MICHAEL GARBA

ADAPTIVE HETEROGENEOUS PARALLELISM FOR SEMI-EMPIRICAL LATTICE DYNAMICS IN COMPUTATIONAL MATERIALS SCIENCE

ADAPTIVE HETEROGENEOUS PARALLELISM FOR SEMI-EMPIRICAL LATTICE DYNAMICS IN COMPUTATIONAL MATERIALS SCIENCE

MICHAEL GARBA

A Thesis Submitted in Partial Fulfilment of the Requirements of the

Robert Gordon University

for the degree of

Doctor of Philosophy

April 2015

Michael Garba: *Adaptive Heterogeneous Parallelism for Semi-Empirical Lattice Dynamics in Computational Materials Science*

Dedicated to the memory of my mother, Celine.

1951 – 2013

ABSTRACT

With the variability in performance of the multitude of parallel environments available today, the conceptual overhead created by the need to anticipate runtime information to make design-time decisions has become overwhelming. Performance-critical applications and libraries carry implicit assumptions based on incidental metrics that are not portable to emerging computational platforms or even alternative contemporary architectures. Furthermore, the significance of runtime concerns such as makespan, energy efficiency and fault tolerance depends on the situational context.

This thesis presents a case study in the application of both Mattson's prescriptive pattern-oriented approach and the more principled structured parallelism formalism to the computational simulation of inelastic neutron scattering spectra on hybrid CPU/GPU platforms. The original ad hoc implementation as well as new pattern-based and structured implementations are evaluated for relative performance and scalability. Two new structural abstractions are introduced to facilitate adaptation by lazy optimisation and runtime feedback. A deferred-choice abstraction represents a unified space of alternative structural program variants, allowing static adaptation through model-specific exhaustive calibration with regards to the extrafunctional concerns of runtime, average instantaneous power and total energy usage. Instrumented queues serve as mechanism for structural composition and provide a representation of extrafunctional state that allows realisation of a market-based decentralised coordination heuristic for competitive resource allocation and the Lyapunov drift algorithm for cooperative scheduling.

PUBLICATIONS

Journal Publications

1. Garba, M. T., González–Vélez, H., and Roach, D. L. (2013). GPU Acceleration for Hermitian Eigensystems. In Transactions on Computational Collective Intelligence X (pp. 150-161). Springer Berlin Heidelberg.
2. Garba, M. T., González–Vélez, H. (2012). Asymptotic Peak Utilisation in Heterogeneous Parallel CPU/GPU Pipelines: A Decentralised Queue Monitoring Strategy. Parallel Processing Letters, 22(02).

Peer-Reviewed Conference Proceedings

1. Goli, M., Garba, M. T., & Gonzalez-Vélez, H. (2012). Streaming dynamic coarse-grained CPU/GPU workloads with heterogeneous pipelines in FastFlow. In High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on (pp. 445-452). IEEE.
2. Garba, M. T., González–Vélez, H. Towards Ad-Hoc GPU Acceleration of Parallel Eigensystem Computations. In ECMS 2011: 25th European Conference on Modelling and Simulation, Krakow, Poland, June 2011. ECMS.
3. Garba, M. T., González–Vélez, H., and Roach, D. L. Parallel Computational Modelling of Inelastic Neutron Scattering in Multi-node and Multi-core Architectures. In High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on, pages 509–514. IEEE.

There are no eternal facts,
as there are no absolute truths.

— Friedrich Nietzsche

ACKNOWLEDGMENTS

I would like to take this opportunity to express my immense gratitude to all those persons who have given their invaluable support and assistance.

In particular, I am profoundly indebted to my supervisory team, Prof. John McCall, Prof. Horacio Gonzalez-Velez and Dr. Daniel Roach, who have provided guidance, feedback, and expertise over the course of completing this thesis. This work aspires to be a synthesis of the unique perspectives they have shared.

The research for this thesis was financially supported by a postgraduate studentship from the IDEAS institute at the Robert Gordon University, Aberdeen and the EU FP7 Project *ParaPhrase*: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. Additional support has been provided by the UK central neutron scattering facility ISIS at the Rutherford Appleton Laboratories, the Partnership for Advanced Computing in Europe (PRACE) and NVIDIA Corporation.

I am also thankful for the support of my colleagues, friends and family.

Notwithstanding all of the above support for this project, any errors and/or omissions are solely my own.

NOMENCLATURE

APUs Accelerated Processing Units

CINS Coherent Inelastic Neutron Spectroscopy

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DFT Density Functional Theory

DSL Domain Specific Language

FIFO First-in First-out

FPGA Field Programmable Gate Array

GUI Graphical User Interface

HPC High Performance Computing

INS Inelastic Neutron Spectroscopy

MBC Market Based Control

MIC Many Integrated Core

MVC Model View Controller

polyCINS Polycrystralline Coherent Inelastic Neutron Spectroscopy

RSO Reciprocal Space Onion

SPMD Single Program Multiple Data

ToF Time of Flight

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

The multicore era that started with the sudden shift from higher CPU clock rates towards multiple integrated on-chip cores is now firmly established. Manycore accelerators, with at least an order of magnitude more computational cores, represent a leap in the sophistication of these devices. In theory, for certain classes of workloads, these general-purpose computational accelerators should be able to achieve performance comparable to a small CPU cluster at a fraction of the previous cost and energy requirements. In practice, however, heroic efforts are often necessary to achieve that level of efficiency.

Today, there is a proliferation of manycore accelerators with tens or hundreds of cores from different manufacturers and radical new designs on the horizon. Examples include different models of low to high-end GPUs from Nvidia and AMD/ATI, Accelerated Processing Units (APUs) that integrate onboard CPUs on a GPU, Intel's new Many Integrated Core (MIC) architecture termed Xeon Phi and upcoming general purpose field programmable gate arrays (FPGAs) from Altera. Each accelerator from any of several families carries unique performance characteristics, usage constraints and programming idioms that are critical to the performance and efficiency of an application.

The heterogeneous host platforms themselves may be drawn from a much wider range of possible hardware configurations with interconnection characteristics that are impossible to anticipate during development. The task of tuning and adapting an application for a limited set of execution platforms is itself a daunting challenge, highly divergent possibilities for code deployment in mobile, desktop, cloud and HPC are usually met with compromise or sheer resignation.

This situation is not without historical precedent. It is possible to draw parallels with the early days of the computing industry when manufacturers created proprietary and incompatible architectural configurations. Application software was frequently developed for a specific machine. In the best case, only minor modifications would be necessary when new models were released by the manufacturer. At the other extreme, major changes or a complete rewrite of the codebase would be necessary to account for new capabilities, innovations and restrictions on the hardware.

Thus, although reaching the limits of Moore's law has forced the adoption of new parallel architectures, and these modern incarnations are highly unlikely to be as symmetric, homogeneous and predictable as their parallel supercomputer predecessors, an opportunity now exists to rethink the way software is constructed, taking into account the insights and best practices from related domains. However, this may be done without the burden of legacy tools and established practices that, despite the

substantial investment they represent constituting strongly entrenched interests, have proven inadequate for new heterogeneous platforms.

The hardware crisis was mitigated by abstraction and the adoption of vendor-neutral standards and protocols, with operating systems taking over responsibility for managing low level hardware and exposing consistent and reliable interfaces to applications. Modern virtualisation technology and runtime virtual machines have arguably solved the portability problem.

Separation of concerns is the motivation behind attempts to develop parallel middleware that has the potential to bridge the growing divide created by the relentless advance of hardware sophistication and lagging capability of existing software. As an alternative to constructing novel parallel Domain Specific Languages (DSLs), middleware may take advantage of existing language infrastructure and expertise to enable incremental updates to existing applications while allowing new programs to take full advantage of a set of structural patterns that facilitate portability and efficiency across multiple heterogeneous accelerators and deployment environments.

As computers become ever more ubiquitous, extra-functional considerations which are secondary to the intended application are growing in significance. Maintaining performance or quality of service while maximising energy efficiency and fault tolerance have become engineering problems in their own right.

This thesis takes the position that part of the complexity inherent in parallel programming is created by the need to anticipate runtime information during development. With the multitude of parallel environments and variability in performance, this conceptual overhead has become overwhelming for the programmer and constraining for the application. It proposes that these choices should be deferred until as late as is practical in the program lifecycle in an approach that may be described as *lazy design*.

1.1 CONTRIBUTION

Too frequently, the outcomes of research into idealised theoretical systems and artificial problems are disconnected from the incidental complexities of actual applications. Their usefulness may further be limited by the institutional resistance that arises when they neglect to account for social, technical and economic considerations. Outside the research community, commenters have called the observed tendency towards failure of solutions that are, in principle, technically superior "*worse is better*." It arguably explains why historic attempts to introduce many parallel languages and architectures outside the niche of high performance computing research have been unsuccessful.

From the perspective of a real application in computational science, this thesis approaches the *research question:*

> Given existing and emerging computational accelerators, execution environments and increasingly sophisticated extrafunctional user concerns, how can application software take advantage of the capabilities of the heterogeneous resources available, subject to their unique constraints, without compromising performance?

It proposes the *hypothesis* that

> As performance relies on endogeneous and exogeneous factors that are impossible to anticipate completely during development, avoiding premature choices, such as program structure, based on incidental measurements allows the systematic exploration of a decision space to determine the optimal configuration *pro re nata*.

Therefore, the *objective* is

> to demonstrate that using simple and minimally intrusive abstractions can allow applications and frameworks to adapt to environmental variation, evolving extrafunctional user concerns and application-specific demands.

The outcomes of this project will be the development of

1. adaptation mechanisms for heterogeneous platforms that may form the basis of pattern-oriented parallel middleware, an algorithmic skeleton library or the runtime of a domain specific language.

2. high performance simulation codes for computational investigation of materials by inelastic neutron spectroscopy. The results would constitute a significant contribution to the ability of researchers within the domain to revisit well-understood materials for additional insights into their physical properties and approach new materials with an expanded analysis toolbox.

1.2 ORGANISATION

This rest of this thesis is organised as follows:

Chapter 2 is the background, providing a historical overview, survey of the literature and description of the application domain that places this work in context.

Chapter 3 presents a definition of a descriptive structured domain specific language that represents structured parallel programs as directed flow graphs connecting queues and informally describes the semantics and transformation rules. This is extended into a high-level structural adaptation framework that is based on the instrumented queue and deferred choice operator abstractions.

Chapter 4 is a case study in the implementation of pattern-based parallelism to high performance polyCINS modeling. It presents an initial pattern-based re-implementation of the SCATTER code that represents the composition of the Monte Carlo and dense linear algebra dwarfs. While it demonstrates that this principled approach is scalable across traditional multicore and multinode environments, subsequent chapters introduce adaptive functionality by the deployment of our framework.

Chapter 5 considers the application of the deferred choice abstraction for describing structural variants of a program. It presents an implementation of numerical linear

algebra routines central to the Scatter application targeting GPU architectures. Integrated into the an alternative structured implementation, it demonstrates static adaptation with regards to multiple runtime environments, application-specific demands and extrafunctional user concerns including runtime, total energy consumption and instantaneous power requirements.

Chapter 6 is an examination of dynamic runtime adaptation in the structured Scatter implementations and two implementations of an ancillary validation application for the linear algebra kernels. The instrumented queue abstraction allow description of structural application variants and incorporation of heuristic coordination mechanisms for centralised coordination with a Lyapunov drift algorithm and decentralised control with a market-based framework.

Chapter 7 restates the conclusions and outlines some possibilities for future work.

2

BACKGROUND

In a widely influential paper [13], Asanovic et al identify the inadequacy of software as the most significant obstacle to connecting user applications with emerging parallel architectures. They propose a research agenda that aims to create programming frameworks that ease the development of portable, efficient and correct programs that are able to scale to the increasing number of cores available today and in the future.

These frameworks will provide tools such as compilers, libraries, code generators and runtime systems that allow:

1. productivity through the provision of reusable primitives and composable patterns that separate coordination from computation and allow domain experts and other programmers to build applications without the added complexity introduced by the low-level management of parallelism and concurrency.
2. efficiency through
    a) schedulers that take advantage of the expertise of parallel computing specialists and awareness of the patterns being deployed
    b) autotuners that choose optimal parametric configurations for algorithms
3. portability across parallel architectures such as multicore systems, FPGAs and manycore accelerators like the Xeon Phi
4. scalability across execution environments to include mobile, cloud environments, HPC and desktop
5. satisfaction of runtime objectives that include
    a) traditional objectives e.g. performance and makespan
    b) nontraditional objectives e.g. energy efficiency and fault tolerance.

2.1 PATTERNS

Christopher Alexander's work, *A Pattern Language* [5], introduced the concept of patterns as named elements of a language for describing solutions to design problems in Architecture and Urban Planning. Similar to actual languages, there are associated rules that govern the hierarchical composition of patterns into larger designs. While patterns simplify complex design problems and represent collective expertise accumulated over time, they serve the dual purpose of allowing concise communication between practitioners in a domain.

This systematic approach was brought into software engineering by Gamma et al [55] in the context of object-oriented design as the influential Design Patterns that

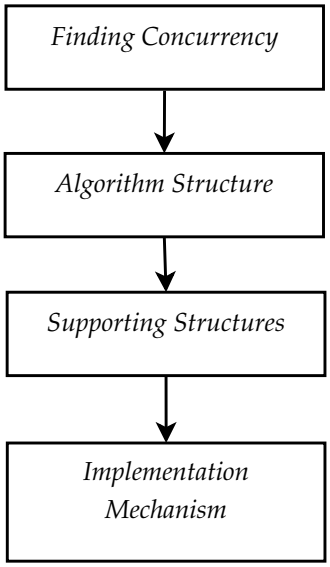


Figure 2.1: Mattson's Parallel Pattern hierarchy for the development of parallel programs.[89]

document names, intentions, motivations, applicability, collaboration and constraints as well as the larger architectural forms of entire software systems [51]. As an example, the Model-View-Controller (MVC) pattern has been particularly successful in the development of graphical user interface (GUI) applications and libraries and further forms the basis of high-productivity web development frameworks such as Ruby on Rails [58].

However, Norvig [96], using the functional language Lisp as a counter-example, has argued that some of these patterns may only be necessary when the implementation languages do not provide sufficiently powerful abstractions. Recent updates to the C++ and Java languages have adopted features previously found in functional languages such as lambda expressions and closures [134, 72, 78].

Influenced by Gamma et al, Mattson et al [89, 88] propose a pattern language that is applicable to a broad range of parallel programming problems and is analogous to the design patterns that have found widespread adoption in the methodology of object oriented design. In their view, the progression towards a parallel implementation of a program can be decomposed into the distinct stages outlined in Figure 2.1 on page 6 and Table 2.1 on page 7. Mattson's process draws heavily on experience and the selection between overlapping alternative patterns and interpretation of their definitions to user judgement.

Edsger Dijkstra's paper "Go To Statement Considered Harmful" [38], a reaction to the complexity of low-level code that had the tendency to become unmanageable, started the practice of structured programming [35]. The Böhm-Jacopini theorem [19], the theoretical basis, states that arbitrary programs may be composed of simple control structures that allow:

1. Sequential execution of statements
2. Conditional execution

Stage	Description
Finding Concurrency	An attempt is made to identify potential concurrency within the application at as many levels as possible. The results of this phase should include a task decomposition with potential for concurrent execution, the associated data required by each task and an understanding of the dependencies that may exist between these tasks.
Algorithm Structure	This follows from the previous phase. The objective is to select a major algorithmic organising principle that will form the basis of the task decomposition previously identified. Constraints imposed by the target platform are taken into account, alongside the sometimes conflicting objectives of efficiency, simplicity, portability and scalability. The result is a parallel algorithm that specifies how the cooperating tasks solve the problem.
Supporting Structure	An intermediate stage between design and implementation, these "supporting structures" describe the manner in which the algorithm will find expression in a programming environment. Typical program structuring patterns include the SPMD (Single Program Multiple Data), Master-Worker, Loop Parallelism and Fork-Join patterns. These patterns are not necessarily exclusive and frequently overlap.
Implementation Mechanisms	Here the design is mapped onto an existing implementation framework that provides the management of the execution environment and processing elements, synchronisation and communication that is required by the parallel program. The program is expressed in terms of the low-level operations in an implementation language.

Table 2.1: Mattson's Parallel Pattern hierarchy for the development of parallel programs.

3. Iteration

Thus well-formed structured programs are characterised by a single input and a single output from arbitrary procedures or functions. Although, given the widespread adoption of object oriented and functional programming abstractions, goto statements persist in modern programming languages and strict adherence to structured programming is no longer prevalent. However, it is widely accepted that their use should be avoided.

It may be argued that structured parallelism is a similar reaction to the complexity of parallel programs where issues relating to concurrency, synchronisation and shared state can rapidly lead to unmanageable complexity. Structured parallelism advocates the construction of programs from a restricted set of patterns with specified compositional semantics. This approach was pioneered by Cole [32] using functional notation to define algorithmic skeletons as higher order functions that capture the structure of classes of parallel programs.

Asanovic et al [12] present thirteen "dwarfs" of high performance computing that represent commonly occuring patterns of computation and communication covering a broad class of applications:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce or Monte Carlo
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch & Bound
12. Graphical Models
13. Finite State Machine

These dwarfs represent common patterns of computation and coordination that recur across the high performance computing application domain. The Monte Carlo dwarf in particular, characterised by the absence of dependencies between tasks, allows scalable parallel implementations to be easily realised in principle. However, in practice, large scale, potentially geographically distributed and dynamic environments with

heterogeneous computational resources and interconnection characteristics raise problems that include data partitioning, failure handling, efficient resource utilisation, management of large datasets and communication to achieve scalability. These problems are addressed by frameworks such as CONDOR [85] and BOINC [6] for combining dedicated resources and opportunistic execution on distributed volunteer machines. More recently, MapReduce [37, 135] has been deployed in large scale data analysis for web, scientific workloads [120, 130, 66] and big data applications [27, 2]. Similarly, Spark [138] has been developed for iterative processing such as may arise in machine learning applications.

These frameworks are intended to orchestrate coarse grained computation over distributed resources that frequently involve nested compositions of other patterns within the Monte Carlo structural form. The mechanism of this composition is usually a non-relational distributed data store that is characterised by high availability and designed to be resilient to node failures and network partitions [139, 122, 114]. Thus, these frameworks appear to have been influenced [137] by the tuple space model of distributed computation proposed by Gelernter for the coordination language Linda [59, 61, 60]. While the language extensions proposed in Linda did not gain widespread acceptance, the tuple space concept persists in the architecture of large distributed applications [95, 44, 16] where temporal decoupling and the avoidance of rendezvous greatly facilitates the composability of systems.

In contrast, the fine-grained counterparts to these frameworks, such as StreamIt [121], Skandium [83], Fastflow [4] and Intel Threading Building Blocks [104], that are optimised for multicore shared-memory parallelism involving threads, expose the primitive functions to be composed by a skeleton into an application directly to the user as overridable methods. From this perspective, composition occurs via queues that are relegated to the status of mere implementation artifacts that are typically hidden and understandably lightweight. Therefore, when instrumented for profiling, it is not uncommon for these frameworks to ignore queue behaviour in lieu of user functions.

2.2 PERFORMANCE

Structured parallelism, as implemented in algorithmic skeleton frameworks, enables a separation of the two orthogonal concerns of computation and coordination [13]. At the *productivity layer*, middleware should expose high-level, and possibly domain-specific, primitives for programmers to specify computation and structure. The lower-level *performance layer* should provide the necessary distributed coordination, messaging, scheduling and error recovery. The performance layer may be heavily optimised and tuned by parallel domain experts. Although complexity is introduced in the attempt to reconcile the algorithmic skeleton abstraction with the imperative or object-oriented paradigms of their implementation languages, the potential to isolate the user from the complexity of managing concurrent execution and shared state may justify this conceptual overhead. Therefore, the existence of efficiency-oriented heuristic

coordination runtimes would constitute a strong argument for the adoption of these frameworks. However, the specific methods by which this may be achieved remains an open problem.

Performance requirements are evolving as platforms mature in execution environments that include clusters, grids, cloud resources, volunteer computing and mobile devices. Alternative performance metrics for high performance computing applications are now being considered for their effect on operational costs that include measures of FLOPs-per-watt for energy efficiency, carbon emissions and heat dissipation requirements [48, 76]. Large scale frameworks such as MapReduce implementations and BOINC have experienced such extensive deployment on significant problems that performance criteria other than makespan have become relevant [73].

Autotuners, deployed extensively for linear algebra libraries, explore the search space of parametric variations for algorithms within the library, converging on value combinations that optimise for cache characteristics and other attributes of the specific host architecture at install-time. The importance of autotuners has risen with the emergence of GPU architectures where they provide isolation from the low-level hardware details of changing hardware models [84, 132].

2.3 GENERAL PURPOSE COMPUTING ON GPUS

The Compute Unified Device Architecture (CUDA) is NVIDIA's proprietary platform for GPU computing. As represented in the OpenCL standard, other GPUs conform to this general architecture. CUDA allows the execution of *kernels*, written in CUDA C, on the GPU device. A kernel executes as a configurable grid of independent thread blocks that may contain up to 1024 threads in second generation CUDA devices. A *Single Instruction Multiple Thread (SIMT)* abstraction, where threads within a block execute identical instructions and may operate on different memory locations, allows fine-grained data parallelism within thread blocks and task parallelism with independent execution of multiple blocks at the kernel level [94]. Thread blocks are divided into *warps* of 32 threads in second generation CUDA devices. For a given block, only one of these warps is scheduled to execute on the actual hardware at any time. Therefore, GPUs do not fit neatly into Flynns taxonomy [49].

Although GPU accelerators are well suited to exploiting fine-grained parallelism, they require complete translation of code to a new language using low-level constructs that are closely tied to the hardware. GPU memory is hierarchically organised and independent from host memory. Global memory, high-latency and high-bandwidth DRAM, is the primary memory available on the device and is accessible by all executing kernels as well as for host to GPU data transfer. Limited high-speed shared memory, essentially a user-managed cache, exists locally on each streaming multiprocessor to allow the explicit avoidance of expensive off-chip global memory accesses. Also present are register, texture and constant memories with various performance characteristics. The low-bandwith and latency of data transfer between host memory,

global memory and thread register memory on the GPU constitute the predominant restriction on achievable performance. For optimising bandwidth usage within the memory hierarchy, a critical performance consideration is that high-cost global memory operations can be performed simultaneously or *coalesced* for a thread warp if certain access constraints are satisfied. In practice, significant efforts are usually dedicated to maximising the *compute to global memory access (CGMA) ratio* [77] by what is, in essence, a hit-or-miss attempt to balance between conflicting trade-offs. Furthermore, for efficient execution on the device, it is necessary to avoid complex control flow and conditional branching as thread divergence incurs a significant performance penalty [97, 77].

In general, the efficient *computation* problem makes it more practical to identify smaller kernels within programs that constitute bottlenecks and obtain speed-ups by relocation to the GPU. Ideally, these kernels may be identified and implemented to facilitate reuse as building blocks in other programs. High performance implementations may be further optimised to target multiple parallel architectures with aggressive optimisation such as through the use of autotuners [132]. This approach has been applied to linear algebra computations that are fundamental to numerical analysis and computational science applications [123, 84]. Different authors have suggested various approaches to optimising memory usage for scientific applications in GPU architectures, such as the use of cache analysis techniques to improve tiling algorithms [64], the deployment of low-level compiler annotations within CUDA source files to steer traversal of the memory hierarchy [126], ad-hoc data structures [87], and the automatic translation of OpenMP structures into CUDA primitives [82].

Accounting for memory transfer overheads becomes a critical consideration when optimised kernels are integrated with existing parallel applications. Therefore, even when efficient kernels exist, a related *coordination* problem remains in scheduling kernel execution on the resources available in a heterogeneous parallel environment with potentially multiple GPUs. The StarPU scheduling infrastructure [14], as deployed in Magma [124], provides a unified machine abstraction that allows the implementation of function variants for different architectures while transparently handling data transfers, caching and dependencies for different environments. Extensive efforts are also geared towards the use of platform-agnostic GPU frameworks which can deal with standard unified language deployments such as CUDA and OpenCL [42, 68]. However, approaches such as this make limited use of structural information that would otherwise be available to a pattern-based or algorithmic skeleton framework.

With growth in the use of environments such as mobile and cloud computing, the definition of what constitutes a performance measure has broadened to include runtime concerns such as energy efficiency and economic cost. Inefficiencies that may be acceptable over shorter program runs or smaller data sets may scale up to represent a significant performance compromise for larger applications. These scheduling decisions are further complicated by the range of possible variation in performance and memory hierarchy characteristics between different applications, environments and

GPU architectures. Broadly stated, these aspects of the coordination problem are representative of variation in the execution environment, application-specific demands, and runtime concerns of the user. It may be argued that these separate but related problems of *computation* and *coordination* are the reason why the full potential of these accelerators is yet to be attained outside their original application domain in real-time 3D graphics and gaming.

2.4 APPLICATION: COMPUTATIONAL NEUTRON SCATTERING

2.4.1 *Inelastic Neutron Spectroscopy*

In condensed matter research, Inelastic Neutron Spectroscopy (INS) is an experimental technique that is widely used to investigate the vibrational characteristics of materials. There has been limited application of coherent inelastic neutron spectroscopy (CINS) to polycrystalline materials given the complexity of the spectra generated by the superimposition of scattering intensities over all orientations of the crystalline material and the tendency of this method to lose relevant information that is available from the direct measurement of dispersion curves using a Triple Axis Spectrometer. Therefore INS experiments have traditionally been restricted to either incoherent scattering from polycrystals or coherent scattering from single crystals.

Nevertheless, a broad class of important materials, particularly nanomaterials, are only obtainable in polycrystalline form. The development of new methods of interpreting the coherent scattering data from such samples is therefore an important research problem. The complexity and significant resource demands that arise from the application of computational modelling techniques to this problem require new approaches and the support of advanced computational infrastructure.

The purpose of an INS experiment is to determine the scattering function $S(Q,\omega)$, which provides information about the relative position and motion of each atom in a target sample. As originally derived by Van Hove [127], this scattering function can be expressed in terms of the respective coherent and incoherent scattering functions. These scattering functions, as presented in Eqns. (2.1) and (2.2), depend only on the interactions between the nuclei and define the corresponding cross sections [116]. The coherent component depends on the average value of the scattering amplitude and contains all the information about the relative positions and motions of every nuclear pair. The incoherent scattering contribution depends only on the motions of each atom taken in isolation.

In Eqns. (2.1) and (2.2), $S_{coh}(Q,\omega)$ and $S_{inc}(Q,\omega)$ are respectively the coherent and incoherent scattering functions in a system of d atoms, for phonon mode s, reciprocal lattice vector τ, scattering length b_d, atomic mass M_d, Debye-Waller factor W_d, momentum transfer vector $\mathbf{Q}$, atomic position r_d, polarisation vector $\mathbf{e}_{ds}$, frequency ω, and phonon wavevector $\mathbf{q}$ with neutron energy gain/loss $\left\langle n_s + \frac{1}{2} \mp \frac{1}{2} \right\rangle \delta(\omega \pm \omega_s)$. N represents the number of atoms in the unit cell in the (non-Bravais) system.

$$
\begin{aligned}
S_{coh}(Q,\omega) &= \frac{1}{2N}\sum_{s}\sum_{\tau}\frac{1}{\omega_s}\left|\sum_{d}\frac{\bar{b}_d}{\sqrt{M_d}}\exp(-W_d)\exp(i\mathbf{Q}\cdot\mathbf{r}_d)(\mathbf{Q}\cdot\mathbf{e}_{ds})\right|^2 \\
&\quad\times\left\langle n_s+\frac{1}{2}\mp\frac{1}{2}\right\rangle\delta(\omega\pm\omega_s)\delta(\mathbf{Q}\mp\mathbf{q}-\tau) \qquad (2.1)\\
S_{inc}(Q,\omega) &= \sum_{d}\left\{\overline{b_d^2}-(\overline{b_d})^2\right\}\frac{1}{2M_d}\exp(-2W_d) \\
&\quad\times\sum_{s}\frac{(\mathbf{Q}\cdot\mathbf{e}_{ds})}{\omega_s}\left\langle n_s+\frac{1}{2}\mp\frac{1}{2}\right\rangle\times\delta(\omega\pm\omega_s) \qquad (2.2)
\end{aligned}
$$

The energy transfer of inelastic scattering results from the neutron energy gain or loss on interaction with the lattice. For a given momentum transfer vector $\mathbf{Q}$ representing the momentum change between incident and scattered wave vectors, and a vibrational frequency of the quantised lattice vibration (or phonon) created or annihilated by the scattering event, the frequency of the phonon is directly related to the modulus of the energy transfer between the target material and the scattered neutron, as determined by momentum and energy conservation and the principle of detailed balance [116].

2.4.2 *Neutron Scattering in GULP*

The General Utility Lattice Program (GULP) is a generalised symmetry-adapted lattice dynamics and simulation environment for the study of solid materials that provides routines for the modelling, prediction and interpretation of experimental data in the study of atomic, molecular, and bulk crystalline structures [53]. GULP is intended to solve a range of problems in molecular modelling and experimental data interpretation, with routines covering potential applications that range from simulation to model fitting. Symmetry is exploited within GULP to minimise redundant computation and provide a performance advantage over existing software in the same problem domain [53].

As a research tool in the physical sciences with interdisciplinary applications in chemistry, physics, and material science, GULP is available as an open source software package for non-commercial use and as part of the Materials Studio from Accelrys. Execution options, required and optional simulation parameters, general program options, and structural information are specified in the input deck. GULP outputs results to file or standard output alongside intermediate files that may contain data relevant to their interpretation. The time and space complexities of these GULP routines are directly related to the parametric and structural characteristics of the solid under investigation.

SCATTER, a new routine, makes extensive use of existing functionality to bring coherent and incoherent inelastic neutron scattering (INS) capabilities for lattice mod-

els to GULP [105, 106]. Until recently, INS was considered impractical on account of its significant computational requirements and the high cost and general unavailability of neutron scattering experimental facilities and instrumentation. However, INS modelling is experiencing increasing popularity for structural determination in solids among the materials science community, as a result of its suitability to problems such as those that occur in the study of nano-materials. A growing need has been created for tools that aid in the effective interpretation of the complex data generated [105]. Nonetheless, the implementation of efficient parallel INS solutions remains an open problem.

SCATTER allows the comparison and refinement of theoretical models against experimentally obtained results on the accuracy of which space sampling resolution has direct bearing. However, determinations of the scattering function for all values of a large set of magnitudes and directions create a significant computational load, frequently requiring days to weeks of execution time. Different authors have employed ab-initio approaches with GULP to produce several computational models [70, 125]. The analogously computationally demanding nature of molecular dynamics [103] is believed to be of particular relevance to an implementation of SCATTER.

Other libraries for INS modelling include PHONON [100] and A-CLIMAX [26]. However, these packages focus on modelling INS datasets from *ab-initio* and density functional theory techniques, and lack semi-empirical modelling capabilities, which are of core interest in the study of a wide range of technological nanocarbons, hydrogen storage materials and carbon composite materials.

2.4.3 *The* SCATTER *Code*

Materials researchers already employ neutron scattering simulations as a means of validating and refining their models [70]. Nevertheless, limited research has been devoted to the systematic application of computational solutions for the modelling of polycrystalline materials in current simulation packages (such as A-CLIMAX [26, 102] or PHONON [101]) that require model output from *ab initio* software, such as CASTEP [113] or VASP [80]. The SCATTER code allows the generation of poly-CINS modelling data using semi-empirical potential models (as well as output from the DFT codes mentioned) via the General Utility Lattice Program (GULP) software package [53, 54], a widely used lattice dynamics and simulation package in the materials science community.

The methodology behind the application of this software to systems with small unit cells (such as aluminium [108] or graphite [107]) has been presented elsewhere. The present work has been motivated by the need to allow the application of SCATTER to the analysis of polycrystalline samples with large unit cells, as well as to enable future real-time instrumentation applications, where simulations may be conducted alongside an inelastic neutron scattering experiment and theoretical models refined in real time as the data becomes available. Typical experimental run-times for the

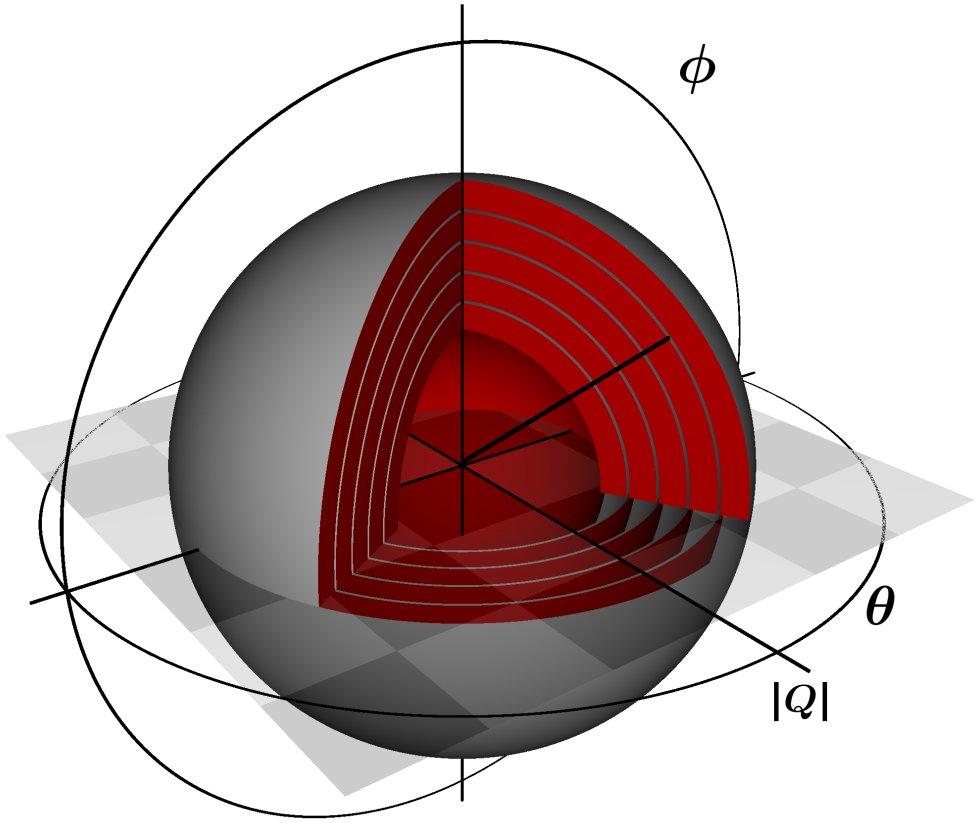


Figure 2.2: Reciprocal Space Onion Sampling in SCATTER. Concentric shells are traced in reciprocal space as the momentum transfer vector **Q** varies in magnitude and orientation in a spherical polar coordinate system.

collection of poly-CINS data even on high-flux instruments can exceed 24 hours, as excellent statistics are required to resolve one-phonon features in experimental data. New instruments and beam-lines are under development that will provide unprecedented neutron fluxes, allowing orders of magnitude more neutrons on sample in a given time period than are currently available, even at high flux beam-lines such as those found at the Spallation Neutron Source, ORNL, USA. For this reason alone, an investment in effective performance is critical to ensure that researchers using these methods are well positioned to take advantage of the next generation of distributed, GPU architectures entering the market over the next decade.

As described in the work of Roach et al [106, 108], SCATTER samples reciprocal space and models the one-phonon scattering of neutrons incident on single crystal and polycrystalline samples, predicting the coherent and incoherent scattering intensities of Equations 2.1 and 2.2. Here the phonons are determined from the analytic second derivatives of a force field model using the program GULP [105, 106].

The coherent component of the scattering intensity in Equation 2.1 takes into account cross-correlation of pairwise interactions of the nuclei in the system and describes inelastic interference effects, which provide information on the positions and vibrational modes of planes of atoms. Equation 2.2 represents the self-correlation, incoherent component of the scattering intensity, arising from the vibrational contributions of individual atoms considered in isolation, without this interference term. SCATTER is capable of determining both cross-correlation and self-correlation components, however the primary application of this method is for models of coherent scattering. Carbon is particularly notable in this regard as it has an entirely coherent cross-section.

For simulations of single crystal experimental configurations, it is sufficient to perform these calculations along fixed directions in reciprocal space. However, general application to the broad range of materials that are usually available only in polycrystalline powder form requires calculation over the full range of magnitudes and spatial orientations of the momentum transfer vector $\mathbf{Q}$ in three dimensions.

SCATTER implements several space sampling techniques that determine the closest corresponding lattice vectors $\mathbf{q}$, and the Brillouin zones for each $\mathbf{Q}$. The Reciprocal Space Onion (RSO) sampling method, illustrated in Figure 2.2 on page 15, takes values of $\mathbf{Q}$ as it is rotated about a series of concentric spheres of varying magnitude $|\mathbf{Q}|$ at angles θ and ϕ in a spherical polar coordinate system [106]. RSO sampling traces concentric shells that correspond to the trajectory of modern multi-chopper time-of-flight (ToF) neutron spectrometers in multi-angle configurations to facilitate the direct comparison of these theoretical models with experimental data.

SCATTER performs a weighted sum over the calculated scattering contributions for each sampled q-point to generate the polycrystalline-averaged scattering intensities of equation (2.3) [90] where ω is the phonon frequency:

$$S(Q,\omega)_{powder} = \frac{1}{4\pi}\int S(\mathbf{Q},\omega)\mathrm{d}\mathbf{Q} \tag{2.3}$$

In practice, the quality of results obtained is a function of the density of spatial sampling. For higher values of $|\mathbf{Q}|$, where sampling is increasingly sparse, artifacts in the final output may begin to appear in the form of additional texture in the intensity plots. For a high-resolution RSO grid, the generation and diagonalisation of large dynamical matrices for each determination of Equation 2.1 over the full range of values for $\mathbf{Q}$ is achieved at significant computational expense. Complete solution of the associated Hermitian eigensystems is a dominant aspect of this process and a trade-off is typically necessary between the desired model resolution and the actual execution time.

2.4.4 *Eigenproblems in Lattice Dynamics*

Neutrons interact with the collective vibrational excitations or *phonons* of the constituent atoms of a crystalline lattice. Given an interatomic potential model, a *dynamical matrix* may be computed whose eigenvectors are the polarisation vectors of the reciprocal space atoms and eigenvalues are the frequencies corresponding to the phonon modes of the lattice [41]. The interatomic force constants and potentials determine the magnitude and phase characteristics of the associated phonons.

For a lattice with a unit cell of n atoms, the dynamical matrix is a sparse $3n \times 3n$ complex symmetric or Hermitian matrix with $3n$ phonon modes. Although, a linear relationship exists between the number of atoms in the unit cell of a material and the size of the associated dynamical matrices, the computational complexity of solving these eigensystems is of order $O(n^3)$ [15]. However, lattice symmetries may result in degenerate modes that may be exploited to simplify the problem. There is a strong diagonal dominance in the dynamical matrix that follows from the nature of interatomic bonds in the lattice.

The GULP program generates the dynamical matrix for a broad class of potential models [53, 54]. SCATTER, as part of GULP, makes extensive use of the EISPACK [39] and LAPACK [7] libraries for linear algebra. Specifically, phonon modes are calculated with the support of EISPACK.

Dense numerical linear algebra is one of the thirteen dwarfs identified in the Berkeley technical report. They are the fundamental algorithms behind a large class of scientific computing problems and their importance in high performance computing is evident as they form the basis of the Linpack benchmarks used to rank the top supercomputers. Their inherent data parallelism make these problems well suited to GPU accelerators which were originally designed to solve three dimensional computations, themselves matrix operations.

2.4.5 *Hermitian Eigensystems*

A *Hermitian* matrix H is a complex matrix that is invariant under the operation of conjugate transposition (Equation 2.4).

$$H = H^{\dagger} \tag{2.4}$$

Given a general square matrix A, a non-zero vector $\mathbf{v}$ is an eigenvector of the matrix A, if and only if there exists a corresponding non-zero scalar λ, or eigenvalue, that satisfies Equation 2.5. It may be demonstrated that all eigenvalues λ_i of a Hermitian matrix are real.

$$A\mathbf{v} = \lambda\mathbf{v} \tag{2.5}$$

Given the relationship of Equation 2.6 and an invertible matrix T , the matrices A and B are said to be *similar* and they share several important properties that include identical eigenvalues λ_i and closely related eigenvectors (Equation 2.7).

$$T^{-1}AT = B \tag{2.6}$$

$$E_A = TE_B \tag{2.7}$$

Equation 2.6 describes similarity transformations are the basis of several matrix algorithms. When possible, a reduction of a given matrix to a similar matrix may allow the use of algorithms that are more efficient or direct. The Schur decomposition of A is a representation of the matrix as the result of a similarity transformation (Equation 2.8) on a strictly upper-triangular matrix S with a unitary matrix Q.

$$A = QSQ^{-1} \tag{2.8}$$

A transformation such as Equation 2.8 preserves symmetry when Q is unitary i.e. $QQ^{\dagger} = Q^{\dagger}Q = I$. Thus, if A is Hermitian and S is strictly upper triangular then S must be a diagonal matrix. Since $AQ = QS$, the columns of Q must be eigenvectors of A and the diagonal entries of S are the corresponding eigenvalues. Therefore, the Hermitian eigensystem problem is equivalent to determining the Schur decomposition of the Hermitian matrix A.

To compute the Schur decomposition, the matrix is typically reduced to a similar Hessenberg matrix via a potentially infinite sequence of symmetry-preserving similarity transformations. Since an upper Hessenberg matrix has all entries below the first subdiagonal set to zero (Equation 2.9), another property which follows from symmetry is that the Hessenberg form of a Hermitian matrix is tridiagonal. In practice, a finite number of these elementary Householder or Givens reflections will cause off-diagonal

Routine	Description
htridi	Reduction of complex Hermitian matrix to real symmetric tridiagonal matrix via unitary similarity transformations.
tql2	Eigenvalues and eigenvectors of symmetric tridiagonal matrix by QL method.
htribk	Eigenvectors of complex Hermitian matrix by back-transformation of corresponding real symmetric tridiagonal matrix.

Table 2.2: Relevant Hermitian Eigensystem routines in EISPACK used by the ch driver

elements to effectively reach zero as they are reduced to less than machine roundoff error.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2(n-1)} & a_{2n} \\ 0 & a_{32} & a_{33} & \dots & a_{3(n-1)} & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ 0 & 0 & 0 & \dots & a_{n(n-1)} & a_{nn} \end{bmatrix} \tag{2.9}$$

The eigenvalues and eigenvectors of the tridiagonalised matrix are evaluated via the QR algorithm. While the eigenvalues are identical, the actual eigenvectors E_A of the original matrix are retrieved by back-transforming the computed eigenvectors.

2.4.6 *Eigensystem Solvers*

EISPACK, and its expanded successor LAPACK, provide dense linear algebra routines that have arguably become the *de facto* standard in mathematical and scientific computing applications. The accuracy and numerical stability of EISPACK and LAPACK have been established by numerous deployments over the past three decades [115, 40]. The original EISPACK library was developed as a Fortran port of a set of Algol routines for numerical computation set out in the Handbook for Automatic Computation [133]. Under the ch driver for double precision Hermitian matrices, EISPACK relies on the three subroutines outlined in Table 2.2. LAPACK provides multiple implementations of the same algorithms with variants for real and complex matrices with single or double precision floating point entries. In this application domain, the dynamical matrices are double precision to maximise accuracy. The MAGMA library is such an effort that now provides many hybrid multicore-CPU/GPU implementations of LAPACK routines [124] with substantial impact in the physical sciences [3]. For sparse eigensystems, ARPACK employs Arnoldi iteration and Krylov subspace methods.

2.5 RESEARCH GAP

Asanovic et al present an argument, drawn from experience, that powerful frameworks are not invented but mined from successful applications and state

> "The goal of research into parallel computing should be to find compelling applications that thirst for more computing than is currently available and absorb biennially increasing number of cores for the next decade or two."

The existence of efficiency-oriented heuristic coordination runtimes would constitute a strong argument for the adoption of parallel frameworks. However, the specific methods by which this may be achieved remains an open problem.

Although structural transformations have been considered, there is little investigation of systematic ways to represent structural variants of a program and exploration of this space is typically done on an ad hoc basis. Applications of autotuners have been primarily parametric, e.g. optimising block sizes and integrated into libraries.

Thus, in the context of the SCATTER application, a large-scale workload with important research implications in its own right, we propose to investigate mechanisms and abstractions that allow the representation of structural program information to facilitate offline and offline adaptation and runtime coordination in heterogeneous CPU/GPU environments. Here, our definition of an adaptive system is intended to capture the possible variations in the runtime environment between and during execution, changing application requirements and evolving extra-functional user concerns such as energy efficiency.

3

A STRUCTURAL ADAPTATION FRAMEWORK

This chapter presents a descriptive structured domain specific language that represents structured parallel programs as directed flow graphs connecting intermediate queues and informally describes the semantics and transformation rules that will be used subsequently. This is extended into a high-level structural adaptation framework where the instrumented queue and deferred choice operator abstractions are introduced to allow the representation of adaptation as the stochastic optimisation of functions that are representative of extra-functional runtime concerns.

3.1 PRELIMINARIES

3.1.1 *Generalised Queues*

A *generalised queue* C is any shared resource that supports the operations:

1. $\mathtt{put}(C, x)$ adds the item x to a collection,
2. $\mathtt{get}(C)$ returns an item x and removes it from the collection.

Therefore, lists, stacks, sets, multisets(bags) and FIFO queues may be considered as instances of generalised queues [112]. However, this broad definition also applies to resources that are not data structures e.g. NoSQL databases, files, and UNIX pipes.

3.1.2 *A Descriptive Domain Specific Language (DSL)*

Here, we present the notation that will subsequently be used to describe structural patterns and the transformation of programs that is based on the presentation in [83]. An important distinction from the implemention in the Skandium framework is that we consider a subset of the structural forms and their composition defined in terms of queues. As the intent is not to establish a formal theory, these concepts are neither rigorously defined nor their transformation rules strictly formalised. Instead, a descriptive convention is adopted to allow concise communication between domain experts [5]. Knowledge of the transformation rules associated with these patterns is an integral part of this domain expertise. Nevertheless, these constructs may be implemented as functions and primitives of a domain specific language[1] or algorithmic skeleton framework.

1 A simple python implementation of this DSL is used to generate the figures in this thesis via output to the Graphviz visualisation tool [43]. These functions may be directly adapted for constructing actual realisations of the corresponding flow graphs. The source code is available in appendix A on page 131

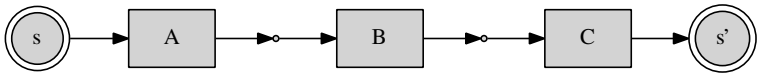


Figure 3.1: **SEQ**(A, B, C) represents sequential composition.

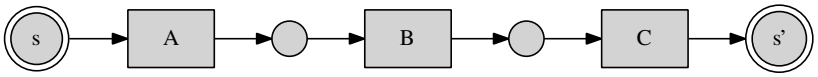


Figure 3.2: **PIPE**(A, B, C) represents concurrent composition into a pipeline with intermediate queues between stages

3.1.2.1 *Structural Forms*

Consider two stateless and referentially transparent functions or operations A and B that perform some computation. Then we define the composition of these two functions $A \cdot B$ as a function that applies the operation B to the output of A.

SEQ$(A_1, A_2, \ldots, A_n)$ is a higher order function that, given an input and output queue pair (s, s'), repeatedly applies the composed function $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ on data derived from the input queue and places the results in the output queue. Applied to some data x, **SEQ**$(A_1 A_2, \ldots A_n)$ is equivalent to the invocation $A_n(\ldots A_2(A_1(x)))$.

Although defined in terms of queues, **SEQ** represents the sequential composition of functions. We may now consider an analogous concurrent composition.

