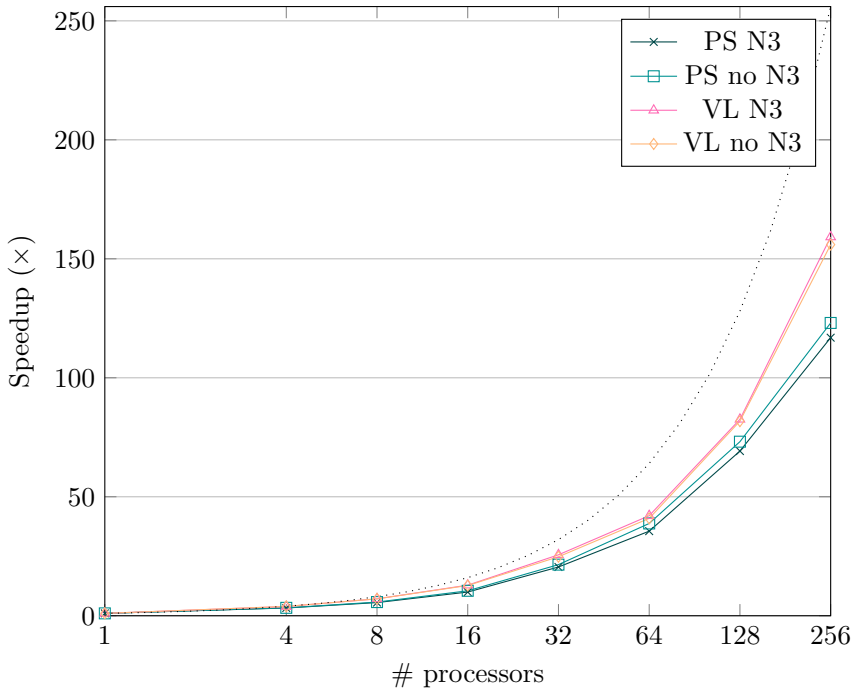


(b)

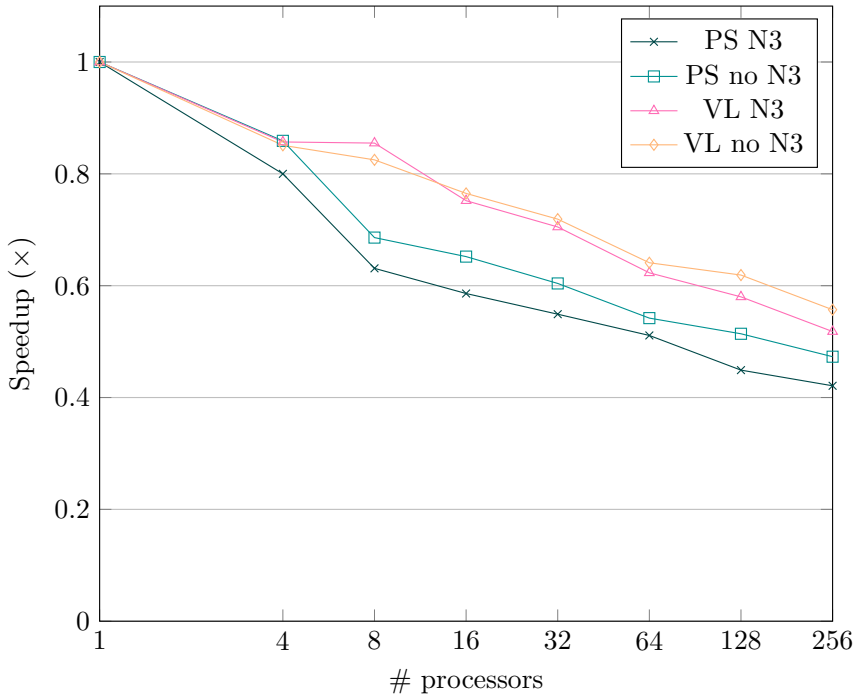


(c)

Figure 7.3: Speedup for PS kernel costs relative to VL kernel costs on ARCHER up to 512 processors, over 2,048,000 particles. (a) shows periodic costs, (b) shows force calculation costs, and (c) shows communication costs. Higher speedup is better.

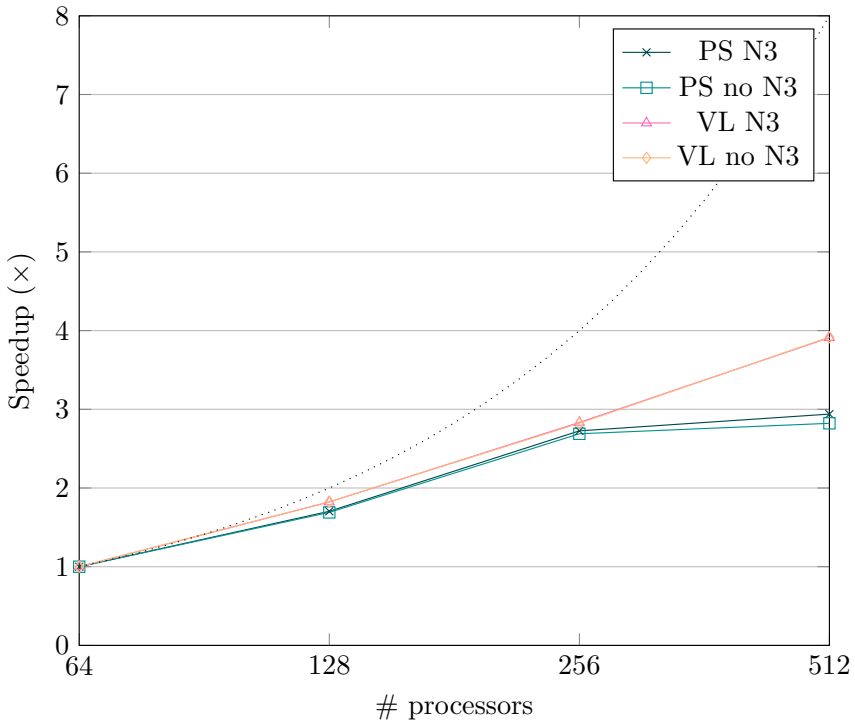


(a)

