WARWICK
THE UNIVERSITY OF WARWICK

A Thesis Submitted for the Degree of PhD at the University of Warwick

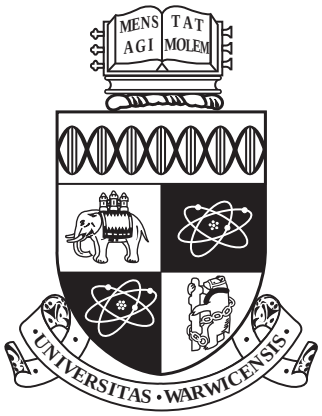
Permanent WRAP URL:

http://wrap.warwick.ac.uk/177400

Copyright and reuse:
This thesis is made available online and is protected by original copyright.
Please scroll down to view the document itself.
Please refer to the repository record for this item for information to help you to cite it.
Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

warwick.ac.uk/lib-publications



Higher-Order Particle Representation for a Portable Unstructured Particle-in-Cell Application

by

Dominic Alan Sidney Brown

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

The University of Warwick

October 2020

Contents

List of Figures

List of Tables

This thesis is dedicated to the memory of my Grandparents.

Alan Brown

(1939 - 2020)

Maureen Audrey Brown

(1941 - 2019)

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Stephen Jarvis, for his encouragement and guidance over the course of the last four years, for providing me with funding, and for giving me the opportunity to undertake a Ph. D.

Secondly, I would like to extend my utmost gratitude to my co-supervisor, Dr. Matthew Bettencourt, at Sandia National Laboratories for his continuous advice and mentorship throughout the project. Without you this work would not have been possible, and I am a better scientist as a result of your efforts. I would also like to thank you for your immeasurable patience when teaching me many domain science concepts, and answering my never-ending barrage of related questions.

Thirdly, I would like to acknowledge the best office-mates I will probably ever have: Richard Kirk, David Truby, and Dean Chester. Particular thanks are reserved for Richard – there from the very start, and still there at the end of all things while writing this thesis. My time in the lab with all of you has been simultaneously the most entertaining and productive time of my academic career, and I will miss your company greatly. Special thanks are also given to Dr. Steven Wright for his advice and support throughout the project, even after leaving Warwick for pastures new at the University of York. Additionally, I thank the other members of the High Performance and Scientific Computing Group, past and present: Dr. James Davis, Dr. Tim Law, Dr. James Dickson, Andrew Owenson, Andrew Lamzed-Short, and Dr. Gihan Mudalige – for both technical and humourous discussions inside and outside of the lab.

Thanks are also given to members of staff at AWE for their support and for providing both guidance and feedback on the work. In particular: Dr. Satheesh Maheswaran, Dr. Martin Nolten, Dr. Seimon Powell, and Professor Richard

Smedley-Stevenson. I am also grateful to John Jones of the Radiation Science Group for initiating the project, and for facilitating my trip to Sandia National Laboratories.

Additionally, thanks go to my friends and family – Dad, Chloe, my grandparents, Daniel Foode, Levi Kane-Thorpe, Pedram Amirkhalili, Christopher Hunt, Fiona Farnsworth, Sophie Lewis, and Alexander Caton – I am extremely grateful for your endless positivity and encouragement. Last, and certainly not least, huge thanks go to my girlfriend, Zoe, without whom this work would likely not have been completed; you deserve a medal for putting up with me throughout my Ph. D.

Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- The EMPIRE-PIC application used throughout this thesis was predominantly developed by a large team of developers at Sandia National Laboratories. However, the kernels analysed, and the algorithmic extensions made in this thesis were developed and tuned by the author, unless stated otherwise.
- Performance data in Chapter 4 for the Trinity, Astra, and Sierra supercomputers were collected by Dr. Matthew Bettencourt and Dr. Stan Moore of Sandia National Laboratories.

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

[18] M. T. Bettencourt, D. A. S. Brown, and G. Radtke. Unstructured Higher-order PIC Methods. In *The 25th International Conference on Numerical Simulation of Plasmas (ICNSP 2017)*, Leuven, Belgium, September 2017

[33] D. A. S. Brown, M. T. Bettencourt, S. A. Wright, J. P. Jones, and S. A. Jarvis. Performance of Second Order Particle-in-Cell Methods on Modern Many-Core Architectures. In *IOP Computational Plasma Physics Conference*, November 2017

[35] D. A. S. Brown, S. A. Wright, and S. A. Jarvis. Performance of a Second Order Electrostatic Particle-in-Cell Algorithm on Modern Many-Core Ar-

chitectures. *Electronic Notes in Theoretical Computer Science*, 340:67–84, October 2018

[17] M. T. Bettencourt, D. A. S. Brown, K. L. Cartwright, E. C. Cyr, C. A. Glusa, P. T. Lin, S. G. Moore, D. A. O. McGregor, R. P. Pawlowski, E. G. Phillips, N. V. Roberts, S. A. Wright, S. Maheswaran, J. P. Jones, and S. A. Jarvis. EMPIRE-PIC: A Performance Portable Unstructured Particle-in-Cell Code. *Communications in Computational Physics*, April 2021. (Accepted)

[34] D. A. S. Brown, M. T. Bettencourt, S. A. Wright, S. Maheswaran, J. P. Jones, and S. A. Jarvis. Higher-Order Particle Representation for Particle-in-Cell Simulations. *Journal of Computational Physics*, June 2021. (In Press)

Sponsorship and Grants

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

- UK Atomic Weapons Establishment (AWE plc):
  "Centre in Computational Plasma Physics" (2014-2020)
  "Electromagnetic Plasma in Radiation Environments" (2016-2019)
  "AWE Technical Outreach Programme" (CDK0724, 2016-2020)

- Engineering and Physical Sciences Research Council (EPSRC):
  "CCP Flagship: A Radiation-Hydrodynamics Code for the UK Laser-Plasma Community" (EP/M011534/1, 2015-2018)

Abstract

As the field of High Performance Computing (HPC) moves towards the era of Exascale computation, computer hardware is becoming increasingly parallel and continues to diversify. As a result, it is now crucial for scientific codes to be able to take advantage of a wide variety of hardware types. Additionally, the growth in compute performance has outpaced the improvement in memory latency and bandwidth; this issue now poses a significant obstacle to performance.

This thesis examines these matters in the context of modern plasma physics simulations, specifically those that make use of the Particle-in-Cell (PIC) method on unstructured computational grids. Specifically, we begin by documenting the implementation of the particle-based kernels of such a code using a performance portability library to enable the application to run on a variety of modern hardware, including both CPUs and GPUs. The use of hardware specific tuning is also explored, culminating in a 3× speedup of a key component of the core PIC algorithm. We also show that portability is achievable on both single-node machines and production supercomputers of multiple hardware types.

This thesis also documents an algorithmic change to particle representation within the same code that improves solution accuracy, and adds compute intensity – an important property where memory bandwidth is limited and the ratio of the amount of computation to memory accesses is low. We conclude the work by comparing the performance of the modified algorithm to the base implementation, where we find that shifting the simulation workload towards computation can improve parallel efficiency by up to 2.5×. While the performance improvements that were hoped for were not achieved, we end this thesis by postulating that the proposed methods will become more viable as compilers and hardware improve.

Abbreviations

ALE Arbitrary Langrangian-Eulerian

ANL Argonne National Laboratory

AVX Advanced Vector Extensions

AVX-512 Advanced Vector Extensions-512

CFD Computational Fluid Dynamics

CFL Courant-Friedrichs-Lewy

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DOE Department of Energy

FDTD Finite Difference Time Domain

FEM Finite Element Method

FLOP/s Floating-Point Operations per Second

FPGA Field Programmable Gate Array

GB/s Gigabytes per Second

GFLOP/s Giga-Floating-Point Operations per Second

GHz Gigahertz

GPGPU General Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HPC High Performance Computing

HPCG High Performance Conjugate Gradients

ICF Inertial Confinement Fusion

ILP Instruction Level Parallelism

I/O Input/Output

ISA Instruction Set Architecture

ITER Iternational Thermonuclear Experimental Reactor

KE Kinetic Energy

KNL Knights Landing

LANL Los Alamos National Laboratory

LINPACK Linear Algebra Package

LLNL Lawrence Livermore National Laboratory

MCF Magnetic Confinement Fusion

MD Molecular Dynamics

MHz Megahertz

MIMD Multiple Instruction Multiple Data

MIPS Millions of Instructions per Second

MISD Multiple Instruction Single Data

MPI Message Passing Interface

NAS NASA Advanced Supercomputing Division

NERSC National Energy Research Scientific Computing Center

NIF National Ignition Facility

NPB NAS Parallel Benchmarks

NUMA Non-Uniform Memory Access

OoO Out-of-Order

PE Processing Element

PGAS Partitioned Global Address Space

PIC Particle-in-Cell

PPC Particles per Cell

RAM Random Access Memory

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SISD Single Instruction Single Data

SPMD Single Program 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

TDP Thermal Design Power

TEM Transverse Electromagnetic

TFLOP/s Tera-Floating-Point Operations per Second

TTS Time-to-Solution

UVM Unified Virtual Memory

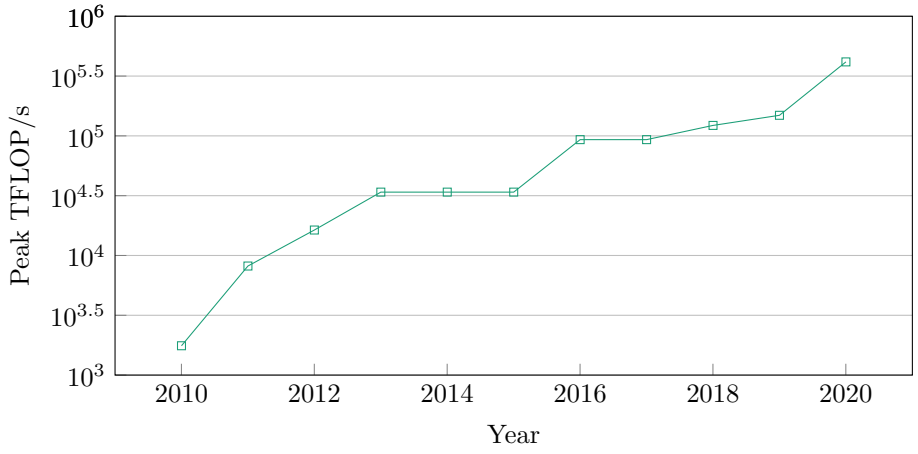
WARPP Warwick Performance Prediction Toolkit

CHAPTER 1

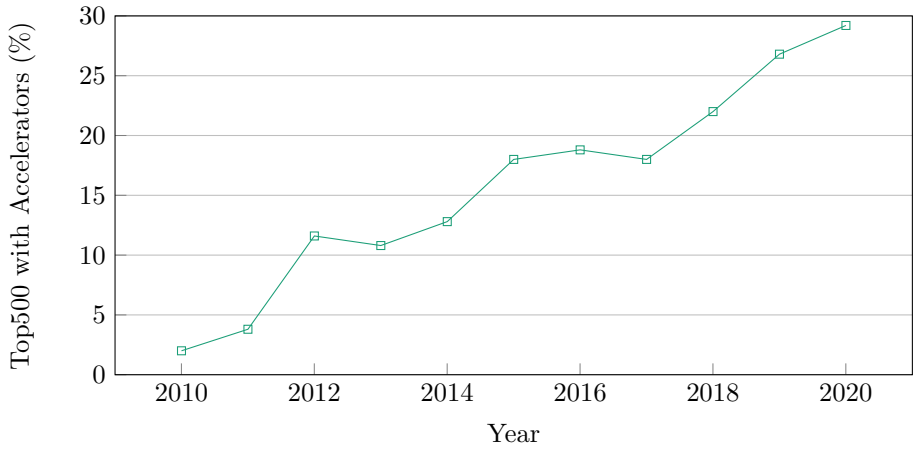
Introduction

Since the invention of the first programmable computer, Colossus, in the 1940s computers have rapidly become a core part of modern society. Along with this significant shift, there has been a radical change in the way that scientific experiments are performed by researchers of today, with computers being an invaluable tool for many scientists. Whereas previously all experimentation was conducted physically, it is now possible to carry out mathematical simulations of real-world phenomena in a variety of theoretical and applied fields. The ability to do so is especially useful in situations where the required experiments are particularly dangerous, time consuming, infeasible, or prohibitively expensive to carry out. The discovery of new chemical compounds, the simulation of fluid dynamics systems, and weather forecasting are three examples of situations where computer systems play a vital role.

While simulation via computation has a wide variety of applications, the fidelity of a given simulation is limited by the amount of computational power available, meaning that general purpose, consumer grade machines are unsuited for these workloads. Therefore, we turn to the use of *supercomputers*, machines that are many multitudes more powerful than standard desktops. The branch of computer science concerned with the development of supercomputers and their efficient use is known as High Performance Computing (HPC). Modern supercomputers are capable of performing vast numbers of mathematical operations every second, and the peak performance of production supercomputers is ever improving. Figure 1.1(a) highlights this fact, showing how the performance of the fastest supercomputer in the world has changed over the past decade; peak performance has increased by approximately 2.5 orders of magnitude since



(a) Achieved peak performance.



(b) Accelerator count.

Figure 1.1: Achieved peak performance of the number one supercomputer in the world and accelerator count in the TOP500 list from 2010–2020.

2010. Computer scientists and engineers are now striving to reach the milestone of Exascale computing – the ability to carry out one quintillion (10^{18}) calculations per second. At the time of writing, the world's fastest machine is the ARM-based Fugaku [61], located at the RIKEN Center for Computational Science in Kobe, Japan, which achieved a peak performance of 415,530 Tera-Floating-Point Operations per Second (TFLOP/s) in the June 2020 TOP500 rankings [143].

Historically, supercomputers were based on many interconnected homoge-

nous compute nodes, each with few Central Processing Units (CPUs) operating at high clock rates resulting in relatively low levels of intra-node parallelism. Improvements in the performance of these machines was typically driven by increased CPU frequency, allowing more instructions to be issued in the same period of time. However, growing cost, thermal output and power consumption mean that this strategy is no longer viable; a different approach is needed in order to realise Exascale computing. As a result, hardware manufacturers now opt to make use of many lower frequency cores, or seek to pair the CPU with specialised accelerators that are suited to highly parallel computation, i.e., the use of a heterogeneous system. The use of many-core CPUs and accelerators is becoming increasingly common, with nearly 30% of the current top 500 supercomputers using some form of accelerator (Figure 1.1(b)). This trend is expected to continue as growing numbers of HPC centres seek the power efficiency and performance advantages offered by heterogeneous computing.

Science aided by HPC systems continues to advance, and maintaining this progress requires computational simulations of much greater complexity and accuracy, thus demanding improved supercomputer performance. This requirement is a key motivation for the continued financial investment and research effort into the field of HPC. However, the advent of modern hardware that is progressively more complicated makes it difficult for application developers to leverage the increased performance that is theoretically available. As a consequence, developers must spend more time optimising current codes or developing algorithms which are better suited for modern machines, a task which is made more challenging as computer hardware now operates at an unprecedented scale and continues to diversify.

This thesis investigates the performance issues present in modern computational plasma physics applications, and seeks to understand how they can be alleviated. Specifically, it examines the use of a performance portability library to allow such codes to execute and perform well on a variety of modern architectures while maintaining only a single codebase. It additionally explores algorith-

mic changes that are now viable on massively parallel modern hardware. The proposed extensions raise the computational intensity of the simulation and also offer increased accuracy, potentially having an advantage where limited memory bandwidth poses an obstacle to performance.

1.1 Motivations

As discussed above, the emerging compute architectures of the modern age offer unparalleled levels of floating-point performance, and the peak performance of production supercomputers continues to increase at a rapid rate. This means that both new and existing scientific simulation codes must be able to fully exploit the extreme amounts of parallelism available in order to make optimal use of the hardware, allowing experiments to be carried out at a much larger scale. Moreover, the growth in compute performance has historically exceeded the improvement in memory latency and bandwidth, and this trend seems set to continue. As a result, the memory subsystem now increasingly poses an obstacle to performance – compute units can currently process data more quickly than it can be delivered for a variety of application types. Consequently, achieving the peak performance of a modern machine is a challenging and ever-evolving task.

The surge in architecture diversity has produced a rich landscape of available hardware and programming models. The MPI-only approach of the past is now less than ideal – many alternatives exist for intra-node parallelism. In the case of accelerators, the use of a specialised programming model is often a requirement. Therefore, it is becoming ever harder for developers to modernise existing large scale production codes (or develop modern replacements) to enable them to fully leverage this hardware. This problem is exacerbated by the fact that different architectures require specific code optimisations in order to function optimally. Scientific codes must be able to adapt to these requirements without the difficulty of developing and maintaining several versions of the same

application, a concept known as performance portability.

The work documented in this thesis focuses on modern computational plasma physics simulations, specifically those that make use of the Particle-in-Cell (PIC) method on unstructured computational meshes. Simulation of plasma via the PIC algorithm has a variety of applications in both academia and industry including, but not limited to: simulation of fusion energy devices, plasma behaviour in astrophysical settings, interactions between laser beams and plasmas, and various types of nuclear experimentation. The United Kingdom is a key contributor to many of these fields. For example, the UK-based Orion Laser Facility is an Inertial Confinement Fusion (ICF) device used to recreate extreme temperature and pressure conditions – devices such as these are often simulated with PIC codes. It is clear to see that both the improvement and future-proofing of plasma simulation codes aids scientific solutions and discoveries. Such future-proofing is a key focus of the work in this thesis; Chapter 4 seeks to assess the suitability of a performance portability library for an unstructured PIC code on a range of modern hardware, and demonstrates that such portability is achievable. This thesis also documents the addition of an algorithmic change to particle representation within the same code that improves solution accuracy, and adds compute intensity – an important property where memory bandwidth is limited and the ratio of the amount of computation to memory accesses is low. The performance behaviour of the modified implementation is contrasted to the base code, where it is shown that the proposed particle representation is viable on modern hardware.

1.2 Thesis Contributions

The research presented in this thesis makes the following contributions:

- The development and implementation of the particle-based routines of the unstructured PIC application EMPIRE-PIC using the Kokkos performance portability library is documented. While traditional flat parallelism

is used for the majority of the routines, the use of hierarchical parallelism to express more complex algorithms that make improved use of highly parallel hardware is also explored. Through a benchmarking comparison it is shown that EMPIRE-PIC performs well across a variety of modern hardware types including both Intel and ARM CPUs, Intel's Xeon Phi Knights Landing (KNL), and NVIDIA Tesla GPU-based systems at the single node level. Additionally, a scaling study is performed, demonstrating that EMPIRE-PIC successfully scales up to more than two thousand GPUs, and greater than one hundred thousand CPUs;

- The core PIC algorithm within EMPIRE-PIC is extended to make use of a higher-order particle representation, specifically a smooth quadratic shape function with compact support on some defined radius. This shape is represented as a collection of virtual particles surrounding each super-particle in order to give the super-particle this smoother shape. The virtual particles have fixed offsets and weights obtained from Gaussian quadrature rules and the chosen radius. As the virtual particles are purely computational, we obtain the additional benefit of adding increased arithmetic intensity to traditionally memory-bound particle routines. The algorithmic changes are validated, and the accuracy of the modified code is analysed and compared to the base implementation of EMPIRE-PIC using four representative benchmark problems;

- A performance analysis of the higher-order particle representation via the virtual particles method versus the base EMPIRE-PIC code is performed. We show that the use of virtual particles is less expensive than expected on all systems considered, and improves strong scaling for highly parallel CPUs for electrostatic simulations. A cost versus error analysis is also performed, comparing the original implementation to the higher-order particle representation. We show that while virtual particles can outperform the base EMPIRE-PIC implementation on specific inputs, in general it is

still preferable to increase the amount of traditional simulation particles used.

1.3 Thesis Overview

The remainder of this thesis is structured as follows:

Chapter 2 introduces the reader to the various types of parallel computational hardware that exist today, alongside the different programming models that can be used to leverage them. It also details the theoretical concepts that underpin parallel computing, and a brief introduction to the practical sides of performance engineering, and the challenges involved in making full use of the hardware. The chapter concludes by detailing the hardware specifications of the various systems used to collect the performance data presented in this thesis.

Chapter 3 provides an overview of conducting simulations of plasma phenomena using the PIC method, and motivates the use of unstructured meshes for such simulations. We begin with the underlying physical equations, before moving on to the derivation of the unstructured PIC algorithm for both electrostatic and electromagnetic problems. It also covers the notion of higher-order methods in computational physics, and the associated benefits and drawbacks of using these methods. Finally, we end the chapter with a discussion of the hybrid nature of PIC algorithms, and general approaches to how they can be parallelised.

Chapter 4 documents the use of Kokkos to develop performance portable unstructured PIC simulations. Specifically, it presents the implementation of the particle-based routines of the PIC code EMPIRE-PIC, and also explores hardware specific tuning for the particle move and charge deposition routines. The performance of the application is evaluated at the single node level, and at scale on multiple supercomputers: Trinity, Astra and Sierra. The analysis considers a variety of hardware types including traditional CPUs, many-core CPUs, and NVIDIA GPUs.

Chapter 5 extends the core algorithm of EMPIRE-PIC to make use of an alternative higher-order approach to the representation of simulation particles for both electrostatic and electromagnetic unstructured FEM-PIC simulations. In the modified PIC algorithm we represent particles as having a smooth quadratic shape function limited by some specified finite radius, r_0. A key feature of our approach is the representation of this shape by surrounding simulation particles with a set of computational virtual particles with delta shape, with fixed offsets and weights derived from Gaussian quadrature rules and the value of r_0. In addition to raising simulation accuracy, the modifications have the effect of increasing the arithmetic intensity of traditionally memory-bound particle routines with only a minor increase in memory usage. The effect of the algorithm on simulation solutions is explored using four representative benchmark problems that cover both electrostatics and electromagnetics, in addition to two- and three-dimensional geometries. Good error reduction across all of the chosen problems is achieved as the particles are made progressively smoother, with the optimal particle radius appearing to be problem-dependent.

Chapter 6 builds on the all of the work previously presented in this thesis. It details the implementation of the higher-order particle representation in EMPIRE-PIC using Kokkos, and explains the design decisions made from a performance perspective regarding both the additional memory required, and the modifications made to the particle-based routines. A performance analysis is presented, quantifying the cost and strong-scalability of the proposed extensions on a variety of modern compute architectures. This chapter concludes with a cost versus error analysis of the base and extended PIC algorithms for electromagnetic simulations.

Chapter 7 concludes the thesis, and discusses the implications of the work that has been presented. The limitations of the research are also highlighted, and avenues for future work are proposed.

CHAPTER 2

Parallel Hardware and Performance Engineering

As High Performance Computing (HPC) moves towards the major milestone of Exascale computing, modern hardware is becoming increasingly parallel and continues to diversify. Additionally, increases in peak computational performance have historically driven scientific development as a whole by allowing domain scientists to tackle problems of ever-increasing complexity [102]. In light of these facts, this chapter provides an overview of modern computational hardware and programming models, and also introduces various key concepts that underpin parallel computing and performance engineering. Finally, we provide detail on the various platforms used to collect the performance data presented in this thesis.

2.1 Types of Parallelism

Flynn's Taxonomy, proposed by Michael J. Flynn in 1966, provides a framework for the classification of computer architectures that considers whether they can handle multiple streams of data and/or instructions [56]. In this context an instruction refers to a single simple operation such as a multiply, and data can be thought of as the number(s) to which instructions can be applied. Under this system both instructions and data can be classed as either single or multiple. These can be considered as serial or parallel, respectively, which results in a classification consisting of four sub-groups in total. Figure 2.1 shows Flynn's Taxonomy and demonstrates the possible combinations of single/multiple instructions and data.

Single Instruction Single Data (SISD) refers to computation that is exclusively serial, where only a single instruction is performed on a single data item

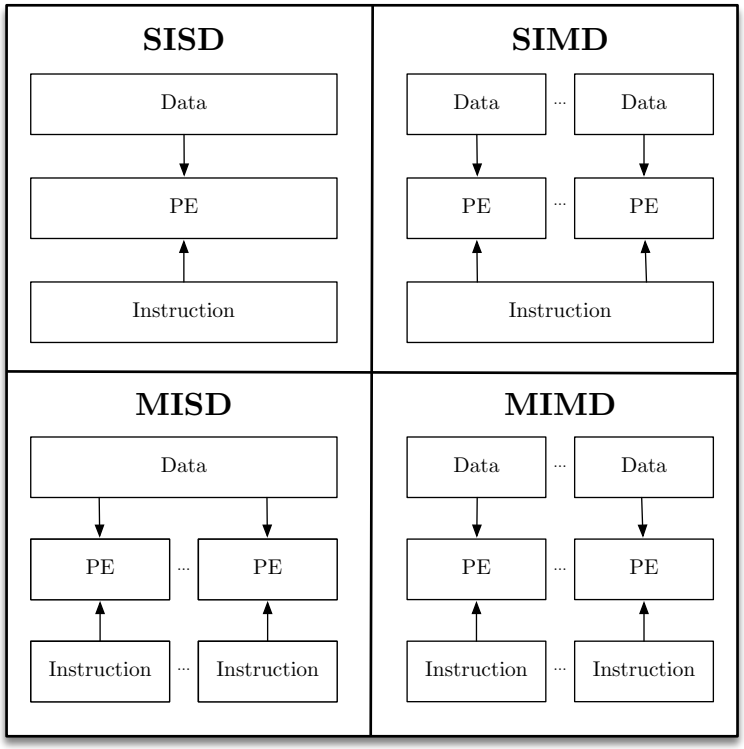


Figure 2.1: Flynn's Taxonomy shows how parallel processing elements (PEs) can be applied to instructions and data.

at any given time during execution. In terms of hardware this is analogous to a single-core, sequential computer loading a single instruction from memory at a time, and dispatching this instruction to a single Processing Element (PE). SISD computers are rare today as almost all modern hardware exhibits some level of parallelism, such as possessing multiple Central Processing Unit (CPU) cores. This is seen with both Personal Computers (PCs) and supercomputers.

Single Instruction Multiple Data (SIMD) is a form of parallel computation where a single common instruction can be executed on multiple data items simultaneously. For example, the elementwise multiplication of one or more numerical vectors can be achieved by carrying out all of the multiplications at once, as opposed to performing each operation in turn. This is typically realised in hardware through the use of specialised vector registers and vector processing units. Example implementations of SIMD include Intel's Streaming

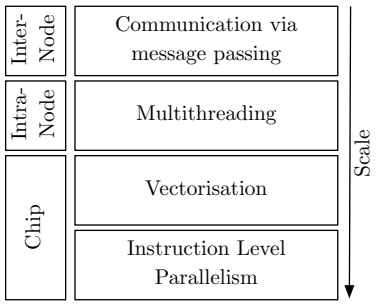


Figure 2.2: The various levels of parallelism, from highest scale down to lowest scale.

SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) extensions to the x86 Instruction Set Architecture (ISA).

Multiple Instruction Single Data (MISD) is by far the least commonly seen type of parallel computer. This is primarily due to it being difficult for different hardware to process the same data item in parallel despite this being theoretically possible. This leads to MISD only being used in extremely specialised use cases; one such use case is the flight control computer of a space shuttle [137].

Multiple Instruction Multiple Data (MIMD) is by far the most commonly used form of parallelism in the computers of today. Both PCs and supercomputer systems are often found within this category. This type of parallelism is characterised by having multiple independent PEs, where each element is capable of reading separate data items and operating on them independently. Typically, this is implemented in the form of multithreading. The Single Program Multiple Data (SPMD) paradigm can also be considered to be part of MISD. In the case of SPMD, multiple processors run the same program independently, with the data explicitly decomposed over the processors. As all processors are independent, each can be at different points in the program execution at any one time.

In highly diverse modern hardware parallelism can be thought of as *hierarchical*, i.e., various levels of parallelism exist that must be properly leveraged in order to achieve peak application performance. These levels of parallelism are shown in Figure 2.2, where each layer can make use of the layers below it. The

top layer refers to parallelism across multiple compute nodes in a supercomputer where data must be transferred using inter-node message passing. This is known as distributed memory parallelism. Below this level we have intra-node parallelism achieved by multithreading across many CPU cores, with each thread handling its own computation. This is often referred to as shared memory parallelism. One step lower, vector instructions taking advantage of SIMD can be used to process data items in parallel. At the lowest level we have Instruction Level Parallelism (ILP), meaning that instructions without dependencies between them can be executed simultaneously.

2.1.1 Instruction Level Parallelism

ILP refers to the idea that some portion of the instructions that comprise a computer program can be executed in parallel. This can be implemented in a variety of ways in computer hardware. One such method is instruction pipelining within a processor, where the pipelined processor can handle multiple instructions at the same time. Typically this means that while one instruction is being executed, subsequent instructions can still be loaded from memory and decoded. Therefore, all parts of the processor can be kept busy with useful work, reducing the amount of time spent idle. Additional methods to expose parallelism at the instruction level include superscalar instruction dispatch and Out-of-Order (OoO) execution. Unlike a traditional scalar processor, superscalar processors can execute multiple instructions in a given CPU clock-cycle as they are capable of dispatching multiple instructions to separate functional units of the processor at the same time. This can additionally be combined with instruction pipelining by fetching, decoding, and executing multiple instructions simultaneously. OoO execution allows a set of instructions in a queue to be executed in an arbitrary order, with the caveat that the chosen order cannot violate dependencies between instructions. Dependencies can include both *data dependencies*, where a later instruction requires the output of a previous instruction, and *control dependencies*, where the outcome of a branch statement will deter-

mine whether a given instruction is executed or not. In the case where a data dependency can be resolved by the compiler it is still possible to achieve ILP. However, in the case of a true data dependency which cannot be resolved no ILP can occur if correctness is to be maintained. The performance impact of control dependencies can be mitigated through the use of branch prediction, where a branch predictor attempts to guess which code path will be taken during program execution. This is known as *speculative execution*, where the processor will execute the instructions of the most likely branch before the branch has actually occurred with the aim of reducing idle time in the pipeline, thus leading to performance improvements. In the case of an incorrect branch being chosen there is a performance penalty as the pipeline must be emptied and the correct instructions loaded and executed. In light of this, significant research has been undertaken to develop accurate dynamic branch prediction methods [156].

2.1.2 Vectorisation

Vectorisation is the concept represented by the SIMD sub-group of Flynn's Taxonomy (Figure 2.1), which seeks to exploit parallelism at the data level. Typically, vectorisation is achieved in hardware through the use of vector registers and vector processing units which allow for vector operations to be performed on arrays of data. Example implementations include Intel's SSE and AVX instruction sets, which exploit 128-bit and 256-bit vector registers, respectively. In the case of AVX, this means that eight integer data items can be processed simultaneously, with this number reducing to four when double-precision floating-point numbers are used. Fully exploiting the SIMD capability of modern hardware is crucial – as the width of vector units grows, so does the performance penalty for failing to take advantage of this functionality. This is particularly the case for compute-bound HPC codes. In recent years with the release of Intel's Xeon Phi Knights Landing (KNL) many-core processor and the Skylake family of Xeon CPUs, the AVX instruction set has been expanded to support 512-bit operations through the use of Advanced Vector Extensions-512 (AVX-512), allowing

up to eight double-precision values to be handled at once. However, the use of AVX-512 can cause certain Intel processors to operate at a lower clock frequency to reduce thermal output, requiring the performance trade-off to be carefully considered [80].

In general, there are two ways that application developers can ensure that codes vectorise as desired. The first is allowing the compiler to auto-vectorise code where possible, by generating vector instructions from scalar code. This relies on the compiler being able to prove there are no data dependencies that would render vectorisation unsafe to perform. In cases where the compiler refuses to auto-vectorise code due to falsely perceived dependencies, a developer can encourage vectorisation through the use of compiler directives that allow the dependencies to be ignored. The second approach allows programmers to write hand-implemented vectorised code, either by directly writing assembly code or using wrapper functions available in higher level languages. These wrapper functions are referred to as SIMD intrinsics. While having explicit manual control over vectorisation can lead to performance improvements there are also significant downsides to using this approach. The code tends to be much more complex as a result of the low-level nature of intrinsics. This makes the code much harder to read and, therefore, significantly more challenging to extend and maintain. Additionally, intrinsics are often platform-specific or optimised for specific SIMD hardware, severely reducing the portability of an application. For these reasons compiler auto-vectorisation of scalar code is often the preferred method, with hand-vectorisation used as a last resort for when auto-vectorisation fails even with the assistance of directives.

2.1.3 Multithreading

Improved computational performance has historically been associated with increased CPU complexity and clock speeds, as was seen throughout the 90s and the beginning of the 2000s. This is in part due to the trend of the number of transistors present in an integrated circuit doubling approximately every two

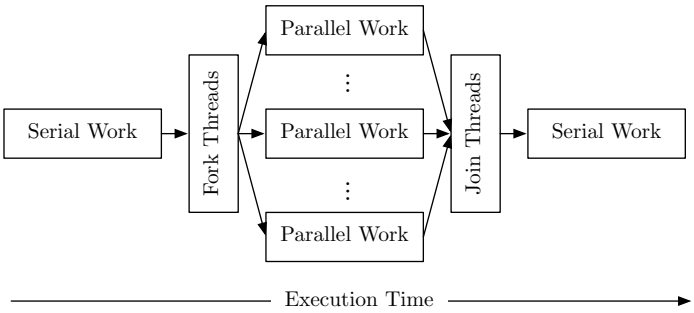


Figure 2.3: Fork-Join Model of Multithreaded Execution

years, as predicted by Moore's Law [106]. It has since become clear that the consequential increase in CPU power consumption (and the associated costs) and resultant thermal output is no longer sustainable nor acceptable to consumers. As a result, chip manufacturers have instead adopted the multi-core model, where a single high-frequency core is replaced by multiple cores that each operate at a lower clock rate. In this way multi-core systems offer manageable thermal output and improved efficiency in terms of Floating-Point Operations per Second (FLOP/s) per Watt consumed. However, these benefits can only be realised if the programs being run are capable of properly leveraging the additional CPU cores that are available. In general this is accomplished by applications using multiple *threads* of program execution, where each thread can act independently. These threads can exploit both instruction and data parallelism as in the MIMD class of Flynn's Taxonomy. Modern hardware also supports the use of numbers of threads greater than the number of physical cores present on-chip. This can be accomplished via *temporal* multithreading (also known as super-threading) where the CPU can context switch between threads at runtime, giving the appearance of simultaneous execution despite only the instructions of a single thread being present in the pipeline. More modern systems often support Simultaneous Multithreading (SMT), otherwise known by 'hyperthreading' as coined by Intel. This allows multiple threads to share the functional units of a single core, enabling instructions from different threads to be present in the same pipeline simultaneously [145]. Two-way SMT

is the most common option available in modern processors, but some architectures such as Intel's KNL and ARM's ThunderX2 support up to four-way SMT. SMT is also used on Graphics Processing Unit (GPU) cards, to a much higher degree than the typical CPU system. On a CPU system, multithreading at the low-level is exposed via the `pthreads` library, which allows developers to write multithreaded software in a shared memory context according to the POSIX standard [111].

Similarly to SIMD intrinsics, while `pthreads` offers powerful low-level control to application developers, this again comes at the expense of code readability and extensibility. Moreover, `pthreads` code is often more difficult to implement than multithreaded code written with the aid of a higher-level library. It is for this reason that the Open Multi-Processing (OpenMP) standard is the most commonly used method of writing multithreaded scientific applications [118]. OpenMP is written using compiler directives known as pragmas that instruct the compiler to parallelise sections of application code. Usually, this means that the iterations of a `for` loop are broken up into chunks and distributed across threads. OpenMP employs the well-known fork-join model, where a master thread forks additional threads which carry out the pragma marked sections of code in parallel. Once the parallel section of code has been executed, the forked threads then rejoin the master thread, and serial code execution continues. This process is shown pictorially in Figure 2.3.

While there are clear benefits to the multithreaded model, there are also some downsides that must be considered. Most multi-core systems are made up of at least two separate CPU sockets, while logically appearing as a single processor. Each socket has relatively fast access to its own memory, but much slower access to memory that is closer to the other processor. This is known as Non-Uniform Memory Access (NUMA), where each socket is considered as an individual NUMA node. This means that NUMA must be considered with regards to process and/or thread placement in order to minimise the number of non-local memory accesses that will occur in order to avoid huge performance

penalties as a result of large amounts of memory traffic. Additionally, while cores have the benefit of having individual L1 caches, when data is accessible by multiple cores the issue of cache coherence arises. When a shared variable is written to by a given core, each other core must then invalidate the copy of the variable in its own local cache, and subsequently re-fetch the value from main memory. False sharing can also occur when multiple threads alter different variables that share the same cache line, causing the entire cache line to be invalidated and re-fetched. As a result, great care must be taken when writing multithreaded code in order to avoid this issue.

2.1.4 Message Passing

At the highest level of scale depicted in Figure 2.2, parallelism extends to multiple compute nodes. In this case, processors only have access to their own local memory space and must instead communicate over a network fabric explicitly via *message passing* in order to exchange data with remote nodes. The most ubiquitous library used to accomplish this within the HPC space is the Message Passing Interface (MPI) library [104]. In an MPI program each process runs a separate copy of the program and communicates with other processors when necessary. This is an example of the SPMD paradigm within Flynn's Taxonomy. Alternatives to MPI have also been proposed, including but not limited to: HPX [84], and Charm++ [85]. The idea behind these other options is to abstract away the need for programmers to write explicit message-passing code, instead allowing for a more conceptual approach, and also offering additional features such as load balancing and fault-tolerance. Partitioned Global Address Space (PGAS) programming models that assume a logically partitioned, but global, memory address space have also been proposed as explicit message-passing alternatives. Examples of these include coarray Fortran [113], and UPC++ [157]. Despite the existence of these alternatives, MPI currently remains the de facto standard for writing distributed memory HPC programs.

While MPI assumes a distributed memory model, it is frequently used for

intra-node parallelism instead of a shared memory solution. This is particularly true for legacy production codes that were developed before modern multi-threading libraries. While this approach is often simpler due to developer familiarity with MPI, overuse can lead to inefficient use of on-node shared memory and an increase in overheads due to the large amount of MPI communications that occur between processes (as each thread is a separate process). In order to alleviate the issue of large amounts of communications, many modern applications adopt a *hybrid* MPI plus OpenMP approach. Typically, hybrid applications use one MPI process per socket, with OpenMP used for shared memory parallelism within a socket, thus reducing communications while also taking advantage of NUMA hardware. Some studies have shown that hybrid codes exhibit comparable or even superior performance to their MPI only variants at large scale due to the reduced communications profile [40, 124]. However, modern MPI implementations often conduct intra-node communications through the use of shared memory, narrowing the performance gap between MPI-only codes and their hybrid counterparts.

2.2 The Memory Hierarchy

All computer systems require the ability to store data and the instructions that are carried out on that data. As CPU operating speeds have increased, the memory subsystem has become a performance bottleneck for many HPC applications. As a result, it is key to maximise memory bandwidth, and minimise access latency as much as possible to ensure that the CPU does not sit idle while waiting for new data to be retrieved from memory. However, memory that is quick to access is expensive to produce meaning that such memory is extremely small in capacity. Registers and L1 cache made up of Static Random Access Memory (SRAM) are common examples of low capacity memory with fast access times. On the other hand, access time can be sacrificed in situations where storage capacity is the primary concern – building an entire storage system out

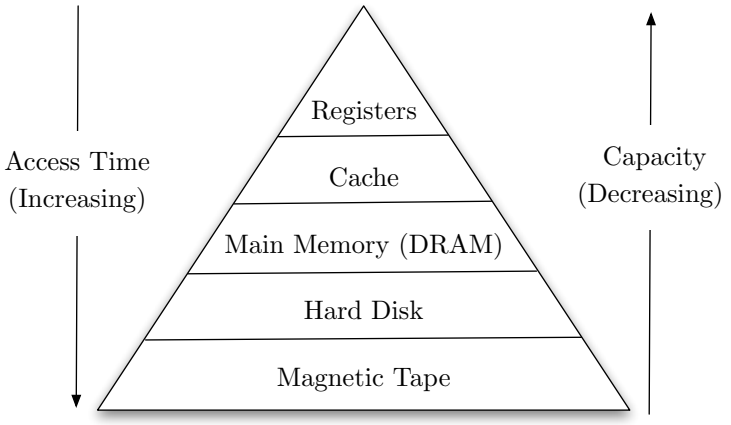


Figure 2.4: The memory hierarchy. The fastest, lowest capacity, and most expensive memory at the top, the slowest, largest capacity, and cheapest memory at the bottom.

of SRAM would be hugely expensive. Data backups are one such example where storage capacity is key and speed of access is largely unimportant. Here, storage with lower cost per byte can be chosen, e.g., traditional spinning hard disk drives or magnetic tape.

The above concepts mean that, much like parallelism, memory systems can also be considered to be hierarchical in nature. This is illustrated by the well-known memory hierarchy diagram shown in Figure 2.4, where the trade-off between access time, capacity, and cost can be clearly seen.

2.3 Many-Core & Heterogeneous Computing

As compute architectures have continued to diversify, so-called *many-core* processors have become a core part of the HPC processor space. Where multi-core processors are capable of acceptable performance for both serial and parallel workloads, a many-core processor prioritises high levels of parallelism, at the expense of subpar serial performance. As a result, codes that make poor use of high levels of parallelism are unlikely to perform well on such a processor.

The many-core class of processors can include CPU-like architectures such as Intel's Xeon Phi range (both as coprocessor cards, and standalone CPUs), as

well as GPUs manufactured by AMD and NVIDIA. Systems containing some type of many-core processor are becoming increasingly common. For example, at the time of writing, seven of the top ten supercomputers on the TOP500 list make use of some sort of many-core processor [143].

The increased prevalence of many-core processors has also given rise to the concept of *heterogeneous* computing, where a single compute node contains processors of differing types. This is frequently set up as a traditional CPU, accompanied by one or more accelerators, such as GPUs or Field Programmable Gate Arrays (FPGAs). GPUs have long been used to accelerate massively parallel computation – processing graphical image and video data is but one example. The application of GPU accelerators to standard computation is referred to as General Purpose Graphics Processing Unit (GPGPU) computing, with NVIDIA's Compute Unified Device Architecture (CUDA) [116] being perhaps the most well known framework used to achieve this. Many scientific codes are beginning to make use of this constantly evolving technology as a result of the increased computational power and memory bandwidth on offer. Therefore, it has become increasingly important for codes to be able to take full advantage of this hardware. However, as different supercomputers make use of varying technologies, the *portability* of a given codebase has become a key issue that must be considered.

2.3.1 Performance Portability

Heterogeneous systems that make use of accelerators or many-core CPUs are rapidly becoming more prevalent as we continue to move away from the traditional homogeneous cluster systems that were previously the norm [15]. The upcoming Exascale machines Aurora and Frontier will follow this same trend by using Intel and AMD GPUs respectively. As a result, it is now highly desirable for scientific codes to be able to perform well across a wide variety of systems, a concept often referred to as 'performance portability'. However, this increase in architectural diversity brings greater difficulty in implementing production codes

that are capable of fully exploiting the available hardware resources when compared to the traditional SPMD MPI-based approach of the past. This problem is exacerbated by the fact that each architecture requires specific optimisations in order to achieve peak performance. Scientific codes must be able to adapt to this changing landscape without the difficulty of developing and maintaining several versions of the same application.

There are various standards that have been proposed to remedy this issue by providing directives to the compiler that sections of code should be run in parallel and/or on a given device – commonly an accelerator card. These include OpenMP [118] and OpenACC [117]. Another approach being considered to aid performance portability is the use of parallel programming frameworks or libraries. Examples include Kokkos [52], from Sandia National Laboratories (SNL), and RAJA [76], from Lawrence Livermore National Laboratory (LLNL), both of which make use of C++ template meta-programming to inject hardware-specific device code, targeting a system during compilation. Other notable examples of parallel programming frameworks include Khronos' OpenCL [86] and SYCL [87], and Intel's newly developed OneAPI [79]. The work presented in Chapters 4 and 6 makes use of Kokkos to enable the use of a variety of modern compute architectures.

While it is clear that there are many performance portability options available to developers, actually quantifying the portability of a given code in a rigorous manner remains a difficult task. At the time of writing, there exists only a single metric for measuring the performance portability of an application, proposed by Pennycook et al. [120], which takes the harmonic mean of an application's performance efficiency across a specific set of platforms. This metric has driven multiple further studies into the portability of several application types by various authors [44, 88, 127].

2.4 Performance Engineering

Performance engineering refers to the set of methods and techniques used by application engineers and computer scientists to predict, measure, and improve the performance of a given application. In this context, *performance* can refer to several different metrics including, but not limited to: Time-to-Solution (TTS), memory bandwidth and footprint [121], Input/Output (I/O) throughput [46], and total power consumed [130]. This section provides a broad overview of both the practical and theoretical sides of performance engineering, which is crucial to understand parts of the work presented in this thesis.

2.4.1 Theory of Parallel Computing

While performance engineering is clearly a practical exercise, it is also underpinned by theoretical concepts. These are presented below in the form of definitions and mathematical equations which can be applied in practice to compare and understand HPC application performance.

Speedup

Speedup is the most basic metric with which to quantify application performance improvements by considering how runtime changes as the amount of PEs is increased. Given the values of serial runtime, T_S, and the runtime using P processors, T_P, speedup can be defined as follows:

$$S_P = \frac{T_S}{T_P} \tag{2.1}$$

This is often the first metric considered when analysing application scalability. In the ideal case $S_P = P$ – this is referred to as *linear* speedup. Speedup is often the result of *strong scaling*, where a fixed problem size is ran on an increased number of processors in an attempt to reduce the TTS.

Parallel Efficiency

Parallel efficiency builds on the notion of speedup by quantifying how much of the theoretical performance improvement an application makes use of when running on P processors.

$$E_P = \frac{S_P}{P} \tag{2.2}$$

The value of parallel efficiency generally resides in the interval between zero and one, where the ideal linear case would result in a value of one. It is also possible to obtain values of parallel efficiency greater than one, often known as a *super-linear* speedup. This result is extremely uncommon for production applications, and is usually the result of code exploiting architecture specific benefits. One example would be where the problem size per processor becomes small enough to reside entirely within CPU cache as a result of strong scaling.

Amdahl's Law

Initially proposed in 1967, Amdahl's Law is an equation which characterises the maximum speedup achievable by increasing the amount of processors used to run a parallel code [4]. The intuition behind the theory is that if a code possesses some serial fraction F_S (and by extension some parallel portion $F_P = 1 - F_S$) then the maximum speedup achievable for that code will become limited by F_S as the number of processors continues to grow. The formal statement of Amdahl's Law is as follows:

$$S_P \leq \frac{1}{F_S + \frac{F_P}{P}} \tag{2.3}$$

It is clear to see that as P tends towards ever-larger values, the value of S_P becomes dominated by the serial fraction. One should also note that Amdahl's Law is closely related to the previously discussed notion of strong scaling, and defines a theoretical limit to the benefits of such scaling. This limit can be increased by adopting algorithms that have smaller values of F_S.