PIPE$(A_1, A_2, \ldots, A_n)$ is a higher order function that, given an input and output queue pair (s, s'), constructs a series of $n - 1$ intermediate queues, and concurrently executes the functions **SEQ**(A_i) such that, for $i > 1$, the input to **SEQ**(A_i) is the output of **SEQ**(A_{i-1}), the input of **SEQ**(A_1) is s and the output of **SEQ**(A_n) is s'.

As the application of **SEQ**(A) to the queue entries are independent operations, they may be executed concurrently. Thus **PIPE**$(A_1, \ldots, A_n)$ is the parallel analogue of **SEQ**$(A_1, \ldots, A_n)$ with the introduction of intermediate queues as the mechanism for concurrent function composition.

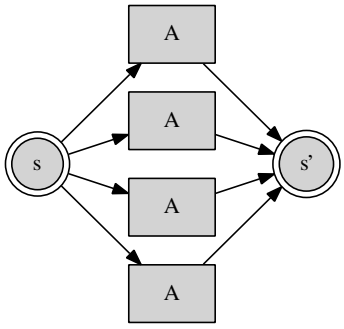


Figure 3.3: **FARM**(A) represents multiple instances of A executing concurrently

FARM(A) is a higher order function that, given an input and output queue pair (s, s′), concurrently executes **SEQ**(A) multiple times with the same input and output queue pair.

3.1.2.2 *Production Rules*

In this domain language, some valid program transformations that introduce parallelism may be represented by production or rewrite rules. We define a valid transformation as an expression that generates a directed graph where a sequence of functionally equivalent edges are visited in all directed walks from the source node s to sink nodes s'. The directed flow graphs generated by the application of these transformation rules represent the same computation. The symbol $\rightarrow$ represents a transformation.

$$\begin{aligned} A &\rightarrow \mathbf{SEQ}(A) \\ \mathbf{SEQ}(A) &\rightarrow \mathbf{FARM}(A) \\ \mathbf{SEQ}(A, B) &\rightarrow \mathbf{PIPE}(A, B) \end{aligned}$$

3.1.2.3 *Compositional Semantics and Identities*

As a critical performance optimisation, redundant queues are not permitted in this representation. Therefore, the following composed expressions generate identical directed graphs as the simplified expressions.

$$\mathbf{SEQ}(A, \mathbf{SEQ}(B, C)) = \mathbf{SEQ}(\mathbf{SEQ}(A, B), C) = \mathbf{SEQ}(A, B, C)$$

$$\mathbf{FARM}(\mathbf{FARM}(A)) = \mathbf{FARM}(A)$$

$$\mathbf{PIPE}(\mathbf{PIPE}(A, B), C) = \mathbf{PIPE}(A, \mathbf{PIPE}(B, C)) = \mathbf{PIPE}(A, B, C)$$

Therefore, the only singly nested expressions that may not be simplified are alternate compositions of the **FARM** and **PIPE** constructs.

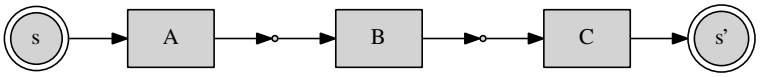
$$\mathbf{PIPE}(\mathbf{FARM}(A), \mathbf{FARM}(B))$$

$$\mathbf{FARM}(\mathbf{PIPE}(A, B))$$

These identities and composed forms are summarised in Figure 3.4 on page 24.

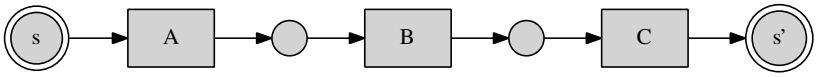
3.2 ADAPTATION FRAMEWORK

The three fundamental characteristics of general adaptive systems are variation, feedback and selection [71]. We define an adaptive program as possessing the ability to select between run-time alternatives in a manner that tends towards optimisation of some extra-functional performance measure, that provides feedback, subject to variations in the *performance measure* itself, *environmental factors* and *application-specific demands.*



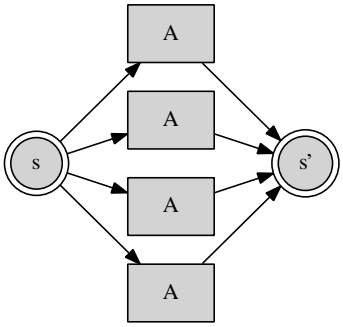
(a)

$$\mathbf{SEQ}(A, \mathbf{SEQ}(B, C)) = \mathbf{SEQ}(\mathbf{SEQ}(A, B), C) = \mathbf{SEQ}(A, B, C)$$



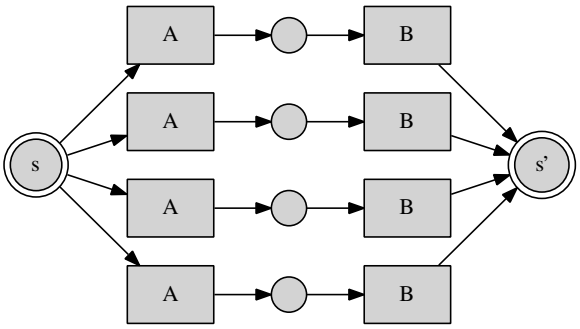
(b)

$$\mathbf{PIPE}(A, \mathbf{PIPE}(B, C)) = \mathbf{PIPE}(\mathbf{PIPE}(A, B), C) = \mathbf{PIPE}(A, B, C)$$



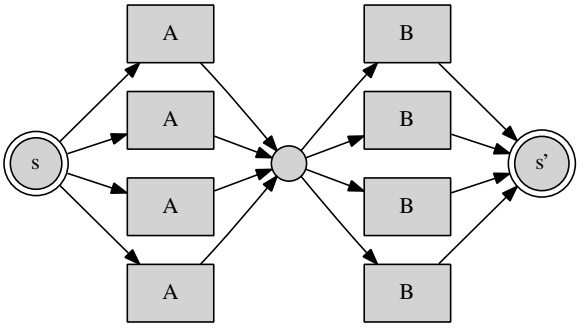
(c)

$$\mathbf{FARM}(\mathbf{FARM}(A)) = \mathbf{FARM}(A)$$



(d)

$$\mathbf{FARM}(\mathbf{PIPE}(A, B))$$



(e)

$$\mathbf{PIPE}(\mathbf{FARM}(A), \mathbf{FARM}(B))$$

Figure 3.4: Compositional semantics of nested patterns avoid redundant queues.

For example, changes in the performance measure may result from evolving user concerns. Environmental factors that affect the application may include hardware or software configuration and the effect of competing workloads. Application-specific demands may include changing load profiles that correspond to computational phases in the application.

In general, program dynamics, environmental variations and the effects of select actions on the performance measure are stochastic. Redefining the structured parallel constructs in terms of operations on queues allows the formulation of a Markov decision problem with the following elements:

3.2.1 *Queues as Extra-functional State*

The state space S is the set of all possible queue levels. S_t is the queue state vector at time t with components $s_1, s_2, ..., s_n$ where $s_i \in \mathbb{Z}^\star$, the set of non-negative integers.

The queues characterise the extra-functional state of the program. If we consider the state of the program queues to be a vector in an n-dimensional state space, or *queue space*, then, it is possible to represent the individual throughputs at each stage T_A, T_B and T_C as velocity vectors for which:

1. The n components of the state vector are the n queue levels
2. The velocity vector magnitudes are the magnitudes of the throughput T_i
3. The velocity vector orientations follow from the dataflow topology i.e. stages between queues (e.g. T_B in Figure 6.10 on page 102) have a negative component on the incoming queue and a positive component on the outgoing queue state vector components
4. The overall system trajectory is in the direction of the resultant velocity vector

To support this usage, it is necessary to extend the definition of a generalised queue to include operations that allow gathering of feedback and configuration of queuing behaviour. This definition is the basis of the dynamic adaptation mechanisms considered in Chapter 6.

An *instrumented queue*[2] Q, with functional attributes `hash`, `put_filter` and `get_filter`, is a generalised queue that supports the additional operations:

1. `put`(Q, x, m) adds the item x, with associated metadata m, to a collection if the condition `put_filter`(m) is `true`. If `put_filter`(m) is `false` then the behaviour is implementation-dependent and will either fail, block until it becomes `true` or a timeout occurs.
2. `get`(Q, m) removes and returns an item x from the collection for which the condition `get_filter`(m) is `true` for m, the metadata associated with x. If `get_filter`(m)

2 Unless otherwise stated, in the rest of this thesis usage of the term 'queue' will refer to instrumented queues.

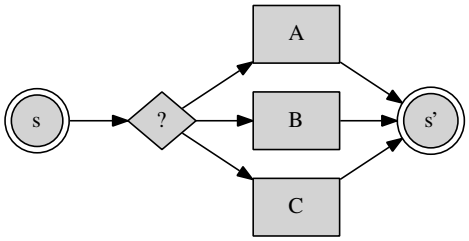


Figure 3.5: **ANY**(A, B, C) represents the lazy or deferred choice operator, selecting between functionally equivalent alternatives A,B or C by an indeterminate mechanism.

is $false$ then the behaviour is implementation-dependent and will either fail, block until it becomes true or a timeout occurs.

3. $reduce(Q, operator)$ returns the result of a reduction operation on the metadata associated with all items in the collection. $operator$ is a function of two arguments.

For the set M of all possible metadata associated with items in Q, the functional attributes are:

1. a function $hash : M \to \mathbb{R}$ determines the the queuing discipline. Items returned by get operations are ordered by key or priority value $hash(m)$ in descending order.
2. a function $put\_filter : M \to \{true, false\}$.
3. a function $get\_filter : M \to \{true, false\}$.

3.2.2 *Operations as Decision Space*

The set of all actions A, contains all transition paths, operations or functions between queue pairs. Structural information about the program is contained in the input and output queue pairs associated with any operation. This information determines the effect of any operation on the extra-functional program state.

The **ANY** *Operator*

Let us introduce **ANY** as the lazy or deferred choice operator between functionally equivalent alternatives,

ANY$(A_1, \ldots, A_n)$ is a higher order function that may be evaluated to return any one of A_i.

ANY is intended to explicitly capture otherwise implicit choices in the program design space that may also be made entirely statically (e.g. manually, at source level, compilation or install time), or dynamically at run-time by some generic mechanism.

ANY allows the structural specification of alternatives that may be made by the framework. Since these alternatives are functionally equivalent, we propose an approach where the choices are determined by an external optimiser. In this context:

1. **Static Adaptation** may be achieved by forcing evaluation of the **ANY** operator at any stage before program execution. The actual decisions may be based on a multitude of criteria including historical profiling information or direct user intervention. Whether the mechanism of this selection is via human agency or software tooling is irrelevant. The output of this process may be generated or runtime-specialised code, configuration files or program parameters. However, implementation of **ANY** as a compiler or language construct may be necessary.
2. **Dynamic Adaptation** may be achieved by re-evaluating the **ANY** operator during the execution of the program or deploying structures that implicitly make these choices at run-time. These decisions may be based on criteria including online performance metrics. The effect of this is a change in run-time behaviour.

3.2.3 *Performance Measure as Feedback*

We define a reward function $R(S_t, A_t, S_{t+1})$ as the expectation value of the performance metric when the action A_t causes a transition from S_t to S_{t+1}. The feedback needed from any operation is determined by a function f executed before and after that operation is performed where the cost is defined as the difference in computed values. The cost function may be considered as a negative reinforcement signal that captures the specific non-functional requirements that are the external preferences of a user or environment. Cost functions may also describe qualitative attributes that would otherwise be difficult to clearly describe and offer an opportunity to realise behavioural programs.

In this scheme, the queue vector also forms the basis of adaptive decision making with the objective of achieving a compromise between maximising some performance metric and minimising a measure of overall program queue state.

3.3 ADAPTIVE PARALLELISM

3.3.1 *Static Adaptation with* **ANY**

Static adaptation is equivalent to selection between runtime alternatives or *variants* before program execution. The **ANY** construct allows the description of these structural variants of the program subject to the only restriction being their sequential equivalence. In many applications, a significant part of the development effort is devoted to the manual, incremental and often ad hoc traversal of this decision space.

We may introduce a static adaptation function S,

$$S(E, p, i) \quad = \quad E'$$

where E is an arbitrary expression in our DSL, p is a set of relevant run-time parameters and i provides optional profiling information that is collected after every program

execution. Invocation of S forces evaluation of all instances of **ANY** in E to generate a simplified expression E′, using p and i to guide the choices between alternatives. The new expression E′ describes a functional subgraph of E.

S allows adaptive selection between program variants by returning a subexpression of E, E′, that describes a functionally equivalent subgraph of the flow graph to be constructed. The profiling information i is feedback of the measured value of a performance metric collected from previous decisions and allows the adaptation function S to decide between alternatives based on their learned suitability for run-time parameters p.

3.3.2 *Dynamic Adaptation with Instrumented Queues*

At run-time, execution resources on the platform, whether CPU cores or streaming multiprocessors on a GPU, need to be matched to computational operations. To execute on the device, these operations must be performed in the context of operating system threads or processes. Thus, there are two sets of pairwise mappings between three sets, (i) the execution resources available on the hardware platform, (ii) the operating system threads and processes and (iii) the computational operations to be performed.

Although **ANY** is potentially able to represent a wide range of functionally equivalent programs including alternative structures and refactorings, we only consider the mapping of execution resources in a runtime environment to the computational operations of a program. Two implementation forms of the structured graphs will be considered:

1. **Competitive Scheduling with Native Threads or Processes:**
   A one-to-one mapping between operations and native threads or processes as in FastFlow, Intel TBB and similar frameworks [4, 104]. The complexity, in terms of the number of operations is constrained by the performance overhead and limitations on the number of threads and processes imposed by the operating system. As effectively, only the mapping from threads or processes to hardware resources remains, there is limited scope for runtime decision making.

2. **Cooperative Scheduling with Dynamic Thread or Process Pools:**

   Sets of dynamic thread or process pools are allocated to subsets of the available computational operations as determined by their hardware requirements. The pools themselves may be threads (as with OpenMP and pthreads) or processes (as with MPI or native processes) and are individually mapped on some primary hardware resource. This implementation style circumvents the operating system limitations and performance penalties associated with larger programs of greater complexity and allows decisions to be made with a broader range of possibilities.

   To allow redirection of resources to stages as determined by the requirements of the program in a runtime environment, a thread or process pool is allocated

to subsets of all available pipeline stages. Alternative stages are selectively executed on available hardware as determined by a heuristic algorithm. Thus, the effect of the adaptive runtime policy is to determine the probability $p_{ij}(t)$ of executing pipeline stage i at any time t. This *stochastic allocation* avoids the premature choice that is associated with a one-to-one mapping of individual stages to native operating system threads or processes and follows our deferred choice approach.

Given a run-time performance metric, we may formulate this as a Markov Decision Problem over the space of queue vectors where $P(S_{t+1}|S_t, A_t)$ describes the dynamics of the system as the probability that the effect of action A_t performed in state S_t is a transition to state S_{t+1}.

The typical approaches for solving Markov Decision Problems involve dynamic programming and the application of policy and value iteration algorithms to determine the optimal policy function $\pi : S \to A$ that provides a mapping between state S_t and an optimal action A_t. However, the policy and value iteration schemes require *a priori* knowledge of the transition probabilities.

The cost or penalty function is a mapping from the set of states to a real number $c : S \to \mathbb{R}$. In the language of Markov Decision Processes, the cost function is related to the value function that completely describes a policy by determining the associated value of any chosen action in a given state. It is related to but distinct from the overall objective function that is to be optimised.

3.4 SUMMARY

Therefore our adaptive run-time may be realised as a higher order function with the following arguments Table 3.1 on page 30:

1. a structured flow graph, with associated operations,
2. a static adaptation function that takes profiling information and returns a specialised subgraph
3. a filter function for the expression $\texttt{get}(q, \texttt{filter})$, this determines the selection criteria
4. a reduce function to perform a measure on the associated queue
5. a hash function to determine the queuing discipline
6. a local feedback function to be performed before and after an operation, this determines the associated metadata
7. a global state cost function

	Function	Arguments	Returns	Description
	hash	item, metadata	real number	hash function to determine the queuing discipline
	reduction operator	metadata	real number	reduction operator for summary of queue information, e.g. count, mean, max, etc
	filter	metadata	item	selection criteria
	reward	item, metadata	real number	called before and after an operation, (operation, post_reward-pre_reward) is appended to item's metadata
	static adapter	graph	graph	returns a subgraph of the program

Table 3.1: Function parameters in the generalised adaptation framework for static and dynamic adaptation.

4

HIGH PERFORMANCE POLYCINS MODELLING

This chapter introduces the SCATTER application as a case study in high performance computing to which the adaptation framework presented in Chapter 3 will be applied in subsequent chapters. While it presents a scalable pattern-based implementation following Mattson's approach, the following represent a direct contribution to the research domain:

1. A high performance parallel implementation of the **Scatter** code.
2. A new graphical analysis frontend, the *Profile Refinement Tool* **Prefit**, that is a software interface to the novel polyCINS analysis methodology. PREFIT facilitates the identification of coherence features in theoretical models and experimental data that serve as inputs to a multi-stage iterative refinement process Figure 4.16 on page 55.
3. Integration with **Paraview** [25, 46], the high performance analysis and visualisation package from Sandia National Laboratories. Paraview's multi-tier architecture allows dynamic visualisations in distributed parallel environments of the large datasets arising from simulation in SCATTER.

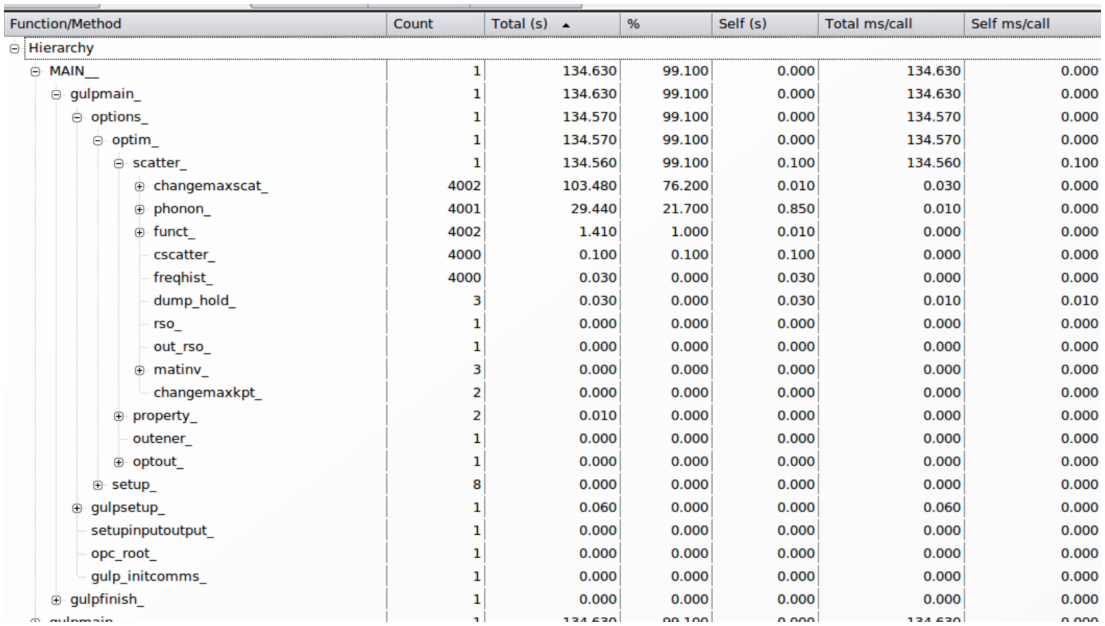
Appendix C presents details of PREFIT and Paraview integration, a high level overview of the analysis process and examples of visualisations and their use for coherence feature identification.

4.1 SCATTER ON SHARED AND DISTRIBUTED MEMORY PLATFORMS

The practical feasibility of INS modelling in SCATTER depends on the availability of a high performance implementation. While previous attempts by the GULP authors to incorporate parallelism have resulted in modest performance gains with limited scalability, larger models require significant computational resources. Therefore, a high performance implementation of SCATTER, and the relevant modules of the GULP program, has been undertaken to target both shared and distributed memory parallel platforms as well as platforms with integrated GPU accelerators (Chapter 5). This implementation, derived from the original serial version, systematically follows the prescriptive pattern approach of Mattson [89].

4.1.1 *Finding Concurrency*

The initial stage of the pattern approach examines the performance of SCATTER for an indicative model and attempts to identify computational bottlenecks. Profile-driven



Function/Method	Count	Total (s) ▲	%	Self (s)	Total ms/call	Self ms/call
⊖ Hierarchy						
⊖ MAIN__	1	134.630	99.100	0.000	134.630	0.000
⊖ gulpmain_	1	134.630	99.100	0.000	134.630	0.000
⊖ options_	1	134.570	99.100	0.000	134.570	0.000
⊖ optim_	1	134.570	99.100	0.000	134.570	0.000
⊖ scatter_	1	134.560	99.100	0.100	134.560	0.100
⊕ changemaxscat_	4002	103.480	76.200	0.010	0.030	0.000
⊕ phonon_	4001	29.440	21.700	0.850	0.010	0.000
⊕ funct_	4002	1.410	1.000	0.010	0.000	0.000
cscatter_	4000	0.100	0.100	0.100	0.000	0.000
freqhist_	4000	0.030	0.000	0.030	0.000	0.000
dump_hold_	3	0.030	0.000	0.030	0.010	0.010
rso_	1	0.000	0.000	0.000	0.000	0.000
out_rso_	1	0.000	0.000	0.000	0.000	0.000
⊕ matinv_	3	0.000	0.000	0.000	0.000	0.000
changemaxkpt_	2	0.000	0.000	0.000	0.000	0.000
⊕ property_	2	0.010	0.000	0.000	0.000	0.000
outener_	1	0.000	0.000	0.000	0.000	0.000
⊕ optout_	1	0.000	0.000	0.000	0.000	0.000
⊕ setup_	8	0.000	0.000	0.000	0.000	0.000
⊕ gulpsetup_	1	0.060	0.000	0.000	0.060	0.000
setupinputoutput_	1	0.000	0.000	0.000	0.000	0.000
opc_root_	1	0.000	0.000	0.000	0.000	0.000
gulp_initcomms_	1	0.000	0.000	0.000	0.000	0.000
⊕ gulpfinish_	1	0.000	0.000	0.000	0.000	0.000

Figure 4.1: changemaxscat and phonon dominate execution time.

analysis provides insights into the relationship between model complexity, input parameters and execution time that are relevant to the identification of potential concurrency.

4.1.1.1 *Execution Profiling*

The time demand of various SCATTER subroutines are obtained by execution profiling [129] to inform the subsequent optimisation and parallelisation process. Information about the sequence, frequency and total time spent in functions or subroutines are extracted from the profiling data obtained with the GNU profiler `gprof` [65], which is widely available as part of the GNU Compiler Collection (GCC) and is extensively used for the generation of flat and hierarchical profiles.

With `gprof`, an example coarse-resolution test model in the serial version of SCATTER, with 200 shells (`nq_step`) and 20 angular steps (`nq_intstep`) in θ and ϕ, produces the hierarchical call graph in Appendix B (some entries are omitted for brevity). This information is presented graphically in (Figure 4.1 on page 32).

The profiler output indicates that 99% of execution time is spent in SCATTER itself, of which 76.2 % and 21.7 % are dedicated to the `changemaxscat` and `phonon` subroutines respectively. The relevant subroutine invocations exist within a nested loop structure (Listing 4.1), supporting the observation that the most significant fraction of computation time for scientific and technical programs is spent in loop execution [10].

For the coarse-resolution test model, there are predictably `(nq_step`$\times$`nq_intstep)` = 4000 iterations of the nested kernel evaluation loops. These nested loops dominate SCATTER execution and correspond to the evaluations of scattering for individual points in RSO-sampled space. In principle, individual iterations should be independ-

```
1       do shellcount = 1, nq_step
           ...
3          do thetacount = 1, nq_intstep
              ...
5             call changemaxscat
              ...
7             call phonon(.true.,fc)

9             !   initiate calls to scattering kernels...
              call cscatter(nq_intstep,Qvector,tauvector,scatlencoh,sofomega,sflag)
11                        ...
           enddo  ! over theta
13         ...
        enddo ! over shells
```

Listing 4.1: Primary Nested Loop Structure in Scatter. The Profile indicates that 99% of execution time is spent in these loops.

ent operations that allow the adoption of several possible parallel partitioning schemes. However, incidental dependencies introduced in the course of a sequential implementation in a highly procedural Fortran style, with deeply nested loops and multiple global arrays in modules shared over a significantly large codebase require resolution before this is possible.

4.1.2 *Algorithm Structure*

The pattern approach classifies parallel algorithm structures into three types on the basis of organisation [89]. They may be organised by:

1. Task (Task Parallelism, Divide and Conquer)
2. Data Decomposition (Geometric Decomposition, Recursive Data)
3. Flow of Data (Pipeline, Event Based Coordination)

In principle, the distinct tasks corresponding to individual loop iterations identified in the previous section are completely independent and this problem falls into the *embarrassingly parallel* class of problems to which a task parallel algorithm is particularly applicable [89].

The choice of task parallelism is justified by the following considerations:

1. These tasks are associated with loop iterations, are completely defined in number and scope at the start of computation and must all complete before the solution is found.
2. The individual tasks present the same computational requirements and generally constitute balanced loads to the execution elements. This translates to higher efficiency as resources may otherwise be wasted in the absence of complex scheduling.

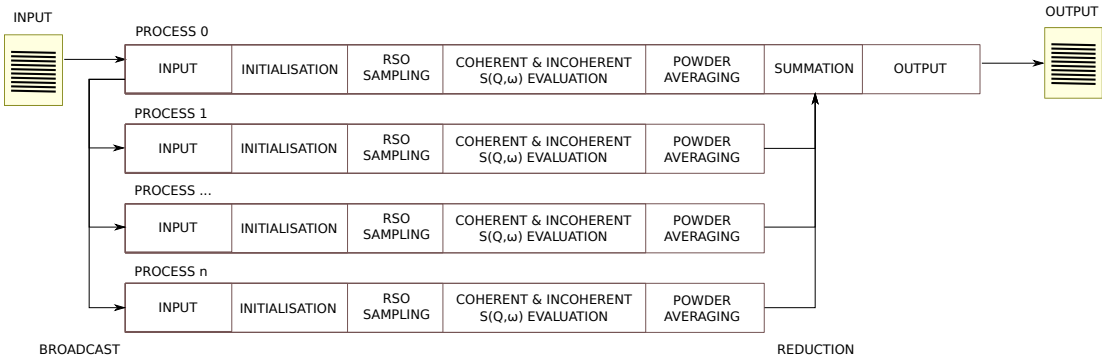


Figure 4.2: Coordination and communication structure between MPI processes in SCATTER.

3. The associated computation within the tasks is sufficiently significant to justify any overhead created for task initiation and management by the implementation.

4.1.3 *Supporting Structure*

In identifying the supporting structure, the transition from design to implementation begins. Typical, and occasionally overlapping, patterns at this level include the SPMD (Single Program Multiple Data), Master-Worker or Farm, Loop Parallelism and Fork-Join structural patterns.

A Loop Parallelism supporting structure with individual iterations distributed between execution units is natural given that loops form basis of the task parallelism described in the algorithm structure.

The SPMD model and the supporting message passing framework provided by MPI is already used by other GULP routines. All processing elements execute the same program on different data, making use of the process rank as a unique identifier to select subsets of a larger data structure and to achieve customised behaviour.

Figure 4.2 on page 34 outlines SCATTER implemented with the SPMD model. Parallel execution begins with the initial program input deck, of relatively small size, read via standard input and distributed to cooperating processes by the root process (*Process 0*). With complete details of all execution parameters, each process proceeds to independently generate the entire global sample space and perform the actual SCATTER kernel evaluation. In the final stages, each process generates a three-dimensional $S(Q,\omega)$ "histogram" representing the polycrystalline average for the relevant region of the partitioned sampled space that represents a local contribution to the final result. A final global summation reduction operation communicates these results to the root process, merging the data generated from each task into a polycrystalline average that is written to a specified output file.

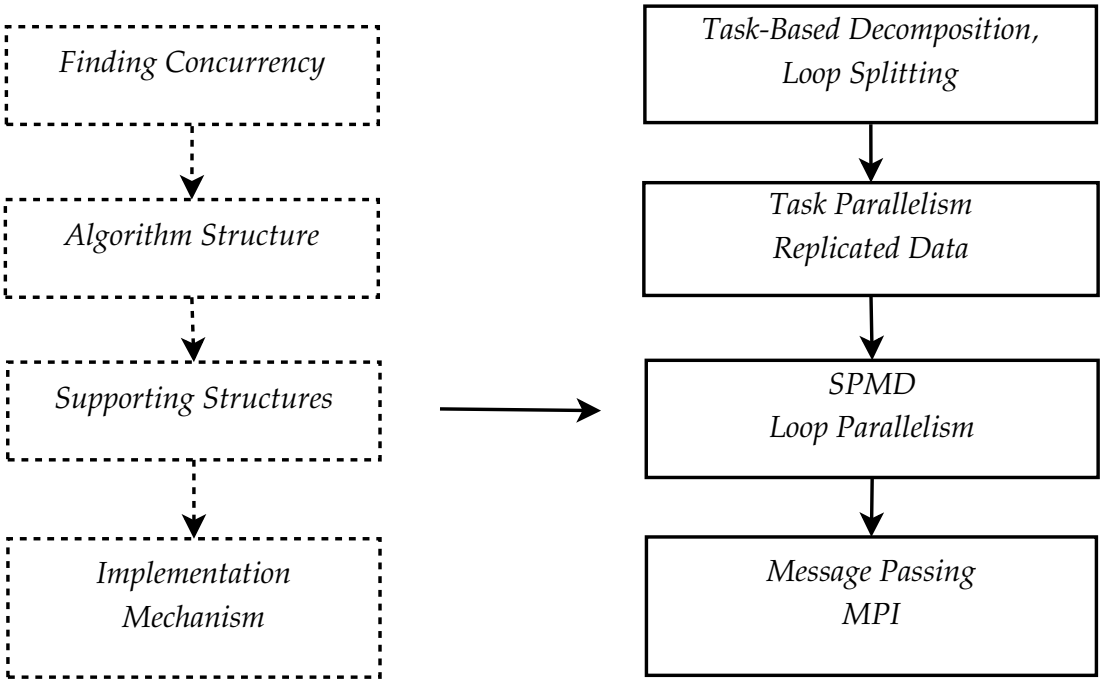


Figure 4.3: Complete pattern outline for the parallel Scatter implementation

4.1.4 *Implementation Mechanism*

The implementation mechanism forms the low-level framework and interface that provides facilities for the management of:

1. Processing Elements
2. Synchronisation
3. Communication

and provides an execution environment for the parallel program.

4.1.4.1 *Preliminary Optimisations*

1. Dynamical Matrix

   Scatter's Reciprocal Space Onion (RSO) sampling yields a discrete grid of points in spherical polar coordinates over which scattering contributions are evaluated. These evaluations are computationally demanding for systems of even moderate complexity. For each unique triple ($|\mathbf{Q}|$,θ,ϕ), corresponding to a point P in RSO-sampled space, Gulp derives a new dynamical matrix, by summation of phased second derivative contributions of neighbouring atoms within a cut-off distance, and proceeds to compute the associated phonon modes required by Scatter. Repeated derivation of the dynamical matrix corresponding to each of these points in the spherical grid is computationally expensive for potential models that consider large numbers of nearest neighbour interactions.

```
do shellcount = 1,nq_step
  ...
  do thetacount = 1, nq_intstep
    phi = 0.0_dp
    ...

    call changemaxscat
    ...

    call funct(2_i4,nvar,xc,fc,gc)
    call phonon(.true.,fc)

    !calls to scattering kernels...
    call cscatter(nq_intstep,...)
        ...
  enddo  ! over theta
  ...
enddo ! over shells
```

Listing 4.2: Pre-optimised primary loop in SCATTER. changemaxscat is invoked in every iteration, leading to significant overhead.

Figure 4.4 on page 37 is a visualisation of a representative dynamical matrix based on the potential model of D.W. Brenner [20] for a 240-Carbon atom system, the low temperature cubic phase of C60. It illustrates a pattern of sparsely regular non-zero blocks corresponding to the pairwise derivatives of the nth nearest neighbour atoms at a given phase angle. The Brenner potential is notably expensive to calculate in GULP as it has a relatively complex form (being a bond order potential) and accounts for a large number of possible neighbour interactions (having a long spatial cutoff). As an optimisation strategy, the intermediate first and second order derivative vectors for each atom may be cached in a space-efficient dynamic linked list to avoid recalculation. Subsequent dynamical matrices are generated by summation of the cached vectors at the appropriate phase angle.

In principle, this optimisation approach is applicable to a wider range of models and effectively reduces the dominant aspect of the computation to eigenvector and eigenvalue determination, leaving a standard numerical linear algebra problem for which efficient numerical solution techniques exist, as extensively discussed in [69, 40].

2. Redundant Memory Allocation

   In Section 4.1.1.1, profiling SCATTER reveals that approximately 75 % of execution time is spent in the `changemaxscat` subroutine. This overhead is created by the repeated reallocation of several large arrays for every iteration of the inner loop in listing 4.2. However, the dimensions of the dynamic arrays allocated remain constant between iterations.

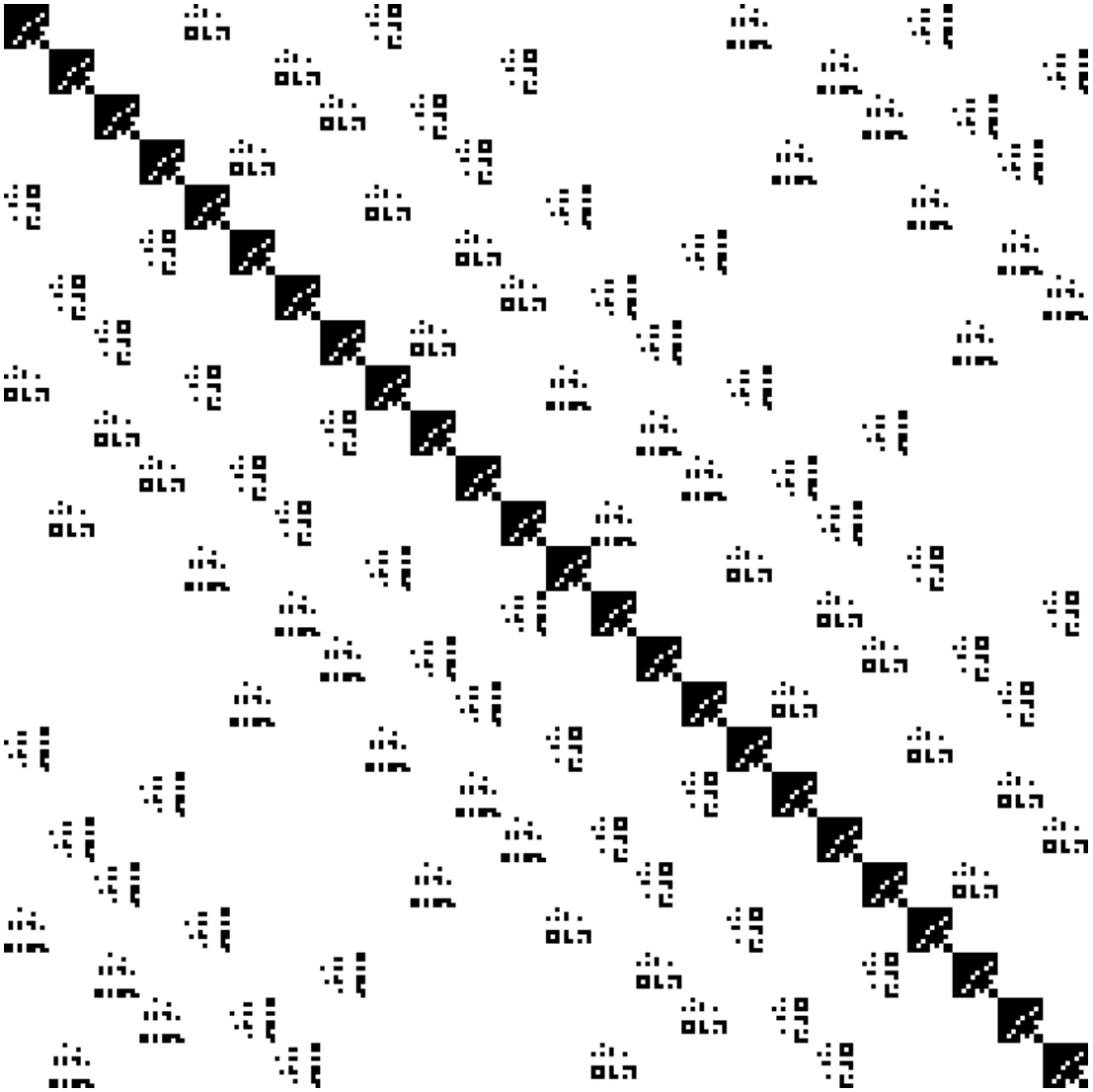


Figure 4.4: Sparse visualisation of the Hermitian dynamical matrix for a 240-atom Carbon nanotube model with 720 modes.

```
call changemaxscat
do shellcount = 1,nq_step
  ...
  do thetacount = 1, nq_intstep
    phi = 0.0_dp
    ...
    call funct(2_i4,nvar,xc,fc,gc)
    call phonon(.true.,fc)

    !calls to scattering kernels...
    call cscatter(nq_intstep,...)
        ...
  enddo  ! over theta
  ...
enddo ! over shells
```

Listing 4.3: Post-optimised primary loop in SCATTER. changemaxscat is invoked only once before the loop, significant overhead is eliminated leading to improved performance of 450 % in the test case.

Invoking `changemaxscat` once before the loop and reusing memory between iterations can avoid this performance penalty. The impact of this minor modification (listing 4.3) is significant. Further profiling reveals that time spent in the `changemaxscat` subroutine is almost completely eliminated, leading to a performance increase of 450 % for the test model.

4.1.4.2 *Partitioning by Loop Splitting*

As part of the supporting structure, loop parallelism minimises the need for a detailed understanding of the complex algorithms that provide SCATTER functionality. One of the guiding principles behind loop parallelism is a *sequential equivalence* or the expectation that both sequential and parallel versions of the program produce identical output, disregarding round-off errors.

Partitioning of the loop iteration space relies on transformations to the sequential loop that are semantically neutral. While the code fragments that follow are based on Fortran 90 and the GULP SPMD model, the underlying transformations are language agnostic and can be expressed with equal validity in other languages and implementation platforms.

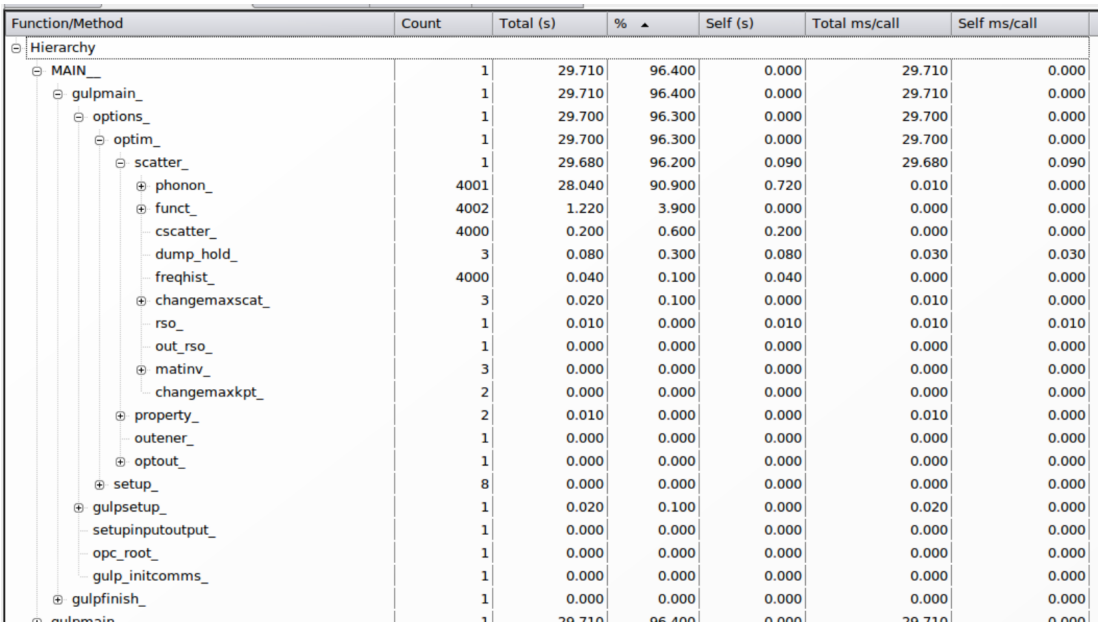
1. Dependency Elimination.

   The new statements

   ```
   Qmodu = init_par(1) + init_par(3) * (shellcount - 1)
   theta = 0.0_dp      + init_par(4) * (thetacount - 1)
   ```

   decouple the cumulatively increasing values of `Qmodu` and `theta` from previous iterations in the program, a modification necessary to satisfy the dependency conditions.

   Also relevant are deeply nested implicit dependencies in the code from subroutine calls and their own nested loop structures that are non-evident. These



Function/Method	Count	Total (s)	% ▲	Self (s)	Total ms/call	Self ms/call
Hierarchy						
MAIN__	1	29.710	96.400	0.000	29.710	0.000
gulpmain_	1	29.710	96.400	0.000	29.710	0.000
options_	1	29.700	96.300	0.000	29.700	0.000
optim_	1	29.700	96.300	0.000	29.700	0.000
scatter_	1	29.680	96.200	0.090	29.680	0.090
phonon_	4001	28.040	90.900	0.720	0.010	0.000
funct_	4002	1.220	3.900	0.000	0.000	0.000
cscatter_	4000	0.200	0.600	0.200	0.000	0.000
dump_hold_	3	0.080	0.300	0.080	0.030	0.030
freqhist_	4000	0.040	0.100	0.040	0.000	0.000
changemaxscat_	3	0.020	0.100	0.000	0.010	0.000
rso_	1	0.010	0.000	0.010	0.010	0.010
out_rso_	1	0.000	0.000	0.000	0.000	0.000
matinv_	3	0.000	0.000	0.000	0.000	0.000
changemaxkpt_	2	0.000	0.000	0.000	0.000	0.000
property_	2	0.010	0.000	0.000	0.010	0.000
outener_	1	0.000	0.000	0.000	0.000	0.000
optout_	1	0.000	0.000	0.000	0.000	0.000
setup_	8	0.000	0.000	0.000	0.000	0.000
gulpsetup_	1	0.020	0.100	0.000	0.020	0.000
setupinputoutput_	1	0.000	0.000	0.000	0.000	0.000
opc_root_	1	0.000	0.000	0.000	0.000	0.000
gulp_initcomms_	1	0.000	0.000	0.000	0.000	0.000
gulpfinish_	1	0.000	0.000	0.000	0.000	0.000

Figure 4.5: Hierarchical execution profile of Scatter after reorganisation. The changemaxscat overhead is eliminated, achieving a 450 % performance increase.

dependencies are ad hoc artifacts of an implementation. Although compilers may perform extensive dependency analysis [74], no tools exist to allow a satisfactory and systematic approach to their identification and elimination. Ultimately, these dependencies were resolved by extensive testing and review of the code .

2. Loop Collapse

   This transformation combines nested loops into a single larger loop. Applied to nested loops, the result is a larger merged iteration space that is more readily parallelised. The loop transformation takes the following general loop structure

```
do i = 1, M
  do j = 1, N
    F(i,j)
  enddo
enddo
```

   and transforms it into

```
do k = 1, M*N
  i = (k-1) / N + 1
  j = (k-1) % N + 1
  F(i,j)
enddo
```

   A necessary condition for the validity of this transformation is that the loop iterations are independent i.e. $F(i,j)$ does not depend on $F(k,l)$ for all $i \neq k$ and $j \neq l$.

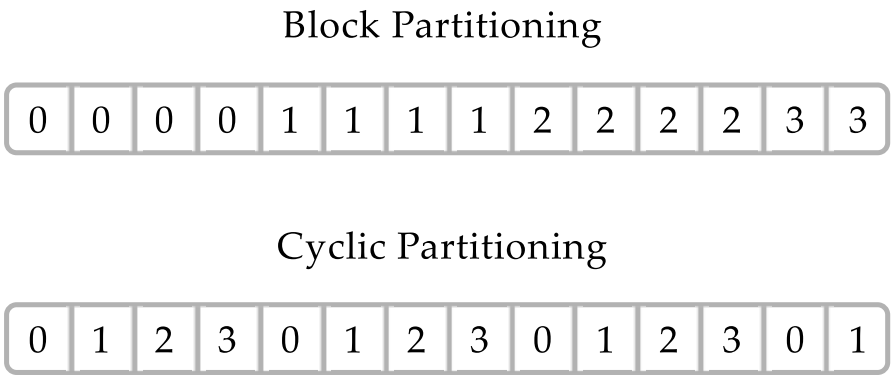


Figure 4.6: Example block and cyclic loop partitioning schemes for 14 iterations over 4 processors [9].

This approach has the advantage of increasing the number of iterations that may be distributed between tasks, improving the granularity and scalability of the parallel implementation to a larger number of execution elements.

3. Loop Partitioning

   With semantically neutral transformations applied, the loop iterations are completely independent. The next problem is the division of loop iterations between execution elements. As a basic requirement for efficiency, the number of loop iterations should exceed the number of execution elements or resources will be underutilised. Two techniques for achieving this data distribution include (Figure 4.6 on page 40):

   a) Block Partitioning

      The loop iterations are split into n contiguous chunks allocated to n different processes. Data locality and cache coherence improve performance and reduce communication costs in problems where dependencies exist. In the simplest cases, the number of loop iterations to be split is an exact multiple of the number of execution elements and an exact distribution can be achieved with ideal balancing between these processes. However, this is not generally the case and it may be necessary to apply an uneven partitioning scheme.

   b) Cyclic Partitioning

      The loop iterations are distributed consecutively between execution elements until the last execution element is reached and subsequent assignment continues from the first in a cyclical manner. Cyclic distribution is straightforward to implement and ensures good load distribution in a simple implementation. However, communication costs may be higher than that of block distribution if it is necessary to use the results of adjacent computations and cache behaviour may be inefficient for fine-grained problems. Cyclic distribution is appropriate to this problem, allowing good load distribution with minimal modifications to existing code.

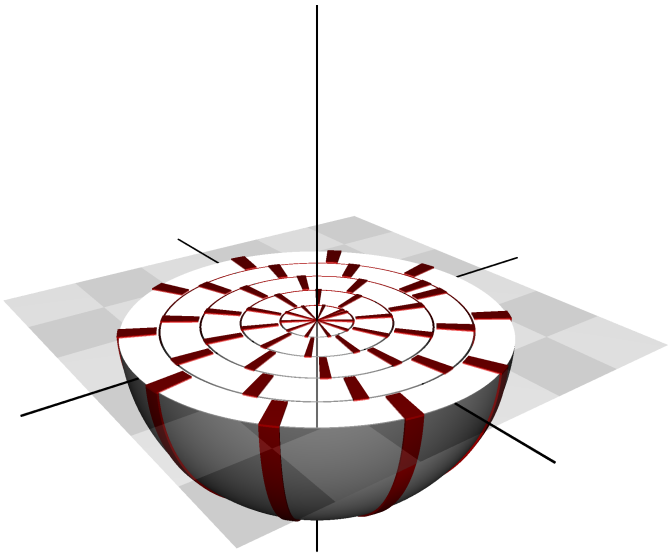


Figure 4.7: Cyclic parallel partitioning scheme for a single process. Dark regions represent constant-ϕ values. The light sections of the RSO grid are ignored by this process. A round-robin assignment of circles of constant ϕ between processes provides a compromise between simplicity and the availability of sufficient independent work units to scale to a large number of parallel processes.