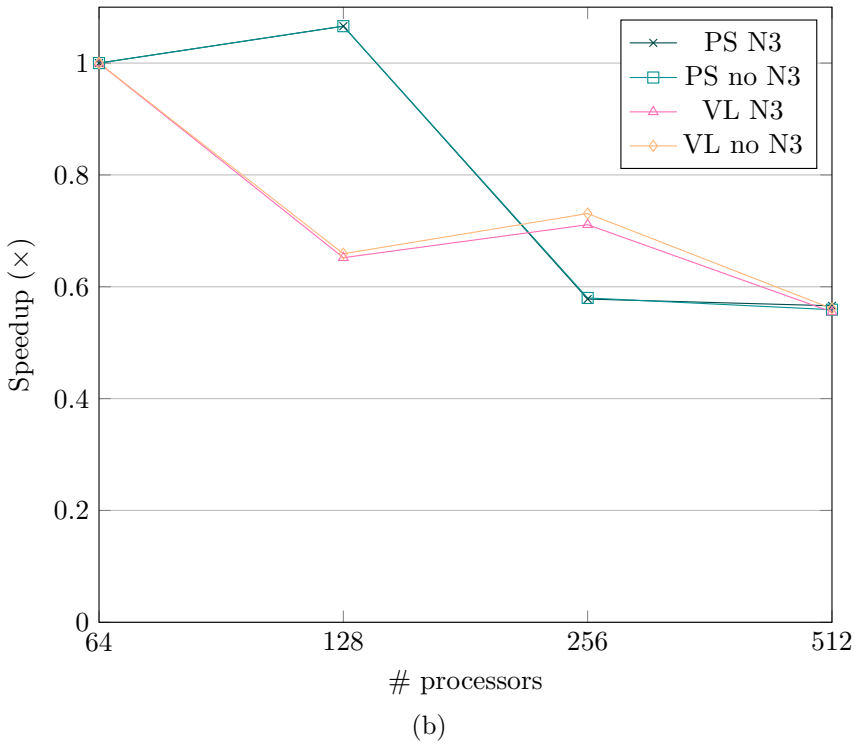


(b)

Figure 7.4: Scaling on Orac up to 256 processors. (a) shows strong scaling over 2,048,000 particles and (b) shows weak scaling with 8000 particles per processors. Higher speedup is better.



(a)



(b)

Figure 7.5: Scaling on ARCHER up to 512 processors. (a) shows strong scaling over 2,048,000 particles and (b) shows weak scaling with 4000 particles per processors. Higher speedup is better.

7.4 Summary

We extend our shared memory implementation of projection sorting to hybrid distributed/shared memory using MPI, and reimplement key kernels for the Intel Broadwell and KNL architectures. We also relax the skin distance r_s by changing the context of the simulation, which benefits Verlet lists. Similar to our conclusions from Chapter 6, we observe significant speedups due to projection sorting up to $5\times$. We scale the simulation to previously infeasible dataset sizes by running on large numbers of cores on clusters of Xeon and Xeon Phi nodes.

Chapter 8

Conclusions

The ability to perform simulations on a large scale is crucial to the continued progress of science across a range of domains. Such simulations require great computational resources, and scientists working in the field of HPC seek to reduce their expense through innovations in hardware, software and underlying mathematical principles.

In this thesis we investigated Molecular Dynamics (MD) simulations, and in particular an MD simulation of chromosome condensation. We presented *projection sorting*, an alternative to the traditional Verlet list algorithm for pairwise short-range force calculations, and showed that it can be significantly more effective under certain conditions.

In Chapters 6 and 7 we explored the efficient implementation of this strategy for the modern multi- and many-core architectures: Intel Haswell, Broadwell, Knights Corner (KNC) and Knights Landing (KNL). We covered optimisations across the parallel stack, from SIMD vectorisation up to communication strategies in a message passing implementation for clusters. Our implementations are parallelised using shared memory (OpenMP), distributed memory (MPI) and a hybrid of both techniques, representative of MD simulations used across a large cross-section of modern science.

We demonstrated speedups relative to Verlet lists of up to 5× across a range

of problem sizes and processor counts. Our algorithm scales comparably to the state of the art in both strong and weak senses, with the runtime being reduced by up to 125× over 256 processors.

These results are not just theoretical: our algorithm and optimisations have been, and continue to be used by domain scientists to facilitate further experiments in chromosome condensation using the simulation described in Chapter 5. As a result of our optimisation work, we see serial speedups of up to 10× relative to the original implementation of this simulation.

8.1 Contributions

In Chapter 1 we posed three Research Questions (RQ1 − 3) which were tackled throughout the course of the thesis. We restate these questions below:

> **Conjecture**: It is possible to accelerate pairwise short-range force calculations relative to classic MD methods by taking closer account of the domain-specific properties of the simulation (such as non-uniform particle density).

- **RQ1:** How might an algorithm designed to accelerate an MD simulation in accordance with the above conjecture look?
- **RQ2:** How might such an algorithm perform (and how might it be best implemented) in a shared-memory parallel environment?
- **RQ3:** How might such an algorithm perform (and how might it be best implemented) in a distributed-memory parallel environment?

In Chapter 4 we proposed the projection sorting algorithm as a possible answer to RQ1. This algorithm produces a 1-dimensional ordering over the set of particles which creates a linear search space for the neighbours of any given particle, and also

attempts to minimise the size of this search space. The search space is efficiently traversed at each timestep in order to calculate forces. This is in contrast to Verlet lists, where an explicit list of candidate neighbours is built and stored on a per-particle basis, and reused over multiple timesteps. Projection sorting's approach of linearising the search space allows for more efficient traversal than is possible with Verlet lists, and the manner of linearisation is computationally cheaper than the construction of Verlet lists. However projection sorting is not efficient when particles are spread uniformly throughout the simulation space, as in this case it requires more pair-checks.

In Chapter 6 we sought to answer RQ2 by implementing and optimising projection sorting in a shared-memory parallel environment, and comparing it against an optimised implementation of Verlet lists in the same setting. We explored the efficient implementation of the two primary components of the projection sorting algorithm: the sort and the force sweep, on the Intel Haswell and KNC architectures. We found that a thread-local quicksort followed by a multi-threaded merge yielded the best performance when sorting, and observed minor speedups due to vectorisation. The force sweep offers excellent cache efficiency and high performance in practice, vectorisation in particular is very effective here.