Gustafson's Law

Gustafson's Law is an attempt to address specific shortcomings of Amdahl's Law [66]. Specifically, Amdahl's Law is based on the fundamental assumption that the problem size remains fixed as the number of processors is increased. Gustafson's Law instead considers that as available computing power increases larger problem sizes can be solved in the same fixed amount of time. As a result of having more computational power, the total amount of computation can be increased, easing the limitations posed by the serial part of the program. This law is formally stated below:

$$S_P = P - F_S \times (P - 1) \tag{2.4}$$

This idea is intertwined with the notion of *weak scaling*, where a fixed problem size per processor is used. This provides a different perspective to the strong scaling considered by Amdahl's Law. In an ideal embarrassingly parallel case, application runtime would remain fixed as additional processors are added. As a result, weak scaling studies allow HPC application developers to analyse the overhead of increasing the amount of processors used.

2.4.2 Benchmarking

Benchmarking allows users of a computer system to assess the relative performance of some component of the system, e.g., peak performance of the compute units. Historically, the peak performance of a CPU or supercomputer has been expressed in terms of Millions of Instructions per Second (MIPS) and FLOP/s. These values are often quoted by vendors to allow comparison between differing machines. However, these metrics merely provide the theoretical upper bound of how well a given system will perform during execution. This is because almost all real-world applications will fail to achieve full utilisation of the hardware at all times, resulting in suboptimal performance. As a consequence, metrics such as peak FLOP/s are seen as a poor measure of the actual performance

of a computer system in production [48]. It is this issue that comprehensive benchmarking seeks to address.

A *benchmark* can be thought of as a small code designed to capture the performance of a given subsystem of a computer, typically by stressing this subsystem in some methodical fashion. Generally, the benchmark will attempt to mimic the behaviour of real-world applications in order to generate meaningful performance data that can later be used for comparison to other configurations or systems. As a consequence, a wide variety of benchmarks have been developed by the scientific community which can be used as a collective to rank existing supercomputers. The TOP500 [142] and Graph500 [64] lists are the most well-known rankings. Some of the most common benchmarks in use to measure peak FLOP/s include Linear Algebra Package (LINPACK) [49], the linear algebra based benchmark that is used to determine the TOP500 rankings, and the High Performance Conjugate Gradients (HPCG) [48] benchmark. HPCG provides an alternative ranking of the TOP500 machines that is intended to complement LINPACK by being representative of codes that operate on sparse data structures, in contrast to the dense systems of equations handled by LINPACK. Other key benchmarks used to test computer subsystems include STREAM [101], which is used to test sustained data throughput from Random Access Memory (RAM), and the SKaMPI benchmark [128] and the Intel MPI benchmarks [78] that both stress the installed MPI software and, by extension, the communication fabric between individual compute nodes.

While specialised benchmarks allow engineers to analyse and draw conclusions regarding the performance of individual system components, it is often much more difficult to extrapolate this data to reason about a full application that will use all of the components in tandem. For this purpose we require less specialised *macro-benchmarks* that are more representative of real-world applications than the *micro-benchmarks* described above. Macro-benchmarks can be written to be broadly representative of a given class of application in such a way that they are easily ported to be tested on new systems and, as such, are often

shared with vendors. Additionally, it is common for HPC centres to develop and maintain their own benchmark suites. These suites typically contain a set of macro-benchmarks that are broadly representative of the type of workloads that will be run at the site, making them a key part of the procurement process. Examples of these include the NAS Parallel Benchmarks (NPB) [10] used to represent codes used by the NASA Advanced Supercomputing Division (NAS), the Rodinia benchmark suite for heterogeneous computing [37], and the National Energy Research Scientific Computing Center (NERSC) benchmark suite [7], used by the laboratory of the same name.

2.4.3 Application Profiling

While benchmarks provide a general idea of how a given application could perform on a specific hardware and software stack, they generally do not provide the fine-grained detail that is required to identify the underlying cause of a performance issue. For example, STREAM may report that system memory bandwidth is underutilised, but not why this is the case. For this reason, application engineers often use a *profiler* to conduct detailed analyses of code behaviour. Typically, a profiler will accomplish this by monitoring program execution and system performance counters in order to collect metrics that provide a 'profile'. This profile can then be analysed to discern the true source of suboptimal performance. Several profiler options exist today, some of which have a specialised focus (e.g. application I/O tracing [154], memory read and write monitoring [110], and MPI communication tracing [108]), and some of which are more general. Examples of general profilers include GNU `gprof` [63], Intel's VTune Amplifier tool [77], and NVIDIA's `nvprof` [114], which provide performance breakdowns at the kernel[1] level, alongside function call trees.

From the above, it is clear to see that profiling plays a key role in the process of performance engineering. Many of the performance tuning decisions presented

[1]Note that in this thesis *kernel* refers to a collection of program code grouped within the same subroutine or function, that performs a well-defined task, e.g., updating the pressure in a simulation.

in Chapter 4 are initially guided by profiling results, followed by hand inspection of code once problematic areas have been identified. This is especially useful for optimisations that exploit features of specific GPU hardware, and the Single Instruction Multiple Thread (SIMT) paradigm employed by CUDA warps.

2.4.4 Representative Applications

While benchmarking can provide a great deal of information to assist with tuning applications, full-scale scientific production codes are far more complex than any benchmark. Additionally, these applications can consist of upwards of one million lines of code, making for an extremely complex codebase. As a result, exploring new technologies and optimisations for such codes is an arduous undertaking which can take years to achieve. *Representative applications*, some times referred to as mini-applications (or mini-apps), are smaller codes written to capture the performance behaviour of their parent production code. Due to their more manageable size, mini-applications can be readily ported to new technologies, or otherwise rapidly altered in terms of optimisation, often by a single developer, or at most a small team. Therefore, mini-applications are a key part of the HPC software development life cycle and supercomputer procurement process in the sense that large commitments do not have to be made blindly.

There are a wide variety of mini-applications in the HPC space. The Mantevo suite [70], maintained by SNL, is one major set of mini-applications, and contains codes representative of Finite Element Method (FEM) based solvers, Computational Fluid Dynamics (CFD), Molecular Dynamics (MD), and magnetohydrodynamics schemes. Other notable mini-applications maintained by the UK Mini-App Consortium include CloverLeaf [99] (structured hydrodynamics), TeaLeaf [103] (structured heat conduction), and BookLeaf [96, 144] (unstructured hydrodynamics). The performance behaviour of all of these codes has been evaluated on multiple systems and, as such, is well understood by the community.

2.4.5 Performance Modelling

Performance modelling refers to the practice of attempting to predict performance data (most commonly runtime) from knowledge of the performance behaviour of a code, and the specifications of the hardware that is being considered. This performance prediction can then be used to facilitate procurement decisions, enabling HPC sites to gain a preliminary understanding of how their workloads may perform on a candidate system [68]. This can be especially useful for commercially sensitive work, where a model can be shared, but the parent code cannot [47]. Additionally, as a model provides an approximation of what performance a code *should* achieve, modelling can be used to expose and resolve performance issues, or evaluate potential code optimisations.

The techniques used in performance modelling can generally be separated into two categories – the first of these is *analytical modelling*. Analytical modelling seeks to express the performance behaviour of an application as a system of equations, where the terms of the equations represent the performance critical parts of the code, and the values of which are determined from initial empirical experiments. In the case of an MPI application, these terms are typically the time spent carrying out computation, and the time spent executing inter-process communications. The analytical approach has several advantages, the most obvious of which is that, given new parameters, the model can return new predictions extremely quickly. However, it also comes with the disadvantage that it can be challenging (or sometimes impossible) to accurately capture the performance behaviour of complex applications. Analytical models have previously been applied to wavefront codes [107], and CFD simulations [119], to name but two examples.

The second common modelling approach is *simulation*, where the behaviour of an application is simulated on a defined hardware setup. The scale of this simulation can vary greatly, from a completely simulated approach where all hardware is modelled, down to systems that merely simulate the communications between nodes, but continue to use the analytical approach described

above to handle computation. Simulation has a clear advantage over analytical modelling when inter-node communication accuracy is key, but this comes with the downside that detailed simulations take much longer to deliver results than an analytical model. There are many simulators available, including the Warwick Performance Prediction Toolkit (WARPP) [67], and the Structural Simulation Toolkit (SST) [131]. SST has previously been used by Bird et al. to model the performance of the Lagrangian-Eulerian code, Lare [22].

2.5 Benchmarking Platforms

The research presented in this thesis makes use of a variety of different machines and compute architectures. Additionally, we make a distinction between single-node systems, and supercomputers that are made up of many individual compute nodes. Therefore, this section presents a detailed summary of the different types of hardware used, and the various differences between them, both at the single-node and cluster level. Where possible, every attempt has been made to carry out all experiments on the same hardware in order to ensure consistency in performance results. When this was not achievable (e.g. due to restricted access to classified machines) this is made clear in the text. All experiments are repeated five times, with the lowest runtime being selected as the final result. As the code used in this thesis is deterministic, we can be sure that variation in timings is due to background noise on the node, e.g., operating system processes and other such services. Therefore, the taking the minimum is more appropriate than averages such as the mean or median which may be skewed by abnormally high values. Additionally, raw performance data for all graphs in Chapters 4, and 6 can be found in Appendices A and C, respectively.

2.5.1 Single Nodes

The majority of the experiments documented in this thesis were carried out on single-node systems. Note that in the case of the CPU systems, all nodes are

	Intel Xeon				
	E5-2698v3	**E5-2660v4**	**Gold 6252**	**Phi 7210**	**Phi 7250**
Cores	16	14	24	64	68
Clock Speed (GHz)	2.3	2.0	2.1	1.3	1.4
Peak GFLOP/s	294.4	224.0	806.4	1331.27	1523.2
Bandwidth (GB/s)	68.0	76.8	141.0	102.0	115.2
TDP (W)	135	105	150	215	215
Instruction Set	AVX2	AVX2	AVX-512	AVX-512	AVX-512
Micro-architecture	Haswell	Broadwell	Cascade Lake	KNL	KNL

Table 2.1: Hardware specifications of the Intel CPUs used in this thesis.

	Cavium ThunderX2		IBM POWER
	CN9975	**CN9980**	**POWER9 22c**
Cores	28	32	22
Clock Speed (GHz)	2.0	2.1	3.4
Peak GFLOP/s	224.0	268.8	598.4
Bandwidth (GB/s)	158.9	158.9	170.0
TDP (W)	180	180	190
Instruction Set	NEON	NEON	Power v3
Micro-architecture	Vulcan	Vulcan	POWER9

Table 2.2: Hardware specifications of the other CPUs used in this thesis.

dual-socket unless otherwise stated, meaning that each node contains two of the individual chips specified in the tables. Tables 2.1, and 2.2 detail the CPUs used in this thesis, and Table 2.3 shows the GPUs used. For each component, the peak computational performance for double-precision computation is reported in Giga-Floating-Point Operations per Second (GFLOP/s), and memory bandwidth as quoted by the manufacturer is given in Gigabytes per Second (GB/s). This data should be thought of as the theoretical peak performance achievable used to broadly compare systems – these values are almost never realised during a typical production experiment. The base clock rate is also provided in Megahertz (MHz) or Gigahertz (GHz) as appropriate, along with the power consumption in the form of Thermal Design Power (TDP) in Watts. It is important to keep all of these metrics in mind when comparing application performance between systems in order to ensure that fair comparisons are made, especially when the architectures differ greatly. The importance of making such fair comparisons has been highlighted by multiple previous authors acknowledging that the level of performance tuning of each application version and the appropriateness of the metrics compared is crucial [97, 150]. For example, comparing

	NVIDIA		
	K40c	**P100**	**V100**
Compute Units	15	56	80
FP64 Cores	960	1792	2560
Clock Speed (MHz)	745.0	1328.0	1312.0
Peak GFLOP/s	1680.0	5300.0	7800.0
Bandwidth (GB/s)	288.0	732.0	900.0
TDP (W)	235	300	300
Micro-architecture	Kepler	Pascal	Volta

Table 2.3: Hardware specifications of the NVIDIA GPUs used in this thesis.

	Trinity (Haswell)	**Trinity (KNL)**
Processor	Xeon E5-2698v3	Xeon Phi 7250
Cores/node	32	68
Nodes	9436	9984
Accelerators/node	None	None
Memory/node (GB)	128	96
Interconnect	Cray Aries	Cray Aries

Table 2.4: Hardware specifications of both partitions of Trinity.

an unoptimised CPU implementation to a fully-tuned GPU code would yield extremely biased results.

2.5.2 Supercomputers

In addition to the single-node systems described above, the work documented in Chapter 4 makes use of three different United States Department of Energy (DOE) supercomputers in order to conduct scaling studies. The first of these machines is Trinity, a Cray XC40 located at Los Alamos National Laboratory (LANL), New Mexico, USA. Trinity initially consisted of two separate partitions, with one partition made up of dual-socket Intel Xeon E5-2698v3 nodes, and the other made up of Intel Xeon Phi 7250 KNL nodes. As of June 2017, both partitions have been merged to create a single heterogeneous cluster. Full details of Trinity are shown in Table 2.4.

Secondly, we have Astra – a petascale ARM-based HPE Apollo 70 installed at and operated by SNL, New Mexico, USA, and the first ARM system to make it into the TOP500 listings. Astra is a homogenous cluster made up of dual-socket Cavium ThunderX2 CN9975 nodes. Full details of this system can be found in Table 2.5.

	Astra	**Sierra**
Processor	TX2 CN9975	POWER9 22c
Cores/node	56	44
Nodes	2592	4340
Accelerators/node	None	4× NVIDIA V100
Memory/node (GB)	128	256
Interconnect	EDR Infiniband	EDR Infiniband

Table 2.5: Hardware specifications of Astra and Sierra.

Finally, we come to Sierra, an IBM Power System AC922 located at LLNL, California, USA. Sierra is a heterogeneous supercomputer consisting of dual-socket IBM POWER9 22c CPUs, with each node also containing four NVIDIA V100 GPU accelerator cards. Additionally, the system has NVLink capabilities, allowing separate GPUs on the same node direct read/write access to each other's device memory. As with Astra, comprehensive hardware specifications for Sierra can be found in Table 2.5.

2.6 Summary

This chapter has detailed the various classes of parallel hardware that exist today, and the types of parallelism that can be used to fully exploit them. We have also introduced the concept of performance engineering, and highlighted various theoretical and practical concepts that can be used to measure, analyse, and improve HPC application performance. Particular focus has been placed on emerging many-core architectures and heterogeneous computing, and how these are driving the need for applications to be portable across multiple modern hardware types, and how such portability could be achieved. Much of this background knowledge is fundamental to the performance related work presented in this thesis. Finally, we have presented a summary of the various different hardware components and supercomputers used to carry out the experiments presented in Chapters 4 and 6.

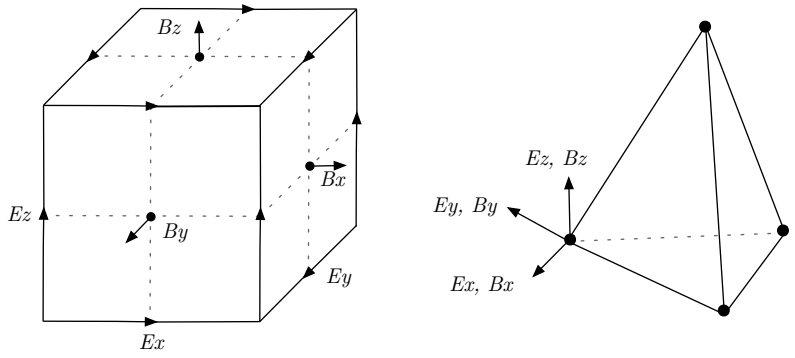
CHAPTER 3

Particle-in-Cell Simulations

The work presented in this thesis focuses on the unstructured EMPIRE-PIC Particle-in-Cell (PIC) application developed at Sandia National Laboratories (SNL). While this thesis is primarily concerned with the development of the particle-based kernels of the code, and related algorithmic extensions, this chapter also contains a discussion of Maxwell's equations, and the formulation of the relevant field solvers in the interest of completeness. Therefore, this chapter provides an overview of the unstructured Finite Element Method (FEM) PIC method, beginning with the motivations behind using unstructured grids for PIC simulations, then describing underlying physical equations, before moving on to present the formulation of each step of the algorithm implemented in EMPIRE-PIC in detail.

3.1 Motivation

High Performance Computing (HPC) can be applied within a variety of scientific fields, with the areas of fusion energy research, and the behaviour of plasmas under various conditions being notable examples. PIC codes are commonly used to carry out simulations of charged particles in plasmas under the influence of electric and magnetic fields [23, 42, 73]. Examples of applications in fusion energy research include the simulation of both Inertial Confinement Fusion (ICF) and Magnetic Confinement Fusion (MCF) devices. Such devices include the National Ignition Facility (NIF), located at Lawrence Livermore National Laboratory (LLNL), and the Iternational Thermonuclear Experimental Reactor (ITER) located in France, which each attempt ICF and MCF respectively. Other applications include the behaviour of magnetrons in microwave



(a) Staggered Yee grid. (b) Tetrahedral finite element.

Figure 3.1: Comparison of grid representation in structured versus unstructured PIC. The finite element shown here assumes the fields are located at the element nodes.

generation systems, charged particle beams, laser-plasma interaction [8], astrophysical plasmas [129], and applications in biomedicine [59].

Traditionally, PIC employs a structured grid approach to represent the space being simulated, storing the field values on cell edges and faces [155], modelling particles as discrete objects in the problem space. The algorithm is commonly implemented using a Finite Difference Time Domain (FDTD) scheme with the fields represented on a staggered Yee grid (Figure 3.1(a)), with discrete particles present within the domain [43, 94, 155]. Notable examples of such PIC codes include the Extendable PIC Open Collaboration (EPOCH) [8], OSIRIS [57], ICEPIC [26], the Plasma Simulation Code (PSC) [62] and VPIC [30, 31]. Gyrokinetic PIC algorithms have also been applied to the challenge of kinetic plasma simulation in five-dimensional phase space. Two such codes are GTC-P, the Gyrokinetic Toroidal Code [152], and XGC, the X-point Gyrokinetic Code, developed at Princeton University [91].

These codes are varied in their features and implementations but, with the exception of GTC-P and XGC, each operates on a traditional structured rectilinear grid. The application of such meshes to problems with high fidelity geometries is challenging due to the extreme resolution that is needed to accurately represent the simulation space. One approach to resolve this is through

the use of Adaptive Mesh Refinement (AMR) to refine the problem only in areas of interest, reducing the total number of cells required for simpler sections of the geometry. The use of AMR-PIC has previously been explored for both electrostatic and electromagnetic problems by Vay et al. in WARP [147] and Warp-X [146], respectively.

Alternatively, a solution to the problem of representing complex geometry is to use an unstructured computational mesh with finite elements of arbitrary shapes and sizes. Figure 3.1(b) shows an example of one such element type. Like AMR, this provides the flexibility of refining the problem in areas of key interest, but without the restriction that the grid cells themselves retain their structured properties. Unstructured PIC algorithms have previously been explored by multiple authors, with work by Squire et al. [138] and Moon et al. [105] being but two examples. Instances of PIC codes implementing unstructured schemes include PTetra [100] and the open-source Spacecraft Plasma Interaction Software (SPIS) [132].

3.2 Physical Equations

PIC algorithms are commonly used for the simulation of plasma dynamics. This is accomplished by numerically solving the Vlasov equation that governs the time evolution of a given collisionless plasma [149]. A collection of N_p discrete particles is used to represent the plasma, each of which has an associated position $\vec{x}$, velocity $\vec{v}$, charge q, and mass m. These are used to approximate a probability distribution, f, as a series of delta functions, δ, across all particles i.

$$f = \sum_{i=1}^{N_p} f_i = \sum_{i=1}^{N_p} \delta(\vec{x} - \vec{x}_i)\delta(\vec{v} - \vec{v}_i) \tag{3.1}$$

The probability distribution for each particle is updated via Newton's Law and the Lorentz force equation, where $\vec{B}$ and $\vec{E}$ are the magnetic and electric fields,

respectively. These can be expressed as the updates shown in (3.2) and (3.3).

$$\frac{d\vec{x}_i}{dt} = \vec{v}_i \tag{3.2}$$

$$\frac{d\vec{v}_i}{dt} = \frac{q_i}{m_i}\left(\vec{E}(\vec{x}_i) + \vec{v}_i \times \vec{B}(\vec{x}_i)\right) \tag{3.3}$$

These equations can then be assembled into the Vlasov equation, which is related to the Klimontovich equation for particle dynamics [50, 89, 112]:

$$\frac{\partial f_i}{\partial t} + \frac{\vec{v}_i}{m} \cdot \nabla f_i + \frac{q_i}{m_i}\left(\vec{E}(\vec{x}_i) + \vec{v}_i \times \vec{B}(\vec{x}_i)\right)\frac{\partial f_i}{\partial \vec{v}} = 0 \tag{3.4}$$

With given magnetic and electric fields, this system can be used to fully describe the particle evolution, and Maxwell's equations can used to couple the charged particles to the electric and magnetic fields. They consist of the following: Gauss' Law, the magnetic divergence constraint, Faraday's Law, and Ampère's Law. For clarity, these equations are given below in the form of differential equations. Here ρ and $\vec{J}$ are the charge and current densities, and ϵ_0 and μ_0 are the permittivity and permeability of free space, respectively.

$$\nabla \cdot \vec{E} = \frac{\rho}{\epsilon_0} \tag{3.5}$$

$$\nabla \cdot \vec{B} = 0 \tag{3.6}$$

$$\frac{\partial \vec{B}}{\partial t} = -\nabla \times \vec{E} \tag{3.7}$$

$$\frac{\partial \vec{E}}{\partial t} = \frac{1}{\mu_0 \epsilon_0}\nabla \times \vec{B} - \frac{1}{\epsilon_0}\vec{J} \tag{3.8}$$

Finally, the particles can be coupled back to Maxwell's equations via the charge and current densities defined below.

$$\rho = \sum_{i=1}^{N_p} q_i f_i \tag{3.9}$$

$$\vec{J} = \sum_{i=1}^{N_p} q_i \vec{v}_i f_i \tag{3.10}$$

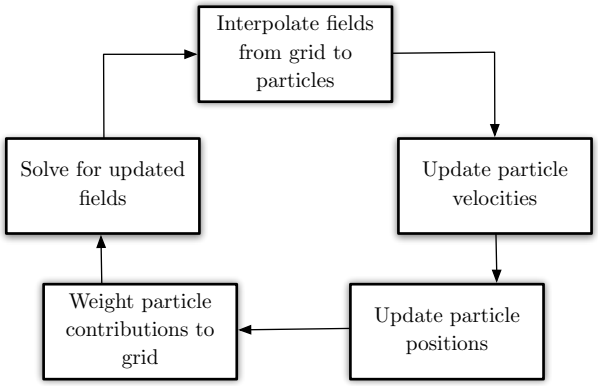


Figure 3.2: The main time loop of a Particle-in-Cell simulation.

3.3 The Particle-in-Cell Method

Introduced by Birdsall and Dawson, the PIC method is a well established procedure for modelling the behaviour of charged particles in the presence of electric and magnetic fields [24, 43]. Discrete particles are tracked in a Lagrangian frame, while the electric and magnetic fields are stored on stationary points on a fixed Eulerian mesh. Therefore, the algorithm can be thought of as two coupled solvers where one is responsible for updating the electric and magnetic fields, and another updates the particles via the method of characteristics and calculates their charge/current contributions back to the grid. These are referred to as the field solver and the particle mover (sometimes called the particle pusher), respectively. Combining these solvers results in the main time loop of the core PIC algorithm that consists of several key steps, summarised in Figure 3.2. In short this consists of: solving for the field values on the computational mesh, weighting these values to determine the fields at particle locations, updating the particle velocities and positions, and carrying out the particle charge/current deposits to grid points. The number of physical particles (defined at the level of atoms and/or electrons) required to simulate a modern plasma system is exceedingly large. Thus, so-called super-particles are employed in order to make simulation via computation feasible. These can be thought of as 'computational particles' that reflect the behaviour of a collection of physical particles.

For example, one super-particle may represent many billions of electrons within a plasma. This allows the number of computational particles within the system to be much lower, therefore reducing the workload required. It should be noted that as the Lorentz force is only related to the charge-to-mass ratio of particles, these super-particles will exhibit the same movement as their physical counterparts. However, they will affect the fields by an amount proportional to the chosen weight.

3.3.1 Solving Maxwell's Equations

In order to obtain the values of the electric and magnetic fields it is necessary to solve Maxwell's equations. Maxwell's equations hold true for all cases, but approximations can be made if certain conditions are satisfied. Specifically, if the movement of the plasma particles is slow in comparison to the speed of light, c, the equations can be reduced such that only Gauss' Law (3.5) must be solved, and the magnetic divergence constraint (3.6) is implicitly maintained. This is known as the *electrostatic* approximation, i.e., where the electric field is irrotational: $\nabla \times \vec{E} = 0$. However if the particles move at relativistic velocities, or the current density is large, then the electrostatic approximation is no longer valid. In such cases we must solve the full set of Maxwell's equations in order to account for the changing magnetic field. We refer to this class of problems as *electromagnetic*.

We now show in detail the formulation of both electrostatic and electromagnetic problems, such that they can be solved via the FEM [83]. Once the problem has been formulated correctly, it can be solved using a variety of iterative or direct methods.

Electrostatic Field Solver

When conducting an electrostatic simulation we need only solve (3.5) during the field solve. In this case, the electric field can be represented as a gradient of

electric potential, ϕ, as in (3.11), where $\vec{E}$ is defined as specified in (3.12).

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0} \tag{3.11}$$

$$\vec{E} = -\nabla \phi \tag{3.12}$$

In order to solve this system via FEM we must first expand the equation into a basis function, multiply by a test function, and subsequently integrate. Here we are using a Galerkin method, i.e., the basis functions and test functions are the same. In our electrostatic case, we are using basis functions $\hat{v} \in H_{Grad}$, sometimes known as the P_1 basis elements, or the space of linear basis functions. By putting the expression into the weak form and using a test function $\hat{v}_i$ we obtain the following:

$$\int_0^L \left[\frac{d^2\phi}{dx^2} + \frac{\rho}{\epsilon_0} \right] \hat{v}_i \; dx = 0 \tag{3.13}$$

Expanding the electric potential into a finite-dimensional basis function and substituting into the weak form expression, where N is equal to the number of bases, yields:

$$\phi \approx \sum_{j=0}^{N} \phi_j \hat{v}_j \tag{3.14}$$

$$\int_0^L \left[\frac{d^2 \sum_{j=0}^{N} \phi_j \hat{v}_j}{dx^2} + \frac{\rho}{\epsilon_0} \right] \hat{v}_i \; dx = 0 \tag{3.15}$$

As ϕ_j does not depend on x the summation can be moved outside of the integral, which can then be separated into two parts. However, leaving the equation in this form would require us to compute the second derivative of the basis function – a process which is often intractable. In order to resolve this issue, we apply integration by parts to move one of the derivatives from the basis function to the test function instead.

$$\sum_{j=0}^{N} \phi_j \left[\int_0^L -\frac{d\hat{v}_j}{dx} \frac{d\hat{v}_i}{dx} \; dx + \frac{d\hat{v}_j}{dx} \hat{v}_i |_0^L \right] + \int_0^L \frac{\rho}{\epsilon_0} \hat{v}_i \; dx = 0 \tag{3.16}$$

As we have chosen P_1 elements (the space of linear basis functions), we can represent any piecewise linear function. Therefore we can define a sparse tridiagonal stiffness matrix, S, which allows us to rewrite (3.16).

$$S_{i,j} = \int_0^L -\frac{d\hat{v}_j}{dx}\frac{d\hat{v}_i}{dx}\, dx \tag{3.17}$$

$$\sum_{j=0}^{N} \phi_j \left[S_{i,j} + \frac{d\hat{v}_j}{dx}\hat{v}_i|_0^L \right] = -\int_0^L \frac{\rho}{\epsilon_0}\hat{v}_i\, dx \tag{3.18}$$

In order to handle the right hand side of the above expression we assume that the charge density, ρ originates from k discrete particles at some location $\vec{x}_k$. This allows us to represent the equation as a summation over the particles as shown in (3.19). For the P_1 basis elements this results in a linear interpolation to the nodes.

$$\int_0^L \frac{\rho}{\epsilon_0}\hat{v}_i\, dx = \sum_{k=1}^{N_P}\int_0^L \frac{q_k\delta(\vec{x}_k)}{\epsilon_0}\hat{v}_i\, dx = \sum_{k=1}^{N_P}\frac{q_k\hat{v}_i(\vec{x}_k)}{\epsilon_0} \tag{3.19}$$

As the term $\frac{d\hat{v}_j}{dx}\hat{v}_i|_0^L$ in (3.18) is only nonzero for $i = j \in \{0, N\}$, and can often be ignored through the use of alternative boundary condition handling techniques, (3.18) results in a sparse matrix equation which can be solved through a variety of iterative or direct approaches when combined with (3.19). Once we have computed the values for ϕ, we can easily determine the electric field.

$$\vec{E} \approx -\sum_{j=0}^{N} \phi_j \nabla\hat{v}_j \tag{3.20}$$

It should be noted that extending this one-dimensional example to multidimensional problems is trivial, however the resultant stiffness matrix is no longer tridiagonal, but will remain sparse. This extension is shown in (3.21), where Ω represents some finite domain.

$$S_{i,j} = \int_\Omega -\nabla\hat{v}_j \cdot \nabla\hat{v}_i\, d\Omega \tag{3.21}$$

It is relatively simple to implement a linear field solver for the electrostatic formulation. To achieve this in EMPIRE-PIC the traditional Krylov iterative methods provided by the Belos Trilinos package are used [14]. The MueLu Trilinos package is used in conjunction with Belos to provide an Algebraic Multigrid (AMG) preconditioner [16].

Electromagnetic Field Solver

When solving Maxwell's equations for electromagnetic simulations, we use similar techniques to those employed for electrostatic problems. However, we use different basis functions to expand both the electric and magnetic fields. Specifically, we make use of the lowest order Nédélec [109] edge elements $\hat{e} \in H_{Curl}$ and Raviart-Thomas [32] face elements $\hat{b} \in H_{Div}$, respectively. In addition, we must also consider the temporal term. We begin our electromagnetic FEM formulation with the weak form of Faraday's Law, shown in (3.22). This works because the chosen Nédélec elements are a compatible discretisation, i.e., the curl of the H_{Curl} basis is within the divergence space H_{Div}. Therefore, the $\nabla \times \hat{e}_j$ maps directly into the space of $\vec{B}$. The expansion is shown in (3.22) where N_{face} and N_{edge} are the number of element faces and edges, respectively.

$$\sum_{j=0}^{N_{face}} \frac{\partial B_j}{\partial t} \int_\Omega \hat{b}_i \cdot \hat{b}_j \; d\Omega = -\sum_{j=0}^{N_{edge}} E_j \int_\Omega \nabla \times \hat{e}_j \cdot \hat{b}_i \; d\Omega \tag{3.22}$$

Similarly to the electrostatic equations we form and substitute a curl matrix, K_B. Furthermore, we also define a new term, M_B, that represents a mass matrix. We can then use these terms to rewrite the weak form of Faraday's Law (3.22).

$$M_B = \int_\Omega \hat{b}_i \cdot \hat{b}_j \; d\Omega \tag{3.23}$$

$$K_B = \int_\Omega \nabla \times \hat{e}_j \cdot \hat{b}_i \; d\Omega \tag{3.24}$$

$$M_B \frac{\partial B}{\partial t} = -K_B E \tag{3.25}$$

We deal with the formulation of Ampère's Law, (3.8), in a similar fashion, beginning by expressing it in the weak form, and applying integration by parts as shown in (3.26) and (3.27). This allows us to move the curl operator from $\hat{b}_j$ to $\hat{e}_i$ in order to avoid the issue of the curl of the H_{Div} space being not well defined causing poor definition of $\nabla \times \hat{b}_j$.

$$\begin{aligned} &\sum_{j=0}^{N_{edge}} \frac{\partial E_j}{\partial t} \int_\Omega \hat{e}_i \cdot \hat{e}_j \; d\Omega \\ &\quad = -\frac{1}{\mu_0 \epsilon_0} \sum_{j=0}^{N_{face}} B_j \int_\Omega \nabla \times \hat{b}_j \cdot \hat{e}_i \; d\Omega - \frac{1}{\epsilon_0} \int_\Omega \vec{J} \cdot \hat{e}_i \; d\Omega \end{aligned} \tag{3.26}$$

$$\begin{aligned} &\sum_{j=0}^{N_{edge}} \frac{\partial E_j}{\partial t} \int_\Omega \hat{e}_i \cdot \hat{e}_j \; d\Omega \\ &\quad = -\frac{1}{\mu_0 \epsilon_0} \sum_{j=0}^{N_{face}} B_j \int_\Omega \hat{b}_j \cdot \nabla \times \hat{e}_i \; d\Omega - \frac{1}{\epsilon_0} \int_\Omega \vec{J} \cdot \hat{e}_i \; d\Omega \end{aligned} \tag{3.27}$$

Similarly to Faraday's Law, we can now form the corresponding mass matrix, M_E, and curl matrix, K_E, for Ampère's Law. However, these matrices may not necessarily be square in all cases.

$$M_E = \int_\Omega \hat{e}_i \cdot \hat{e}_j \; d\Omega \tag{3.28}$$

$$K_E = -\frac{1}{\mu_0 \epsilon_0} \int_\Omega \hat{b}_j \cdot \nabla \times \hat{e}_i \; d\Omega \tag{3.29}$$

We must also now consider the time derivative terms for both Faraday's and Ampère's Laws. Common approaches include backward Euler, Crank-Nicolson (C-N), and Friedman [60] time integration schemes. Here we consider the C-N scheme that is unconditionally stable, energy conservative, and second-order accurate. Here n is the discrete time level, we approximate the time derivative via a one-sided difference, and the expression is evaluated at both the new and current time. It should be noted that while C-N offers second order convergence

and exact energy conservation, it is not always an ideal choice of integrator. The scheme is non-dissipative, so high frequency modes once perturbed will persist throughout the simulation. This, combined with the stochastic nature of particle simulations, can result in particle noise coupling to the fields, creating undesirable feedback [65]. In such situations a Friedman integrator can be used to damp out any high frequency noise. For simplicity and ease of presentation, the C-N formulation for first-order Maxwell's equations is presented here. We can now show the following:

$$M_E E^{n+1} - \frac{\Delta t}{2} K_E B^{n+1} = M_E E^n - \frac{\Delta t}{2} K_E B^n - \frac{\Delta t}{\epsilon_0} \int_\Omega \vec{J} \cdot \hat{e}_i \; d\Omega \tag{3.30}$$

$$M_B B^{n+1} + \frac{\Delta t}{2} K_B E^{n+1} = M_B B^n - \frac{\Delta t}{2} K_B E^n \tag{3.31}$$

$$\begin{bmatrix} M_B & \frac{\Delta t}{2} K_B \\ -\frac{\Delta t}{2} K_E & M_E \end{bmatrix} \begin{bmatrix} B^{n+1} \\ E^{n+1} \end{bmatrix} = \begin{bmatrix} M_B B^n - \frac{\Delta t}{2} K_B E^n \\ M_E E^n - \frac{\Delta t}{2} K_E B^n - \frac{\Delta t}{\epsilon_0} J^{n+1/2} \end{bmatrix} \tag{3.32}$$

Here, we express the current as $J^{n+\frac{1}{2}}$, a vector form of $\frac{1}{\epsilon_0} \int_\Omega \vec{J} \cdot \hat{e}_i d\Omega$. If we assume that charge is represented by a point delta function with some charge q, we can define the current as shown in (3.33).

$$\begin{aligned} \frac{1}{\epsilon_0} \int_\Omega \vec{J} \cdot \hat{e}_i \; d\Omega &= \frac{1}{\epsilon_0} \int_{n\Delta t}^{(n+1)\Delta t} \int_\Omega q\vec{u}(t) \delta(\vec{x}(t)) \cdot \hat{e}_i d\Omega \; dt \\ &= \frac{1}{\epsilon_0} \int_{n\Delta t}^{(n+1)\Delta t} q\vec{u}(t) \cdot \hat{e}_i(\vec{x}(t)) \; dt \end{aligned} \tag{3.33}$$

The electromagnetic formulation requires a more specialised solver, detailed here. We know that the system has the following form, where R_B and R_E represent the right-hand side of the system of equations to be solved.

$$\underbrace{\begin{bmatrix} M_B & \frac{\Delta t}{2} K_B \\ -\frac{\Delta t}{2} K_E & M_E \end{bmatrix}}_{A} \begin{bmatrix} B^{n+1} \\ E^{n+1} \end{bmatrix} = \begin{bmatrix} R_B \\ R_E \end{bmatrix} \tag{3.34}$$

The above system can then be altered by a block LU decomposition,

$$A = \begin{bmatrix} I & 0 \\ -\frac{\Delta t}{2} K_E M_B^{-1} & I \end{bmatrix} \begin{bmatrix} M_B & \frac{\Delta t}{2} K_B \\ 0 & S_E \end{bmatrix} \tag{3.35}$$

where the electric field Schur complement, S_E, is given by:

$$S_E = M_E + \frac{\Delta t^2}{4} K_E M_B^{-1} K_B \tag{3.36}$$

This formulation can then be solved by a preconditioned conjugate gradient method. In this case, EMPIRE-PIC uses the 'refMaxwell' AMG preconditioner proposed by Bochev et al. [27].

3.3.2 Weighting of Fields to Particles

After solving for the field values at the nodes of the simulation grid, we require a method of determining the value of the fields at any specific particle position. In order to do so we weight the field values from the grid nodes to the required location. Assuming that we know the values of the electric and magnetic fields we can evaluate the value of the fields at a given point as shown below by applying the basis function. Here N_{edge} and N_{face} refer only to the number of edges and faces for the *containing* element, respectively.

$$\vec{E}(\vec{x}_i) = \sum_{j=0}^{N_{edge}} E_j \hat{e}_j(\vec{x}_i) \tag{3.37}$$

$$\vec{B}(\vec{x}_i) = \sum_{j=0}^{N_{face}} B_j \hat{b}_j(\vec{x}_i) \tag{3.38}$$

However, using the raw edge values of the electric field produces a large 'self-force' on the particle being pushed, as edge fields conserve energy whereas nodal fields conserve momentum [74]. The standard structured PIC algorithm uses special averaging of the edge fields to the nodes in order to generate the fields that are used to push the particles. However, in our unstructured algorithm, we

mimic the approach used in the structured method by using projections from the edge based electric fields to create fields that are based at the grid nodes. These projections have the form:

$$\begin{aligned}&\int_{\Omega}\left(\vec{E}_{nodal}-\vec{E}_{edge}\right)\hat{v}\ d\Omega \\ &\quad=\sum_{j=1}^{N_n}\sum_{i=1}^{N_n}E_{nodal,i}\int_{\Omega}\hat{v}_i\hat{v}_j\ d\Omega-\sum_{j=1}^{N_n}\sum_{i=1}^{N_e}E_{edge,i}\int_{\Omega}\hat{e}_i\hat{v}_v\ d\Omega=0\end{aligned} \tag{3.39}$$

This can then be reduced to a solve of a mass matrix equation $M_{nodal}E_{nodal} = M_{nodal,edge}E_{edge}$. One possible technique is to 'lump' the mass matrices, i.e., convert them into diagonal matrices with dual area on the diagonal. This leads to a volume-based field averaging which, when used on a uniform mesh, is identical to the method used within structured PIC simulations.

3.3.3 Particle Mover

The force felt by a charged particle due to the presence of electric and magnetic fields is described by the Lorentz force equation, shown in (3.40). The particle mover within a PIC code is responsible for solving for this force on each particle within the simulation, and subsequently updating the particle velocities and positions. In our method we make use of the well known Boris algorithm to handle the acceleration due to the electric field, and rotation about the magnetic field [29].

$$\vec{F}=q\left(\vec{E}+\vec{v}\times\vec{B}\right) \tag{3.40}$$

To make the algorithm relativistic, here we define $\vec{u}=\gamma\vec{v}$, where γ is the Lorentz factor. Therefore we are solving the for the new velocity defined as:

$$\frac{\vec{u}^{\,n+1/2}-\vec{u}^{\,n-1/2}}{\Delta t}=\frac{q}{m}\left[\vec{E}^{\,n}+\frac{1}{c}\frac{\vec{u}^{\,n-1/2}+\vec{u}^{\,n+1/2}}{2\gamma^n}\times\vec{B}^{\,n}\right] \tag{3.41}$$

The Boris method eliminates the electric field from the magnetics rotation via the following substitutions:

$$\vec{u}^{\,n-1/2} = \vec{u}^{\,-} - \frac{q\vec{E}^{\,n}}{m}\frac{\Delta t}{2} \tag{3.42}$$

$$\vec{u}^{\,n+1/2} = \vec{u}^{\,+} + \frac{q\vec{E}^{\,n}}{m}\frac{\Delta t}{2} \tag{3.43}$$

$$\frac{\vec{u}^{\,+} - \vec{u}^{\,-}}{\Delta t} = \frac{q}{2\gamma^n mc}\left(\vec{u}^{\,+} + \vec{u}^{\,-}\right) \times \vec{B}^{\,n} \tag{3.44}$$

Next we derive the expression for performing the rotation about the magnetic field. First we find the vector bisecting the angle formed between the velocity before and after the rotation. In a given time-step, the velocity will rotate through the following angle: $\tan(\theta/2) = -(q\vec{B})\Delta t/2\gamma^n mc$. Rewriting this as a vector we obtain: $\vec{t} \equiv -\hat{b}\tan\theta/2$. We can then define $\vec{u}\,'$ as shown below:

$$\vec{u}\,' = \vec{u}^{\,-} + \vec{u}^{\,-} \times \vec{t} \tag{3.45}$$

$$\vec{u}^{\,+} = \vec{u}^{\,-} + \vec{u}\,' \times \vec{s} \tag{3.46}$$

$$\vec{s} = \frac{2\vec{t}}{1 + \vec{t}^{\,2}} \tag{3.47}$$

Once we have calculated $\vec{u}^{\,+}$, we can obtain the new particle velocity by adding an additional half of the acceleration as per (3.42)-(3.44).

3.3.4 Weighting of Particles to Grid

During each time-step in the PIC algorithm we must weight the contributions of each particle back onto the grid before we can commence the next field solve, though this contribution changes depending on whether the simulation is electrostatic or electromagnetic. For electrostatics, we apply (3.19) at the end of the particle move at the newly updated particle position. An electromagnetic simulation requires us to evaluate the the current (3.33) as shown in (3.48). For simplex elements, it is sufficient to use a midpoint rule for the integration, however, for higher-order elements we must evaluate this temporal integration

with higher-order numerical cubature. Specifically, EMPIRE-PIC uses two-point Gaussian quadrature with points at $\left(1 \pm 1/\sqrt{3}\right)/2$, each with a weight of $1/2$ when using non-simplex elements. This reduces to the charge conservation scheme for regular hexahedral elements previously presented by Villasenor and Buneman [148].

$$\begin{aligned}\int_{\Omega_j} \vec{J}\hat{e}_i \; dV &= \sum_{k=1}^{N_P} \int_{\Omega j} \frac{1}{\Delta t} \int_{n\Delta t}^{(n+1)\Delta t} q_k \vec{u}_k \cdot \hat{e}_i \; dt \; dV \\ &= \sum_{k=1}^{N_P} \Delta t q_k \vec{u}_k \left(\vec{x}_k^{\,n+1/2}\right) \cdot \hat{e}_i \left(\vec{x}_k^{\,n+1/2}\right)\end{aligned} \tag{3.48}$$

3.4 Higher-Order Methods

In addition to unstructured or adaptively refined meshes, many domain scientists have also experimented with the use of higher-order methods that make use of finite elements that possess additional degrees of freedom. Figure 3.3 shows a comparison of first-order and second-order tetrahedral finite elements. While these methods have previously been seen as prohibitively computationally intensive, the extreme levels of parallelism offered by modern supercomputers is causing a revival of such methods – the BLAST Arbitrary Langrangian-Eulerian (ALE) code developed at LLNL is one notable example [5, 6]. This resurgence is due to the increased arithmetic intensity of these methods improving the amount of Floating-Point Operations per Second (FLOP/s) performed per byte moved from Random Access Memory (RAM), providing an advantage in situations where limited memory capacity and bandwidth poses an obstacle to performance. The additional computational cost is also accompanied with improved simulation accuracy and convergence. Such methods have the benefit of enabling the use of coarser computational grids and reduced simulation constraints, while still reaching an acceptable solution due to the increased accuracy that they can provide. However, higher-order methods also require smooth source terms to integrate. In higher-order PIC this means that smooth particle shape functions are required to achieve the desired higher-order conver-

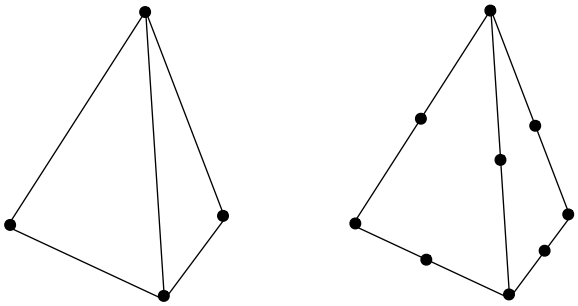


Figure 3.3: Comparison of first-order (left) and second-order (right) tetrahedral elements.

gence. A smoother particle representation also results in a better sampling of the surrounding fields when interpolating the fields to the particles, and reduced aliasing when depositing charge/current as particles move between cells.

Structured PIC codes generally implement higher-order methods by using smooth particle shapes extending over multiple cells [8], combined with higher-order field solvers. One example of such a particle shape can be achieved by implementing the Cloud-in-Cell (CIC) representation proposed by Birdsall and Fuss [24]. Unfortunately, smooth particle shapes are non-trivial to implement in practice for unstructured PIC codes as evaluating a higher-order basis often becomes intractable when spanning multiple elements.

Jacobs and Hesthaven present a discontinuous Galerkin PIC method that incorporates both higher-order time domain solution of Maxwell's equations and smooth particle shapes [71, 82]. The algorithm is implemented for unstructured grids with the problem space being discretised into non-overlapping, triangular finite elements. The particles are treated as fixed size clouds, and multiple smooth shape functions are considered. Through testing on a set of benchmark problems, the authors demonstrate the ability to simulate plasma phenomena with geometric flexibility while exhibiting reduced solution noise.

Essex and Bridson also show a higher-order PIC algorithm, HOPIC, that extends the PIC method to fourth-order accuracy for transport problems [53]. Specifically, the authors compare first- and third-order B-spline interpolation functions to a reference Cell Mean Value (CMV) approach which distributes

the charge of all particles in a cell equally to all cell nodes. The scheme is implemented at fourth order, with explicit Runge-Kutta methods (namely RK4) being used to achieve this. Weighting from the grid to the particles is achieved via a smooth fourth order interpolation method, while particles are weighted to the grid using moving least squares (MLS) and a basis of cubic polynomials. The implementation was tested on both two- and three-dimensional test problems, and was shown to exhibit the theoretical fourth order convergence.