The evaluations of scattering for points in RSO-sampled space are, in principle, independent operations that allow the adoption of several possible parallel partitioning schemes. Given the large number of points in a typical RSO grid, a cyclic space decomposition, with a round-robin assignment of constant-ϕ circles between processes (Figure 4.7 on page 41), provides sufficient independent tasks to scale to a large number of parallel processes. As a possible optimisation, block partitioning may eliminate some of the redundancy associated with the calculation of energies for a given value of θ, however this offers only marginal performance gains for current models when other optimisations are applied.

The two variables nprocs and procid are introduced to represent the number of parallel executing processes and the unique identifier for each parallel process respectively ($0 \leqslant$ procid $<$ nprocs). Where the starting position for each process is offset by its procid and nprocs is the stride, a general cyclic partitioning takes the following form:

```
1 do i = start, finish
    ...
3 enddo
```

```
call changemaxscat
do stcount = procid+1, nq_step*nq_intstep, nprocs
  shellcount =    (stcount - 1)/nq_intstep  + 1
  thetacount = mod(stcount - 1, nq_intstep) + 1

   Qmodu = init_par(1)  + init_par(3) * (shellcount - 1)
   theta = 0.0_dp       + init_par(4) * (thetacount - 1)
    ...
   call funct(2_i4,nvar,xc,fc,gc)
   call phonon(.true.,fc)

   !   initiate calls to scattering kernels...
   call cscatter(nq_intstep,...)
   ...
   enddo  ! over theta
   ...
enddo ! over shells
```

Listing 4.4: Transformed SCATTER primary loop structure with eliminated dependencies, collapsed loops and cyclic partitioning

```
do i = start + procid, finish, nprocs
  ...
enddo
```

and for a nested loop with loop collapse:

```
do i = 1, M
  do j = 1, N
    F(i,j)
  enddo
enddo
```

```
do k = procid + 1, M*N, nprocs
  i = (k-1) / N + 1
  j = (k-1) % N + 1
  F(i,j)
enddo
```

After the application of all of these transformations, the original kernel evaluation loop structure in Listing 4.1 assumes the form of Listing 4.4.

4. I/O

Input is distributed to cooperating processes by the root process in a MPI broadcast operation at program initiation. With complete details of all execution parameters, each process independently computes $S(Q, \omega)$ over an appropriate subset of the global sample space in a rank-based domain-decomposition of the

	IBM BladeCenter JS21	IBM pSeries 575 (Huygens)
Nodes	4	101
Processing Elements per Node	4	32
Processor Clock	2.3 GHz	4.7 GHz
Architecture	IBM PowerPC 970MP	IBM Power6
Memory per Node	4 GB	128 GB
Network	FastEthernet 100 Mbps	Infiniband 160 Gbit/s
Operating System	Redhat Linux (kernel 26.18-8)	GNU Linux (kernel 2.6.27)
Compiler	GCC gfortran 4.1.2 (-O3)	GCC gfortran 4.3.2 (-O3) IBM XL Fortran for Linux, V12.1 (-O3 -qstrict -qarch=auto -qtune=auto)
MPI Version	OpenMPI 1.4.3	OpenMPI 1.3.3

Table 4.1: Multicore/Multinode Test Configurations

primary SCATTER loop iteration space. A final global summation reduction operation merges these contributions to the root process into a final polycrystalline average for the scattering system.

The intermediate calculation results, detailing individual eigenvector contributions at each point to the overall scattering intensities, have an important role in subsequent analysis. A new HDF5 output format over MPI/IO replaces the previous verbose textual output. MPI/IO provides scalable, distributed, simultaneous output of this large dataset, allowing nodes in a cluster environment to take advantage of dedicated high-speed interconnects that are capable of significantly reducing the associated communication overhead.

	Dell Precision T7500 Workstation	Xookik
Nodes	1	4
Processing Elements per Node	4	12
Processor Clock	2.0 GHz	3.07 GHz
Architecture	Intel x86-64	
Memory per Node	4 GB	50 GB
Network	FastEthernet 100 Mbps	Infiniband 160 Gbit/s
Operating System	Ubuntu Linux 11.04 (kernel 2.6.38-8)	GNU Linux (kernel 2.6.27)
Compiler	GCC gfortran 4.4.5 (-O3)	GCC gfortran 4.3.2 (-O3)
MPI Version	OpenMPI 1.4.3	OpenMPI 1.3.3

Table 4.2: Heterogeneous Test Configurations

4.2 EVALUATION

4.2.1 *Optimisation*

For the optimisations of subsubsection4.1.4.1, Figure 4.8 on page 45 represents the un-optimised and optimised runtimes for a 40-atom Single-Walled Carbon Nanotube model, $C(10,10)$, and a 60-atom C_{60} Buckminsterfullerene model, both involving the Brenner potential, on the Dell Precision T7500 Server. Optimisation significantly lowers the computational cost of calculating the dynamical matrix for the class of models that use the Brenner potential, yielding model-dependent performance increases between a factor of 10 and 50 in overall runtime. The remaining runtime is dominated by calculation of eigenvectors and eigenvalues of the dynamical matrix.

4.2.2 *Comparative Performance*

Comparing the performance of the unoptimised pattern-based SCATTER implementation against the original ad hoc GULP implementation on the IBM BladeCenter JS21, Figure 4.9 on page 46 shows ideal scaling in the new version for a simulation of Graphite with the Young-Koppel potential with up to 16 processes across 4 nodes. In contrast, the original GULP implementation begins to exhibit significant performance degradation at 6 processes and above.

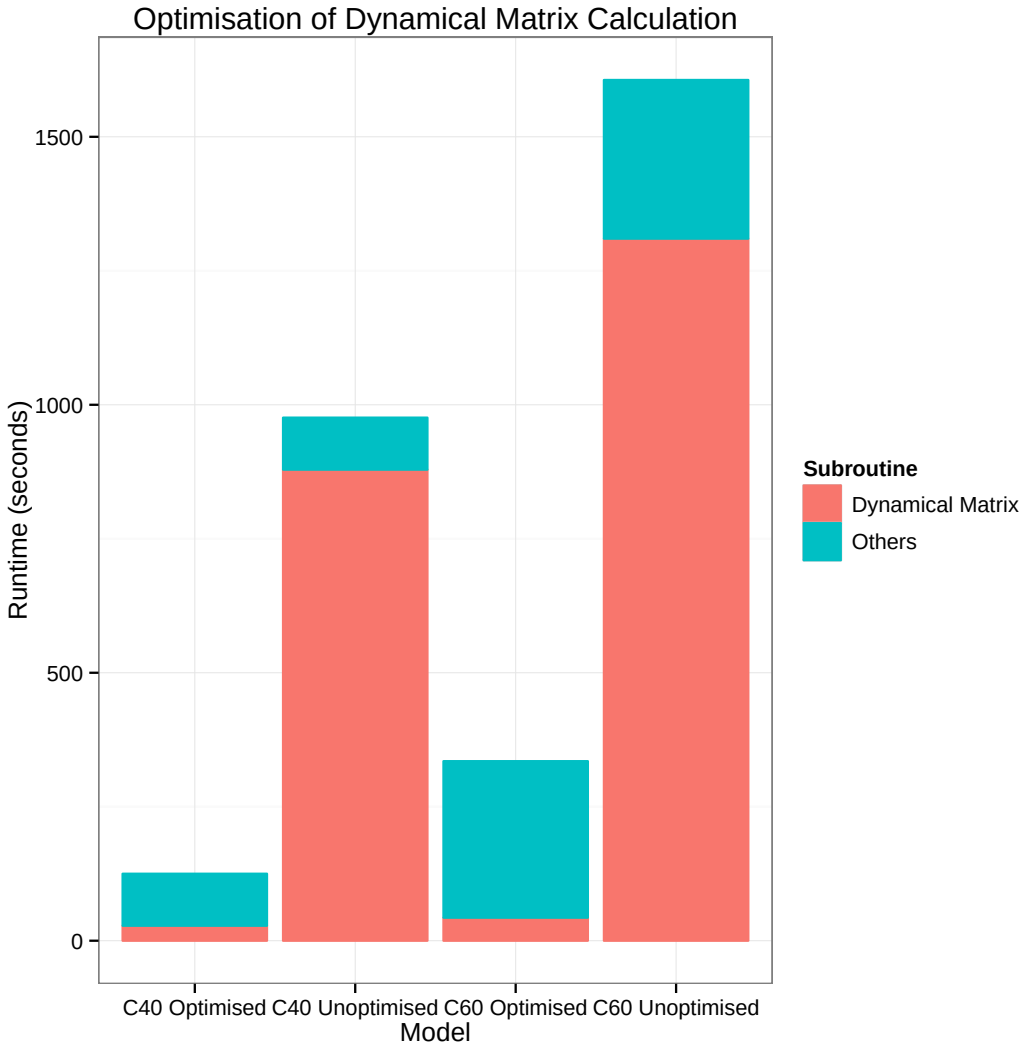


Figure 4.8: Unoptimised and optimised runtimes for 40-atom Single-Walled Carbon Nanotube, $C(10, 10)$, and 60-atom C_{60}, Buckminsterfullerene models with the Brenner potential. Optimisation significantly lowers the computational cost of calculating the dynamical matrix for the class of models that use the Brenner potential, yielding model-dependent performance increases of between $10\times$ to $50\times$ in overall runtime.

Performance however deviates from ideal for the un-optimised run of the Brenner model in Figure 4.10 on page 47 where the nature of the model reveals the limitations of the interconnecting network and limiting characteristics of the machine appear as discernible consecutive groupings of four timing values with a consistent pattern of variation. Here, the original SCATTER implementation fails to complete for all but a few test cases where the number of processes is a factor of 10.

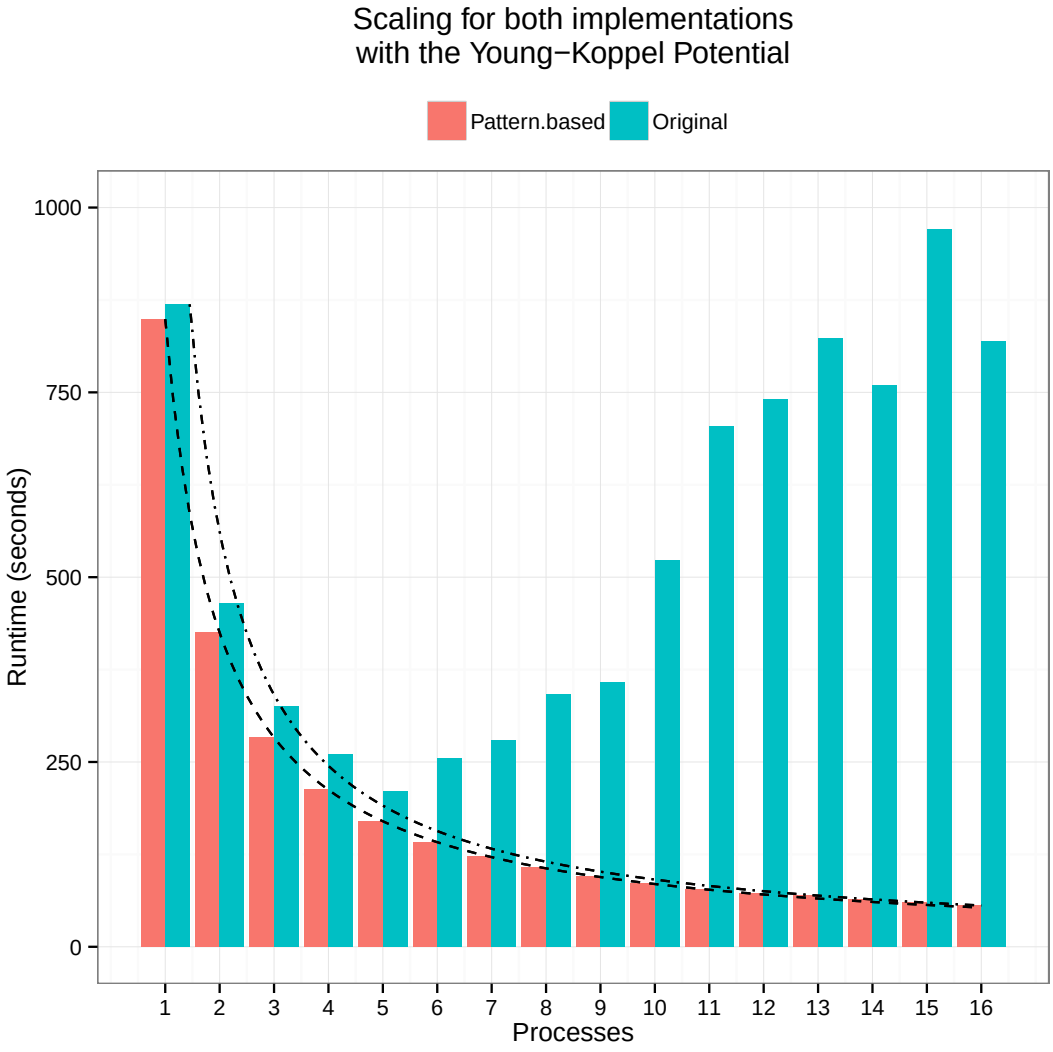


Figure 4.9: Execution time in seconds for the Young Koppel Model on the test machine with 1 to 16 processes compared against ideal scaling (dotted line). Our version of SCATTER exhibits ideal scaling up to 16 processes.

4.2.3 *Scalability*

4.2.3.1 *Calibration*

The runtime of a SCATTER model may be predicted from the relationship of (4.1) where t is the approximate completion time, k is a constant for a given execution environment and model, $|\mathbf{Q}|_{max} - |\mathbf{Q}|_{min}$ is the difference between the maximum and minimum magnitudes of the momentum transfer vector $\mathbf{Q}$, $\delta\mathbf{Q}$ is the finite increment in momentum transfer between successive RSO shells and $\delta\theta = \delta\phi$ is the finite change in angular orientation of the momentum transfer vector.

$$t \approx k\left(\frac{|\mathbf{Q}|_{max} - |\mathbf{Q}|_{min}}{\delta|\mathbf{Q}|}\right) \times \left(\frac{2\pi}{\delta\theta}\right)^2 \tag{4.1}$$

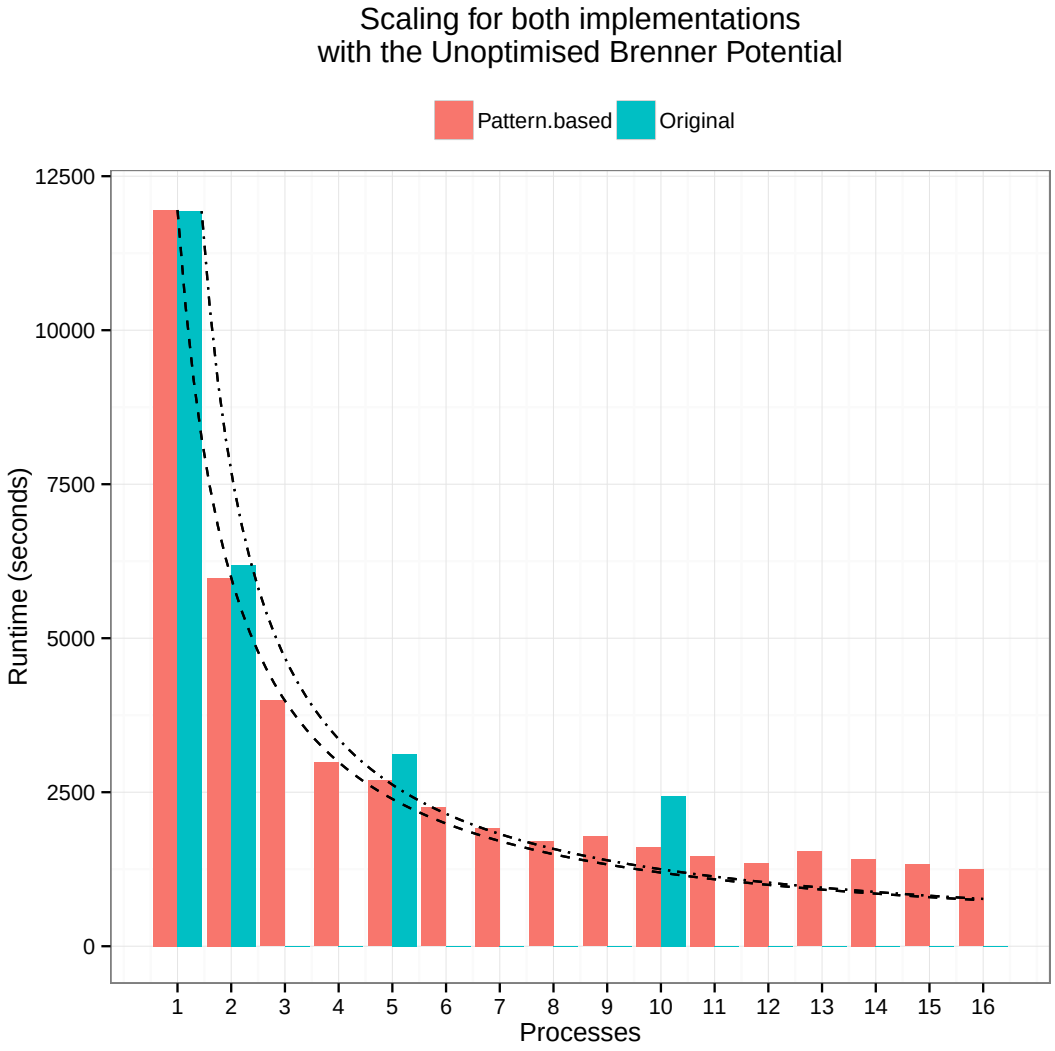


Figure 4.10: Execution time in seconds for the unoptimised Brenner Model on the test machine with 1 to 16 processes compared against ideal scaling (dotted line). Our version of Scatter deviates moderately from ideal scaling. The Gulp version fails to run for most test cases.

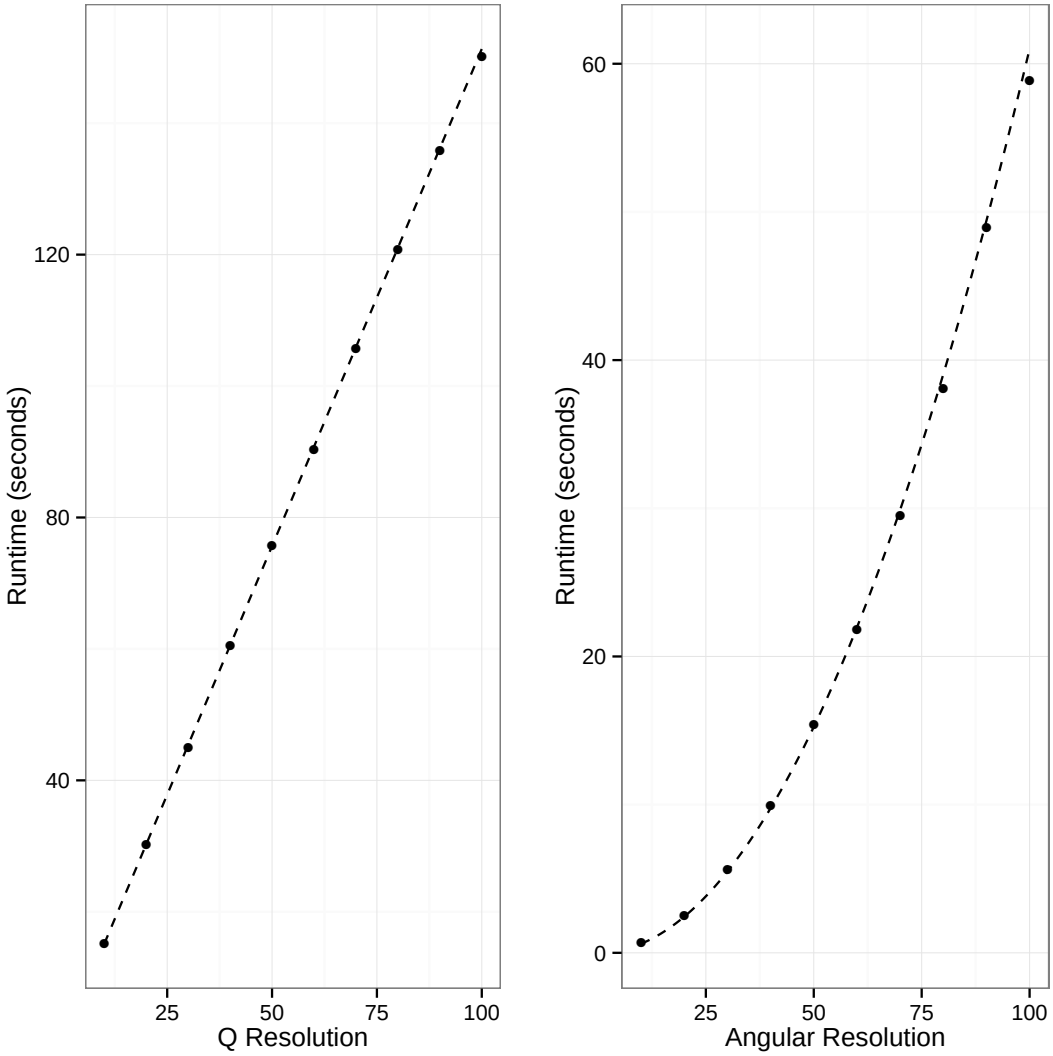


Figure 4.11: Predicted (solid line) vs. actual (data points with dashed line fit) problem scaling by model resolution for a 60-atom Carbon nanotube with (a) $\left(\frac{|\mathbf{Q}|_{max}-|\mathbf{Q}|_{min}}{\delta\mathbf{Q}}\right)$ shells demonstrating linear scaling and (b) $\frac{2\pi}{\delta\theta}$ angular steps in θ and ϕ demonstrating quadratic scaling.

The runtime t is determined by the number of points $P(|\mathbf{Q}|, \theta, \phi)$ in RSO space for a given model resolution as specified in the parametric inputs to the program. These integer-valued parameters are the number of shells, $\left(\frac{|\mathbf{Q}|_{max}-|\mathbf{Q}|_{min}}{\delta|\mathbf{Q}|}\right)$, and the number of angular steps, $\frac{2\pi}{\delta\theta} = \frac{2\pi}{\delta\phi}$. Figure 4.11 on page 48 compares predicted and actual problem scaling for a 60-atom model. (4.1) allows the estimation of full model runtime by calibration against a low-resolution test case to determine k for an execution environment.

4.2.3.2 *Scalability Testing*

Figure 4.12 on page 50 presents the execution times for the 60-Carbon atom model with 4, 8, and 16 processes on the IBM JS21 BladeCenter. SCATTER demonstrates linear scaling across multiple cores and multiple nodes. Scaling is compared against the estimated sequential runtime obtained from a calibration run with a coarse RSO grid as predicted by Equation 4.1.

Subsequently, the complete performance evaluation has been conducted on the PRACE supercomputing prototype (huygens) located at SARA, the Dutch National High Performance Computing and e-Science Support Centre. This prototype has large shared memory (4-8 GB/core) and fast I/O configuration with the new IBM Power6 processors and IBM Power Cluster fat node architecture. In order to find the most suitable system optimisations, both the GNU Fortran and the IBM XL Fortran compiler have been employed as shown in the last column of Table 4.1 on page 43.

The huygens system [111] has 1664 dual core processors, offering 3328 cores in total, 15.25 terabytes of main memory, and 700 terabytes of secondary memory storage. The input dataset was a 40-atom single-walled carbon nanotube model, C(10,10), with RSO Grid resolution parameters $\left(\frac{|\mathbf{Q}|_{max}-|\mathbf{Q}|_{min}}{\delta|\mathbf{Q}|}\right) = 256$, and $\frac{2\pi}{\delta\theta} = \frac{2\pi}{\delta\phi} = 200$. An initial calibration run with coarse angular resolution provided an estimated time-to-completion for the full model using (4.1).

The results of the initial calibration run in Figure 4.13 on page 52 indicate significantly improved performance with the IBM XL Fortran compiler over GNU Fortran. This difference may be attributed to the ability of the IBM XL Fortran compiler to exploit the on-chip parallelism and other architecture-specific optimisations.

The estimated sequential completion time for the full model was originally 2733 hours (114 days) with the GNU Fortran version and 552 hours (23 days) with the IBM XL Fortran version on a single processor. This initial calibration also revealed near-linear scaling up to 128 processes with a small discontinuity between 32 to 64 processes indicating moderately degraded performance at the intra-node to inter-node transition boundary.

Execution times for the model at full resolution over 32, 64, 128 and 256 processes are presented in Figure 4.14 on page 53 for a simple block partitioning scheme. This version, compiled with the IBM XL Fortran compiler, exhibits linear scaling [57] in the multi-node case, despite a memory and I/O intensive nature. We believe that the fast interconnection at Huygens and the MPI implementation have made a significant

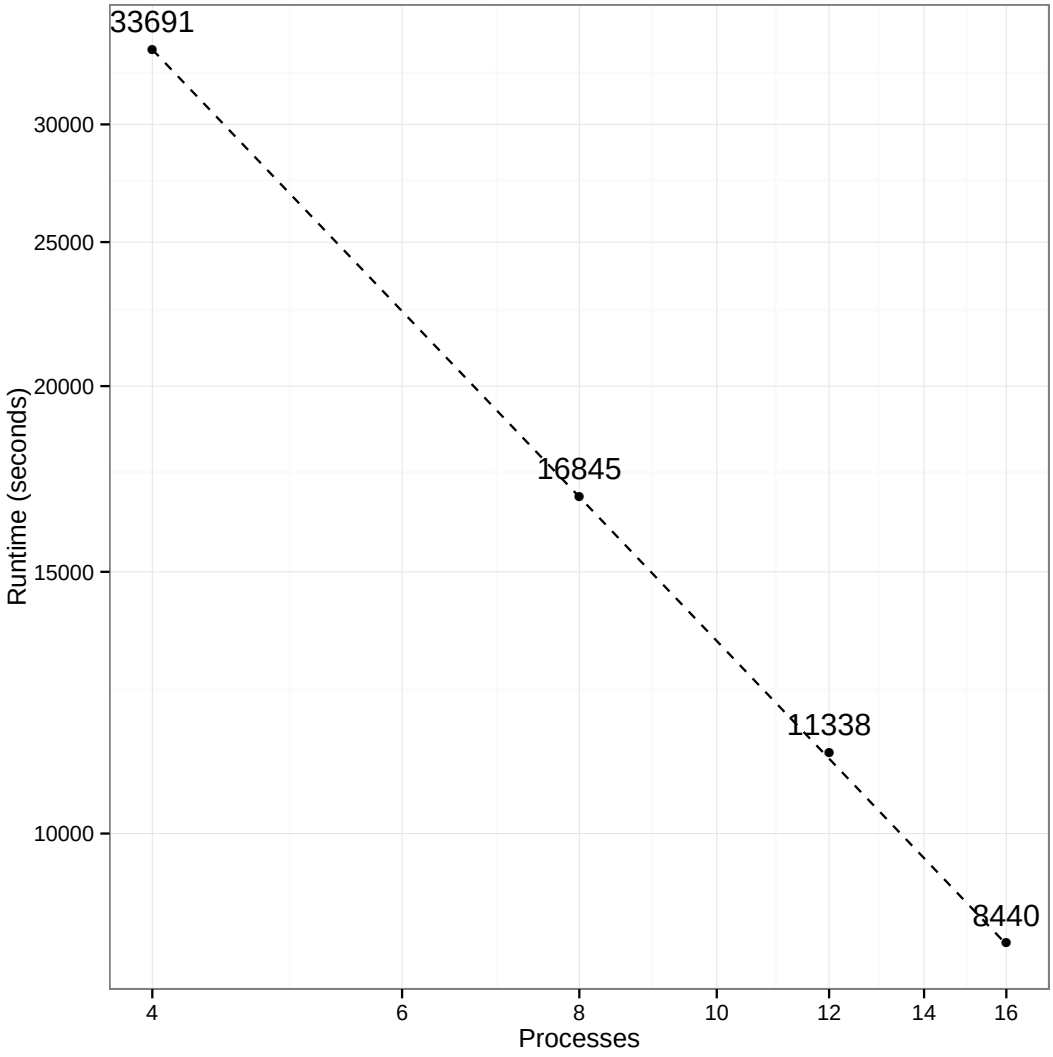


Figure 4.12: Linear scaling for the C60 model with 4, 8, 12 and 16 MPI processes on the initial IBM JS21 BladeCenter test cluster.

difference in easing the bottleneck. The final version of SCATTER, with a cyclic loop partitioning over the finer-grained iteration space exposed by the loop collapse transformation, demonstrates near-ideal scaling for up to 1024 MPI Processes (Figure 4.15 on page 54).

4.3 DISCUSSION

The high performance implementation of SCATTER presented in this chapter makes it possible to model significantly larger systems than was previously practical – a critical contribution to the computational feasibility of the new analysis method. In conjunction with the visual presentation and analysis of the large datasets generated in PREFIT and Paraview, investigations may be conducted with an expanded analysis toolbox to allow new insights into the dynamics of these systems.

Application of Mattson's design hierarchy, in conjunction with performance tools has led to a demonstrably scalable parallel implementation that is an improvement over the original ad hoc implementation. In the context of pure multicore and multinode systems, this application is an instance of the Monte Carlo or MapReduce dwarf. The scalability of this implementation is consistent with the expectations of this class of applications.

From the application perspective, the work outlined in this chapter has been a fundamental contribution to the development of a new analysis method for spectroscopic data from PolyCINS experiments from powder materials that is based on the iterative minimisation of discrepancies between experimental results and predicted simulation output to derive the appropriate force constant parameters of crystalline systems [108]. Bulk properties of solids such as the Young's modulus of elasticity, thermal expansion coefficient and various chemical properties may be predicted from these force-constant parameters and the dynamics of their lattice systems. Using the SCATTER software to perform semi-empirical simulation with initial starting parameters obtained from density functional theory (DFT) codes such as CASTEP and VASP, Roach's method has been validated on Aluminium, a well-understood reference material, and is currently being applied to the investigation of other systems.

Experience has shown that the viability of this approach rests on the ability to leverage high performance computing resources for large scale simulation, data handling and visualisation. Thus, the three fundamental tools presented have been central to the development of this new iterative workflow (Figure 4.16 on page 55).

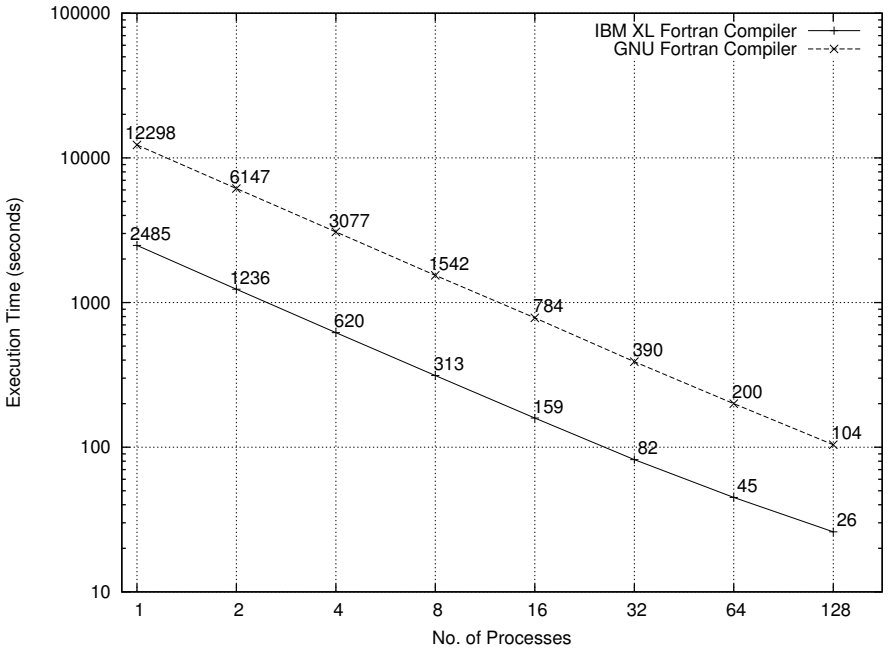


Figure 4.13: Performance of GNU Fortran and IBM XL Fortran (with architecture-specific optimisations) versions of GULP with 2^n processes a (10,10) Carbon Nanotube model in a coarse reciprocal space onion sampling for runtime estimation.

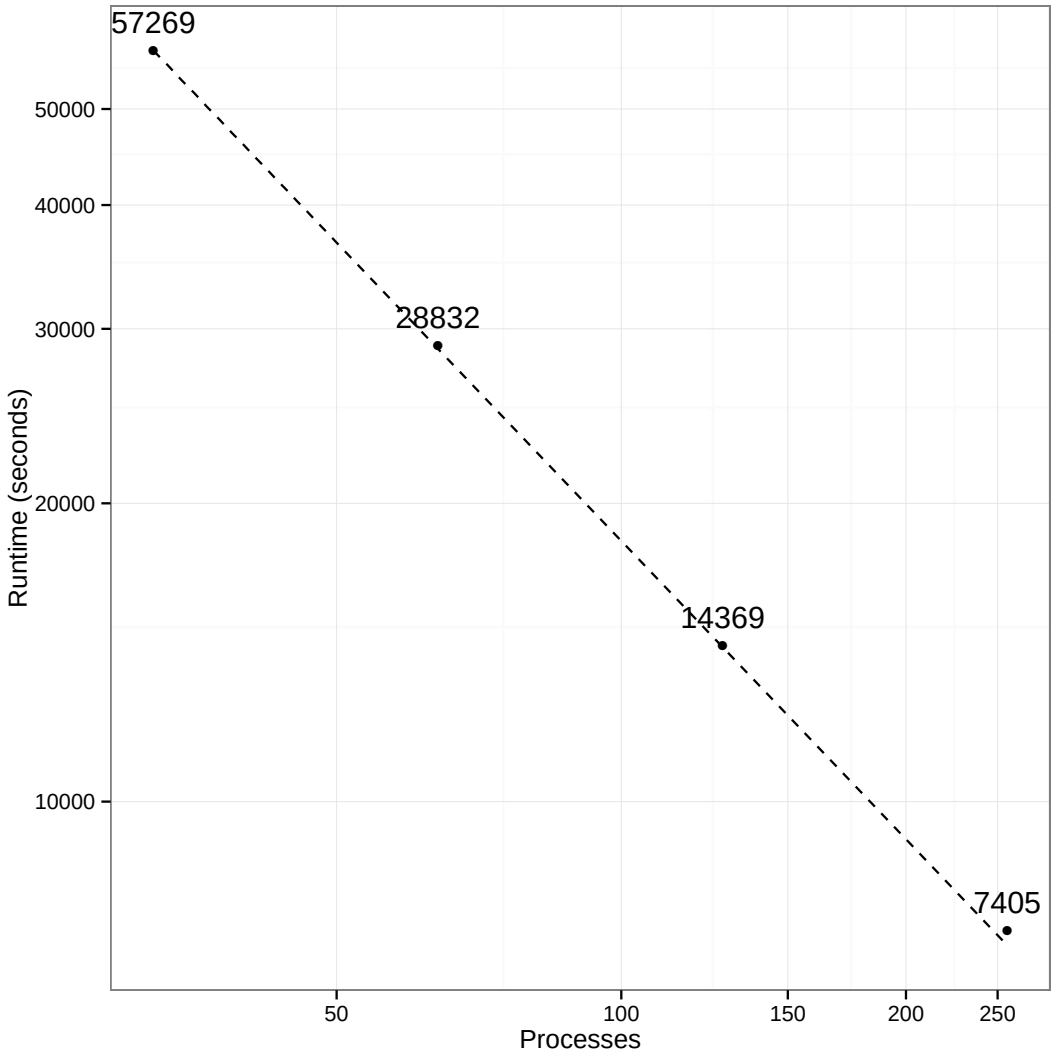


Figure 4.14: Computation times for the actual (10,10) Carbon Nanotube model with nq_step = 256 and nq_intstep = 200, generating a 120GB dataset. Results indicate near-linear scalability for 32, 64, 128 and 256 processes in the multi-node configuration. Scalability is limited by block partitioning over nq_step loop iterations.

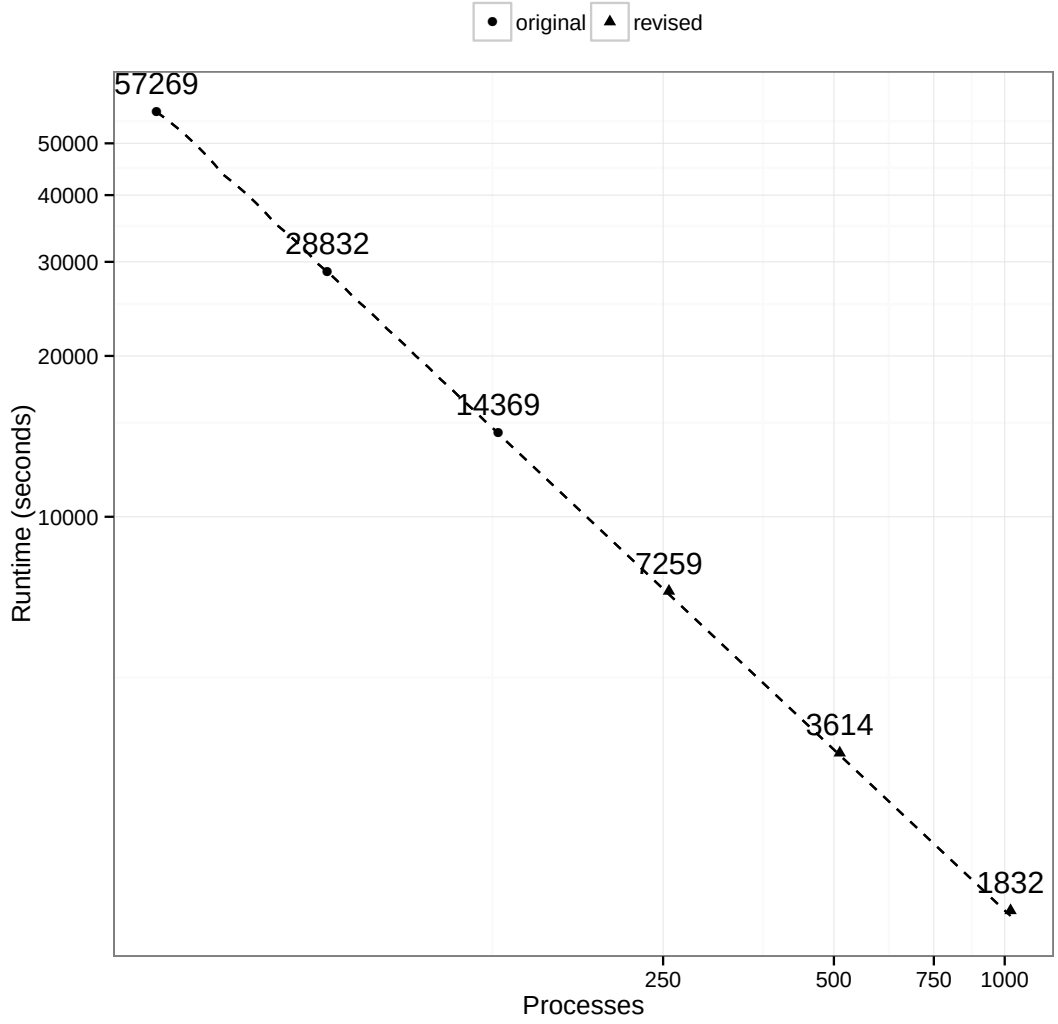


Figure 4.15: Revised computation times for the actual (10,10) Carbon Nanotube in 4.14 for the final implementation. Cyclic loop partitioning over the finer-grained iteration space exposed by the loop collapse transformation achieves moderately improved performance for 256 processes and near-linear scaling up to 1024 processes.

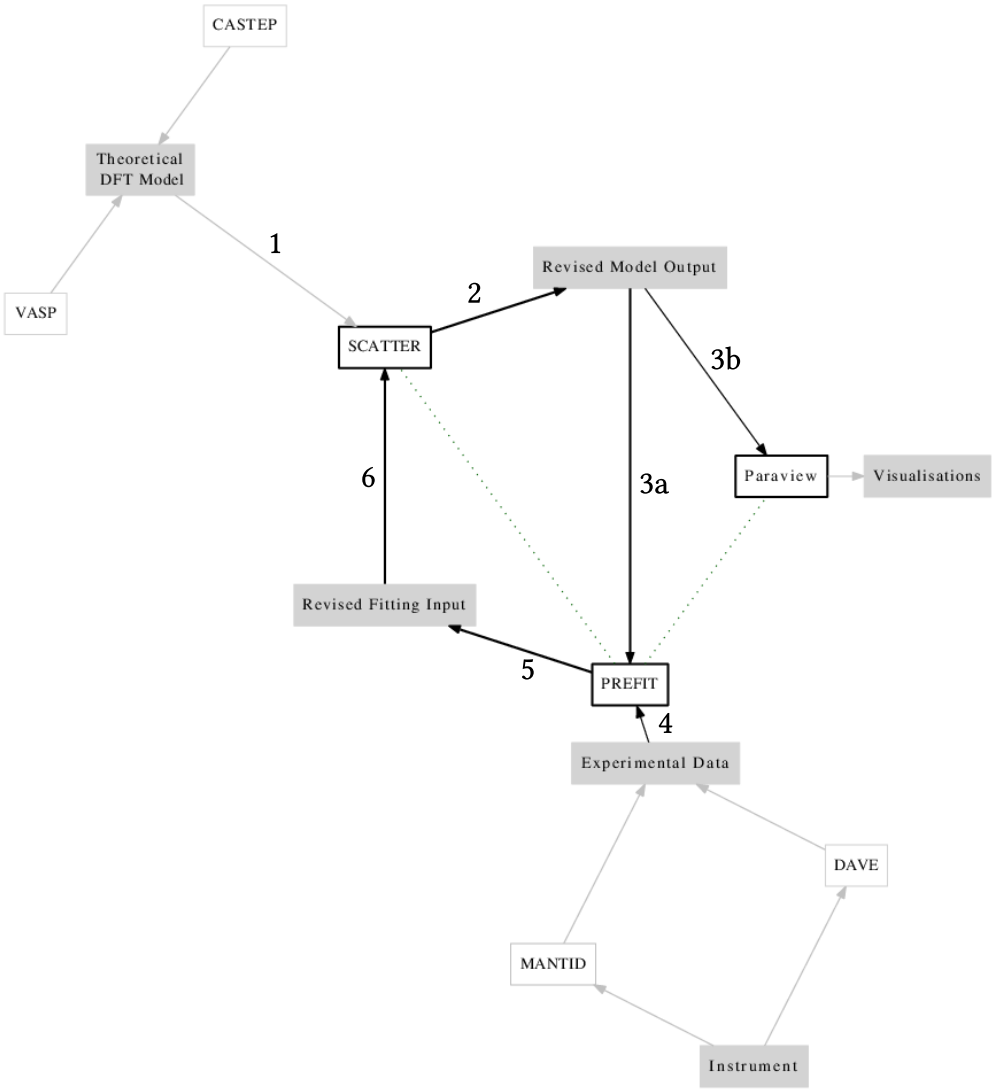


Figure 4.16: An iterative workflow for polyCINS analysis leverages high performance computing resources for large scale simulation, data handling and visualisation. The sequence of bold arrows (2-3-5-6) represents the iterative process.

5

STATIC STRUCTURAL ADAPTATION IN HETEROGENEOUS ARCHITECTURES

The previous chapter presented a scalable implementation of the Scatter code on multicore and multinode systems. For models based on the Brenner potential, the optimisations outlined reduces the computationally expensive aspects of Scatter to the determination of polarisation vectors and frequencies of the phonon modes. These are respectively the eigenvectors and eigenvalues of the Hermitian dynamical matrix for each point in reciprocal space [116]. Eispack and Lapack are used extensively within Gulp for problems of this kind. However, as GPU accelerators are naturally suited to the inherent data parallelism of dense and sparse linear algebra problems, computational accelerators present a significant opportunity to meet the computational requirements of larger crystalline systems such as the low-temperature phase of Buckminsterfullerene (C_{60}) that presents as a lattice with a 240-atom basis and 720 vibrational modes. This chapter considers the integration of support for GPU accelerators, a composition of the dense linear algebra dwarf within the Monte Carlo pattern exposed in the course of the parallel implementation, and demonstrates the need created for static adaptation in hybrid CPU/GPU codes.

5.1 GPU ACCELERATION FOR HERMITIAN EIGENSYSTEMS

In early 2011, although work on Magma [3, 123, 124], a re-implementation of Lapack for heterogeneous multicore/GPU architectures, was underway, no efficient implementations of GPU solvers for Hermitian eigensystems were widely available. Driven by the practical requirements of Scatter, the development of a new GPU solver for Hermitian eigensystems was undertaken with the required functional subset of Eispack as a basis [56]. While the challenges of achieving efficient performance on a GPU may have justified the extended effort of developing custom algorithms suited to the specific strengths of the platform [128], the original algorithms of the legacy Eispack library were retained for several reasons:

1. This work was motivated by a practical application for which the Eispack eigensolver had proven adequate.

2. As Eispack has been in production use for nearly 40 years, the numerical characteristics and accuracy have been established by exhaustive application and testing.

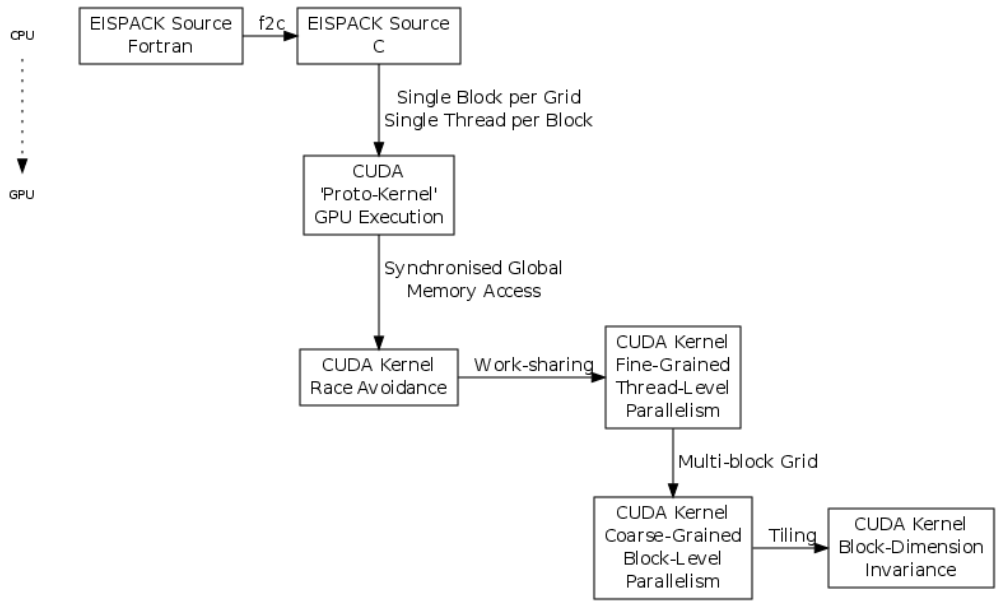


Figure 5.1: Stepwise progression from CPU to GPU implementation of Hermitian eigensystem kernels before optimisation and performance tuning

3. The problem of creating a data-parallel GPU version is conceptually similar to that of creating a vector-processor version of the EISPACK routines. A vector implementation was created for the IBM 3090-VF in 1989 [31].

4. While alternative algorithms used in LAPACK exhibit superior cache usage characteristics and performance in modern processor architectures, they provide this at the expense of software complexity and reliance on an efficient BLAS implementation [8].

Therefore, the intention was to gain parallel performance in heterogeneous CPU/GPU architectures while preserving the original high-quality algorithmic implementation of EISPACK. This was in anticipation of advances in the MAGMA project that have brought wider algorithmic coverage and steady performance improvements between releases. As of MAGMA v1.6, released in 2015, several highly tuned implementations of LAPACK eigensolver routines are available. However, given that MAGMA's exploitation of the data parallelism available in larger matrices may lead to inefficiency for smaller problems, the EISPACK implementation presented here has remained an alternative that may be preferable in some contexts. The changing performance profile of MAGMA between releases, model parameters, algorithmic variants and across environments highlights the need for adaptation.