Finally, in Chapter 7 we sought to answer RQ3 by implementing and optimising projection sorting in a distributed-memory parallel environment, and comparing it against an optimised implementation of Verlet lists in the same setting. Here we explored the consequences of distributed-memory on the implementation of the algorithm, and investigated its scaling to large numbers of cores. We found that projection sorting scales comparably to Verlet lists, indicating its suitability for large scale simulations.

Our results build in particular on those from Gonnet [32] and Pennycook et al. [70] (discussed in Chapter 3). We extend Gonnet's ideas regarding sorting particles along a vector beyond simply searching cell lists. Pennycook et al. studied in detail how

best to vectorise short-range pairwise force calculations, with a particular focus on the Xeon Phi architectures—we build on this work to determine how best to vectorise projection sorting.

8.2 Limitations

The primary limitation of this thesis is its focus upon a single scientific application as a means of establishing the effectiveness of projection sorting as an alternative to Verlet lists. We discuss in Chapter 3 how pairwise short-range force calculations are almost ubiquitous across MD, both in their application and in their relative computational expense. Nearly all MD codes use a computational shell identical in structure to the one present in the simulation from Cheng et al. [20], where for each particle, the code needs to search through all other particles in some way to find its near neighbours, and then some force is applied between the pair as a function of distance. Our research is completely application agnostic in the sense that any such calculation could make use of projection sorting, which serves to accelerate the search rather than the force calculation itself (which does vary from application to application). However, we would certainly expect the *effectiveness* of projection sorting to vary significantly based on the domain-specific characteristics of each simulation—from the outset we have emphasised that the algorithm could only be expected to provide improvements in cases where non-uniform particle density is exhibited. Further work on this algorithm would certainly involve implementation in other MD packages. Implementation of our ideas in a larger scale package such as NAMD [71] (which is commonly used for biological simulations exhibiting non-uniform particle densities) would enable experimentation with a wider variety of different simulation domains.

We also note that our algorithm sees slightly poorer scaling to large problem sizes than Verlet lists (refer to Figures 7.4 and 7.5), despite being significantly faster

overall. We have two ideas to pursue in regard, which we discuss in Section 8.3.

8.3 Further Work

In closing, we discuss several avenues for future exploration of the projection sorting algorithm and its applicability to the optimisation of short-range pairwise force calculations. We discuss three main areas: improvements to the algorithm itself, and alternate ways of making use of it; implementations using different hardware or parallel frameworks; and implementation of the algorithm in other MD packages.

8.3.1 Improvements to Projection Sorting

Using a Skin Distance

As has been mentioned many times (see Chapter 3), Verlet lists make use of a skin distance r_s, which quantifies the maximum distance particles may move in any given given timestep. Incorporating this allows reuse of one set of lists over k timesteps. It is conceivable to use the same strategy with projection sorting. The idea is to only sort the particles every k timesteps, and to continue up to the "Verlet" cut-off radius, $r_v = r_c + 2kr_s$, when performing force sweeps. As the sort is considerably cheaper than a Verlet rebuild, it is possible that the additional costs associated with the longer force sweeps outweigh the benefits, but we note that it would also improve the communication costs, which may balance this increase at scale.

Alternate MPI Decomposition Strategies

We are also investigating alternate spatial decomposition strategies to offset projection sorting's main disadvantage: that it must inspect more particles in directions orthogonal to the projection axis. As we use a spatial MPI decomposition, part of the work of finding neighbours in short-range pairwise force calculations is already done in a sense by the comms layer. Each MPI rank only needs to search within its

own local particles and a surrounding layer of ghost particles.

We hypothesise that a "french fry" decomposition—decomposing space into long thin slivers along the axis of projection rather than into more regular cuboids as is typically done—could significantly decrease the number particles to be processed.

Typically the MPI decomposition is tuned to minimise the surface area of the adjoining faces between partitions. This minimises the amount of data that needs to be transferred between processes, and therefore is optimal in terms of the necessary network bandwidth. The "french fry" approach would increase this surface area, but decrease the number of neighbouring partitions, which would improve the comms performance in a high latency environment.

Using Projection Sorting as a Verlet List Building Method

One of the most expensive parts of the Verlet list algorithm is actually building the Verlet lists (using cell lists), performed every k timesteps. It is possible to use projection sorting as a means of constructing these lists in place of cell lists. The algorithm would proceed as usual, except rather than calculating forces during the force sweep, one would instead populate the Verlet lists. This hybrid method may potentially provide speedups in scenarios with long Verlet lists where cell lists are a bottleneck.

8.3.2 Alternative Shared-Memory Parallelisations

In addition to what has been studied in this thesis, it would be interesting to analyse the performance of projection sorting on other architectures, in particular GPUs. GPUs have traditionally been highly effective tools when it comes to scientific simulation in general, and MD on GPU is well-studied (e.g. [9, 72, 16, 54, 31] amongst many others). By their nature, GPUs are more effective at rapidly executing many independent iterations of small non-branching numerical computational kernels than traditional fat CPU cores, which are designed to be flexible and provide good per-

formance for many kinds of workload.

The first-class way to target NVIDIA GPUs is the Compute Unified Device Architecture (CUDA) toolkit, which is now a mature piece of software, and makes the addition of GPU support to existing code relatively easy. Other ways to target NVIDIA GPUs exist, such as the OpenACC [68] (short for accelerator) framework, which is directive-driven in the same manner as OpenMP, but offers better support for device offload.

A more recent development are so-called performance portability frameworks, which allow developers to write a single version of an application which can be compiled to run against a variety of different parallel back-ends with minimal modification. Two examples of these are Kokkos [24] and RAJA [42]: both use advanced C++ metaprogramming constructs to enable low overhead implementation of this portability idea. Implementation of the simulation of chromosome condensation using either of these would be an effective way to try different shared-memory parallelisations.

8.3.3 Implementation in Other Molecular Dynamics Packages

As mentioned above, implementation of projection sorting in other existing MD software would be valuable to establish the extent of its applicability in a more general sense. NAMD is a popular production MD package designed for simulation of large biomolecular structures which has been the subject of substantial HPC research, and has been shown to scale to over 500,000 cores.

NAMD is implemented using Charm++, a parallel programming system from the University of Illinois Parallel Programming Laboratory [47]. Charm++ applications are written in primarily in C++, with an interface description language to specify Charm++ parallel objects. Execution is separated into "chares": lightweight process-like objects which communicate in a message-driven fashion. Charm++ supports modular applications, executing across heterogeneous architectures using

a variety of load balancing strategies.