Stindl et al. have also investigated higher-order methods within an electromagnetic discontinuous Galerkin PIC code, with a particular focus on the coupling of the particles and the unstructured grid [139]. Specifically, the authors compare first and third order B-spline interpolation functions to a reference Cell Mean Value (CMV) approach which distributes the charge of all particles in a cell equally to all cell nodes. It was found that the higher-order interpolation methods provide improved result accuracy against a refined solution, and also reduced runtime due to the decrease in the amount of computational particles required to carry out the simulation. It is finally suggested that such coupling methods are particularly appropriate for the simulation of complex engineering problems in order to achieve acceptable results.

3.5 Stability and Accuracy of PIC Simulations

As EMPIRE-PIC uses an implicit Crank-Nicolson method to handle time integration, the FEM-PIC algorithm implemented in the code is unconditionally stable. However, initial problem conditions must be carefully selected in order to ensure the accuracy of a given simulation.

In our case, particular care must be taken when selecting the grid spacing, Δx, and the time-step size, Δt to guarantee the accuracy of any experiments performed. In the case of distorted unstructured grids, Δx is often taken to be the average cell size of the computational mesh. In cases where the degree of grid distortion is large, the minimum cell size is sometimes preferred. While

smaller values of these parameters increase accuracy, the computational cost in terms of time and/or compute resources required also grows, meaning that the available equipment limits the fidelity of PIC simulations carried out by domain scientists.

The first important accuracy condition of the PIC method relates Δx to the Debye length [38] of the plasma, λ_D, which quantifies the the electrostatic effect of a charge carrier in a solution and how far this effect persists. λ_D is shown in Equation (3.49), where k_B is the Boltzmann constant, T_e is the temperature of the electrons in the plasma, n_e is the number density of the electrons, and q_e is the charge of an electron. The resultant condition is shown in (3.50).

$$\lambda_D = \sqrt{\frac{\epsilon_0 k_B T_e}{n_e q_e^2}} \tag{3.49}$$

$$\lambda_D \geq \Delta x \tag{3.50}$$

When the Debye length is under-resolved the Kinetic Energy (KE) of the simulation can increase erroneously until T_e reaches a value such that λ_D becomes resolved. This process is known as *numerical heating*, and is explored experimentally in Section 5.3.3.

Additionally, the mesh spacing and time-step size in PIC are related and constrained by the Courant-Friedrichs-Lewy (CFL) condition, which is a necessary condition for the convergence of solvers that handle certain systems of partial differential equations. The formal statement of the CFL condition in one dimension is given below for velocity u, and C_{max} is the maximum allowable CFL for a given solver. In the case of an explicit solver C_{max} is typically equal to one.

$$\text{CFL} = \frac{u\Delta t}{\Delta x} \leq C_{max} \tag{3.51}$$

Note that the CFL condition is treated differently for electrostatic and electromagnetic simulations. Specifically, for electrostatics u is taken to be the velocity of the fastest particles in the simulation, whereas for electromagnetics the speed

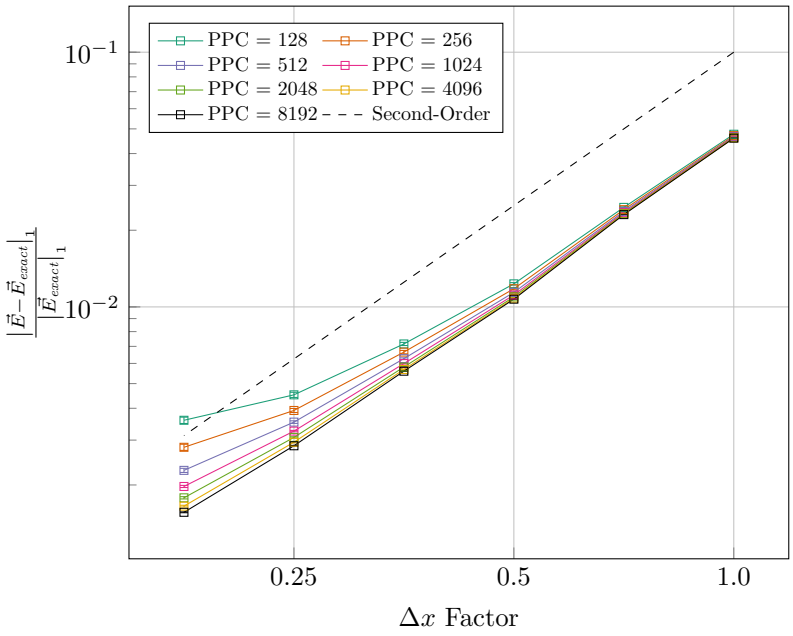


Figure 3.4: Graph showing how electric field error behaves for various PPC values as a simple electromagnetic problem is refined.

of light, c, is used due to the more complex nature of the simulation. As a consequence, this places much stricter upper bounds on the values of Δx and Δt, meaning that electromagnetics requires higher spatial and temporal resolutions.

Finally, we must consider how the amount of computational particles used affects the accuracy of the simulations conducted. We are generally concerned with the value of Particles per Cell (PPC) at the start of a simulation, which is usually set as an initial condition parameter. Total particle count is a major factor in limiting the amount of numerical heating that occurs as a simulation progresses, where low values of PPC can lead to extremely rapid growth in system KE. It is also necessary to use enough particles that the particle distribution does not dominate solution error to the extent that spatial and temporal convergence and accuracy is greatly reduced. Here we consider this phenomenon for a simple electromagnetic test case, described in detail in Section 5.3.2. From Figure 3.4 it is clear to see that, as the value of PPC is increased, we approach

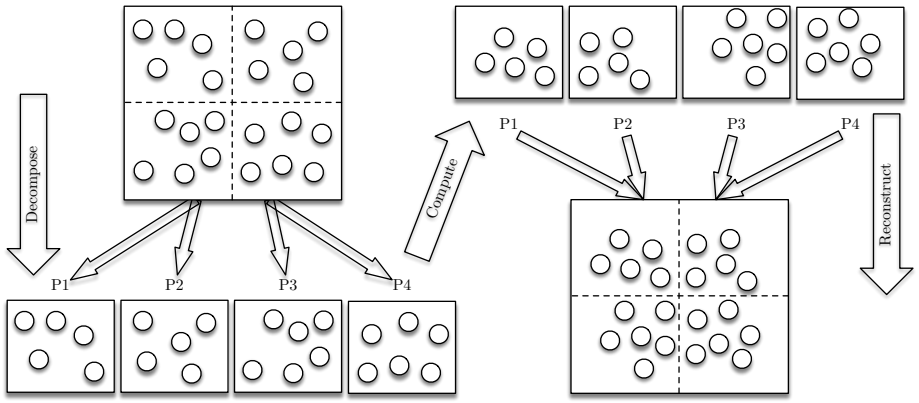


Figure 3.5: An example of how a simple PIC simulation can be parallelised across four processors.

the theoretical rate of second-order convergence in space and time.

3.6 Parallelisation of PIC Simulations

Due to the hybrid nature of the PIC algorithm coupling both grid-based and particle-based workloads, PIC codes exhibit a unique set of performance challenges to application developers on both current and future compute architectures. Fortunately, PIC methods contain a large amount of inherent parallelism that can be exploited in order to achieve high performance [3, 92, 151]. In general this is achieved by spatially decomposing the problem with respect to both the collection of particles being simulated, and the spatial grid containing the field data. Typically, this means that the grid is broken up into a set of chunks, where each chunk is assigned to a given processor. That processor then loads the particles that occupy that area of the grid, and is subsequently responsible for updating those particles during the simulation. When particles leave the domain of their host processor, they are packed into send buffers and migrated to the destination processor(s). Figure 3.5 depicts an example of how such a spatial decomposition can be achieved for an example problem being executed on four processors.

Since both the particle mover and field solver are completely independent

entities, the parallelisation of each of them can and should be considered separately. As discussed in Section 3.3.1, unstructured PIC simulations formulate the field solver as an FEM problem that can be solved by a variety of iterative or direct linear solvers. The parallelisation, performance, and scalability of linear solvers is well studied, with multiple library options available to end users. Examples of these include PETSc [1, 12, 13] from Argonne National Laboratory (ANL), Trilinos [69, 141] from SNL, and Hypre from LLNL [54, 55]. The performance behaviour of these libraries at scale has been detailed extensively in previous studies [11, 90, 95, 98, 125, 134].

The parallelisation of particle-based routines is also well documented. Such algorithms are well-suited to parallelisation as an individual particle can be handled as a completely independent entity, with minimal inter-particle data dependencies. For example, Molecular Dynamics (MD) codes that have no concept of a mesh have long been used by domain scientists to study the behaviour of dynamic systems of particles. As a result, the performance of such codes is well understood with a variety of parallelisation strategies available [28, 58, 122]. Many of the lessons learnt from MD codes can also be applied to PIC applications in the sense that particles can be processed independently, and decomposed over multiple processors much like a mesh-based problem. The primary difference between the two code types is the lack of direct particle-particle interactions[1] and the presence of a computational mesh in a PIC simulation. PIC instead approximates the interactions between particles via the forces interpolated from the fields that are calculated on the grid. This reduces a potentially worst-case complexity of $O(n^2)$ particle calculations to a single $O(n)$ loop over the particles in the simulation, executed once per time-step.

The most significant obstacle to parallelising the particle-based routines of a PIC code is the weighting of the particles to the grid, commonly referred to as *charge deposition* in electrostatics, and *current deposition* in electromagnetics. This issue occurs in the form of a *write-conflict*, as the particle contributions

[1] One should note that there are PIC codes that implement collisions between particles, but that these codes are beyond the scope of this thesis.

require a global charge or current array to be written to by all particles. As a given grid cell can contain many particles, some form of protection is required to maintain correctness, and it is this synchronisation that limits performance and scalability. This is especially true for electromagnetic problems, as a particle must deposit current to all cells crossed in a step, leading to more particle-to-grid writes than its electrostatic counterpart. This performance issue, alongside others, is explored in greater depth in Chapter 4.

In addition, the performance behaviour of PIC has been previously studied by other authors across a variety of hardware types. The performance of the GTC-P code at scale has been demonstrated on a number of notable HPC systems, such as Sequoia, Piz Daint, Titan, and Tianhe-2 [2, 140, 153], performing well in terms of both strong and weak scaling on a variety of compute architectures, including both Central Processing Units (CPUs) and accelerators. The representative code Mini-EPOCH has been used to understand the performance behaviour of, and explore novel optimisations to, the main EPOCH code by Bird et al. [20]. The ICEPIC code [26] has also been optimised to scale up to thousands of CPUs [19].

The use of Graphics Processing Units (GPUs) to accelerate the PIC algorithm has also been documented by multiple authors, showing good speedup relative to CPU implementations [39, 45]. One example, PIConGPU [36], consists of a CUDA [116] implementation of the structured PIC algorithm, demonstrating scalability across multiple GPU compute nodes. Additionally, the performance of the EPOCH code has also been evaluated for accelerator architectures, showing promising results across GPU-based systems [21]. The XGC code has demonstrated weak scaling on nearly the full pre-Exascale Summit machine (over $24,000$ GPUs) at Oak Ridge National Laboratory [133] by using the Cabana particle algorithm library [136].

As a collective, the modern computational hardware and parallel programming strategies discussed above mean that the supercomputers of today can be used to conduct PIC simulations at huge scales. In the case of EMPIRE-PIC,

a typical problem can consist of many hundreds of millions of mesh elements with over one hundred particles per element, where approximately 20% of the problem domain initially contains particles in an unbalanced problem. By comparison a balanced problem such as that considered in Chapter 4 loads particles in the vast majority of the elements. Production simulations are generally executed for over one hundred thousand time-steps, due to the large periods of time that must be simulated. As a result, runtimes of multiple days for production problems are not uncommon.

3.7 Summary

In this chapter, we have introduced the reader to the idea of plasma simulation via the PIC method, and its usefulness to the scientific community. Specifically, we have covered traditional structured FDTD-PIC simulations, and highlighted the issues that arise with representing complex geometries using this method, and how these can be addressed via an FEM-PIC approach using unstructured meshes. We also detail the relevant theoretical concepts that underpin the algorithm implemented in FEM-PIC applications, and how this pertains to the EMPIRE-PIC code. We conclude the chapter with a broad overview of higher-order algorithms, and a general high-level discussion of how PIC codes can be parallelised in a performant manner. PIC code performance and changes to the underlying algorithm are key themes throughout this thesis, and this chapter aims to provide insight into the relevant background of these topics.

CHAPTER 4

Performance Portable Finite Element Method Particle-in-Cell Simulations

While codes that simulate the behaviour of plasmas under the influence of electric and magnetic fields are a key part of the computational physics research that is often conducted on many supercomputers, such codes are extraordinarily expensive to produce and/or maintain. This is because these applications are extremely complex and producing them requires large amounts of specialist knowledge, both in terms of the mathematical methods that must be faithfully implemented, and the software engineering and tuning required to achieve acceptable performance. As a result, developing a new production application from scratch is extremely expensive; this is the case in terms of both financial cost and the time taken (timescales of many years are not uncommon). These costs are then increased by the need to continually rewrite or tune existing applications as new compute architectures continue to be made available to High Performance Computing (HPC) users. Consequently, there is a clear need for the codes that are produced to remain fit for purpose for as long as possible in order to maximise the efficiency of the resources used to develop them.

As touched on in Section 2.3.1, many traditional supercomputers are homogenous in nature and, as such, a large number of the production applications that exist today were written with this in mind. Frequently, this means that these codes are solely parallelised by using MPI with a large focus on efficient communication patterns between different nodes. While MPI processes can be placed on the same node in order to saturate multiple cores, there are other frameworks such as OpenMP that are specialised in leveraging shared memory parallelism. Additionally, an MPI-only code will be unable to take advantage

of the accelerators that are increasingly more common as compute architectures continue to diversify.

As the number of architectures continues to grow, so too does the number of programming models and libraries that are produced so that programmers can use this new equipment. Moreover, it is clearly not feasible to both produce and maintain separate versions of a code for each type of hardware an end user may want to use. Ideally, a new production code would be developed in such a way that only a single codebase must be written, and subsequently compiled for whichever hardware is desired. To this end, the Kokkos [52] framework developed by Sandia National Laboratories (SNL) has been used for the development of EMPIRE-PIC.

In this chapter, we outline the implementation of the particle-based kernels of EMPIRE-PIC using Kokkos to achieve portability across modern architectures, employing OpenMP for CPUs and NVIDIA's CUDA for GPUs; and also highlight the performance challenges of these kernels. We additionally explore the usage of the performance tuning opportunities made available by the more specialised features of Kokkos. This allows us to evaluate the performance of EMPIRE-PIC across multiple architecture types in the form of a performance study.

4.1 Kokkos Implementation

Kokkos makes various parallel patterns available to application developers, with the aim of being 'no more conceptually difficult than OpenMP' to write [51]. The options on offer are parallel for loops, reductions, and pre- and post-fix array sums (parallel scans), accessed via Kokkos' `parallel_for`, `parallel_reduce`, and `parallel_scan` functions, respectively. Parallel loops can then be written by passing a C++ function object (sometimes called a functor), or lambda function that overrides the `()` operator to the relevant Kokkos function. The operator takes a `const` index argument, such that a call to the function rep-

```
struct HelloWorld {
  KOKKOS_INLINE_FUNCTION
  void operator() (const int idx) const {
    printf("Hello iteration %d\n", idx);
  }
};

// Launch a basic for-loop with 100 iterations
Kokkos::RangePolicy<> range_policy(0,100);
Kokkos::parallel_for(range_policy, HelloWorld());
```

Figure 4.1: Flat Kokkos parallel for-loop.

```
typedef Kokkos::TeamPolicy<> TeamPolicy;
typedef TeamPolicy::member_type TeamMember;

struct HelloWorldTeam {
  KOKKOS_INLINE_FUNCTION
  void operator() (const TeamMember thr) const {
    const int team = thr.league_rank();
    const int t_id = thr.team_rank();

    // Nested loop using threads within team
    Kokkos::parallel_for(
      Kokkos::TeamThreadRange(thr, 10), [=](const int i) {
        printf("Hello team %d, thread %d, iteration %d\n",
            team, t_id, i);
      }
    );
  }
};

// Launch 50 teams, with Kokkos choosing the team size
TeamPolicy team_policy(50, Kokkos::AUTO);
Kokkos::parallel_for(team_policy, HelloWorldTeam());
```

Figure 4.2: Hierarchical Kokkos parallel for-loop.

resents the ith iteration of the loop body. These function calls are generally inlined in order to avoid the overhead of repeated function calls causing many context switches in succession. We must also pass a `RangePolicy` to the parallel function call that defines the desired iteration space. Figure 4.1 shows a simple 'hello world' example expressed as a Kokkos `parallel_for` loop using the functor-style implementation. For Central Processing Units (CPUs), this results in a typical OpenMP parallel `for` loop. In the case of CUDA builds, the choice of block and grid size is determined by Kokkos.

In addition to flat parallelism, Kokkos also allows developers to leverage multiple levels of nested shared memory hierarchical parallelism. This functionality is exposed through the use of Kokkos 'thread teams'. Teams provide a CUDA-like interface where threads are grouped into blocks; threads in the same team have access to a local shared memory, can synchronise with each other, and can also carry out team-local nested parallel loops. An example of Kokkos hierarchical parallelism is shown in Figure 4.2 (note that `RangePolicy` has been replaced with a `TeamPolicy`). In this way, a maximum of three layers of nesting can be used: team-, thread-, and vector-level parallelism. On GPUs, this maps to warp, and sub-warp parallelism, while on CPUs it is implemented as distributing nested loop iterations between threads, and decorating vector-level loops with directives such as `#pragma ivdep` to encourage SIMD vectorisation. While the maximum number of teams that can be created is arbitrary, the team size is constrained by hardware. The optimal team size varies by compute architecture; on CPUs it is typically one, for KNL it is the number of hyperthreads used per core, and for CUDA it is generally some multiple of 32 (the warp size). Therefore, in order to achieve performance portable code, it is usually best to allow Kokkos to decide the team size at compile-time, based on the target architecture.

The majority of the kernels detailed in this chapter were implemented using a flat approach unless otherwise stated. However, the hierarchical parallelism features described above can be used to implement more complex algorithms that can address performance issues by making better use of the underlying hardware. We explore such tuning in this chapter for the particle mover (Section 4.1.3), and the charge deposition scheme for electrostatic simulations (Section 4.1.4). In all cases where the teams interface is used we allow Kokkos to select the most appropriate team size for each build.

4.1.1 Application Data Layout

Efficient memory accesses are crucial for achieving high performance PIC in general, particularly for the particle-based kernels, due to the low arithmetic intensity relative to the amount of bytes moved to and from main memory. As a result, it is important to consider the way that the data used by an application is laid out in memory. A memory layout that performs well on a CPU-based architecture is not guaranteed to perform well on GPUs. Kokkos abstracts the notion of memory layout away from the application developer by storing data in so-called Kokkos 'views'. Each view has an associated memory layout template parameter, allowing the appropriate layout for a given architecture to be selected at compile-time. CPU-based systems default to row-major layout such that a given thread can access consecutive data entries in order to make good use of cache. For GPUs a column-major layout is chosen as the default, such that consecutive threads in the same warp access consecutive locations in memory; this is known as coalesced access, and makes better use of the available global memory bandwidth.

Views also have an assigned memory space that specifies where their data is stored. In the case of EMPIRE-PIC, host memory is used for CPU systems, whereas CUDA Unified Virtual Memory (UVM) is required for NVIDIA GPUs to allow for compatibility with CUDA builds of Trilinos. This has the additional benefit of removing the need for explicit data transfers between the host and device. Both the electric and magnetic field data are stored in $N \times 3$ two-dimensional Kokkos views, where N is the number of degrees-of-freedom for the specific field. The particle data is stored in a Structure-of-Arrays (SoA) layout, using one-dimensional Kokkos 'dynamic views' that support constant-cost runtime resizing, greatly simplifying the addition of new particles to the data structure. The use of an SoA layout has the benefit of allowing each dynamic view to be accessed in unit stride, facilitating both vectorisation on CPUs and coalesced access on GPUs.

4.1.2 Weighting of Fields to Particles

As the values of the fields are only known at the points of the problem mesh, it is necessary to interpolate their values to the position of the particles as shown in Section 3.3.2. To accomplish this, the particle container contains both $\vec{E}$ and $\vec{B}$ arrays to store the results of gathering the fields to each particle. This also improves spatial locality during the particle move by avoiding the repeated reading of irregular memory locations that would result from computing the field values for each particle on the fly. Moreover, the operations of this kernel are free from dependencies, making the code much easier to parallelise. Representing the kernel in Kokkos is a simple matter – a single particle is assigned to each Kokkos thread.

For electrostatic runs no further tuning is required as there is no need to calculate a magnetic field. For electromagnetics, we employ loop fusion by merging the loops that perform the magnetic and electric field weighting. This halves the number of times each particle must be fetched from main memory (this is crucial, given the memory-bound nature of Particle-in-Cell (PIC) algorithms). As a consequence this also eliminates redundant evaluation of the basis functions at the particle location – one of the most expensive calculations in the particle-based parts of the code.

4.1.3 Particle Move

The particle mover updates the velocity and position of each particle based on the field values gathered during the field weighting step. This is done via the method detailed in Section 3.3.3. As the update of each particle is completely independent from the movement of the others, this kernel also lends itself well to parallelisation due to the lack of dependencies. This is especially apparent on GPU-based systems where an extremely large number of threads can be executed in parallel, combined with high-bandwidth memory. However, the additional control flow to handle particles crossing process and/or element

boundaries can lead to warp divergence on GPUs and make achieving satisfactory vectorisation challenging on CPU systems.

In the case where a particle would cross an element boundary the move is broken up into its segments, and the move routine is applied to each segment in turn. Consequently, particles that move at a higher velocity or are in areas of the mesh that are more refined will make more crossings in a time-step than other particles in the simulation. It is this behaviour that leads to warp divergence on GPUs. The fundamental principles of the Single Instruction Multiple Thread (SIMT) paradigm mandate that each thread in a warp must carry out the same instruction at the same time, i.e., threads in a warp operate in a lock-step manner. This means threads that were assigned particles that make less crossings must sit idle while other threads in the same warp finish processing their particles, thus leading to a performance penalty. When parallelised in a flat manner using Kokkos this issue is unavoidable, but with the usage of hierarchical parallelism features it can be somewhat remedied.

By assigning each Kokkos team a chunk of particles, this chunk can then be drained by the assigned team. The key principle behind the idea is that using nested parallelism allows threads that are idle to fetch another particle to move, thus keeping the warps full with useful work. Finally, for Graphics Processing Unit (GPU) builds, additional team-local data is stored in block shared memory in order to reduce DRAM traffic. The team-based particle move takes place as follows:

1. Launch one team per particle chunk, and compute the start and end particle indices for the team.
2. Each thread in the team carries out one iteration of the move.
3. Execute a team-local parallel reduction, counting how many threads require a new particle.
4. Execute a team-local parallel prefix sum, determining the chunk offsets of the new particles to be fetched.

5. Each idle thread fetches its assigned particle, and the next iteration begins.

This process is repeated until the assigned chunk of particles has been drained *and* all threads in the team are idle, a state which is detected by a final parallel reduction.

4.1.4 Weighting of Particles to Grid

For electrostatic problems each particle must deposit charge to each node of its current cell at the end of the time step as shown in (3.19), while in electromagnetic simulations the particle must contribute current to each element crossed during the particle move, as defined in (3.48).

As there can be many particles occupying the same grid cell at any one time, there is the possibility of data hazards when executing these procedures in parallel. This occurs in the form of a write-conflict when multiple threads attempt to deposit charge or current to the same memory location(s) simultaneously. Therefore, some method of protection is required in order to prevent erroneous results. Possible solutions to the data hazard problem include the use of colouring methods to ensure that threads write only to non-conflicting locations, element-local reductions, or keeping thread-local copies of the data, only requiring an atomic operation or reduction for the final deposit. In this chapter, we consider three different approaches: the use of atomic writes, the use of data replication with a follow-up reduction, and element-local reductions. Note that element-local reductions are considered for electrostatic problems only as the approach relies on the guarantee that the particles processed do not leave their cells. This assumption does not apply in electromagnetics as particles make deposits as they move across cells.

The implementation of these different methods is a relatively simple matter. Beginning with atomic writes, all Kokkos views have the ability to accept a `Kokkos::MemoryTraits` template parameter, that controls various configurable properties of the view. In our case it is sufficient to specify the memory trait `Kokkos::Atomic` to ensure that all updates to the underlying data do not con-

flict. This ensures that the correct atomic instructions for the target hardware are emitted during compilation.

Data replication is also handled by Kokkos through the `ScatterView` construct, which performs the data replication for the user, and provides an access function which allows a thread to modify its local data. The user can then invoke a follow-up reduction to aggregate the contributions of each thread into a destination view – the charge or current arrays, in our case. This implementation is only tested for CPUs as such a strategy for GPUs would result in vast amounts of copies of the original view. For example, a V100 would require 80×2048 copies of the entire array (the number of streaming multiprocessors multiplied by the maximum amount of threads for each), which is clearly not feasible the as device has much less memory capacity than the host.

Implementing an element-local reduction strategy for electrostatics is the most complex of the three approaches considered here; the use of Kokkos hierarchical parallelism is required. As the particles are sorted by element after the move step, we can launch a Kokkos team for each element of the mesh, with team-local shared memory being used to store the intermediate results. This process can be summarised as follows:

1. Launch one team per element.
2. Use the threads of each team to perform a nested `parallel_reduce` over the particles in the assigned element.
3. Contribute the result for each team to the global charge array using atomic writes.

While this approach may seem poorly suited to problems where the particle distribution is non-uniform (as we are parallelising over elements at the top level), in practice the impact of this is low as long as the number of elements that contain particles is greater than or equal to the maximum number of teams that can execute in parallel. Again, using a V100 GPU as an example, the mesh

would need to be made up of at least 80 elements that contain particles; this will be the case for almost all real problems.

4.2 Results

The specifications of all the hardware used to conduct the experiments presented here is detailed in Section 2.5, where single-node machines and supercomputers are listed separately. Every effort has been made to consider a diverse set of hardware, in terms of both age and type. For Intel CPUs, the usage of both Broadwell and Cascade Lake generations allows the performance of EMPIRE-PIC pre- and post-AVX-512 SIMD instructions to be considered. This is also covered by the Intel Xeon Phi Knights Landing (KNL), which also enables us to observe the behaviour of the code on a many-core CPU-like architecture, which typically rely heavily on vectorisation to achieve performance. The final CPU considered is the Cavium ThunderX2, enabling comparison to non-Intel hardware. Regarding GPUs, we consider three generations of NVIDIA hardware: Kepler, Pascal, and Volta.

For all runs carried out on Intel CPUs, version 18.0.5 of the Intel compiler is used, with the highest level of code optimisations enabled (`-O3`). On the Broadwell system AVX2 vector instructions are enabled (`-xCORE-AVX2`), while for Cascade Lake and KNL, AVX512 is enabled (using `-xCORE-AVX512`, and `-xMIC-AVX512`, respectively). For the runs on the ThunderX2, GNU version 8.3 `C` and `C++` compilers are used, again with level three optimisations, and ARM SIMD extensions enabled (`-march=armv8.1-a+simd`). All CUDA builds of EMPIRE-PIC make use of the GNU compilers version 7.2, combined with `nvcc` version 9.2.88. As with other compilers, level three optimisations are enabled. Finally, we pass the relevant CUDA architecture flags to `nvcc` (i.e. `-arch=sm_[35,60,70]` for Kepler, Pascal, and Volta respectively).

Regarding the configuration of the runs performed, a hybrid MPI+OpenMP approach is taken for all CPU systems, as well as the many-core KNL. One

Size	Num. Elements	Num. Particles	Particles/Element
S	337.0 k	16.0 M	47.5
M	2.7 M	128.0 M	47.8
L	20.7 M	1.0 B	49.5
XL	166.0 M	8.2 B	49.4
XXL	1.3 B	65.6 B	49.2

Table 4.1: Problem sizes used to test the performance of EMPIRE-PIC.

MPI process is used per-socket in order to account for Non-Uniform Memory Access (NUMA), and all cores on the socket are then saturated using OpenMP threads. For the KNL runs, the hardware is launched in quadrant mode. We therefore use one process per quadrant, with 16 OpenMP threads used per process. For GPU runs a single process per accelerator card is used.

In this section, we simulate a three-dimensional problem on a tetrahedral mesh, with both electrostatic and electromagnetic runs considered. The problem domain is uniformly filled with equal amounts of both electrons and hydrogen ions to a number density of 1×10^{16} m^{-3}, with particles being loaded randomly within their assigned elements. The initial temperature is set to approximately 10 eV. The plasma is evolved for a time of 6×10^{-10} s over 100 simulation time-steps, and the Courant-Friedrichs-Lewy (CFL) condition ranges from 3–30. Full details of all problem sizes are given in Table 4.1. For single node runs we solely consider problem size S. The larger problem sizes are chosen such that each subsequent problem is a factor of eight larger than the previous one – this facilitates both strong and weak scaling studies.

4.2.1 Effects of Optimisations

Write-Conflict Resolution

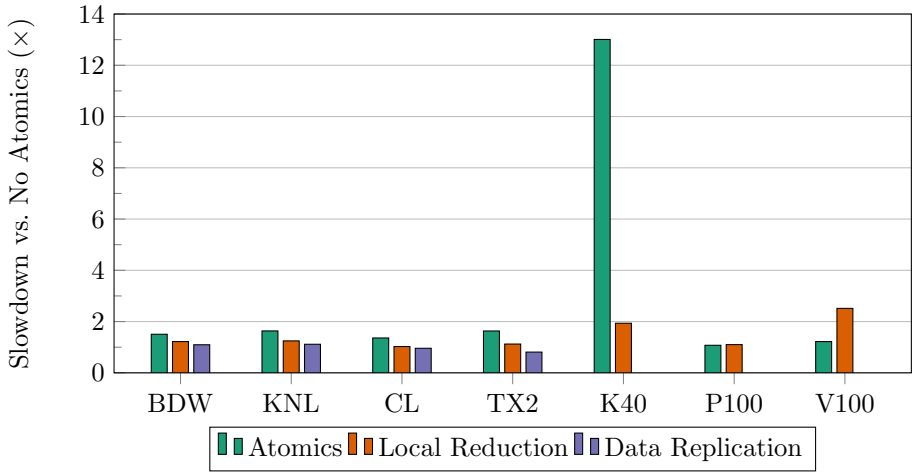
In order to assess the overhead of the various resolution methods described above, it is necessary to establish a baseline execution time for both the charge and current depositions on all of the platforms considered. To this end, both electrostatic and electromagnetic EMPIRE-PIC simulations were conducted using unprotected writes to the charge and current arrays. While this means

	Electrostatic		Electromagnetic	
Architecture	**No Atomics**	**Atomics**	**No Atomics**	**Atomics**
Broadwell	3.459	5.207	18.326	28.711
KNL	6.611	10.808	36.826	56.416
Cascade Lake	1.934	2.631	9.937	15.467
ThunderX2	4.417	6.771	12.800	24.580
K40	2.264	29.451	9.682	49.211
P100	1.190	1.215	6.925	7.445
V100	0.437	0.537	3.650	3.913

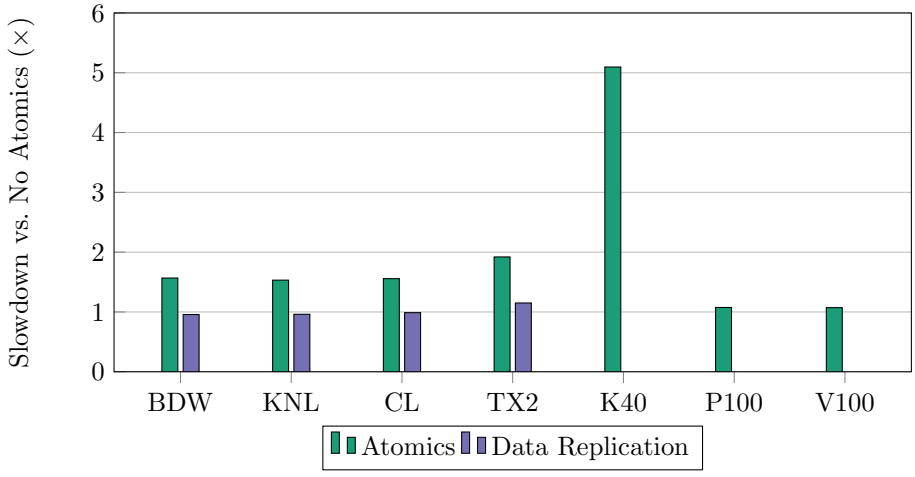
Table 4.2: Time spent in charge weighting, and moving particles using no atomics versus atomics (in seconds) for electrostatic and electromagnetic problems, respectively.

that the code will reach an incorrect answer due to the ensuing data hazards (therefore making the data slightly unrealistic), it does provide a best-case performance target to aim for. Table 4.2 shows the results of these experiments for both problem types, and provides a comparison to the performance achieved by the simplest resolution method considered in this chapter, atomic writes. An immediately obvious trend in the data is that the overhead of atomic writes is often higher for electrostatics than electromagnetics. The reasons for this are twofold: first, in electromagnetics the current deposit is made as part of the particle move, meaning that the kernel itself contains more work. This means that atomic contention dominates the kernel runtime more in the electrostatic case. Secondly, in electrostatics, all charge contributions happen concurrently, meaning that contention between threads is more likely (i.e. a thread is much more likely to need to back off and attempt its contribution again).

When comparing across hardware it is also apparent that the NVIDIA P100 and V100 exhibit much less overhead for the use of atomic writes than the CPU platforms and the Tesla K40 due to the advanced hardware-acceleration of atomics supported by the Pascal and Volta architectures. With performance penalties between 2%–7%, the need to focus on deposition schemes that actively avoid the use of atomics is less relevant on modern NVIDIA accelerators as the extra work induced by such schemes often has a greater overhead. The same cannot be said for the CPU systems and K40 where we see overheads ranging between 36%–63% percent (CPUs), and 400%–1200% (K40). It is clear that



(a) Electrostatic charge weighting kernel.



(b) Electromagnetic particle move kernel.

Figure 4.3: Slowdown incurred from various write-conflict resolution strategies.

significant improvements can be made here.

Figure 4.3(a) shows the slowdown (versus unprotected writes) incurred by each of the write-conflict resolution methods considered in this chapter for the electrostatic problem. It is clear to see that, as discussed previously, that atomic writes are the worst option in all cases for the CPU systems. This is also disproportionately the case with the K40, where the lack of specialised atomics combined with a large degree of parallelism renders the use of atomics impractical when compared to the other systems. We expect the use of element-local

reductions via Kokkos teams to result in notable improvements in performance due to the large decrease in the number of atomic writes required. Specifically, in our 3D tetrahedral case this method results in $4 \times N_{elem}$ writes to the global charge array, contrasted to $4 \times N_P$ writes for a naïve approach (recall that $N_{elem} \ll N_P$ in almost all cases). These improvements are seen across almost all systems, with the K40 showing a huge reduction from 13× to approximately 2× slowdown. This is not surprising given that the K40 by far had the most to gain from the decrease in atomics used. The outliers in the data are the P100 and V100, whose worsened performance can be explained by the overhead of the additional reductions being greater than that of the hardware-accelerated atomics. Moving on to analyse the data replication and follow-up reduction approach, we can see that this method is the best choice for all of the CPU systems, in part due to the total lack of atomic writes. Of particular interest is that the data replication implementation outperforms the version of the code where no conflict resolution is implemented for the Cascade Lake and ThunderX2 systems. This is due to improved cache behaviour as a result of reduced false sharing penalties, as each thread now only operates on thread-local data instead of a shared global Kokkos view, i.e., less cache lines are evicted.

Figure 4.3(b) compares the use of atomic writes to the data replication approach for the electromagnetic problem. While the only option for the GPUs is atomics, they are included in the analysis in order to assess the relative overhead in the particle move for all systems considered. As before, data replication followed by a reduction results in greatly improved performance versus atomic updates to the global current array. In this case, all of the CPUs show the benefit of the improved cache behaviour discussed above, with the exception of the ThunderX2. As current is accrued to each cell a particle crosses during its move, an electromagnetic simulation typically results in a greater number of particle to grid writes, thus increasing the benefit to be had from use of cache. While the GPU systems continue to use atomics, the overhead is almost insignificant, resulting in comparable results to the CPUs. This is not the case for

the K40, which continues to be at a performance disadvantage when considering this kernel.

Particle Move Scheduling

The graph displayed in Figure 4.4 shows the relative performance of the Kokkos team-based particle move kernel on all three GPUs used in this thesis, compared to the version of the kernel parallelised in the traditional flat manner, for both electrostatic and electromagnetic simulations. Specifically, we consider how the achieved speedup varies with chunk size, where a chunk size x and team size y means that a given team is assigned xy particles in total. It is clear from the data that the use of the hierarchical approach leads to performance gains across all three of the GPUs used for the analysis, with electromagnetics benefiting more than electrostatics. This benefit levels off as the CUDA warps become saturated with useful work and begins to slowly degrade as warps become oversubscribed. Further tests showed that a chunk size significantly larger than shown here means that fewer total teams are launched. For example, for the V100, a chunk size of one thousand resulted in speedup degrading to values less than one. This is the result of reduced achieved occupancy on GPUs significantly harming performance due to idle streaming multiprocessors.

The most pronounced improvements occur on the K40 for both of the simulation types. This is unsurprising for two reasons: (i) as the base particle move time is much higher, the K40 has the most to gain, and (ii) older GPU hardware is worse at compensating for warp divergence. This is demonstrated by the performance improvement steadily decreasing as we move to newer architectures such as Volta, where we observe only an approximately ten percent improvement in runtime. This phenomenon can be explained by hardware improvements; NVIDIA GPUs based on the Volta architecture allow individual threads to have their own call stack and program counter, therefore allowing independent thread scheduling to achieve greater parallel efficiency [115].

We now consider the overall benefit of the team-based approach on GPUs in

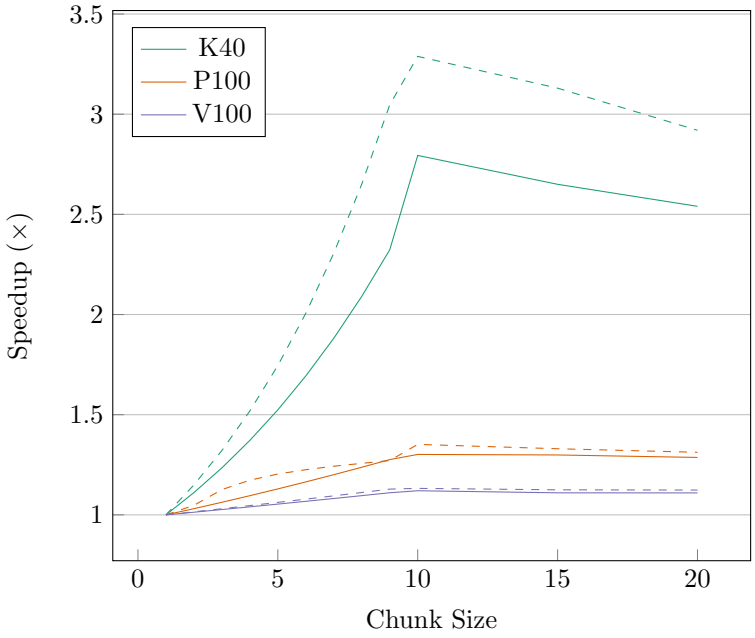


Figure 4.4: Impact of chunk size on particle move speedup on all GPUs used. Solid lines and dashed lines denote electrostatic and electromagnetic simulations, respectively.

comparison to CPU systems for the optimal GPU chunk size of ten. Note that while we do not expect to see performance gains on non-GPU hardware (we instead expect a penalty), these systems are included in the analysis in order quantify to what level CPU performance is harmed by such an accelerator-oriented approach. This allows us to gain an understanding of the performance portability of the team-based move kernel across multiple architecture types. This data is shown in Figure 4.5.

As expected, for the CPU systems we observe slowdowns for both problem classes when using the hierarchical version of the particle move kernel. Recall that as non-GPU systems do not suffer from thread divergence, the addition of the extra parallel reductions and scans simply adds more work for no benefit. This effect is most noticeable for the electrostatic simulations, where this additional workload will make up a greater proportion of the time spent in the particle mover. It is clear that two versions of the move kernel would need to be maintained in order to reach peak performance on both CPUs and GPUs.

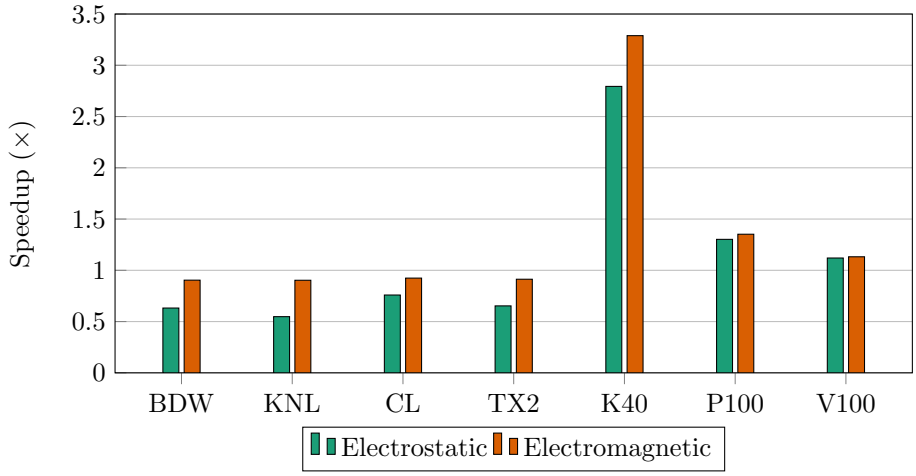


Figure 4.5: Impact of Kokkos team-based approach on particle move kernel execution time.

4.2.2 Overall Performance

This section now details the overall performance of the EMPIRE-PIC application across a variety of platforms, while also taking into account the previously detailed optimisations. For CPU runs we use a data replication charge/current deposition strategy, and the flat parallelised particle move, while for GPUs we use atomic writes, and the team-based particle mover. For electrostatic tests the K40 is set to use element-local reductions to deposit charge. While the work in this chapter has focused on the implementation and tuning of the particle-based kernels, we will now also include the time spent solving Maxwell's equations for the updated electric and magnetic fields. This allows us to conduct our analysis in the context of overall application performance by observing the Time-to-Solution (TTS) for each run.

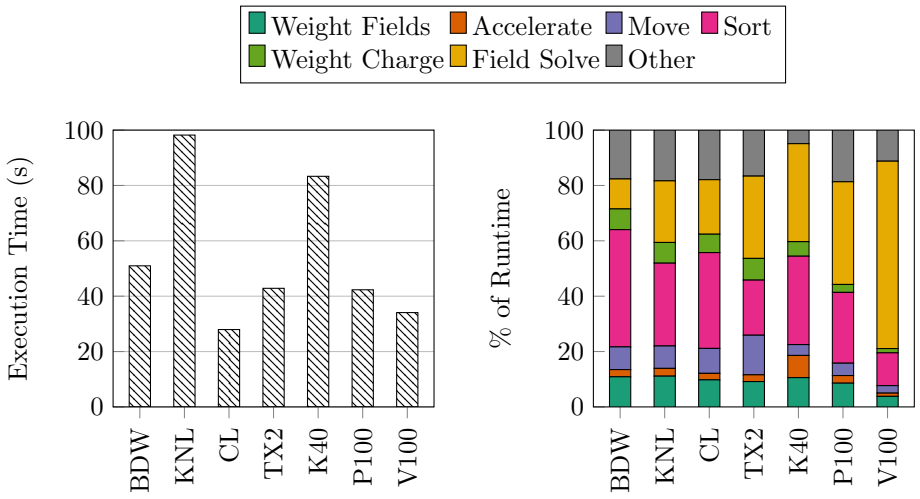
Single Node Comparison

We begin our analysis with single node runs of the electrostatic problem across all platforms. TTS can be seen in Figure 4.6(a). Alongside the total execution time, Figure 4.6(b) shows the proportion of the execution time spent in each of the key EMPIRE-PIC kernels. The cost of migrating particles between pro-

cesses is using MPI included in the move kernel time. Finally, the 'Other' time measurement includes time spent performing Input/Output (I/O), and other tasks such as filling views at the start of a time-step.

Our results show that the combined total time spent processing the particles is greater than for the linear solve of Maxwell's equations for all systems, with the exception of the V100. We also observe comparable overall results when contrasting the performance of the ARM ThunderX2 to the Broadwell, with the ThunderX2 performing better on both the linear solve and most of the particle kernels, despite having the same peak FLOP rate. This difference can be explained by the ARM system having double the number of memory channels of the Broadwell system – a total of eight channels versus four providing an advantage for traditionally memory-bound algorithms. The traditional CPU systems also vastly outperform the KNL. This is unsurprising as much of the complex control flow present in unstructured PIC inhibits vectorisation (which is key to achieve good performance on KNL due to the low clock rate). When comparing between GPUs and CPUs it is clear that, when fair comparisons are made (e.g. P100 vs. BDW, V100 vs. CL), comparable TTS is achieved. The GPUs tend to perform better on the particle-based kernels due to the high degree of parallelism and memory bandwidth, but are noticeably slower at solving for the updated fields, with the P100 outperforming the V100. It is clear that the performance of the electrostatic field solver should be a key focus of future work.

The same analysis was also carried out for the electromagnetic problem, resulting in the data shown in Figures 4.7(a) and 4.7(b). While the actual time spent processing particles is much longer, we see similar performance proportions to electrostatics for the particle-based kernels across all systems, with the particle move increasing in cost as a result of now containing the current deposition. The weighting of fields to particles, and the follow-up particle acceleration also takes more time than in electrostatics as we must now handle a magnetic field. In general, the performance of the particle-based kernels appears to be a



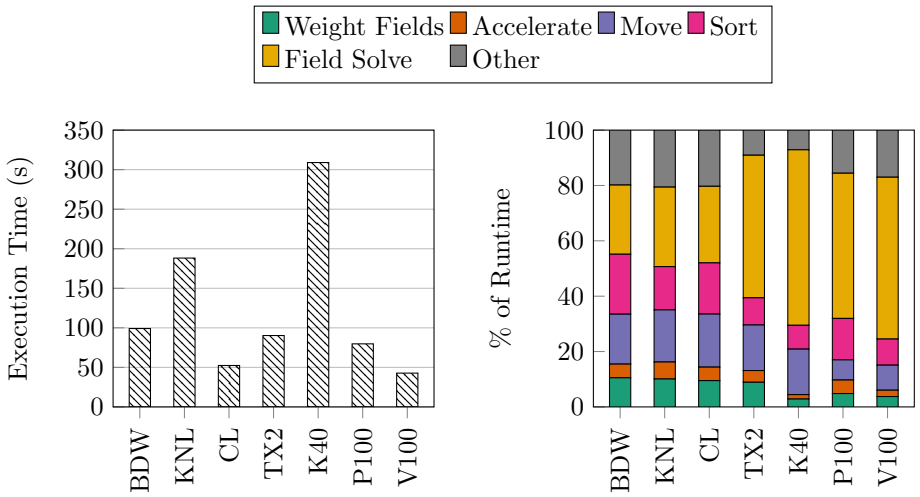
(a) Total time.