5.1.1 *An Implementation Outline*

EISPACK provides the ch driver for double-precision Hermitian matrices as a convenient wrapper around the subroutines shown in Table 2.2. Subsequent references to EISPACK will be restricted to this subset. These subroutines serve as the basis of functionally-equivalent GPU kernels, requiring source-level translation into equivalent C code for compatibility with the C/C++-based CUDA SDK and compilation toolchain. The f2c source-to-source compiler [47] allows direct compilation of standard Fortran77 code into functionally equivalent C code with transparent handling of notable language incompatibilites such as the row-major vs. column-major array representation formats.

The implementation of all three kernels followed the sequence of stages outlined in Figure 5.1 on page 58. To maintain correctness, each stage is succeeded by testing and verification for sequential equivalence against the original program as low-level CUDA code has a tendency towards opaque errors that are difficult to locate. It may be argued that this constitutes a viable methodology for porting legacy codes to GPU platforms that manages the complexity of debugging large and complex kernels.

For each routine, an initial *proto-kernel* is implemented without any CUDA parallel constructs to test code execution on the GPU for data transfer, kernel launch and data retrieval using a single thread. This proto-kernel is highly inefficient as it makes only marginal use of the computational capabilities of the device.

With the execution model and interaction between the disjoint memory spaces of the computational processors verified, synchronisation constructs are necessary for global memory operations to avoid race conditions before the actual fine-grained work distribution is introduced. Performance gains emerge as data-parallel operations are distributed between cooperating threads. These loops are identified from source-level line-profiling on the original CPU version of EISPACK with callgrind, part of the Valgrind framework [93], the rationale being that CPU performance is strongly indicative of potential performance hotspots in the GPU kernels. This is a necessary workaround as the relatively basic CUDA profiling tools provide no information at this level of granularity.

5.1.2 *Performance Considerations*

Arguably, optimisation is currently the most challenging aspect of GPU programming and, as memory transfer constitutes the predominant limit to achievable performance, the objective is usually to maximise the *Compute to Global Memory Access* (CGMA) ratio by minimising global memory access operations and exploiting coalesced access patterns when possible. On account of the novelty and complexity of the platform, the compilation tools do not provide the same level of optimised code generation that traditional CPUs have available. As a result, this responsibility rests with the

```
__global__ void htribk_kernel(integer  *nm, integer *n,
                              doublereal *tau, integer *m,
                              doublereal *zr, doublereal *zi) {

    int blockId = (gridDim.x * blockIdx.y) + blockIdx.x;

    zr  += blockId * *nm * *n;
    zi  += blockId * *nm * *n;
    tau += blockId *   2 * *n;

    /* Parameter adjustments for 0 based indexing in C */
    tau -= 3;
    zi -= (1 + *nm);
    zr -= (1 + *nm);

        ...
}
```

Listing 5.1: Work distribution by block

programmer and a mental model of the hardware architecture of the GPU platform is necessary.

5.1.2.1 *Work Distribution Patterns*

Multiple independent blocks provide coarse-grained block-level parallelism, allowing the GPU to solve several independent eigensystems simultaneously in a single kernel invocation. While, the CUDA platform provides the ___syncthreads() primitive for thread synchronisation within a block, global kernel synchronisation across different thread blocks is unsupported.

In this implementation, a number of thread blocks independently handle the solution of multiple eigensystems in parallel. With a thread block or cooperative thread array (CTA) mapped to an input problem set, parallelism is available at both independent block and cooperative thread levels.

Inherently data-parallel operations on arrays that are identified within the algorithms are distributed between threads in a thread block to realise parallel equivalents. In these operations, tiling allows the actual dimensions of the thread block to remain independent of the dimensions of the matrix problem. Many of the work distribution operations are instances of the following recurring patterns [17]:

1. **Transformations** or mappings are operations that assign new computed values to an array based on the original values of the entries. An example is scaling of a row, in matrices ar and ai by a constant factor $scale$ in the tridiagonalisation kernel `htridi` (Listing 5.2)

$$ar_{ij} = ar_{ij} \times scale$$
$$ai_{ij} = ai_{ij} \times scale$$

As the entries are independent, this pattern is straighforward to implement efficiently with coalesced read and write global memory accesses.

2. **Reductions** such as summation operations combine multiple entries into a single value. An example is the evaluation of the value f within the htridi kernel (Listing 5.3)

$$f_i = \sum_j (e_j ar_{ij} - \tau_{2j+2} ai_{ij})$$

Accesses to global memory are only necessary to read values into a thread block and can be coalesced. For efficiency, the reduction operation is performed in shared memory that is private to that thread block.

3. **Reorderings** retrieve subsets of values from one array and assign new indices in an output array. For example, in Listing 5.4:

$$d_i = ar_{ii}$$

In general, because the indices in the source array are arbitrary functions of the output indices, coalesced access may only be possible for the output array.

4. **Composite Operations**
Many of the application-specific operations in the EISPACK routines are not simply categorised as transformations, reductions or re-orderings and require further consideration for efficiency. When possible, performance is greatly enhanced by the opportunity to interleave the simpler patterns into more complex operations that compose them.
For example, in the Householder tridiagonalisation kernel htridi, for $1 \leqslant j \leqslant l$, there is a mapping of each entry of e_j and τ_{2j} to the value of a sum of several elements in the arrays ar and ai (Listing 5.5).

$$\begin{aligned} e_j &= \frac{1}{h}\left[\sum_{k=1}^{j}(ar_{jk}ar_{ik} + ai_{jk}ai_{ik}) + \sum_{k=j+1}^{l}(ar_{kj}ar_{ik} - ai_{kj}ai_{ik})\right] \\ \tau_{2j+2} &= \frac{1}{h}\left[\sum_{k=1}^{j}(-ar_{jk}ai_{ik} + ai_{jk}ai_{ik}) - \sum_{k=j+1}^{l}(ar_{kj}ai_{ik} + ai_{kj}ar_{ik})\right] \end{aligned}$$

In terms of the patterns above, the calculation of each new entry in e_j and τ_{2j} – themselves instances of the transformation operation – may be decomposed into four separate reductions over two input arrays.

$$\begin{aligned} e_j &= \frac{1}{h}\left[\sum_{k=1}^{j} ar_{jk}ar_{ik} + \sum_{k=1}^{j} ai_{jk}ai_{ik} + \sum_{k=j+1}^{l} ar_{kj}ar_{ik} - \sum_{k=j+1}^{l} ai_{kj}ai_{ik}\right] \\ \tau_{2j+2} &= \frac{1}{h}\left[-\sum_{k=1}^{j} ar_{jk}ai_{ik} + \sum_{k=1}^{j} ai_{jk}ar_{ik} - \sum_{k=j+1}^{l} ar_{kj}ai_{ik} - \sum_{k=j+1}^{l} ai_{kj}ar_{ik}\right] \end{aligned}$$

However, an alternative decomposition is possible that eliminates redundant memory operations[1].

$$
\begin{aligned}
e_j &= \frac{1}{h}\left[\sum_{k=1}^{j}(ar_{jk}ar_{ik}+ai_{jk}ai_{ik})+\sum_{k=j+1}^{l}(ar_{kj}ar_{ik}-ai_{kj}ai_{ik})\right] \\
&= \frac{1}{h}\sum_{k=1}^{l}(ar_{jk}ar_{ik}+ai_{jk}ai_{ik})[k\leqslant j]+\frac{1}{h}\sum_{k=1}^{l}(ar_{kj}ar_{ik}-ai_{kj}ai_{ik})[k>j] \\
&= \frac{1}{h}\sum_{k=1}^{l}(ar_{ik}(ar_{jk}[k\leqslant j]+ar_{kj}[k>j])+ai_{ik}(ai_{jk}[k\leqslant j]-ai_{kj}[k>j])) \\
\tau_{2j+2} &= \frac{1}{h}\left[\sum_{k=1}^{j}(-ar_{jk}ai_{ik}+ai_{jk}ar_{ik})-\sum_{k=j+1}^{l}(ar_{kj}ai_{ik}+ai_{kj}ar_{ik})\right] \\
&= \frac{1}{h}\sum_{k=1}^{l}(-ar_{jk}ai_{ik}+ai_{jk}ar_{ik})[k\leqslant j]-\frac{1}{h}\sum_{k=1}^{l}(ar_{kj}ai_{ik}+ai_{kj}ar_{ik})[k>j] \\
&= \frac{1}{h}\sum_{k=1}^{l}(-ai_{ik}(ar_{jk}[k\leqslant j]+ar_{kj}[k>j])+ar_{ik}(ai_{jk}[k\leqslant j]-ai_{kj}[k>j]))
\end{aligned}
$$

Defining the conditional terms in both expression as new variables,

$$
\begin{aligned}
arjk &= ar_{jk}[k\leqslant j]+ar_{kj}[k>j] \\
aijk &= ai_{jk}[k\leqslant j]-ai_{kj}[k>j]
\end{aligned}
$$

The following simplified expressions for e_j and τ_{2j} are obtained

$$
\begin{aligned}
e_j &= \frac{1}{h}\sum_{k=1}^{l}(ar_{ik}arjk+ai_{ik}aijk) \\
\tau_{2j+2} &= \frac{1}{h}\sum_{k=1}^{l}(-ai_{ik}arjk+ar_{ik}aijk)
\end{aligned}
$$

The original expressions may have been realised as two separate mapping operations, each requiring eight separate reduction operations per element with two global memory read operations per reduction term. The combined kernel of 5.6 takes advantage of the overlap between argument values to evaluate the expressions using only four global memory read operations in a single reduction nested within a single map operation. Further avoided is the associated overhead and synchronisation that would have been incurred by multiple summations.

5.1.2.2 *Structural Alternatives*

While the CUDA platform documentation provides a guide to performance best practices [97], trade-offs remain necessary between possible efficiency measures. Limited

1 We use Iverson's notation [p] to denote a term that evaluates to 1 if the condition p is true and 0 otherwise.

Listing 5.2: A row scaling transformation of matrices `ar` and `ai` by a constant factor `scale` in the tridiagonalisation kernel `htridi`. Independent entries allow efficient implementation with coalesced read and write global memory accesses

```
/* Serial Transformation */
for (k = 1; k <= l; ++k) {
  ar[i + k * *nm] = scale * ar[i + k * *nm];
  ai[i + k * *nm] = scale * ai[i + k * *nm];
}

/* Kernel Operation */
k = threadId + 1;
while (k <= l){
  ar[k + i * *nm] *= scale;
  ai[k + i * *nm] *= scale;
  k += blockDim.x*blockDim.y;
}
```

Listing 5.3: A reduction operation in the the htridi kernel evaluates the value `f` as the sum of values from multiple arrays.

```
/* Serial Reduction */
for (j = 1; j <= i__2; ++j) {
  ...
  f = f + e[j] * ar[i + j * *nm]
      - tau[(j << 1) + 2] * ai[i + j * *nm];
}

/* Kernel Operation */
int threadId = (blockDim.x * threadIdx.y) + threadIdx.x;
j = threadId + 1;
memblock[threadId] = 0;
while (j <= l){
  memblock[threadId] += e[j] * ar[j + i__ * *nm]
           - tau[(j << 1) + 2] * ai[j + i__ * *nm];
  j += blockDim.x*blockDim.y;
}
for (integer stride = 1;
       stride < l && stride < blockDim.x*blockDim.y; stride *= 2){
  __syncthreads();
  if (threadId % (2*stride) == 0  && (threadId + stride) < l
       && (threadId + stride) < blockDim.x*blockDim.y){
    memblock[threadId] +=  memblock[threadId + stride];
  }
}
```

Listing 5.4: A reordering retrieves subsets of values from one array and assigns new indices in an output array. Source indices are arbitrary functions of the output indices and coalesced access may only be possible for the output array.

```
/* A Selection */
for (i = 1; i <= i__1; ++i) {
        d[i] = ar[i + i * *nm];
}

/* Kernel Operation */
k = (blockDim.x * threadIdx.y) + threadIdx.x + 1;
  while (k <= *n) {
    d[k] = ar[k + k * *nm];
    k += blockDim.x*blockDim.y;
}
```

```
for (j = 1; j <= l; ++j) {
  g = 0.;
  gi = 0.;

  /*     .......... form element of a*u .......... */
  for (k = 1; k <= j; ++k) {
     g =  g + ar[j + k * *nm] * ar[i + k * *nm]
            + ai[j + k * *nm] * ai[i + k * *nm];
    gi = gi - ar[j + k * *nm] * ai[i + k * *nm]
            + ai[j + k * *nm] * ar[i + k * *nm];
  }

  jp1 = j + 1;

  if (l < jp1) {
    goto L220;
  }

  j = l;

  for (k = jp1; k <= j; ++k) {
     g =  g + ar[k + j * *nm] * ar[i + k * *nm]
            - ai[k + j * *nm] * ai[i + k * *nm];
    gi = gi - ar[k + j * *nm] * ai[i + k * *nm]
            - ai[k + j * *nm] * ar[i + k * *nm];
  }

  /*   .......... form element of p .......... */
L220:
  e[j] = g / h;
  tau[(j << 1) + 2] = gi / h;
  f = f + e[j] * ar[i + j * *nm] - tau[(j << 1) + 2] * ai[i + j * *nm];
}
```

Listing 5.5: A mapping of each entry of e_j and τ_{2j} to the value of a sum of several elements in the arrays `ar` and `ai` in the Householder tridiagonalisation kernel `htridi`.

```
 g_sh = (doublereal*)&memblock[blockDim.x * threadIdx.y];
gi_sh = (doublereal*)&g_sh[blockDim.x * blockDim.y];
...
/*     .......... form element of a*u .......... */

doublereal arjk, arik, aijk, aiik;
j = threadIdx.y + 1;

while (j-threadIdx.y <= l){

   g_sh[threadIdx.x] = 0;
  gi_sh[threadIdx.x] = 0;

  arjk = arik = aijk = aiik = 0;

  k = threadIdx.x + 1;

  while (j <= l && k <= l){
    if (k > j){
      arjk = ar[j + k * *nm];
      aijk = -ai[j + k * *nm];
    } else {
      arjk = ar[k + j * *nm];
      aijk = ai[k + j * *nm];
    }

    arik = ar[k + i__ * *nm];
    aiik = ai[k + i__ * *nm];

     g_sh[threadIdx.x] +=  arjk * arik + aijk * aiik;
    gi_sh[threadIdx.x] += -arjk * aiik + aijk * arik;

    k += blockDim.x;
  }

  for (integer stride = blockDim.x>>1; stride > 0; stride >>=1 ){
    __syncthreads();
    if (threadIdx.x < stride){
       g_sh[threadIdx.x] +=  g_sh[threadIdx.x + stride];
      gi_sh[threadIdx.x] += gi_sh[threadIdx.x + stride];
    }
  }

  __syncthreads();

   g  = g_sh[0];
  gi = gi_sh[0];

/*     .......... form element of p .......... */
  if (j <= l){
    e[j] = g / h__;
    tau[(j << 1) + 2] = gi / h__;
  }

  j += blockDim.y;
}
```

Listing 5.6: A composite operation performs mapping of each entry of e_j and τ_{2j} to the value of a sum of several elements in the arrays `ar` and `ai` in the Householder tridiagonalisation kernel `htridi`, combining transformation, reordering and reduction into a single kernel eliminates redundant memory accesses and the overhead of multiple summation reductions.

exploration of the space of possible parametric optimisations, such as optimal block dimensions and tile size, was facilitated by the code generation tools in PyCUDA [79], the high-level Python language bindings to the CUDA API.

However, using the **ANY** abstraction, choices between alternatives that arise in the course of optimisation may be represented structurally. The original Hermitian eigensolver in the EISPACK library may represented as

$$\mathsf{E} \quad \rightarrow \quad \mathbf{SEQ}(\mathtt{htridi}, \mathtt{tql2}, \mathtt{htribk})$$

and the direct implementation of the kernels corresponding to the relevant EISPACK subroutines may be represented as the transformation

$$\mathbf{SEQ}(\mathtt{htridi}, \mathtt{tql2}, \mathtt{htribk}) \quad \rightarrow \quad \mathbf{SEQ}(\mathtt{htridi}', \mathtt{tql2}', \mathtt{htribk}')$$

where the notation A' represents a functional equivalent of A. Each functionally equivalent reimplementation creates a new variant. All combinations of possible eigensolvers using these functions may be represented as

$$\mathsf{E} \quad \rightarrow \quad \mathbf{SEQ}(\mathbf{ANY}(\mathtt{htridi}, \mathtt{htridi}'), \mathbf{ANY}(\mathtt{tql2}, \mathtt{tql2}'), \mathbf{ANY}(\mathtt{htribk}, \mathtt{htribk}'))$$

However, further possible optimisations exist. In all cases, for these optimisations to be valid, the combined costs of the split kernels and associated multiple kernel launch overheads must be less than the performance gained from transposed memory access and reduced register usage.

1. For coalesced memory access in the htridi′ kernel, array accesses may be transposed. In this case, the negation of the imaginary matrix is necessary to represent the operation of complex conjugation. Thus, we introduce a kernel that performs matrix negation:

$$\mathtt{htridi}' \quad \rightarrow \quad \mathbf{SEQ}(\mathit{negate}, \mathtt{htridi}'')$$

2. The tql2′ kernel concludes with an ordering of eigenvectors by eigenvalue. Separation of the ordering step into a smaller kernel that may execute with large thread blocks due to lower register usage is a possible optimisation.

$$\mathtt{tql2}' \quad \rightarrow \quad \mathbf{SEQ}(\mathtt{tql2}'', \mathtt{order})$$

3. The htribk′ kernel may be decomposed into two smaller kernels

$$\mathtt{htribk}' \quad \rightarrow \quad \mathbf{SEQ}(\mathtt{htribkA}, \mathtt{htribkB})$$

with the new htribkB kernel operating on pitched memory. This must be preceeded and succeeded by a matrix transposition for coalesced access without altering correctness

$$\text{htribkB} \rightarrow \textbf{SEQ}(\text{transpose}, \text{htribkB}', \text{transpose})$$

Combining all of these possible optimisations considered,

$$\begin{aligned}
E_E \rightarrow\ & \textbf{SEQ}(\\
& \textbf{ANY}(\text{htridi}, \text{htridi}', \\
& \ \textbf{SEQ}(\text{negate}, \text{htridi}'')), \\
& \textbf{ANY}(\text{tql2}, \text{tql2}', \\
& \ \textbf{SEQ}(\text{tql2}'', \text{order})), \\
& \textbf{ANY}(\text{htribk}, \text{htribk}', \\
& \ \textbf{SEQ}(\text{htribkA}, \\
& \ \ \textbf{ANY}(\text{htribkB}, \\
& \ \ \ \textbf{SEQ}(\text{transpose}, \text{htribkB}', \text{transpose})))))
\end{aligned}$$

5.2 STRUCTURAL APPLICATION VARIANTS

5.2.1 *Hermitian Eigensystem Variants*

The space of possible Hermitian eigensystem solver components of SCATTER incorporates routines from different library options.

$$E \rightarrow \textbf{ANY}(E_E, E_L, E_M)$$

E combines E_E, the Hermitian eigensolvers described above, E_L, the CPU-only routines available in the LAPACK library and E_M, the recent GPU re-implementation in the MAGMA library of LAPACK functions.

Here, the equivalent functions in LAPACK and MAGMA are

$$E_L \rightarrow \textbf{ANY}(\text{lapack\_zheev}, \text{lapack\_zheevd}, \text{lapack\_zheevr}, \text{lapack\_zheevx})$$

$$\begin{aligned}
E_M \rightarrow\ & \textbf{ANY}(\text{magma_zheevd}, \text{magma_zheevd_gpu}, \text{magma_zheevd_m}, \\
& \text{magma_zheevdx}, \text{magma_zheevdx_2stage}, \text{magma_zheevdx_2stage_m}, \\
& \text{magma_zheevdx_gpu}, \text{magma_zheevdx_m}, \\
& \text{zheevr}, \text{zheevr_gpu}, \text{zheevx}, \text{zheevx_gpu})
\end{aligned}$$

Figure 5.2 on page 69 is a graphical representation of E. Any path from the left terminal node to the right terminal node, constitutes a valid eigensolver. Although, the

```
/* Check if matrix is very small then just call LAPACK on CPU,
   no need for GPU */
if (n <= 128) {
  #ifdef ENABLE_DEBUG
  printf("-------------------------------------------------------------\n");
  printf("warning matrix too small N=%d NB=%d, calling lapack on CPU\n", (int) n, (int) nb
      );
  printf("-------------------------------------------------------------\n");
  #endif
  lapackf77_zheevr(jobz_, range_, uplo_,
      &n, a, &lda, &vl, &vu, &il, &iu, &abstol, m,
      w, z, &ldz, isuppz, work, &lwork,
      rwork, &lrwork, iwork, &liwork, info);
  return *info;
}
```

Listing 5.7: Implicit variant selection in MAGMA. Assumptions about the relative performance of LAPACK and MAGMA are not valid on all platforms .

routines of LAPACK and MAGMA appear trivial in this representation, implicit choices have already been made in these libraries that are not exposed. Listing 5.7 is an example of implicit variant selection in the MAGMA library that is performed on an ad hoc basis. The zheevr function defaults to CPU execution for matrix sizes under 128, an assumption that is not necessarily valid for all platforms.

5.2.2 SCATTER *Variants*

The SCATTER program may be represented as

$$\mathbf{FOR}(\mathbf{SEQ}(P, E, C), n)$$

where P represents the production of a new dynamical matrix, E the calculation of eigenvectors and eigenvalues and C the calculation of scattering contributions.

Similarly, the pattern-based parallel implementation presented in Chapter 4 may be represented as the transformation

$$\mathbf{FOR}(\mathbf{SEQ}(P, E, C), n) \quad \rightarrow \quad \mathbf{FARM}(\mathbf{SEQ}(P, E, C))$$

Multiple alternative implementations of the Hermitian eigensolver E now exist for execution on both CPU and GPU resources. Therefore, we may consider direct integration of the LAPACK and MAGMA libraries. However, the GPU implementation of EISPACK is a special case that requires batched inputs to amortise the cost of memory transfer operations and minimise idling on the device.

$$\begin{aligned} \mathbf{FOR}(\mathbf{SEQ}(P, E, C), n) \quad &\rightarrow \quad \mathbf{FARM}(\mathbf{SEQ}(P, E, C)) \\ &\rightarrow \quad \mathbf{FARM}(\mathbf{SEQ}(P, \mathbf{ANY}(E_L, E_M), C)) \end{aligned}$$

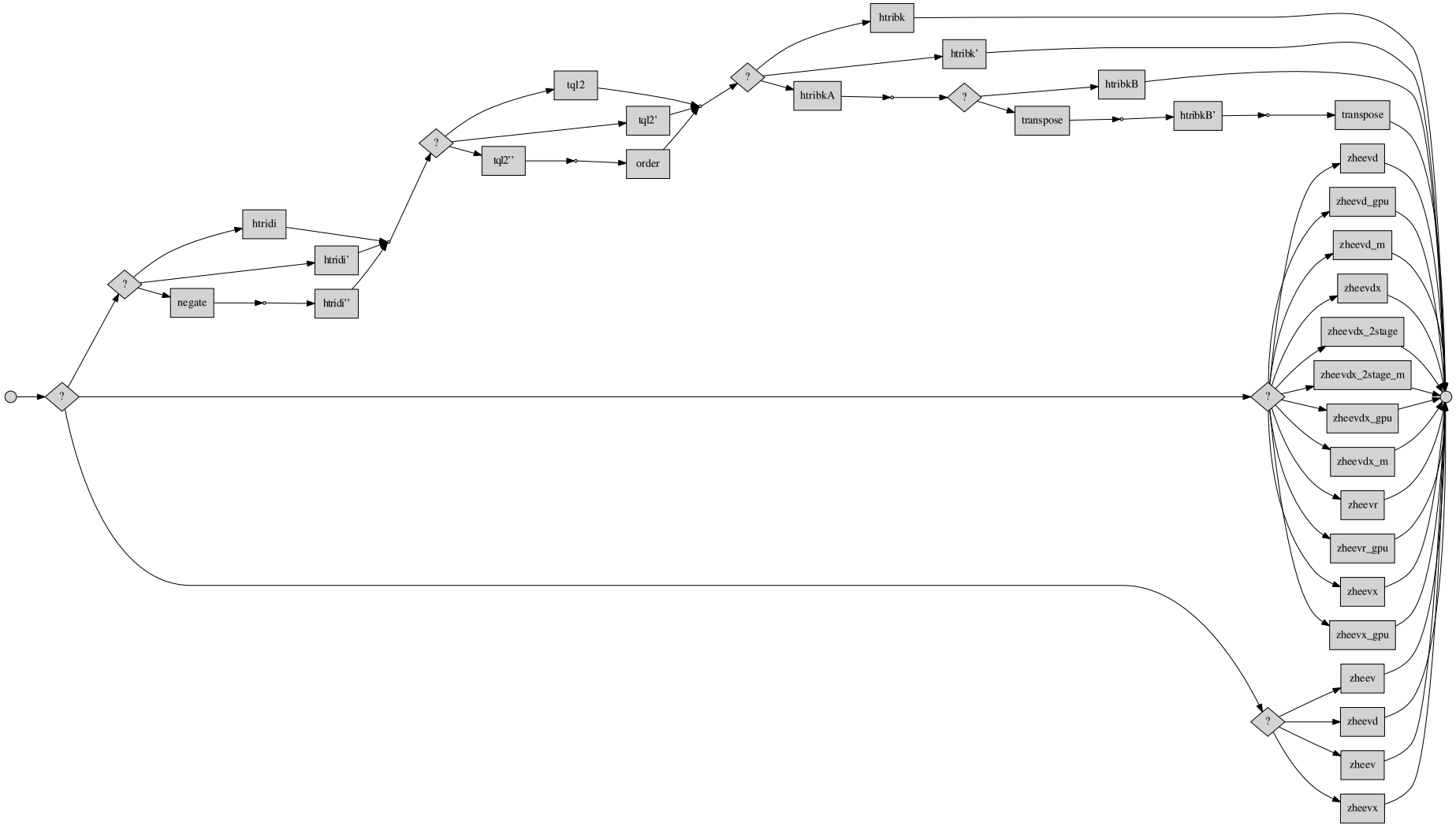


Figure 5.2: Possible eigensolver variants including EISPACK, EISPACK$_{gpu}$, LAPACK and MAGMA alternatives described with **ANY**. Any directed path from the left terminal to the right terminal constitutes a valid eigensolver.

The pipeline structural form, with intermediate queues, allows a computational stage to collect b queue entries into a batch, decoupling the size of the farm from the number of simultaneous eigensystems being solved on the GPU. Therefore, we consider the alternate pipeline structural form, with nested farm stages, and elect to include the MAGMA library as the effect of this decoupling may conceivably confer a performance gain from increased utilisation.

$$\begin{aligned}\textbf{FOR}(\textbf{SEQ}(P, E, C), n) &\rightarrow \textbf{PIPE}(P, E, C)\\ &\rightarrow \textbf{PIPE}(P, \textbf{ANY}(E_E, E_M), C)\\ &\rightarrow \textbf{PIPE}(\textbf{FARM}(P), \textbf{FARM}(\textbf{ANY}(E_E, E_M)), \textbf{FARM}(C))\end{aligned}$$

For this application, with a decomposition based on native processes, the NoSQL datastore Redis [110, 22, 23, 67], noted for performance and extensive language bindings, provides an out-of-process shared instrumented queue that supports distributed access and persistence. The queues form the mechanism of composition across heterogeneous resources, decoupling the concurrently executing processes on the CPU host from the concurrently executing kernels on the GPU streaming multiprocessors.

The final program specification combines all of these structural alternatives

$$\begin{aligned}\textbf{FOR}(\textbf{SEQ}(P, E, C), n) \rightarrow \textbf{ANY}(&\\ &\textbf{FARM}(\textbf{SEQ}(P, \textbf{ANY}(E_L, E_M), C)),\\ &\textbf{PIPE}(\textbf{FARM}(P), \textbf{FARM}(\textbf{ANY}(E_E, E_M)), \textbf{FARM}(C)))\end{aligned}$$

5.2.3 *Variant Selection*

Variant selection is driven by an adaptive launcher program that takes the SCATTER input file, containing relevant model parameters, and a structured graph as arguments.

In the structured graph specification, all individual **SEQ** nodes are associated with executable binaries that interface to the queue implementation via the Redis client library. A corollary of this architectural style and the capabilities of Redis is that distributed check-pointing and fault-tolerance become realisable. In exhaustive calibration mode, feasible variants, represented as directed paths through the structured graph encoded with **ANY**, are evaluated on a coarse-resolution grid to gather performance metrics. Nodes in the graph that are unassociated with an executable file, render the variant associated with that path unfeasible and are ignored. This allows the incremental introduction of new variants paths as they become available. Subsequently, the full model is executed with the optimal selected variant.

To avoid redundant re-calibration, performance metrics are memoised as historic profiling information. These look-up tables are keyed by variant and relevant model parameters parsed from the SCATTER input file that include lattice parameters and

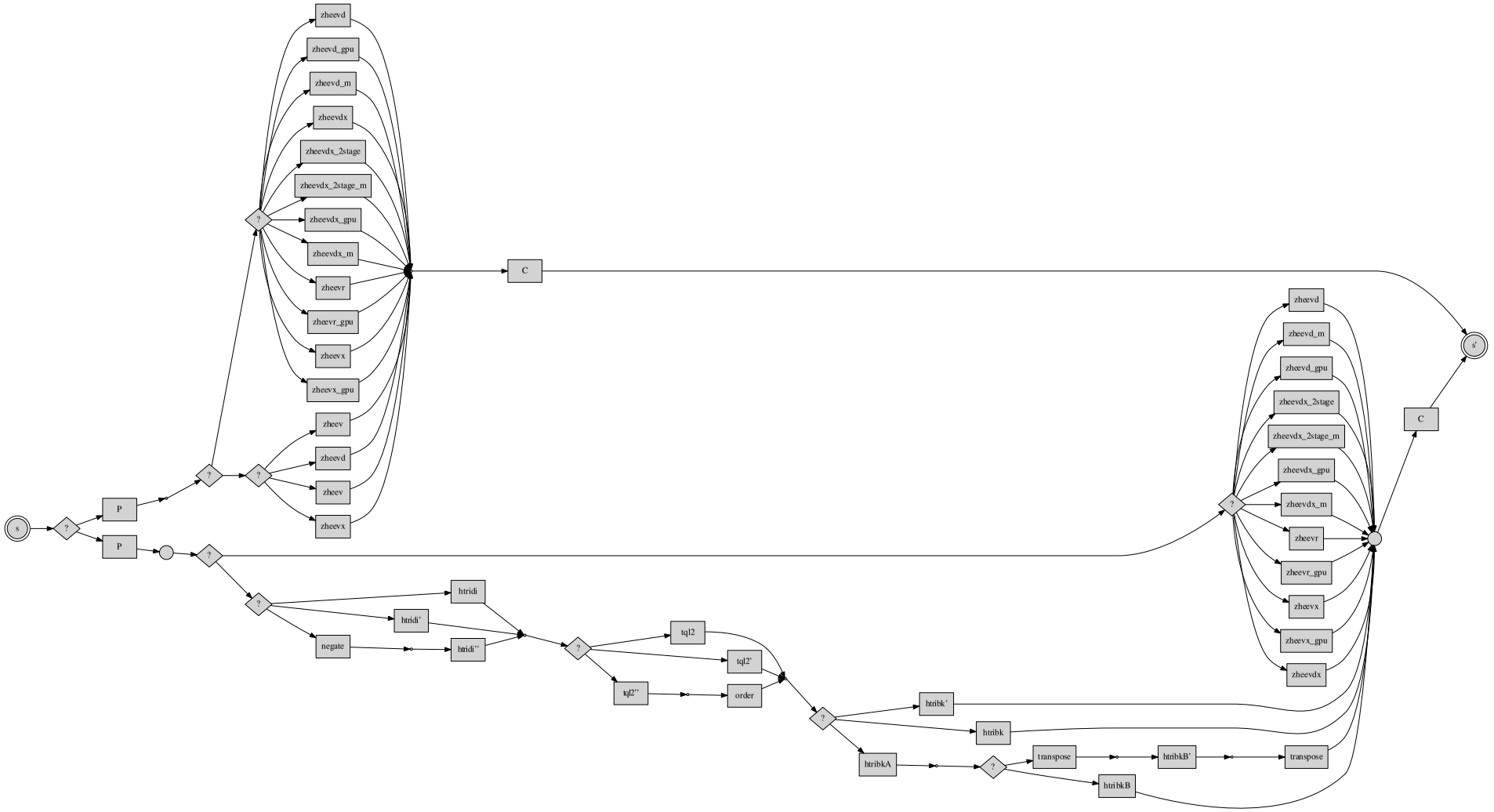


Figure 5.3: SCATTER variants combine eigensolvers and alternative application structures **SEQ**(P, E, C) and **PIPE**(P, E, C). The queues in **PIPE**(P, E, C) allow batched execution to minimise data transfer overhead.

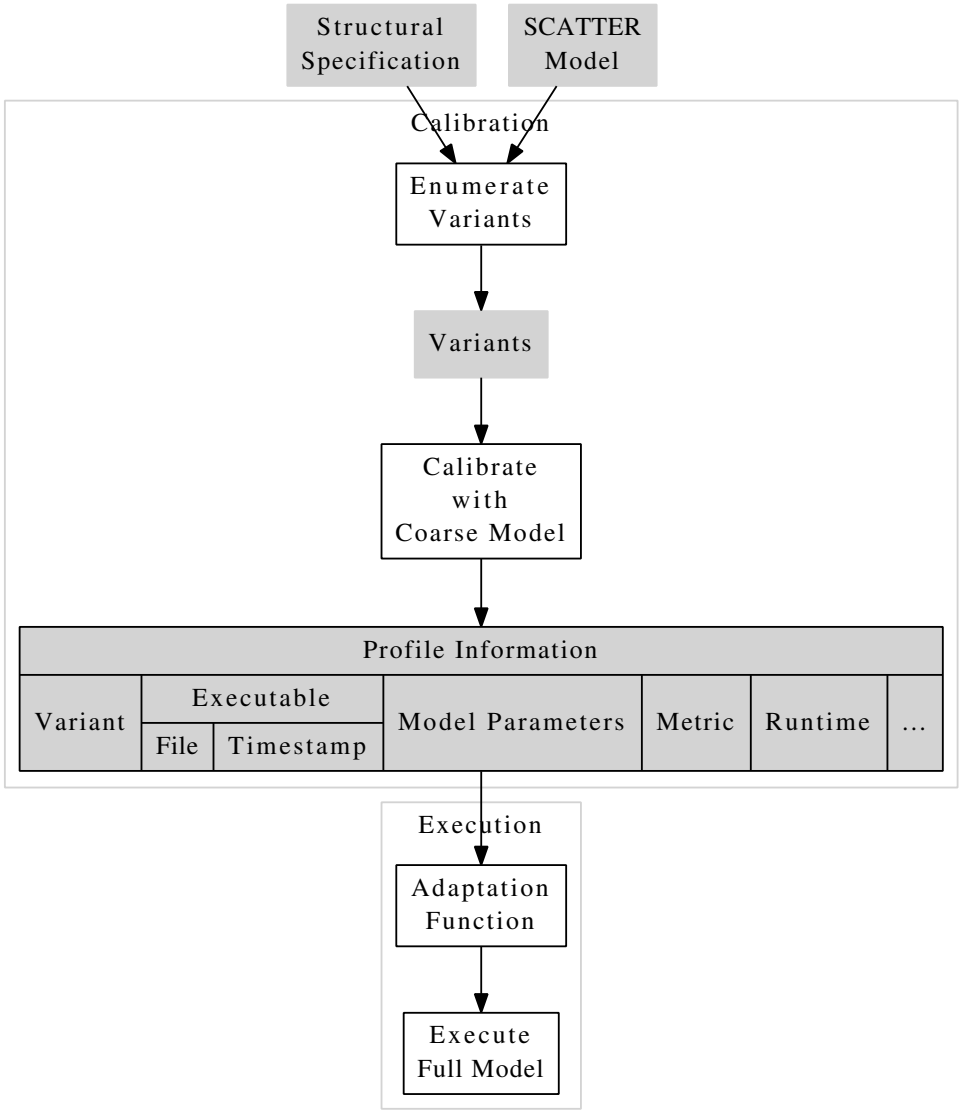


Figure 5.4: Variant selection is driven by a launcher script that takes the SCATTER input file, containing relevant model parameters, and a structured graph as inputs.

potential functions. Selective re-calibration is performed for new or updated program binaries associated with graph nodes as they become available.

5.3 EVALUATION

5.3.1 *Eigensolver Performance*

Performance evaluations have been carried out using a 64-bit Dell Precision T7500 Server Host machine (Table 5.2) with 4 Intel Xeon 2GHz CPU cores, 4GB RAM and a NVIDIA Tesla C2050 GPU connected via a PCI express interface running Version 3.2

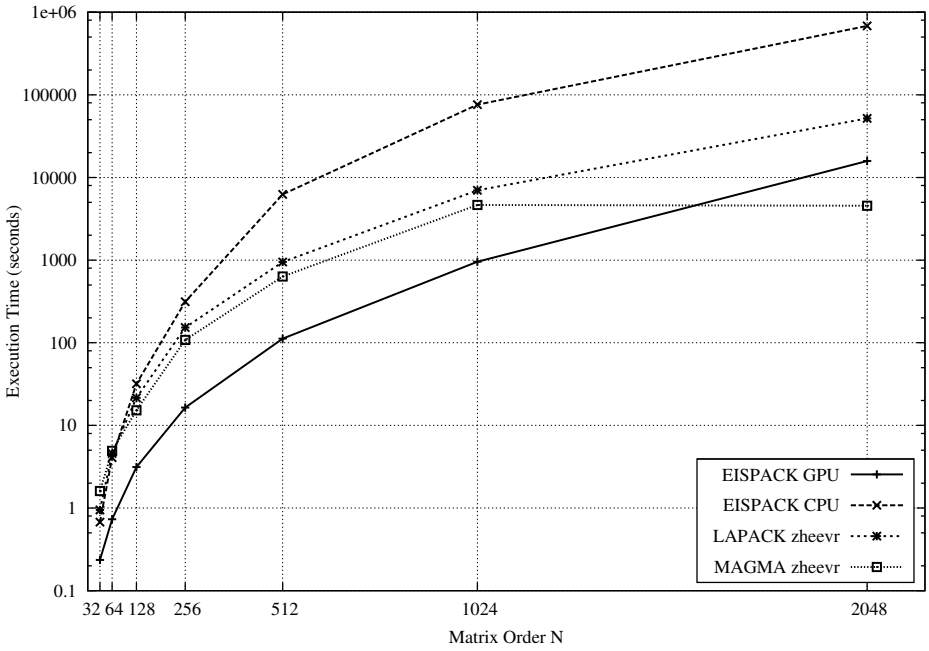


Figure 5.5: Execution times for 1000 double precision Hermitian matrices of order N (32, 64, 128, 256, 1024, 2048) for four different implementations: (i) the EISPACK GPU implementation; (ii) the reference EISPACK CPU implementation; (iii) the LAPACK zheevr eigensolver; and, (iv) the MAGMA zheevr solver using one NVIDIA Tesla C2050 GPU.

of the CUDA SDK on 64-bit Ubuntu 10.04 Linux. Given that the second-generation NVIDIA Tesla C2050 GPU is designed specifically for scientific and numerical computing applications, it offers 14 streaming multiprocessors (SM), each providing 32 streaming processors (SP), for a total count of 448 parallel cores. While earlier GPUs completely lacked double precision support, the Tesla GPU provides improved double-precision floating point performance.

Figure 5.5 on page 73 compares runtimes for 1000 N-order input matrices with the reference EISPACK implementation, the zheevr routines from LAPACK and MAGMA and our EISPACK_{gpu} implementation. GPU times are collected via the platform timers and are inclusive of memory transfer overhead. Notably, the EISPACK_{gpu} implementation benefits from batched execution.

Within a critical window (N = 512–2048), the current GPU implementation is capable of yielding performance increases of between 50–100× over the reference EISPACK implementation, a result of performance gains at both thread and block levels. As the matrix order increases, the GPU memory is able to accommodate fewer matrices to provide any block-level performance advantage and resource idling increases. Therefore, the scalability of the approach is restricted for higher values of N by the hard limit that memory places on GPU occupancy despite the still-observable benefits of thread-level parallelism.

The superior LAPACK cache behaviour delivers consistently higher performance over EISPACK for larger values of N. While equivalent routines in both LAPACK and EISPACK

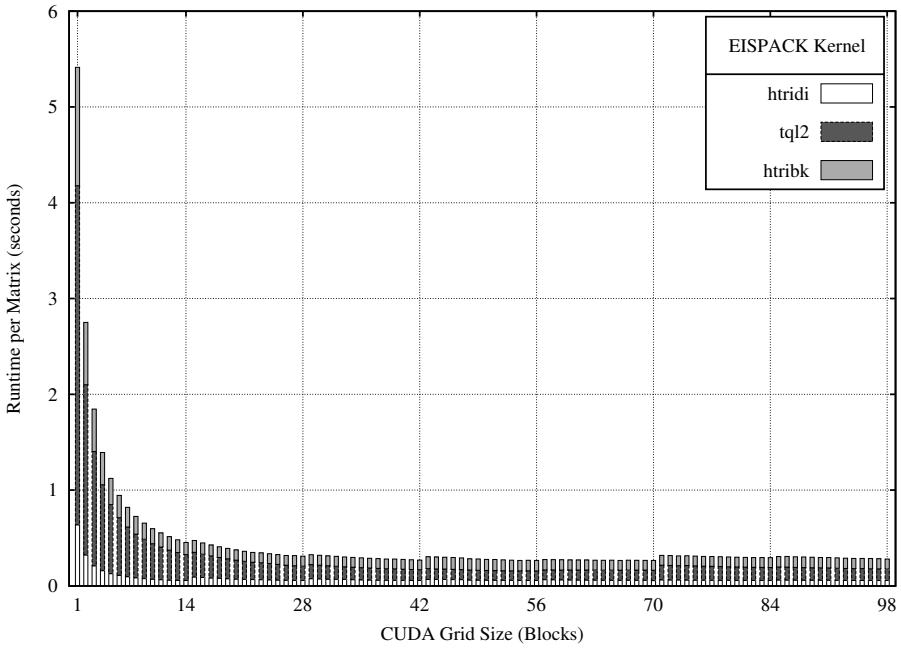


Figure 5.6: Individual and total Execution time per 720-order matrix at various CUDA grid sizes for htridi, tql2, and htribk, the main three EISPACK kernels from the ch driver (as described in Table 2.2). Each block in the grid maps to a single eigensystem.

are of storage order $O(n^2)$, LAPACK reuses the same input matrix memory for output and is therefore more memory efficient.

At smaller problem sizes, our GPU implementation delivers appreciable speedups even when the overhead of using the MAGMA libraries results in a performance slowdown. Within a critical window ($N = 512 - 2048$), the current GPU routine yields performance increases of between 50 to 90 times over the baseline EISPACK implementation, and one order of magnitude over LAPACK, a result of performance gains at both thread and block levels. As the matrix order increases, the GPU memory is able to accommodate fewer matrices to provide any block-level performance advantage and execution resources begin to idle. Therefore, the scalability of the approach is restricted for higher values of N by the hard limit that memory places on GPU occupancy despite the still-observable benefits of thread-level parallelism. However, MAGMA performance exhibits a distinct improvement at this point as it is well suited to larger problems.

The extent to which block-level parallelism is important in our implementation is demonstrated in Figure 5.6 on page 74, allowing comparison of the effect of different grid sizes on the average completion time of each of the main kernels in a 720-order matrix eigensystem. Low grid sizes with few simultaneous input matrices provide little block-level parallelism and make poor use of the available hardware resources. Local minima at block sizes that are integral multiples of 14 deliver the best performance, a reflection of the 14 SMs in the C2050.

5.3.2 SCATTER *Variants*

Four structural variants (Table 5.3) of SCATTER with the **SEQ** and **PIPE** forms integrating the EISPACK$_{gpu}$ eigensolvers and the LAPACK and MAGMA equivalents zheevr were evaluated for execution time with different test models on the Dell Precision Server and Xookik. In addition, energy requirements were collected for the Dell Precision Server using a Prodigit 2000MU Plug-in Power and Energy Monitor. Two possible variants are excluded from this examination:

1. **SEQ**(P, E_G, C), the sequential structural variant integrating the EISPACK$_{gpu}$ solver, requires batched kernels for efficiency.

2. **PIPE**(P, E_L, C), the pipeline structural variant of the LAPACK solver would be subjected to overhead that is difficult to justify given that no use of heterogeneous resources would be made. This leads to consistently worse performance than the sequential variant **SEQ**(P, E_L, C).

Test Model parameters on the Dell Precision Server and Xookik are outlined in Tables 5.5a and 5.4a respectively. Total runtime and energy consumption (for Dell Precision Server) are presented in Tables 5.5 and 5.4.

5.3.3 *Modelling* C_{60} *Buckminsterfullerene*

C_{60} remains of interest to materials scientists as, despite extensive study since its discovery, the vibrational modes are yet to be completely and definitively assigned. To date, the most complete assignment process is the work by Parker *et al* [99], which compares INS, Raman and infrared spectroscopic data to ab initio model calculations of the vibrational frequencies for the two phases of polycrystalline C_{60} at low and ambient temperatures. However, this treatment still has significant limitations, especially with assignment of inter-molecular vibrational modes that are poorly described by density functional theory, and it has proven necessary to apply a semi-empirical approach (at least initially) to the correct assignment of these modes. Furthermore, the low temperature ordered phase of C_{60} has a lattice structure with a 240 atom basis, presenting matrices for diagonalisation of order 720.

Figure 5.7 on page 76 presents a visualisation of scattering intensities for a the low-temperature phase of C_{60}, at a constant momentum transfer of $Q = 19.5$, after execution on all four nodes on the Xookik cluster. The selected SCATTER variant for this environment and model, as determined by the calibration data presented in Table 5.4 was the MAGMA version. Figure 5.8 on page 77 presents 3 selected mode contributions and the associated frequencies to the scattering intensity at this value of Q. Simulation shows that, although the symmetry of the system remains evident, there is significantly increased complexity in comparison to Aluminium.