NAMD is a large code, and implementing a new algorithm would be a time consuming process, however we believe this would be a worthwhile endeavour, and is crucial to gaining further insight into the properties of the projection sorting algorithm, and directions in which to take it.

Bibliography

[1] Linux perf. URL `https://perf.wiki.kernel.org/index.php/Main_Page`.

[2] MPI: a message-passing interface standard. Technical report, 1994.

[3] IEEE standard for floating-point arithmetic. IEEE, August 2008.

[4] F Abraham. Computational statistical mechanics methodology, applications and supercomputing. *Advances in Physics*, 35(1):1–111, January 1986.

[5] M J Abraham, T Murtola, R Schulz, S Páll, J C Smith, B Hess, and E Lindahl. GROMACS: high performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2:19–25, September 2015.

[6] B J Alder and T E Wainwright. Studies in molecular dynamics. I. general method. *Journal of Chemical Physics*, 31(2):459–466, 1959.

[7] R J Allan. Survey of HPC performance modelling and prediction tools. *Daresbury Laboratory Technical Reports*, 2009.

[8] R Allen and K Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[9] J A Anderson, C D Lorenz, and A Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, May 2008.

[10] K E Batcher. Sorting networks and their applications. In *Proceedings of the 1968 AFIPS Conference*, pages 307–314. ACM Press, 1968.

[11] M Berger and S Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, 1987.

[12] A Bhatele, S Kumar, C Mei, J C Phillips, G Zheng, and L V Kalé. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 1988*, pages 1–12. IEEE, 2008.

[13] R Blumofe, C Joerg, B Kuszmaul, C Leiserson, K Randall, and Y Zhou. Cilk. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 30 (8):207–216, August 1995.

[14] B Boghosian. Computational physics on the connection machine. *Computers in Physics*, 4(1):14, 1990.

[15] C Brooks. Computer simulation of liquids. *Journal of Solution Chemistry*, 18 (1):99–99, January 1989.

[16] W M Brown, P Wang, S J Plimpton, and A N Tharrington. Implementing molecular dynamics on hybrid high performance computers – short range forces. *Computer Physics Communications*, 182(4):898–911, April 2011.

[17] S Browne, C Deane, G Ho, and P Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999.

[18] R A Bunt, S A Wright, S A Jarvis, Y K Ho, and M J Street. Predictive evaluation of partitioning algorithms through runtime modelling. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 351–361. IEEE, 2016.

[19] J Cao, D J Kerbyson, E Papaefstathiou, and G R Nudd. Performance modeling of parallel and distributed computing using PACE. In *2000 International Conference on Performance, Computing and Communications (IPCCC 2000)*, pages 485–492. IEEE, 2000.

[20] T M K Cheng, S Heeger, R A G Chaleil, N Matthews, A Stewart, J Wright, C Lim, P A Bates, and F Uhlmann. A simple biophysical model emulates budding yeast chromosome condensation. *eLife*, 4:e05565, 2015.

[21] J Chhugani, A D Nguyen, V W Lee, W Macy, M Hagog, Y Chen, A Baransi, S Kumar, and P Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *Proceedings of the VLDB Endowment 2008*, pages 1313–1324. VLDB Endowment, August 2008.

[22] J A Davis, G R Mudalige, S D Hammond, J A Herdman, I Miller, and S A Jarvis. Predictive analysis of a hydrodynamics application on large-scale CMP clusters. *Computer Science - Research and Development*, 26(3-4):175–185, April 2011.

[23] J Dongarra. The LINPACK benchmark: an explanation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Springer Berlin Heidelberg, 1988.

[24] H C Edwards, C R Trott, and D Sunderland. Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, December 2014.

[25] R Everaers and K Kremer. A fast grid search algorithm for molecular dynamics simulations with short-range interactions. *Computer Physics Communications*, 81(1-2):19–55, June 1994.

[26] J Fenlason. GNU gprof. URL `https://sourceware.org/binutils/docs/gprof/`.

[27] E Fermi, J Pasta, S Ulam, and M Tsingou. Studies of non-linear problems. Technical report, May 1955.

[28] M Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[29] G Fox, M Johnson, G Lyzenga, S Otto, J Salmon, D Walker, and Richard L White. Solving problems on concurrent processors vol. 1: general techniques and regular problems. *Computers in Physics*, 3(1):83, 1989.

[30] R S Francis and I D Mathieson. A benchmark parallel sort for shared memory multiprocessors. *IEEE Transactions on Computers*, 37(12):1619–1626, 1988.

[31] J Glaser, T D Nguyen, J A Anderson, P Lui, F Spiga, J A Millan, D C Morse, and S C Glotzer. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Computer Physics Communications*, 192:97–107, July 2015.

[32] P Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *Journal of Computational Chemistry*, 28(2):570–573, January 2007.

[33] S D Hammond, G R Mudalige, J A Smith, S A Jarvis, J A Herdman, and A Vadgama. *WARPP: a toolkit for simulating high-performance parallel scientific codes*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), March 2009.

[34] S D Hammond, J A Smith, G R Mudalige, and S A Jarvis. *Predictive simulation of HPC applications*. IEEE, May 2009.

[35] A Harode, A Gupta, B Mathew, and N Rai. Optimization of molecular dynamics application for Intel Xeon Phi coprocessor. In *Proceedings of the International Conference on High Performance Computing and Applications 2014*, pages 1–6. IEEE, 2014.

[36] M Hazewinkel. *Runge-Kutta Method*, volume 227-228. Encyclopedia of Mathematics, January 2001.

[37] T N Heinz and P H Hünenberger. A fast pairlist-construction algorithm for molecular simulations under periodic boundary conditions. *Journal of Computational Chemistry*, 25(12):1474–1486, September 2004.

[38] M A Heroux and R Barrett. Mantevo project. May 2011.

[39] M A Heroux, D W Doerfler, and P S Crozier. Improving performance via mini-applications. Technical report, 2009.

[40] R W Hockney, S P Goel, and J W Eastwood. Quiet high-resolution computer models of a plasma. *Journal of Computational Physics*, 14(2):148–158, February 1974.

[41] A Hoisie, G Johnson, D Kerbyson, M Lang, and S Pakin. A performance comparison through benchmarking and modeling of three leading supercomputers: Blue Gene/L, Red Storm, and Purple. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2006*, pages 3–3. IEEE, 2006.

[42] R D Hornung and J A Keasler. The RAJA portability layer: overview and status. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), September 2014.