(b) Kernel breakdown.

Figure 4.6: Breakdown of best kernel performance across all platforms for the electrostatic problem.

function of memory bandwidth. However, the kernels with more compute (such as the move) also benefit from higher peak FLOP/s, leading to the Cascade Lake outperforming the ThunderX2, despite having a slightly lower bandwidth. This also matches the trend seen in electrostatic simulations.

As before, the modern GPUs continue to outperform the CPU systems when processing particles, with the K40 doing disproportionately worse than the other cards due to its slower atomic writes. As with electrostatics, the field solver again vastly dominates the runtime when using GPUs (representing more than 50% of time for all cards), while remaining a relatively low proportion of the total time on most CPUs. This performance issue means that the advantage gained through rapid particle processing does not result in significantly faster TTS when comparing to the other systems, and serves to highlight the difficulty of achieving performance portability for hybrid workloads such as PIC. As such, the electromagnetic solver on GPUs should also be a focus of future development efforts – the resolution of this issue would result in a sizeable TTS performance gap between GPUs and CPUs.



(a) Total time.

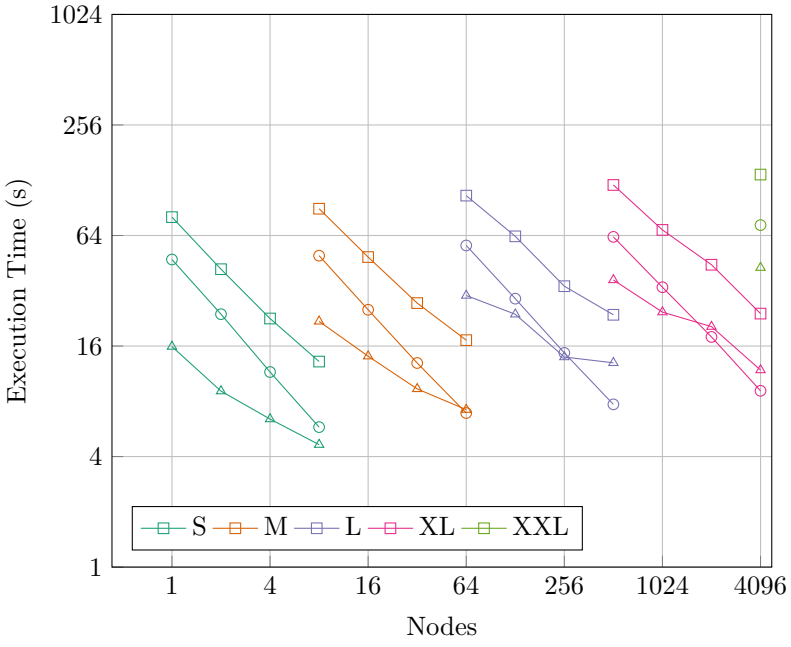
(b) Kernel breakdown.

Figure 4.7: Breakdown of best kernel performance across all platforms for the electromagnetic problem.

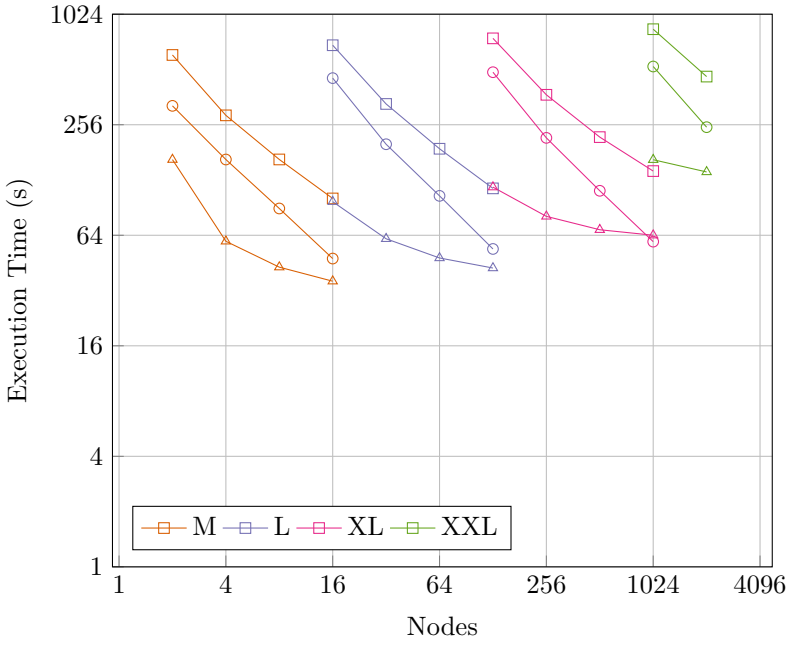
Scaling Study

Figures 4.8 and 4.9 show the results of both strong and weak scaling studies of EMPIRE-PIC for all of the supercomputers detailed in Section 2.5.2, for the electromagnetic problem. We present data for the total time spent processing particles including MPI communications, the time spent solving for the updated fields, and the total execution time of the main loop of the application. It is also important for the reader to note that this scaling study does not seek to directly compare TTS between the four supercomputers, as such a comparison would be unfair – particularly with Sierra. Instead, the objective is to assess whether EMPIRE-PIC achieves acceptable performance and scalability across all systems when accounting for the age of the hardware, with the aim of demonstrating the *portability* of the code.

It is clear to see that EMPIRE-PIC achieves near ideal strong scaling for the particle update on all systems with the exception of Astra when high levels of strong scaling are applied. This is to be expected as the particle kernels contain no dependencies, resulting in only minimal communications between processes when particles must be migrated to a neighbour. The linear solve

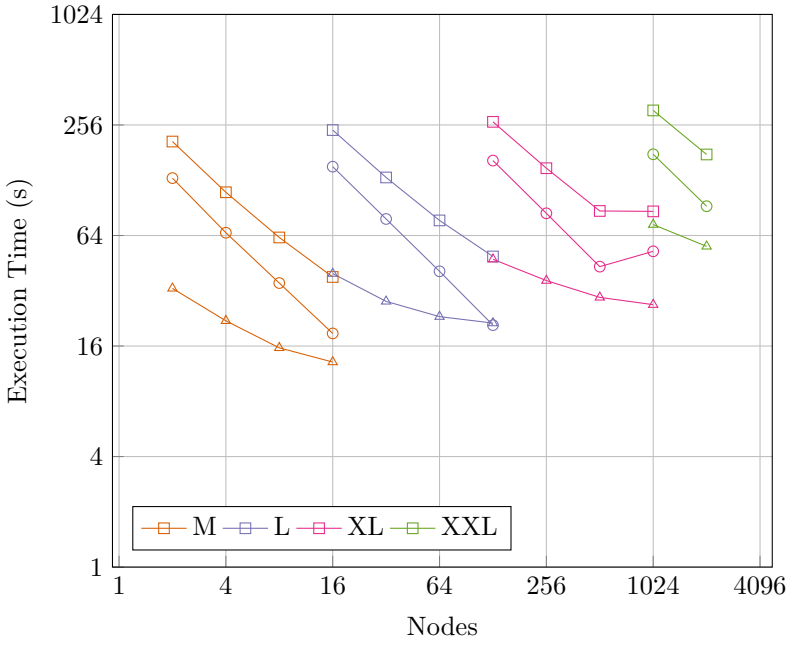


(a) Trinity (Haswell)

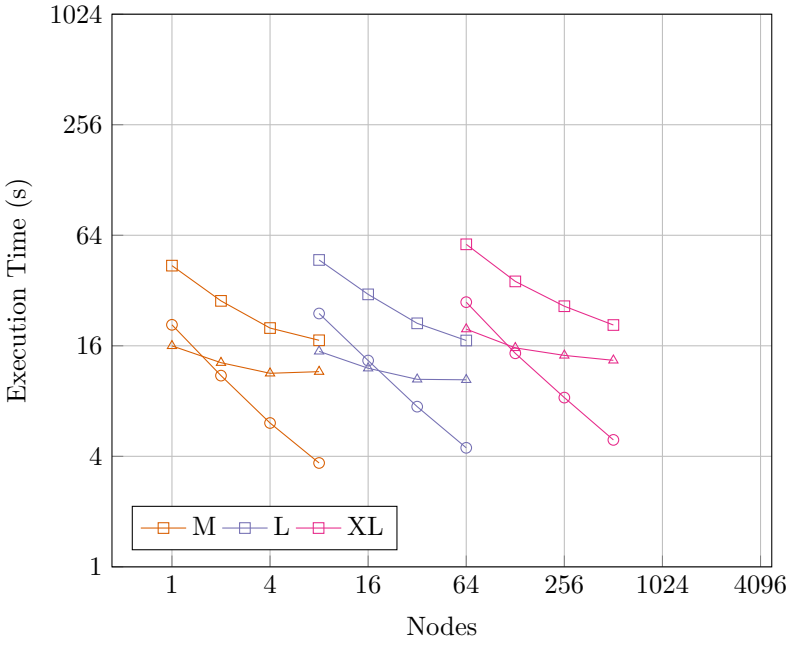


(b) Trinity (KNL)

Figure 4.8: EMPIRE-PIC strong and weak scaling study results for both partitions of the Trinity supercomputer. Squares, circles, and triangles represent the main time loop, particle update, and field solve respectively.



(a) Astra



(b) Sierra

Figure 4.9: EMPIRE-PIC strong and weak scaling study results for the Astra and Sierra supercomputers. Squares, circles, and triangles represent the main time loop, particle update, and field solve respectively.

strong scales well on both Haswell and ThunderX2 but there is little strong scaling benefit observed for either the KNL partition of Trinity or for Sierra. It is also evident that the field solver is the main performance bottleneck for both of these supercomputers when strong scaling.

With regards to weak scaling, EMPIRE-PIC scales well across all of the chosen systems, demonstrating acceptable TTS even as the problem size being simulated is vastly increased versus the base case. While the number of nodes used on Sierra is comparatively low relative to the other systems, resulting in fewer data points, there is no evidence that the CUDA version of the code will not continue to both strong and weak scale. As a result, we have demonstrated that EMPIRE-PIC can successfully scale up to greater than one hundred thousand CPUs and two thousand GPUs, facilitating the solution of problems of great complexity.

4.3 Summary

As we approach the major milestone of Exascale computing, modern computational architectures will continue to diversify. It is crucial that current and developing HPC applications can adapt to this ever-increasing hardware heterogeneity. As with traditional structured PIC methods, FEM-PIC is highly parallel, and is thus well-suited to execution on a variety of modern compute architectures. In this chapter, the implementation of the particle-based kernels of the EMPIRE-PIC C++ FEM-PIC application using the Kokkos performance portability framework is documented.

While in most cases the flat parallelism supported by Kokkos' most basic parallel patterns was used to express the kernels documented in this chapter, the use of more advanced hierarchical parallelism features was also explored. We have seen that the use of nested parallelism can be used to express more complex algorithms, and that performance gains can be made as a result. Specifically, we have shown that element-local reductions can be used to reduce the

performance impact of atomic writes on CPU-based systems, as well as older GPUs with poorer hardware support for atomics. We have also demonstrated that attempting to address the warp divergence present in a flat particle move strategy leads to performance gains across all GPUs used in this study, even where hardware has been designed to reduce the impact of this divergence in naïve implementations.

Regarding overall performance and portability, we have found that EMPIRE-PIC performs well across a variety of different compute platforms at the single node level, including traditional CPUs, many-core CPUs, and NVIDIA GPUs, while consisting of a single codebase. When fair comparisons are made, the CPU systems show comparable TTS to systems with accelerators. However, it is likely that accelerators will outperform the CPU systems in the future if the performance bottleneck posed by the linear solver for CUDA builds can be alleviated, particularly for electromagnetic simulations.

When considering multi-node supercomputers, we have observed near-ideal strong scaling and acceptable weak scaling for the electromagnetic particle-based kernels. On the Sierra platform, we have seen comparatively low levels of strong scalability for the linear solver, but good weak scaling is still achieved even up to problems consisting of more than one-hundred million mesh elements.

CHAPTER 5

Higher-Order Particle Representation

In Section 3.4, we discussed the background and potential benefits of higher-order methods for Particle-in-Cell (PIC) algorithms. It was also highlighted that the use of higher-order elements for PIC necessitates the use of smoother particle shape functions in order to achieve the desired simulation convergence, and that the implementation of these shapes in unstructured PIC codes is non-trivial.

In this chapter we propose modifications to the core algorithm of EMPIRE-PIC by representing particles as having a smooth quadratic shape, with compact support on a fixed radius, which is numerically integrated against the test function representing the weak form of the currents or charge densities. This integration is performed by numerical cubature where the cubature points are represented by virtual particles surrounding each super-particle. Each virtual particle has an associated offset and weight derived from Gaussian quadrature rules and the chosen radius. This approach also has the advantage of requiring little extension to the core PIC kernels. This chapter documents the algorithmic implementation of these modifications into EMPIRE-PIC, for both electrostatic and electromagnetic problems with periodic boundary conditions. The effect of the algorithm on simulation solutions is explored using four representative benchmark problems.

While smooth particle shapes have been explored previously by other authors, the use of virtual particles is a key feature of our implementation. This differs from the approach used by Jacobs and Hesthaven [82], where particles in a discontinuous Galerkin PIC code are represented as a cloud of constant size with particles weighted to all elements within the cloud radius. We instead

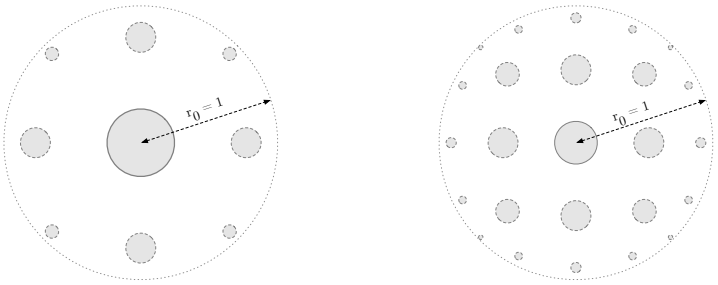
examine weighting each virtual particle to/from its associated element in a continuous Galerkin code. The virtual particle approach also has the advantage of being able to tune the offsets and weights of the virtual particles to reproduce a given shape function for the particle cloud with relative ease. Finally, as the virtual particles are computational, we obtain the additional benefit of adding increased arithmetic intensity to traditionally memory-bound particle kernels within the PIC method.

5.1 Higher-Order Particle Shapes

Let us first consider the electrostatic formulation of the standard PIC algorithm due to its simplicity, as the scheme can be later expanded to electromagnetics. As discussed previously we must formulate the weak form of Gauss' Law such that we can integrate the electric potential with a given test function and generate a stiffness matrix. Solving Gauss' Law also requires the charge density ρ to be computed from the computational particles. These particles are generally represented as shape functions S in space. In the standard algorithm, the shape function is generally the Dirac delta function, δ. The charge density can then be integrated with the test function. One should note that integrating δ with the linear nodal basis is equivalent to piecewise linear interpolation in Finite Difference Time Domain (FDTD) PIC. This results in a summation at the particle locations when $S = \delta$.

$$\int_{\Omega_j} \rho \hat{v}_i \, dV = \sum_{k=1}^{N_P} \int_{\Omega_j} S\left(\vec{x} - \vec{x}_k\right) q_k \hat{v}_i \, dV = \sum_{k=1}^{N_P} q_k \hat{v}_i(\vec{x}_k) \tag{5.1}$$

This simple integration is valid, independent of the order of the test function. However, the use of the Dirac delta function results in a particle shape that is not a smooth representation due to its point-like nature. In the following section, we show how δ can be replaced with a smooth shape function, and how this can be implemented through the use of virtual particles.



(a) Three quadrature points (b) Five quadrature points

Figure 5.1: Example virtual particle layouts of differing orders. Virtual particles are grey with dashed borders, with the physical location is represented by the central virtual particle (solid border). Virtual particles are sized proportionally to their weights.

5.1.1 Smooth Particle Shape Function

In order to solve the problem of a non-smooth particle shape we now propose representing particles as having some defined fixed size. Specifically, we assume that particles possess some radius r_0, and have a parabolic shape subject to the following shape function – replacing the usual Dirac delta function. In (5.2) we also have normalisation constant $c = 2/\pi {r_0}^2$.

$$S(\vec{x} - \vec{x}_0) = \begin{cases} c\left[1 - \left(\frac{r}{r_0}\right)^2\right] & \text{if } r \leq r_0 \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

The exact integral of the shape function with the test function is given below in (5.3). However, integration of this shape function with the test function is generally computationally intractable when spanning more than a single element. We therefore handle the integration of this function via the application of Gaussian numerical quadrature.

$$\int_0^{r_0} \int_0^{2\pi} rc\left[1 - \left(\frac{r}{r_0}\right)^2\right] \hat{v}_i(r,\theta)\ d\theta\ dr \tag{5.3}$$

Point	**Position** x_i	**Weight** w_i
0	$-\sqrt{\frac{3}{5}}$	$\frac{5}{9}$
1	0	$\frac{8}{9}$
2	$\sqrt{\frac{3}{5}}$	$\frac{5}{9}$

Table 5.1: Positions and weights for three-point Gaussian quadrature.

5.2 Implementation

The smooth particle shapes described above are implemented by taking a given simulation particle, and surrounding it with a set of computational virtual particles. This allows one to move the quadrature weights from the mesh onto the virtual particles themselves. In this representation the particle radius is fixed independently of the size of its containing element, and the central particle is used to track the physical location of the particle in the simulation space. The virtual particles represent quadrature points for the particle; each has a fixed associated position offset $\vec{o}_v$ and weight factor w_v, where the weight incorporates Gaussian quadrature weights and the shape function. It should be noted that the sum over the set of virtual particle weights must be equal to one to ensure the correct total contribution once all virtual particles are processed. Example particles represented in this way are shown in Figure 5.1. We now derive virtual particle weights and offsets using a shape represented by mapping a square to a circular shape. For 3D problems we instead map a cube to a sphere. The choice of a circular/spherical shape has certain benefits. Firstly, this shape prevents the grid biasing that would occur with the use of a square/cube layout. Secondly, such a shape captures the notion of the Debye sphere [25, 38] and allows for a better representation of this concept in a simulation.

Given Gaussian quadrature of an arbitrary order, let r_0 be the chosen particle radius, and x and y be the positions of two Gaussian quadrature points. Additionally, let w_x and w_y be the weights of these points and $\mathbf{J}$ be the determinant of the Jacobian for the mapping at these points, which we include in

order to correctly map from the reference volume to the mapped volume. For convenience, Table 5.1 shows the positions and weights for three-point Gaussian quadrature. We can now calculate x' and y' which together make up the offset for each virtual particle being mapped, and we can also determine the values of w_v. Each permutation (with repetition) of the Gaussian quadrature points used maps to a single virtual particle; in the case of thee-point Gaussian quadrature this results in a total of $3^2 = 9$ virtual particles. Equation (5.4) shows the mapping used for 2D problems, and (5.5) shows the weight calculation. As a final step, the weights must be normalised to sum to one.

$$\vec{o}_v = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x\sqrt{1 - \frac{y^2}{2}} \\ y\sqrt{1 - \frac{x^2}{2}} \end{bmatrix} \tag{5.4}$$

$$w_v = w_x w_y \left(1 - \left(x'^2 + y'^2\right)\right) \mathbf{J} \tag{5.5}$$

For 3D problems we additionally define z, z', and w_z and carry out the mapping as shown below in (5.6) and (5.7). Each permutation (with repetition) of the Gaussian quadrature points continues to map to a single virtual particle, which for three-point quadrature results in a total of $3^3 = 27$ virtual particles. As before, the weights are normalised to sum to one.

$$\vec{o}_v = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x\sqrt{1 - \frac{y^2}{2} - \frac{z^2}{2} + \frac{y^2 z^2}{3}} \\ y\sqrt{1 - \frac{z^2}{2} - \frac{x^2}{2} + \frac{x^2 z^2}{3}} \\ z\sqrt{1 - \frac{x^2}{2} - \frac{y^2}{2} + \frac{x^2 y^2}{3}} \end{bmatrix} \tag{5.6}$$

$$w_v = w_x w_y w_z \left(1 - \left(x'^2 + y'^2 + z'^2\right)\right) \mathbf{J} \tag{5.7}$$

While the shape function of a particle is usually represented by a delta function when this is put into the weak form it has an action on all the bases of the element it occupies thus being equivalent to using piecewise linear shape

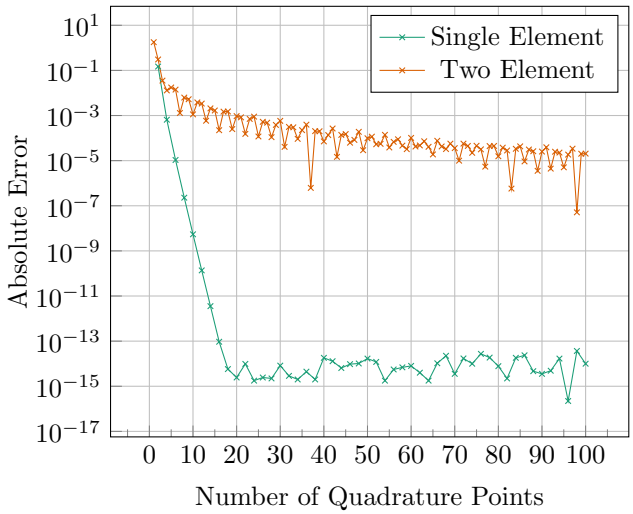


Figure 5.2: Graph showing how the absolute error of the modified integration converges with the number of quadrature points for a single element, and when spanning two elements.

functions in FDTD-PIC. As described above the delta function is extended to a quadratic shape with compact support on the specified radius r_0 which is numerically integrated against the test function representing the weak form of the currents or charge densities. As long as it is guaranteed that the weights sum to unity then the properties of charge conservation will continue to be maintained. Errors in the cubature can effectively be thought of as deviations to the shape function $S(r) = c(1 - r^2/r_0^2 + \epsilon f(r))$ where ϵ is the cubature error. Figure 5.2 shows that for a single element ϵ quickly converges to on the order of 10^{-15}, but much slower convergence is observed once a particle spans multiple elements. As long as ϵ is below the statistical convergence rate of $\sqrt{(1/N)}$ this error is expected to be small when compared to other terms.

Given the offsets and weights defined above, the implementation of the PIC algorithm can now be modified to leverage this new particle shape. One should note that the positions of the virtual particles do not need to be stored, it is sufficient to track the physical particle location and apply the assigned offset. The extension to virtual particles only changes the coupling between the particles and the mesh, making the extensions to the particle move trivial. The

modifications made to the PIC algorithm are now described in the subsequent sections.

5.2.1 Weighting of Fields to Particles

As the electric and magnetic fields are only known on the computational mesh, they must be interpolated from the mesh to the particles in order to be able to update the particle forces and velocities. PIC usually accomplishes this by applying the basis functions to determine the field values at specific particle locations as described in Section 3.3.2. In our algorithm we use the same approach to calculate these values at the position of the replicated virtual particles, and multiplying the field value by the virtual particle's associated weight. This can be expressed mathematically as shown in (5.8) and (5.9), where $\vec{x}_i$ is the physical position of particle i, and N_v is the number of virtual particles. Once this has been done for all virtual particles it is then trivial to accumulate each individual contribution to the central particle via summation.

$$\vec{E}(\vec{x}_i) = \sum_{v=0}^{N_v} \sum_{j=0}^{N_{edge}} E_j \hat{e}_j(\vec{x}_i + \vec{o}_v) w_v \tag{5.8}$$

$$\vec{B}(\vec{x}_i) = \sum_{v=0}^{N_v} \sum_{j=0}^{N_{face}} B_j \hat{b}_j(\vec{x}_i + \vec{o}_v) w_v \tag{5.9}$$

5.2.2 Particle Acceleration and Movement

Implementing a particle mover using the proposed modifications to the particle shapes as described is a relatively simple matter. This is due to the offset of each virtual particle used being fixed relative to the position of the central particle that is used to track the physical location. As in the standard PIC algorithm we apply the typical Boris Pusher in order to update the velocities of the central particles in the simulation (see Section 3.3.3 for details). Additionally, we specify that the surrounding virtual particles share the same velocity as their associated central particle, meaning that they do not need to be processed during the acceleration step.

5.2.3 Weighting of Particles to Grid

As described in Section 3.3.4, the particles are coupled to the grid and must therefore make contributions back to the grid prior to the field solve that will take place at the beginning of the next time-step. Informally, this can be thought of as each constituent virtual particle making its own separate charge or current contribution, scaled by its pre-calculated weight factor. These couplings take place as defined in (3.19) and (3.48) for charge and current, respectively. The implementation of the charge weighting for electrostatic problems using the extension to virtual particles is simple. As the virtual particle weights sum to one, the total amount of charge deposited will remain unchanged. We define this modified coupling below, using the same notation as defined previously. W_k is equal to the number of physical particles represented by simulation particle k.

$$\int_{\Omega_j} \rho \hat{v}_i \, dV = \sum_{k=1}^{N_P} W_k \sum_{v=1}^{N_v} w_v \hat{v}_i \left(\vec{x}_k + \vec{o}_v \right) q_k \tag{5.10}$$

A similar approach to that employed above can also be applied to the current weighting procedure with the difference that each virtual particle will make a contribution during its individual move, instead of all virtual particles making a deposit at the end of the move step. Additionally, deposits will be made to all elements crossed by the virtual particle during the move step. Specifically, the trajectory of each virtual particle is individually split as it passes through each element. The particle to grid coupling for current deposition using virtual particles is given below in (5.11). This remains analogous to each virtual particle making a separate weight-scaled current contribution to the grid. Along with the base code, we continue to use two-point Gaussian quadrature in the case of non-simplex elements. As the virtual particle weights sum to one and our base implementation conserves charge, this current deposition is also charge conserving.

$$\int_{\Omega_j} \vec{J}^{\,n+1/2}\, \hat{e}_i \, dV = \sum_{k=1}^{N_P} \Delta t W_k \sum_{v=1}^{N_v} w_v q_k \vec{u}_k \left(\vec{x}_k^{\,n+1/2} + \vec{o}_v \right) \cdot \hat{e}_i \left(\vec{x}_k^{\,n+1/2} + \vec{o}_v \right) \tag{5.11}$$

5.3 Results

In the following section we present results for four numerical experiments. These have been selected to be broadly representative of the problems that can be solved with EMPIRE-PIC. First, a simple 2D electrostatic electron orbit problem is examined. Second, the 3D simulation of a Transverse Electromagnetic (TEM) wave propagating through plasma is discussed. Third, we analyse the effect of our higher-order particle shapes on the amount of numerical heating observed. Finally, we look at a more complex electrostatic problem – the 1D expansion of a neutral plasma slab into a vacuum. For the results collected in the following experiments we used a virtual particle layout as defined in Section 5.2. We used five-point Gaussian quadrature resulting in 25 virtual particles for the 2D problems, and 125 virtual particles for the 3D problem. For the electrostatic problems we also examine the effects of particle smoothing when second-order basis functions are used. This analysis was not conducted for electromagnetics as higher-order basis functions are not currently available in EMPIRE-PIC for electromagnetic problems.

5.3.1 Electrostatic 2D Electron Orbit

We first consider the behaviour of our algorithm on a very basic electrostatic problem, consisting of a stationary H^+ ion being orbited by a single electron for one period. Using this simple test case we examine the effect of varying particle radius on the accuracy of the tracking of basic particle motion. The particles are situated on a quadrilateral mesh, the ion positioned at the centre, and the electron has an orbit radius of $r_{orbit} = 5.291 \times 10^{-8}$ m. The total

length of the domain in both x and y directions is equal to $3.0 \times r_{orbit}$, with initial $N_x = N_y = 14$. We also specify the problem boundary conditions to an analytical value defined as the exact value of the potential at the boundary: $\phi = \frac{q}{2\pi\epsilon_0} \ln\left(r^{-1}\right)$.

The initial conditions of the problem can be derived as follows. Given the electrostatic assumption, we can reduce the Lorentz force to $\vec{F} = q\vec{E}$. Then, from centripetal force and Gauss' Law we can write the following to obtain an expression for the electric field:

$$qE\left(r\right) = m\omega^2 r \tag{5.12}$$

$$\int \vec{E} \cdot \hat{n} \; dA = \int_V \frac{\rho}{\epsilon} \tag{5.13}$$

We are using an electron that does not deposit charge to the mesh in order to simplify the boundary conditions, due to having zero charge, but finite charge-to-mass ratio. Therefore, as the fields are not changed, the above can be simplified. We can now rewrite and substitute (5.12) and (5.13) in order to derive an expression for the angular velocity ω, and also velocity $v = \omega r$ which can then be resolved into its x and y components.

$$E\left(r\right) = -\frac{1}{2\pi r}\frac{q}{\epsilon_0} \tag{5.14}$$

$$\omega^2 = \frac{q^2}{2\pi m\epsilon_0} \tag{5.15}$$

Therefore we can define angular velocity $\omega = \sqrt{q^2/2\pi m\epsilon_0}$. As we know that $x = r_{orbit}\cos\left(\omega t\right)$ and $y = r_{orbit}\sin\left(\omega t\right)$ it is now trivial to compare the simulated orbit to every point on the trajectory defined by the analytical solution.

For the base case of this test we place the hydrogen ion at the centre of the mesh, directly on top of an element vertex. In order to avoid the special case (a particle will almost never occupy this position in an actual problem), we repeat the test placing the central particle at 100 randomised positions within the element quadrant. As a result of all cell quadrants being identical, we can obtain

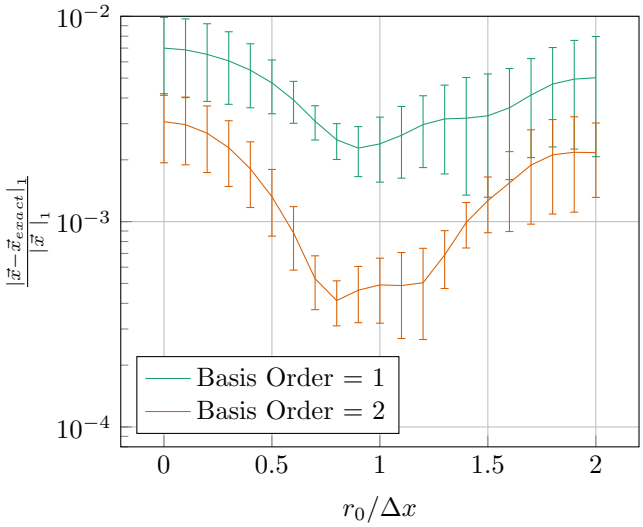


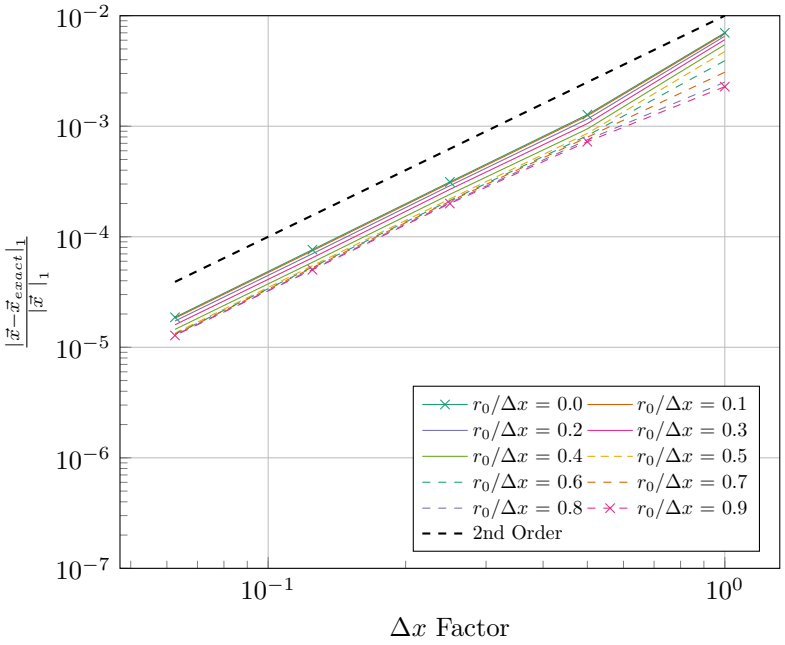
Figure 5.3: Graph showing results for the 2D electron orbit experiments on the structured mesh. Error bars represent one standard deviation in the L_1 norm due to variation in the starting locations.

data that consider a representative range of possible particle positions within an element. In the remainder of this section we examine the effects of increasing particle radius on the L_1 error of the position of the orbiting electron against the analytical solution (normalised via the orbit radius), and also consider these effects at increased levels of problem refinement, where we hold the ratio $\Delta x/\Delta t$ fixed in order to maintain a constant CFL value. We first examine the effects of increasing particle radius for the base level of mesh refinement, consisting of 14 elements in both dimensions as defined above. In order to definitively rule out the influence of time integration on the orbit error due to large time-step size, we present data collected using a refined Δt to ensure that the improvement due to smoothing is visible. In this case, we use 320 time-steps per electron orbit. Figure 5.3 shows how the L_1 error varies as particle radius is increased over various fractions of the cell size Δx. The error bars are used to represent the standard deviation in the error due to the position of the hydrogen ion in the element quadrant. After the initial radii, it is clear to see that as the particle radius is increased the computed answer moves much closer to the analytical solution, with a radius value of $0.9 \times \Delta x$ appearing to be optimal in this case.

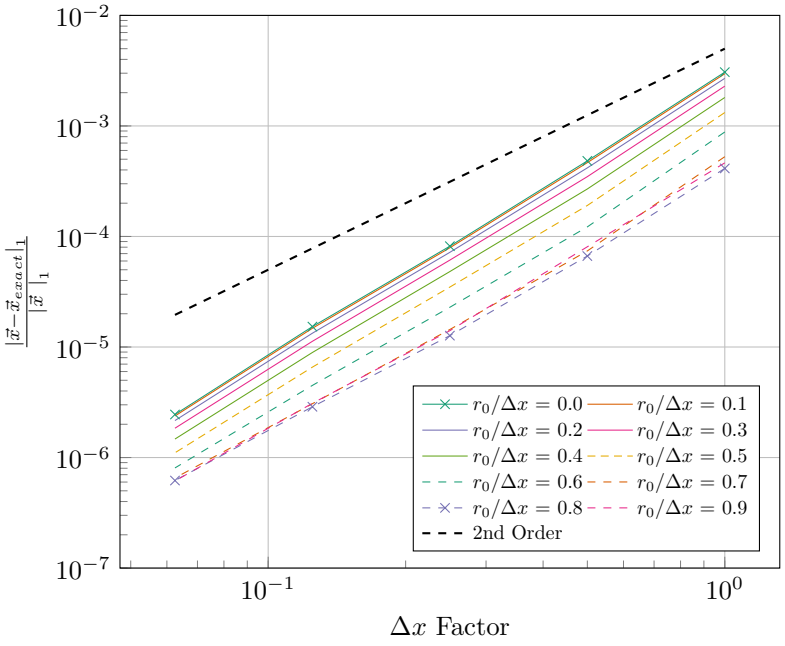
We also observe a significant reduction in the standard deviation due to altering the position of the central H^+ ion within its quadrant, i.e., the level of statistical noise is lower. However, this improvement in error and statistical variation is reversed as particle radius continues to increase. From this we can conclude that some smoothing of the charge distribution of the particle improves the ability of the PIC algorithm to track basic particle motion, whereas excessive smoothing has a detrimental effect.

Using second-order basis functions results in a 50% improvement over the base code in terms of error and standard deviation when no smoothing is applied. When smoothing is applied the results follow a similar trend to the first-order basis data, with an optimal radius value of $0.8 \times \Delta x$. This error value is significantly lower than the equivalent data point for the first-order basis. In the best case the electron position error is approximately 85% lower than the base code, and the standard deviation in the result is reduced by an order of magnitude.

We now examine the effect of mesh refinement on this problem via a convergence study, with a base level of $\Delta x = 1.13 \times 10^{-8}$ m, $\Delta t = 4.615 \times 10^{-11}$ s and $N_x = N_y = 14$ grid elements. This results in a CFL condition value of $\vec{v}\Delta t/\Delta x \approx 0.0742$. Figure 5.4 shows the results of this study for various particle radii, using the L_1 norm of the electron position as the error metric, as in the previous test. As seen in Figure 5.4(a), a particle radius of approximately $0.9 \times \Delta x$ appears to be optimal for the majority of refinement levels used for the convergence study when considering the first-order basis. Figure 5.4(b) shows that when using a second-order basis the optimal radius remains consistent as the problem is refined. These results show a consistent improvement in the L_1 norm across a wide range of Δx values, consistent with the previous results for the coarse mesh. The error reduction appears to be approximately a stable factor of two when comparing results for the vanilla code against runs using the optimal radius value for the first-order basis, and a factor of seven for the second-order basis. It is also evident from these results that the use of smoother particles causes earlier solution convergence then the standard PIC algorithm,



(a) Basis Order = 1



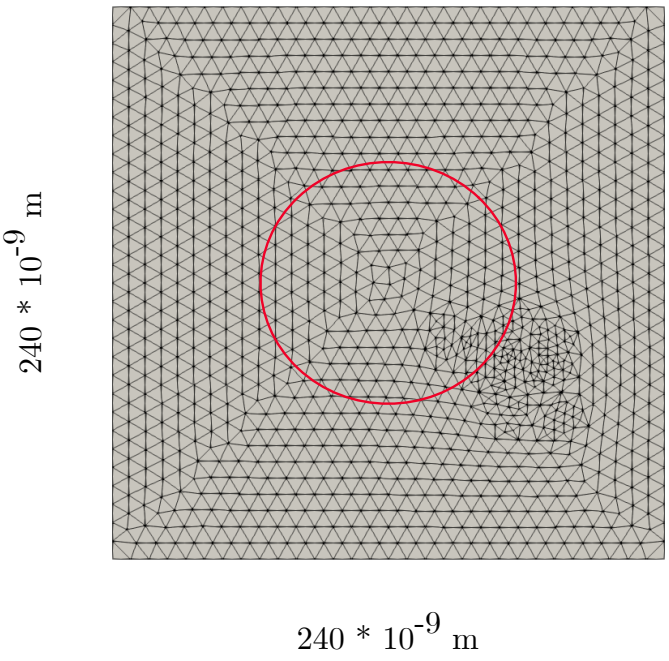
(b) Basis Order = 2

Figure 5.4: Convergence study results for the 2D orbit problem on the structured mesh.

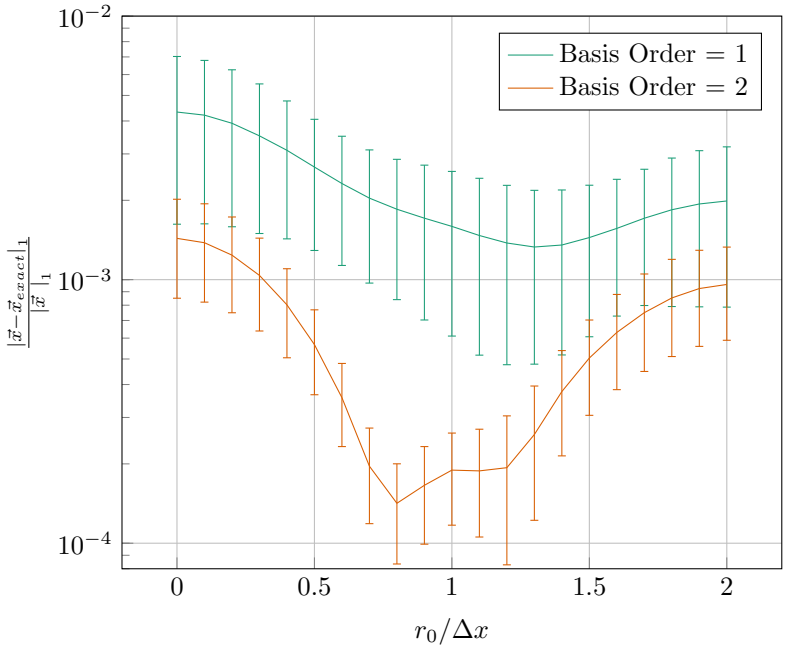
indicating that this may provide acceptable convergence rates while allowing the use of coarser meshes which are less computationally expensive.

Finally, in order to assess the benefits of particle smoothing for non-regular grids, we carried out an additional parameter scan over $r_0/\Delta x$ using an unstructured mesh of 2272 triangular elements. This mesh and its dimensions are shown in Figure 5.5(a), with the base orbit trajectory shown in red. The mesh has an average Δx value of approximately 7×10^{-9} m. As with the previous experiments, data was collected for each input using 100 randomised starting locations, this time varying the starting position by at most $\pm 0.5 \times r_{orbit}$ m in each dimension. In this way we can determine the variation in the result due to the electron travelling through various levels of mesh distortion. We now use a refined time-step of 640 steps per orbit in order to rule out time integration error[1]. Figure 5.5(b) shows the results of this experiment, with error bars again representing 1 standard deviation in the error due to the variation in orbit position. As in the radius scan experiment that was conducted for the structured mesh we again see a smooth reduction in average error as particle radius is increased. In this case we have an optimal value of $r_0/\Delta x = 1.3$, suggesting that greater amounts of smoothing may be beneficial on a distorted mesh. However, due to the high level of mesh distortion the improvements in the standard deviation are less significant. A similar trend is observed for the second-order basis where improvements are visible, but more pronounced than for the first test. Using the second-order basis causes the optimal amount of smoothing to become similar to the results in Figure 5.3. In the best case both the electron position error and standard deviation are reduced by an order of magnitude. From these results it is clear to see that the altered algorithm is capable of coping with such varying distortion, particularly when using a second-order basis.

[1]The increase in step count serves to maintain a similar CFL to the structured mesh tests.



(a) Mesh used for unstructured orbit tests. The orbit base case is shown in red.



(b) Parameter scan of particle radius for the unstructured orbit tests

Figure 5.5: A parameter scan where (a) represents the geometry being studied and (b) shows the L_1 norm of the error in the electron position.

5.3.2 3D Transverse Electromagnetic Wave

To test the performance of our algorithm for 3D and electromagnetic problems we now consider an infinite, planar TEM wave propagating through an infinite neutral plasma made up of H^+ ions and electrons. This problem was chosen as an electromagnetic case study and has an analytical solution, given certain assumptions. The solution is given in Chen [38], which derives the differences between a TEM wave in a vacuum and a TEM wave in a plasma where the wave vector is held constant.

In this problem we choose the key controlling parameters as follows: we have the plasma number density as $n_0 = 10^{15}$ m^{-3}, initial temperature of 0 K, with a maximum electric field magnitude of $E_{mag} = 100$ V/m, and the vacuum frequency of the wave $f_v \approx 1.420$ GHz, and $\omega_v = 2\pi f_v$. Additionally, we assume that the electromagnetic wave is of such a high frequency that the ions within the plasma remain stationary throughout the simulation, and also that the $\vec{J} \times \vec{B}$ forces on the particles are negligible. This has the effect that electrons are assumed to only oscillate linearly in the plane of the electric field. We have the plasma frequency and actual wave frequency as follows:

$$\omega_p = \sqrt{\frac{n_0 q^2}{m_e \epsilon_0}} \approx 1.784 \times 10^9 \text{ rad/s} \tag{5.16}$$

$$f = \frac{\omega}{2\pi} = \frac{1}{2\pi}\sqrt{\omega_p^2 + \omega_v^2} \approx 1.448 \text{ GHz} \tag{5.17}$$

As the wave is an infinite, steady wave, we can derive the constant phase velocity:

$$v_p = \sqrt{\frac{f}{f_v}} c \approx 1.02c > c \tag{5.18}$$

This gives the maximum initial electron velocity as defined below, which is then initialised in phase with the electric field. The values of v_x and v_z are initialised to zero.

$$v_y = \frac{qE_{mag}}{m_e \omega} \approx 1932.5 \text{ m/s} \tag{5.19}$$

The velocity $\vec{u}$ of a given particle can now be calculated as follows, where p is the particle position projection onto the wavevector $(0, 0, 1)$:

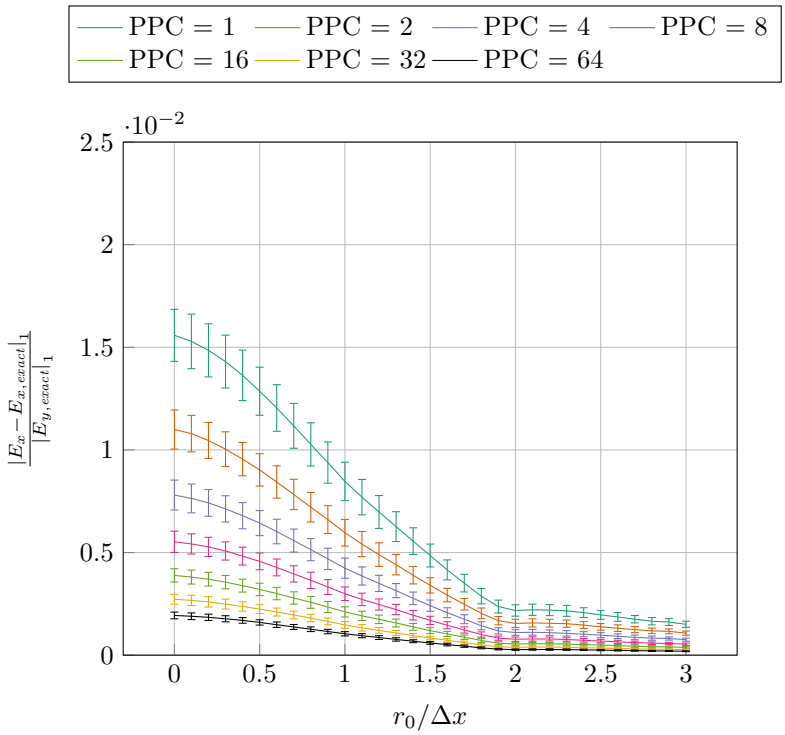
$$\vec{u} = \vec{v} \sin\left(p + \frac{\pi}{2}\right) \text{ m/s} \tag{5.20}$$

Finally, we define the maximum magnitude of the magnetic field such that it is congruous with the magnitude of the electric field.