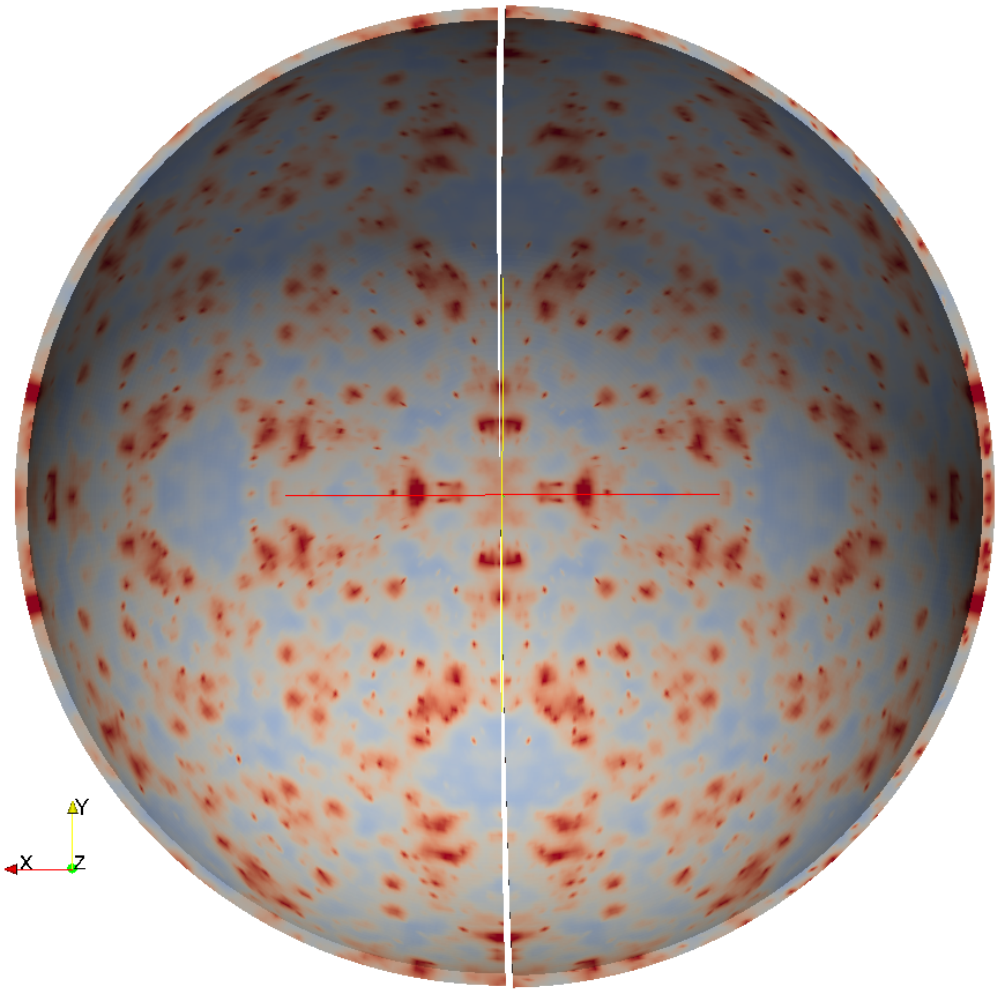
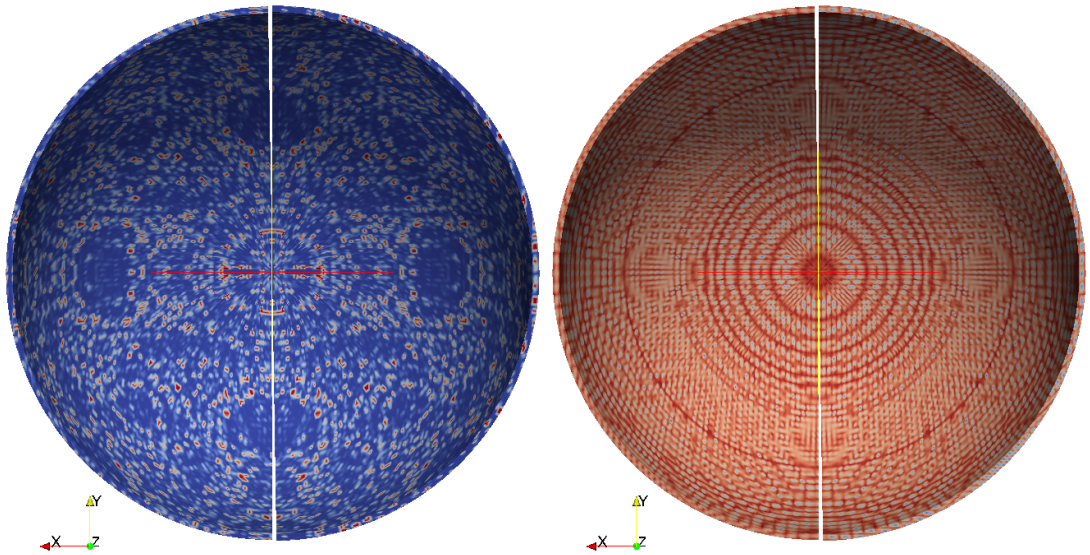
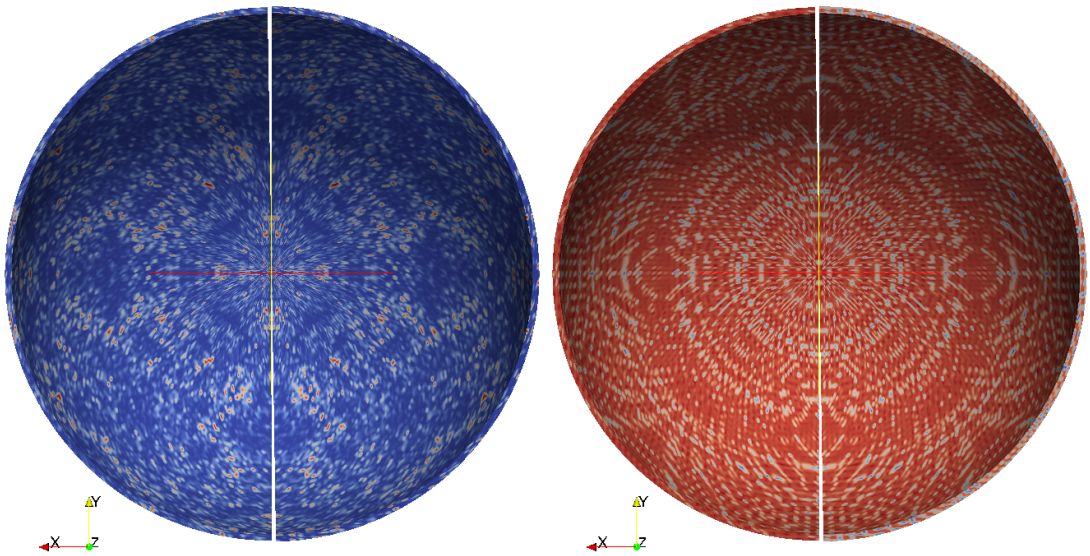


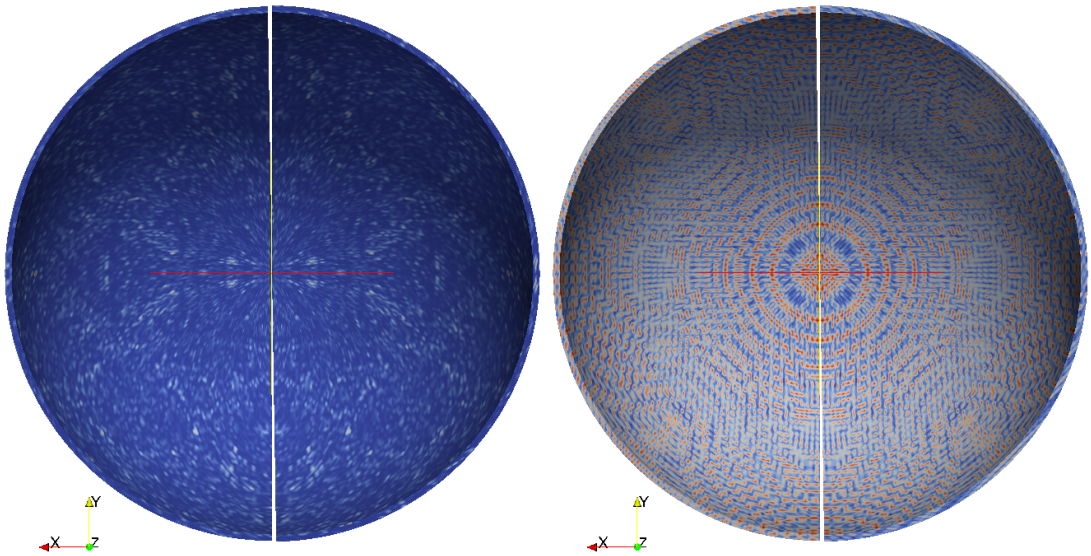
Figure 5.7: Scattering intensity in reciprocal space at a constant momentum transfer of Q = 19.5 for the low temperature phase of C_{60}.



(a) Mode 1



(b) Mode 10



(c) Mode 101

Figure 5.8: Scattering intensity contributions to 5.7 by mode (left) with frequencies (right) for 3 select modes out of 720 of the low temperature phase of C_{60} at constant momentum transfer ($Q = 19.5$).

5.4 DISCUSSION

The heterogeneous structured implementation combines pattern-based farm and structured pipeline forms with the **ANY** construct. Although adaptation has been reduced to a graph traversal problem via structural abstraction, the results of empirical optimisation as static adaptation strategy demonstrate conclusively that, even in this relatively simple case, there is no optimal structural form for possible variations in:

1. Environmental Conditions

   The different execution platforms (Dell Workstation and Xookik) present different hardware and software environments with specific constraints e.g. available CPU and GPU memory. The optimal variant, for the execution time performance metric and any given model, is generally different in each of these platforms. For example, the variant **PIPE**(P, E_G, C), using our EISPACK eigensolver in a pipeline pattern, is the optimal variant in all but one case on the Dell Workstation. On the less resource-constrained Xookik however, the **FARM**(P, E_L, C) and **FARM**(P, E_M, C) variants, with LAPACK and MAGMA are consistently superior.

   Further, runtime parameters (e.g. number of processes) and the software environment (e.g compilers and library versions) alter relative performance profiles. In particular, software libraries are subject to performance variations over time with new releases or different compilation and configuration options. Counterintuitively, the LAPACK versions of the program, without use of GPU resources are often the best choice, particularly for smaller systems. Here, the Xookik installation benefits from an optimised and tuned version of the BLAS libraries and avoids the overhead of memory transfer to the GPU. The eigensolver benchmarks demonstrate that despite the diminishing popularity of EISPACK, it provides an effective basis for the creation of application-specific tuned implementations of linear algebra routines for modern GPU platforms. However, the routines in the early MAGMA library have benefitted from sustained work towards optimisation and efficiency. On higher-end platforms such as the test Xookik cluster, MAGMA is now a consistently superior option for larger models.

2. Extra-functional performance measures.
   On the Dell Workstation, performance measures such as runtime lead to different variant selection than instantaneous power draw and cumulative energy usage. Instantaneous Power draw is an important metric in mobile applications, where batteries exhibit nonlinear efficiency characteristics, and in large data centers, where cooling costs may constitute a significant proportion of the energy budget.
   In practice, the performance measure is a composite function of other simpler such metrics that represents a user-weighted or context-specific multi-objective compromise. Given that these weights are subject to arbitrary variation, it is con-

ceivable that for many structural variants, a metric can be found for which it is optimal. Such a reversal raises the possibility of using metrics as a control mechanism for structural and qualitative behaviours.

3. Application-Specific Demands
   The test models and potential functions impose different performance requirements. Smaller lattice systems such as Aluminium and Graphite may even execute more efficiently without recourse to GPU resources. Large systems with more atoms in a basis cell become increasingly more efficient on GPUs as the data parallelism available increases.

While abstraction and the construction of larger functional systems by the composition of verified components has historically provided the foundation of software engineering, here, we have demonstrated that an optimal choice between a set of structural alternatives does not exist. These results carry the much broader implication that software is not performance-composable. Restated alternatively, software performance does not always exhibit optimal substructure [33] and therefore larger applications cannot be optimised by the ad hoc greedy heuristic that has become common practice where components are individually optimised in libraries and subsequently integrated into programs after distribution. As functional modules are developed, any early decisions made without the benefit of complete information regarding all of the relevant runtime factors, component interactions and overall structural context represents a hidden compromise. Given this dependence, the only way to consistently achieve absolute optimal performance may be empirical optimisation in a specific runtime setting. However, this requires that black box components in libraries and modules expose options to a performance-optimising layer through a consistent interface.

Limitations

Practical considerations of the real application have kept this evaluation necessarily simple. However, the size of the variant graph can be potentially very large and offer complex options and constraints. In this setting, exhaustive search may not be a feasible method of identifying the best program variants. Heuristic search techniques, simulation and approximate performance modelling may eliminate the need for exhaustive calibration to achieve near-optimal performance at reasonable computational cost.

The process of variant selection has been only semi-automated for this limited search space. However, the full potential of this method requires novel software tooling and language integration to facilitate systematic exploration of program variants. The new features in existing languages (e.g. Java, C++), that introduce first class functions, create the possibility of high-level integration with existing applications.

The pipeline variants introduce out-of-process memory transfer overhead and contention that constitutes a bottleneck to the individual processes as memory transfers are serialised. The implications of more efficient queueing may include significant per-

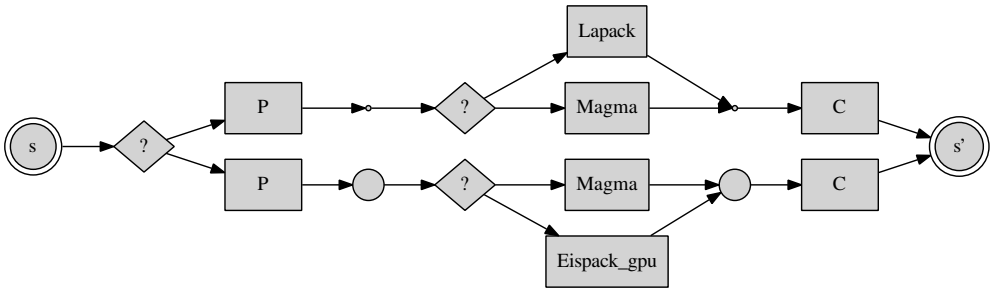
formnance improvements in the pipeline variants. Nevertheless, the pipeline implementation consistently delivers better performance on the memory-constrained Dell Workstation.

Global Memory Bandwidth	Coalesced memory access by algorithm reorganisation. Transposed matrix layout in some subtasks is necessary to achieve higher memory transfer bandwidth
Overlapped Execution	Asynchronous transfers between Host and GPU memory over multiple streams allow concurrent kernel execution and overlapped I/O.
GPU Occupancy	Improved register memory usage by the elimination (or reuse when appropriate) of extraneous register variables to improve GPU occupancy and facilitate latency hiding on the streaming multiprocessors.
Shared Memory	Use of explicit caching in shared memory to limit costly global memory accesses.
Grid and Block Dimensions	Empirical determination of launch configuration by trial and error. While, the guidelines recommend that thread blocks sizes should be multiples of a warp to allow latency hiding for multiple warps, it is necessary to determine actual optimal block sizes by testing. The different kernels performed optimally at distinct block dimensions.

Table 5.1: Performance optimisations applied.

	Dell Precision T7500 Workstation	Xookik
Nodes	1	4
Processing Elements per Node	4	12
Processor Clock	2.0 GHz	3.07 GHz
Architecture	Intel x86-64	
Memory per Node	4 GB	50 GB
Network	FastEthernet 100 Mbps	Infiniband 160 Gbit/s
Operating System	Ubuntu Linux 11.04 (kernel 2.6.38-8)	GNU Linux (kernel 2.6.27)
Compiler	GCC gfortran 4.4.5	GCC gfortran 4.3.2
MPI Version	OpenMPI 1.4.3	OpenMPI 1.3.3
GPUs per Node	1	1
GPU Model	NVIDIA Tesla C2050	NVIDIA Tesla M2090
GPU Memory	3 GB	6 GB
CUDA Cores	448	512
CUDA Version		4.0 V0.2.1221
CUDA Driver Version		290.10

Table 5.2: Heterogeneous Test Configurations



	SCATTER Variant	Description
1	**FARM**(**SEQ**(P, E_L, C))	Pattern-based implementation with LAPACK zheevr solver
2	**FARM**(**SEQ**(P, E_M, C))	Pattern-based implementation with MAGMA zheevr solver
4	**PIPE**(P, E_M, C)	Structured Lyapunov drift-stabilised pipeline with MAGMA zheevr
5	**PIPE**(P, E_G, C)	Structured Lyapunov drift-stabilised pipeline with EISPACK$_{gpu}$

Table 5.3: SCATTER variants evaluated.

	Model	Potential	Matrix Order	$\left(\frac{\lvert\mathbf{Q}\rvert_{max}-\lvert\mathbf{Q}\rvert_{min}}{\delta\lvert\mathbf{Q}\rvert}\right)\times\left(\frac{2\pi}{\delta\theta}\right)^2$
1	Aluminium	Lennard Jones	3	80000
2	Graphite	Brenner	6	80000
3	SWNT C(10,10)	Brenner	120	80000
4	C_{60} Ambient Temperature Phase	Brenner (intramol) Lennard Jones (intermol)	180	80000
5	C_{60} Low Temperature Phase		720	80000

(a) Xookik Test Models

	Procs	**FARM**(P, E_L, C)	**FARM**(P, E_M, C)	**PIPE**(P, E_M, C)	**PIPE**(P, E_G, C)
3	2	91.6	103.3	88.0	**85.6**
	4	**47.1**	55.5	48.0	48.1
	8	**26.0**	33.6	30.0	33.6
	12	**18.6**	27.5	26.6	35.1
4	2	177.1	**141.8**	193.8	195.8
	4	89.9	**74.8**	102.8	104.9
	8	48.6	**43.4**	55.9	57.0
	12	**33.4**	34.1	45.0	45.8
5	2	182.5	**89.7**	111.3	92.8
	4	93.7	**48.6**	65.5	71.8
	8	53.4	**30.3**	42.8	62.4
	12	59.7	**33.3**	33.7	51.3

(b)

Table 5.4: Runtimes (seconds) of SCATTER variants on single node of Xookik cluster.

	Model	Potential	Matrix Order	$\left(\frac{\|\mathbf{Q}\|_{max}-\|\mathbf{Q}\|_{min}}{\delta\|\mathbf{Q}\|}\right) \times \left(\frac{2\pi}{\delta\theta}\right)^2$
1	Aluminium	Lennard Jones	3	10000
2	Graphite	Brenner	6	10000
3	SWNT C(10,10)	Brenner	120	$100 \times 50^2 = 250000$
4	C_{60} Ambient Temperature Phase	Brenner (intramol) Lennard Jones (intermol)	180	$20 \times 60^2 = 72000$
5	C_{60} Low Temperature Phase		720	$10 \times 15^2 = 2250$

(a) Dell Server Test Models

	Procs	**FARM**(P, E_L, C)	**FARM**(P, E_M, C)	**PIPE**(P, E_M, C)	**PIPE**(P, E_G, C)
2	1	573.5	739.3		
	2	288.3	391.78		
	4	149.4	263.6		
3	1	423.9	391.0	247.7	**211.7**
	2	209.9	198.0	175.3	**109.4**
	4	114.0	107.3	137.8	**66.2**
4	1	760.0	569.5	612.0	**569.2**
	2	391.0	301.1	329.8	**288.7**
	4	209.7	**153.9**	232.0	159.0
5	1	562.2	403.3	156.1	**94.0**
	2	339.6	242.8	152.1	**74.0**
	4	220.4	142.8	153.00	**72.0**

(b)

Table 5.5: Runtimes (seconds) of SCATTER variants on Dell Workstation.

	Model	Potential	Matrix Order	$\left(\frac{\|\mathbf{Q}\|_{max}-\|\mathbf{Q}\|_{min}}{\delta\|\mathbf{Q}\|}\right) \times \left(\frac{2\pi}{\delta\theta}\right)^2$
1	Aluminium	Lennard Jones	3	10000
2	Graphite	Brenner	6	10000
3	SWNT C(10,10)	Brenner	120	$100 \times 50^2 = 250000$
4	C_{60} Ambient Temperature Phase	Brenner (intramol) Lennard Jones (intermol)	180	$20 \times 60^2 = 72000$
5	C_{60} Low Temperature Phase		720	$10 \times 15^2 = 2250$

(a) Dell Server Test Models

	Procs	**FARM**(P, E_L, C)	**FARM**(P, E_M, C)	**PIPE**(P, E_M, C)	**PIPE**(P, E_G, C)
2	1	32.61	1.18		
	2	17.59	30.54		
	4	10.02	21.73		
3	1	25.05	29.48	19.78	**17.18**
	2	12.81	15.69	14.45	**9.65**
	4	7.65	9.12	11.71	**6.47**
4	1	45.01	**42.93**	47.5	45.28
	2	23.98	**23.77**	26.97	24.53
	4	13.93	**13.18**	19.86	15.03
5	1	33.2	30.58	13.31	**9.13**
	2	21.59	18.77	13.02	**7.75**
	4	15.07	12.23	13.2	**7.59**

(b)

Table 5.6: Total energy consumption (Watt-hours) of SCATTER variants on Dell Workstation .

	Model	Potential	Matrix Order	$\left(\frac{\|\mathbf{Q}\|_{max}-\|\mathbf{Q}\|_{min}}{\delta\|\mathbf{Q}\|}\right) \times \left(\frac{2\pi}{\delta\theta}\right)^2$
1	Aluminium	Lennard Jones	3	10000
2	Graphite	Brenner	6	10000
3	SWNT C(10,10)	Brenner	120	$100 \times 50^2 = 250000$
4	C_{60} Ambient Temperature Phase	Brenner (intramol) Lennard Jones (intermol)	180	$20 \times 60^2 = 72000$
5	C_{60} Low Temperature Phase		720	$10 \times 15^2 = 2250$

(a) Dell Server Test Models

	Procs	**FARM**(P, E_L, C)	**FARM**(P, E_M, C)	**PIPE**(P, E_M, C)	**PIPE**(P, E_G, C)
3	1	212.8	271.4	287.5	292.1
	2	219.7	285.2	296.7	317.4
	4	241.5	305.9	305.9	351.9
4	1	213.2	271.4	279.5	286.4
	2	220.8	285.2	294.4	305.9
	4	239.2	308.2	308.2	340.4
5	1	212.8	237.0	307.1	349.6
	2	228.9	278.3	308.2	377.2
	4	246.1	308.2	310.5	379.5

(b)

Table 5.7: Instantaneous Power Consumption (Watts) of SCATTER variants on Dell Workstation

6

DYNAMIC ADAPTATION IN THROUGHPUT-ORIENTED PIPELINES

Chapter 5 considered a structural representation of program alternatives that allows static adaptation by an exhaustive exploration of the space of program variants represented by the **ANY** operator. The performance measures considered – runtime, total energy consumption and average power draw – were cumulative. This chapter considers runtime optimisation of instantaneous performance measures in the structured pipeline variant of the Scatter application and further begins to explore the possibility of using these metrics as a dynamic control mechanism in an ancillary application.

While the structural forms in the previous chapter demonstrate static adaptation, this chapter considers the instrumented queues of Section 3.2.1 as mechanisms of composition and dynamic adaptation. Therefore, one objective is to demonstrate that the adaptation framework of Chapter 3 presents sufficient flexibility to allow the realisation of diverse dynamic adaptation mechanisms. Another objective is to highlight the importance of the instrumented queues as a source of runtime feedback that is representative of the extrafunctional program state as well as their role as mechanisms for the imposition of dynamic control. Thus, in considering both centralised and decentralised coordination approaches, heuristics are presented that are inherently centred around the unifying concept of cost while differing significantly in the manner in which information about state is gathered and, conversely, control decisions are effected.

6.1 OPTIMISING UTILISATION

Parallel pipelines decompose computational problems into concurrent stages with outputs from one stage providing inputs to its immediate successors. The tasks flowing through the pipeline may correspond to individual loop iterations. The pipeline is a fundamental compositional unit of more complex streaming heterogeneous applications that combine CPU and GPU resources [98, 109].

Consider **FOR**(P, n), a loop within a program with n iterations executing a body P with no dependencies between iterations as is characteristic of a divisible workload [18]. Let us assume that the program is being adapted for execution on a heterogeneous accelerator. Whether by profiling or otherwise, a contiguous block of statements B is identified within P as a good candidate for re-implementation that meets the performance criteria of the device. B is subsequently rewritten for the new platform as a functionally equivalent alternative B'. In general, if A and C are statements

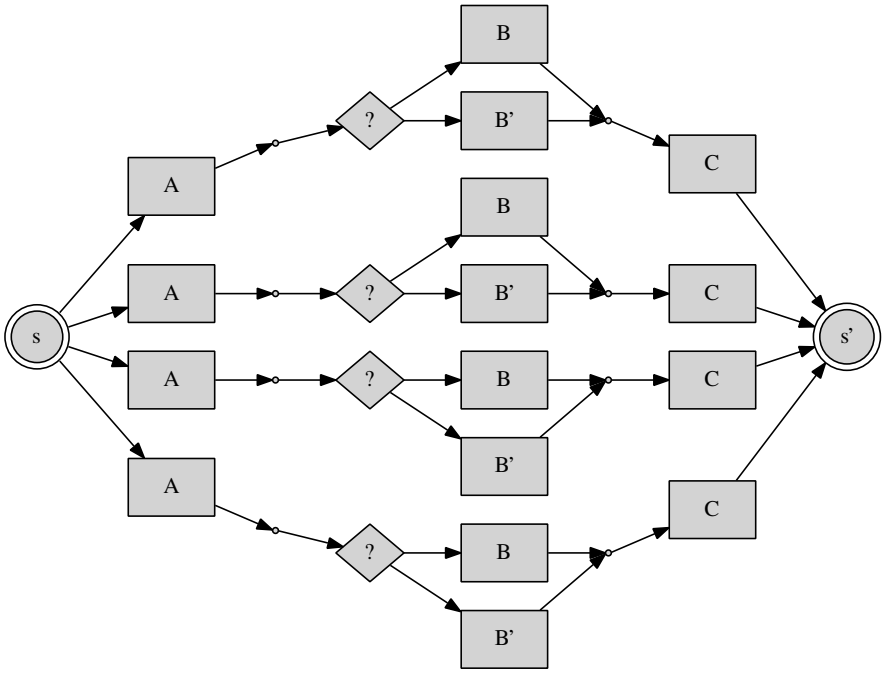


Figure 6.1: Farm transformation of a divisible workload with GPU payload.

executed before and after the accelerator kernel B', then dependencies exist that impose a strict ordering on the subroutines.

$$A_i \prec B_i \prec C_i \; \forall \, i = 1...N$$

where A_i, B_i and C_i represent the ith iteration of A, B and C respectively.

The implementation of a GPU version may be represented as the structural transformation:

$$\begin{aligned} P &\rightarrow \textbf{SEQ}(A, B, C) \\ B &\rightarrow \textbf{ANY}(B, B') \end{aligned}$$

Thus, the transformed program may be represented as

$$\begin{aligned} \textbf{FOR}(P, n) &\rightarrow \textbf{FOR}(\textbf{SEQ}(A, B, C), n) \\ &\rightarrow \textbf{FOR}(\textbf{SEQ}(A, \textbf{ANY}(B, B'), C), n) \end{aligned}$$

Since $\textbf{FOR}(P, n)$ is a divisible workload, a decomposition into a simple **FARM** based on the master-worker pattern would be natural.

$$\begin{aligned} \textbf{FOR}(P, n) &\rightarrow \textbf{FOR}(\textbf{SEQ}(A, B, C), n) \\ &\rightarrow \textbf{FARM}(\textbf{SEQ}(A, \textbf{ANY}(B, B'), C)) \end{aligned}$$

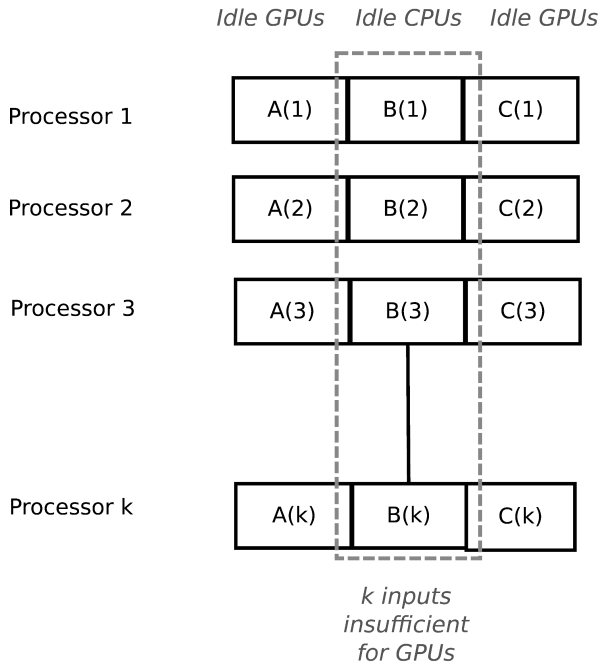


Figure 6.2: A, B and C are code sections in a main loop decomposed cyclically over k processes in an SPMD pattern.

However, this conventional GPU-as-accelerator model leads to a number of limitations (Figure 6.2 on page 91):

1. The GPU kernels may only perform optimally for batched inputs that are able to use all available streaming multiprocessors. While the limited host memory available to the application may only hold k processes, k instances of B will, in general, not necessarily be sufficient for peak efficiency of the GPU kernels (Figure 5.6 on page 74).

2. Latency hiding, necessary to offset the cost of expensive host-to-device memory transfer overhead, is not possible.

3. There is alternate resource idling as the GPU is typically idle during A and C while the CPUs are idle during B.

The actual, relative performance of A, B and C on the CPUs and GPUs may only be determined at runtime as they depend on application-specific requirements, capabilities of the hardware and software environment and the choice of performance measure. Potentially, the presence of multiple GPUs adds further complexity to the overall problem.

In practice, the large memory requirements of an application may impose additional limitations on the number of processes that can be held in main memory, even when the performance penalty of inefficient virtual memory is disregarded. This application may be unable to generate sufficient GPU workload within the constraints of available memory to maximise utilisation of the computational resources available.

6.1.1 *Structure and Utilisation*

To avoid these limitations, the program may be decoupled into a pipeline to extend the parallelism. The simple pipeline structural pattern may be regarded as a special case of the following alternative composed **PIPE** transformations in which each farm is restricted to a single worker:

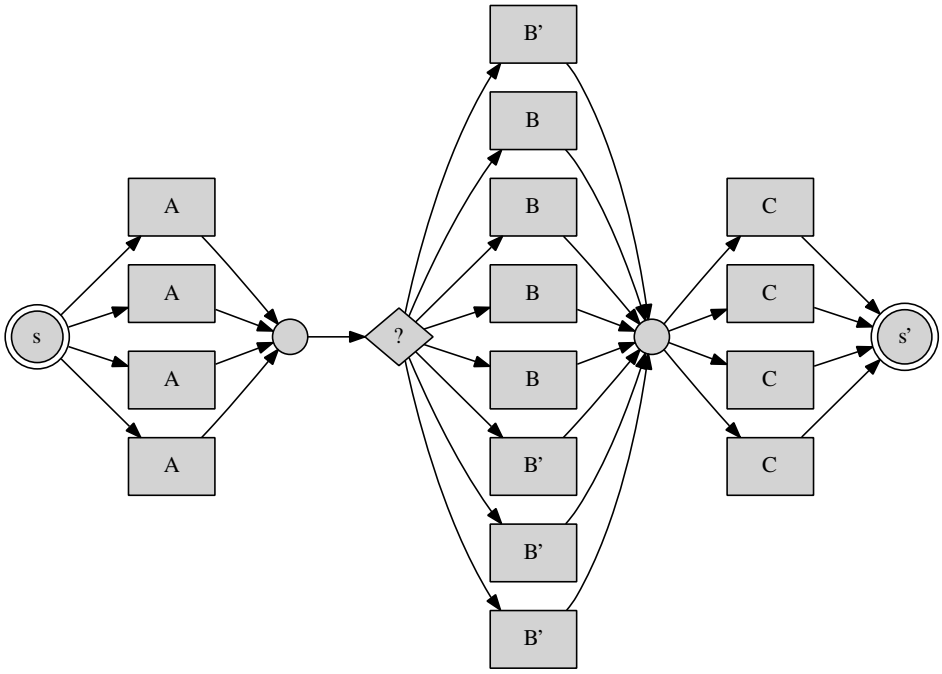


Figure 6.3: Pipeline-of-farms transformation

1. **Pipeline-of-Farms**

   A nested skeleton representation with the individual stages of a pipeline executed by multiple workers in a farm (Figure 6.3).

$$\begin{aligned}\textbf{FOR}(P, n) &\rightarrow \textbf{FOR}(\textbf{SEQ}(A, B, C), n)\\ &\rightarrow \textbf{PIPE}(A, \textbf{ANY}(B, B'), C)\\ &\rightarrow \textbf{PIPE}(\textbf{FARM}(A), \textbf{FARM}(\textbf{ANY}(B, B')), \textbf{FARM}(C))\end{aligned}$$

2. **Farm-of-Pipelines**

   Another nested skeleton representation with a farm comprising multiple workers that are themselves instances of the pipeline pattern (Figure 6.4).

$$\begin{aligned}\textbf{FOR}(P, n) &\rightarrow \textbf{FOR}(\textbf{SEQ}(A, B, C), n)\\ &\rightarrow \textbf{FARM}(\textbf{SEQ}(A, \textbf{ANY}(B, B'), C))\\ &\rightarrow \textbf{FARM}(\textbf{PIPE}(A, \textbf{ANY}(B, B'), C))\end{aligned}$$

For the pipeline-of-farms pattern, (Figure 6.3 on page 93), the non-blocking semantics of an asynchronous pipeline avoid the constraints imposed by tight coupling. As processing at each stage is independent, inputs to an accelerator may be arbitrarily grouped into batch sizes that match the ideal performance requirements of any kernels. Furthermore, latency hiding, through overlapped computation and communication, becomes possible.

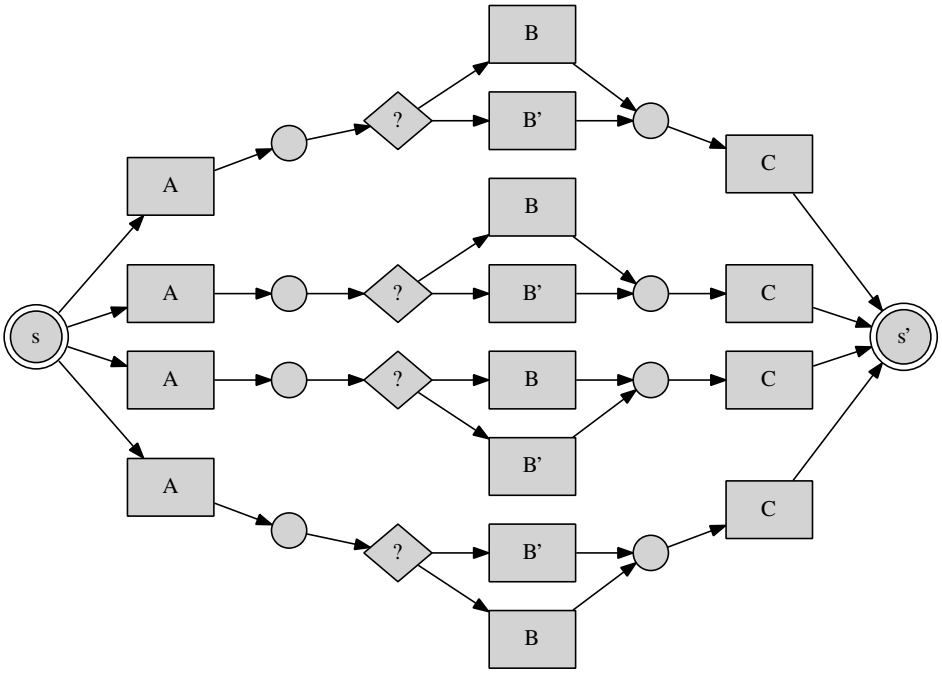


Figure 6.4: Farm-of-pipelines transformation

6.1.2 *Queues and Utilisation*

Following the choice of an alternative structure, an examination of queue levels in a three-stage heterogeneous pipeline makes the utilisation problem evident. In a simple simulation of the pipeline of farms structure in Figure 6.3 on page 93, stages I and III of the pipeline are strictly CPU-only stages as GPU implementations are unavailable. As Stage II is the computationally dominant stage of the pipeline and has both CPU and GPU implementations, only the GPU implementation is in use as it offers higher performance. For stages I and II, there are two expected scenarios depending on whether the CPU or GPU constitutes the bottleneck in the pipeline (Figure 6.5 on page 95).

1. **Producer Deficit**: Stage I constitutes the bottleneck to the pipeline if and only if it is unable to provide and process data for the stage II GPU at a sufficient rate. This may arguably be the result of performance or memory limitations on the host machine and is illustrated in Figure 6.5 on page 95(a) for a simple simulation. In this case, it may be necessary to use external resources to increase output from this stage for better GPU utilisation. Adding external CPUs will eventually lead to the next scenario.

2. **Consumer Deficit**: If Stage II constitutes the computational bottleneck in the application pipeline, the CPUs at stage I will complete their work units early after filling the output queue and will wait for the GPU to complete the current batch of work at stage II. As presented in Figure 6.5 on page 95(b), there is noticeable resource waste at the idle stages of the pipeline caused by a pattern of intermittent CPU activity.

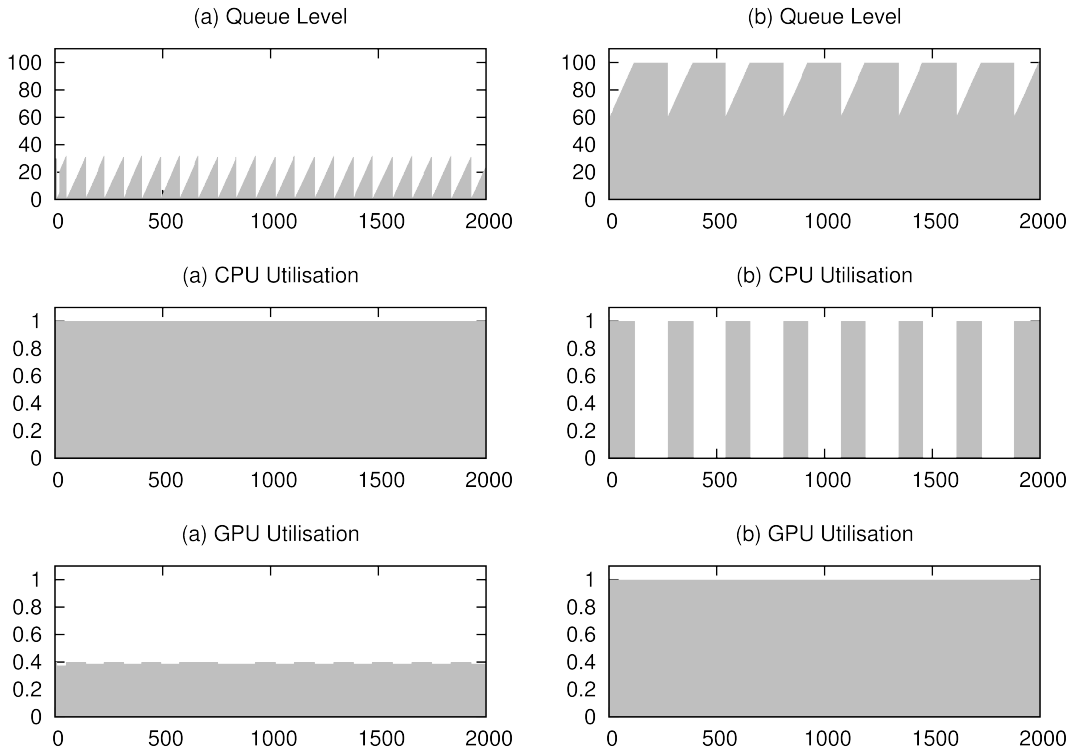


Figure 6.5: Simulation of the queue levels and CPU/GPU usage patterns for (a) Producer deficit and (b) Consumer deficit.

In Figure 6.6 on page 96, the simple heuristic is applied to the same simulation to achieve improved resource utilisation.

In applying the adaptation framework, a program with n queues is regarded as representing a discrete n-dimensional extrafunctional state space of queue levels. For any given operation or stage, the corresponding input and output queues constitute a queue pair within a two-dimensional subspace where operations result in transitions to neighbouring states.

As an example, consider only the input and output queues for the GPU stage of the three-stage pipeline in Figure 6.3 on page 93, the effect of executing any of the stages A, B and C on this subspace (q_1, q_2) of the program state are the transition vectors (1,0), (-1,1) and (0,-1). Throughput vectors may be defined that represent the expectation throughput of each operation as $(T_A, 0)$, $(-T_B, T_B)$, $(0, -T_C)$. These are analogous to the components of a velocity vector in the state space.

In general, the components of the throughput vectors follow from the topology of the dataflow graph and are subject to stochastic variation that cause the program to perform a random walk in the queue state space. However, at runtime, two conflicting concerns exist:

1. Provision of sufficient instantaneous input workload to the batch-oriented GPU kernels to maintain utilisation, efficiency and minimise idling due to latency given their nonlinear performance characteristics (Figure 5.6 on page 74). For a

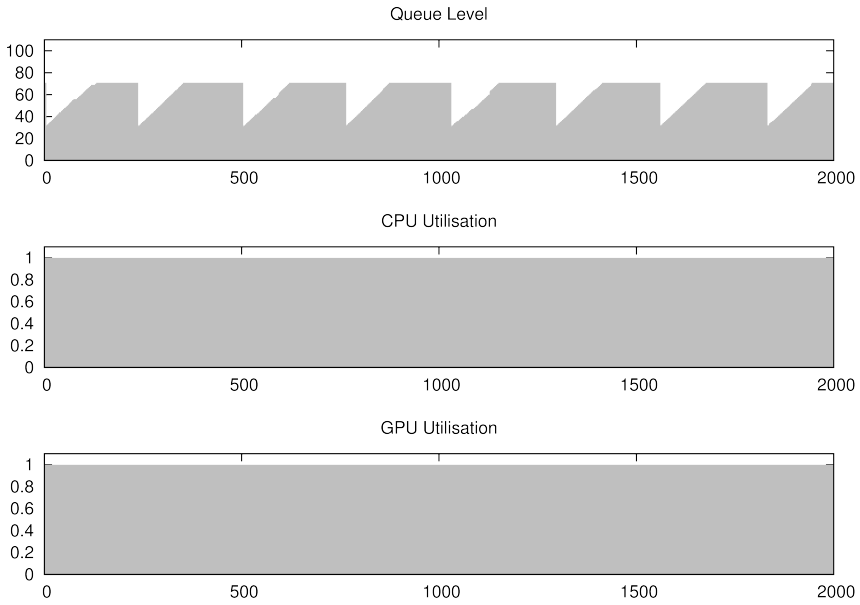


Figure 6.6: A basic simulation of the our heuristic strategy, demonstrating that bottlenecks have been eliminated by queue monitoring and dynamic process reassignment.

GPU input queue level q_g and ideal kernel batch size b, this corresponds to the runtime efficiency requirement of Equation 6.1.

$$q_g \geqslant b \tag{6.1}$$

2. Avoidance of queue overflow for long-running throughput-oriented jobs that occur when the memory required by the program exceeds available primary memory. Where c_i is the memory required for each entry in queue i, q_i is the corresponding queue level and M is the total memory available to the program, maintaining memory utilisation within defined bounds to avoid the degraded performance from paging or outright program failure is the hard constraint of Equation 6.2.

$$\sum_i c_i q_i < M \tag{6.2}$$

Subject to the additional constraint that the queue levels q_i are non-negative, these considerations impose boundaries that describe a triangular feasible region for the GPU stage input and output queues. With a stochastic controller that alters the probability of executing any operation, the individual stage throughputs T_i may be adjusted to maximise expected utilisation or optimisation of some performance measure. In this circumstance, instrumented queues allow information to be gathered about the current queue states in order to select the appropriate action determined by the policy.

6.2 PIPELINE INSTANCES

	FastFlow	**Market Based**	**Scatter**
G	Random Hermitian Matrix		Dynamical Matrix for $P(Q, \theta, \phi)$
S	EISPACK$_{gpu}$	**ANY**(*magma_zheevr*, *lapack_zheevr*)	**ANY**(*magma_zheevr*, EISPACK$_{gpu}$)
V	Error Measure		Scattering Contribution
Queue Type	Generalised Queues	Queue	Queue
Queueing Discipline	FIFO	fuzzy cost as priority	FIFO
Optimiser	Simple Heuristic	Market Based Control	Lyapunov Drift
Queue size	N/A	Available	Available
Queue Implementation	non-blocking internal data structure	shared internal data structure, ZeroMQ, HTTP	Redis
Supporting Structure	Native Threads	Native Processes	SPMD Process Pool
Implementation Mechanism	pthreads	UNIX Processes	MPI Processes

Table 6.1: Three adaptive pipeline instances of **FARM**(**PIPE**(**FARM**(G), **FARM**(S), **FARM**(V))) using generalised queues in FastFlow with a simple heuristic and instrumented queues for decentralised coordination in a market-based system and centralised coordination in the SCATTER application.

The divisible workload with a GPU payload is a general application pattern that includes the SCATTER program where the parallel implementation has resulted in a hierarchical composition of two dwarf instances:

1. MapReduce or Monte Carlo for the embarrassingly parallel application.
2. Linear Algebra for the GPU Kernels.

The composition of dwarfs may be realised using queues, allowing the application of our framework. We consider dynamic adaptation in three instances of this pattern:

1. Kernel Validation Applications
   a) A FastFlow implementation used to perform validation of the EISPACK$_{gpu}$ eigensolver kernels.
   b) A Market-Based re-implementation of the validation framework with instrumented queues used to explore decentralised coordination with the MAGMA eigensolvers.
2. The pipeline variant of the SCATTER application presented in the previous chapter used to explore centralised coordination.

These instances of the adaptive pipeline, summarised in Table 6.1 on page 97, are realisations of the dynamic adaptation framework outlined in Chapter 3. The general definition of instrumented queues (Section 3.2.1) are specialised into auction mechanisms as the basis of a decentralised market with emergent coordination. For the pipeline variant of SCATTER, a centralised heuristic is implemented based on a cost function that is derived from the extrafunctional state associated with the queues.

6.2.1 *Kernel Validation Applications*

Establishing the numerical accuracy of the kernel implementation is an ancillary problem that is approached by performing validation testing on large numbers of test Hermitian matrices. This testing process involves:

1. **Generation** of suitable Hermitian test matrices, A.
2. **Solution** using the test GPU kernels to compute the eigenvectors E and eigenvalues D on the GPU.
3. **Verification** of the computed eigenvectors and eigenvalues on the CPU. For a matrix of eigenvectors E and a corresponding diagonal matrix of eigenvalues D,

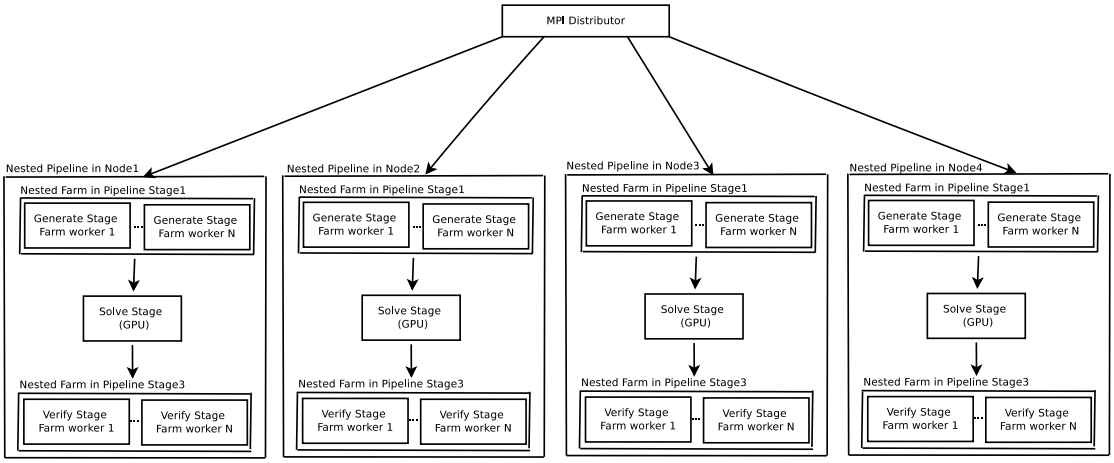


Figure 6.7: A three-level hierarchy of skeletons, consisting of a distributed farm of MPI workers at the top-level containing nested pipeline workers with nested farm stages and a GPU stage.

it is expected that $AE = ED$. Therefore, a possible error function is the Frobenius norm of the matrix

$$\epsilon = \|AE - ED\|_F$$

where the Frobenius norm of an $m \times n$ matrix M is defined as

$$\|M\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|m_{ij}|^2}$$

The error function is verified to be less than a tolerance value that accounts for acceptable floating point truncation error.

This is a generic streaming test application for evaluating the correctness of kernels for a computational accelerator that are the result of the transformation $S \rightarrow S'$ on a potentially large number of specimen inputs. Kernels executing on the device take some input data generated on the host, perform computations on that data on the accelerator and produce results that require verification on the host. This program may be represented as

$$\mathbf{FOR}(\mathbf{SEQ}(G, S', V), n)$$

where the stages G, S' and V correspond to data generation, solution and verification operations respectively and n is the number of specimen inputs.