[43] Intel Corporation. Intel VTune Amplifier XE. URL `https://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[44] Intel Corporation. An introduction to MCDRAM (high bandwidth memory) on Knights Landing, June 2016. URL `https://software.intel.com/en-us/blogs/2016/01/20/an-intro-to-mcdram-high-bandwidth-memory-on-knights-landing`.

[45] W Jiang, J C Phillips, L Huang, M Fajer, Y Meng, J C Gumbart, Y Luo, K Schulten, and B Roux. Generalized scalable multiple copy algorithms for molecular dynamics simulations in NAMD. *Computer Physics Communications*, 185(3):908–916, March 2014.

[46] I T Jolliffe and J Cadima. Principal component analysis: a review and recent developments. *Phil. Trans. R. Soc. A*, 374(2065):20150202, April 2016.

[47] L V Kalé and S Krishnan. CHARM++: a portable concurrent object oriented system based on C++. Technical report, October 1993.

[48] D J Kerbyson, H J Alme, A Hoisie, F Petrini, H J Wasserman, and M Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2001*, pages 37–37, New York, New York, USA, 2001. ACM Press.

[49] D J Kerbyson, A Hoisie, and H J Wasserman. Modelling the performance of large-scale systems. *IEE Proceedings - Software*, 150(4):214–221, August 2003.

[50] D J Kerbyson, A Hoisie, and H J Wasserman. A comparison between the Earth Simulator and AlphaServer systems using predictive application performance models. In *Proceedings of the International Symposium on Parallel and Distributed Processing 2003*, page 10. IEEE Comput. Soc, 2003.

[51] S Kumar, D Kim, M Smelyanskiy, Y Chen, J Chhugani, C J Hughes, C Kim, V W Lee, and A D Nguyen. Atomic vector operations on chip multiprocessors. In *Proceedings of the 35th International Symposium on Computer Architecture 2008*, pages 441–452. IEEE, 2008.

[52] J Law and R Rennie. *A dictionary of physics*. Oxford University Press, USA, April 2015.

[53] T R Law, J Hancox, T M K Cheng, R A G Chaleil, S A Wright, P A Bates, and S A Jarvis. Optimisation of a molecular dynamics simulation of chromosome condensation. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 126–133. IEEE, 2016.

[54] S Le Grand, A W Götz, and R C Walker. SPFP: Speed without compromise—A mixed precision model for GPU accelerated molecular dynamics simulations. *Computer Physics Communications*, 184(2):374–380, February 2013.

[55] C Luk, R Cohn, R Muth, H Patil, A Klauser, G Lowney, S Wallace, V Reddi, and K Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, June 2005.

[56] A C Mallinson, D A Beckingsale, and W P Gaudin. Cloverleaf: preparing hydrodynamics codes for exascale. *The Cray User Group*, 2013.

[57] G Marcais and C Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, March 2011.

[58] W Mattson and B M Rice. Near-neighbor calculations using a modified cell-linked list method. *Computer Physics Communications*, 119(2-3):135–148, June 1999.

[59] J D McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, 1995.

[60] J Meng, V Morozov, K Kumaran, V Vishwanath, and T Uram. GROPHECY. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2011*, page 1, New York, New York, USA, 2011. ACM Press.

[61] H Meuer, E Strohmaier, J Dongarra, H Simon, and M Meuer. Top500, June 2017. URL `https://www.top500.org`.

[62] D C Montgomery, E A Peck, and G G Vining. *Introduction to linear regression analysis*. 5 edition, April 2012.

[63] B Moon, H V Jagadish, C Faloutsos, and J H Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[64] G R Mudalig, S D Hammond, J A Smith, and S A Jarvis. Predictive analysis and optimisation of pipelined wavefront computations. In *Proceedings of the International Symposium on Parallel and Distributed Processing 2009*, pages 1–8. IEEE, 2009.

[65] R Murphy, K Underwood, A Rodrigues, and P Kogge. The Structural Simulation Toolkit: a tool for exploring parallel architectures and applications. Technical report, 2007.

[66] M Nelson, W Humphrey, A Gursoy, A Dalke, L V Kalé, R D Skeel, and K Schulten. NAMD: a parallel, object-oriented molecular dynamics program. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(4):251–268, September 1996.

[67] A Nicolau. Loop quantization: unwinding for fine-grain parallelism exploitation. Technical report, Cornell University, October 1985.

[68] OpenACC Standard Board. OpenACC 2.6 specification, November 2017.

[69] OpenMP Architecture Review Board. OpenMP 4.5 specification, November 2015.

[70] S J Pennycook, C J Hughes, M Smelyanskiy, and S A Jarvis. Exploring SIMD for molecular dynamics, using Intel® Xeon® processors and Intel® Xeon Phi™

coprocessors. In *Proceedings of the International Symposium on Parallel and Distributed Processing 2013*, pages 1085–1097. IEEE Computer Society, May 2013.

[71] J C Phillips, R Braun, W Wang, J Gumbart, E Tajkhorshid, E Villa, C Chipot, R D Skeel, L Kalé, and K Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, December 2005.

[72] J C Phillips, J E Stone, and K Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2008*, pages 1–9. IEEE, 2008.

[73] S J Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117:1–19, March 1995.

[74] B Quentrec and C Brot. New method for searching for neighbors in molecular dynamics computations. *Journal of Computational Physics*, 13(3):430–432, November 1973.

[75] D Rapaport. Large-scale molecular dynamics simulation using vector and parallel computers. *Computer Physics Reports*, 9:1–53, 1988.

[76] R Reussner, P Sanders, and J Träff. SKaMPI: a comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[77] V Rokhlin. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics*, 60(2):187–207, September 1985.

[78] S Satish, C Kim, J Chhugani, A D Nguyen, V W Lee, D Kim, and P Dubey. *Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort.* a case for bandwidth oblivious SIMD sort. ACM, New York, New York, USA, June 2010.

[79] S Shende and A Malony. The Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, September 2016.

[80] Y Sun, G Zheng, C Mei, E J Bohm, J C Phillips, L V Kalé, and Terry R Jones. Optimizing fine-grained communication in a biomolecular simulation application on Cray XK6. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2012*, pages 1–11. IEEE, 2012.

[81] P Terdiman. Radix sort revisited, April 2000. URL `http://codercorner.com/RadixSortRevisited.htm`.