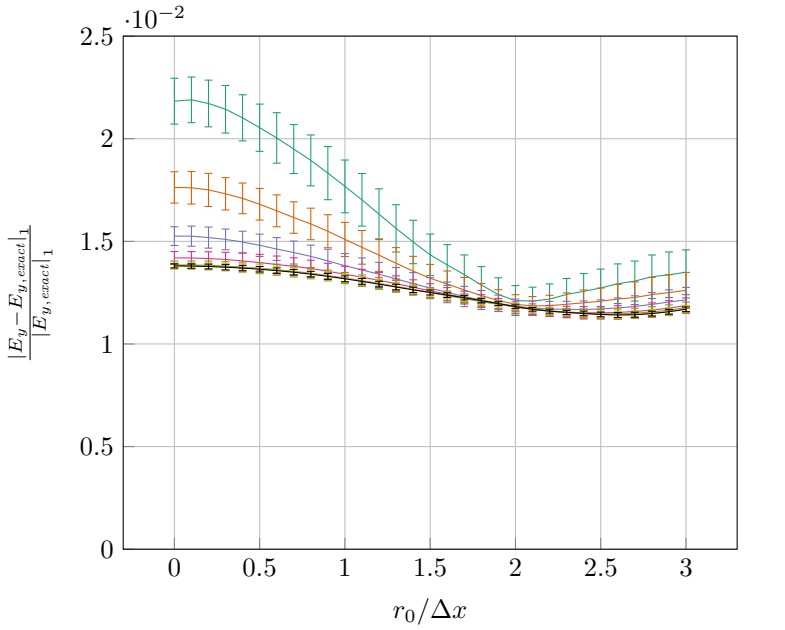
$$B_{mag} = \frac{\lambda}{2\pi} \frac{E_{mag}}{c^2} \left(\frac{n_0 q^2}{m_e \epsilon_0 \omega} + \omega\right) \approx 3.53 \times 10^{-7} \text{ T} \tag{5.21}$$

Using the derivation above it is simple to formulate a computational description of the problem. We set up the problem on a three-dimensional grid of hexahedral finite elements with periodic simulation boundaries in all directions, effectively creating infinite space for the TEM wave, which we simulate for one wave period. The wave is defined to travel in the z dimension of the computational mesh, with the majority of grid elements also in the z dimension. The x and y dimensions are each defined to have a constant 4 elements, while the z dimension has 24 elements. As we assumed the ions to be stationary in our derivation, we force them to remain immobile during the simulation. The computational particles are placed randomly within each element and weighted in order to achieve our previously specified plasma number density. Each cell is loaded with an equal amount of particles of each species. We additionally ensure that the initial electron velocity is confined to the transverse direction in the plane of the electric field.

We now present results for this problem for a variety of Particles per Cell (PPC) counts, showing the average of 100 runs using random initial particle loads, using error bars to represent the standard deviation in the data. Figures 5.6 and 5.7 show the effect of increased particle radius on the average L_1 error of the simulated electric and magnetic fields at the end of the simulation, presented as a breakdown of the field components. As the problem is set up with plane wave polarisation with only non-zero E_y and B_x, we refer to these com-

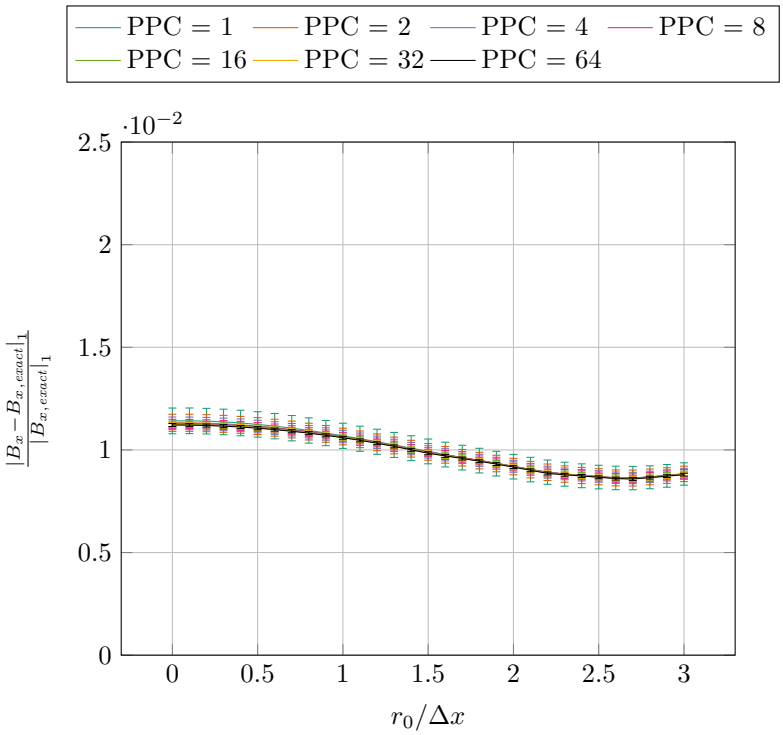


(a) E_x component

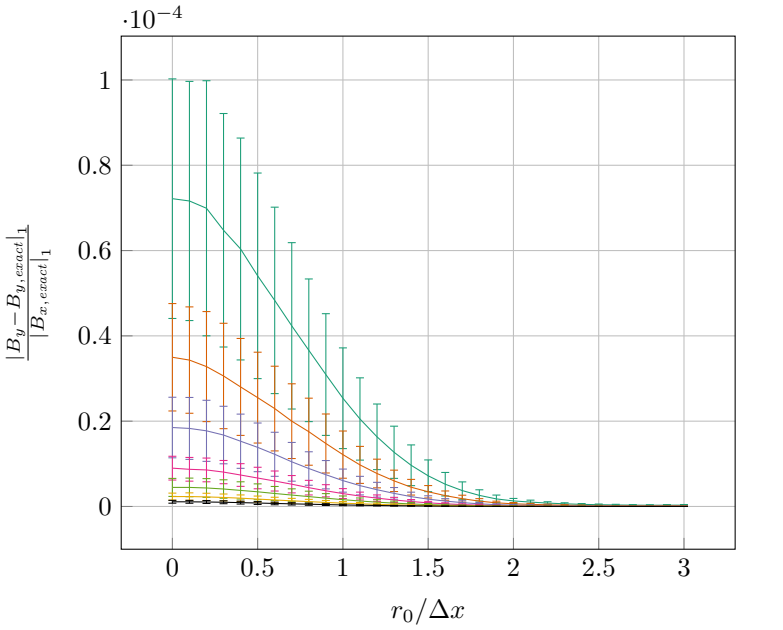


(b) E_y component

Figure 5.6: Graphs showing variation in L_1 norm of electric field components as particle radius is increased.



(a) B_x component



(b) B_y component

Figure 5.7: Graphs showing variation in L_1 norm of magnetic field components as particle radius is increased.

ponents as the signal components, and the remaining components as non-signal components. As each of the non-signal components for a given field behave in the same manner, we choose to show data for E_x and B_y for these components, and E_y and B_x for the signal components. At first it is clear that we observe a smooth reduction in the L_1 error of the electric field which is reflected in the results shown for both the signal and non-signal components. This improvement continues to occur beyond the previously optimal value of $r_0 = 0.9 \times \Delta x$ observed in the orbit problem, showing no signs of stopping even as the value of $r_0/\Delta x$ approaches 1.5. Additionally, we see a slight overall reduction in the variation from the initial seeds, but this effect appears to be negligible. It should be noted that, especially for low particle per cell counts (specifically 1 and 2), the maximum benefit from particle smoothing leads to a greater reduction in L_1 error than doubling the amount of particles per cell for the zero radius case. Also of interest is that the error reduction due to smoothing for the non-signal field components is much greater than that observed in the component that contains the wave itself, suggesting that the noise in the wave is more sensitive to the particle coupling used. These differences are apparent in Figures 5.6(a) and 5.6(b). Of particular note is that the E_y error converges to an approximate value of 0.014, whereas the other components continue to improve by tending towards zero at higher particle counts. We therefore conclude that the remaining E_y error is due to error in the scheme, and can be reduced by refining the problem further in space and/or time. This was verified through additional convergence tests, where we found that the remaining error decreases in line with second-order convergence as the problem is refined.

Secondly, we examine the effects of particle smoothing on the computed result for the magnetic field. In accordance with the electric field data, we see a smooth reduction in L_1 error for both the signal and non-signal magnetic field components as particle radius is increased. There is good reduction in the B_x error, particularly as increasing the number of computational particles per cell has a negligible effect when compared to smoothing. However, the same does not

hold true for the non-signal components where both smoothing and increasing particle count show good results, with smoothing performing particularly well at low particle counts. At higher particle counts smoothing reduces the error in these components to near zero. Regarding the statistical noise shown by the error bars, the B_x component shows almost no reduction in noise, in keeping with the trend observed regarding the electric field. Interestingly, the opposite holds true for the B_y component, showing a large reduction in statistical noise as radius is increased.

In general we conclude that, for this problem, the application of particle smoothing has the primary effect of reducing the noise in the solution for both the electric and magnetic fields in various ways. Specifically, where the solution should be zero there is a large reduction in the error in these components, and where the solution should be non-zero the errors converge to a seemingly constant value representing the space and time errors.

As a final note, we also examined the effect of smoothing on the frequency distribution of the error in the final result by applying a Fast Fourier Transform (FFT) to the E_y component of the electric field. However, we do not show these results in this thesis as there appears to be little to no observable effect, beneficial or otherwise, on the resultant frequency distribution for this problem.

5.3.3 Numerical Heating

It is well documented that PIC codes are particularly susceptible to a phenomenon known as *numerical heating*, which leads to an erroneous growth in the Kinetic Energy (KE) of the system over the course of a simulation. This has previously been studied in detail by various authors [72, 75, 93], and is particularly prevalent in momentum conserving schemes such as that employed in EMPIRE-PIC [24]. This heating is typically controlled by three factors: (i) cell size, (ii) time-step size, and (iii) the number of computational particles used in the simulation. It has also been shown that the use of higher-order weighting schemes can significantly suppress such heating, even in cases where the De-

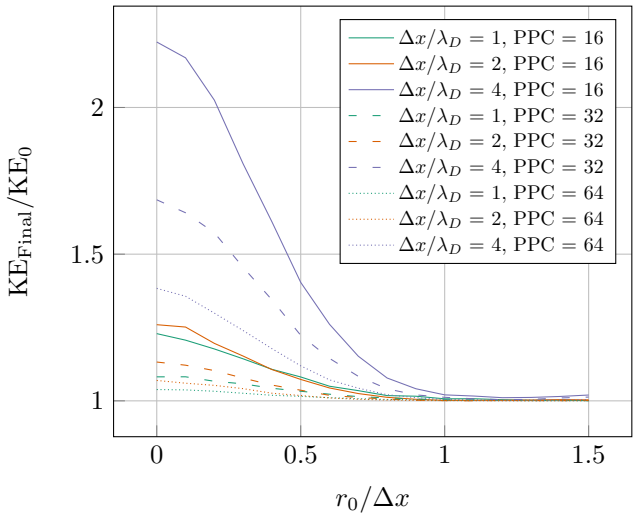
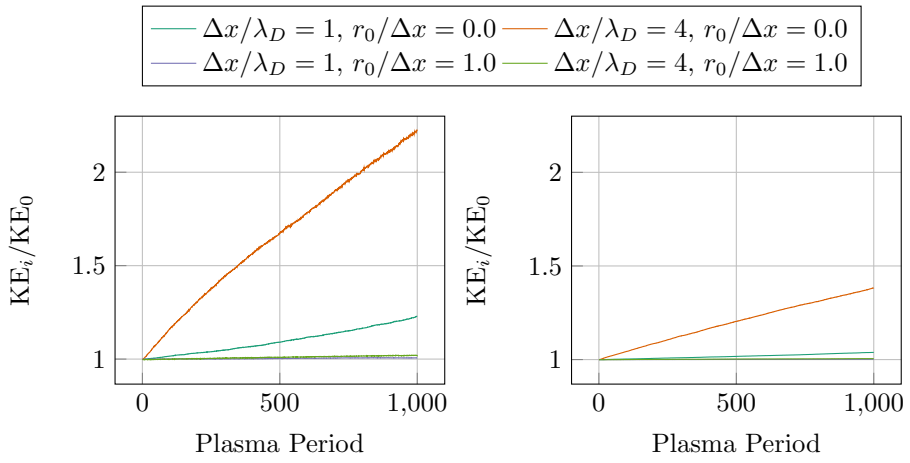


Figure 5.8: Ratio of final kinetic energy to starting kinetic energy for various particle radii.

bye length is not completely resolved by the spatial grid [126, 135]. We now present our findings from numerical heating experiments within EMPIRE-PIC, with and without using the implemented higher-order particle shapes presented in this chapter. To this end, we examine the total KE of a neutral plasma consisting of electrons and hydrogen ions over 1000 plasma periods, at an initial temperature of 1.0 eV. Therefore we derive the key parameters of this problem as follows. We chose a number density of $n_0 = 10^{15}$ m^{-3} resulting in a plasma frequency $\omega_p \approx 1.784 \times 10^9$ rad/s, assuming the thermal motion of the electrons can be ignored. Computationally, we use a 16×16 mesh of triangular elements with periodic boundaries in x and y, 10 time-steps per plasma period, with various amounts of particles per grid element and a range of particle radii. We also keep the amount of grid elements fixed, instead altering the size of the problem domain in order to determine the ratio between the Debye length and the cell size, Δx.

Figure 5.8 shows the variation of the growth in simulation KE as the ratio of particle radius relative to Δx is increased, for problems using 16, 32, and 64 computational particles per cell. Additionally, the ratio of Δx to the Debye



(a) Using 16 particles per cell

(b) Using 64 particles per cell

Figure 5.9: Kinetic energy change over time for the vanilla code vs the optimal particle radius, for resolved and under-resolved λ_D.

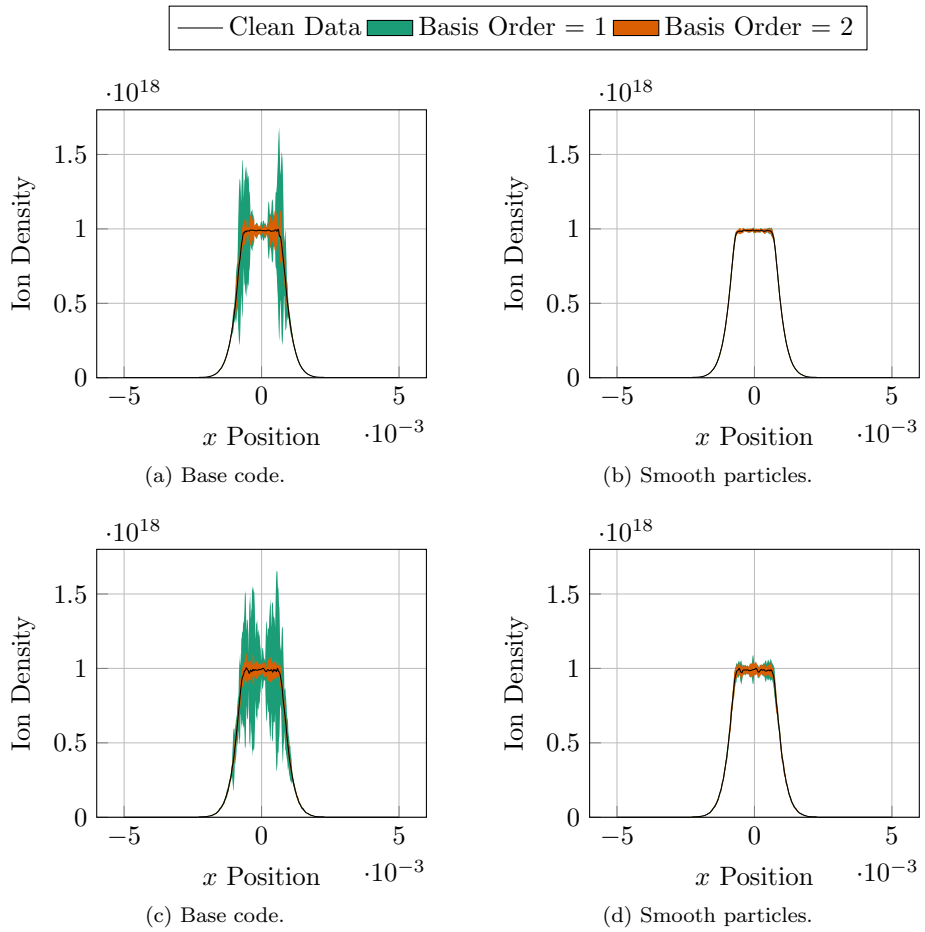
length is set at one of three levels: 1, 2, or 4. As expected, when the Debye length is severely under-resolved we observe large increases in the overall KE at the end of the simulation against that at the beginning. However, we see that such growth rapidly decreases as the particles are made smoother, particularly for the $\Delta x/\lambda_D = 4$ case. Interestingly, as we approach the optimal radius of $r_0 = \Delta x$, we observe very little difference in the growth of KE for the problems with $\Delta x/\lambda_D \leq 2$ at both 16 and 32 particles per cell. This is promising in terms of performance, as we can maintain similar KE stability while using less grid cells and super-particles, reducing both computational requirements and load on the memory system.

The results are further validated in Figure 5.9, which shows the growth of the system KE at each simulated plasma period for values of $\Delta x/\lambda_D$ of 1 and 4, for the vanilla code and the optimal radius value of $r_0 = \Delta x$, using 16 and 64 particles per grid cell. Where the Debye length is under-resolved we observe extremely rapid growth in the KE of the system, increasing by 50% in approximately 300 periods for the 16 particle per cell case. In the case where $\Delta x/\lambda_D = 1$, the smoothed particles all but eliminate the numerical heating effects, with only very mild growth throughout the simulation. The benefits

are also significant for the under-resolved case, with optimal particle smoothing resulting in a heating trend similar to that of the base code with a resolved mesh. Additionally, these results remain consistent for Figure 5.9(b), exhibiting good reduction in heating effects. Again, the smooth particles almost eliminate self heating where λ_D is resolved, and continue to show good performance on an under-resolved mesh – on par with the $r_0/\Delta x = 0$ results in the resolved case.

5.3.4 Electrostatic Plasma Slab Expansion

In order to properly assess the behaviour of our algorithm for electrostatic PIC simulations it is prudent to examine a more complex test case than the simple orbit discussed in Section 5.3.1. We now consider the 1D expansion of a collisionless slab of plasma into a vacuum, a benchmark problem that has previously been used for verifying PIC simulations [41]. As EMPIRE-PIC is a 2D/3D PIC code, it cannot be used to directly simulate an entirely 1D problem. We therefore set up a 2D mesh with fixed $N_y = 2$, using periodic simulation boundaries for the y direction, and quadrilateral elements. We use a Dirichlet boundary condition in the x direction, setting the electric potential to zero in order to ensure that the problem is well-posed. The problem starts with a charge neutral slab with a thickness of 2 mm placed at the centre of a domain of length 1 cm, allowed to expand for a total time of 2.5×10^{-9} s. The ions are initialised cold, whereas the electrons are assigned a finite initial temperature of 1 eV. Additionally, the ions have a mass of $10 \times m_e$. We choose such an artificially low ion to electron mass ratio in order to accelerate the expansion of the plasma slab. Each grid cell of the simulation that contains plasma is initially loaded with 8000 particles of both species, weighted such that we achieve a plasma number density of $n_0 = 1 \times 10^{18}$ m^{-3}. Given these parameters, we can now derive the plasma frequency and Debye length as follows. $\omega_p \approx 5.641 \times 10^{10}$ rad/s, and $\lambda_D \approx 6.89 \times 10^{-6}$ m. This allows us to choose a base N_x such that $\Delta x/\lambda_D \approx 1$, and Δt such that $\omega_p \Delta t < 0.1$. We now have $N_x = 1600$, $N_y = 2$, and $N_{\Delta t} = 250$.



(a) Base code.

(b) Smooth particles.

(c) Base code.

(d) Smooth particles.

Figure 5.10: Graphs showing the noise in the simulated ion density for the vanilla code and smoothed particles. (a) and (b) use 8000 particles per cell, (c) and (d) use 800 particles per cell. Smooth particles use $r_0/\Delta x = 1$.

We now show results for the simulated cell-centred ion density for this problem for the base code, and for the smooth particle implementation with $r_0/\Delta x = 1$. The number of computational particles used per cell is kept constant for all runs at $\mathrm{NP}_{cell} = 8000$ as specified above. As the density solution output by the vanilla code is extremely noisy, we filter the data using a one-dimensional Gaussian filter with $\sigma = 3$. We display the error in the ion density as a shaded area, which shows the standard deviation in the raw data used to generate a given filtered point.

Figure 5.10 shows the results of these experiments. It is clear to see from

Figure 5.10(a) that the vanilla code exhibits a very high amount of noise in the simulated ion density, with most of this noise building up at the interface between the slab of plasma and the vacuum. It is also evident that the use of a second-order basis can reduce this noise. Figure 5.10(b) shows the results of the same experiment for the smoothed particle representation. The magnitude of the noise in the solution is greatly reduced by particle smoothing, both at the interface and in the centre of the plasma slab to the point of being only marginally visible. Close inspection reveals that the second-order basis continues to outperform the first-order basis.

These experiments were also repeated using 800 particles per cell. These results are shown in Figures 5.10(c) and 5.10(d). When comparing Figure 5.10(a) to Figure 5.10(d) we can see that using 800 smoothed particles produces a result that is significantly less noisy than when 8000 traditional particles are used for both basis orders. This is significant as we can use an order of magnitude less particles while also maintaining a greatly improved solution over the base code.

We also examined the error in simulated electric potential for both a resolved and under-resolved Debye length. This error did not appear to be sensitive to particle smoothing in this case. This result is interesting as it suggests that the large reduction in density noise has negligible effect on the simulated potential.

5.3.5 Comparison to Other Schemes

It is also prudent to compare our implementation to that of other higher-order weighting schemes, as traditional implementations weight the particles to all grid cells within the range of the particle shape function, whereas we only handle elements in which a virtual particle resides. The method proposed by Jacobs and Hesthaven [71, 82] tests a similar shape function to the one used in our algorithm (see (5.2)) for discontinuous Galerkin FEM-PIC, as distinct from the continuous Galerkin algorithm of EMPIRE-PIC. A key difference between the implementation in EMPIRE-PIC versus the Jacobs-Hesthaven method is that our approach is charge conserving from first principles, whereas Jacobs-Hesthaven requires an

additional correction procedure to be executed every time-step in order to ensure Gauss' law remains enforced. In terms of error reduction there is little difference between the two schemes – this is unsurprising given the similarity in the shape functions that are used. Of particular note is that both implementations demonstrate near elimination of numerical grid heating, even in cases where the Debye length is poorly resolved by the computational mesh. This analysis also applies to the cell mean value approach presented by Stindl et al. whose results generally agree with those of Jacobs and Hesthaven [81, 139].

It is difficult to directly compare the modified EMPIRE-PIC scheme with the charge conserving particle-to-grid weighting method introduced by Pointon [123], as the latter only considers structured grid meshes of regular cells. However, comparisons can be made again with regards to numerical heating, and also the slab problem (see Section 5.3.4). As with Jacobs-Hesthaven, this method also demonstrates greatly reduced heating on low resolution meshes. However, of particular interest is that for a similar electrostatic slab problem, Pointon also observes that a higher-order weighting scheme is often required to achieve a simulation that is not dominated by noise. This result largely agrees with the data shown in Figure 5.10.

Thirdly, we consider the moving least squares weighting approach included in the HOPIC method presented by Essex and Bridson [53]. The most notable difference is that HOPIC converges to analytical solutions much more quickly than EMPIRE-PIC. This is unsurprising given that the scheme in HOPIC uses a fourth-order accurate Runge-Kutta method – recall that EMPIRE-PIC is second-order accurate in both space and time. Additionally, the particle shape function used by Essex and Bridson is also of a greater order, namely a cubic polynomial. However, the modified EMPIRE-PIC remains competitive in terms of numerical heating, seeing similar results to HOPIC. Finally, HOPIC provides no guarantees on the conservation of charge.

Finally, we note that all of the studies included in the above comparison provide little detail regarding the effect of higher-order particle shapes on the

performance of their associated PIC applications, meaning that a meaningful cost-benefit comparison cannot be made to other methods. Therefore, we cover this specific issue for EMPIRE-PIC in Chapter 6, where we provide a detailed performance study, and examine the trade-offs between the increased computation time and solution accuracy brought about by the algorithmic modifications that have been made.

5.4 Summary

As the need to simulate the behaviour of plasmas under various conditions using complex geometry continues to grow, PIC algorithms must adapt to these changing requirements. As a result, both higher-order PIC methods and the use of unstructured FEM-PIC has become an area of great interest to the plasma simulation community. While higher-order methods show promise, they also impose the additional requirement that the particles being simulated possess some smooth shape instead of the usual Dirac delta function.

In this chapter, we have proposed a higher-order representation of particles in PIC algorithms, where each particle has a smooth shape function with compact support on some finite radius, and implemented this representation in EMPIRE-PIC. A unique feature of our approach is that the implementation of this smooth representation is achieved by surrounding super-particles with delta shape computational virtual particles that have fixed offsets and weights derived from Gaussian quadrature rules. As this moves the quadrature from the mesh to a set of points surrounding the particle we can use the same PIC procedures as the base code with minimal modifications. The applications of this representation are broad as the offsets and weights may be tuned to represent any desired shape of the particle cloud.

The accuracy of the modified algorithm was examined and compared to the behaviour of the base code. Our results show approximately 70% improvement in the tracking of basic particle motion on a distorted mesh, with this increas-

ing to an order of magnitude improvement when a second-order basis is used. We additionally show extremely successful suppression of numerical heating for both resolved and under-resolved grids, and a significant reduction in noise of the simulated ion density in an electrostatic plasma slab expansion while being able to use an order of magnitude less super-particles. We have also demonstrated that, for the selected problems, the optimal particle radius appears to be problem-dependent. Consequently, the work presented in this chapter represents a step towards more accurate PIC applications, enabling improved simulations of plasma phenomena.

CHAPTER 6

Performance of Higher-Order Particle Representation

So far in this thesis we have considered how an unstructured Particle-in-Cell (PIC) code can be implemented in a C++ template-based performance portability library, namely the Kokkos framework. In Chapter 5 we examined an algorithmic change to particle representation in the FEM-PIC algorithm that offers both improved solution accuracy and increased floating point intensity for the particle-based kernels of EMPIRE-PIC. In this chapter we aim to tie together the work previously presented in this thesis by analysing and quantifying the performance of the modified FEM-PIC algorithm such that it can be compared and contrasted to the base EMPIRE-PIC implementation. As we continue to make use of Kokkos as our parallel programming model, we conduct this analysis for the hardware types previously considered in Chapter 4, specifically: Intel Central Processing Units (CPUs), Intel's Xeon Phi Knights Landing (KNL), ARM ThunderX2, and NVIDIA Tesla Graphics Processing Units (GPUs).

6.1 Implementation

In order to reduce the engineering burden of implementing the modified PIC algorithm in Kokkos, we choose to extend the existing particle-based kernels that were documented in Chapter 4. As EMPIRE-PIC is a production code under constant development, this also allows us to examine the change in performance behaviour as a result of our algorithmic modifications in isolation from any changes made in the primary codebase. This section documents the implementation of the modified EMPIRE-PIC kernels using Kokkos, and quantifies the amount of additional memory required in order to use virtual particles.

6.1.1 Additional Memory

While a key feature of our modifications to the algorithm in EMPIRE-PIC is increased floating point intensity, this does come at the cost of some additional data being stored in memory. The first of these is the offsets used to locate the virtual particles relative to their central particle, and the weights used to scale their field accumulations, and charge or current depositions. As these offsets and weights are currently fixed and shared by all simulation particles, they can be precomputed and stored prior to the main time loop of a run. Assuming a three-dimensional case this results in $8 \times N_v$ additional bytes for the weights, and $24 \times N_v$ additional bytes for the offsets, where N_v is the number of *virtual* particles used per simulation particle. Therefore, if five-point Gaussian quadrature is used (as is the case in the experiments in Chapter 5) the total additional memory used for this data is approximately four kilobytes. As particles in the base code consist of 125 bytes, this is equivalent to using an additional 33 particles, an almost insignificant increase in the context of the size of a typical problem. Additionally, as the offsets and weights will be frequently used by all threads during execution, they are likely to remain in cache for a significant proportion of kernel runtime. As a result, a significant number of the reads of these values will result in cache hits, thus improving performance.

It is also necessary to track the containing element of each virtual particle (see Section 6.1.3) throughout a simulation. This results in storing a single integer per virtual particle, per particle, i.e., $4 \times N_v N_p$ extra bytes, where N_p is the number of *simulation* particles. It now follows that when $4 \times N_v \approx 125$ the memory increase from virtual particles is the same as using a factor of two more simulation particles. Looking more closely, this result can be used to roughly quantify the increase in the ratio of Floating-Point Operations per Second (FLOP/s) performed per byte moved from DRAM due to the use of virtual particles. From the above expression we can see that using approximately 32 virtual particles per simulation particle increases the amount of operations by a factor of the same amount, while only increasing memory pressure by a

factor of two, i.e., a 16× growth in the ratio of FLOP/s per byte.

6.1.2 Weighting of Fields to Particles

Both flat and hierarchical Kokkos approaches were considered for the interpolation of the mesh data to the particles. As the collation of the contributions from a given simulation particle's virtual particles can be expressed as a reduction over the virtual particles, it is possible to perform a nested `parallel_reduce` for each particle to carry out this operation. In initial tests a flat approach performed best across all hardware due to additional overhead introduced by the reductions so we use a flat kernel in the final implementation. Finally, for CUDA builds, we use the team scratch memory exposed by Kokkos to store any common values in GPU shared memory. This includes the particle weights and offsets, the velocity and position of the central particle associated with its virtual particles, and the electric and magnetic field data.

6.1.3 Particle Move

As detailed in Section 5.2.2, no modifications to the acceleration step are necessary. However, despite having a fixed location relative to a given central particle, it is not possible to ignore the virtual particles during the position update. Firstly, in an electromagnetic PIC simulation, each virtual particle must make a current contribution to any cells it crosses during the move step, as is the case with a standard PIC super-particle. Secondly, as EMPIRE-PIC is an unstructured code, it is necessary to update the values of the array that stores the element a virtual particle currently resides in. While it is possible to locate the new element number manually, initial experimentation determined that it is cheaper computationally to move the virtual particles along with their associated central particle, and store this result in a Kokkos view of size $N_p \times N_v$.

Regarding the implementation of the modified move kernel using Kokkos, for OpenMP builds we continue to use a flat `parallel_for` approach, as the use of hierarchical parallelism resulted in no change in kernel performance, beneficial

or otherwise. For builds using the Kokkos CUDA backend, we again make use of the hierarchical approach detailed in Section 4.1.3. However, we are now able to use the full three levels of nesting: chunks of particles are assigned to teams; the threads of each team handle process particles contained in the chunk, and the third (vector) level loops over the virtual particles. Additionally, we continue to use Kokkos scratch memory to store any common values in GPU shared memory when compiling for CUDA.

6.1.4 Weighting of Particles to Grid

Apart from the the additional processing of the virtual particles, the charge and current deposition kernels are largely unchanged from those described in Section 4.1.4. We continue with a data replication approach for OpenMP builds, and atomic writes for CUDA builds. However, the K40 must now also use atomic writes for charge deposition in electrostatic simulations, as the assumption that all charge will be deposited to the element of the simulation particle no longer holds; virtual particles may occupy any of the surrounding elements. The resultant performance penalty means that it is not viable to use virtual particles on this hardware.

6.2 Experimental Setup

In order to examine the performance of our modified algorithm we must compare its behaviour to that of the base EMPIRE-PIC implementation for both electrostatic and electromagnetic FEM-PIC schemes. To this end we make use of two problems used earlier in this thesis, both for convenience and continuity. For electrostatics we consider the the expansion of a collisionless slab of plasma into a vacuum, the derivation and formulation of which can be found in Section 5.3.4. As before we consider a mesh of size 1600×2 grid cells, simulated for 250 time-steps. We also repeat the experiments at several different particle counts, in order to investigate both how performance changes with problem size.

Intermediary problem sizes are generated by increasing the number of particles by a factor of two, where 2^{15} particles is the base case. When using virtual particles we choose a radius of $r_0 = \Delta x$ as this gives the optimal result in terms of error for this problem (see Section 5.3.4).

For electromagnetics we consider the three-dimensional simulation of the Transverse Electromagnetic (TEM) wave propagating through a plasma, as derived in Section 5.3.2. In our analysis in this chapter we modify the parameters of this problem by refining the simulation such that 200 simulation time-steps are carried out, with a grid of dimensions $100 \times 4 \times 4 = 1600$ grid cells. The reasons for this are twofold: (i) the previous problem was small enough that there is a realistic possibility that the entire problem would fit in CPU cache regardless of particle count, resulting in unrealistic performance results, and (ii) refining the problem in space and time maximises the visibility of error reduction due to particle smoothing, allowing us to examine the trade-off between increased time spent processing particles and error reduction in the simulation as a result of a smoother particle representation. In a less refined simulation with large amounts of particles it is likely that the improvement in the solution would quickly become limited by the space and time errors, meaning that the true benefits of particle smoothing cannot be observed. As with the electrostatic problem, subsequent problem sizes are generating by doubling the number of particles used (again with a base case of 2^{15} particles), and when using virtual particles we choose the optimal particle radius for this problem, specifically $r_0 = 2.5\Delta x$ (see Section 5.3.2).

We test EMPIRE-PIC on the following compute architectures: NVIDIA Pascal and Volta GPUs; Intel Broadwell and Cascade Lake CPUs; Intel's KNL; and the ARM-based ThunderX2. We exclude the NVIDIA Tesla K40 used in Chapter 4 from this analysis as initial testing found that the use of smooth particle shapes via virtual particles is not viable in terms of Time-to-Solution (TTS) on this hardware[1]. For the purposes of this work, all systems used consist

[1]Initial testing showed that the K40 was greater than one order of magnitude slower than the next slowest system.

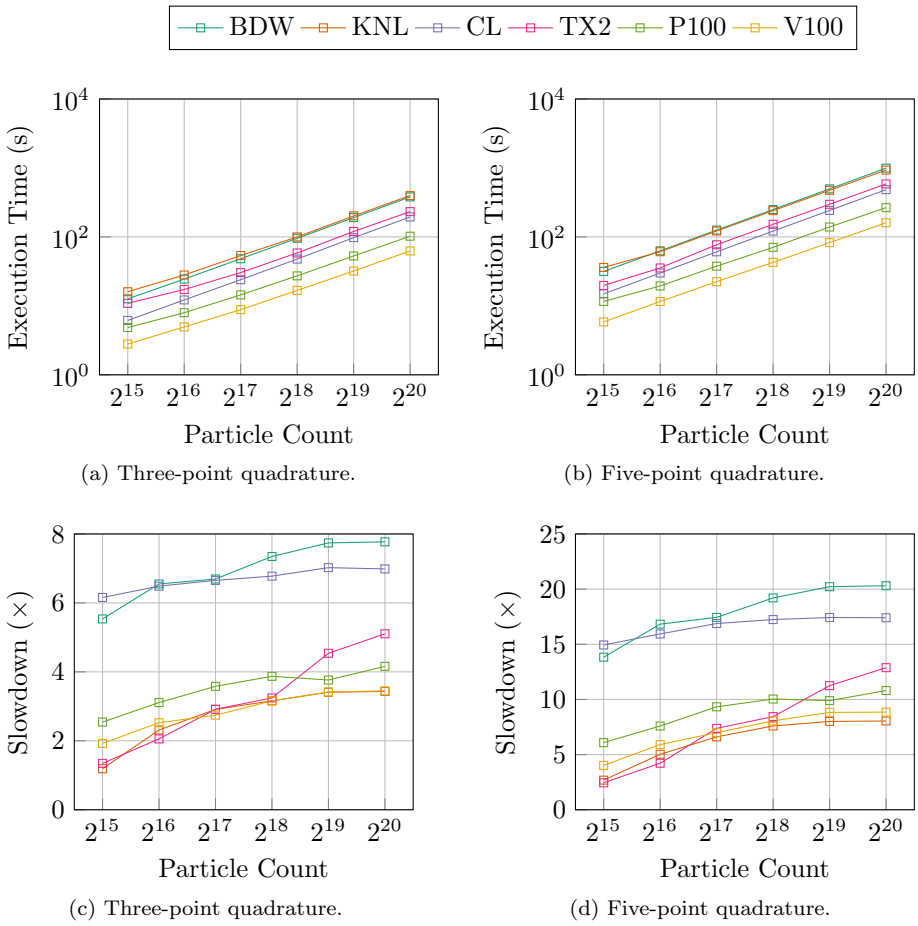
of a single node, and experiments conducted on GPUs make use of a single accelerator card. For the experiments run on the traditional CPU systems, we use a single socket in order to ensure that threads remain in the same Non-Uniform Memory Access (NUMA) region to avoid the performance penalty for inter-socket memory traffic. The socket is then saturated with OpenMP threads bound to cores, with hyperthreading disabled as preliminary testing found that this resulted in the best performance. On the KNL we found that using bound threads, with two hyperthreads per core, lead to the best results, therefore we used a total of 128 threads for the KNL experiments.

6.3 Results

In this section we seek to numerically quantify the difference in performance behaviour between the base implementation of EMPIRE-PIC, and the version that has been extended to use higher-order representation via virtual particles. To achieve this we present a raw performance comparison between the two versions for a range of particle counts. We then present an on-node strong scaling study of each implementation on the CPU systems, in order to examine how the performance changes as the number of processors is varied. Finally, we show a cost versus error analysis to establish whether the error reduction due to the use of virtual particles is worth the cost in terms of application performance.

6.3.1 Raw Performance Comparison

Figures 6.1(a) and 6.1(b) show the performance of the modified version of EMPIRE-PIC across all hardware for a variety of total particle counts. Additionally, we compare two quadrature orders resulting in 10 and 26 virtual particles per simulation particle, for three- and five-point quadrature, respectively. Using five-point quadrature results in a runtime approximately 2.5× longer than using three-point quadrature – this is in line with our expectations given the relative difference in the resultant number of virtual particles to be



(a) Three-point quadrature.

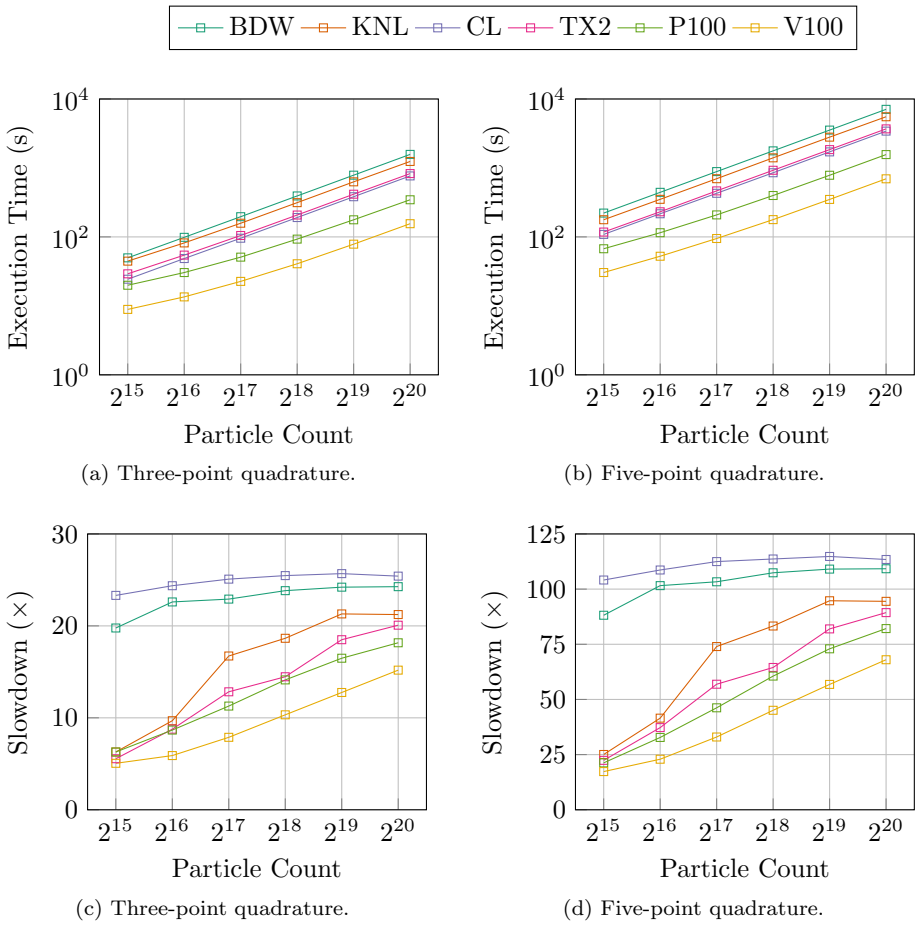
(b) Five-point quadrature.

(c) Three-point quadrature.

(d) Five-point quadrature.

Figure 6.1: Particle update execution time using smoothed particles at various particle counts for the electrostatic problem at differing quadrature orders. (a) and (b) show the raw performance; (c) and (d) consider the relative slowdown in contrast to the base implementation.

processed. As was seen in Chapter 4, the GPUs are best suited to executing particle-based kernels due to the high degree of parallelism and memory bandwidth on offer. Of the CPU systems, the results are consistent with those seen previously – the ThunderX2 and Cascade Lake perform the best due to having a higher memory bandwidth available, with the Intel system performing slightly better due to its higher peak performance. Unlike previous tests the KNL outperforms the Broadwell, however this would likely change if both sockets of the Broadwell were used. We additionally observe linear scaling with particle count for all hardware used, with the exception of lower particle counts where the



(a) Three-point quadrature.

(b) Five-point quadrature.

(c) Three-point quadrature.

(d) Five-point quadrature.

Figure 6.2: Particle update execution time using smoothed particles at various particle counts for the electromagnetic problem at differing quadrature orders. (a) and (b) show the raw performance; (c) and (d) consider the relative slowdown in contrast to the base implementation.

sub-linear scaling can be explained by cache effects.

Figures 6.1(c) and 6.1(d) present a different view of the data by displaying the relative slowdown versus an equivalent particle count for the base code, allowing us to quantify the overhead of enabling particle smoothing. The results are positive in the sense that the cost of virtual particles is lower than what we would expect theoretically for all hardware, at both quadrature orders. For three-point quadrature we would expect a 10× slowdown, while for five-point quadrature this penalty would increase to 26×. This is explained by two phenomena. Firstly, much of the data used by a simulation particle is shared by its

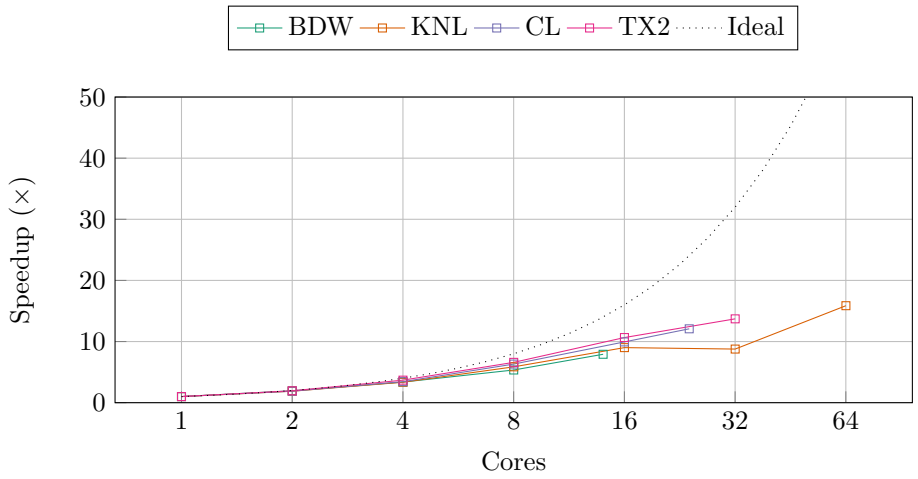
associated virtual particles; this means that this data will likely have already been loaded into cache and thus we gain the associated performance benefits. Secondly, as the particle-based kernels are heavily memory-bound, we are observing the effects of the increased FLOP to byte ratio discussed previously – we are making better use of the compute performance available on the hardware. The trends also suggest that the overhead will become constant once a large enough problem size is reached, representing the point where performance is totally memory bandwidth limited.

Figure 6.2 shows the results of the same experiments, this time carried out for the three-dimensional electromagnetic problem. We observe similar trends to the electrostatic simulation; notably that the performance of all platforms relative to each other is generally unchanged. The overall runtime continues to scale linearly with particle count, as expected. Sub-linear scaling at lower particle counts is now less common due to the inclusion of a magnetic field consuming additional space in cache. As before, the use of virtual particles has less overhead than one would expect (here we expect $26\times$ and $126\times$ slowdown for three- and five-point quadrature, respectively). Again, this is due to increased cache usage and arithmetic intensity. While the Cascade Lake shows a high overhead, this is of little consequence – in terms of absolute performance it remains the fastest of all CPUs considered. As with electrostatics, we see that the CPU systems eventually reach a constant overhead at larger problem sizes. While this is not the case for the GPUs, we conjecture that they would eventually reach such a point were the problem size increased further. The final point of interest here is that we reach the maximum overhead later than for the previous problem. This is due to electromagnetic simulations requiring more FLOP/s than their electrostatic counterparts; as current is deposited each time a particle crosses an element, the expensive evaluation of the basis functions must be performed multiple times per particle, per time-step. As a result, a larger problem size is required to cause this problem to become totally memory-bound.

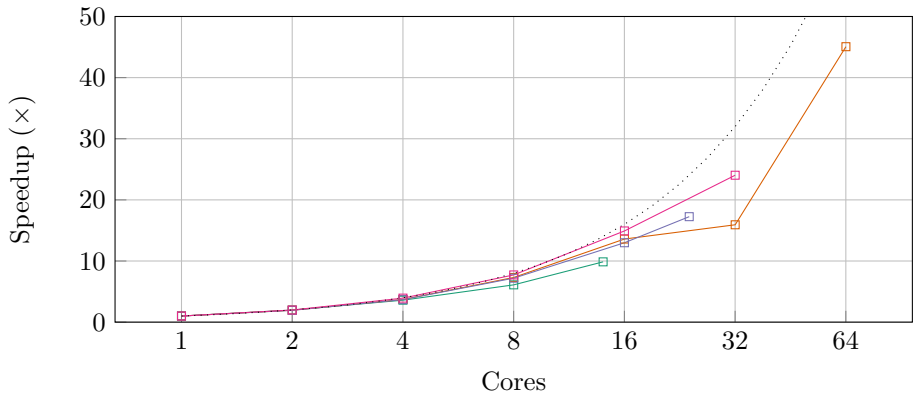
6.3.2 Strong Scaling

We now seek to quantify how the usage of virtual particles affects the scalability of both electrostatic and electromagnetic PIC simulations in contrast to the algorithm implemented in the original EMPIRE-PIC codebase. To this end we perform a strong scaling study on the CPU systems with virtual particles disabled, and enabled – again considering three- and five-point quadratures. For the base code we use the a problem size of 2^{20} particles. We modify the particle counts used for the runs where virtual particles are enabled such that the level of work performed in all runs is identical, i.e., for virtual particle tests we divide the total particle count by N_v, such that $N_{p,base} = N_v N_{p,smooth}$. This enables us to examine how the shift in work from memory to computation affects the scalability of the particle-based kernels, and facilitates a fairer comparison as, in general, we would always expect a larger workload to scale better than a smaller one.