6.2.1.1 *Testing Kernels in FastFlow*

The FastFlow framework [4] is a C++ template library that provides structural constructs for the implementation of parallel shared memory programs for cache-coherent multicore architectures. Parallel programs are implemented as compositions of simple skeleton primitives, that include farm and pipeline primitives, into complex skeleton trees. Despite relying on an efficient implementation of lock-free and fence-free

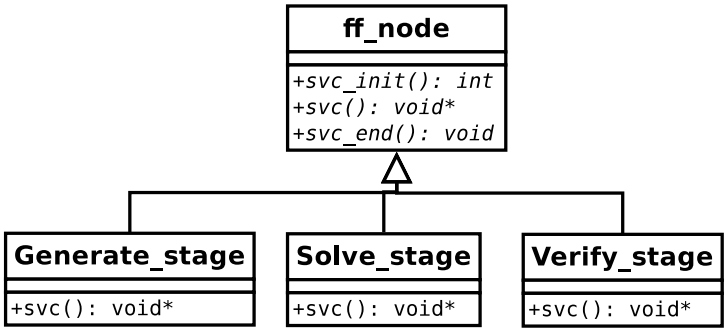


Figure 6.8: Pipeline stages inherit from the ff_node class.

queues, FastFlow emphasises profiling of user functions over queues as inputs to internal schedulers.

FastFlow's streaming patterns provide coordination to control the flow of work between multiple concurrent threads. While extensively tested in shared-memory multicore environments, GPU integration remains an area of active research where the suitability of FastFlow for workloads that incorporate GPU kernels is yet to be established.

To test the numerical accuracy of the EISPACK$_{gpu}$ eigensolvers, an implementation of the kernel validation test application is implemented using the pipeline constructs of the FastFlow framework. Generation and verification in the first and final stages of the pipeline operate on these individual batches of independent test problems. This presents another level of parallelism that is well suited to the farm skeleton. Employing skeletal composition allows a nested parallel hierarchy.

The abstract FastFlow pipeline prototype class provides three virtual overridable member functions:

- svc_init for stage initialisation,
- svc to perform the actual computations at each stage,
- svc_end for finalisation and cleanup.

A class derived from the ff_node prototype, may represent either a generic pipeline stage or worker in a farm. The ff_node class is subclassed to implement a Generate_stage, Solve_stage and Verify_stage representing the initial, intermediate and terminal stages of the pipeline (Figure 6.8 on page 100). The Solve_stage invokes the EISPACK$_{gpu}$ driver, as the current version of FastFlow does not provide any direct mechanisms for managing the execution of CUDA kernels.

The pipeline stages themselves are added to a ff_pipeline container object in the proper sequence and execution started by an invocation of the run_and_wait_end method (Figure 6.9 on page 101).

It is difficult to justify the conceptual complexity of nesting the FastFlow farm skeleton within the initial and final stages of the pipeline, in order to achieve simple multithreaded functionality. Therefore, the nested farms are implemented with OpenMP work distribution.

```
int main(int argc, char * argv[])
{
  ...
  ff_pipeline pipe;
  ff_node *generate_stage;
  ff_node *solve_stage;
  ff_node *verify_stage;

  //Instantiate pipeline stages
  generate_stage= new Generate_stage();
  solve_stage = new Solve_stage();
  verify_stage = new Verify_stage();

  //Add stages to pipeline
  pipe.add_stage(generate_stage);
  pipe.add_stage(solve_stage);
  pipe.add_stage(verify_stage);

  ...

  //start the pipeline
  pipe.run_and_wait_end();

  ...
}
```

Figure 6.9: Pipeline instantiation and execution.

An additional level of nesting, using an MPI farm to handle distribution over multiple nodes in a cluster, allows a three-level hierarchy, consisting of a distributed farm of nested pipeline workers with nested farms at each stage (Figure 6.7 on page 99).

Simple Heuristic with Generalised Queues

While FastFlow has been evaluated on small-scale and fine-grained parallel jobs, applications to large-scale, coarse grained workloads with significant memory demands are an area of active work. Early testing revealed that the pipeline implementation in FastFlow leads to steadily increasing buffer levels before bottleneck stages. For long-running high-throughput pipelines, this very rapidly consumes all available memory on the machine and leads to a fatal program error.

As the queues in FastFlow, an existing skeleton framework, do not meet the definition of generalised queues, limited information is available. Therefore, a simple heuristic strategy is necessary. The extreme cases that lead to buffer overruns and underruns may be avoided by dynamically reallocating the CPU at stage I to other stages of the pipeline (Stage II) when necessary. This approach can be reduced to two simple rules local to each process:

1. Keep all queue levels q_i above a set lower bound q_L to ensure the availability of work for the inputs to succeeding pipeline stages i.e. $q_i > q_L$.

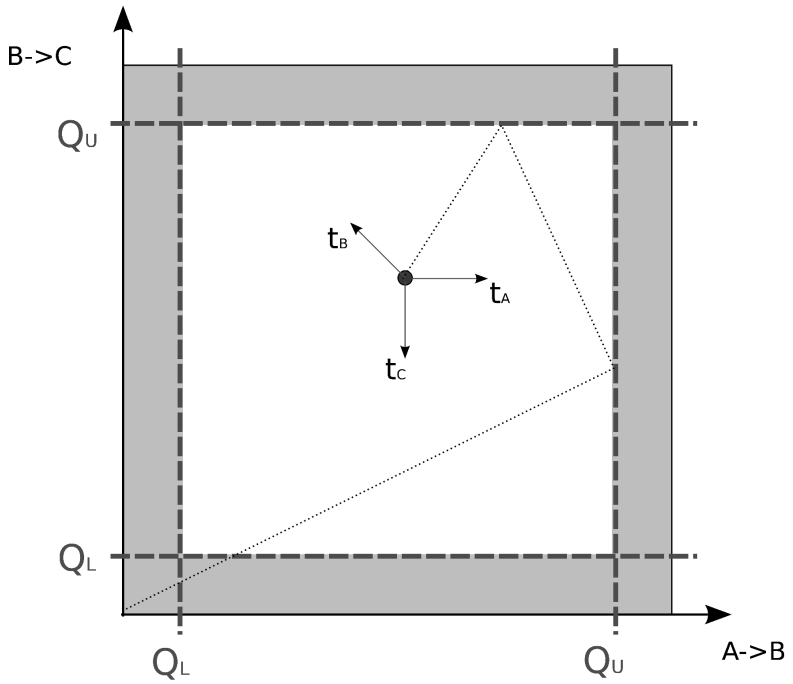


Figure 6.10: Queue space visualisation. The effect of the heuristic is to keep the system within the bounding box.

2. Keep all queue levels q_i below a set upper bound q_U to ensure the availability of space for the outputs of preceding pipeline stages i.e. $q_i < q_U$.

Queue stability occurs when the time-averaged rates of production and consumption are matched and buffer overruns and underruns are avoided. This is only possible when the buffer levels have lower variability. Figure 6.10 on page 102 illustrates that the effect of the heuristic is to keep the system within the bounding box.

6.2.1.2 *A Market-Based Validation Pipeline*

One approach to solving complex scheduling problems is *Market Based Control* (MBC) in which virtual economies of trading agents provide loosely-coupled coordination of resources [117]. The MBC paradigm attempts to exploit the tendency of economic systems to exhibit complex self-organising characteristics that arise from the interactions of competing agents seeking to maximise their individual interests with limited information. In markets, price signals carry implicit information about system dynamics resulting from the interplay of supply and demand. The competitive equilibria that arise in market systems are typically allocatively efficient [36] and respond to external influences as if directed by an *invisible hand*. They are evolving complex systems [11] which are resilient to shocks, degrade gracefully in the event of failure and are highly scalable as they avoid the communication overhead of centralised control in very large systems. Applications of MBC include distributed air conditioning control in buildings [30], job shop scheduling [136], auction-based manufacturing control [21] and multi-robot coordination[62]. Particularly relevant to this domain are auction protocols for decentralised scheduling [131] and resource allocation[81].

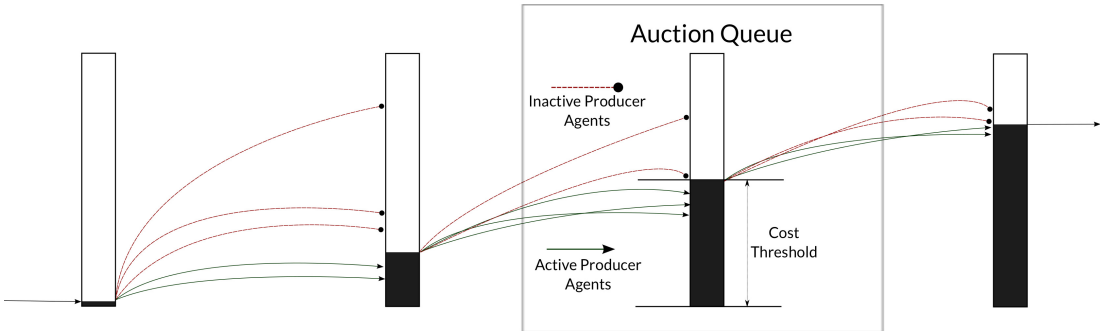


Figure 6.11: Cost-based auctions allow data exchange between agents

This alternative implementation of the test framework of Subsection 6.2.1.1 uses market-inspired coordination to achieve decentralised coordination with the LAPACK and MAGMA libraries as CPU and GPU alternatives. A distributed ensemble of agents perform transformations on data until they reach a terminal *collector agent* at the system boundary.

To impose market-based interactions on these agents, auctions are introduced where competition is based on minimisation of a user-defined cost-function. The fairness of these competitions is controlled by a probabilistic temperature parameter.

Reverse Auctions with Instrumented Queues

At intermediate queues, cost-based auctions allow data exchange between agents (Figure 6.11 on page 103). This approach provides an indirect reverse single-sided auction where producer agents compete to be chosen by consumer agents seeking the lowest cost items. The reverse auction is asymmetric and places the competitive pressure on the producers. With blocking semantics, it is achieved by:

1. Allocating a *fixed number of slots per producer agent* in the queue. A producer agent is only allowed to add a new entry to the queue if it has an available unused slot. Producers enter a wait state until a free slot becomes available. The `put_filter` function of the instrumented queue implementation returns `true` if the current agent, as determined by the metadata `m` associated with each entry, has not exceeded its slot quota.

2. *Allowing unrestricted consumer agent access to queues*. Each consumer agent request is granted the lowest cost item available in the queue at that time. A slot is freed for the winning producer agent. Consumers enter a wait state if the queue is empty. The `get_filter` function of the instrumented queue implementation always returns `true`.

3. *Prioritising queue entries by their total cumulative production cost with a temperature-dependent valuation error*. Producer agents append a resource usage field on each

item they put in the queue. The overall execution cost, C, is the sum of all individual unit costs (Equation 6.3).

$$C = \sum_j \|\mathbf{v}_j\| \tag{6.3}$$

Where $\|\mathbf{v}_j\|$ is the cost of the item with index j. which is a function of its measured resource usage vector $\mathbf{v}_j$ associated with its computation. The unit cost is a norm on the space of resource usage vectors, specifically a weighted Taxicab or Manhattan norm with user-defined weights w_i (Equation 6.4),

$$\|\mathbf{v}_j\| = \sum_i w_i |v_{ij}| \tag{6.4}$$

for a resource usage vector $\mathbf{v}_j$ and a user defined weight vector $\mathbf{w}$. Therefore the cost function may be expressed as Equation 6.5.

$$C = \sum_j \|\mathbf{v}_j\| = \sum_j \sum_i w_i |v_{ij}| \tag{6.5}$$

In a typical application, relevant measurable parameters that constitute components of $\mathbf{v}_j$ may include CPU time, memory, I/O, network communication, energy consumption, latency and failure risk. Changing or evolving user preferences may be reflected by reassigning the components of the weight vector $\mathbf{w}$. In a suitable dynamic runtime framework, the weight parameters should be capable of flexibly guiding execution towards or away from certain resources.

In this competitive auction setting, the higher-cost agents may never get an opportunity to participate in an exchange until the system approaches termination. However, given a dynamic execution environment such as a cloud platform with changing spot pricing or a cost function redefined by deliberate user action, the cost landscape may be subject to arbitrary variation.

We define the *temperature* parameter σ^2 to introduce controlled randomness by increasing the valuation error at the auction queues. This allows sub-optimal producer agents an opportunity to win and re-evaluate their possibly improved positions. Given an item cost $\mu = \|\mathbf{v}_j\|$ and temperature σ^2, the *apparent cost* or *valuation*, returned by the instrumented queue function `hash`, becomes a random variable characterised by the Gaussian probability density function (Equation 6.6) with the actual cost as expectation value.

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{6.6}$$

The temperature parameter allows for exploration and facilitates convergence towards stable equilibria rather than metastable states. It is based on analogous statistical mechanical concepts in condensed matter physics that have been applied in econophysics and simulated annealing [24, 1].

6.2.2 SCATTER *Pipeline Variant*

The pipeline variant of SCATTER in Chapter 5 integrates alternative GPU kernels using intermediate queues to decouple the subroutines that require CPU and GPU resources. The kernel validation applications presented in Subsection 6.2.1 are simplified versions of this pipeline that are implemented without the practical considerations and constraints imposed by the existing procedural code structure of SCATTER. However, as is commonly the case with scientific codes, parallel constructs are often added after the performance of a sequential version has proven to be inadequate and significant architectural choices have already been made. Although, these parallel adaptations may be systematically implemented in conformance with established patterns [89], in general, they do not map neatly to the **FARM** and **PIPE** constructs that have so far been considered.

Therefore, it is necessary to consider alternative methods by which this adaptive framework may be integrated into the existing application without recourse to a complete re-implementation. To achieve this, a loop-switch transformation (Listing 6.1) is introduced using surrogate queues for the outer **FARM**. The stage operations A, B and C are performed on the data from the actual queues $queue_1$ and $queue_2$. Where get_i and put_i represent dequeue and enqueue operations on queue $queue_i$, this alternative form is represented by (Figure 6.12 on page 107):

$$\begin{aligned} \textbf{FOR}(P, n) &\rightarrow \textbf{FOR}(\textbf{SEQ}(A, B, C), n) \\ &\rightarrow \textbf{FARM}(\textbf{ANY}(\textbf{SEQ}(A, put_1), \textbf{SEQ}(get_1, B, put_2), \textbf{SEQ}(get_2, C))) \end{aligned}$$

In the SPMD pattern of SCATTER, the **FARM** is executed by a pool of native MPI processes and requires an out-of-process shared queue implementation. Although this loosely-coupled architecture incurs some performance overhead, this penalty may be justified if the ratio of computation to communication is sufficient to offset the penalty of interprocess communication. Furthermore, end-to-end tracking of issued and completed iteration indices at the pipeline endpoints allows fault tolerance through the isolation of local process or node failures. The application state itself may be "checkpointed" by persisting enqueued data to secondary storage.

Lyapunov Optimisation with Instrumented Queues

Lyapunov optimisation [92] is an approach to the stochastic optimal control of queueing systems that is based on the use of Lyapunov functions. Lyapunov functions are important for establishing stability in control systems and impose a measure by associating a scalar value with all possible states of the dynamical system.

$$f : \mathbb{Z}^n \rightarrow \mathbb{Q}$$

In the *Lyapunov drift* method, a Lyapunov function is selected such that less desirable states correspond to higher values. For an action a applied at time slot t where $L(t)$

Listing 6.1: Loop-switch transformation of procedural Fortran loop to decouple queues in SCATTER

```
!Original Loop
do i = 1, n

  !block A...

  !block B...

  !block C...

end do


!Transformed loop with intermediate queues
do

  call getnextaction(action)

  if (action .eq. PRODUCE) then

    !block A...
    call put_1

  else if (action .eq. SOLVE) then

    call get_1
    !block B...
    call put_2

  else if (action .eq. CONSUME) then

    call get_2
    !block C...

  else if (action .eq. EXIT_) then

    !exit the loop
    exit

  else

    !unknown action
    cycle

  end if

end do
```

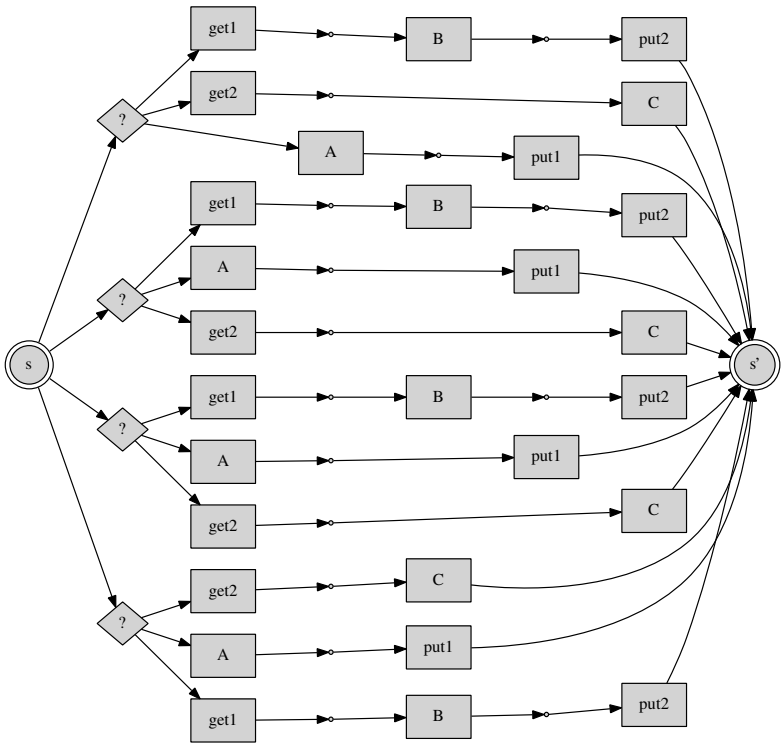


Figure 6.12: SPMD realisation of a farm pattern with surrogate queues

and $L(t+1)$ are the respective values of the Lyapunov function before and after the action, the Lyapunov drift is defined as

$$\Delta(t)_a = L(t+1) - L(t)$$

Decisions between the elements of the set of possible actions at time t are made to greedily minimise the Lyapunov drift. In queuing networks, this method allows optimisation of performance measures such as throughput or utilisation subject to stability and has been extensively applied in multi-hop packet switching networks [45, 118], concurrent databases and parallel processing [119].

The *drift-plus-penalty* algorithm is an extension of Lyapunov drift that allows minimisation of the time average of a penalty function subject to queue stability. Without a penalty function, drift-plus-penalty reduces to the backpressure algorithm (or max-weight algorithm) where the Lyapunov function f is defined as a quadratic function of queue backlogs q_i.

$$f(q) = \sum_i q_i^2$$

For selection of the next action in Listing 6.1, we consider the two-dimensional subspace of the queue vector space constituted by the input and output queues of the intermediate GPU stage in the pipeline. As discussed earlier, this triangular region is bounded by the constraints of available memory and non-negative integer-valued queue sizes. For each point in the state space, we define a Lyapunov function that is

a variation of the quadratic function of queue backlogs adopted in the backpressure algorithm [92].

$$f(\mathbf{q}) = \|\mathbf{q} - \overline{\mathbf{q}}\| = \sum_i (q_i - \overline{q}_i)^2$$

With the choice of a drift-minimal state at another point $\overline{\mathbf{q}}$ of the region (e.g. the centroid), this function takes into account considerations of minimum GPU kernel batchsize and avoidance of the undesirable boundary states.

6.3 EVALUATION

These experiments on the 4-node multi-GPU cluster Xookik (hardware specifications in Table 4.2 on page 44) are intended to demonstrate that:

1. queues provide sufficient information to guide runtime decision making towards improving performance.
2. queues provide a mechanism for achieving both decentralised and centralised coordination.
3. desired qualitative system behaviours may be achieved by suitable choice of the associated functions and parameters that capture extrafunctional user concerns.

6.3.1 *Simple Heuristic in the FastFlow Kernel Validation Pipeline*

For the FastFlow kernel validation pipeline, a performance evaluation was carried out on Xookik with 4 MPI processes, and 12 workers at each nested farm in the pipeline stages. 55,296 pseudorandom double precision floating point Hermitian test matrices of order 1024 were streamed through the pipeline to establish that the computed error is within acceptable tolerance.

Table 6.2: Active execution times on individual GPUs and Total Program Runtime on Test Cluster.

Resource	Runtime (seconds)
GPU1	8995.6
GPU2	8930.1
GPU3	8930.7
GPU4	8995.4
Total Cluster Runtime	9227.0

Table 6.2 indicates that while the total application runtime on the cluster was 9227 seconds (or 2.53 hours), all GPUs in the cluster were computationally active for a minimum of 8930 seconds (or 2.48 hours). This represents good overlap of execution for

both the CPU and GPU stages of the pipeline, validating the choice of this architectural style for maximising resource utilisation.

Figure 6.13a on page 110 presents memory usage over one hour of execution for the entire cluster. Set at 50% of the total 200 GB of physical memory available, memory usage remains within the pre-configured limits with new tasks injected into the pipeline when free memory rises above 100 GB and throttling enforced below 80 GB. It is evident that the memory demands of this application are substantial.

As expected, the GPU stage constitutes the primary bottleneck to the pipeline. Figure 6.13b on page 110 indicates that CPU usage varies between 30% and 60% following variations in the number of active processes.

6.3.2 *Decentralised Coordination in the Market-Based Kernel Validation Pipeline*

These experiments are intended to establish that the MBC approach achieves some degree of cost minimisation in a heterogeneous environment when compared to a naive random process allocation. In addition, they demonstrate behaviour in response to variations in the control parameters. As we assert that cost functions should be capable of capturing user preferences, we consider a common scenario in which the desire exists to maximise GPU utilisation in the heterogeneous Xookik cluster by choosing an appropriate cost function. In this case, the cost function adopted is simply the total CPU time in CPU-seconds.

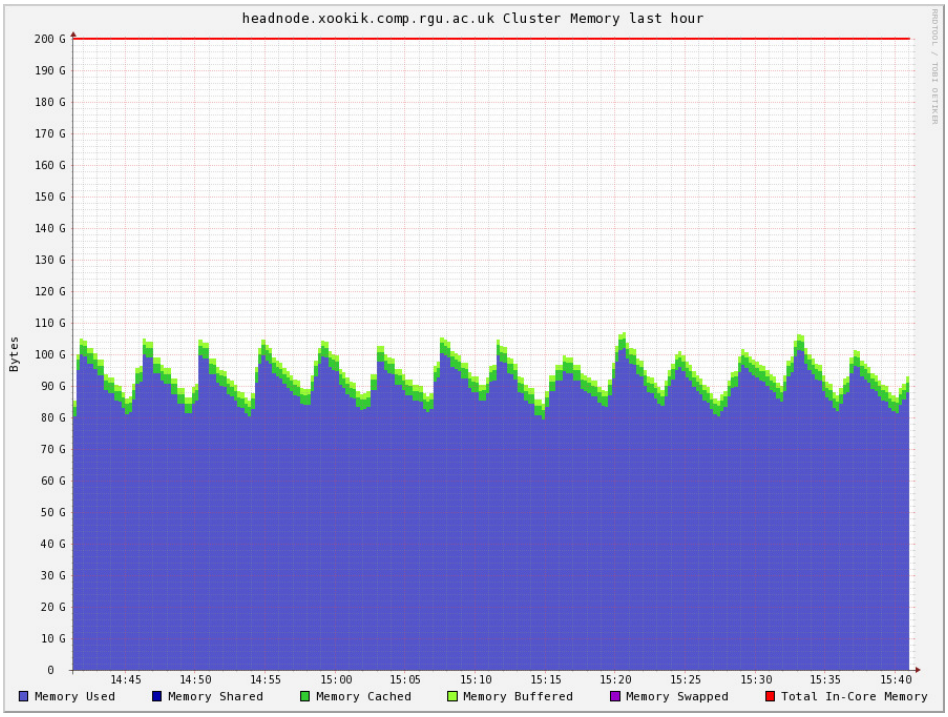
In the three-stage pipeline for eigensystem generation, solution and verification with inter-stage auction queues, a GPU-based alternative to the computational bottleneck solution stage is available from the MAGMA library of linear algebra routines [3]. We compare the cost of processing 2000 input matrices of order 2048 with the test application using naive FIFO queues, as in StreamIt [121, 63], against that obtained with the competitive auction mechanism on Xookik.

Subsequently, to demonstrate the response to variation of the control parameters – temperature and collection delay – 20,000 input matrices of order 384 are provided as inputs to the test application. The actual details of interest emerge from a statistical analysis of unit cost frequency distributions for larger numbers of smaller matrices with:

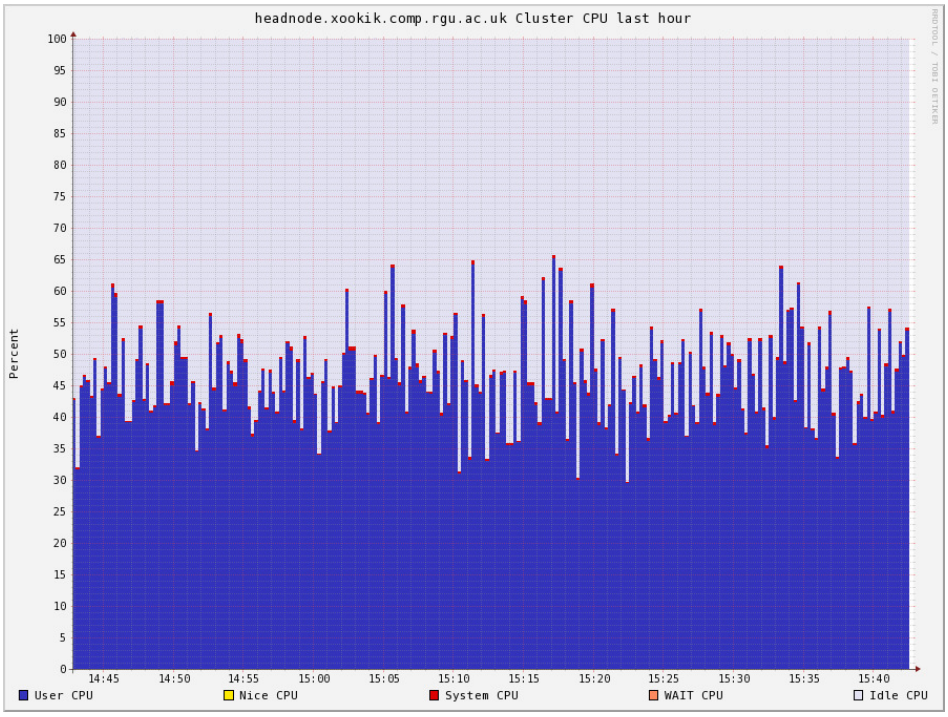
1. collection delays of 0.0, 0.001, 0.008, 0.01, 0.015, 0.02, 0.03 and 0.07
2. temperatures σ^2 of 1,2,4,8,16,64

Competitive Auctions

Figure 6.14 on page 111 compares the use of FIFO work queues, a classic load balancing approach for parallel pipelines [121, 63], against the MBC auction queues. The total costs for the entire execution are 110,278 CPU-seconds and 89,730 CPU-seconds respectively. This represents an 18.6% reduction in execution costs for the selected cost function when using the MBC framework.



(a) Overall memory usage over one hour of execution for all four cluster nodes. STOP_THRESHOLD and START_THRESHOLD are respectively 50% and 40% of the total 200GB available physical memory. The distinctive saw-tooth waveform follows form intermittent throttling of the pipeline.



(b) Overall cluster CPU usage percentage over one hour of execution. As the GPU stage constitutes the bottleneck, throttling adaptively imposes a limit on CPU usage, preventing pipeline queue overflows.

Figure 6.13: Overall cluster CPU and memory usage over one hour of execution for all four cluster nodes

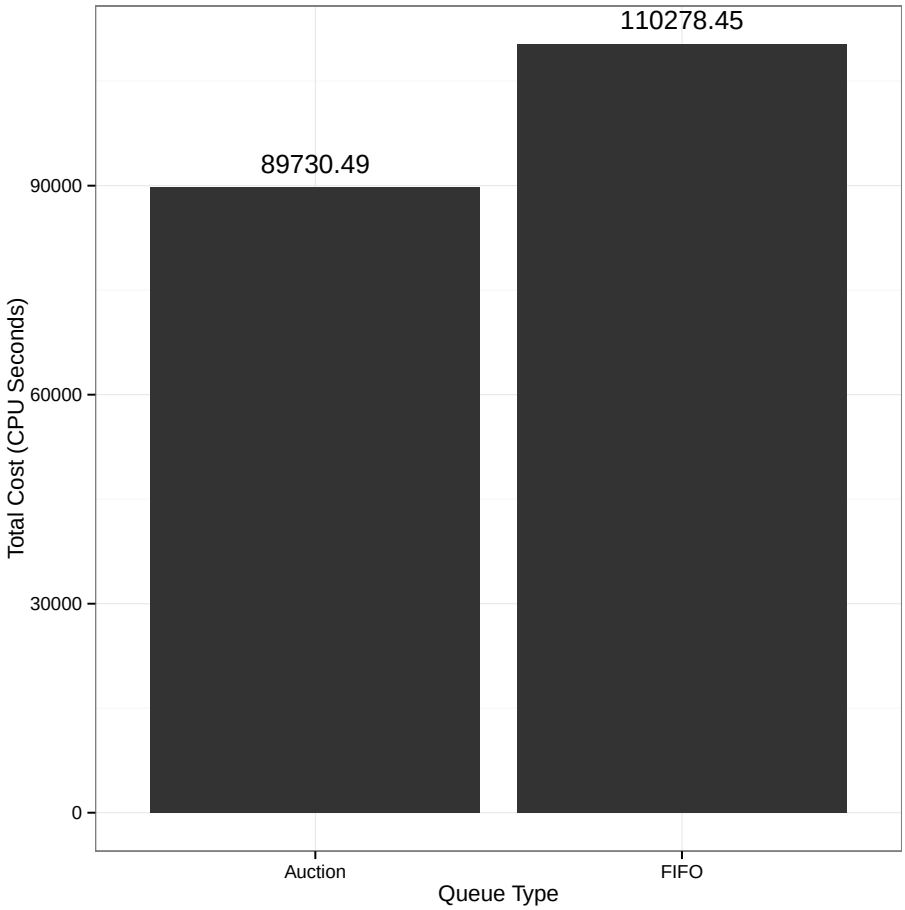


Figure 6.14: Total computation cost C for staged auctions and traditional FIFO queues

Clearly, the MBC technique is capable of lowering total costs. However, a natural question is whether the cost function captures the intended user preferences. In this case, a cost function has been chosen that penalises CPU usage in order to favour GPU resources when available. It should therefore be expected that as the GPU is more efficient at processing large matrices of this size, the overall throughput should be increased. For this run, the execution time was reduced from 2170 to 1909 seconds.

Control Parameters

Collection Delay

Without restricting the throughput of the pipeline, an opportunistic collector agent *impatiently* retrieves results as they become available. This corresponds to a collection delay of 0 at which a total cost of 9264 CPU-seconds was observed. Introducing the smallest collection delay of 0.001 causes a dramatic total cost reduction by 10% to 8319 CPU-seconds. Counter-intuitively, this delay *increases* overall pipeline throughput from 116 to 118 arrivals per second. Incorporating a deterministic delay between collections allows the framework sufficient time to generate new low-cost outputs and maintain the competitive pressure imposed by finite economic demand.

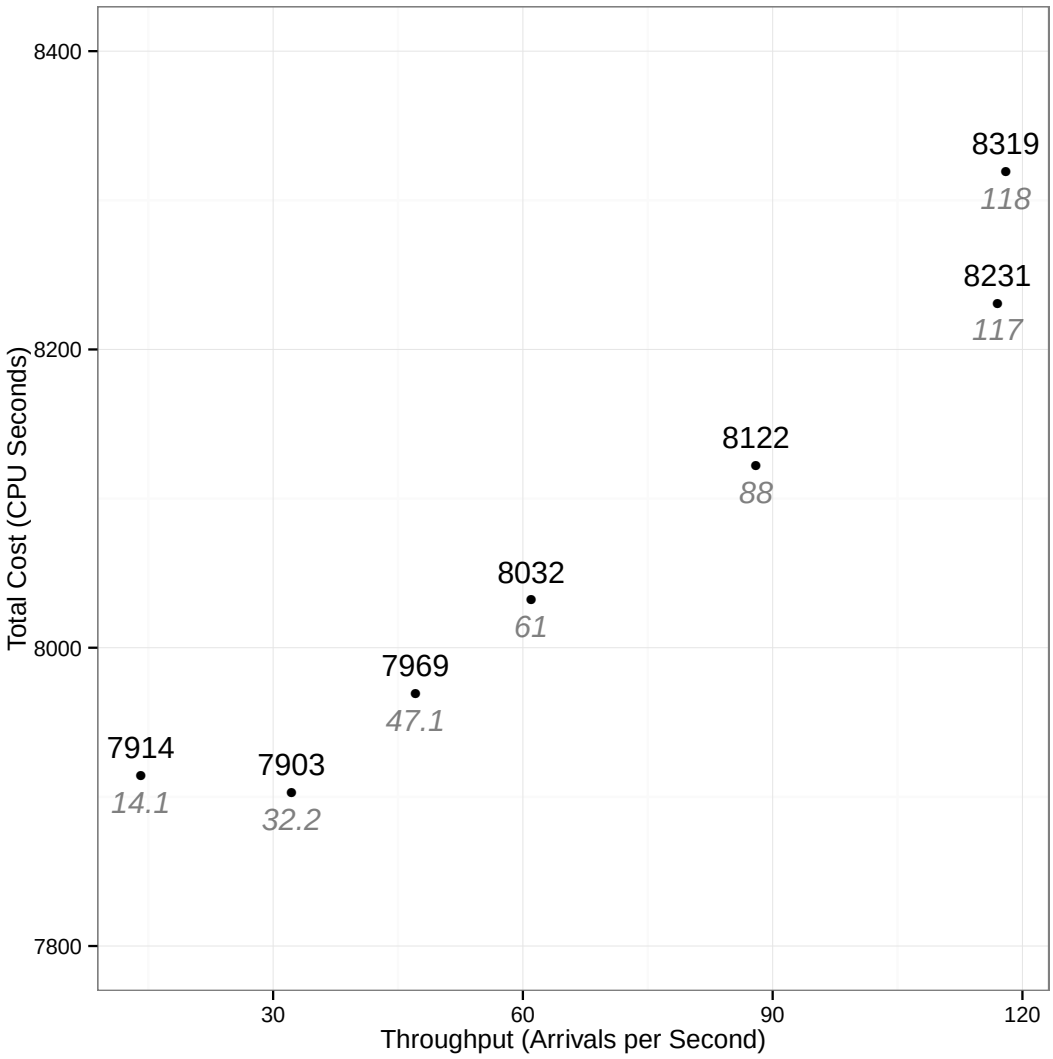


Figure 6.15: Throughput vs Cost Trade-off. Varying the collection rate at the collector agent. Starting at a peak throughput of 118 collections per second, there is a nearly steady decrease in total cost C with decreasing throughput.

Further increases in delay lowers throughput and incur progressively lower costs (Figure 6.15 on page 112). Therefore, the collection delay parameter allows trade-off of throughput for lower costs.

By the economic metaphor, this corresponds to a *Monopsony*, a market in which there exists a single buyer and multiple sellers. Monopsonies are a form of imperfect competition that may become exploitative. The monopsonistic collector agent indirectly controls the entire market by artificially controlling demand. Unlike real economic systems, exploitation of computational resources is a desirable objective of this framework

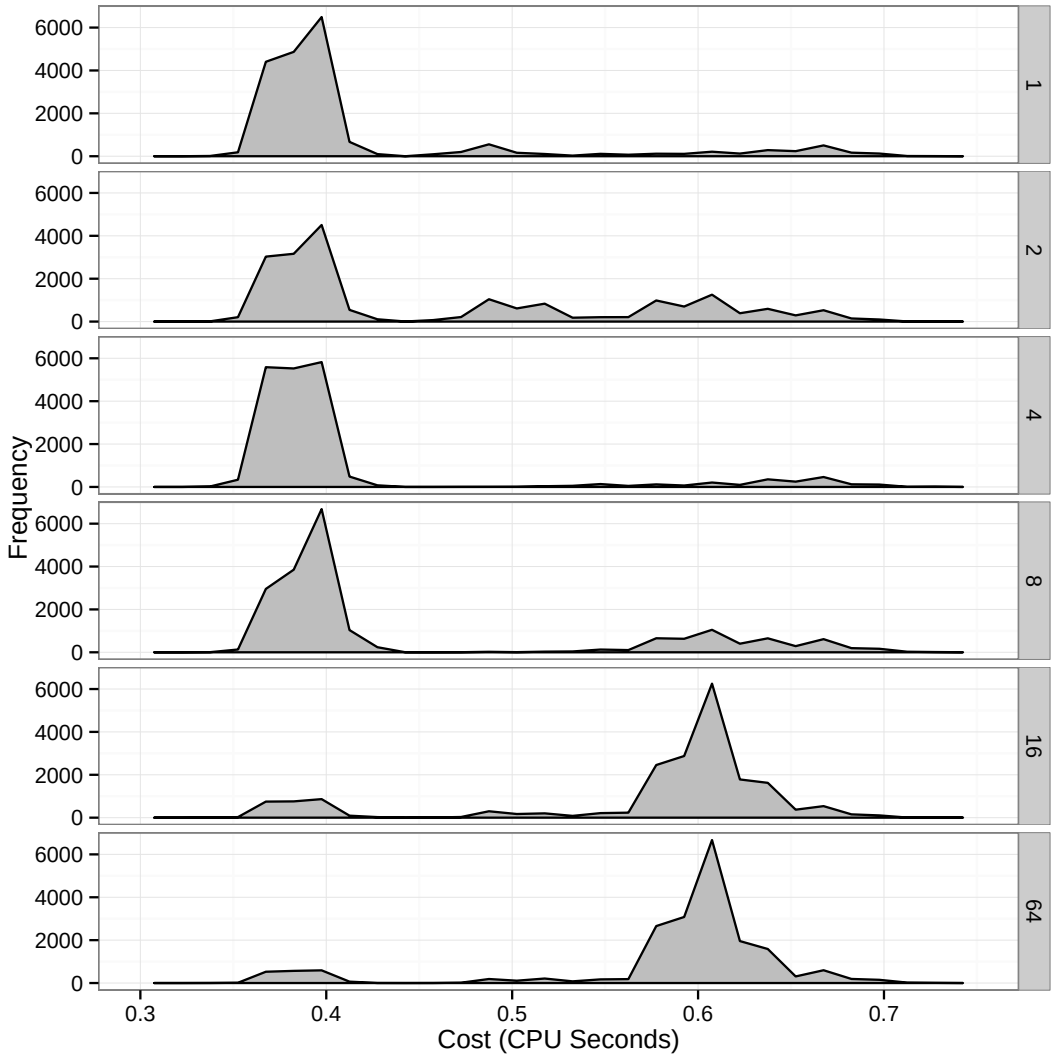


Figure 6.16: Cost Frequency Distribution by Temperature. The two prominent peaks correspond to GPU and CPU resources. At lower temperatures, the lower-cost GPU agents are favoured. At $\sigma = 16, 64$, distributions overlap as the queues are almost entirely random

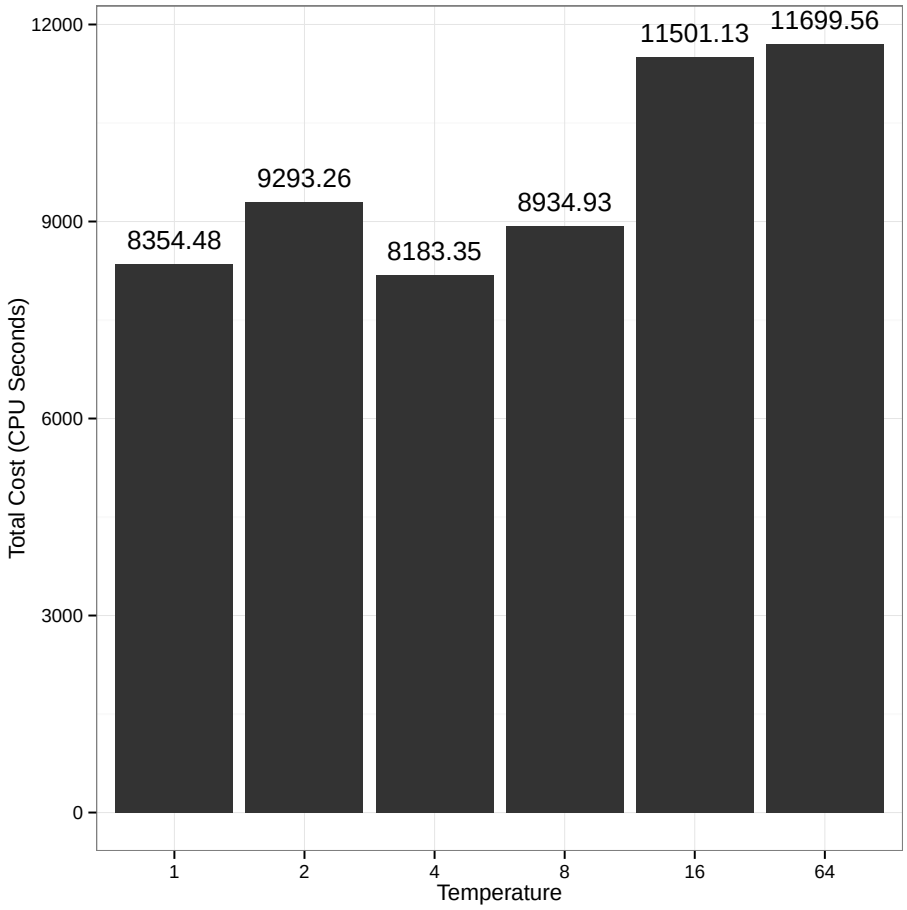


Figure 6.17: Total Cost C at different temperatures

	Model	Potential	Matrix Order	$\left(\frac{\lvert\mathbf{Q}\rvert_{max}-\lvert\mathbf{Q}\rvert_{min}}{\delta\lvert\mathbf{Q}\rvert}\right)\times\left(\frac{2\pi}{\delta\theta}\right)^2$
1	C_{60} Ambient Temperature Phase	Brenner (intramol) Lennard Jones (intermol)	180	
5	C_{60} Low Temperature Phase		720	

Table 6.3: Xookik Test Models

Temperature

Figure 6.16 on page 113 presents statistics for the second experiment. The unit costs of each item are accumulated for a given temperature into a histogram showing the item counts (y-axis) for each cost bin (x-axis).

The two prominent peaks at approx 0.38 and 0.6 correspond to the lower-cost GPU producer agents and the higher-cost CPU producer agents respectively. The smaller features likely represent complex interactions between the agents as they contend for processing resources. The MAGMA scheduler is a likely contributor to this behaviour.

At lower temperatures, the lower-cost GPU producer agents dominate the computation. As the temperatures rise, there is noticeable increase in the number of active CPU-agents. At higher temperatures, $\sigma^2 = 16$ and 64, there is almost no discernable change as the process appears to have become completely random and the CPU-agents dominate the computation.

Figure 6.17 on page 114 shows the total cost for each of these temperatures and demonstrates an overall trend towards increasing cost with rising temperatures. However, a slightly decreased cost at $\sigma^2 = 4$ is very likely the result of escaping premature convergence at pipeline start-up and allowing better exploration of the cost landscape during execution. This anomaly may point at the need to balance exploration and exploitation.

6.3.3 *Centralised Coordination in the* SCATTER *Pipeline Variant*

With the structured SCATTER pipeline variant based on a Redis queue implementation (section 5.2.2 on page 68), we evaluate models of the ambient and low temperature phases of C_{60} with 60 and 240 atom bases presenting dynamical matrices with 180 and 720 modes respectively (Table 6.3 on page 115). Both models are executed with:

1. an ad hoc scheduling policy based on random process pool worker allocation
2. the Lyapunov drift algorithm and target queue levels of $(400, 400)$ and $(100, 100)$ respectively for the input and output queues of the intermediate pipeline stage.

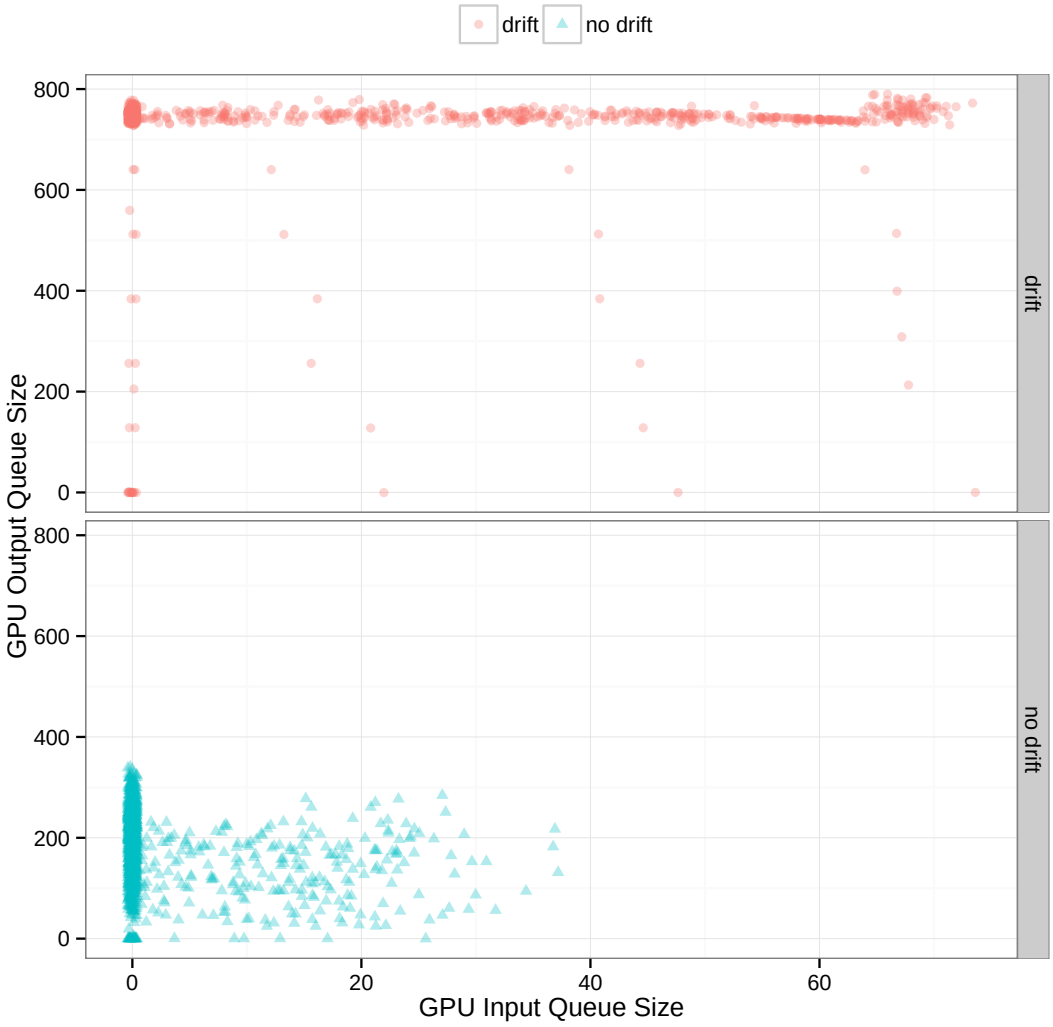


Figure 6.18: Queue space visualisations for the 180-mode ambient temperature model of C_{60} with Lyapunov drift (top) and with random allocation (bottom). The GPU stage is not the pipeline bottleneck.

These values are selected to represent a reasonable compromise between the required GPU kernel batch sizes and the limits of available memory on the cluster nodes.