[82] I T Todorov, W Smith, K Trachenko, and M T Dove. DL_POLY_3: new dimensions in molecular dynamics simulations via massive parallelism. *Journal of Materials Chemistry*, 16(20):1911–1918, May 2006.

[83] D M Tullsen, S J Eggers, and H M Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403. ACM, 1995.

[84] L Verlet. Computer "experiments" on classical fluids. I. thermodynamical properties of lennard-jones molecules. *Physical Review*, 159(1):98–103, July 1967.

[85] M Weiss. *Strip mining on SIMD architectures.* ACM, New York, New York, USA, June 1991.

[86] U Welling and G Germano. Efficiency of linked cell algorithms. *Computer Physics Communications*, 182(3):611–615, March 2011.

[87] R H S Winterton. Van der Waals forces. *Contemporary Physics*, 11(6):559–574, November 1970.

[88] M Wolfe. More iteration space tiling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 1989*, pages 655–664, New York, New York, USA, 1989. ACM Press.

[89] Z Yao, J Wang, G Liu, and M Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. *Computer Physics Communications*, 161(1-2):27–35, August 2004.

[90] C Young, N Gloy, and M D Smith. *A comparative analysis of schemes for correlated branch prediction*, volume 23. ACM, May 1995.

[91] G Zhao, J R Perilla, E L Yufenyuy, X Meng, B Chen, J Ning, J Ahn, A M Gronenborn, K Schulten, C Aiken, and P Zhang. Mature HIV-1 capsid structure by cryo-electron microscopy and all-atom molecular dynamics. *Nature*, 497 (7451):643–646, May 2013.

Appendix A

Code Listings

Here we reproduce some lengthier code fragments from our implementations.

A.1 AVX/AVX2 Projection Sorting Force Sweep

```
__m256d vrc   = _mm256_set1_pd(rc);
__m256d vrcsq = _mm256_set1_pd(rcsq);
__m128i vw    = _mm_set1_epi32(4);
__m256d v0    = _mm256_set1_pd(0.0);
__m128i vnt   = _mm_set1_epi32(nt);

for (i = 0; i < nt; i++) {
    if (indices[i] >= nb) continue;

    __m128i vi = _mm_set1_epi32(i);

    __m256d vpxi   = _mm256_broadcast_sd(ps_x + i);
    __m256d vpyi   = _mm256_broadcast_sd(ps_y + i);
    __m256d vpzi   = _mm256_broadcast_sd(ps_z + i);
    __m256d vproji = _mm256_broadcast_sd(projs + i);

    __m256d vfxi = _mm256_set1_pd(0.0);
    __m256d vfyi = _mm256_set1_pd(0.0);
```

```
19      __m256d vfzi = _mm256_set1_pd(0.0);
20
21      int const start = (i+1) & (((int) -1) ^ 0x3);
22      __m128i vj = _mm_add_epi32(
23                      _mm_set1_epi32(start),
24                      _mm_set_epi32(3, 2, 1, 0));
25
26      for (j = start; j < nt; j += 4) {
27          __m256d kprojcutoff = _mm256_cmp_pd(
28                  _mm256_sub_pd(_mm256_load_pd(projs + j), vproji),
29                  vrc,
30                  _CMP_LE_OQ);
31          if (!_mm256_movemask_pd(kprojcutoff)) break;
32
33          __m256d vdx = _mm256_sub_pd(_mm256_load_pd(ps_x + j), vpxi);
34          __m256d vdy = _mm256_sub_pd(_mm256_load_pd(ps_y + j), vpyi);
35          __m256d vdz = _mm256_sub_pd(_mm256_load_pd(ps_z + j), vpzi);
36
37          __m256d vdsq = _mm256_mul_pd(vdx, vdx);
38          #ifdef ENABLE_AVX2
39              vdsq = _mm256_fmadd_pd(vdy, vdy, vdsq);
40              vdsq = _mm256_fmadd_pd(vdz, vdz, vdsq);
41          #else
42              vdsq = _mm256_add_pd(_mm256_mul_pd(vdy, vdy), vdsq);
43              vdsq = _mm256_add_pd(_mm256_mul_pd(vdz, vdz), vdsq);
44          #endif
45
46          __m256d krccutoff = _mm256_cmp_pd(vdsq, vrcsq, _CMP_LE_OQ);
47          __m256d vid = _mm256_cvtepi32_pd(vi);
48          __m256d vjd = _mm256_cvtepi32_pd(vj);
49          __m256d vntd = _mm256_cvtepi32_pd(vnt);
50          krccutoff = _mm256_and_pd(krccutoff,
51                                    _mm256_cmp_pd(vjd, vid, _CMP_GT_OQ));
52          krccutoff = _mm256_and_pd(krccutoff,
```

```
53                                      _mm256_cmp_pd(vjd, vntd, _CMP_LT_OQ));
54
55          if (_mm256_movemask_pd(krccutoff)) {
56              __m256d vcoeff = /* app-dependent */;
57
58              vdx = _mm256_blendv_pd(v0, vdx, krccutoff);
59              vdy = _mm256_blendv_pd(v0, vdy, krccutoff);
60              vdz = _mm256_blendv_pd(v0, vdz, krccutoff);
61
62              #ifdef ENABLE_AVX2
63                  vfxi = _mm256_fmadd_pd(vcoeff, vdx, vfxi);
64                  vfyi = _mm256_fmadd_pd(vcoeff, vdy, vfyi);
65                  vfzi = _mm256_fmadd_pd(vcoeff, vdz, vfzi);
66              #else
67                  vfxi = _mm256_add_pd(vfxi, _mm256_mul_pd(vcoeff, vdx));
68                  vfyi = _mm256_add_pd(vfyi, _mm256_mul_pd(vcoeff, vdy));
69                  vfzi = _mm256_add_pd(vfzi, _mm256_mul_pd(vcoeff, vdz));
70              #endif
71
72  #ifdef N3
73              __m256d vfx = _mm256_mul_pd(vcoeff, vdx);
74              __m256d vfy = _mm256_mul_pd(vcoeff, vdy);
75              __m256d vfz = _mm256_mul_pd(vcoeff, vdz);
76
77              __m128i vind = _mm_load_si128((__m128i *) (indices + j));
78
79              #ifdef ENABLE_AVX2
80                  __m256d vfxj = _mm256_mask_i32gather_pd(
81                          v0, f_x, vind, krccutoff, 8);
82                  __m256d vfyj = _mm256_mask_i32gather_pd(
83                          v0, f_y, vind, krccutoff, 8);
84                  __m256d vfzj = _mm256_mask_i32gather_pd(
85                          v0, f_z, vind, krccutoff, 8);
86              #else
```

```
                __m256d vfxj = shim_mm256_mask_i32gather_pd(
                        v0, f_x, vind, krccutoff, 8);
                __m256d vfyj = shim_mm256_mask_i32gather_pd(
                        v0, f_y, vind, krccutoff, 8);
                __m256d vfzj = shim_mm256_mask_i32gather_pd(
                        v0, f_z, vind, krccutoff, 8);
            #endif

            vfxj = _mm256_add_pd(vfxj, vfx);
            vfyj = _mm256_add_pd(vfyj, vfy);
            vfzj = _mm256_add_pd(vfzj, vfz);

            shim_mm256_mask_i32scatter_pd(f_x, vind, vfxj, krccutoff, 8);
            shim_mm256_mask_i32scatter_pd(f_y, vind, vfyj, krccutoff, 8);
            shim_mm256_mask_i32scatter_pd(f_z, vind, vfzj, krccutoff, 8);
#endif
        }

        vj = _mm_add_epi32(vj, vw);
    }

#ifndef N3
    vj = _mm_add_epi32(_mm_set1_epi32(start), _mm_set_epi32(3, 2, 1, 0));
    vj = _mm_sub_epi32(vj, vw);

    for (j = start - 4; j >= 0; j -= 4) {
        __m256d kprojcutoff = _mm256_cmp_pd(
                _mm256_sub_pd(vproji, _mm256_load_pd(projs + j)),
                vrc,
                _CMP_LE_OQ);
        if (!_mm256_movemask_pd(kprojcutoff)) break;
        // ...
        // as above
        // ...
```

```
121
122         vj = _mm_sub_epi32(vj, vw);
123     }
124 #endif
125
126     f_x[indices[i]] -= shim_mm256_reduce_add_pd(vfxi);
127     f_y[indices[i]] -= shim_mm256_reduce_add_pd(vfyi);
128     f_z[indices[i]] -= shim_mm256_reduce_add_pd(vfzi);
129 }
```