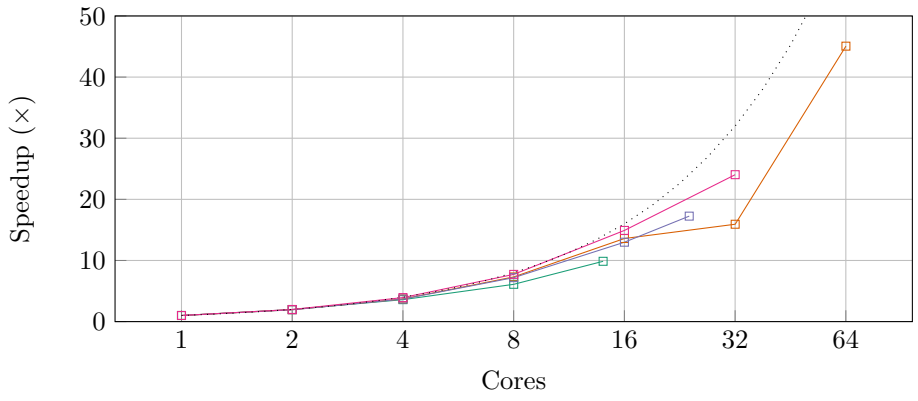
We begin our analysis by considering the electrostatic problem; this data is shown in Figure 6.3, with reference lines to facilitate comparison to ideal speedup. It is clear from Figure 6.3(a) that the base implementation of EMPIRE-PIC scales near perfectly up to four cores on all systems, but rapidly degrades beyond this point. This is unsurprising, given the memory-bound nature of the code. Figure 6.3(b) considers the same experiment for three-point quadrature. It is clear to see that shifting the workload to consist of a higher compute component greatly improves the strong scalability of the particle update on all systems. This is especially true for the KNL and ThunderX2, where we see approximately 43× and 24× speedup when using all available cores, contrasted to 16× and 14× speedup for the equivalent data points for the base code. As these machines possess the highest core counts, it is unsurprising that they benefit the most from this workload shift. When comparing to five-point quadrature (Figure 6.3(c)), we see a similar improvement over the base code to when three-point quadrature is used. The only discernible difference is that the KNL gains a slight improvement to a speedup of 45× when using all 64 cores available, due



(a) Base code.

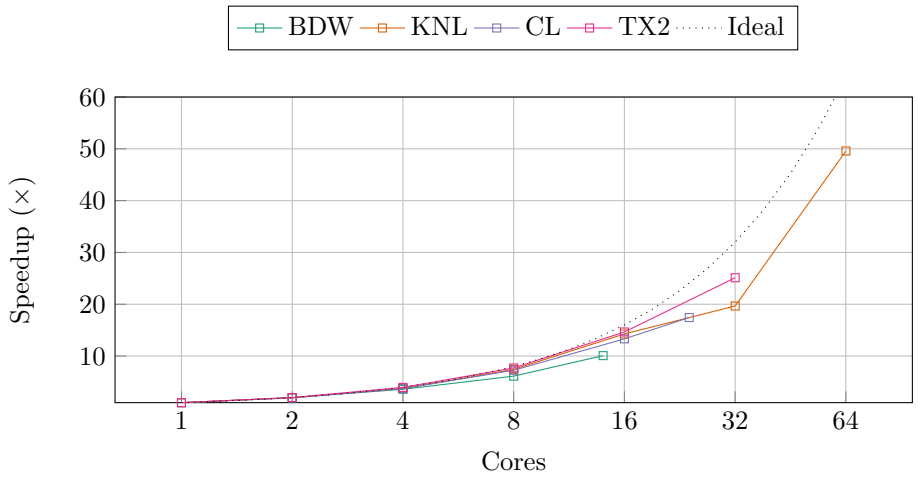


(b) Three point quadrature.

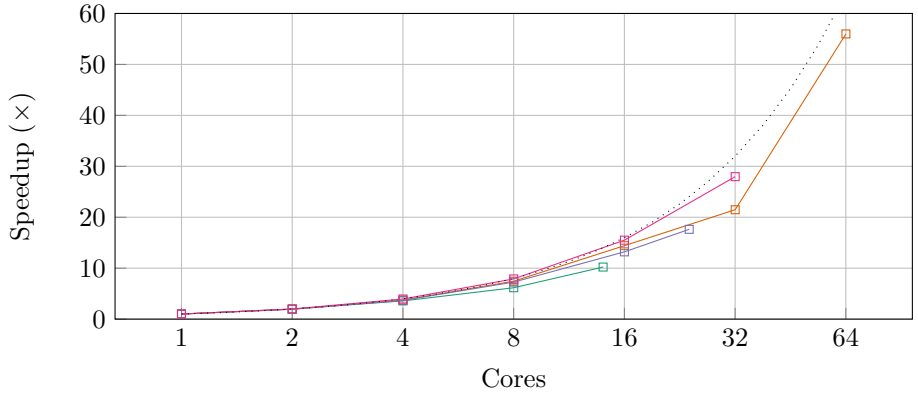


(c) Five point quadrature.

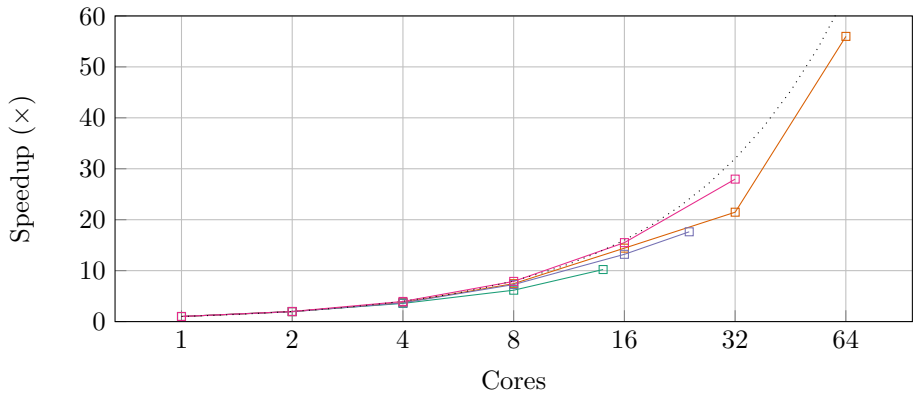
Figure 6.3: Strong scaling results for the particle update on the electrostatic problem.



(a) Base code.



(b) Three point quadrature.



(c) Five point quadrature.

Figure 6.4: Strong scaling results for the particle update on the electromagnetic problem.

to having the greatest amount of parallelism available. This data shows that the benefits of increasing the amount of computation and decreasing total memory usage has diminishing returns in terms of performance. Therefore, when choosing between three- and five-point quadrature, one should consider the effects of the level of smoothing on the problem solution to inform this decision.

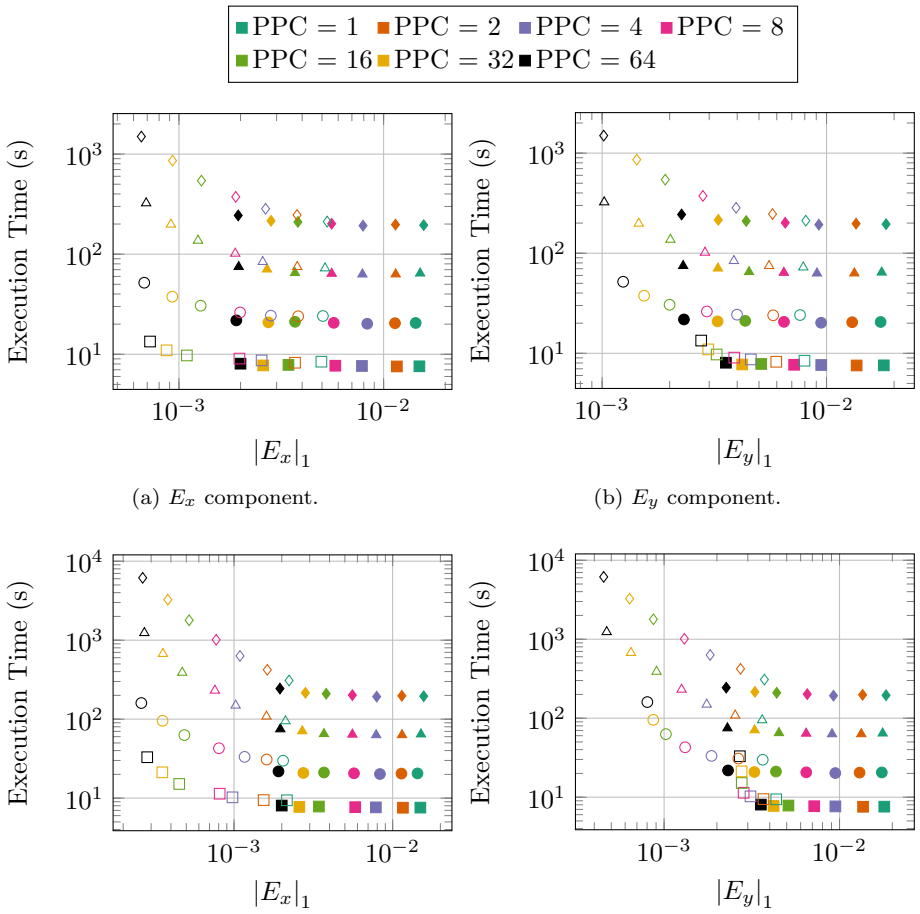
Figure 6.4 repeats this experiments for the electromagnetic simulation. It is clear to see from the graph shown in Figure 6.4(a) that the base implementation of EMPIRE-PIC maintains good strong scalability for much longer than in the electrostatic test. As explained in Section 6.3.1, this is the result of the electromagnetic scheme already containing a larger compute component for a similar amount of memory usage. In this case, the use of virtual particles with three-point quadrature (Figure 6.4(a)) has no effect on the scaling of either the Broadwell or Cascade Lake systems. However, we continue to see benefits for the ThunderX2 and KNL, improving from 50× and 25× speedup to 54× and 28×, respectively. This again is due to the high levels of parallelism on offer. Consistent with the results for the electrostatic test, we again observe that moving to five-point quadrature has a negligible effect on scaling (relative to using three points) with the exception of the KNL, where full-scale speedup improves marginally to 56×. As a result, we conclude that three-point quadrature should be used unless the problem benefits greatly from quadrature of higher orders.

6.3.3 Cost Versus Error Analysis

We now seek to quantify whether the reduced error in problem solutions due to the use of virtual particles is worth the additional computational cost that is incurred. This analysis is performed for the electromagnetic test; the ready availability of an analytical solution to the problem means that the change in error can be easily quantified, facilitating the error comparison. For this test, we continue to use the plasma parameters defined in Section 5.3.2, and we cover the same range of Particles per Cell (PPC) values. When using virtual particles we continue to use the optimal radius specified earlier in this chapter.

Figure 6.5 shows a comparison of both three- and five-point quadrature to the base EMPIRE-PIC implementation, in terms of solution error, and the total execution time required to complete the run. In the graphs shown, solid points represent the base implementation, while hollow points denote the tests using virtual particles. Each successive shape represents a level of simulation refinement, with squares showing data for the coarsest mesh and largest time-step size. Each level of refinement decreases the cell spacing and time-step size by a factor of two, in order to maintain a constant ratio in the CFL condition. Here, the base case consists of a mesh of $4 \times 4 \times 50$ cells, and 100 time-steps, one level of refinement higher than was used in Section 5.3.2. This facilitates a meaningful performance analysis by ensuring the smallest problem size runs for a reasonable amount of time. All performance data is collected using a single NVIDIA Tesla V100 GPU. As the code is deterministic, the usage of different systems will only influence the TTS, and not the resultant error. As we have seen how the other systems used perform relative to each other on this problem, this analysis is applicable to all of them.

We begin our analysis with the usage of three-point quadrature, shown in Figures 6.5(a) and 6.5(b), which consider the L_1 error in the x and y components of the electric field. Recall that this covers both the main signal (E_y) and the noise (E_x) in the field solution; the problem is set up with plane wave polarisation so E_z will have the same behaviour as E_x. Beginning with the noise component we see that refining the simulation has very little effect on the error while noticeably raising the TTS; this is true for both the base code, and when using higher-order particle representation. However, we observe that increasing the particle count, or enabling smoother particles moderately reduces this error while resulting in a comparatively low increase in application runtime. This suggests that the noise in the simulation is dominated by the particle distribution – refining the simulation in space and/or time would only have a noticeable effect if the PPC value was increased significantly. Comparing the use of the higher-order representation to simply adding more simulation particles,



(a) E_x component.

(b) E_y component.

(c) E_x component.

(d) E_y component.

Figure 6.5: Error versus cost analysis for the electromagnetic problem. (a) and (b) show three-point quadrature. (c) and (d) show five-point quadrature. Closer to the origin is better.

the data suggests that in general it is more cost effective to simply increase the number of basic particles used. Moving on to the E_y component, we can see that once sufficient numbers of particles are used refining the simulation reduces the error in the solution. Note that this point occurs at lower PPC counts when using higher-order representation due to the particle error dominating the solution to a lesser degree. As with the noise component of the electric field, it appears that the usage of a virtual particles approach is less cost effective than increasing the number of standard simulation particles. Looking at both field components it is apparent that carefully choosing simulation parameters while

using the higher-order representation can result in improved solutions over the base code at comparable or reduced runtimes. This usually occurs when both PPC and problem refinement are reduced together, causing the particle count to decrease significantly. However, similar data points can also be found for the base implementation. We observe similar trends for the runs using five-point quadrature (Figures 6.5(c) and 6.5(d)), with the difference that the larger number of virtual particles means that the use of particle smoothing results in a much larger growth in runtime. This data makes it clear that, for this problem, the additional error reduction gained by using more virtual particles to represent the underlying shape function is not worth the extra costs.

As the usage of virtual particles results in an increased arithmetic intensity, the poor vectorisation in EMPIRE-PIC is a larger factor in the high cost of using the method rather than limited memory bandwidth availability. Achieving vectorisation using portability libraries such as Kokkos is challenging – in order to maintain portability we must rely on the compiler to vectorise the scalar instructions. The addition of a portability layer makes this more difficult; it is harder for the compiler to detect vectorisation opportunities and prove that data dependencies do not exist. If satisfactory vectorisation could be realised, the additional performance costs would be greatly reduced as vector units could process multiple virtual particles simultaneously via SIMD instructions. This would mean that the results shown in Figure 6.5 could look very different in the future as compilers become more advanced. Specifically, this would result in a reduced performance gap versus the base implementation, while maintaining the same level of error reduction.

6.4 Summary

In this chapter we have documented the implementation of the extensions to the PIC algorithm introduced in Chapter 5 using the Kokkos performance portability library, and contrasted its performance to the particle representation in-

cluded in the base EMPIRE-PIC code. Specifically, we have shown that our method scales linearly with particle count, and the performance penalty incurred through the use of virtual particles is less than what would be theoretically expected due to shared data improving cache reuse. Additionally, we have seen that using virtual particles to shift the PIC workload to be more compute-focused improves the strong scaling of the particle-based kernels on CPUs with many cores, namely the ARM-based ThunderX2, and Intel's KNL. This effect is most noticeable on electrostatic simulations, as electromagnetics already possesses a larger compute component in the base case. Finally, a cost versus error analysis comparing the use of virtual particles to the base EMPIRE-PIC implementation using a three-dimensional TEM wave problem was conducted. Broadly, we observe that the use of three-point quadrature is more cost effective than five-point quadrature, and that in general, it is preferable from a performance perspective to increase the number of simulation particles used in the base code instead of enabling the higher-order particle representation. However, we find that virtual particles can outperform the base EMPIRE-PIC implementation on specific inputs while maintaining a comparable error in the problem solution. We concluded the chapter by postulating that, as compilers become better at vectorising performance portable code, the performance gap between the base implementation and the use of virtual particles could significantly narrow in the future.

CHAPTER 7

Conclusions and Future Work

The work presented in this thesis has focused on modern computational plasma physics simulations that make use of the Particle-in-Cell (PIC) algorithm, specifically those that represent the problem space using an unstructured mesh via the Finite Element Method (FEM). In particular we have examined the performance of FEM-PIC codes on both current and emerging High Performance Computing (HPC) architectures, the portability of such codes across these architectures, and algorithmic modifications that are expected to be better suited for massively parallel hardware. This research is likely to become even more relevant as computational hardware continues to diversify as we pursue the key milestone of Exascale computing. While we have focused on the FEM-PIC C++ code, EMPIRE-PIC, the research detailed in this thesis is also applicable to other codes, especially those that make use of performance portability libraries such as Kokkos.

Specifically, Chapter 4 has demonstrated that through the use of Kokkos it is possible to run FEM-PIC simulations on a variety of current and emerging compute architectures. Machine-specific tuning was also explored through the use of hierarchical parallelism features to express more complex algorithms that are better suited to modern hardware. Through this work we have alleviated performance issues in both the particle move and charge deposition PIC procedures, which typically represent a performance bottleneck in traditional PIC codes. This culminated in up to 3× speedup for the particle mover, and a large reduction in the number of atomic operations used when conducting charge deposition. Additionally, we have observed that we can also achieve *portability* across radically different architecture types, including traditional CPUs, many-

core CPUs, and NVIDIA GPUs. Chapter 4 concluded with a scaling study on three production U.S. Department of Energy (DOE) supercomputers, showing scalability of the EMPIRE-PIC code up to more than two thousand GPUs, and greater than one hundred thousand CPUs.

Chapter 5 extends the core algorithm of EMPIRE-PIC to use a higher-order representation of simulation particles for both electrostatic and electromagnetic unstructured FEM-PIC simulations with periodic boundaries. This method makes use of a smooth quadratic shape function with compact support on a finite radius to model the particles more smoothly, in contrast to the delta-shape particles typically used in FEM-PIC. A defining feature of the approach is the representation of this shape by surrounding simulation particles with computational virtual particles with delta shape, with fixed offsets and weights derived from the chosen radius and Gaussian quadrature rules. Along with raising simulation accuracy, the changes increase the arithmetic intensity of traditionally memory-bound particle kernels with only a minor increase in memory usage, with the aim of leveraging the high compute performance of modern hardware. Using four representative benchmark problems that cover both electrostatics and electromagnetics, the modified algorithm is both validated and shown to reduce the error in the problem solutions as the particles are made smoother, with the optimal radius appearing to be problem-dependent. Notably, we observed improvement in the tracking of basic particle motion on both uniform and distorted meshes, and suppressed numerical heating effects on resolved and under-resolved grids. Also of note is the near-elimination of the noise in simulated ion density for a plasma slab expansion problem.

Finally, Chapter 6 built on all of the work previously presented in this thesis, by documenting the Kokkos implementation of the changes to particle representation in EMPIRE-PIC and quantifying the resultant change in performance behaviour. Specifically, we observed that the performance penalty incurred through the use of virtual particles is less than what would be theoretically expected due to cache effects, i.e., an overhead less than a factor of the num-

ber of virtual particles used due to the reuse of shared data. Additionally, we have seen that using virtual particles to shift the balance between FLOP/s and memory bandwidth can lead to improved strong scalability of FEM-PIC simulations. In the best case, this shift improved parallel efficiency by up to 2.5× on electrostatic simulations. Chapter 6 concluded with a cost versus error analysis comparing the cost-effectiveness of both versions of EMPIRE-PIC. We observed that, in general, it remains preferable to increase the number of traditional particles used in the base code instead of enabling smoother particles. We ended the chapter by theorising that the performance gap between the base implementation and the modified algorithm could reduce in the future, as both hardware and compilers become more advanced.

7.1 Limitations

The primary limitation of the work presented in this thesis is that the research focuses on a single code, namely the EMPIRE-PIC unstructured PIC application. While this may seem to limit the general applicability of the analyses and extensions carried out, much of the algorithmic features of EMPIRE-PIC are commonplace in other PIC codes. For example, particle movers based on the Boris method [29], and charge/current deposition schemes derived from the charge-conserving method presented by Villasenor and Buneman [148] are considered to be the de facto standard methods used in PIC. As such, the findings of this thesis could be extrapolated to these other codes. This is particularly true of the work presented in Chapter 5; a key feature of the proposed particle shape representation is the ability to be implemented with minimal modifications to existing PIC infrastructure.

A further limitation of this thesis is that EMPIRE-PIC has been directly developed using the Kokkos performance portability library, i.e., there is no 'plain' C++ implementation to compare to in order to establish the overheads incurred through the use of Kokkos. This is especially relevant to the work

detailed in Chapters 4 and 6. However, the overhead of Kokkos has previously been assessed by multiple authors on a variety of scientific applications and has been shown to be low in many cases [9, 52, 88, 96]. When considered alongside the performance data shown in this thesis, there is no evidence to suggest that the runtime of EMPIRE-PIC is significantly impacted by Kokkos overheads. It is also important to note that this work seeks to examine the process of developing a new unstructured PIC code with performance portability from a single codebase being a key aim from the beginning. As a consequence, the development and maintenance of a second version of the code would be antithetical to this objective. Rather, the work documented in Chapter 4 provides the reader with a realistic view of the development process and performance expected when writing a performance portable plasma simulation code.

Regarding the algorithmic changes proposed in Chapter 5, one limitation is that the modified PIC algorithm is mostly tested on simple problems with readily available analytical solutions, and more realistic 'production' simulations are not considered. While this makes reasoning about the effects of the algorithm on the solution to such problems challenging, much of the results presented can still be applied to more complex cases. The simple problem considered in Section 5.3.1 demonstrated that the use of a smooth particle representation increased the accuracy of the tracking of particle motion when using first- and second-order basis functions. Section 5.3.3 illustrated that the use of smooth particles significantly improved Kinetic Energy (KE) stability in EMPIRE-PIC, especially in simulations where coarse meshes are used. Both of these improvements would be beneficial to PIC simulations as a whole. Additionally, the problems were chosen to be broadly representative, covering electrostatics and electromagnetics, and also considering both two- and three-dimensional geometries.

One final limitation of the work presented in this thesis is that the smooth particle representation featured in Chapters 5 and 6 is not extended for use in multi-node settings. While this means that comprehensive large-scale strong and

weak scaling studies could not be performed, the primary goal of the work was to assess whether the use of 'virtual particles' is viable on modern hardware even at the single node level. The extension of this representation to a distributed memory MPI + Kokkos paradigm is an ongoing research effort. Section 7.2.3 partially addresses this issue by outlining how this might be achieved.

7.2 Future Work

The work documented in this thesis is open to extension in a variety of ways. In general, these can be split into two categories: (i) further extension to the smooth particle representation added to EMPIRE-PIC in Chapter 5, and (ii) additional performance studies to expand the experiments conducted in Chapters 4 and 6. In this section we present an overview of these additional research avenues and provide details on how they could be initially explored.

7.2.1 Non-Periodic Boundaries for Virtual Particles

A key limitation of this thesis discussed in Section 7.1 is that the version of EMPIRE-PIC extended to support higher-order particle shape functions was not tested on a realistic production problem. One of the major limiting factors preventing this is that the modified algorithm does not currently handle simulation boundaries that are non-periodic, i.e., boundaries that result in particle reflection, absorbtion, or emission. The addition of any or all of these boundary conditions would greatly expand the amount of problems that could be tested using virtual particles. However, the implementation of these boundaries using virtual particles is non-trivial – it is not immediately clear how to handle a situation where some virtual particles collide with the boundary and others do not. Pointon has previously proposed a method for handling non-periodic boundary conditions while using higher-order particle shapes where the shape function is smoothly transitioned back to first-order as a particle approaches a boundary [123]. In our case, this would involve slowly reducing the radius of a

particle back to zero on the approach so that the particle is 'point-like' once the boundary is hit. It is thought that this method would be feasible to implement in EMPIRE-PIC.

7.2.2 Variable Radius for Virtual Particles

While the existing version of the extended PIC algorithm has been shown to be able to cope with distorted meshes with elements of varying size (see Section 5.3.1), it is likely that this ability would be improved if the defined radius of a particle could change as it moves through the problem domain. This is a consequence of the optimal particle radius being some fraction of the cell size, meaning that the value of this optimum depends on the size of the element in which the particle currently resides. The addition of variable radius particle shapes would also facilitate the implementation of the Pointon boundary conditions described above. This further extension could be implemented via modifications to the particle move step of the algorithm. Instead of advancing all of the virtual particles in lock-step, we would first move the central particle that represents the true position to its updated location. The new radius could then be determined, allowing the new virtual particle offsets and weights to be calculated. The virtual particles would then be moved to their new positions using the standard move routine as normal; as all element crossings would continue to be tracked, the resulting current deposition scheme would remain charge-conserving as long as the newly calculated weights continue to sum to one.

7.2.3 Distributed Memory for Virtual Particles

Perhaps the most significant limitation of the work documented in this thesis is that the smooth particle representation using virtual particles implemented in EMPIRE-PIC is not extended to function in an MPI + Kokkos, distributed memory paradigm. The primary issue with such an implementation is that it is not obvious how to handle the case where a portion of a particle's associated

virtual particles are outside the domain of the current owning processor. In its current form, the particle migration scheme within EMPIRE-PIC does not make use of a halo of ghost cells – particles are moved to the edge of the current processor's domain, and after the current move iteration all marked particles are migrated to their neighbour and moved again. This process is repeated until it is detected that no particles have migrated during an iteration of the move kernel. It would be necessary to introduce such halo regions in order to handle the virtual particle case – the ghost layer would need to be as thick as the maximum particle radius at the process boundary. Note that due to the use of an unstructured mesh, the extra halo regions would vary in terms of cell count and thickness depending on the resolution of the mesh in the occupied area. Once the central particle has entered the halo region it, and its associated virtual particles, could be migrated to the destination processor as normal.

7.2.4 Evaluation of Emerging Architectures

The increasing diversity of compute architectures and their associated programming models has been a core theme of much of the work presented in this thesis. While every attempt has been made to consider a variety of hardware types and generations, many of the currently emerging architectures have not been covered. The most notable examples include Intel's upcoming line of Xe-HPC datacentre grade GPUs, as well as AMD's EPYC CPUs (codenamed 'Rome') and Radeon Instinct GPUs. These hardware items are of particular interest to the scientific community as the upcoming supercomputers Aurora and Frontier will be based on Intel and AMD Graphics Processing Units (GPUs), respectively. As the EMPIRE-PIC codebase is written entirely using the Kokkos performance portability framework, extending the work presented in Chapters 4 and 6 to include performance comparisons to these emerging hardware types would be simple to achieve – being able to tolerate new hardware releases is a core design goal of both Kokkos and EMPIRE-PIC. In theory, testing could begin immediately once appropriate Kokkos backends have been made available; in the case

of Intel GPUs this would be OneAPI/DPC++ or OpenCL compatibility, and in the case of Radeon Instinct this would be ROCm/HIP capability.

7.3 Final Remarks

While the work presented in this thesis has not resulted in the increased performance that was hoped for, we have shown that the use of higher-order methods can be used to alter the balance between FLOP/s and the amount of data that is moved to and from main memory, and that scalability can be improved as a consequence of this. The relevance of higher-order algorithms is expected to increase as the hardware of the Exascale age becomes steadily more parallel, and memory capacity and bandwidth grows at a slower rate. This is primarily due to their increased arithmetic intensity improving application performance and scalability by making better use of the available hardware.

The use of higher-order methods for FEM schemes also improves simulation accuracy and robustness, relative to their low-order counterparts commonly used by the algorithms of today. Potential benefits include: improved simulation convergence, more accurate representation of curved geometries, and reduced simulation constraints. This will allow domain scientists to capture and simulate phenomena that cannot be represented in traditional simulations, thus aiding scientific research and discovery. With this in mind, the key themes of this thesis, namely the use of diverse hardware and higher-order methods, will continue to be relevant throughout the Exascale era that is rapidly approaching.

Bibliography

[1] S. Abhyankar, J. Brown, E. M. Constantinescu, D. Ghosh, B. F. Smith, and H. Zhang. PETSc/TS: A Modern Scalable ODE/DAE Solver Library. *arXiv preprint arXiv:1806.01437*, 2018.

[2] M. F. Adams, S. Ethier, and N. Wichmann. Performance of Particle in Cell Methods on Highly Concurrent Computational Architectures. In *Journal of Physics: Conference Series*, volume 78, page 012001. IOP Publishing, June 2007.

[3] E. Akarsu, K. Dincer, T. Haupt, and G. C. Fox. Particle-in-Cell Simulation Codes in High Performance Fortran. In *Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pages 38–60, November 1996.

[4] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, April 1967. Association for Computing Machinery.

[5] R. Anderson, V. Dobrev, T. Kolev, D. Kuzmin, M. Quezada de Luna, R. Rieben, and V. Tomov. High-Order Local Maximum Principle Preserving (MPP) Discontinuous Galerkin Finite Element Method for the Transport Equation. *Journal of Computational Physics*, 334:102–124, April 2017.

[6] R. W. Anderson, V. A. Dobrev, T. V. Kolev, R. N. Rieben, and V. Z. Tomov. High-Order Multi-Material ALE Hydrodynamics. *SIAM Journal on Scientific Computing*, 40(1):B32–B58, January 2018.

[7] K. Antypas, J. Shalf, and H. Wasserman. NERSC-6 Workload Analysis and Benchmark Selection Process. Technical Report LBNL-1014E, Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States), August 2008.

[8] T. D. Arber, K. Bennett, C. S. Brady, A. Lawrence-Douglas, M. G. Ramsay, N. J. Sircombe, P. Gillies, R. G. Evans, H. Schmitz, A. R. Bell, and C. P. Ridgers. Contemporary Particle-in-Cell Approach to Laser-Plasma Modelling. *Plasma Physics and Controlled Fusion*, 57(11):113001, September 2015.

[9] V. Artigues, K. Kormann, M. Rampp, and K. Reuter. Evaluation of Performance Portability Frameworks for the Implementation of a Particle-in-Cell Code. *Concurrency and Computation: Practice and Experience*, 32(11):e5640, December 2020.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, September 1991.

[11] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang. Scaling Hypre's Multigrid Solvers to 100,000 Cores. In M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied, editors, *High-Performance Scientific Computing: Algorithms and Applications*, pages 261–279. Springer London, London, 2012.

[12] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.13, Argonne National Laboratory, 2020. `https://www.mcs.anl.gov/petsc`.

[13] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[14] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist. Amesos2 and Belos: Direct and Iterative Solvers for Large Sparse Linear Systems. *Scientific Programming*, 20(3):241–255, July 2012.

[15] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the International Conference on Parallel Processing (ICPP'95)*, pages 11–14, 1995.

[16] L. Berger-Vergiat, C. A. Glusa, J. J. Hu, M. Mayr, A. Prokopenko, C. M. Siefert, R. S. Tuminaro, and T. A. Wiesner. MueLu User's Guide. Technical Report SAND2019-0537, Sandia National Laboratories, January 2019.

[17] M. T. Bettencourt, D. A. S. Brown, K. L. Cartwright, E. C. Cyr, C. A. Glusa, P. T. Lin, S. G. Moore, D. A. O. McGregor, R. P. Pawlowski, E. G. Phillips, N. V. Roberts, S. A. Wright, S. Maheswaran, J. P. Jones,

and S. A. Jarvis. EMPIRE-PIC: A Performance Portable Unstructured Particle-in-Cell Code. *Communications in Computational Physics*, April 2021. (Accepted).

[18] M. T. Bettencourt, D. A. S. Brown, and G. Radtke. Unstructured Higher-order PIC Methods. In *The 25th International Conference on Numerical Simulation of Plasmas (ICNSP 2017)*, Leuven, Belgium, September 2017.

[19] M. T. Bettencourt and A. D. Greenwood. Performance Improvements for Efficient Electromagnetic Particle-In-Cell Computation on 1000s of CPUs. *IEEE Transactions on Antennas and Propagation*, 56(8):2178–2186, August 2008.

[20] R. F. Bird, P. Gillies, M. R. Bareford, J. A. Herdman, and S. A. Jarvis. Performance Optimisation of Inertial Confinement Fusion Codes using Mini-applications. *The International Journal of High Performance Computing Applications*, 32(4):570–581, November 2018.

[21] R. F. Bird, S. J. Pennycook, S. A. Wright, and S. A. Jarvis. Towards a Portable and Future-Proof Particle-in-Cell Plasma Physics Code. In *1st International Workshop on OpenCL (IWOCL 13)*, May 2013.

[22] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis. Performance Modelling of Magnetohydrodynamics Codes. In *Computer Performance Engineering*, pages 197–209. Springer, 2012.

[23] C. K. Birdsall and D. Fuss. Clouds-in-Clouds, Clouds-in-Cells Physics for Many-Body Plasma Simulation. *Journal of Computational Physics*, 3(4):494–511, April 1969.

[24] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. Plasma Physics Series. Institute of Physics Publishing, Bristol BS1 6BE, UK, 1991.

[25] J. A. Bittencourt. *Fundamentals of Plasma Physics*. Springer, 3rd edition, 2004.

[26] J. D. Blahovec, L. A. Bowers, J. W. Luginsland, G. E. Sasser, and J. J. Watrous. 3-D ICEPIC Simulations of the Relativistic Klystron Oscillator. *IEEE Transactions on Plasma Science*, 28(3):821–829, June 2000.

[27] P. B. Bochev, J. J. Hu, C. M. Siefert, and R. S. Tuminaro. An Algebraic Multigrid Approach Based on a Compatible Gauge Reformulation of Maxwell's Equations. *SIAM Journal on Scientific Computing*, 31(1):557–583, October 2008.

[28] B. M. Boghosian. Computational Physics on the Connection Machine: Massive Parallelism - a New Paradigm. *Computers in Physics*, 4(1):14–33, January 1990.

[29] J. Boris. Relativistic Plasma Simulation: Optimization of a Hybrid Code. In *Proceedings of the Fourth Conference on Numerical Simulation of Plasmas*, pages 3–68, Naval Research Laboratory, Washington, D.C, July 1971.

[30] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 0.374 Pflop/s Trillion-Particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 63:1–63:11, Piscataway, NJ, USA, November 2008. IEEE Press.

[31] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh Performance Three-Dimensional Electromagnetic Relativistic Kinetic Plasma Simulation. *Physics of Plasmas*, 15(5):055703, May 2008.

[32] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods*, volume 15. Springer Science & Business Media, 2012.

[33] D. A. S. Brown, M. T. Bettencourt, S. A. Wright, J. P. Jones, and S. A. Jarvis. Performance of Second Order Particle-in-Cell Methods on Modern Many-Core Architectures. In *IOP Computational Plasma Physics Conference*, November 2017.

[34] D. A. S. Brown, M. T. Bettencourt, S. A. Wright, S. Maheswaran, J. P. Jones, and S. A. Jarvis. Higher-Order Particle Representation for Particle-in-Cell Simulations. *Journal of Computational Physics*, June 2021. (In Press).

[35] D. A. S. Brown, S. A. Wright, and S. A. Jarvis. Performance of a Second Order Electrostatic Particle-in-Cell Algorithm on Modern Many-Core Architectures. *Electronic Notes in Theoretical Computer Science*, 340:67–84, October 2018.

[36] H. Burau, R. Widera, W. Honig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, and M. Bussmann. PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *IEEE Transactions on Plasma Science*, 38(10):2831–2839, October 2010.

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. IEEE, October 2009.

[38] F. F. Chen. *Introduction to Plasma Physics and Controlled Fusion.* Springer, 3rd edition, 2016.

[39] G. Chen, L. Chacón, and D. C. Barnes. An Efficient Mixed-Precision, Hybrid CPU-GPU Implementation of a Nonlinearly Implicit One-Dimensional Particle-in-Cell Algorithm. *Journal of Computational Physics*, 231(16):5374–5388, June 2012.

[40] M. J. Chorley and D. W. Walker. Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-Core Clusters. *Journal of Computational Science*, 1(3):168–174, August 2010.

[41] B. I. Cohen, A. B. Langdon, D. W. Hewett, and R. J. Procassini. Performance and Optimization of Direct Implicit Particle Simulation. *Journal of Computational Physics*, 81(1):151–168, March 1989.

[42] J. Dawson. One-Dimensional Plasma Model. *The Physics of Fluids*, 5(4):445–459, April 1962.

[43] J. M. Dawson. Particle Simulation of Plasmas. *Reviews of Modern Physics*, 55:403–447, April 1983.

[44] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon. Performance Portability across Diverse Computer Architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 1–13, November 2019.

[45] V. K. Decyk and T. V. Singh. Adaptable Particle-in-Cell algorithms for graphical processing units. *Computer Physics Communications*, 182(3):641–648, March 2011.

[46] J. Dickson, S. Wright, S. Maheswaran, A. Herdman, M. C. Miller, and S. Jarvis. Replicating HPC I/O Workloads with Proxy Applications. In *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 13–18. IEEE, November 2016.

[47] J. Dickson, S. A. Wright, D. Harris, S. Maheswaran, J. Herdman, M. C. Miller, and S. A. Jarvis. Enabling Portable I/O Analysis of Commercially Sensitive HPC Applications through Workload Replication. *Cray User Group*, pages 1–14, 2017.

[48] J. Dongarra and M. A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, 2013.

[49] J. J. Dongarra. The LINPACK Benchmark: An Explanation. In *International Conference on Supercomputing*, pages 456–474. Springer, 1987.

[50] T. H. Dupree. Kinetic Theory of Plasma and the Electromagnetic Field. *Physics of Fluids (1958-1988)*, 6(12):1714–1729, December 1963.

[51] H. C. Edwards, C. R. Trott, and F. Foertter. Kokkos Tutorial. `https://press3.mcs.anl.gov//atpesc/files/2017/08/ATPESC_2017_Track-2_7_8-3_315pm_Edwards-Kokkos.pdf`, May 2017. Accessed: 2020-08-31.

[52] 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.

[53] Essex Edwards and Robert Bridson. A High-Order Accurate Particle-in-Cell Method. *International Journal for Numerical Methods in Engineering*, 90(9):1073–1088, April 2012.

[54] R. D. Falgout, J. E. Jones, and U. M. Yang. The Design and Implementation of Hypre, a Library of Parallel High Performance Preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 267–294. Springer, January 2006.

[55] R. D. Falgout and U. M. Yang. Hypre: A Library of High Performance Preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, April 2002.

[56] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[57] R. A. Fonseca, L. O. Silva, F. S. Tsung, V. K. Decyk, W. Lu, C. Ren, W. B. Mori, S. Deng, S. Lee, T. Katsouleas, and J. C. Adam. OSIRIS: A Three-Dimensional, Fully Relativistic Particle in Cell Code for Modeling Plasma Based Accelerators. In P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, editors, *Computational Science ICCS 2002*, pages 342–351, Berlin, Heidelberg, April 2002. Springer Berlin Heidelberg.

[58] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, and R. L. White. Solving Problems On Concurrent Processors Vol. 1: General Techniques and Regular Problems. *Computers in Physics*, 3(1):83–84, January 1989.

[59] G. Fridman, G. Friedman, A. Gutsol, A. B. Shekhter, V. N. Vasilets, and A. Fridman. Applied Plasma Medicine. *Plasma Processes and Polymers*, 5(6):503–533, August 2008.

[60] A. Friedman. A Second-Order Implicit Particle Mover with Adjustable Damping. *Journal of Computational Physics*, 90(2):292–312, October 1990.

[61] Fujitsu Global. Specifications – Supercomputer Fugaku. `https://www.fujitsu.com/global/about/innovation/fugaku/specifications/`. Accessed: 2020-08-07.

[62] K. Germaschewski, W. Fox, S. Abbott, N. Ahmadi, K. Maynard, L. Wang, H. Ruhl, and A. Bhattacharjee. The Plasma Simulation Code: A Modern Carticle-in-Cell Code with Patch-Based Load-Balancing. *Journal of Computational Physics*, 318:305–326, August 2016.

[63] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. *ACM Sigplan Notices*, 17(6):120–126, June 1982.

[64] Graph500.org. Graph500 - Homepage. `https://www.graph500.org/`. Accessed: 2020-08-07.

[65] P. M. Gresho and R. L. Lee. Don't Suppress the Wiggles - They're Telling You Something! *Computers & Fluids*, 9(2):223–253, June 1981.

[66] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[67] 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. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–10, March 2009.

[68] J. A. Herdman, W. P. Gaudin, D. Turland, and S. D. Hammond. Benchmarking and Modelling of POWER7, Westmere, BG/P, and GPUs: An Industry Case Study. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):16–22, March 2011.

[69] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software*, 31(3):397–423, December 2005.

[70] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-Applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009.

[71] J. S. Hesthaven and T. Warburton. Nodal High-Order Methods on Unstructured Grids: I. Time-Domain Solution of Maxwell's Equations. *Journal of Computational Physics*, 181(1):186–221, September 2002.

[72] R. Hockney. Measurements of Collision and Heating Times in a Two-Dimensional Thermal Computer Plasma. *Journal of Computational Physics*, 8(1):19–44, August 1971.

[73] R. W. Hockney. Computer Experiment of Anomalous Diffusion. *The Physics of Fluids*, 9(9):1826–1835, 1966.

[74] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. crc Press, 1988.

[75] 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.

[76] R. Hornung, H. Jones, J. Keasler, R. Neely, O. Pearce, S. Hammond, C. Trott, P. Lin, C. Vaughan, J. Cook, R. Hoekstra, B. Bergen, J. Payne, and G. Womeldorff. ASC Tri-Lab Co-Design Level 2 Milestone Report 2015. Technical report, Lawrence Livermore National Laboratory, September 2015.

[77] Intel Corporation. Intel VTune Profiler. `https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html`. Accessed: 2020-08-12.

[78] Intel Corporation. Introducing Intel MPI Benchmarks. `https://software.intel.com/content/www/us/en/develop/articles/intel-mpi-benchmarks.html`. Accessed: 2020-08-10.

[79] Intel Corporation. OneAPI Programming Model. `https://www.oneapi.com/`, November 2019. Accessed: 2020-08-14.

[80] Intel Corporation. Intel Xeon Processor Scalable Family: Specification Update. Technical Report 336065-013US, Intel Corporation, March 2020.

[81] G. Jacobs, J. Hesthaven, and G. Lapenta. *Simulations of the Weibel Instability with a High-Order Discontinous Galerkin Particle-In-Cell Solver*. June 2012.

[82] G. B. Jacobs and J. S. Hesthaven. High-Order Nodal Discontinuous Galerkin Particle-in-Cell Method on Unstructured Grids. *Journal of Computational Physics*, 214(1):96–121, May 2006.

[83] J. Jin. *The Finite Element Method in Electromagnetics*. Wiley-IEEE Press, 3rd edition, 2014.

[84] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *2009 International Conference on Parallel Processing Workshops*, pages 394–401, September 2009.

[85] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108, October 1993.

[86] Khronos OpenCL Working Group. OpenCL Specification Version 2.0. `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf`, July 2015. Accessed: 2020-08-14.

[87] Khronos SYCL Working Group. SYCL Specification Version 1.2.1. `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf`, April 2020. Accessed: 2020-08-14.

[88] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis. Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 834–841, September 2017.

[89] Y. L. Klimontovich. *The Statistical Theory of Non-Equilibrium Processes in a Plasma: International Series of Monographs in Natural Philosophy*, volume 9. Elsevier, 2013.

[90] V. M. K. Kotteda, V. Kumar, and W. Spotz. Performance of Preconditioned Iterative Solvers in MFiX–Trilinos for Fluidized Beds. *The Journal of Supercomputing*, 74(8):4104–4126, May 2018.

[91] S. Ku, R. Hager, C. S. Chang, J. M. Kwon, and S. E. Parker. A New Hybrid-Lagrangian Numerical Scheme for Gyrokinetic Simulation of Tokamak Edge Plasma. *Journal of Computational Physics*, 315:467–475, June 2016.

[92] M. Kumar. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.

[93] A. B. Langdon. Effects of the Spatial Grid in Simulation Plasmas. *Journal of Computational Physics*, 6(2):247–267, October 1970.

[94] A. B. Langdon and C. K. Birdsall. Theory of Plasma Simulation Using Finite-Size Particles. *The Physics of Fluids*, 13(8):2115–2122, 1970.

[95] M. Lange, G. Gorman, M. Weiland, L. Mitchell, and J. Southern. Achieving Efficient Strong Scaling with PETSc Using Hybrid MPI/OpenMP Optimisation. In J. M. Kunkel, T. Ludwig, and H. W. Meuer, editors, *Supercomputing*, pages 97–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[96] T. R. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. A. Herdman, and S. A. Jarvis. Performance Portability of an Unstructured Hydrodynamics Mini-Application. In *Proceedings of 2018 International Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*. Association for Computing Machinery, November 2018.

[97] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Computer Architecture News*, 38(3):451–460, June 2010.

[98] P. Lin, M. Bettencourt, S. Domino, T. Fisher, M. Hoemmen, J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, et al. Towards Extreme-Scale Simulations for Low Mach Fluids with Second-Generation Trilinos. *Parallel Processing Letters*, 24(04):1442005, December 2014.

[99] A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, J. M. Levesque, and S. A. Jarvis. CloverLeaf: Preparing Hydrodynamics Codes for Exascale. *The Cray User Group*, 2013, May 2013.

[100] R. Marchand. PTetra, a Tool to Simulate Low Orbit Satellite-Plasma Interaction. *IEEE Transactions on Plasma Science*, 40(2):217–229, February 2012.

[101] J. D. McCalpin et al. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 2:19–25, September 1995.

[102] S. McCartney. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. Walker & Company, 1999.

[103] S. McIntosh-Smith, M. Martineau, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, and D. Beckingsale. TeaLeaf: a Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 842–849. IEEE, September 2017.

[104] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1. Technical report, June 2015.

[105] H. Moon, F. L. Teixeira, and Y. A. Omelchenko. Exact Charge-Conserving Scatter-Gather Algorithm for Particle-in-Cell Simulations on Unstructured Grids: A Geometric Perspective. *Computer Physics Communications*, 194:43–53, September 2015.

[106] G. E. Moore. Cramming More Components onto Integrated Circuits (from 1965). *Proceedings of the IEEE*, 86(1):82–85, April 1998.

[107] G. R. Mudalige, M. K. Vernon, and S. A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–14, April 2008.

[108] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, January 1996.

[109] J.-C. Nédélec. Mixed Finite Elements in $\mathbb{R}^3$. *Numerische Mathematik*, 35(3):315–341, 1980.

[110] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Notices*, 42(6):89–100, June 2007.

[111] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media Inc., 1996.