Over the course of execution on the Xookik cluster, the input and output queue levels for the intermediate eigensolver stage, with both CPU and GPU implementations available, are sampled at random intervals. The models themselves are representative of the producer and consumer deficit scenarios (Subsubsection 6.1.2):

1. Producer Deficit: Figures 6.18 and 6.19 present the queue space visualisations and queue level histograms over both runs of the ambient temperature phase model of C_{60} with 180 modes. For this model, it is evident that the intermediate stage does not constitute the bottleneck to the pipeline as the matrix sizes are sufficiently small to make the generation of dynamical matrices on the CPUs the most computationally expensive operation.

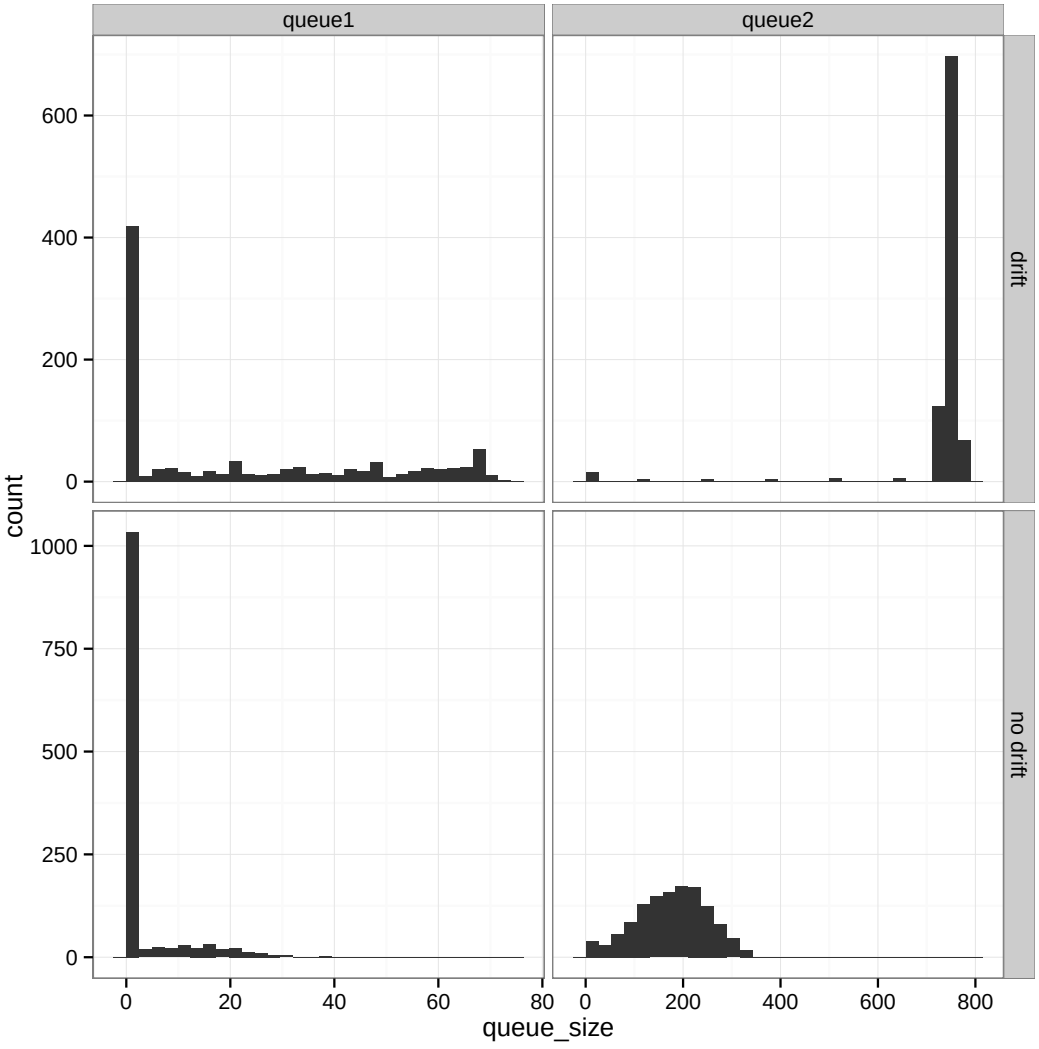


Figure 6.19: Queue level histograms for the 180-mode ambient temperature model of C_{60} with Lyapunov drift (top) and with random allocation (bottom). Lyapunov drift improves the availability of inputs for the input queue (queue1) of the non-bottleneck GPU stage.

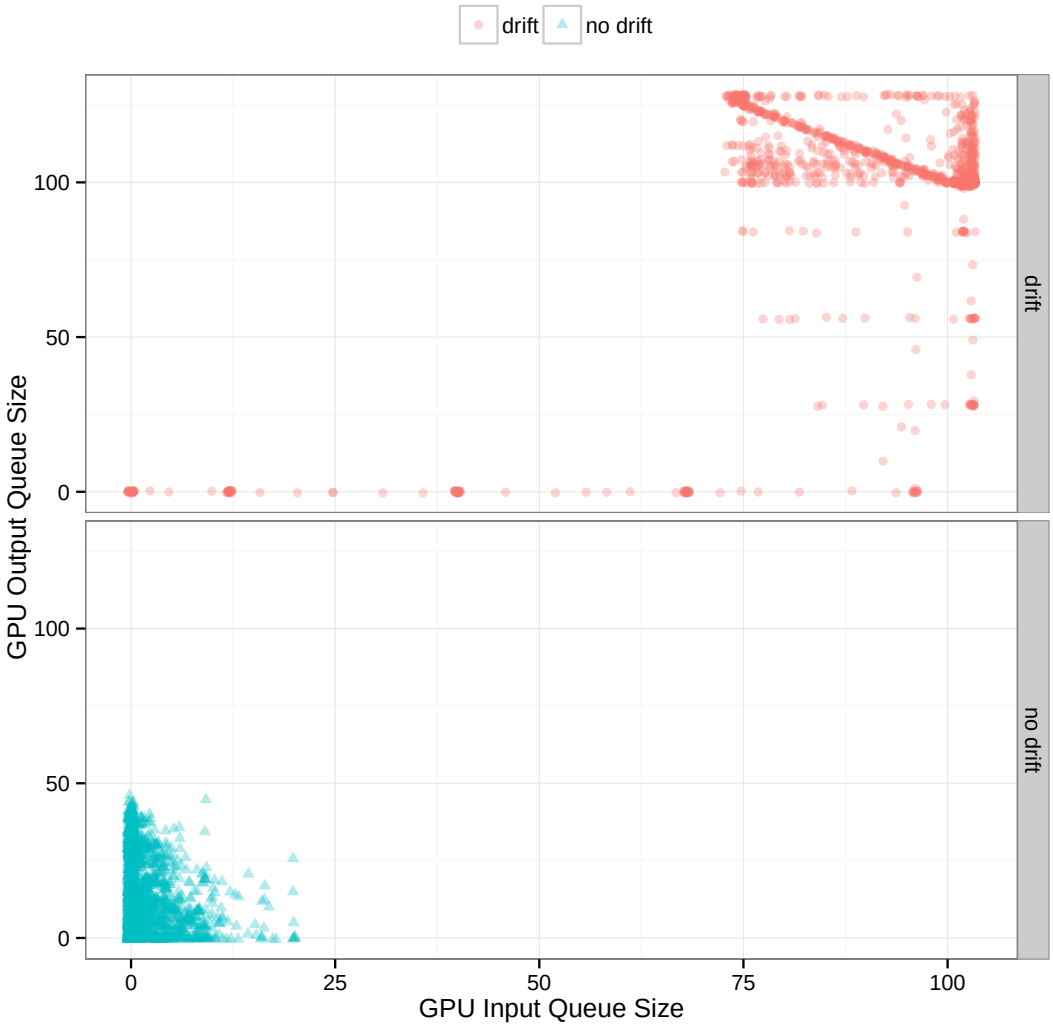


Figure 6.20: Queue space visualisations for the 720-mode low temperature model of C_{60} with Lyapunov drift (top) and with random allocation (bottom). The GPU stage is the pipeline bottleneck.

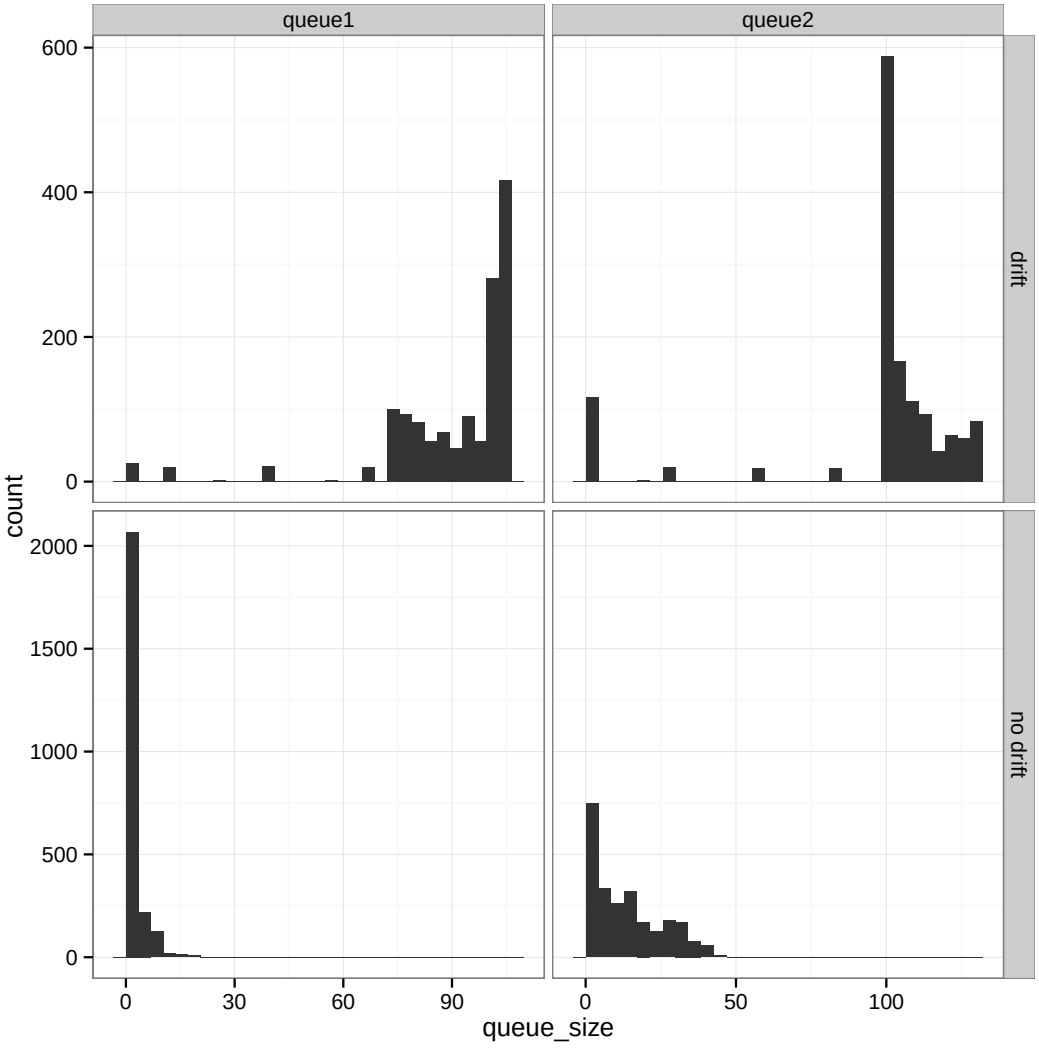


Figure 6.21: Queue level histograms for the 720-mode low temperature model of C_{60} with Lyapunov drift (top) and with random allocation (bottom). Lyapunov drift improves the availability of inputs for the input queue (queue1) of the non-bottleneck GPU stage.

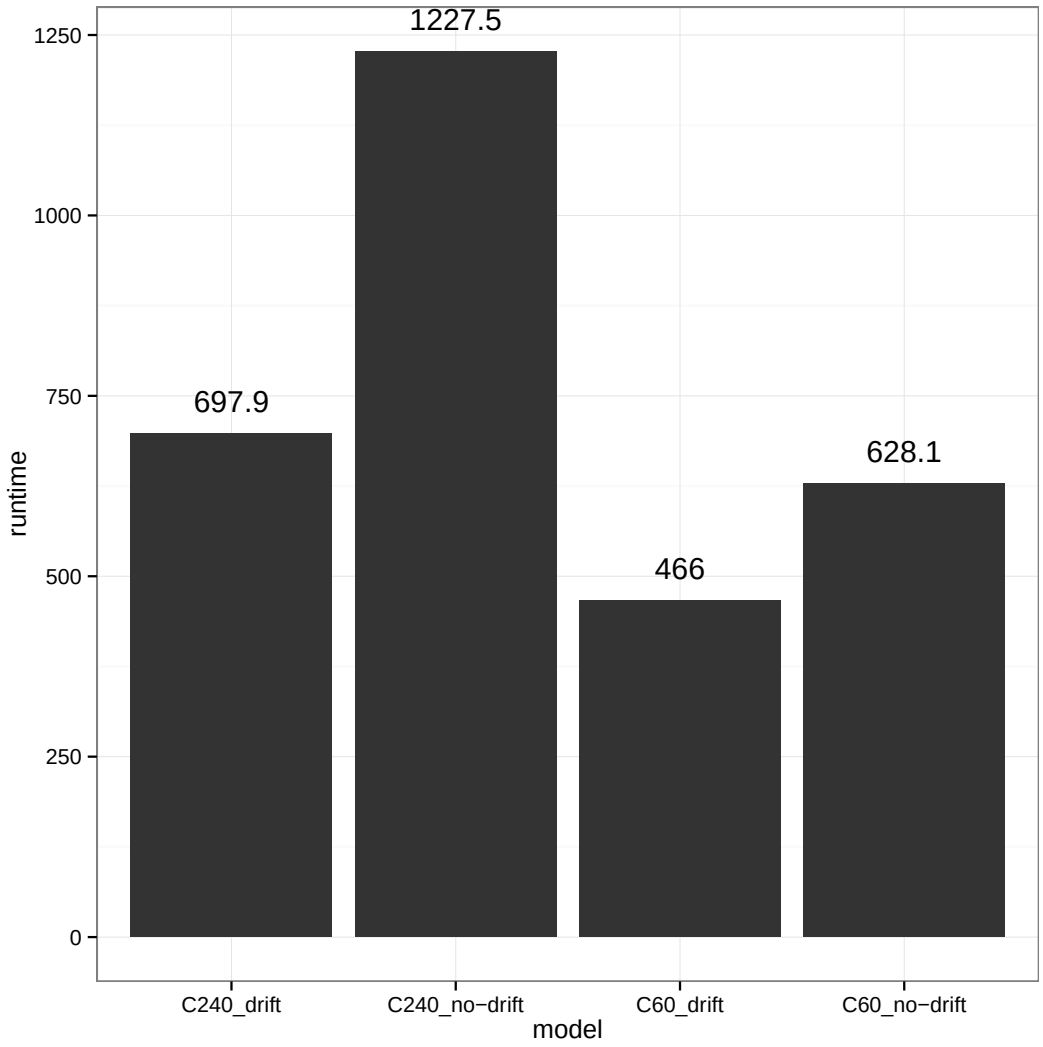


Figure 6.22: Runtimes of models of the ambient and low temperature phases of C_{60} with 60 and 240 atom bases. Lyapunov drift improves performance for both models.

2. Consumer Deficit: Figures 6.20 and 6.21 present the queue space visualisations and queue level histograms over both runs of the low-temperature phase model of C_{60} with 720 modes. For this model, the intermediate stage does constitute the bottleneck to the pipeline as the matrix sizes are sufficiently large to make the solution of the dynamical matric eigensystems the most computationally expensive operation.

Both scenarios demonstrate that Lyapunov drift has an effect on the queue levels over the course of execution. In the random allocation case, the queues exhibit a higher tendency towards the empty or (0,0) state. For the 180-mode model of the ambient temperature phase of C_{60}, the GPU input queue has a more even distribution with the Lyapunov drift algorithm. The algorithm becomes particularly effective for the model of the low-temperature phase of C_{60} with 720 modes where the queue state vector demonstrates a marked tendency to remain within the region of the state space demarcated by a discernible triangular boundary (Figure 6.20). This triangle, centered about the designated equilibrium position, is indicative of the relative magnitudes of the throughput vectors.

However, the intention of maintaining stable queues at the designated levels is to achieve improved throughput by improving utilisation in the batch-oriented kernels. Figure 6.22 presents the relative runtimes of each of these models. It may be observed that, even for the model of the ambient temperature phase of C_{60} where the algorithm is less effective as the intermediate stage is not the bottleneck, the application of Lyapunov drift nevertheless results in an overall performance gain. In the case of the model of the 720-mode low-temperature phase of C_{60}, the execution time is significantly lower when the Lyapunov drift algorithm is applied.

6.4 DISCUSSION

6.4.1 *Adaptation*

Approaches to centralised and decentralised coordination with Lyapunov drift and Market Based Control have been presented using information from instrumented queues to represent the instantaneous state of a structured parallel program. Limited runtime adaptation has been demonstrated for:

1. Environmental variation given a dynamic execution environment, with competing user and system processes and network throughput that is subject to stochastic variation.

2. Application-specific demands given the requirements of different models and matrix sizes. However, the collector interval parameter in the Market Based Framework may be considered as an application-specific requirement. Maintaining a collector interval value, even orders of magnitude less than the average inter-arrival time, allows a trade-off between throughput and cost.

However, greater consideration has been given to the extra-functional performance measures where the emphasis has been on the potential of controlling program behaviour by coupling a cost-minimising runtime with a dynamically re-definable cost function. A simple cost function captures a common non-functional requirement to favour a given resource class without modifications to the running program. In these instances, the expensive resource class (CPUs), defined by that function, remains inactive until there is sufficient demand to necessitate its use. As the runtime adapts to assume low cost configurations that minimise the cost function, suitable manipulation of the function parameters would permit a degree of online control that is unprecedented in loosely coupled distributed applications. Such features are usually only available when software engineers anticipate deployment environments and likely user concerns. In the largest applications, e.g. at web-scale, these engineers are often locked in a perpetual race to tune a single application instance, without downtime, by continuous integration [52] given a very dynamic set of priorities and costs that may reflect spikes in traffic, failures, changing energy tariffs, hardware prices and new architectural or regulatory directives.

Here, the parametric functions of the instrumented queue definition allow the realisation of classes of coordination approaches and raise the possibility of introducing *coordination skeletons* that are analogous to algorithmic skeletons. As algorithmic skeleton frameworks accept functions defining the parallel application, coordination skeletons may analogously capture broad classes of coordination approaches and be integrated into a skeleton framework to achieve behavioural programs.

6.4.2 *Qualitative Comparison of Coordination Methods*

Despite its efficacy, the simple heuristic applied to FastFlow is limited by the absence of information from the framework queuing layer. Whereas, both Lyapunov drift and the MBC heuristic derive feedback from instrumented queues and further use their features to impose control decisions, this implementation was intended to illustrate the utility of queuing information and has led to ongoing work towards the extraction of structured performance metrics from the queues and stages in FastFlow.

The centralised coordination model of Lyapunov drift is better suited to the traditionally predictable environments of high performance computing, characterised by high reliability, dedicated resources available to applications managed by batch schedulers, high-speed interconnects and generally consistent cluster-wide hardware specifications. However, this introduces a central scheduler to which all information must be gathered and from which control decisions must be relayed. Not only does this constitute a central point of failure, but the potential communication latency imposes limitations on feasible scalability over large-scale distributed systems.

In contrast, the loose coupling and absence of a central point of failure facilitates fault and partition tolerance in the decentralised MBC heuristic, making it better suited to large scale unpredictable environments such as cloud computing and volun-

teer computing. These environments are prone to failure and variation in the availability, capability or cost of processing and communication resources. Furthermore, the market based metaphor may arguably be extended to handle the inherent complexity in unstructured environments that cut across multiple domains hosted by different providers with potentially conflicting concerns and costs that are subject to variation by location and over time. However, as is evident from the counter-intuitive result where increased throughput follows the introduction of a collector delay, this is associated with the limited controllabilty arising from the tendency of market systems towards emergent behaviour and increased scheduling overhead.

7

CONCLUSIONS AND FUTURE WORK

As a case study in the application of heterogeneous CPU/GPU platforms to a large-scale problem in computational science, this thesis has deployed adaptive structured parallelism to lattice dynamical simulations of inelastic neutron scattering spectra. The performance and scalability of implementations using both Mattson's prescriptive pattern-oriented approach and structured parallelism have been considered. A framework for static and dynamic adaptation has been introduced and applied to accommodate existing and emerging computational accelerators, execution environments and extrafunctional user concerns that have been subject to the application requirements and constraints.

7.1 RESEARCH FINDINGS AND LIMITATIONS

Application of Mattson's design hierarchy, in conjunction with performance tools has led to a demonstrably scalable parallel implementation of the SCATTER code that is an improvement over the original ad hoc implementation. In the context of pure multicore and multinode systems, this application is an instance of the Monte Carlo or MapReduce dwarf. The scalability of this implementation is consistent with the expectations of this class of applications.

While abstraction and the construction of larger functional systems by the composition of verified components has historically provided the foundation of software engineering, here, we have demonstrated that an optimal choice between a set of structural alternatives does not exist. These results carry the much broader implication that software is not performance-composable. Restated alternatively, software performance does not always exhibit optimal substructure [33] and therefore larger applications cannot be optimised by the ad hoc greedy heuristic that has become common practice where components are individually optimised in libraries and subsequently integrated into programs after distribution. As functional modules are developed, any early decisions made without the benefit of complete information regarding all of the relevant runtime factors, component interactions and overall structural context represents a hidden compromise. Given this dependence, the only way to consistently achieve absolute optimal performance may be empirical optimisation in a specific runtime setting. However, this requires that black box components in libraries and modules expose options to a performance-optimising layer through a consistent interface.

Approaches to centralised and decentralised coordination with Lyapunov drift and Market Based Control have been presented using information from instrumented queues to represent the instantaneous state of a structured parallel program. Limited runtime adaptation has been demonstrated for:

1. Environmental variation given a dynamic execution environment, with competing user and system processes and network throughput that is subject to stochastic variation.

2. Application-specific demands given the requirements of different models and matrix sizes. However, the collector interval parameter in the Market Based Framework may be considered as an application-specific requirement. Maintaining a collector interval value, even orders of magnitude less than the average inter-arrival time, allows a trade-off between throughput and cost.

However, greater consideration has been given to the extra-functional performance measures where the emphasis has been on the potential of controlling program behaviour by coupling a cost-minimising runtime with a dynamically re-definable cost function. A simple cost function captures a common non-functional requirement to favour a given resource class without modifications to the running program. These implementations introduce overhead into the test application with the implication that a trade-off is necessary between performance and adaptability.

Real programs are generally not idealised compositions of stateless functions. This creates significant challenges for integration after the fact. Thus we conclude that static adaptation may be better suited to traditional high performance computing applications as there is no overhead, resources are usually predictable and controlled. Dynamic adaptation in frameworks imposes overhead that make it better suited to applications running in unpredictable environments or where the performance objectives are subject to variation.

Optimised libraries such as MAGMA are an effective way to exploit heterogeneous resources. However, their construction and integration with applications may be improved with the use of constructs such as deferred choice to encapsulate the possible structural program variants and the deployment of structural autotuners for the range of extrafunctional performance measures that will become important in the future.

7.2 PRACTICAL APPLICATIONS AND IMPLICATIONS

The abstractions developed for this application provide simple mechanisms for representing alternative program structures and may be extended to other parallel frameworks and applications. The representation of structural alternatives with deferred choice also allow the development of high performance libraries that avoid the limitations of premature design choices and are able to adapt to the changing landscape of computing with new architectures, platforms and increasingly sophisticated extrafunctional user concerns.

From the application perspective, the work outlined in this thesis represents a fundamental contribution to the development of a new analysis method for spectroscopic data from PolyCINS experiments from powder materials. The high performance implementation of SCATTER presented makes it possible to model significantly larger

systems than was previously practical – a critical contribution to the computational feasibility of Roach's new analysis method. With the availability of the simulation data in conjunction with the visual presentation and analysis of the large datasets generated in PREFIT and Paraview (Appendix C), well understood materials may be revisited for additional insights into their physical properties and new materials may be approached with an expanded analysis toolbox.

It is our experience that the size and complexity of these models rapidly outgrows the computational capabilities available, as the demands for enhanced resolution in nanomaterial characterisation increases. This reflects the strong demand not only for computational tools of this kind among materials researchers, but also for interdisciplinary collaboration between computer and material scientists to stimulate the development of the emerging field of computational materials science.

7.3 RECOMMENDATIONS FOR FURTHER RESEARCH

The wider potential of the cost-function approach for marshaling and orchestrating large geographically distributed applications cannot be overstated. Autonomic and self-adaptive computing is based on the realisation that software complexity is rapidly reaching the limits of human architects and the engineers responsible for their deployment, tuning and management. The control and design of emergent behaviour has been identified as a challenge at the heart of autonomic computing [75].

In particular, the following prospective research directions exist:

1. More *Applications* to real-world high throughput workloads.
2. *Dynamic fitness landscapes* that result from changing cost-function parameters at execution time or environmental factors such as competing processes or failures.
3. State estimation algorithms such as the Kalman filter may provide a better means of measuring the current program extra-functional state despite stochastic variation.
4. Further work may include on the market-based metaphor may include:
    a) *Improving Market Efficiency* by introducing speculators that provide a stabilising influence in markets by anticipating fluctuations in price levels and accelerating convergence towards equilibrium [117].
    b) *Alternative Auction Mechanisms* such as the Continuous Double Auction or the Vickrey-Clarke Grooves (VCG) mechanism that may exhibit faster convergence to equilibrium and may carry guarantees of allocative efficiency derived from economic theory[86].
    c) *Creating Learning Agents* that may use techniques such as reinforcement learning, genetic programming and direct reinforcement [91] to influence their market strategies and response to price signals. These agents would

compete as their survival would be contingent on obtaining profit. They may also attempt to exploit exposed 'black-box' parameters of the user-defined transformation functions in order to achieve some competitive advantage.

APPENDICES

A

DSL IMPLEMENTATION

```
import networkx as nx
from itertools import count

import matplotlib.pyplot as plt

#counter = count()
farm_size = 4

class Pattern(object):
    def __init__(self,*entries):
        self.entries = [entry if isinstance(entry,Pattern) else Seq(entry) for
            entry in entries]

    def __repr__(self):
        return self.__class__.__name__ + '(' + ', '.join([str(e) for e in self.
            entries]) + ')'

    def render(self,graph=None):
        if graph is None:
            graph = nx.MultiDiGraph()
            #graph.graph['graph'] = {"size":"20,20","rankdir":"LR","splines":"
                ortho"}
            graph.graph['graph'] = {"size":"5,5", "rankdir":"LR"}

            graph.graph['edge'] = {"arrowsize":"0.5"}
            graph.counter = count()
            graph.branch_counter = count()

        #inqueue = str(next(graph.counter))
        #outqueue = inqueue + "'"
        inqueue = 's'
        outqueue = 's\''

        graph.add_node(inqueue,shape='doublecircle',width=0.1)
        graph.add_node(outqueue,shape='doublecircle',width=0.1)


        self.render_subgraph(graph,inqueue,outqueue)
        return graph

    def render_subgraph(self, graph, inqueue, outqueue):
        raise NotImplemented
```

```
class Seq(Pattern):

    def __init__(self,*entries):
        self.entries = entries

    def render_subgraph(self, graph, inqueue, outqueue):

        #if len(self.entries) == 1:
        #    graph.add_edge(inqueue,outqueue,label=str(self.entries[0]))
        #else:
        for entry in self.entries[:-1]:
            curr_out_point = 'seq' + str(next(graph.branch_counter))
            graph.add_node(curr_out_point,shape='point')

            if isinstance(entry, Pattern):
                entry.render_subgraph(graph, inqueue, curr_out_point)
            else:
                graph.add_edge(inqueue, curr_out_point, label=str(entry))
            inqueue = curr_out_point

        #connect the final stage
        if isinstance(self.entries[-1], Pattern):
            self.entries[-1].render_subgraph(graph, inqueue, outqueue)
        else:
            graph.add_edge(inqueue, outqueue, label=str(self.entries[-1]))

    def __repr__(self):
        return ''.join([str(e) for e in self.entries])




class Any(Pattern):
    '''Describes alternatives'''
    def render_subgraph(self, graph, inqueue, outqueue):

        if len(self.entries) == 1:
            self.entries[0].render_subgraph(graph, inqueue, outqueue)
        else:

            branch_out = 'any' + str(next(graph.branch_counter))
            branch_in = 'any' + str(next(graph.branch_counter))

            graph.add_edge(inqueue,branch_out)
            graph.add_edge(branch_in,outqueue)
```

```
            graph.add_node(branch_out,shape='point')
            graph.add_node(branch_in,shape='point')

            for entry in self.entries:
                entry.render_subgraph(graph, branch_out, branch_in)


class Farm(Pattern):
    '''Takes a single argument'''
    def __init__(self, entry):
        Pattern.__init__(self, entry)

    def render_subgraph(self, graph, inqueue, outqueue):

        for _ in range(farm_size):
            for entry in self.entries:
                entry.render_subgraph(graph, inqueue, outqueue)



class Pipe(Pattern):

    def __init__(self,*entries):

        if len(entries) < 2:
            raise ValueError('A pipeline must have more than one stage')

        Pattern.__init__(self,*entries)

    def render_subgraph(self, graph, inqueue, outqueue):

        for entry in self.entries[:-1]:
            curr_out_queue = str(next(graph.counter))
            graph.add_node(curr_out_queue,shape='circle',width=0.1)

            entry.render_subgraph(graph, inqueue, curr_out_queue)
            inqueue = curr_out_queue


        #connect the final stage
        self.entries[-1].render_subgraph(graph, inqueue, outqueue)
```

B

Scatter EXECUTION PROFILE

```
index % time    self  children    called     name
                0.00  171.56       1/1           main [2]
[1]     99.1    0.00  171.56       1         MAIN__ [1]
                0.00  171.56       1/1           gulpmain_ [3]
-----------------------------------------------
                                                  <spontaneous>
[2]     99.1    0.00  171.56                 main [2]
                0.00  171.56       1/1           MAIN__ [1]
-----------------------------------------------
                0.00  171.56       1/1           MAIN__ [1]
[3]     99.1    0.00  171.56       1         gulpmain_ [3]
                0.00  171.49       1/1           options_ [4]
-----------------------------------------------
                0.00  171.49       1/1           gulpmain_ [3]
[4]     99.0    0.00  171.49       1         options_ [4]
                0.00  171.49       1/1           optim_ [5]
-----------------------------------------------
                0.00  171.49       1/1           options_ [4]
[5]     99.0    0.00  171.49       1         optim_ [5]
                0.07  171.39       1/1           scatter_ [6]
-----------------------------------------------
                0.07  171.39       1/1           optim_ [5]
[6]     99.0    0.07  171.39       1         scatter_ [6]
                0.01  130.64    4001/4002        changemaxscat_ [7]
                1.02   37.58    4000/4001        phonon_ [9]
-----------------------------------------------
                0.00    0.03       1/4002        initmemory_ [43]
                0.01  130.64    4001/4002        scatter_ [6]
[7]     75.5    0.01  130.67    4002          changemaxscat_ [7]
              130.32    0.00   20010/20016       __reallocate_MOD_realloc_r8_4 [8]
 -----------------------------------------------
                0.01    0.00       1/20016       changemaxcfg_ [91]
                0.01    0.00       1/20016       changemaxreaxffval3_ [92]
                0.03    0.00       4/20016       changemaxreaxffspec_ [62]
              130.32    0.00   20010/20016       changemaxscat_ [7]
[8]     75.3  130.35    0.00   20016         __reallocate_MOD_realloc_r8_4 [8]
-----------------------------------------------
                0.00    0.01       1/4001        optim_ [5]
                1.02   37.58    4000/4001        scatter_ [6]
[9]     22.3    1.02   37.59    4001          phonon_ [9]
                0.19   29.56   80001/80001       dynamic_ [10]
                0.26    7.08   72200/72200       pdiag_ [14]
 -----------------------------------------------
                0.19   29.56   80001/80001       phonon_ [9]
[10]    17.2    0.19   29.56   80001         dynamic_ [10]
                2.43   27.13   80001/80001       realp_ [11]
```

```
-----------------------------------------------
               2.43   27.13   80001/80001       dynamic_ [10]
[11]    17.1   2.43   27.13   80001         realp_ [11]
               0.05   20.30  160002/168006      changemaxdis_ [12]
               0.06    4.90  800010/840030      rsearch3d_ [17]
-----------------------------------------------
               0.00    1.02    8004/168006      reale_ [24]
               0.05   20.30  160002/168006      realp_ [11]
[12]    12.3   0.05   21.32  168006         changemaxdis_ [12]
              14.23    0.11 5040180/5040723     __reallocate_MOD_realloc_r8_1 [13]
               4.91    0.00  168006/180145      __reallocate_MOD_realloc_r8_2 [16]
-----------------------------------------------
[13]     8.3  14.23    0.11 5040723         __reallocate_MOD_realloc_r8_1 [13]
-----------------------------------------------
               0.26    7.08   72200/72200       phonon_ [9]
[14]     4.2   0.26    7.08   72200         pdiag_ [14]
               0.08    6.99   72200/72200       ch_ [15]
               0.01    0.00  144400/527671      cputime_ [73]
-----------------------------------------------
               0.08    6.99   72200/72200       pdiag_ [14]
[15]     4.1   0.08    6.99   72200         ch_ [15]
               1.65    2.26   72200/72200       tql2_ [19]
               1.58    0.00   72200/72200       htribk_ [25]
               1.36    0.14   72200/72200       htridi_ [26]
-----------------------------------------------
               4.91    0.00  168006/180145      changemaxdis_ [12]
[16]     3.0   5.26    0.00  180145         __reallocate_MOD_realloc_r8_2 [16]
-----------------------------------------------
```

Listing B.1: SCATTER execution profile with a simple test model

C

TOOLS FOR POLYCINS ANALYSIS

C.1 THE PREFIT TOOL

The PREFIT tool is a software interface to the novel poly-CINS analysis methodology that is based on the identification of prominent features, or *coherence edges*, in both the theoretical and experimental data that are coincident with highly symmetric crystalline orientations. Coherence edges satisfy the coherence criterion

$$|\tau - q(\omega)| < Q < |\tau + q(\omega)|$$

where τ is the reciprocal lattice vector and $q(\omega)$ is the nearest lattice vector and may be distinguishable in the spectra as sharp peaks followed by abrupt drops in intensity. PREFIT has been instrumental in the rapid development of this technique by automating tedious and error-prone aspects of the analysis process, interoperating with existing software through support of multiple data formats originating from theoretical simulations or instruments and seamless integration as a frontend to the high performance parallel SCATTER implementation.

Figure C.1 on page 138 is the main user interface of the program, illustrating a comparative analysis of a theoretical SCATTER Lennard Jones model for Aluminium compared with experimental data obtained from the MARI spectrometer.

Both standardised and ad hoc textual and binary files are supported in PREFIT (Table C.1 on page 139) that may provide simulation output, experimental spectra, instrument geometry descriptions and lattice configurations. Experimental data is typically provided in 'spe' or 'nxspe' (Nexus [34]) formats with energy and intensity values corresponding to individual detectors. For a detector located at an angle θ, the corresponding momentum transfer Q is given by the relationship of (C.1) where E_{init} is the incident energy of the neutrons, E_{final} is energy after the scattering event, E_{trans} is the energy transfered, m_n is the neutron mass and $\hbar$ is Planck's constant.

$$\begin{aligned} \frac{\hbar^2 Q^2}{2m_n} &= E_{init} + E_{final} - 2\cos\theta\sqrt{E_{init}E_{final}} \\ E_{final} &= E_{init} + E_{trans} \end{aligned}$$

$$Q = \sqrt{8.0655/16.7 * (2E_{init} - E_{trans} - 2\cos\theta\sqrt{E_{init}(E_{init} - E_{trans})}} \quad \text{(C.1)}$$

A preprocessing step allows data interpolation and the application of filtering and preprocessing operations that include intensity scaling, thresholding, edge detection, Gaussian convolution and other digital filter operations.

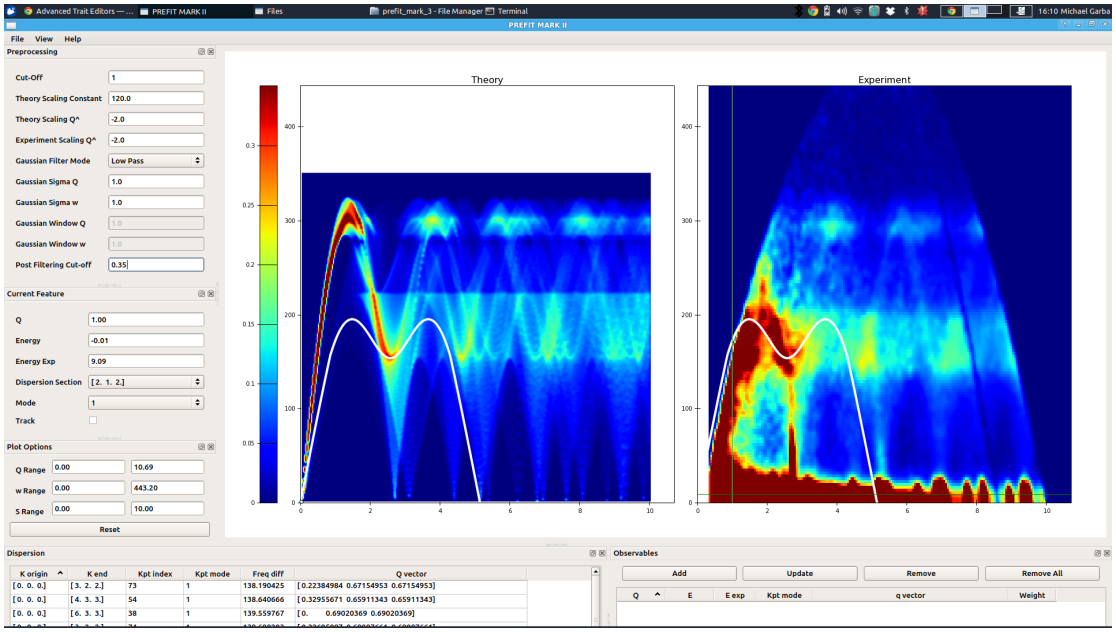


Figure C.1: The Prefit Application

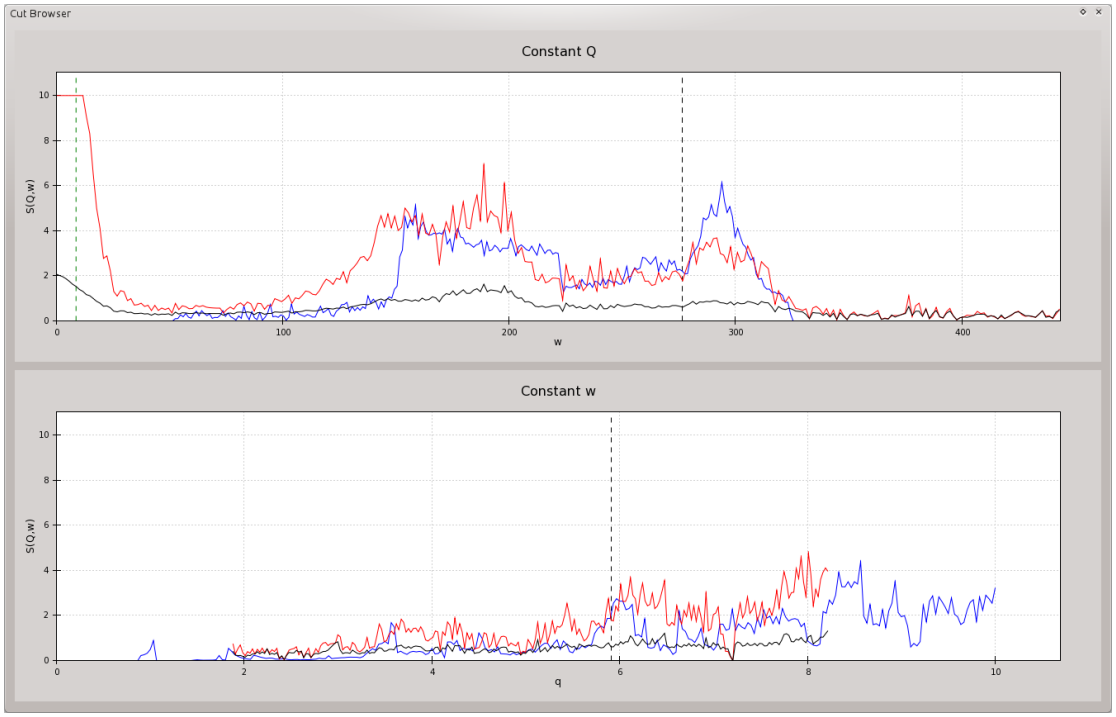


Figure C.2: Constant momentum transfer and constant energy transfer plots in $S(Q, \omega)$ in PREFIT.

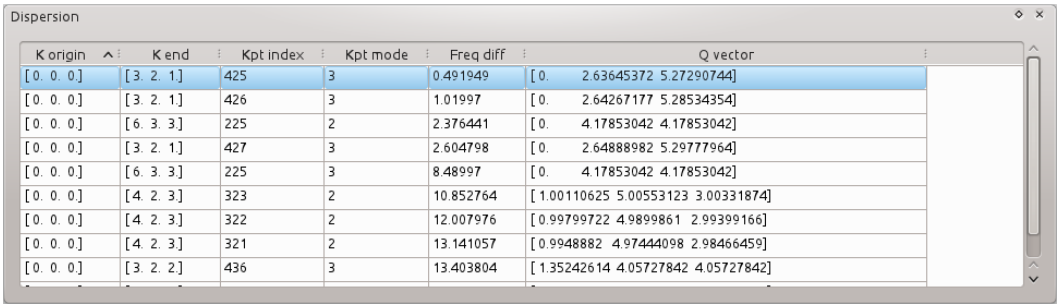


Figure C.3: Nearest dispersion matches for a given feature and their corresponding distances (Freq diff).

	Extension	File type
Theoretical Data	sqw	textual
Theoretical Data	xmf, hdf5	binary
Experimental Data	spe	textual
Experimental Data	nxspe	binary (HDF5)
Dispersion Data	disp	textual
real space lattice vectors	lrv	textual
Instrument Detector Geometry	phx	textual

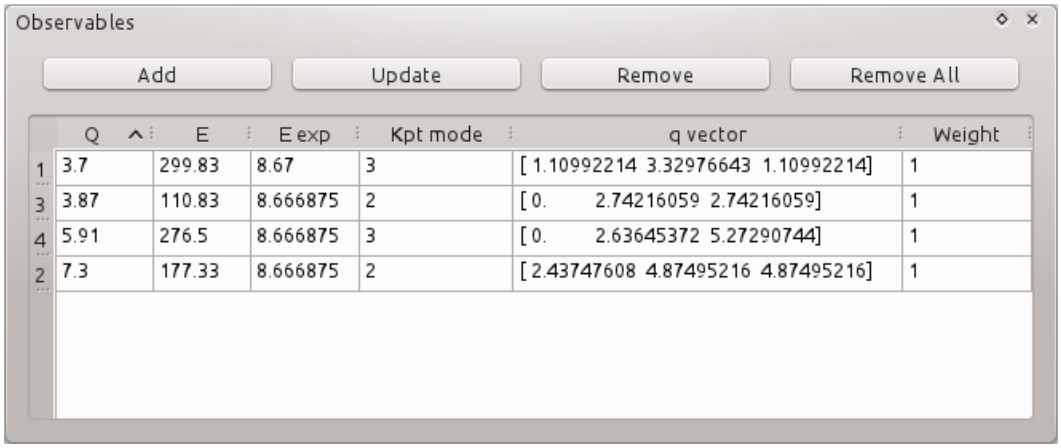
Table C.1: File formats supported by PREFIT

As the basis of the polyCINS fitting method, the identification of coherence edges is possible by inspection of cuts of varying width taken along horizontal (constant-ω or frequency) and vertical (constant-Q or momentum transfer) directions (Figure C.2 on page 138). An analyst seeks coherence edges that coincide with the dispersion curves along high-symmetry directions in the lattice that are also identifiable in the experimental data. For this purpose, PREFIT lists and overlays the nearest dispersion curves from a set of provided high-symmetry directions and their distances from the current feature (Figure C.3 on page 139).

Selected features that are distinguishable in both the theoretical and experimental data and have been determined, with high probability, to be the contribution of a high-symmetry direction may be added to a list of observables (Figure C.4 on page 140) that is subsequently used to generate a fitting input file for GULP based on a user specified template (Figure C.5 on page 140).

C.2 HIGH PERFORMANCE VISUALISATION IN PARAVIEW

Athough, data obtained from a polyCINS experiment is a polycrystalline-averaged representation of the scattering contributions arising from all orientations of a lattice,



	Q	E	E exp	Kpt mode	q vector	Weight
1	3.7	299.83	8.67	3	[1.10992214 3.32976643 1.10992214]	1
3	3.87	110.83	8.666875	2	[0. 2.74216059 2.74216059]	1
4	5.91	276.5	8.666875	3	[0. 2.63645372 5.27290744]	1
2	7.3	177.33	8.666875	2	[2.43747608 4.87495216 4.87495216]	1

Figure C.4: Selected features or observables and are ready to be passed as fitting parameters to the GULP package.

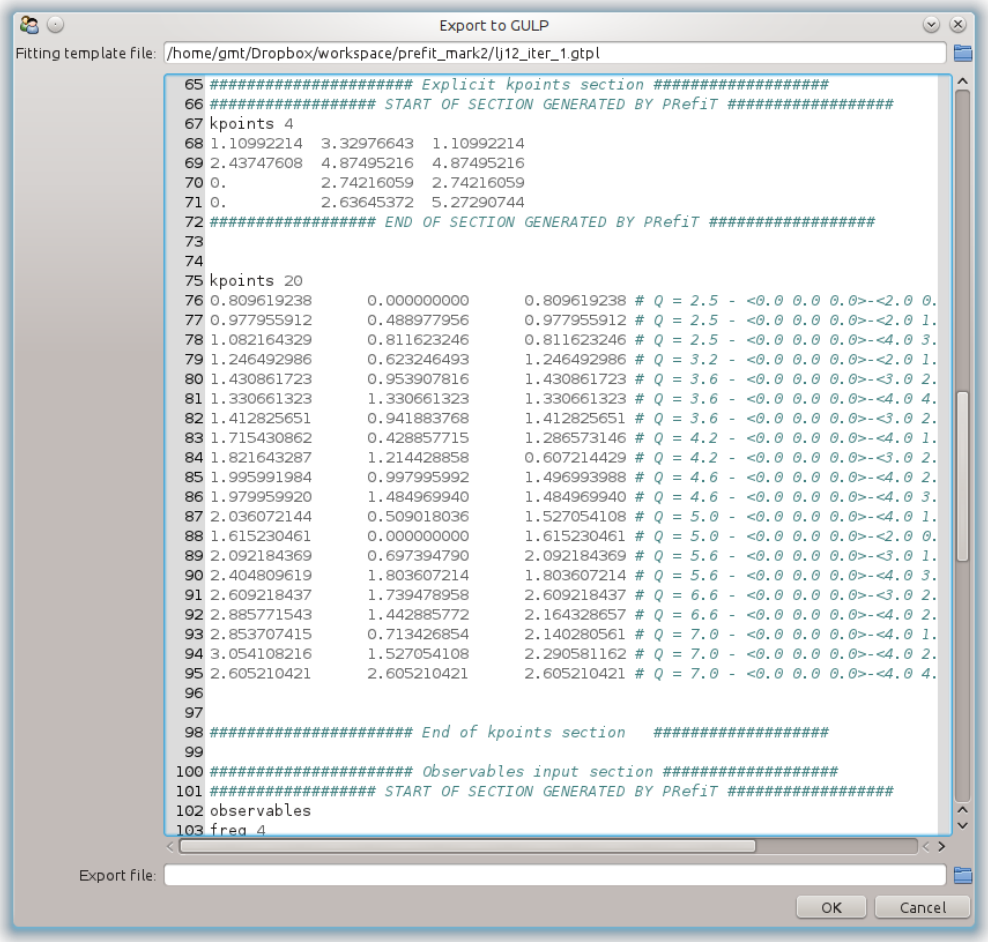


Figure C.5: GULP fitting input generated from observables.

the data obtained from simulation of theoretical models in SCATTER contains information about the predicted separate contributions of all points in RSO-sampled space. However, as millions of q-points are typically considered, a large dataset is generated over the course of simulation that, depending on the size of the lattice systems and RSO grid resolution, may range from multiple gigabytes to terabytes in size.