A.1.1 Shim Gather Intrinsic (AVX)

```
1  inline __m256d
2  shim_mm256_mask_i32gather_pd(__m256d src, double const* base_addr,
3          __m128i vindex, __m256d mask, const int scale)
4  {
5      int const m = _mm256_movemask_pd(mask);
6
7      __m256d res = src;
8      if (m & (0x1 << 0))
9          res = _mm256_blend_pd(
10                 res,
11                 _mm256_set1_pd(base_addr[_mm_extract_epi32(vindex, 0)]), 1);
12     if (m & (0x1 << 1))
13         res = _mm256_blend_pd(
14                 res,
15                 _mm256_set1_pd(base_addr[_mm_extract_epi32(vindex, 1)]), 2);
16     if (m & (0x1 << 2))
17         res = _mm256_blend_pd(
18                 res,
19                 _mm256_set1_pd(base_addr[_mm_extract_epi32(vindex, 2)]), 4);
20     if (m & (0x1 << 3))
21         res = _mm256_blend_pd(
22                 res,
23                 _mm256_set1_pd(base_addr[_mm_extract_epi32(vindex, 3)]), 8);
```

```
24
25      return res;
26  }
```

A.1.2 Shim Scatter Intrinsic (AVX/AVX2)

```
1   inline void
2   shim_mm256_mask_i32scatter_pd(double *base_addr, __m128i vindex,
3           __m256d v1, __m256d mask, const int scale)
4   {
5       #define MM_EXTRACT_PD(v, i) \
6           _mm_cvtsd_f64(_mm_shuffle_pd(v, v, _MM_SHUFFLE2(0, i)))
7
8       int const m = _mm256_movemask_pd(mask);
9
10      if (m & (0x1 << 0))
11          base_addr[_mm_extract_epi32(vindex, 0)] =
12              MM_EXTRACT_PD(_mm256_extractf128_pd(v1, 0), 0);
13      if (m & (0x1 << 1))
14          base_addr[_mm_extract_epi32(vindex, 1)] =
15              MM_EXTRACT_PD(_mm256_extractf128_pd(v1, 0), 1);
16      if (m & (0x1 << 2))
17          base_addr[_mm_extract_epi32(vindex, 2)] =
18              MM_EXTRACT_PD(_mm256_extractf128_pd(v1, 1), 0);
19      if (m & (0x1 << 3))
20          base_addr[_mm_extract_epi32(vindex, 3)] =
21              MM_EXTRACT_PD(_mm256_extractf128_pd(v1, 1), 1);
22
23      #undef MM_EXTRACT_PD
24  }
```

A.1.3 Shim Reduce-Add Intrinsic (AVX/AVX2)

```
1   inline double
2   shim_mm256_reduce_add_pd(__m256d a)
```

```
{
    a = _mm256_hadd_pd(a, a);
    a = _mm256_shuffle_pd(a, _mm256_permute2f128_pd(a, a, 0x1), 0x0);
    a = _mm256_hadd_pd(a, a);
    return _mm256_cvtsd_f64(a);
}
```

A.2 KNC/AVX-512 Projection Sorting Force Sweep

```
__m512d vrc   = _mm512_set1_pd(rc);
__m512d vrcsq = _mm512_set1_pd(rcsq);
__m512i vw    = _mm512_set1_epi64(8);
__m512d v0    = _mm512_set1_pd(0.0);
__m512i vnt   = _mm512_set1_epi64(nt);

for (i = 0; i < nt; i++) {
    if (indices[i] >= nb) continue;

    __m512i vi = _mm512_set1_epi64(i);

    __m512d vpxi   = _mm512_set1_pd(ps_x[i]);
    __m512d vpyi   = _mm512_set1_pd(ps_y[i]);
    __m512d vpzi   = _mm512_set1_pd(ps_z[i]);
    __m512d vproji = _mm512_set1_pd(projs[i]);

    __m512d vfxi = _mm512_set1_pd(0.0);
    __m512d vfyi = _mm512_set1_pd(0.0);
    __m512d vfzi = _mm512_set1_pd(0.0);

    int const start = (i+1) & (((int) -1) ^ 0x7);
    __m512i vj = _mm512_add_epi64(
                    _mm512_set1_epi64(start),
                    _mm512_set_epi64(7, 6, 5, 4, 3, 2, 1, 0));