[112] D. R. Nicholson. *Introduction to Plasma Theory*. Krieger Publishing Company, Malabar, Florida, February 1992.

[113] R. W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM New York, NY, USA, August 1998.

[114] NVIDIA Corporation. Profiler – CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html. Accessed: 2020-08-12.

[115] NVIDIA Corporation. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, August 2017. Accessed: 2020-08-14.

[116] NVIDIA Corporation. CUDA C++ Programming Guide. https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf, November 2019. Accessed: 2020-08-14.

[117] OpenACC Standard Committee. OpenACC Application Programming Interface Version 3.0. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf, November 2019. Accessed: 2020-08-14.

[118] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 5.0. Technical report, November 2018.

[119] A. M. B. Owenson, S. A. Wright, R. A. Bunt, Y. K. Ho, M. J. Street, and S. A. Jarvis. An Unstructured CFD Mini-Application for the Performance Prediction of a Production CFD Code. *Concurrency and Computation: Practice and Experience*, 32(10):e5443, July 2019.

[120] S. J. Pennycook, J. D. Sewall, and V. W. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, March 2019.

[121] O. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. Should We Worry About Memory Loss? *ACM SIGMETRICS Performance Evaluation Review*, 38(4):69–74, March 2011.

[122] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.

[123] T. D. Pointon. Second-order, Exact Charge Conservation for Electromagnetic Particle-in-Cell Simulation in Complex Geometry. *Computer Physics Communications*, 179(8):535–544, October 2008.

[124] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, Weimar, February 2009.

[125] S. Ramalingam, M. Hall, and C. Chen. Improving High-Performance Sparse Libraries Using Compiler-Assisted Specialization: A PETSc Case Study. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 487–496, May 2012.

[126] P. Rambo. Numerical Heating in Hybrid Plasma Simulations. *Journal of Computational Physics*, 133(1):173–180, May 1997.

[127] I. Z. Reguly. Performance Portability of Multi-Material Kernels. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 26–35, November 2019.

[128] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A Detailed, Accurate MPI Benchmark. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 52–59. Springer, 1998.

[129] M. A. Riquelme, E. Quataert, and D. Verscharen. Particle-in-cell Simulations of Continuously Driven Mirror and Ion Cyclotron Instabilities in High Beta Astrophysical and Heliospheric Plasmas. *The Astrophysical Journal*, 800(1):27, February 2015.

[130] S. I. Roberts, S. A. Wright, S. A. Fahmy, and S. A. Jarvis. The Power-Optimised Software Envelope. *ACM Transactions on Architecture and Code Optimization*, 16(3):21:1–21:27, June 2019.

[131] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The Structural Simulation Toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, March 2011.

[132] J. Roussel, F. Rogier, G. Dufour, J. Mateo-Velez, J. Forest, A. Hilgers, D. Rodgers, L. Girard, and D. Payan. SPIS Open-Source Code: Methods, Capabilities, Achievements, and Prospects. *IEEE Transactions on Plasma Science*, 36(5):2360–2368, October 2008.

[133] A. Scheinberg, G. Chen, S. Ethier, S. Slattery, R. F. Bird, P. Worley, and C.-S. Chang. Kokkos and Fortran in the Exascale Computing Project Plasma Physics Code XGC. November 2019.

[134] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland. Large Scale Parallel Solution of Incompressible Flow Problems Using Uintah and Hypre. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 458–465. IEEE, May 2013.

[135] M. Shalaby, A. E. Broderick, P. Chang, C. Pfrommer, A. Lamberts, and E. Puchwein. SHARP: A Spatially Higher-order, Relativistic Particle-in-cell Code. *The Astrophysical Journal*, 841(1):52, May 2017.

[136] S. Slattery, C. Junghans, D. Lebrun-Grandie, S. Fogerty, R. F. Bird, S. Reeve, G. Chen, R. Halver, A. Scheinberg, C. Smith, and E. Weinberg. ECP-copa/Cabana: Version 0.3.0, May 2020.

[137] A. Spector and D. Gifford. The Space Shuttle Primary Computer System. *Communications of the ACM*, 27(9):872–900, September 1984.

[138] J. Squire, H. Qin, and W. M. Tang. Geometric Integration of the Vlasov-Maxwell System With a Variational Particle-in-Cell Scheme. *Physics of Plasmas*, 19(8):084501, August 2012.

[139] T. Stindl, J. Neudorfer, A. Stock, M. Auweter-Kurtz, C.-D. Munz, S. Roller, and R. Schneider. Comparison of Coupling Techniques in a High-Order Discontinuous Galerkin-Based Particle-in-Cell Solver. *Journal of Physics D: Applied Physics*, 44(19):194004, April 2011.

[140] W. Tang, B. Wang, S. Ethier, G. Kwasniewski, T. Hoefler, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, C. Rosales-Fernandez, and T. Williams. Extreme Scale Plasma Turbulence Simulations on Top Supercomputers Worldwide. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 502–513, November 2016.

[141] The Trilinos Project Team. *The Trilinos Project Website*, 2020 (acccessed June 23, 2020). `https://trilinos.github.io`.

[142] TOP500.org. TOP500 - Homepage. `https://www.top500.org/`. Accessed: 2020-08-07.

[143] TOP500.org. TOP500 List - June 2020. `https://www.top500.org/lists/top500/2020/06/`. Accessed: 2020-08-14.

[144] D. Truby, S. Wright, R. Kevis, S. Maheswaran, A. Herdman, and S. Jarvis. BookLeaf: An Unstructured Hydrodynamics Mini-Application. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 615–622. IEEE, September 2018.

[145] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, May 1995. IEEE.

[146] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, J. Park, R. Ryne, O. Shapoval, M. Thvenet, and W. Zhang. Warp-X: A New Exascale Computing Platform for Beam-Plasma Simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 909:476–479, 2018.

[147] J.-L. Vay, P. Colella, J. W. Kwan, P. McCorquodale, D. B. Serafini, A. Friedman, D. P. Grote, G. Westenskow, J.-C. Adam, A. Hron, and I. Haber. Application of Adaptive Mesh Refinement to Particle-in-Cell Simulations of Plasmas and Beams. *Physics of Plasmas*, 11(5):2928–2934, 2004.

[148] J. Villasenor and O. Buneman. Rigorous Charge Conservation for Local Electromagnetic Field Solvers. *Computer Physics Communications*, 69(2):306–316, March 1992.

[149] A. Vlasov. The Vibrational Properties of an Electron Gas. *Soviet Physics Uspekhi*, 10(6):721–733, June 1968.

[150] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the Limits of GPU Acceleration. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*, HotPar10, page 13, USA, June 2010. USENIX Association.

[151] D. Walker. Particle-in-Cell Plasma Simulation Codes on the Connection Machine. *Computing Systems in Engineering*, 2(2):307–319, 1991.

[152] B. Wang, S. Ethier, W. Tang, T. Williams, K. Z. Ibrahim, K. Madduri, S. Williams, and L. Oliker. Kinetic Turbulence Simulations at Extreme Scale on Leadership-Class Systems. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2013.

[153] E. Wang, S. Wu, Q. Zhang, J. Liu, W. Zhang, Z. Lin, Y. Lu, Y. Du, and X. Zhu. The Gyrokinetic Particle Simulation of Fusion Plasmas on Tianhe-2 Supercomputer. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 25–32, November 2016.

[154] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. Bhalerao, and S. A. Jarvis. Parallel File System Analysis Through Application I/O Tracing. *The Computer Journal*, 56(2):141–155, February 2013.

[155] K. S. Yee. Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media. *IEEE Transactions on Antennas and Propagation*, 14(3):302–307, May 1966.

[156] C. Young, N. Gloy, and M. D. Smith. A Comparative Analysis of Schemes for Correlated Branch Prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, volume 23, pages 276–286, May 1995.

[157] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: a PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114. IEEE, May 2014.

Appendices

APPENDIX A
Performance Portable Finite Element Method Particle-in-Cell Simulations

Table A.1: Comparison of various write-conflict resolution methods for the electrostatic problem.

Approach	BDW	KNL	CL	TX2	K40	P100	V100
No Atomics	3.459	6.611	1.934	4.147	2.264	1.191	0.437
Atomics	5.207	10.808	2.631	6.771	29.451	1.215	0.537
Local Reduction	4.220	8.235	1.985	4.659	4.381	1.31	10.099
Data Replication	3.787	7.373	1.852	3.359	-	-	-

Table A.2: Comparison of various write-conflict resolution methods for the electromagnetic problem.

Approach	BDW	KNL	CL	TX2	K40	P100	V100
No Atomics	18.326	36.826	9.937	12.800	31.758	9.361	3.650
Atomics	28.711	56.416	15.467	24.580	161.832	10.064	3.913
Data Replication	17.538	35.383	9.813	14.720	-	-	-

Table A.3: Comparison of base and team particle move for the electrostatic problem.

Approach	BDW	KNL	CL	TX2	K40	P100	V100
Base	5.723	11.125	3.541	5.861	9.134	2.460	1.033
Team	9.059	20.319	4.667	8.975	3.269	1.890	0.922

Table A.4: Comparison of base and team particle move for the electromagnetic problem.

Approach	BDW	KNL	CL	TX2	K40	P100	V100
Base	28.711	56.416	15.467	24.580	161.832	10.064	3.913
Team	31.759	62.510	16.741	26.922	49.211	7.445	3.456

Table A.5: Final EMPIRE-PIC kernel timings for the electrostatic problem.

Timer	BDW	KNL	CL	TX2	K40	P100	V100
MainTimeLoop	50.971	98.203	27.949	42.847	83.309	42.315	34.076
WeightFields	5.546	10.917	2.731	3.923	8.810	3.632	1.299
Accel	1.325	2.754	0.658	1.045	6.685	1.162	0.399
Move	4.182	7.951	2.517	6.131	3.267	1.890	0.923
Sort	21.589	29.380	9.674	8.530	26.594	10.822	4.035
WeightCharge	3.815	7.344	1.858	3.359	4.381	1.214	0.529
LinearSolve	5.518	21.859	5.504	12.739	29.485	15.697	23.066
Other	8.996	17.998	5.007	7.120	4.0872	7.898	3.825

Table A.6: Final EMPIRE-PIC kernel timings for the electromagnetic problem.

Timer	BDW	KNL	CL	TX2	K40	P100	V100
MainTimeLoop	99.124	188.213	52.399	90.249	308.963	79.748	42.742
WeightFields	10.429	18.983	4.977	8.043	8.800	3.850	1.602
Accel	4.905	11.551	2.555	3.810	4.936	3.898	0.981
Move	17.871	35.432	10.043	14.858	50.778	5.805	3.865
Sort	21.474	29.420	9.692	8.863	26.594	11.943	4.035
LinearSolve	24.773	54.063	14.494	46.485	195.875	41.831	24.986
Other	19.672	38.764	10.638	8.190	21.980	12.421	7.273

Table A.7: EMPIRE-PIC scaling study data for the Haswell partition of the Trinity supercomputer.

Nodes	Size	Main Loop	Particle Update	Linear Solve
1	S	80.598	47.368	15.897
2	S	41.938	23.856	9.089
4	S	22.595	11.551	6.403
8	S	13.169	5.787	4.645
8	M	89.565	49.698	21.808
16	M	48.837	25.213	14.084
32	M	27.418	12.933	9.346
64	M	17.224	6.907	7.205
64	L	105.538	56.484	30.164
128	L	63.338	28.977	23.832
256	L	33.889	14.676	13.941
512	L	23.673	7.700	12.962
512	XL	120.624	62.916	36.646
1024	XL	68.633	33.448	24.487
2048	XL	44.354	17.934	20.411
4096	XL	24.030	9.127	11.839
4096	XXL	137.447	72.898	42.771

Table A.8: EMPIRE-PIC scaling study data for the KNL partition of the Trinity supercomputer.

Nodes	Size	Main Loop	Particle Update	Linear Solve
2	M	614.087	324.435	165.355
4	M	287.993	165.509	59.3571
8	M	165.615	89.5499	42.9057
16	M	101.556	47.7837	35.9743
16	L	693.95	459.307	97.5325
32	L	332.234	200.43	61.2402
64	L	189.42	104.91	48.1343
128	L	115.258	53.9081	42.4353
128	XL	756.543	494.701	117.035
256	XL	372.662	217.116	81.1241
512	XL	219.196	111.798	68.5907
1024	XL	143.168	59.0876	64.0335
1024	XXL	848.423	531.4	165.077
2048	XXL	468.36	248.278	141.591

Table A.9: EMPIRE-PIC scaling study data for the Astra supercomputer.

Nodes	Size	Main Loop	Particle Update	Linear Solve
2	M	207.979	131.461	32.9873
4	M	110.077	66.2716	21.9528
8	M	62.4437	35.2526	15.642
16	M	38.0486	18.7408	13.0905
16	L	240.136	151.749	39.688
32	L	132.539	78.751	28.0151
64	L	77.3367	40.8379	23.1358
128	L	49.1968	20.7963	21.3378
128	XL	266.24	163.824	47.5031
256	XL	148.838	84.5286	36.3649
512	XL	87.055	43.3115	29.455
1024	XL	86.6114	52.4775	26.8783
1024	XXL	307.225	177.231	73.3913
2048	XXL	176.804	92.3953	55.8536

Table A.10: EMPIRE-PIC scaling study data for the Sierra supercomputer.

Nodes	Size	Main Loop	Particle Update	Linear Solve
1	M	43.728	20.817	15.987
2	M	28.107	11.006	12.945
4	M	19.993	6.076	11.333
8	M	17.144	3.680	11.570
8	L	46.958	23.989	14.928
16	L	30.499	13.298	12.075
32	L	21.211	7.459	10.508
64	L	17.098	4.451	10.445
64	XL	57.138	27.658	19.729
128	XL	35.890	14.566	15.592
256	XL	26.283	8.355	14.185
512	XL	20.764	4.915	13.335

APPENDIX B
Higher-Order Particle Representation

Table B.1: Parameter scan results for the electron orbit tests using the structured mesh.

	Basis 1		Basis 2	
$r_0/\Delta x$	L_1	σ	L_1	σ
0.0	7.00×10^{-3}	2.88×10^{-3}	3.07×10^{-3}	1.13×10^{-3}
0.1	6.86×10^{-3}	2.85×10^{-3}	2.97×10^{-3}	1.08×10^{-3}
0.2	6.52×10^{-3}	2.67×10^{-3}	2.70×10^{-3}	9.62×10^{-4}
0.3	6.07×10^{-3}	2.34×10^{-3}	2.29×10^{-3}	8.08×10^{-4}
0.4	5.47×10^{-3}	1.88×10^{-3}	1.81×10^{-3}	6.38×10^{-4}
0.5	4.74×10^{-3}	1.39×10^{-3}	1.32×10^{-3}	4.73×10^{-4}
0.6	3.92×10^{-3}	9.04×10^{-4}	8.82×10^{-4}	3.01×10^{-4}
0.7	3.08×10^{-3}	5.86×10^{-4}	5.27×10^{-4}	1.55×10^{-4}
0.8	2.50×10^{-3}	4.97×10^{-4}	4.13×10^{-4}	1.02×10^{-4}
0.9	2.28×10^{-3}	6.23×10^{-4}	4.63×10^{-4}	1.42×10^{-4}
1.0	2.39×10^{-3}	8.35×10^{-4}	4.92×10^{-4}	1.72×10^{-4}
1.1	2.63×10^{-3}	1.01×10^{-3}	4.88×10^{-4}	2.19×10^{-4}
1.2	2.97×10^{-3}	1.13×10^{-3}	5.04×10^{-4}	2.37×10^{-4}
1.3	3.16×10^{-3}	1.46×10^{-3}	6.88×10^{-4}	2.16×10^{-4}
1.4	3.19×10^{-3}	1.84×10^{-3}	9.91×10^{-4}	2.47×10^{-4}
1.5	3.28×10^{-3}	1.96×10^{-3}	1.27×10^{-3}	3.83×10^{-4}
1.6	3.58×10^{-3}	1.98×10^{-3}	1.54×10^{-3}	6.49×10^{-4}
1.7	4.14×10^{-3}	2.09×10^{-3}	1.89×10^{-3}	9.14×10^{-4}
1.8	4.68×10^{-3}	2.37×10^{-3}	2.11×10^{-3}	1.03×10^{-3}
1.9	4.94×10^{-3}	2.69×10^{-3}	2.18×10^{-3}	1.06×10^{-3}
2.0	5.02×10^{-3}	2.95×10^{-3}	2.17×10^{-3}	8.55×10^{-4}

Concluded

Table B.2: Convergence study results for the electron orbit tests.

$r_0/\Delta x$	Basis	Δx	$\Delta x/2$	$\Delta x/4$	$\Delta x/8$
0.0	1	7.00×10^{-3}	1.27×10^{-3}	3.13×10^{-4}	7.64×10^{-5}
0.1	1	6.86×10^{-3}	1.24×10^{-3}	3.07×10^{-4}	7.47×10^{-5}
0.2	1	6.52×10^{-3}	1.16×10^{-3}	2.90×10^{-4}	7.02×10^{-5}
0.3	1	6.07×10^{-3}	1.06×10^{-3}	2.68×10^{-4}	6.45×10^{-5}
0.4	1	5.47×10^{-3}	9.46×10^{-4}	2.42×10^{-4}	5.87×10^{-5}
0.5	1	4.74×10^{-3}	8.60×10^{-4}	2.20×10^{-4}	5.47×10^{-5}
0.6	1	3.92×10^{-3}	8.22×10^{-4}	2.12×10^{-4}	5.32×10^{-5}
0.7	1	3.08×10^{-3}	7.95×10^{-4}	2.09×10^{-4}	5.24×10^{-5}
0.8	1	2.50×10^{-3}	7.54×10^{-4}	2.03×10^{-4}	5.10×10^{-5}
0.9	1	2.28×10^{-3}	7.24×10^{-4}	2.00×10^{-4}	5.03×10^{-5}
1.0	1	2.39×10^{-3}	6.98×10^{-4}	1.97×10^{-4}	5.00×10^{-5}
1.1	1	2.63×10^{-3}	6.68×10^{-4}	1.94×10^{-4}	4.98×10^{-5}
1.2	1	2.97×10^{-3}	6.60×10^{-4}	1.94×10^{-4}	5.05×10^{-5}
1.3	1	3.16×10^{-3}	6.62×10^{-4}	1.95×10^{-4}	5.12×10^{-5}
1.4	1	3.19×10^{-3}	6.85×10^{-4}	2.01×10^{-4}	5.23×10^{-5}

Continued on next page

$r_0/\Delta x$	Basis	Δx	$\Delta x/2$	$\Delta x/4$	$\Delta x/8$
1.5	1	3.28×10^{-3}	7.50×10^{-4}	2.16×10^{-4}	5.49×10^{-5}
0.0	2	3.07×10^{-3}	4.83×10^{-4}	8.17×10^{-5}	1.53×10^{-5}
0.1	2	2.97×10^{-3}	4.66×10^{-4}	7.91×10^{-5}	1.48×10^{-5}
0.2	2	2.70×10^{-3}	4.17×10^{-4}	7.19×10^{-5}	1.33×10^{-5}
0.3	2	2.29×10^{-3}	3.48×10^{-4}	6.11×10^{-5}	1.13×10^{-5}
0.4	2	1.81×10^{-3}	2.69×10^{-4}	4.82×10^{-5}	8.93×10^{-6}
0.5	2	1.32×10^{-3}	1.91×10^{-4}	3.49×10^{-5}	6.58×10^{-6}
0.6	2	8.82×10^{-4}	1.22×10^{-4}	2.29×10^{-5}	4.50×10^{-6}
0.7	2	5.27×10^{-4}	7.40×10^{-5}	1.44×10^{-5}	3.07×10^{-6}
0.8	2	4.13×10^{-4}	6.65×10^{-5}	1.27×10^{-5}	2.88×10^{-6}
0.9	2	4.63×10^{-4}	8.07×10^{-5}	1.41×10^{-5}	3.10×10^{-6}
1.0	2	4.92×10^{-4}	8.76×10^{-5}	1.50×10^{-5}	3.25×10^{-6}
1.1	2	4.88×10^{-4}	8.28×10^{-5}	1.45×10^{-5}	3.15×10^{-6}
1.2	2	5.04×10^{-4}	9.18×10^{-5}	1.49×10^{-5}	3.35×10^{-6}
1.3	2	6.88×10^{-4}	1.33×10^{-4}	2.27×10^{-5}	4.68×10^{-6}
1.4	2	9.91×10^{-4}	1.88×10^{-4}	3.26×10^{-5}	6.34×10^{-6}
1.5	2	1.27×10^{-3}	2.46×10^{-4}	4.27×10^{-5}	8.09×10^{-6}

Concluded

Table B.3: Parameter scan results for the electron orbit tests using the unstructured mesh.

	Basis 1		Basis 2	
$r_0/\Delta x$	L_1	σ	L_1	σ
0.0	4.33×10^{-3}	2.70×10^{-3}	1.43×10^{-3}	5.84×10^{-4}
0.1	4.21×10^{-3}	2.58×10^{-3}	1.38×10^{-3}	5.59×10^{-4}
0.2	3.92×10^{-3}	2.33×10^{-3}	1.24×10^{-3}	4.90×10^{-4}
0.3	3.51×10^{-3}	2.02×10^{-3}	1.04×10^{-3}	3.99×10^{-4}
0.4	3.10×10^{-3}	1.67×10^{-3}	8.03×10^{-4}	2.97×10^{-4}
0.5	2.67×10^{-3}	1.38×10^{-3}	5.67×10^{-4}	2.01×10^{-4}
0.6	2.32×10^{-3}	1.18×10^{-3}	3.57×10^{-4}	1.24×10^{-4}
0.7	2.04×10^{-3}	1.07×10^{-3}	1.96×10^{-4}	7.74×10^{-5}
0.8	1.85×10^{-3}	1.01×10^{-3}	1.42×10^{-4}	5.83×10^{-5}
0.9	1.71×10^{-3}	1.01×10^{-3}	1.66×10^{-4}	6.67×10^{-5}
1.0	1.59×10^{-3}	9.82×10^{-4}	1.89×10^{-4}	7.23×10^{-5}
1.1	1.47×10^{-3}	9.54×10^{-4}	1.88×10^{-4}	8.26×10^{-5}
1.2	1.38×10^{-3}	9.02×10^{-4}	1.93×10^{-4}	1.11×10^{-4}
1.3	1.33×10^{-3}	8.52×10^{-4}	2.58×10^{-4}	1.36×10^{-4}
1.4	1.35×10^{-3}	8.36×10^{-4}	3.76×10^{-4}	1.62×10^{-4}
1.5	1.44×10^{-3}	8.37×10^{-4}	5.04×10^{-4}	1.99×10^{-4}
1.6	1.57×10^{-3}	8.38×10^{-4}	6.31×10^{-4}	2.48×10^{-4}
1.7	1.71×10^{-3}	9.13×10^{-4}	7.50×10^{-4}	3.01×10^{-4}
1.8	1.84×10^{-3}	1.05×10^{-3}	8.52×10^{-4}	3.42×10^{-4}
1.9	1.94×10^{-3}	1.15×10^{-3}	9.25×10^{-4}	3.67×10^{-4}
2.0	1.99×10^{-3}	1.20×10^{-3}	9.59×10^{-4}	3.70×10^{-4}

Concluded

Table B.4: Parameter scan results for electric field in the TEM wave tests.

		E_x		E_y	
PPC	$r_0/\Delta x$	L_1	σ	L_1	σ
1	0.0	1.56×10^{-2}	1.27×10^{-3}	2.18×10^{-2}	1.27×10^{-3}
1	0.1	1.53×10^{-2}	1.33×10^{-3}	2.19×10^{-2}	1.33×10^{-3}
1	0.2	1.49×10^{-2}	1.29×10^{-3}	2.17×10^{-2}	1.29×10^{-3}
1	0.3	1.43×10^{-2}	1.29×10^{-3}	2.14×10^{-2}	1.29×10^{-3}
1	0.4	1.36×10^{-2}	1.23×10^{-3}	2.10×10^{-2}	1.23×10^{-3}
1	0.5	1.29×10^{-2}	1.17×10^{-3}	2.05×10^{-2}	1.17×10^{-3}
1	0.6	1.20×10^{-2}	1.14×10^{-3}	2.00×10^{-2}	1.14×10^{-3}
1	0.7	1.12×10^{-2}	1.09×10^{-3}	1.95×10^{-2}	1.09×10^{-3}
1	0.8	1.03×10^{-2}	1.05×10^{-3}	1.89×10^{-2}	1.05×10^{-3}
1	0.9	9.38×10^{-3}	1.00×10^{-3}	1.83×10^{-2}	1.00×10^{-3}
1	1.0	8.47×10^{-3}	9.34×10^{-4}	1.77×10^{-2}	9.34×10^{-4}
1	1.1	7.70×10^{-3}	8.62×10^{-4}	1.70×10^{-2}	8.62×10^{-4}
1	1.2	6.98×10^{-3}	8.00×10^{-4}	1.63×10^{-2}	8.00×10^{-4}
1	1.3	6.26×10^{-3}	7.27×10^{-4}	1.56×10^{-2}	7.27×10^{-4}
1	1.4	5.56×10^{-3}	6.51×10^{-4}	1.50×10^{-2}	6.51×10^{-4}
1	1.5	4.86×10^{-3}	5.57×10^{-4}	1.43×10^{-2}	5.57×10^{-4}
1	1.6	4.16×10^{-3}	4.87×10^{-4}	1.38×10^{-2}	4.87×10^{-4}
1	1.7	3.51×10^{-3}	4.28×10^{-4}	1.34×10^{-2}	4.28×10^{-4}
1	1.8	2.87×10^{-3}	3.74×10^{-4}	1.29×10^{-2}	3.74×10^{-4}
1	1.9	2.37×10^{-3}	3.15×10^{-4}	1.24×10^{-2}	3.15×10^{-4}
1	2.0	2.18×10^{-3}	2.77×10^{-4}	1.21×10^{-2}	2.77×10^{-4}
1	2.1	2.21×10^{-3}	2.74×10^{-4}	1.21×10^{-2}	2.74×10^{-4}
1	2.2	2.20×10^{-3}	2.69×10^{-4}	1.22×10^{-2}	2.69×10^{-4}
1	2.3	2.16×10^{-3}	2.66×10^{-4}	1.24×10^{-2}	2.66×10^{-4}
1	2.4	2.07×10^{-3}	2.38×10^{-4}	1.26×10^{-2}	2.38×10^{-4}
1	2.5	1.96×10^{-3}	2.19×10^{-4}	1.27×10^{-2}	2.19×10^{-4}
1	2.6	1.86×10^{-3}	2.12×10^{-4}	1.29×10^{-2}	2.12×10^{-4}
1	2.7	1.75×10^{-3}	1.80×10^{-4}	1.31×10^{-2}	1.80×10^{-4}
1	2.8	1.65×10^{-3}	1.79×10^{-4}	1.33×10^{-2}	1.79×10^{-4}
1	2.9	1.61×10^{-3}	1.68×10^{-4}	1.34×10^{-2}	1.68×10^{-4}
1	3.0	1.50×10^{-3}	1.49×10^{-4}	1.35×10^{-2}	1.49×10^{-4}
2	0.0	1.10×10^{-2}	9.54×10^{-4}	1.76×10^{-2}	9.54×10^{-4}
2	0.1	1.08×10^{-2}	8.89×10^{-4}	1.76×10^{-2}	8.89×10^{-4}
2	0.2	1.05×10^{-2}	8.78×10^{-4}	1.75×10^{-2}	8.78×10^{-4}
2	0.3	1.00×10^{-2}	8.47×10^{-4}	1.73×10^{-2}	8.47×10^{-4}
2	0.4	9.55×10^{-3}	8.10×10^{-4}	1.71×10^{-2}	8.10×10^{-4}
2	0.5	9.03×10^{-3}	7.89×10^{-4}	1.68×10^{-2}	7.89×10^{-4}
2	0.6	8.44×10^{-3}	7.89×10^{-4}	1.65×10^{-2}	7.89×10^{-4}
2	0.7	7.83×10^{-3}	7.38×10^{-4}	1.62×10^{-2}	7.38×10^{-4}
2	0.8	7.21×10^{-3}	7.18×10^{-4}	1.58×10^{-2}	7.18×10^{-4}
2	0.9	6.60×10^{-3}	6.76×10^{-4}	1.55×10^{-2}	6.76×10^{-4}
2	1.0	5.96×10^{-3}	6.50×10^{-4}	1.51×10^{-2}	6.50×10^{-4}
2	1.1	5.41×10^{-3}	6.10×10^{-4}	1.47×10^{-2}	6.10×10^{-4}
2	1.2	4.90×10^{-3}	5.75×10^{-4}	1.43×10^{-2}	5.75×10^{-4}
2	1.3	4.41×10^{-3}	4.98×10^{-4}	1.39×10^{-2}	4.98×10^{-4}
2	1.4	3.89×10^{-3}	4.24×10^{-4}	1.36×10^{-2}	4.24×10^{-4}
2	1.5	3.40×10^{-3}	3.73×10^{-4}	1.32×10^{-2}	3.73×10^{-4}
2	1.6	2.93×10^{-3}	3.24×10^{-4}	1.29×10^{-2}	3.24×10^{-4}
2	1.7	2.48×10^{-3}	2.79×10^{-4}	1.26×10^{-2}	2.79×10^{-4}
2	1.8	2.04×10^{-3}	2.61×10^{-4}	1.24×10^{-2}	2.61×10^{-4}
2	1.9	1.68×10^{-3}	2.04×10^{-4}	1.21×10^{-2}	2.04×10^{-4}
2	2.0	1.55×10^{-3}	1.99×10^{-4}	1.19×10^{-2}	1.99×10^{-4}

Continued on next page

		E_x		E_y	
PPC	$r_0/\Delta x$	L_1	σ	L_1	σ
2	2.1	1.57×10^{-3}	2.00×10^{-4}	1.19×10^{-2}	2.00×10^{-4}
2	2.2	1.56×10^{-3}	1.91×10^{-4}	1.19×10^{-2}	1.91×10^{-4}
2	2.3	1.53×10^{-3}	1.77×10^{-4}	1.19×10^{-2}	1.77×10^{-4}
2	2.4	1.47×10^{-3}	1.68×10^{-4}	1.20×10^{-2}	1.68×10^{-4}
2	2.5	1.38×10^{-3}	1.60×10^{-4}	1.21×10^{-2}	1.60×10^{-4}
2	2.6	1.31×10^{-3}	1.64×10^{-4}	1.22×10^{-2}	1.64×10^{-4}
2	2.7	1.25×10^{-3}	1.49×10^{-4}	1.23×10^{-2}	1.49×10^{-4}
2	2.8	1.19×10^{-3}	1.35×10^{-4}	1.24×10^{-2}	1.35×10^{-4}
2	2.9	1.16×10^{-3}	1.19×10^{-4}	1.25×10^{-2}	1.19×10^{-4}
2	3.0	1.07×10^{-3}	1.00×10^{-4}	1.26×10^{-2}	1.00×10^{-4}
4	0.0	7.80×10^{-3}	7.29×10^{-4}	1.53×10^{-2}	7.29×10^{-4}
4	0.1	7.64×10^{-3}	7.01×10^{-4}	1.53×10^{-2}	7.01×10^{-4}
4	0.2	7.42×10^{-3}	6.51×10^{-4}	1.52×10^{-2}	6.51×10^{-4}
4	0.3	7.13×10^{-3}	6.41×10^{-4}	1.51×10^{-2}	6.41×10^{-4}
4	0.4	6.80×10^{-3}	6.38×10^{-4}	1.50×10^{-2}	6.38×10^{-4}
4	0.5	6.44×10^{-3}	6.11×10^{-4}	1.48×10^{-2}	6.11×10^{-4}
4	0.6	6.03×10^{-3}	5.98×10^{-4}	1.46×10^{-2}	5.98×10^{-4}
4	0.7	5.58×10^{-3}	5.60×10^{-4}	1.45×10^{-2}	5.60×10^{-4}
4	0.8	5.14×10^{-3}	5.41×10^{-4}	1.43×10^{-2}	5.41×10^{-4}
4	0.9	4.69×10^{-3}	5.19×10^{-4}	1.40×10^{-2}	5.19×10^{-4}
4	1.0	4.24×10^{-3}	4.91×10^{-4}	1.38×10^{-2}	4.91×10^{-4}
4	1.1	3.86×10^{-3}	4.42×10^{-4}	1.36×10^{-2}	4.42×10^{-4}
4	1.2	3.50×10^{-3}	3.96×10^{-4}	1.34×10^{-2}	3.96×10^{-4}
4	1.3	3.14×10^{-3}	3.44×10^{-4}	1.32×10^{-2}	3.44×10^{-4}
4	1.4	2.77×10^{-3}	3.05×10^{-4}	1.29×10^{-2}	3.05×10^{-4}
4	1.5	2.41×10^{-3}	2.74×10^{-4}	1.27×10^{-2}	2.74×10^{-4}
4	1.6	2.07×10^{-3}	2.42×10^{-4}	1.25×10^{-2}	2.42×10^{-4}
4	1.7	1.75×10^{-3}	2.18×10^{-4}	1.23×10^{-2}	2.18×10^{-4}
4	1.8	1.43×10^{-3}	1.97×10^{-4}	1.21×10^{-2}	1.97×10^{-4}
4	1.9	1.18×10^{-3}	1.54×10^{-4}	1.20×10^{-2}	1.54×10^{-4}
4	2.0	1.10×10^{-3}	1.37×10^{-4}	1.18×10^{-2}	1.37×10^{-4}
4	2.1	1.11×10^{-3}	1.38×10^{-4}	1.18×10^{-2}	1.38×10^{-4}
4	2.2	1.10×10^{-3}	1.32×10^{-4}	1.17×10^{-2}	1.32×10^{-4}
4	2.3	1.06×10^{-3}	1.31×10^{-4}	1.17×10^{-2}	1.31×10^{-4}
4	2.4	1.02×10^{-3}	1.23×10^{-4}	1.17×10^{-2}	1.23×10^{-4}
4	2.5	9.75×10^{-4}	1.22×10^{-4}	1.17×10^{-2}	1.22×10^{-4}
4	2.6	9.21×10^{-4}	1.16×10^{-4}	1.18×10^{-2}	1.16×10^{-4}
4	2.7	8.67×10^{-4}	1.13×10^{-4}	1.18×10^{-2}	1.13×10^{-4}
4	2.8	8.30×10^{-4}	1.02×10^{-4}	1.19×10^{-2}	1.02×10^{-4}
4	2.9	8.12×10^{-4}	7.89×10^{-5}	1.20×10^{-2}	7.89×10^{-5}
4	3.0	7.55×10^{-4}	7.65×10^{-5}	1.22×10^{-2}	7.65×10^{-5}
8	0.0	5.52×10^{-3}	5.18×10^{-4}	1.42×10^{-2}	5.18×10^{-4}
8	0.1	5.42×10^{-3}	4.92×10^{-4}	1.42×10^{-2}	4.92×10^{-4}
8	0.2	5.28×10^{-3}	4.66×10^{-4}	1.42×10^{-2}	4.66×10^{-4}
8	0.3	5.08×10^{-3}	4.42×10^{-4}	1.41×10^{-2}	4.42×10^{-4}
8	0.4	4.83×10^{-3}	4.34×10^{-4}	1.40×10^{-2}	4.34×10^{-4}
8	0.5	4.57×10^{-3}	4.10×10^{-4}	1.40×10^{-2}	4.10×10^{-4}
8	0.6	4.27×10^{-3}	4.08×10^{-4}	1.39×10^{-2}	4.08×10^{-4}
8	0.7	3.96×10^{-3}	3.95×10^{-4}	1.38×10^{-2}	3.95×10^{-4}
8	0.8	3.65×10^{-3}	3.95×10^{-4}	1.37×10^{-2}	3.95×10^{-4}
8	0.9	3.32×10^{-3}	3.62×10^{-4}	1.35×10^{-2}	3.62×10^{-4}
8	1.0	2.99×10^{-3}	3.28×10^{-4}	1.34×10^{-2}	3.28×10^{-4}
8	1.1	2.72×10^{-3}	3.03×10^{-4}	1.33×10^{-2}	3.03×10^{-4}
8	1.2	2.47×10^{-3}	2.70×10^{-4}	1.31×10^{-2}	2.70×10^{-4}
8	1.3	2.22×10^{-3}	2.50×10^{-4}	1.29×10^{-2}	2.50×10^{-4}

Continued on next page

		E_x		E_y	
PPC	$r_0/\Delta x$	L_1	σ	L_1	σ
8	1.4	1.95×10^{-3}	2.30×10^{-4}	1.28×10^{-2}	2.30×10^{-4}
8	1.5	1.69×10^{-3}	2.01×10^{-4}	1.26×10^{-2}	2.01×10^{-4}
8	1.6	1.46×10^{-3}	1.73×10^{-4}	1.24×10^{-2}	1.73×10^{-4}
8	1.7	1.23×10^{-3}	1.50×10^{-4}	1.23×10^{-2}	1.50×10^{-4}
8	1.8	1.01×10^{-3}	1.28×10^{-4}	1.21×10^{-2}	1.28×10^{-4}
8	1.9	8.28×10^{-4}	9.99×10^{-5}	1.20×10^{-2}	9.99×10^{-5}
8	2.0	7.77×10^{-4}	1.04×10^{-4}	1.18×10^{-2}	1.04×10^{-4}
8	2.1	7.89×10^{-4}	1.06×10^{-4}	1.17×10^{-2}	1.06×10^{-4}
8	2.2	7.86×10^{-4}	9.63×10^{-5}	1.16×10^{-2}	9.63×10^{-5}
8	2.3	7.64×10^{-4}	8.54×10^{-5}	1.16×10^{-2}	8.54×10^{-5}
8	2.4	7.26×10^{-4}	8.45×10^{-5}	1.15×10^{-2}	8.45×10^{-5}
8	2.5	6.89×10^{-4}	8.15×10^{-5}	1.15×10^{-2}	8.15×10^{-5}
8	2.6	6.50×10^{-4}	7.50×10^{-5}	1.15×10^{-2}	7.50×10^{-5}
8	2.7	6.14×10^{-4}	6.78×10^{-5}	1.16×10^{-2}	6.78×10^{-5}
8	2.8	5.82×10^{-4}	6.37×10^{-5}	1.16×10^{-2}	6.37×10^{-5}
8	2.9	5.69×10^{-4}	5.52×10^{-5}	1.17×10^{-2}	5.52×10^{-5}
8	3.0	5.35×10^{-4}	5.48×10^{-5}	1.19×10^{-2}	5.48×10^{-5}
16	0.0	3.89×10^{-3}	3.25×10^{-4}	1.39×10^{-2}	3.25×10^{-4}
16	0.1	3.81×10^{-3}	3.39×10^{-4}	1.38×10^{-2}	3.39×10^{-4}
16	0.2	3.70×10^{-3}	3.34×10^{-4}	1.38×10^{-2}	3.34×10^{-4}
16	0.3	3.56×10^{-3}	3.25×10^{-4}	1.38×10^{-2}	3.25×10^{-4}
16	0.4	3.39×10^{-3}	3.14×10^{-4}	1.37×10^{-2}	3.14×10^{-4}
16	0.5	3.20×10^{-3}	3.01×10^{-4}	1.37×10^{-2}	3.01×10^{-4}
16	0.6	3.00×10^{-3}	3.02×10^{-4}	1.36×10^{-2}	3.02×10^{-4}
16	0.7	2.79×10^{-3}	3.00×10^{-4}	1.35×10^{-2}	3.00×10^{-4}
16	0.8	2.57×10^{-3}	2.90×10^{-4}	1.34×10^{-2}	2.90×10^{-4}
16	0.9	2.35×10^{-3}	2.70×10^{-4}	1.33×10^{-2}	2.70×10^{-4}
16	1.0	2.11×10^{-3}	2.49×10^{-4}	1.32×10^{-2}	2.49×10^{-4}
16	1.1	1.92×10^{-3}	2.26×10^{-4}	1.31×10^{-2}	2.26×10^{-4}
16	1.2	1.75×10^{-3}	2.07×10^{-4}	1.30×10^{-2}	2.07×10^{-4}
16	1.3	1.57×10^{-3}	1.93×10^{-4}	1.28×10^{-2}	1.93×10^{-4}
16	1.4	1.38×10^{-3}	1.68×10^{-4}	1.27×10^{-2}	1.68×10^{-4}
16	1.5	1.19×10^{-3}	1.45×10^{-4}	1.25×10^{-2}	1.45×10^{-4}
16	1.6	1.04×10^{-3}	1.25×10^{-4}	1.24×10^{-2}	1.25×10^{-4}
16	1.7	8.76×10^{-4}	1.07×10^{-4}	1.22×10^{-2}	1.07×10^{-4}
16	1.8	7.14×10^{-4}	9.31×10^{-5}	1.21×10^{-2}	9.31×10^{-5}
16	1.9	5.80×10^{-4}	6.99×10^{-5}	1.19×10^{-2}	6.99×10^{-5}
16	2.0	5.43×10^{-4}	6.55×10^{-5}	1.18×10^{-2}	6.55×10^{-5}
16	2.1	5.58×10^{-4}	6.06×10^{-5}	1.17×10^{-2}	6.06×10^{-5}
16	2.2	5.55×10^{-4}	5.99×10^{-5}	1.16×10^{-2}	5.99×10^{-5}
16	2.3	5.43×10^{-4}	5.89×10^{-5}	1.15×10^{-2}	5.89×10^{-5}
16	2.4	5.15×10^{-4}	6.06×10^{-5}	1.15×10^{-2}	6.06×10^{-5}
16	2.5	4.93×10^{-4}	5.56×10^{-5}	1.15×10^{-2}	5.56×10^{-5}
16	2.6	4.63×10^{-4}	5.49×10^{-5}	1.15×10^{-2}	5.49×10^{-5}
16	2.7	4.34×10^{-4}	5.54×10^{-5}	1.15×10^{-2}	5.54×10^{-5}
16	2.8	4.11×10^{-4}	5.17×10^{-5}	1.16×10^{-2}	5.17×10^{-5}
16	2.9	4.03×10^{-4}	4.25×10^{-5}	1.16×10^{-2}	4.25×10^{-5}
16	3.0	3.77×10^{-4}	3.76×10^{-5}	1.18×10^{-2}	3.76×10^{-5}
32	0.0	2.72×10^{-3}	2.41×10^{-4}	1.38×10^{-2}	2.41×10^{-4}
32	0.1	2.67×10^{-3}	2.52×10^{-4}	1.38×10^{-2}	2.52×10^{-4}
32	0.2	2.60×10^{-3}	2.52×10^{-4}	1.38×10^{-2}	2.52×10^{-4}
32	0.3	2.49×10^{-3}	2.42×10^{-4}	1.38×10^{-2}	2.42×10^{-4}
32	0.4	2.37×10^{-3}	2.27×10^{-4}	1.37×10^{-2}	2.27×10^{-4}
32	0.5	2.24×10^{-3}	2.16×10^{-4}	1.37×10^{-2}	2.16×10^{-4}
32	0.6	2.10×10^{-3}	1.92×10^{-4}	1.36×10^{-2}	1.92×10^{-4}

Continued on next page

		E_x		E_y	
PPC	$r_0/\Delta x$	L_1	σ	L_1	σ
32	0.7	1.96×10^{-3}	1.88×10^{-4}	1.35×10^{-2}	1.88×10^{-4}
32	0.8	1.81×10^{-3}	1.78×10^{-4}	1.34×10^{-2}	1.78×10^{-4}
32	0.9	1.64×10^{-3}	1.67×10^{-4}	1.33×10^{-2}	1.67×10^{-4}
32	1.0	1.47×10^{-3}	1.64×10^{-4}	1.32×10^{-2}	1.64×10^{-4}
32	1.1	1.33×10^{-3}	1.55×10^{-4}	1.31×10^{-2}	1.55×10^{-4}
32	1.2	1.21×10^{-3}	1.45×10^{-4}	1.29×10^{-2}	1.45×10^{-4}
32	1.3	1.08×10^{-3}	1.24×10^{-4}	1.28×10^{-2}	1.24×10^{-4}
32	1.4	9.52×10^{-4}	1.01×10^{-4}	1.27×10^{-2}	1.01×10^{-4}
32	1.5	8.32×10^{-4}	8.54×10^{-5}	1.25×10^{-2}	8.54×10^{-5}
32	1.6	7.23×10^{-4}	7.86×10^{-5}	1.24×10^{-2}	7.86×10^{-5}
32	1.7	6.13×10^{-4}	7.20×10^{-5}	1.22×10^{-2}	7.20×10^{-5}
32	1.8	5.01×10^{-4}	6.18×10^{-5}	1.21×10^{-2}	6.18×10^{-5}
32	1.9	4.09×10^{-4}	4.84×10^{-5}	1.19×10^{-2}	4.84×10^{-5}
32	2.0	3.81×10^{-4}	4.73×10^{-5}	1.18×10^{-2}	4.73×10^{-5}
32	2.1	3.95×10^{-4}	4.57×10^{-5}	1.17×10^{-2}	4.57×10^{-5}
32	2.2	3.93×10^{-4}	5.13×10^{-5}	1.16×10^{-2}	5.13×10^{-5}
32	2.3	3.82×10^{-4}	4.75×10^{-5}	1.15×10^{-2}	4.75×10^{-5}
32	2.4	3.63×10^{-4}	4.24×10^{-5}	1.15×10^{-2}	4.24×10^{-5}
32	2.5	3.47×10^{-4}	3.69×10^{-5}	1.14×10^{-2}	3.69×10^{-5}
32	2.6	3.26×10^{-4}	3.85×10^{-5}	1.14×10^{-2}	3.85×10^{-5}
32	2.7	3.07×10^{-4}	4.00×10^{-5}	1.14×10^{-2}	4.00×10^{-5}
32	2.8	2.91×10^{-4}	3.96×10^{-5}	1.15×10^{-2}	3.96×10^{-5}
32	2.9	2.85×10^{-4}	3.35×10^{-5}	1.16×10^{-2}	3.35×10^{-5}
32	3.0	2.71×10^{-4}	2.86×10^{-5}	1.17×10^{-2}	2.86×10^{-5}
64	0.0	1.93×10^{-3}	1.64×10^{-4}	1.38×10^{-2}	1.64×10^{-4}
64	0.1	1.90×10^{-3}	1.63×10^{-4}	1.38×10^{-2}	1.63×10^{-4}
64	0.2	1.84×10^{-3}	1.54×10^{-4}	1.38×10^{-2}	1.54×10^{-4}
64	0.3	1.77×10^{-3}	1.48×10^{-4}	1.37×10^{-2}	1.48×10^{-4}
64	0.4	1.69×10^{-3}	1.37×10^{-4}	1.37×10^{-2}	1.37×10^{-4}
64	0.5	1.59×10^{-3}	1.29×10^{-4}	1.36×10^{-2}	1.29×10^{-4}
64	0.6	1.48×10^{-3}	1.31×10^{-4}	1.36×10^{-2}	1.31×10^{-4}
64	0.7	1.38×10^{-3}	1.20×10^{-4}	1.35×10^{-2}	1.20×10^{-4}
64	0.8	1.26×10^{-3}	1.13×10^{-4}	1.34×10^{-2}	1.13×10^{-4}
64	0.9	1.15×10^{-3}	1.08×10^{-4}	1.33×10^{-2}	1.08×10^{-4}
64	1.0	1.04×10^{-3}	1.04×10^{-4}	1.32×10^{-2}	1.04×10^{-4}
64	1.1	9.50×10^{-4}	9.96×10^{-5}	1.31×10^{-2}	9.96×10^{-5}
64	1.2	8.59×10^{-4}	9.38×10^{-5}	1.29×10^{-2}	9.38×10^{-5}
64	1.3	7.67×10^{-4}	8.26×10^{-5}	1.28×10^{-2}	8.26×10^{-5}
64	1.4	6.79×10^{-4}	7.07×10^{-5}	1.26×10^{-2}	7.07×10^{-5}
64	1.5	5.94×10^{-4}	6.51×10^{-5}	1.25×10^{-2}	6.51×10^{-5}
64	1.6	5.16×10^{-4}	6.24×10^{-5}	1.24×10^{-2}	6.24×10^{-5}
64	1.7	4.37×10^{-4}	5.36×10^{-5}	1.22×10^{-2}	5.36×10^{-5}
64	1.8	3.56×10^{-4}	4.41×10^{-5}	1.21×10^{-2}	4.41×10^{-5}
64	1.9	2.92×10^{-4}	3.59×10^{-5}	1.20×10^{-2}	3.59×10^{-5}
64	2.0	2.69×10^{-4}	3.30×10^{-5}	1.18×10^{-2}	3.30×10^{-5}
64	2.1	2.75×10^{-4}	3.31×10^{-5}	1.17×10^{-2}	3.31×10^{-5}
64	2.2	2.75×10^{-4}	3.57×10^{-5}	1.16×10^{-2}	3.57×10^{-5}
64	2.3	2.70×10^{-4}	3.30×10^{-5}	1.15×10^{-2}	3.30×10^{-5}
64	2.4	2.58×10^{-4}	3.05×10^{-5}	1.15×10^{-2}	3.05×10^{-5}
64	2.5	2.46×10^{-4}	2.89×10^{-5}	1.14×10^{-2}	2.89×10^{-5}
64	2.6	2.34×10^{-4}	2.95×10^{-5}	1.14×10^{-2}	2.95×10^{-5}
64	2.7	2.22×10^{-4}	2.73×10^{-5}	1.14×10^{-2}	2.73×10^{-5}
64	2.8	2.08×10^{-4}	2.52×10^{-5}	1.15×10^{-2}	2.52×10^{-5}
64	2.9	2.01×10^{-4}	1.99×10^{-5}	1.16×10^{-2}	1.99×10^{-5}
64	3.0	1.90×10^{-4}	1.98×10^{-5}	1.17×10^{-2}	1.98×10^{-5}

Concluded

Table B.5: Parameter scan results for magnetic field in the TEM wave tests.