```

```
26      for (j = start; j < nt; j += 8) {
27          if (!_mm512_cmple_pd_mask(
28                      _mm512_sub_pd(_mm512_load_pd(projs + j), vproji),
29                      vrc)) break;
30
31          __m512d vdx = _mm512_sub_pd(_mm512_load_pd(ps_x + j), vpxi);
32          __m512d vdy = _mm512_sub_pd(_mm512_load_pd(ps_y + j), vpyi);
33          __m512d vdz = _mm512_sub_pd(_mm512_load_pd(ps_z + j), vpzi);
34
35          __m512d vdsq = _mm512_mul_pd(vdx, vdx);
36                  vdsq = _mm512_fmadd_pd(vdy, vdy, vdsq);
37                  vdsq = _mm512_fmadd_pd(vdz, vdz, vdsq);
38
39          __mmask8 krccutoff = _mm512_cmple_pd_mask(vdsq, vrcsq);
40          krccutoff = _mm512_kand(krccutoff, _mm512_cmpgt_epi64_mask(vj, vi));
41          krccutoff = _mm512_kand(krccutoff, _mm512_cmplt_epi64_mask(vj, vnt));
42
43          if (krccutoff) {
44              __m512d vcoeff = /* app-dependent */;
45
46              vfxi = _mm512_mask3_fmadd_pd(vcoeff, vdx, vfxi, krccutoff);
47              vfyi = _mm512_mask3_fmadd_pd(vcoeff, vdy, vfyi, krccutoff);
48              vfzi = _mm512_mask3_fmadd_pd(vcoeff, vdz, vfzi, krccutoff);
49
50  #ifdef N3
51              __m512d vfx = _mm512_mask_mul_pd(v0, krccutoff, vcoeff, vdx);
52              __m512d vfy = _mm512_mask_mul_pd(v0, krccutoff, vcoeff, vdy);
53              __m512d vfz = _mm512_mask_mul_pd(v0, krccutoff, vcoeff, vdz);
54
55              __m256i vind = _mm256_load_si256((__m256i *) (indices + j));
56
57              __m512d vfxj = _mm512_mask_i32gather_pd(
58                      v0, krccutoff, vind, f_x, 8);
59              __m512d vfyj = _mm512_mask_i32gather_pd(
```

```
60                      v0, krccutoff, vind, f_y, 8);
61              __m512d vfzj = _mm512_mask_i32gather_pd(
62                      v0, krccutoff, vind, f_z, 8);
63
64              vfxj = _mm512_add_pd(vfxj, vfx);
65              vfyj = _mm512_add_pd(vfyj, vfy);
66              vfzj = _mm512_add_pd(vfzj, vfz);
67
68              _mm512_mask_i32scatter_pd(f_x, krccutoff, vind, vfxj, 8);
69              _mm512_mask_i32scatter_pd(f_y, krccutoff, vind, vfyj, 8);
70              _mm512_mask_i32scatter_pd(f_z, krccutoff, vind, vfzj, 8);
71  #endif
72          }
73
74          vj = _mm512_add_epi64(vj, vw);
75      }
76
77  #ifndef N3
78      vj = _mm512_add_epi64(
79              _mm512_set1_epi64(start),
80              _mm512_set_epi64(7, 6, 5, 4, 3, 2, 1, 0));
81      vj = _mm512_sub_epi64(vj, vw);
82
83      for (j = start - 8; j >= 0; j -= 8) {
84          if (!_mm512_cmple_pd_mask(
85                      _mm512_sub_pd(vproji, _mm512_load_pd(projs + j)),
86                      vrc)) break;
87          // ...
88          // as above
89          // ...
90
91          vj = _mm512_sub_epi64(vj, vw);
92      }
93  #endif
```

```
94
95      f_x[indices[i]] -= _mm512_reduce_add_pd(vfxi);
96      f_y[indices[i]] -= _mm512_reduce_add_pd(vfyi);
97      f_z[indices[i]] -= _mm512_reduce_add_pd(vfzi);
98  }
```

A.3 Shim Packed Store Intrinsic (AVX/AVX2)

This is used to emulate the KNC *packed store* and AVX-512 *compressed store* intrinsics for the AVX and AVX2 code paths.

```
1   inline void
2   shim_mm_mask_packstore_epi32(void *mt, int const mask, __m128i v1)
3   {
4       static __m128i const packmasks[16] = {
5           _mm_set_epi32(3, 2, 1, 0), // 0000, identity
6           _mm_set_epi32(3, 2, 1, 0), // 0001
7           _mm_set_epi32(3, 2, 1, 1), // 0010
8           _mm_set_epi32(3, 2, 1, 0), // 0011
9           _mm_set_epi32(3, 2, 1, 2), // 0100
10          _mm_set_epi32(3, 2, 2, 0), // 0101
11          _mm_set_epi32(3, 2, 2, 1), // 0110
12          _mm_set_epi32(3, 2, 1, 0), // 0111
13          _mm_set_epi32(3, 2, 1, 3), // 1000
14          _mm_set_epi32(3, 2, 3, 0), // 1001
15          _mm_set_epi32(3, 2, 3, 1), // 1010
16          _mm_set_epi32(3, 3, 1, 0), // 1011
17          _mm_set_epi32(3, 2, 3, 2), // 1100
18          _mm_set_epi32(3, 3, 2, 0), // 1101
19          _mm_set_epi32(3, 3, 2, 1), // 1110
20          _mm_set_epi32(3, 2, 1, 0)  // 1111
21      };
22
23      static __m128i const writemasks[5] = {
```

```
24          _mm_set_epi32(        0 ,         0,          0 ,         0 ),
25          _mm_set_epi32(        0 ,         0 ,         0 , ~((int) 0)),
26          _mm_set_epi32(        0 ,         0 , ~((int) 0), ~((int) 0)),
27          _mm_set_epi32(        0 , ~((int) 0), ~((int) 0), ~((int) 0)),
28          _mm_set_epi32(~((int) 0), ~((int) 0), ~((int) 0), ~((int) 0))
29      };
30
31      int const n = _popcnt32(mask);
32      __m128 v = _mm_permutevar_ps(_mm_castsi128_ps(v1), packmasks[mask]);
33      _mm_maskstore_ps((float *) mt, writemasks[n], v);
34  }
```