		B_x		B_y	
PPC	$r_0/\Delta x$	L_1	σ	L_1	σ
1	0.0	1.14×10^{-2}	6.27×10^{-4}	7.22×10^{-5}	6.27×10^{-4}
1	0.1	1.14×10^{-2}	6.24×10^{-4}	7.16×10^{-5}	6.24×10^{-4}
1	0.2	1.14×10^{-2}	6.23×10^{-4}	6.99×10^{-5}	6.23×10^{-4}
1	0.3	1.14×10^{-2}	6.22×10^{-4}	6.48×10^{-5}	6.22×10^{-4}
1	0.4	1.13×10^{-2}	6.21×10^{-4}	6.04×10^{-5}	6.21×10^{-4}
1	0.5	1.12×10^{-2}	6.19×10^{-4}	5.41×10^{-5}	6.19×10^{-4}
1	0.6	1.12×10^{-2}	6.17×10^{-4}	4.83×10^{-5}	6.17×10^{-4}
1	0.7	1.11×10^{-2}	6.17×10^{-4}	4.24×10^{-5}	6.17×10^{-4}
1	0.8	1.09×10^{-2}	6.16×10^{-4}	3.66×10^{-5}	6.16×10^{-4}
1	0.9	1.08×10^{-2}	6.14×10^{-4}	3.09×10^{-5}	6.14×10^{-4}
1	1.0	1.07×10^{-2}	6.13×10^{-4}	2.54×10^{-5}	6.13×10^{-4}
1	1.1	1.05×10^{-2}	6.12×10^{-4}	2.05×10^{-5}	6.12×10^{-4}
1	1.2	1.04×10^{-2}	6.11×10^{-4}	1.63×10^{-5}	6.11×10^{-4}
1	1.3	1.02×10^{-2}	6.09×10^{-4}	1.27×10^{-5}	6.09×10^{-4}
1	1.4	1.01×10^{-2}	6.06×10^{-4}	9.66×10^{-6}	6.06×10^{-4}
1	1.5	9.93×10^{-3}	6.03×10^{-4}	7.19×10^{-6}	6.03×10^{-4}
1	1.6	9.77×10^{-3}	6.01×10^{-4}	5.22×10^{-6}	6.01×10^{-4}
1	1.7	9.63×10^{-3}	6.00×10^{-4}	3.72×10^{-6}	6.00×10^{-4}
1	1.8	9.48×10^{-3}	6.00×10^{-4}	2.60×10^{-6}	6.00×10^{-4}
1	1.9	9.33×10^{-3}	6.00×10^{-4}	1.77×10^{-6}	6.00×10^{-4}
1	2.0	9.18×10^{-3}	5.99×10^{-4}	1.30×10^{-6}	5.99×10^{-4}
1	2.1	9.04×10^{-3}	5.97×10^{-4}	1.03×10^{-6}	5.97×10^{-4}
1	2.2	8.92×10^{-3}	5.93×10^{-4}	7.93×10^{-7}	5.93×10^{-4}
1	2.3	8.82×10^{-3}	5.84×10^{-4}	6.23×10^{-7}	5.84×10^{-4}
1	2.4	8.74×10^{-3}	5.81×10^{-4}	4.98×10^{-7}	5.81×10^{-4}
1	2.5	8.68×10^{-3}	5.75×10^{-4}	4.11×10^{-7}	5.75×10^{-4}
1	2.6	8.63×10^{-3}	5.72×10^{-4}	3.55×10^{-7}	5.72×10^{-4}
1	2.7	8.63×10^{-3}	5.67×10^{-4}	3.15×10^{-7}	5.67×10^{-4}
1	2.8	8.67×10^{-3}	5.59×10^{-4}	2.91×10^{-7}	5.59×10^{-4}
1	2.9	8.74×10^{-3}	5.53×10^{-4}	3.05×10^{-7}	5.53×10^{-4}
1	3.0	8.83×10^{-3}	5.45×10^{-4}	3.14×10^{-7}	5.45×10^{-4}
2	0.0	1.13×10^{-2}	4.16×10^{-4}	3.50×10^{-5}	4.16×10^{-4}
2	0.1	1.13×10^{-2}	4.15×10^{-4}	3.43×10^{-5}	4.15×10^{-4}
2	0.2	1.13×10^{-2}	4.15×10^{-4}	3.28×10^{-5}	4.15×10^{-4}
2	0.3	1.13×10^{-2}	4.14×10^{-4}	3.06×10^{-5}	4.14×10^{-4}
2	0.4	1.12×10^{-2}	4.13×10^{-4}	2.80×10^{-5}	4.13×10^{-4}
2	0.5	1.12×10^{-2}	4.12×10^{-4}	2.55×10^{-5}	4.12×10^{-4}
2	0.6	1.11×10^{-2}	4.11×10^{-4}	2.30×10^{-5}	4.11×10^{-4}
2	0.7	1.10×10^{-2}	4.10×10^{-4}	2.00×10^{-5}	4.10×10^{-4}
2	0.8	1.09×10^{-2}	4.08×10^{-4}	1.75×10^{-5}	4.08×10^{-4}
2	0.9	1.08×10^{-2}	4.07×10^{-4}	1.48×10^{-5}	4.07×10^{-4}
2	1.0	1.07×10^{-2}	4.07×10^{-4}	1.22×10^{-5}	4.07×10^{-4}
2	1.1	1.05×10^{-2}	4.07×10^{-4}	9.75×10^{-6}	4.07×10^{-4}
2	1.2	1.04×10^{-2}	4.07×10^{-4}	7.73×10^{-6}	4.07×10^{-4}
2	1.3	1.02×10^{-2}	4.09×10^{-4}	6.01×10^{-6}	4.09×10^{-4}
2	1.4	1.01×10^{-2}	4.08×10^{-4}	4.51×10^{-6}	4.08×10^{-4}
2	1.5	9.91×10^{-3}	4.06×10^{-4}	3.44×10^{-6}	4.06×10^{-4}
2	1.6	9.76×10^{-3}	4.05×10^{-4}	2.55×10^{-6}	4.05×10^{-4}
2	1.7	9.62×10^{-3}	4.04×10^{-4}	1.82×10^{-6}	4.04×10^{-4}
2	1.8	9.48×10^{-3}	4.03×10^{-4}	1.25×10^{-6}	4.03×10^{-4}
2	1.9	9.34×10^{-3}	4.03×10^{-4}	8.53×10^{-7}	4.03×10^{-4}
2	2.0	9.19×10^{-3}	4.01×10^{-4}	6.25×10^{-7}	4.01×10^{-4}

Continued on next page

		B_x		B_y	
PPC	$r_0/\Delta x$	L_1	σ	L_1	σ
2	2.1	9.04×10^{-3}	4.02×10^{-4}	5.02×10^{-7}	4.02×10^{-4}
2	2.2	8.91×10^{-3}	4.00×10^{-4}	3.86×10^{-7}	4.00×10^{-4}
2	2.3	8.82×10^{-3}	3.98×10^{-4}	3.09×10^{-7}	3.98×10^{-4}
2	2.4	8.75×10^{-3}	3.94×10^{-4}	2.35×10^{-7}	3.94×10^{-4}
2	2.5	8.69×10^{-3}	3.92×10^{-4}	1.94×10^{-7}	3.92×10^{-4}
2	2.6	8.64×10^{-3}	3.88×10^{-4}	1.63×10^{-7}	3.88×10^{-4}
2	2.7	8.63×10^{-3}	3.88×10^{-4}	1.53×10^{-7}	3.88×10^{-4}
2	2.8	8.68×10^{-3}	3.85×10^{-4}	1.49×10^{-7}	3.85×10^{-4}
2	2.9	8.75×10^{-3}	3.81×10^{-4}	1.58×10^{-7}	3.81×10^{-4}
2	3.0	8.83×10^{-3}	3.77×10^{-4}	1.58×10^{-7}	3.77×10^{-4}
4	0.0	1.13×10^{-2}	3.08×10^{-4}	1.85×10^{-5}	3.08×10^{-4}
4	0.1	1.13×10^{-2}	3.08×10^{-4}	1.83×10^{-5}	3.08×10^{-4}
4	0.2	1.13×10^{-2}	3.08×10^{-4}	1.77×10^{-5}	3.08×10^{-4}
4	0.3	1.12×10^{-2}	3.07×10^{-4}	1.67×10^{-5}	3.07×10^{-4}
4	0.4	1.12×10^{-2}	3.07×10^{-4}	1.53×10^{-5}	3.07×10^{-4}
4	0.5	1.11×10^{-2}	3.06×10^{-4}	1.39×10^{-5}	3.06×10^{-4}
4	0.6	1.11×10^{-2}	3.05×10^{-4}	1.22×10^{-5}	3.05×10^{-4}
4	0.7	1.10×10^{-2}	3.05×10^{-4}	1.05×10^{-5}	3.05×10^{-4}
4	0.8	1.09×10^{-2}	3.04×10^{-4}	8.90×10^{-6}	3.04×10^{-4}
4	0.9	1.08×10^{-2}	3.03×10^{-4}	7.45×10^{-6}	3.03×10^{-4}
4	1.0	1.07×10^{-2}	3.03×10^{-4}	6.06×10^{-6}	3.03×10^{-4}
4	1.1	1.05×10^{-2}	3.02×10^{-4}	4.89×10^{-6}	3.02×10^{-4}
4	1.2	1.04×10^{-2}	3.02×10^{-4}	3.92×10^{-6}	3.02×10^{-4}
4	1.3	1.02×10^{-2}	3.01×10^{-4}	3.07×10^{-6}	3.01×10^{-4}
4	1.4	1.01×10^{-2}	3.01×10^{-4}	2.31×10^{-6}	3.01×10^{-4}
4	1.5	9.91×10^{-3}	3.00×10^{-4}	1.72×10^{-6}	3.00×10^{-4}
4	1.6	9.76×10^{-3}	2.96×10^{-4}	1.26×10^{-6}	2.96×10^{-4}
4	1.7	9.63×10^{-3}	2.93×10^{-4}	9.07×10^{-7}	2.93×10^{-4}
4	1.8	9.49×10^{-3}	2.93×10^{-4}	6.19×10^{-7}	2.93×10^{-4}
4	1.9	9.35×10^{-3}	2.92×10^{-4}	4.23×10^{-7}	2.92×10^{-4}
4	2.0	9.20×10^{-3}	2.91×10^{-4}	3.12×10^{-7}	2.91×10^{-4}
4	2.1	9.05×10^{-3}	2.90×10^{-4}	2.61×10^{-7}	2.90×10^{-4}
4	2.2	8.92×10^{-3}	2.90×10^{-4}	2.12×10^{-7}	2.90×10^{-4}
4	2.3	8.83×10^{-3}	2.85×10^{-4}	1.65×10^{-7}	2.85×10^{-4}
4	2.4	8.77×10^{-3}	2.82×10^{-4}	1.28×10^{-7}	2.82×10^{-4}
4	2.5	8.71×10^{-3}	2.81×10^{-4}	1.00×10^{-7}	2.81×10^{-4}
4	2.6	8.65×10^{-3}	2.79×10^{-4}	7.87×10^{-8}	2.79×10^{-4}
4	2.7	8.63×10^{-3}	2.77×10^{-4}	6.70×10^{-8}	2.77×10^{-4}
4	2.8	8.69×10^{-3}	2.74×10^{-4}	6.67×10^{-8}	2.74×10^{-4}
4	2.9	8.76×10^{-3}	2.72×10^{-4}	6.89×10^{-8}	2.72×10^{-4}
4	3.0	8.84×10^{-3}	2.70×10^{-4}	7.23×10^{-8}	2.70×10^{-4}
8	0.0	1.13×10^{-2}	2.21×10^{-4}	8.98×10^{-6}	2.21×10^{-4}
8	0.1	1.13×10^{-2}	2.21×10^{-4}	8.71×10^{-6}	2.21×10^{-4}
8	0.2	1.12×10^{-2}	2.21×10^{-4}	8.57×10^{-6}	2.21×10^{-4}
8	0.3	1.12×10^{-2}	2.21×10^{-4}	8.06×10^{-6}	2.21×10^{-4}
8	0.4	1.12×10^{-2}	2.20×10^{-4}	7.37×10^{-6}	2.20×10^{-4}
8	0.5	1.11×10^{-2}	2.20×10^{-4}	6.67×10^{-6}	2.20×10^{-4}
8	0.6	1.10×10^{-2}	2.19×10^{-4}	5.97×10^{-6}	2.19×10^{-4}
8	0.7	1.10×10^{-2}	2.19×10^{-4}	5.22×10^{-6}	2.19×10^{-4}
8	0.8	1.09×10^{-2}	2.18×10^{-4}	4.41×10^{-6}	2.18×10^{-4}
8	0.9	1.08×10^{-2}	2.18×10^{-4}	3.68×10^{-6}	2.18×10^{-4}
8	1.0	1.06×10^{-2}	2.17×10^{-4}	3.02×10^{-6}	2.17×10^{-4}
8	1.1	1.05×10^{-2}	2.17×10^{-4}	2.44×10^{-6}	2.17×10^{-4}
8	1.2	1.04×10^{-2}	2.16×10^{-4}	1.95×10^{-6}	2.16×10^{-4}
8	1.3	1.02×10^{-2}	2.16×10^{-4}	1.53×10^{-6}	2.16×10^{-4}

Continued on next page

		B_x		B_y	
PPC	$r_0/\Delta x$	L_1	σ	L_1	σ
8	1.4	1.00×10^{-2}	2.15×10^{-4}	1.16×10^{-6}	2.15×10^{-4}
8	1.5	9.89×10^{-3}	2.17×10^{-4}	8.54×10^{-7}	2.17×10^{-4}
8	1.6	9.75×10^{-3}	2.19×10^{-4}	6.19×10^{-7}	2.19×10^{-4}
8	1.7	9.62×10^{-3}	2.19×10^{-4}	4.41×10^{-7}	2.19×10^{-4}
8	1.8	9.49×10^{-3}	2.18×10^{-4}	3.06×10^{-7}	2.18×10^{-4}
8	1.9	9.35×10^{-3}	2.18×10^{-4}	2.01×10^{-7}	2.18×10^{-4}
8	2.0	9.20×10^{-3}	2.17×10^{-4}	1.50×10^{-7}	2.17×10^{-4}
8	2.1	9.04×10^{-3}	2.17×10^{-4}	1.25×10^{-7}	2.17×10^{-4}
8	2.2	8.90×10^{-3}	2.18×10^{-4}	1.00×10^{-7}	2.18×10^{-4}
8	2.3	8.83×10^{-3}	2.19×10^{-4}	7.97×10^{-8}	2.19×10^{-4}
8	2.4	8.77×10^{-3}	2.18×10^{-4}	6.27×10^{-8}	2.18×10^{-4}
8	2.5	8.70×10^{-3}	2.17×10^{-4}	4.97×10^{-8}	2.17×10^{-4}
8	2.6	8.64×10^{-3}	2.16×10^{-4}	4.07×10^{-8}	2.16×10^{-4}
8	2.7	8.62×10^{-3}	2.15×10^{-4}	3.58×10^{-8}	2.15×10^{-4}
8	2.8	8.69×10^{-3}	2.14×10^{-4}	3.52×10^{-8}	2.14×10^{-4}
8	2.9	8.76×10^{-3}	2.13×10^{-4}	3.54×10^{-8}	2.13×10^{-4}
8	3.0	8.83×10^{-3}	2.12×10^{-4}	3.79×10^{-8}	2.12×10^{-4}
16	0.0	1.13×10^{-2}	1.51×10^{-4}	4.47×10^{-6}	1.51×10^{-4}
16	0.1	1.12×10^{-2}	1.51×10^{-4}	4.47×10^{-6}	1.51×10^{-4}
16	0.2	1.12×10^{-2}	1.51×10^{-4}	4.34×10^{-6}	1.51×10^{-4}
16	0.3	1.12×10^{-2}	1.51×10^{-4}	4.10×10^{-6}	1.51×10^{-4}
16	0.4	1.12×10^{-2}	1.50×10^{-4}	3.76×10^{-6}	1.50×10^{-4}
16	0.5	1.11×10^{-2}	1.50×10^{-4}	3.44×10^{-6}	1.50×10^{-4}
16	0.6	1.10×10^{-2}	1.50×10^{-4}	3.04×10^{-6}	1.50×10^{-4}
16	0.7	1.09×10^{-2}	1.50×10^{-4}	2.68×10^{-6}	1.50×10^{-4}
16	0.8	1.09×10^{-2}	1.49×10^{-4}	2.32×10^{-6}	1.49×10^{-4}
16	0.9	1.07×10^{-2}	1.49×10^{-4}	1.95×10^{-6}	1.49×10^{-4}
16	1.0	1.06×10^{-2}	1.48×10^{-4}	1.59×10^{-6}	1.48×10^{-4}
16	1.1	1.05×10^{-2}	1.48×10^{-4}	1.28×10^{-6}	1.48×10^{-4}
16	1.2	1.04×10^{-2}	1.48×10^{-4}	1.01×10^{-6}	1.48×10^{-4}
16	1.3	1.02×10^{-2}	1.47×10^{-4}	7.83×10^{-7}	1.47×10^{-4}
16	1.4	1.00×10^{-2}	1.47×10^{-4}	5.94×10^{-7}	1.47×10^{-4}
16	1.5	9.87×10^{-3}	1.50×10^{-4}	4.37×10^{-7}	1.50×10^{-4}
16	1.6	9.74×10^{-3}	1.52×10^{-4}	3.18×10^{-7}	1.52×10^{-4}
16	1.7	9.61×10^{-3}	1.52×10^{-4}	2.25×10^{-7}	1.52×10^{-4}
16	1.8	9.48×10^{-3}	1.52×10^{-4}	1.55×10^{-7}	1.52×10^{-4}
16	1.9	9.34×10^{-3}	1.51×10^{-4}	1.00×10^{-7}	1.51×10^{-4}
16	2.0	9.19×10^{-3}	1.51×10^{-4}	7.26×10^{-8}	1.51×10^{-4}
16	2.1	9.04×10^{-3}	1.51×10^{-4}	6.04×10^{-8}	1.51×10^{-4}
16	2.2	8.89×10^{-3}	1.54×10^{-4}	4.90×10^{-8}	1.54×10^{-4}
16	2.3	8.82×10^{-3}	1.54×10^{-4}	3.99×10^{-8}	1.54×10^{-4}
16	2.4	8.76×10^{-3}	1.53×10^{-4}	3.22×10^{-8}	1.53×10^{-4}
16	2.5	8.69×10^{-3}	1.52×10^{-4}	2.63×10^{-8}	1.52×10^{-4}
16	2.6	8.63×10^{-3}	1.52×10^{-4}	2.17×10^{-8}	1.52×10^{-4}
16	2.7	8.61×10^{-3}	1.52×10^{-4}	1.86×10^{-8}	1.52×10^{-4}
16	2.8	8.67×10^{-3}	1.51×10^{-4}	1.78×10^{-8}	1.51×10^{-4}
16	2.9	8.74×10^{-3}	1.50×10^{-4}	1.81×10^{-8}	1.50×10^{-4}
16	3.0	8.82×10^{-3}	1.49×10^{-4}	1.92×10^{-8}	1.49×10^{-4}
32	0.0	1.12×10^{-2}	1.11×10^{-4}	2.32×10^{-6}	1.11×10^{-4}
32	0.1	1.12×10^{-2}	1.11×10^{-4}	2.33×10^{-6}	1.11×10^{-4}
32	0.2	1.12×10^{-2}	1.11×10^{-4}	2.25×10^{-6}	1.11×10^{-4}
32	0.3	1.12×10^{-2}	1.11×10^{-4}	2.11×10^{-6}	1.11×10^{-4}
32	0.4	1.11×10^{-2}	1.11×10^{-4}	1.95×10^{-6}	1.11×10^{-4}
32	0.5	1.11×10^{-2}	1.10×10^{-4}	1.73×10^{-6}	1.10×10^{-4}
32	0.6	1.10×10^{-2}	1.10×10^{-4}	1.52×10^{-6}	1.10×10^{-4}

Continued on next page

		B_x		B_y	
PPC	$r_0/\Delta x$	L_1	σ	L_1	σ
32	0.7	1.09×10^{-2}	1.10×10^{-4}	1.32×10^{-6}	1.10×10^{-4}
32	0.8	1.08×10^{-2}	1.10×10^{-4}	1.14×10^{-6}	1.10×10^{-4}
32	0.9	1.07×10^{-2}	1.09×10^{-4}	9.50×10^{-7}	1.09×10^{-4}
32	1.0	1.06×10^{-2}	1.09×10^{-4}	7.75×10^{-7}	1.09×10^{-4}
32	1.1	1.05×10^{-2}	1.09×10^{-4}	6.26×10^{-7}	1.09×10^{-4}
32	1.2	1.03×10^{-2}	1.09×10^{-4}	4.98×10^{-7}	1.09×10^{-4}
32	1.3	1.02×10^{-2}	1.08×10^{-4}	3.87×10^{-7}	1.08×10^{-4}
32	1.4	1.00×10^{-2}	1.08×10^{-4}	2.93×10^{-7}	1.08×10^{-4}
32	1.5	9.85×10^{-3}	1.11×10^{-4}	2.17×10^{-7}	1.11×10^{-4}
32	1.6	9.72×10^{-3}	1.13×10^{-4}	1.56×10^{-7}	1.13×10^{-4}
32	1.7	9.59×10^{-3}	1.13×10^{-4}	1.10×10^{-7}	1.13×10^{-4}
32	1.8	9.46×10^{-3}	1.12×10^{-4}	7.60×10^{-8}	1.12×10^{-4}
32	1.9	9.32×10^{-3}	1.12×10^{-4}	5.04×10^{-8}	1.12×10^{-4}
32	2.0	9.17×10^{-3}	1.12×10^{-4}	3.80×10^{-8}	1.12×10^{-4}
32	2.1	9.02×10^{-3}	1.12×10^{-4}	3.16×10^{-8}	1.12×10^{-4}
32	2.2	8.86×10^{-3}	1.14×10^{-4}	2.50×10^{-8}	1.14×10^{-4}
32	2.3	8.80×10^{-3}	1.14×10^{-4}	1.99×10^{-8}	1.14×10^{-4}
32	2.4	8.74×10^{-3}	1.14×10^{-4}	1.58×10^{-8}	1.14×10^{-4}
32	2.5	8.67×10^{-3}	1.13×10^{-4}	1.29×10^{-8}	1.13×10^{-4}
32	2.6	8.61×10^{-3}	1.12×10^{-4}	1.06×10^{-8}	1.12×10^{-4}
32	2.7	8.59×10^{-3}	1.13×10^{-4}	9.08×10^{-9}	1.13×10^{-4}
32	2.8	8.66×10^{-3}	1.12×10^{-4}	8.84×10^{-9}	1.12×10^{-4}
32	2.9	8.73×10^{-3}	1.11×10^{-4}	9.62×10^{-9}	1.11×10^{-4}
32	3.0	8.80×10^{-3}	1.10×10^{-4}	1.00×10^{-8}	1.10×10^{-4}
64	0.0	1.12×10^{-2}	7.30×10^{-5}	1.10×10^{-6}	7.30×10^{-5}
64	0.1	1.12×10^{-2}	7.30×10^{-5}	1.09×10^{-6}	7.30×10^{-5}
64	0.2	1.12×10^{-2}	7.30×10^{-5}	1.05×10^{-6}	7.30×10^{-5}
64	0.3	1.12×10^{-2}	7.29×10^{-5}	9.88×10^{-7}	7.29×10^{-5}
64	0.4	1.11×10^{-2}	7.28×10^{-5}	9.01×10^{-7}	7.28×10^{-5}
64	0.5	1.11×10^{-2}	7.27×10^{-5}	8.09×10^{-7}	7.27×10^{-5}
64	0.6	1.10×10^{-2}	7.25×10^{-5}	7.07×10^{-7}	7.25×10^{-5}
64	0.7	1.09×10^{-2}	7.23×10^{-5}	6.16×10^{-7}	7.23×10^{-5}
64	0.8	1.08×10^{-2}	7.22×10^{-5}	5.30×10^{-7}	7.22×10^{-5}
64	0.9	1.07×10^{-2}	7.20×10^{-5}	4.43×10^{-7}	7.20×10^{-5}
64	1.0	1.06×10^{-2}	7.19×10^{-5}	3.62×10^{-7}	7.19×10^{-5}
64	1.1	1.05×10^{-2}	7.17×10^{-5}	2.90×10^{-7}	7.17×10^{-5}
64	1.2	1.03×10^{-2}	7.15×10^{-5}	2.29×10^{-7}	7.15×10^{-5}
64	1.3	1.02×10^{-2}	7.12×10^{-5}	1.78×10^{-7}	7.12×10^{-5}
64	1.4	1.00×10^{-2}	7.10×10^{-5}	1.36×10^{-7}	7.10×10^{-5}
64	1.5	9.84×10^{-3}	7.19×10^{-5}	1.00×10^{-7}	7.19×10^{-5}
64	1.6	9.71×10^{-3}	7.17×10^{-5}	7.30×10^{-8}	7.17×10^{-5}
64	1.7	9.58×10^{-3}	7.15×10^{-5}	5.25×10^{-8}	7.15×10^{-5}
64	1.8	9.45×10^{-3}	7.13×10^{-5}	3.62×10^{-8}	7.13×10^{-5}
64	1.9	9.31×10^{-3}	7.11×10^{-5}	2.36×10^{-8}	7.11×10^{-5}
64	2.0	9.16×10^{-3}	7.09×10^{-5}	1.72×10^{-8}	7.09×10^{-5}
64	2.1	9.01×10^{-3}	7.07×10^{-5}	1.43×10^{-8}	7.07×10^{-5}
64	2.2	8.85×10^{-3}	7.07×10^{-5}	1.17×10^{-8}	7.07×10^{-5}
64	2.3	8.79×10^{-3}	7.07×10^{-5}	9.35×10^{-9}	7.07×10^{-5}
64	2.4	8.73×10^{-3}	7.04×10^{-5}	7.53×10^{-9}	7.04×10^{-5}
64	2.5	8.67×10^{-3}	7.01×10^{-5}	6.06×10^{-9}	7.01×10^{-5}
64	2.6	8.60×10^{-3}	6.97×10^{-5}	5.14×10^{-9}	6.97×10^{-5}
64	2.7	8.58×10^{-3}	6.92×10^{-5}	4.70×10^{-9}	6.92×10^{-5}
64	2.8	8.65×10^{-3}	6.87×10^{-5}	4.54×10^{-9}	6.87×10^{-5}
64	2.9	8.72×10^{-3}	6.82×10^{-5}	4.67×10^{-9}	6.82×10^{-5}
64	3.0	8.80×10^{-3}	6.78×10^{-5}	5.02×10^{-9}	6.78×10^{-5}

Concluded

Table B.6: Parameter scan results for numerical heating tests.

		λ_D		
PPC	$r_0/\Delta x$	Δx	$\Delta x/2$	$\Delta x/4$
16	0.0	1.2291	1.2596	2.2231
16	0.1	1.2069	1.2516	2.1692
16	0.2	1.1769	1.1959	2.0251
16	0.3	1.1433	1.1526	1.8078
16	0.4	1.1075	1.1067	1.6103
16	0.5	1.0814	1.0730	1.4038
16	0.6	1.0498	1.0439	1.2609
16	0.7	1.0348	1.0249	1.1521
16	0.8	1.0179	1.0120	1.0779
16	0.9	1.0158	1.0053	1.0413
16	1.0	1.0070	1.0017	1.0204
16	1.1	1.0073	1.0017	1.0165
16	1.2	1.0040	1.0018	1.0108
16	1.3	1.0020	1.0027	1.0117
16	1.4	1.0024	1.0042	1.0149
16	1.5	1.0008	1.0034	1.0199
32	0.0	1.0818	1.1323	1.6853
32	0.1	1.0820	1.1215	1.6411
32	0.2	1.0668	1.1032	1.5731
32	0.3	1.0580	1.0775	1.4551
32	0.4	1.0429	1.0538	1.3443
32	0.5	1.0328	1.0362	1.2234
32	0.6	1.0227	1.0188	1.1448
32	0.7	1.0144	1.0115	1.0836
32	0.8	1.0099	1.0062	1.0409
32	0.9	1.0064	1.0020	1.0203
32	1.0	1.0038	1.0008	1.0121
32	1.1	1.0033	1.0002	1.0070
32	1.2	1.0019	1.0015	1.0040
32	1.3	1.0014	1.0010	1.0049
32	1.4	1.0016	1.0007	1.0099
32	1.5	1.0006	1.0014	1.0129
64	0.0	1.0385	1.0695	1.3834
64	0.1	1.0373	1.0602	1.3565
64	0.2	1.0328	1.0528	1.2991
64	0.3	1.0259	1.0416	1.2396
64	0.4	1.0190	1.0260	1.1779
64	0.5	1.0154	1.0182	1.1193
64	0.6	1.0107	1.0099	1.0715
64	0.7	1.0074	1.0053	1.0432
64	0.8	1.0047	1.0030	1.0210
64	0.9	1.0039	1.0014	1.0108
64	1.0	1.0017	1.0009	1.0052
64	1.1	1.0014	1.0006	1.0037
64	1.2	1.0014	1.0001	1.0026
64	1.3	1.0003	1.0002	1.0028
64	1.4	1.0005	1.0006	1.0030
64	1.5	1.0003	1.0007	1.0044

Concluded

APPENDIX C
Performance of Higher-Order Particle Representation

Table C.1: PPC scaling using smooth particles with three-point quadrature for electrostatics.

Count	BDW	KNL	CL	TX2	P100	V100
2^{15}	12.638	16.036	6.152	10.885	4.850	2.794
2^{16}	24.448	28.034	12.193	17.286	7.964	4.947
2^{17}	48.246	53.697	23.961	30.372	14.435	8.805
2^{18}	95.483	99.916	47.721	58.545	27.224	16.751
2^{19}	190.407	201.696	97.496	119.896	52.936	32.009
2^{20}	380.364	397.226	194.597	232.573	102.376	62.578

Table C.2: PPC scaling using smooth particles with five-point quadrature for electrostatics.

Count	BDW	KNL	CL	TX2	P100	V100
2^{15}	31.577	36.147	14.923	19.788	11.602	5.845
2^{16}	62.798	60.926	29.954	35.489	19.436	11.583
2^{17}	125.676	121.851	60.794	76.922	37.624	22.427
2^{18}	249.527	240.099	121.394	152.292	70.580	42.751
2^{19}	497.373	473.640	241.924	297.179	139.138	82.638
2^{20}	993.782	931.088	484.818	586.873	265.918	160.581

Table C.3: PPC scaling using smooth particles with three-point quadrature for electromagnetics.

Count	BDW	KNL	CL	TX2	P100	V100
2^{15}	49.702	44.527	24.348	29.155	19.909	8.889
2^{16}	98.587	81.819	48.564	54.477	30.420	13.493
2^{17}	196.380	157.910	95.711	104.827	50.896	22.647
2^{18}	392.636	313.926	190.931	207.618	92.895	40.798
2^{19}	785.835	629.013	382.645	416.210	177.084	78.482
2^{20}	1572.320	1241.760	766.342	828.313	345.717	155.461

Table C.4: PPC scaling using smooth particles with five-point quadrature for electromagnetics.

Count	BDW	KNL	CL	TX2	P100	V100
2^{15}	221.610	177.498	108.720	117.608	67.360	30.449
2^{16}	443.003	350.737	216.546	231.420	115.007	52.423
2^{17}	885.496	698.394	429.134	464.816	208.516	94.777
2^{18}	1769.360	1401.660	851.952	925.406	398.444	178.080
2^{19}	3540.050	2796.960	1710.160	1844.310	783.678	349.510
2^{20}	7077.140	5524.040	3420.340	3686.580	1562.870	695.983

Table C.5: Smooth particles slowdown versus base code with three-point quadrature for electrostatics.

Count	BDW	KNL	CL	TX2	P100	V100
2^{15}	5.535	1.191	6.158	1.346	2.544	1.924
2^{16}	6.550	2.313	6.487	2.056	3.109	2.524
2^{17}	6.694	2.912	6.654	2.915	3.581	2.743
2^{18}	7.347	3.161	6.777	3.245	3.871	3.157
2^{19}	7.740	3.409	7.023	4.539	3.764	3.418
2^{20}	7.772	3.433	6.986	5.104	4.160	3.449

Table C.6: Smooth particles slowdown versus base code with five-point quadrature for electrostatics.

Count	BDW	KNL	CL	TX2	P100	V100
2^{15}	13.830	2.685	14.938	2.448	6.087	4.025
2^{16}	16.824	5.027	15.935	4.221	7.589	5.910
2^{17}	17.436	6.607	16.882	7.382	9.333	6.987
2^{18}	19.200	7.595	17.240	8.442	10.037	8.058
2^{19}	20.219	8.006	17.427	11.249	9.893	8.826
2^{20}	20.306	8.047	17.405	12.879	10.806	8.850

Table C.7: Smooth particles slowdown versus base code with three-point quadrature for electromagnetics.

Count	BDW	KNL	CL	TX2	P100	V100
2^{15}	19.771	6.286	23.317	5.553	6.284	5.054
2^{16}	22.605	9.686	24.369	8.765	8.668	5.896
2^{17}	22.914	16.726	25.085	12.832	11.280	7.881
2^{18}	23.832	18.651	25.470	14.468	14.124	10.328
2^{19}	24.212	21.302	25.684	18.498	16.483	12.757
2^{20}	24.267	21.233	25.412	20.075	18.164	15.188

Table C.8: Smooth particles slowdown versus base code with five-point quadrature for electromagnetics.

Count	BDW	KNL	CL	TX2	P100	V100
2^{15}	88.154	25.059	104.118	22.399	21.261	17.313
2^{16}	101.574	41.521	108.660	37.235	32.771	22.908
2^{17}	103.320	73.974	112.473	56.900	46.213	32.981
2^{18}	107.395	83.277	113.650	64.487	60.582	45.083
2^{19}	109.069	94.720	114.790	81.967	72.945	56.812
2^{20}	109.226	94.455	113.419	89.348	82.115	67.993

Table C.9: Broadwell particle update strong scaling data.

	Electrostatic			Electromagnetic		
Cores	**Base**	**Three-Point**	**Five-Point**	**Base**	**Three-Point**	**Five-Point**
1	1.000	1.000	1.000	1.000	1.000	1.000
2	1.943	1.982	1.980	1.976	1.978	1.986
4	3.371	3.544	3.592	3.585	3.588	3.574
8	5.346	6.056	6.100	6.122	6.166	6.161
14	7.917	9.827	9.865	10.075	10.174	10.220

Table C.10: Cascade Lake particle update strong scaling data.

	Electrostatic			Electromagnetic		
Cores	**Base**	**Three-Point**	**Five-Point**	**Base**	**Three-Point**	**Five-Point**
1	1.000	1.000	1.000	1.000	1.000	1.000
2	1.879	1.921	1.931	1.926	1.936	1.933
4	3.436	3.637	3.688	3.699	3.691	3.675
8	6.315	7.114	7.191	7.259	7.212	7.269
16	9.942	12.745	12.989	13.313	13.139	13.206
24	12.084	16.820	17.254	17.431	17.554	17.628

Table C.11: KNL particle update strong scaling data.

	Electrostatic			Electromagnetic		
Cores	**Base**	**Three-Point**	**Five-Point**	**Base**	**Three-Point**	**Five-Point**
1	1.000	1.000	1.000	1.000	1.000	1.000
2	1.875	1.945	1.949	1.951	1.954	1.950
4	3.346	3.695	3.708	3.743	3.760	3.744
8	5.862	7.214	7.309	7.431	7.481	7.447
16	9.007	13.279	13.594	14.269	14.404	14.418
32	8.770	16.615	15.927	19.665	21.328	21.474
64	15.865	42.944	45.060	49.582	53.995	55.986

Table C.12: ThunderX2 particle update strong scaling data.

	Electrostatic			Electromagnetic		
Cores	**Base**	**Three-Point**	**Five-Point**	**Base**	**Three-Point**	**Five-Point**
1	1.000	1.000	1.000	1.000	1.000	1.000
2	1.948	1.991	1.984	1.990	1.997	1.989
4	3.685	3.933	3.930	3.947	3.984	3.954
8	6.597	7.765	7.724	7.703	7.929	7.892
16	10.654	14.627	14.918	14.645	15.619	15.499
32	13.713	24.257	24.034	25.110	27.938	27.968

Table C.13: Error versus cost results for the base code.

Refine	PPC	E_x	E_y	Time
1	1	1.50×10^{-2}	1.80×10^{-2}	7.584
1	2	1.16×10^{-2}	1.36×10^{-2}	7.560
1	4	7.84×10^{-3}	9.47×10^{-3}	7.641
1	8	5.81×10^{-3}	7.17×10^{-3}	7.683
1	16	3.43×10^{-3}	5.11×10^{-3}	7.829
1	32	2.58×10^{-3}	4.21×10^{-3}	7.740
1	64	2.00×10^{-3}	3.56×10^{-3}	8.053
2	1	1.43×10^{-2}	1.74×10^{-2}	20.554
2	2	1.14×10^{-2}	1.30×10^{-2}	20.448
2	4	8.32×10^{-3}	9.46×10^{-3}	20.188
2	8	5.71×10^{-3}	6.46×10^{-3}	20.624
2	16	3.69×10^{-3}	4.34×10^{-3}	21.101
2	32	2.74×10^{-3}	3.27×10^{-3}	20.842
2	64	1.91×10^{-3}	2.31×10^{-3}	21.800
3	1	1.51×10^{-2}	1.76×10^{-2}	64.354
3	2	1.15×10^{-2}	1.33×10^{-2}	63.056
3	4	7.88×10^{-3}	9.07×10^{-3}	62.965
3	8	5.58×10^{-3}	6.46×10^{-3}	64.003
3	16	3.69×10^{-3}	4.52×10^{-3}	65.048
3	32	2.69×10^{-3}	3.28×10^{-3}	70.616
3	64	1.96×10^{-3}	2.30×10^{-3}	74.791
4	1	1.57×10^{-2}	1.84×10^{-2}	195.229
4	2	1.14×10^{-2}	1.35×10^{-2}	197.448
4	4	7.93×10^{-3}	9.24×10^{-3}	192.805
4	8	5.57×10^{-3}	6.55×10^{-3}	201.532
4	16	3.81×10^{-3}	4.39×10^{-3}	209.761
4	32	2.82×10^{-3}	3.29×10^{-3}	215.343
4	64	1.95×10^{-3}	2.26×10^{-3}	243.668

Concluded

Table C.14: Error versus cost results for three-point cubature..

Refine	PPC	E_x	E_y	Time
0	1	4.75×10^{-3}	1.36×10^{-2}	3.379
0	2	3.72×10^{-3}	1.27×10^{-2}	3.236
0	4	2.72×10^{-3}	1.21×10^{-2}	3.281
0	8	1.83×10^{-3}	1.17×10^{-2}	3.433
0	16	1.11×10^{-3}	1.16×10^{-2}	3.598
0	32	8.81×10^{-4}	1.19×10^{-2}	4.000
0	64	6.71×10^{-4}	1.19×10^{-2}	4.771
1	1	4.97×10^{-3}	7.98×10^{-3}	8.432
1	2	3.69×10^{-3}	5.97×10^{-3}	8.230
1	4	2.53×10^{-3}	4.60×10^{-3}	8.687
1	8	1.97×10^{-3}	3.88×10^{-3}	9.055
1	16	1.09×10^{-3}	3.24×10^{-3}	9.730
1	32	8.70×10^{-4}	2.97×10^{-3}	10.941
1	64	7.22×10^{-4}	2.76×10^{-3}	13.407
2	1	5.05×10^{-3}	7.62×10^{-3}	24.118
2	2	3.82×10^{-3}	5.78×10^{-3}	23.972
2	4	2.81×10^{-3}	3.99×10^{-3}	24.315
2	8	1.99×10^{-3}	2.93×10^{-3}	26.183

Continued on next page

Refine	PPC	E_x	E_y	Time
2	16	1.28×10^{-3}	2.00×10^{-3}	30.564
2	32	9.28×10^{-4}	1.54×10^{-3}	37.623
2	64	6.77×10^{-4}	1.24×10^{-3}	51.852
3	1	5.17×10^{-3}	7.88×10^{-3}	72.407
3	2	3.78×10^{-3}	5.55×10^{-3}	74.762
3	4	2.57×10^{-3}	3.87×10^{-3}	83.884
3	8	1.88×10^{-3}	2.87×10^{-3}	101.520
3	16	1.24×10^{-3}	2.01×10^{-3}	136.851
3	32	9.14×10^{-4}	1.46×10^{-3}	198.215
3	64	6.92×10^{-4}	1.02×10^{-3}	323.609
4	1	5.30×10^{-3}	8.08×10^{-3}	211.388
4	2	3.78×10^{-3}	5.75×10^{-3}	246.184
4	4	2.65×10^{-3}	3.95×10^{-3}	284.260
4	8	1.89×10^{-3}	2.81×10^{-3}	373.819
4	16	1.29×10^{-3}	1.91×10^{-3}	542.866
4	32	9.30×10^{-4}	1.43×10^{-3}	860.872
4	64	6.55×10^{-4}	1.02×10^{-3}	1496.260

Concluded

Table C.15: Error versus cost results for five-point cubature..

Refine	PPC	E_x	E_y	Time
0	1	1.97×10^{-3}	1.21×10^{-2}	3.917
0	2	1.63×10^{-3}	1.22×10^{-2}	3.813
0	4	1.14×10^{-3}	1.17×10^{-2}	4.247
0	8	7.67×10^{-4}	1.16×10^{-2}	4.656
0	16	4.33×10^{-4}	1.16×10^{-2}	4.961
0	32	3.19×10^{-4}	1.18×10^{-2}	7.030
0	64	2.22×10^{-4}	1.18×10^{-2}	9.872
1	1	2.15×10^{-3}	4.36×10^{-3}	9.390
1	2	1.54×10^{-3}	3.68×10^{-3}	9.449
1	4	9.78×10^{-4}	3.11×10^{-3}	10.237
1	8	8.12×10^{-4}	2.84×10^{-3}	11.372
1	16	4.51×10^{-4}	2.77×10^{-3}	15.108
1	32	3.53×10^{-4}	2.76×10^{-3}	21.204
1	64	2.84×10^{-4}	2.70×10^{-3}	32.972
2	1	2.04×10^{-3}	3.65×10^{-3}	29.753
2	2	1.61×10^{-3}	2.65×10^{-3}	30.807
2	4	1.17×10^{-3}	1.86×10^{-3}	33.313
2	8	8.03×10^{-4}	1.32×10^{-3}	42.806
2	16	4.86×10^{-4}	1.02×10^{-3}	62.851
2	32	3.54×10^{-4}	8.68×10^{-4}	95.305
2	64	2.60×10^{-4}	8.03×10^{-4}	159.957
3	1	2.11×10^{-3}	3.62×10^{-3}	94.600
3	2	1.60×10^{-3}	2.54×10^{-3}	108.346
3	4	1.02×10^{-3}	1.75×10^{-3}	149.248
3	8	7.58×10^{-4}	1.26×10^{-3}	230.604
3	16	4.71×10^{-4}	9.06×10^{-4}	388.879
3	32	3.56×10^{-4}	6.47×10^{-4}	673.549
3	64	2.72×10^{-4}	4.71×10^{-4}	1235.850
4	1	2.22×10^{-3}	3.73×10^{-3}	308.424
4	2	1.62×10^{-3}	2.73×10^{-3}	421.746
4	4	1.09×10^{-3}	1.83×10^{-3}	631.858

Continued on next page

Refine	PPC	E_x	E_y	Time
4	8	7.72×10^{-4}	1.30×10^{-3}	1012.030
4	16	5.21×10^{-4}	8.69×10^{-4}	1788.800
4	32	3.82×10^{-4}	6.36×10^{-4}	3255.650
4	64	2.65×10^{-4}	4.52×10^{-4}	6151.890

Concluded