The data analysis and handling problems that emerge in a distributed parallel environment are best approached with high performance visualisation tools such as Paraview [25], the VTK frontend from Sandia National Laboratories, that is suited to the visual presentation of large remote and distributed datasets. Based on a pipeline architecture with abstractions for data sources and filter stages, the Visualisation Tookit (VTK) and Paraview are high performance visualisation tools that have been developed for demanding scientific applications [46]. Paraview's multi-tier architecture allows dynamic visualisations on a cluster of machines for real time parallel rendering alongside simulation.

XDMF [29] and HDF5 [50] provide light and heavy data storage formats for the simulation output of the parallel SCATTER implementation for their performance and space-efficiency [28], as well as interoperability with scientific packages such as Paraview itself. With native support for these formats as inputs to associated analysis pipelines, integrated visualisation from PREFIT is possible using Paraview's multi-tier architecture and scripting control interface.

The current applications of the visualisations in the analysis workflow include:

1. **Feature Analysis**
   Paraview allows novel insights into the origin and significance of features in the polycrystalline-averaged $S(Q, \omega)$ spectra by indicating lattice and phonon mode contributions to features of interest. The following are example visualisations for an Aluminium model with the Lennard-Jones potential being analysed in PREFIT:

   a) Figure C.6 on page 142 presents the scattering contributions in reciprocal space at a constant momentum transfer ($|Q| = 15.8 \pm 0.1$).

   b) The contributions of the three phonon modes and their frequencies at a constant momentum transfer ($|Q| = 15.8 \pm 0.1$) are presented in Figure C.7 on page 143.

   c) Figure C.8 on page 144 shows the contributing regions of the individual modes as complex surfaces for a specific feature corresponding to a single point in the $S(Q, \omega)$ output selected in PREFIT. The observed scattering intensity at that point is the sum of surface integrals of scattering intensities for each mode (Figure C.9 on page 145).

2. **Coherence Feature Identification**
   Paraview's large-scale data analysis capabilities enables automation of the identification of coherence features, previously a manual process, by locating lattice coordinates in reciprocal space that satisfy the coherence criterion as predicted

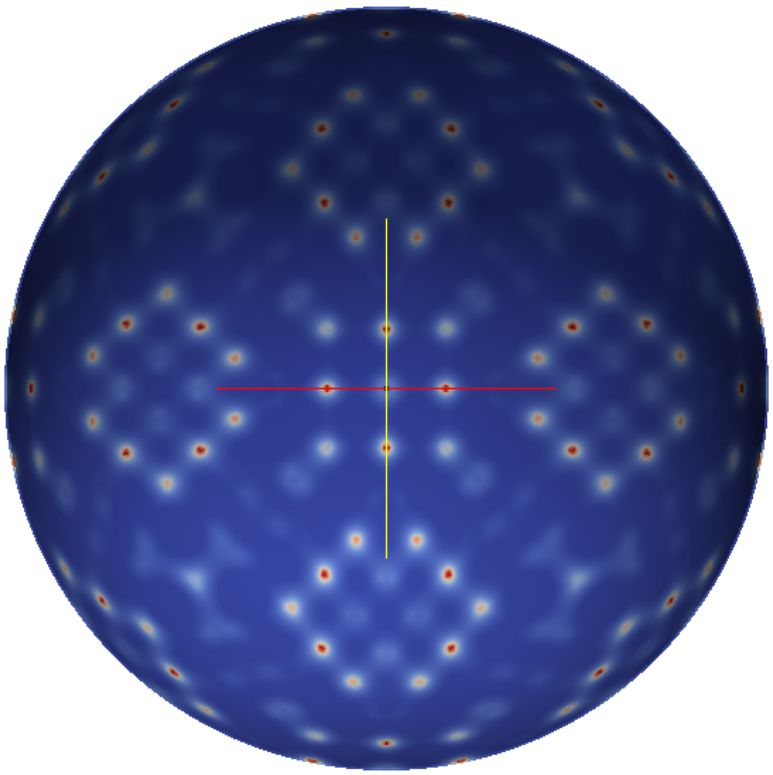
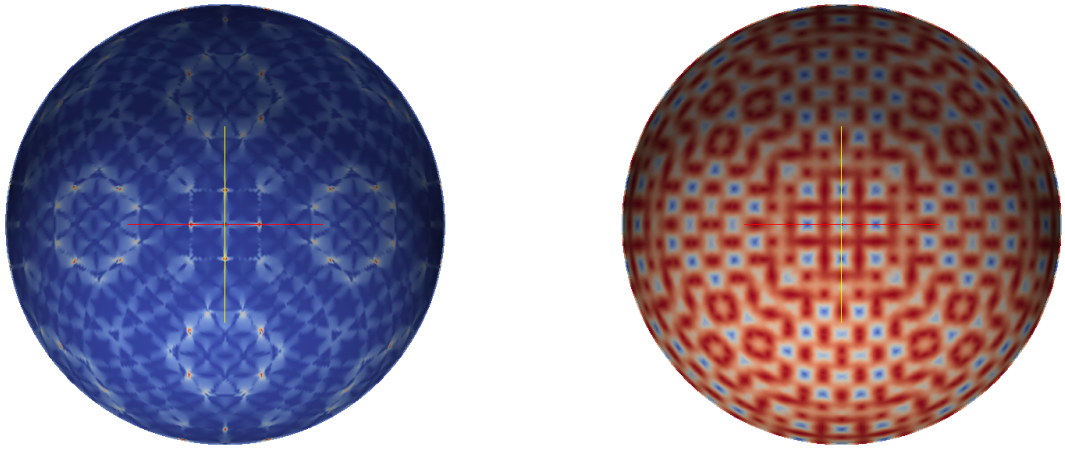


Figure C.6: Scattering contributions in reciprocal space, as visualised in Paraview, at a constant momentun transfer of $Q = 15.8 \pm 0.1$ for an Aluminium model being analysed in PREFIT.

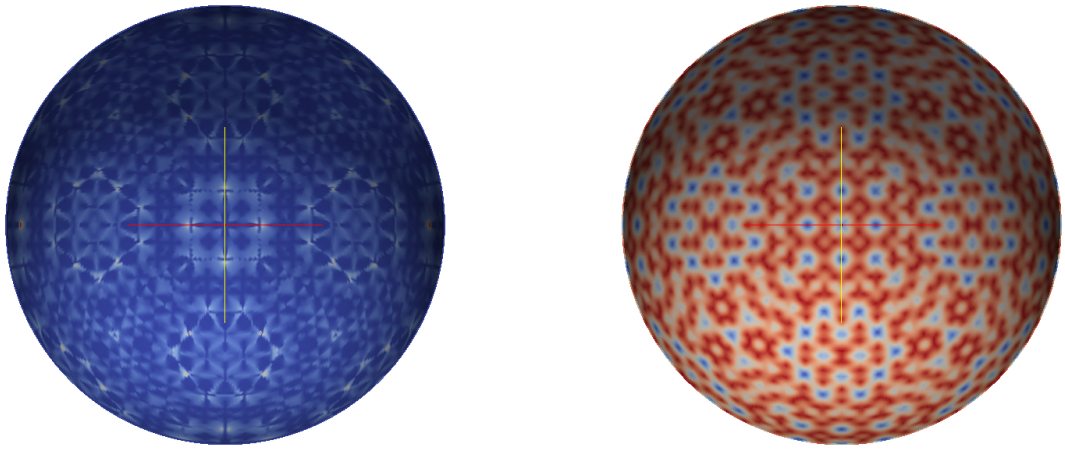
by simulation output. The following are example visualisations of this information:

a) Figure C.10 on page 146, Figure C.11 on page 147 and Figure C.12 on page 148 present predicted locations of features that satisfy the coherence criterion in the reciprocal space lattice based on analysis of the simulation output.

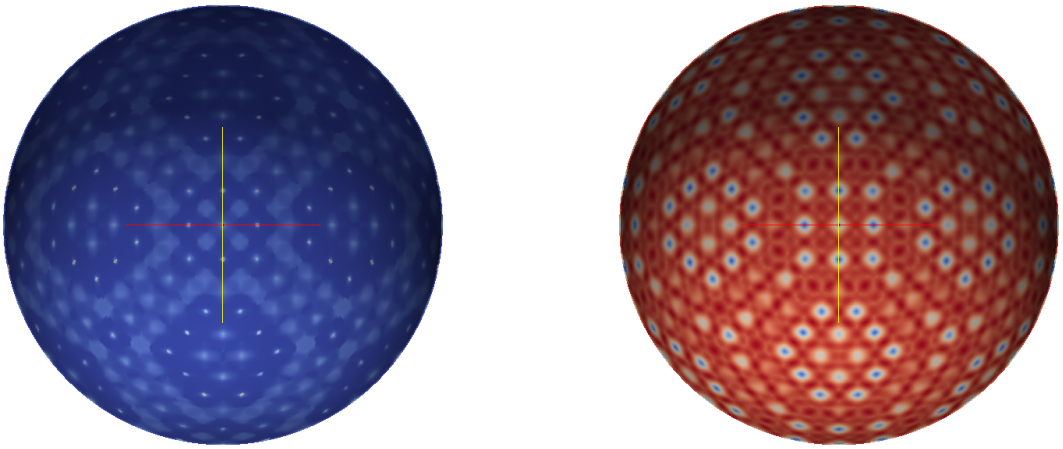
b) The predicted locations of the corresponding coherence edge in $S(Q, \omega)$ are presented in Figure C.13 on page 149.



(a) Mode 1



(b) Mode 2



(c) Mode3

Figure C.7: Scattering intensity contributions to C.6 by mode (left) with frequencies (right) for Aluminium at constant momentum transfer (Q = 15.8)

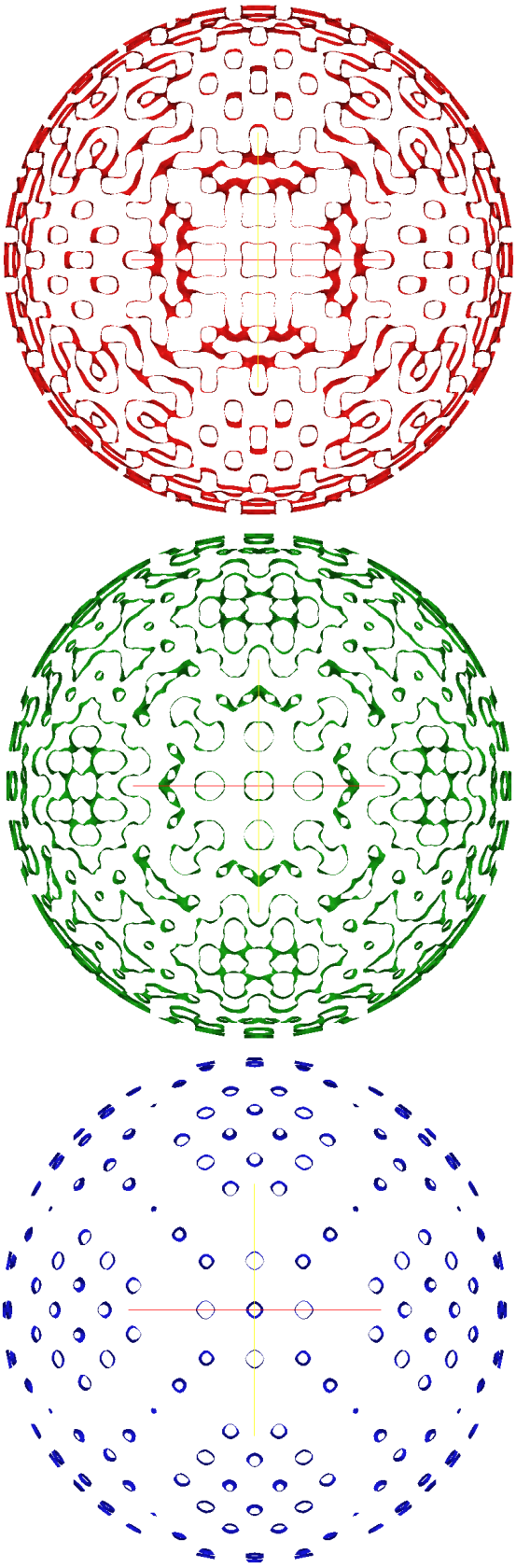


Figure C.8: Contributing regions to specific feature in PREFIT at $Q = 15.8 \pm 0.1$ and $\omega = 183 \pm 1.0$ by individual mode.

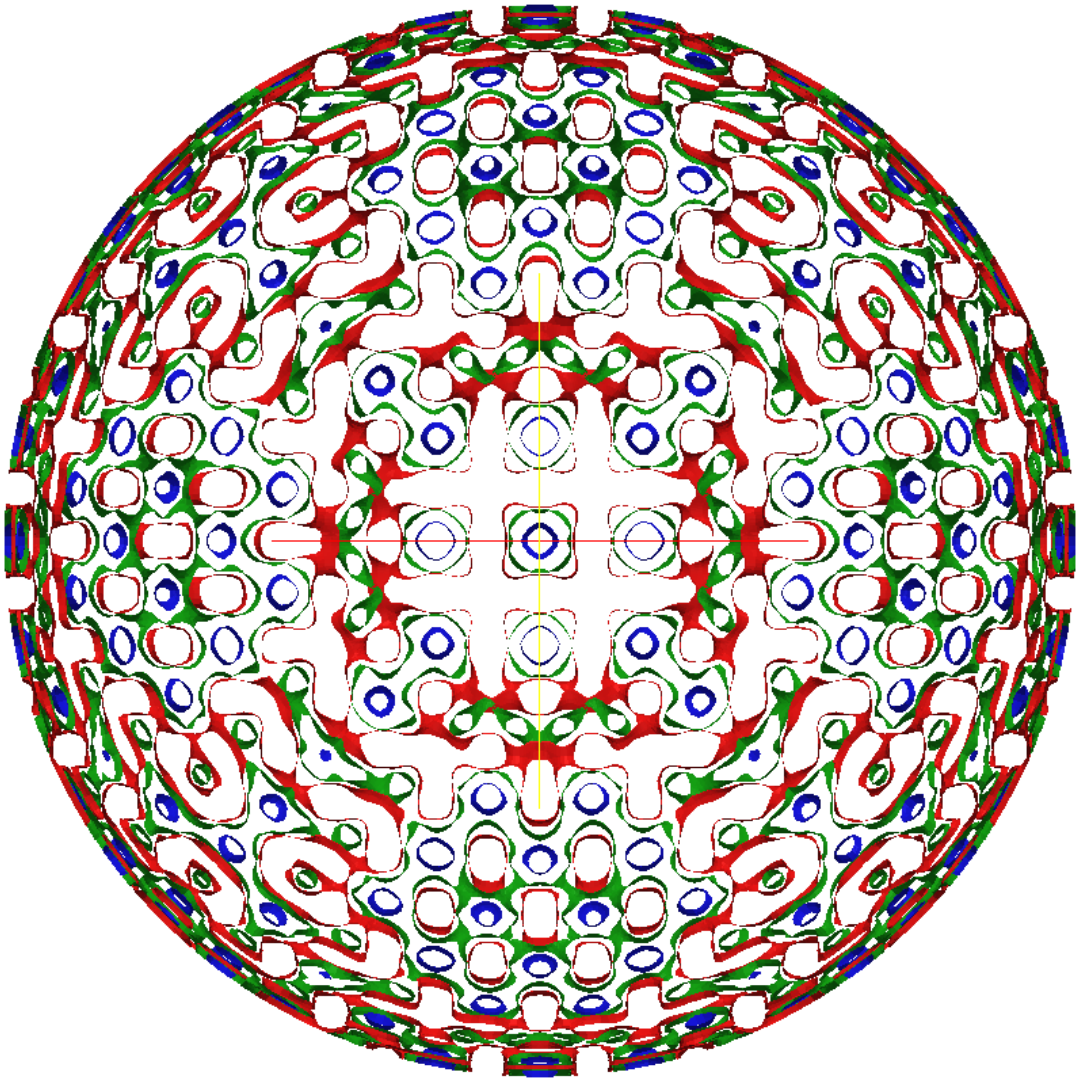


Figure C.9: Superimposed contributing regions to scattering intensity at constant momentum transfer ($Q = 15.8 \pm 0.1$) and frequency ($\omega = 183 \pm 1.0$). The scattering intensity at this point of $S(Q, \omega)$ is the sum of surface integrals of scattering intensities for each mode

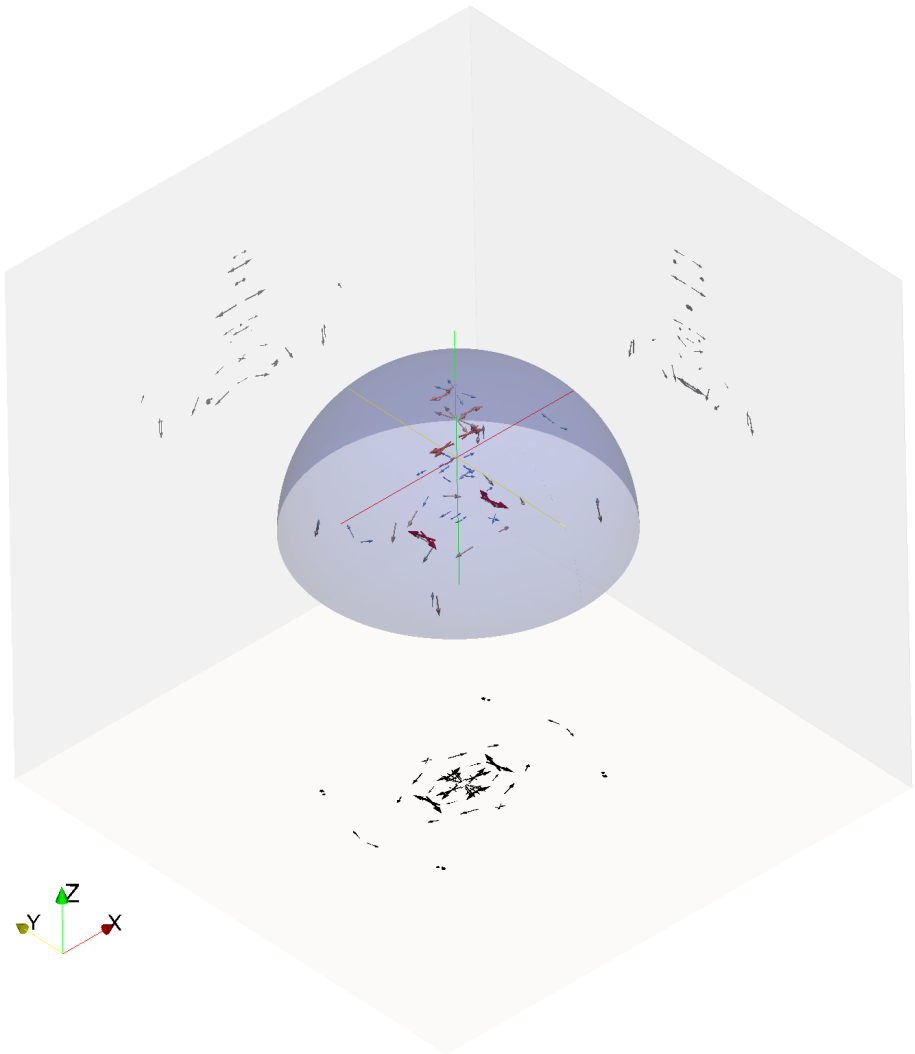


Figure C.10: Coherence locations in reciprocal space for the first phonon mode of Aluminium. Vectors are the directions of steepest change.

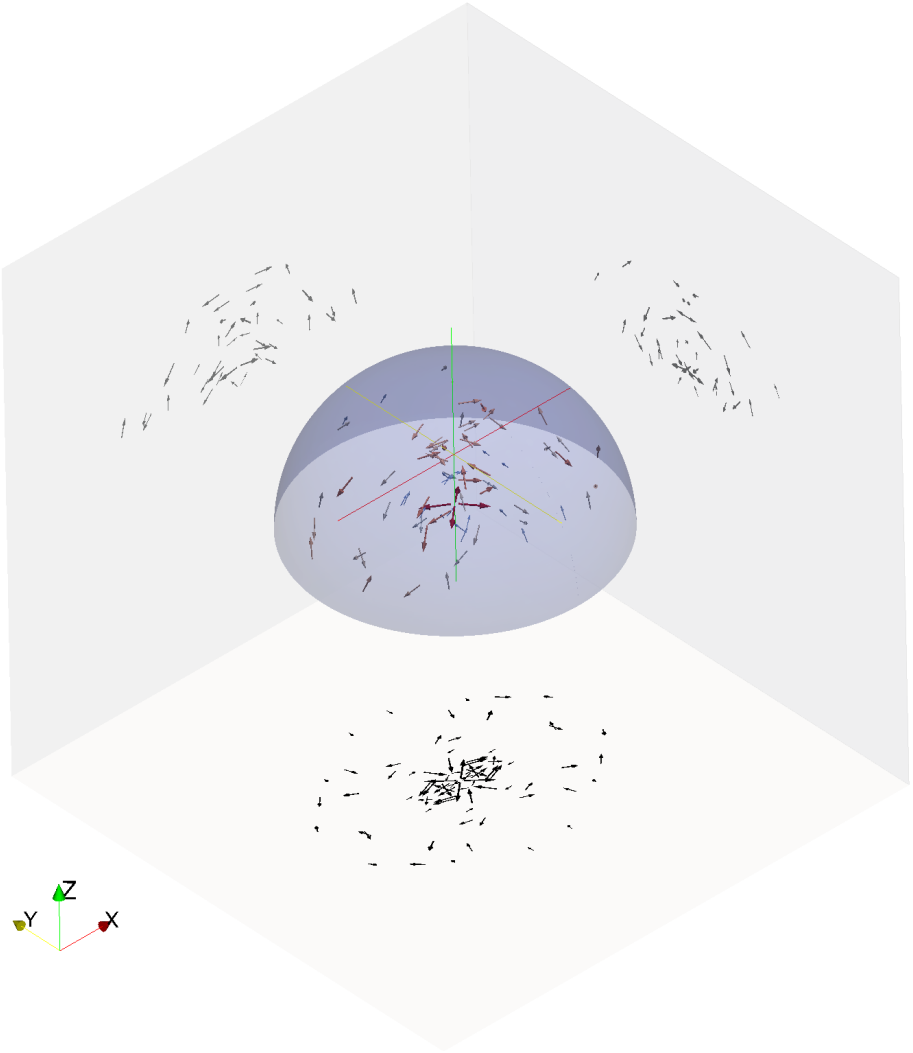


Figure C.11: Coherence locations in reciprocal space for the second phonon mode of Aluminium. Vectors are the directions of steepest change.

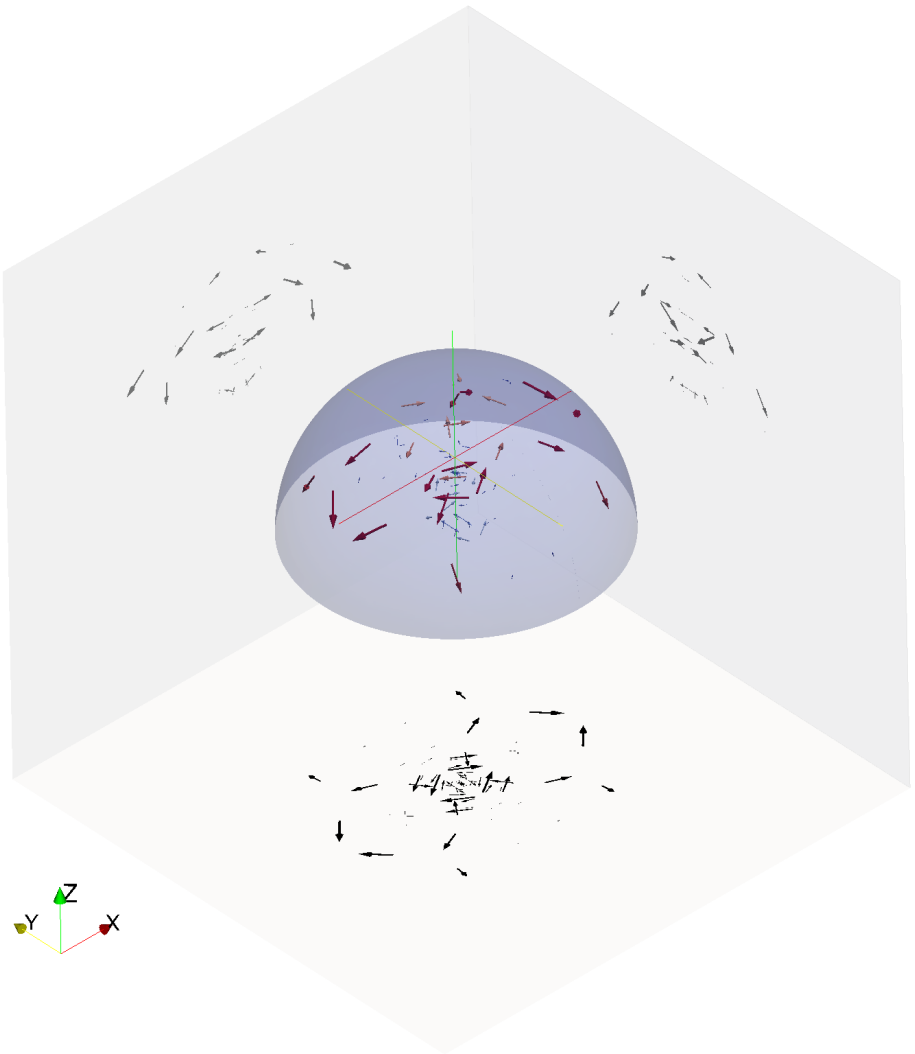


Figure C.12: Coherence locations in reciprocal space for the third phonon mode of Aluminium. Vectors are the directions of steepest change.

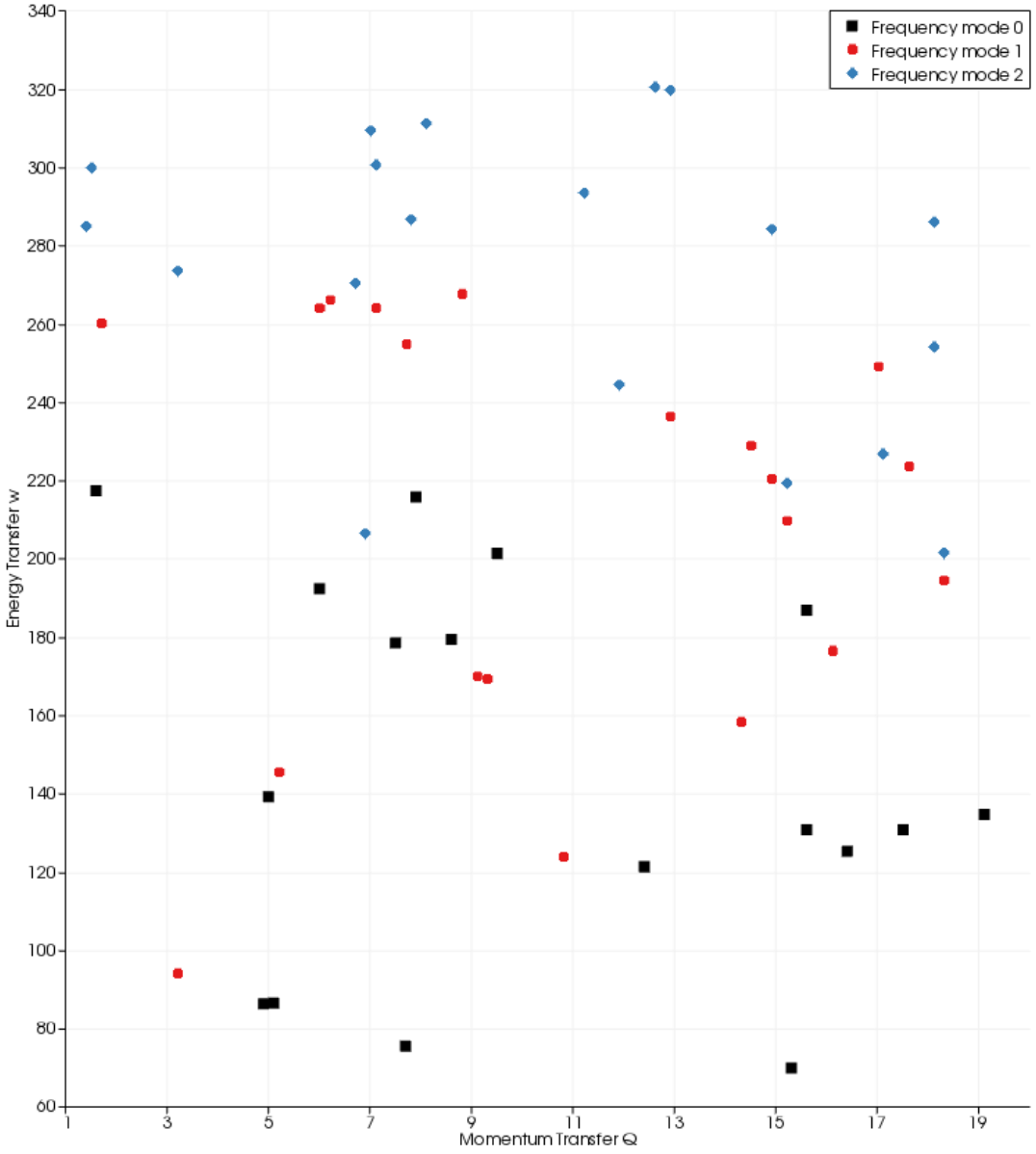


Figure C.13: Predicted coherence locations in $S(Q, w)$ for all phonon modes of Aluminium.

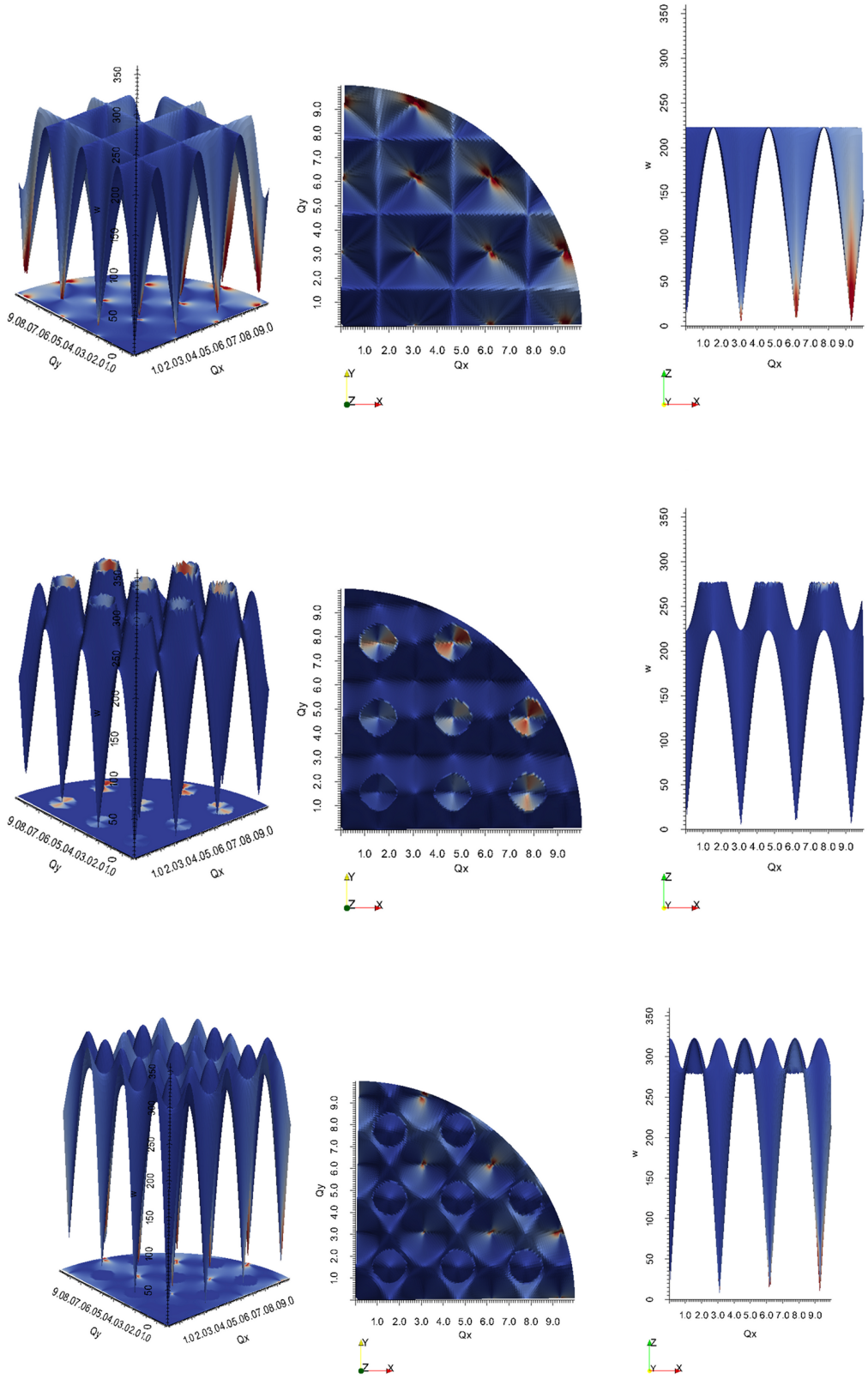


Figure C.14: Dispersion surfaces for three modes of Aluminium in $Q_x Q_y$ plane. Surfaces are coloured by scattering intensity. Prepared for [108]

BIBLIOGRAPHY

[1] E. Aarts and J. Korst. Simulated annealing and boltzmann machines. 1988. (Cited on page 104.)

[2] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533. ACM, 2011. (Cited on page 9.)

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009. (Cited on pages 19, 57, and 109.)

[4] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core.(a fastflow short tutorial). *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2011. (Cited on pages 9, 28, and 99.)

[5] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. A pattern language: Towns, buildings, construction (center for environmental structure series). 1977. (Cited on pages 5 and 21.)

[6] David P Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004. (Cited on page 9.)

[7] Edward Anderson. *LAPACK Users' guide*, volume 9. Siam, 1999. (Cited on page 17.)

[8] Edward Anderson, Zhaojun Bai, Jack Dongarra, A Greenbaum, A McKenney, Jeremy Du Croz, S Hammerling, James Demmel, C Bischof, and Danny Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press, 1990. (Cited on page 58.)

[9] Y Aoyama and J Nakano. *Rs/6000 sp: Practical MPI programming*. Citeseer, 1999. (Cited on pages xvii and 40.)

[10] Y Aoyama and J Nakano. *Rs/6000 sp: Practical MPI programming*. Citeseer, 1999. (Cited on page 32.)

[11] W.B. Arthur, S.N. Durlauf, and D.A. Lane. *The economy as an evolving complex system II*. Addison-Wesley Reading, MA, 1997. (Cited on page 102.)

[12] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. (Cited on page 8.)

[13] K Asanovic, R Bodik, J Demmel, T Keaveny, K Keutzer, J Kubiatowicz, N Morgan, D Patterson, K Sen, J Wawrzynek, and Others. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009. ISSN 0001-0782. (Cited on pages 5 and 9.)

[14] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. (Cited on page 11.)

[15] Z Bai. *Templates for the solution of algebraic eigenvalue problems*, volume 11. Society for Industrial Mathematics, 2000. (Cited on page 17.)

[16] Jyoti Batheja and Manish Parashar. Adaptive cluster computing using javaspaces. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 323–323. IEEE Computer Society, 2001. (Cited on page 9.)

[17] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems*, 7, 2011. (Cited on page 60.)

[18] Veeravalli Bharadwaj, Debasish Ghose, and Thomas G Robertazzi. Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems. *Cluster Computing*, 6(1):7–17, 2003. (Cited on page 89.)

[19] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966. (Cited on page 6.)

[20] D W Brenner. Empirical potential for hydrocarbons for use in simulating the chemical vapor deposition of diamond films. *Physical Review B*, 42(15):9458, 1990. (Cited on page 36.)

[21] S. Bussmann and K. Schild. Self-organizing manufacturing control: An industrial application of agent technology. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, pages 87–94. IEEE, 2000. (Cited on page 102.)

[22] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013. (Cited on page 70.)

[23] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4): 12–27, 2011. (Cited on page 70.)

[24] A. Cavagna, J.P. Garrahan, I. Giardina, and D. Sherrington. Thermal model for adaptive competition in a market. *Physical Review Letters*, 83(21):4429–4432, 1999. (Cited on page 104.)

[25] Andy Cedilnik, Berk Geveci, Kenneth Moreland, James Ahrens, and Jean Favre. Remote large data visualization in the paraview framework. In *Proceedings of the 6th Eurographics conference on Parallel Graphics and Visualization*, pages 163–170. Eurographics Association, 2006. (Cited on pages 31 and 141.)

[26] D Champion, J Tomkinson, and G Kearley. a-CLIMAX: a new INS analysis tool. *Applied Physics A: Materials Science & Processing*, 74:1302–1304, 2002. (Cited on page 14.)

[27] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012. (Cited on page 9.)

[28] Christian M Chilan, M Yang, Albert Cheng, and Leon Arber. Parallel i/o performance study with hdf5, a scientific data package. *TeraGrid 2006: Advancing Scientific Discovery*, 2006. (Cited on page 141.)

[29] Jerry A Clarke and Eric R Mark. Enhancements to the extensible data model and format (xdmf). In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, pages 322–327. IEEE, 2007. (Cited on page 141.)

[30] S.H. Clearwater, R. Costanza, M. Dixon, and B. Schroeder. Saving energy using market-based control. *Market Based Control. World Scientific: Singapore*, pages 253–273, 1996. (Cited on page 102.)

[31] Alan Kaylor Cline and James Meyering. Converting EISPACK To Run Efficiently On A Vector Processor. Technical report, Pleasant Valley Software, Austin, Texas, 1991. (Cited on page 58.)

[32] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press/Pitman, London, 1989. ISBN 0-262-53086-4. (Cited on page 8.)

[33] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001. (Cited on pages 79 and 125.)

[34] Stephen Cottrell, Francis Pratt, Adrian Hillier, Philip King, Freddie Akeroyd, Anders J Markvardsen, Nick Draper, Yuan Yao, and Stephen Blundell. Data formats and analysis codes–new software for μsr. *Physics Procedia*, 30:20–25, 2012. (Cited on page 137.)

[35] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972. (Cited on page 6.)

[36] R.K. Dash, P. Vytelingum, A. Rogers, E. David, and N.R. Jennings. Market-based task allocation mechanisms for limited-capacity suppliers. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 37(3):391–405, 2007. (Cited on page 102.)

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. (Cited on page 9.)

[38] Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968. (Cited on page 6.)

[39] J Dongarra and C B Moler. EISPACK: A package for solving matrix eigenvalue problems. Technical report, Argonne National Lab., IL (USA), 1983. (Cited on page 17.)

[40] Jack Dongarra, Iain S Duff, Danny C Sorensen, and Hank A van der Vorst. *Numerical linear algebra for high-performance computers*. SIAM, 2nd edition, 1998. ISBN 0898714281. (Cited on pages 19 and 36.)

[41] Martin T Dove. *Introduction to lattice dynamics*, volume 4. Cambridge university press, 1993. (Cited on page 17.)

[42] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Computing*, 2011. ISSN 0167-8191. doi: 10.1016/j.parco.2011.10.002. (Cited on page 11.)

[43] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz-open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002. (Cited on page 21.)

[44] Fritjof Boger Engelhardtsen and Tommy Gagnes. Using javaspaces to create adaptive distributed systems. In *Proceedings of Workshop and EUNICE Summer School on Adaptable Networks and Teleservices*, pages 125–130, 2002. (Cited on page 9.)

[45] Atilla Eryilmaz and R Srikant. Fair resource allocation in wireless networks using queue-length-based scheduling and congestion control. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1794–1803. IEEE, 2005. (Cited on page 107.)

[46] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011. (Cited on pages 31 and 141.)

[47] S I Feldman. A Fortran to C converter. In *ACM SIGPLAN Fortran Forum*, volume 9, pages 21–22. ACM, 1990. (Cited on page 59.)

[48] Wu-chun Feng and Kirk W Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, 2007. (Cited on page 10.)

[49] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. (Cited on page 10.)

[50] Mike Folk, Albert Cheng, and Kim Yates. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of Supercomputing*, volume 99, 1999. (Cited on page 141.)

[51] Brian Foote, Hans Rohnert, and Neil Harrison. *Pattern languages of program design 4*. Addison-Wesley Longman Publishing Co., Inc., 1999. (Cited on page 6.)

[52] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works) http://www.thoughtworks. com/Continuous Integration. pdf*, 2006. (Cited on page 122.)

[53] J D Gale. GULP: A computer program for the symmetry-adapted simulation of solids. *Journal of the Chemical Society, Faraday Transactions*, 93(4):629–637, 1997. (Cited on pages 13, 14, and 17.)

[54] J D Gale and A L Rohl. The general utility lattice program (GULP). *Molecular Simulation*, 29(5):291–341, 2003. (Cited on pages 14 and 17.)

[55] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2. (Cited on page 5.)

[56] Michael Garba and Horacio González-Vélez. Towards Ad-Hoc GPU Acceleration of Parallel Eigensystem Computations. In *ECMS 2011: 25th European Conference on Modelling and Simulation*, Krakow, Poland, June 2011. ECMS. (Cited on page 57.)

[57] Michael Garba, Horacio González-Vélez, and Daniel Roach. Parallel Computational Modelling of Inelastic Neutron Scattering in Multi-Node and Multi-Core Architectures. In *IEEE HPCC-10: Int Conf on High Performance Computing and Communications*, pages 509–514, Melbourne, September 2010. IEEE. ISBN 978-0-7695-4214-0. (Cited on page 49.)

[58] David Geer. Will software developers ride ruby on rails to success? *Computer*, 39(2):18–20, 2006. (Cited on page 6.)

[59] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985. ISSN 01640925. doi: 10.1145/2363.2433. URL http://portal.acm.org/citation.cfm?doid=2363.2433. (Cited on page 9.)

[60] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, February 1992. ISSN 00010782. doi: 10.1145/129630.376083. URL http://portal.acm.org/citation.cfm?doid=129630.376083. (Cited on page 9.)

[61] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992. (Cited on page 9.)

[62] B.P. Gerkey and M.J. Mataric. Sold!: Auction methods for multirobot coordination. *Robotics and Automation, IEEE Transactions on*, 18(5):758–768, 2002. (Cited on page 102.)

[63] M.I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 151–162. ACM, 2006. (Cited on page 109.)

[64] Naga K Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. In *SC'06: ACM/IEEE Conf on Supercomputing*, page 6, Tampa, November 2006. IEEE. ISBN 0-7695-2700-0. doi: 10.1109/SC.2006.2. (Cited on page 11.)

[65] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982. (Cited on page 32.)

[66] Thilina Gunarathne, Tak-Lon Wu, Judy Qiu, and Geoffrey Fox. Mapreduce in the clouds for science. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 565–572. IEEE, 2010. (Cited on page 9.)

[67] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011. (Cited on page 70.)

[68] M J Harvey and G De Fabritiis. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 182(4):1093–1099, 2011. ISSN 0010-4655. doi: 10.1016/j.cpc.2010.12.052. (Cited on page 11.)

[69] V Hernández, J E Román, A Tomás, and V Vidal. A survey of software for sparse eigenvalue problems. Technical Report STR-6, Universidad Polit{é}cnica de Valencia, \url{www.grycap.upv.es/slepc/}, 2006. (Cited on page 36.)

[70] Bao-Ling Huang and Massoud Kaviany. Ab initio and molecular dynamics predictions for electron and phonon transport in bismuth telluride. *Physical Review B*, 77(12):125209:1—-19, March 2008. doi: 10.1103/PhysRevB.77.125209. (Cited on page 14.)

[71] Alfred W Hübler and Timothy Wotherspoon. Self-adjusting systems avoid chaos. *Complexity*, 14(4):8–11, 2009. (Cited on page 23.)

[72] Jaakko Järvi and John Freeman. C++ lambda expressions and closures. *Science of Computer Programming*, 75(9):762–772, 2010. (Cited on page 6.)

[73] Rini T Kaushik and Milind Bhandarkar. Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the USENIX Annual Technical Conference*, page 109, 2010. (Cited on page 10.)

[74] K Kennedy and J R Allen. Optimizing compilers for modern architectures: a dependence-based approach. 2001. (Cited on page 39.)

[75] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36 (1):41–50, 2003. (Cited on page 127.)

[76] Volodymyr Kindratenko and Pedro Trancoso. Trends in high-performance computing. *Computing in Science & Engineering*, 13(3):92–95, 2011. (Cited on page 10.)

[77] D B Kirk and W H Wen-mei. *Programming massively parallel processors: A Hands-on approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2010. ISBN 0123814723. (Cited on page 11.)

[78] Oleg Kiselyov. Functional style in c++: Closures, late binding, and lambda abstractions. In *ACM SIGPLAN Notices*, volume 34, page 337. ACM, 1998. (Cited on page 6.)

[79] A Klöckner, N Pinto, Y Lee, B Catanzaro, P Ivanov, and A Fasih. PyCUDA: GPU run-time code generation for high-performance computing. *Arxiv preprint arXiv:0911.3456*, 2009. (Cited on page 66.)

[80] G Kresse and J Furthmüller. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Comput. Mat. Sci.*, 6 (15), 1996. (Cited on page 14.)

[81] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B.A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005. (Cited on page 102.)

[82] Seyong Lee, Seung-jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009. ISSN 0362-1340. doi: 10.1145/1594835.1504194. (Cited on page 11.)

[83] Mario Leyton and José M Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296. IEEE, 2010. (Cited on pages 9 and 21.)

[84] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning gemm for gpus. In *Computational Science–ICCS 2009*, pages 884–892. Springer, 2009. (Cited on pages 10 and 11.)

[85] Michael J Litzkow, Miron Livny, and Matt W Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111. IEEE, 1988. (Cited on page 9.)

[86] A. Mas-Colell, M.D. Whinston, J.R. Green, et al. *Microeconomic theory*, volume 1. Oxford university press New York, 1995. (Cited on page 127.)

[87] K K Matam and K Kothapalli. Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using {GPU}. In *ICPP 2011*, pages 612–621, Taipei City, September 2011. IEEE. doi: 10.1109/ICPP.2011.82. (Cited on page 11.)

[88] Author Tim Mattson. Our Pattern Language (OPL) The Scope of OPL OPL and programmer roles. pages 1–11. (Cited on page 6.)

[89] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004. ISBN 0321228111. (Cited on pages 6, 31, 33, and 105.)

[90] P C H Mitchell, S F Parker, A J Ramirez-Cuesta, and J Tomkinson. *Vibrational Spectroscopy with Neutrons*. WS, World Scientific, 2005. (Cited on page 16.)

[91] J. Moody and M. Saffell. Learning to trade via direct reinforcement. *Neural Networks, IEEE Transactions on*, 12(4):875–889, 2001. (Cited on page 127.)

[92] Michael J Neely. Stochastic network optimization with application to communication and queueing systems. *Synthesis Lectures on Communication Networks*, 3(1): 1–211, 2010. (Cited on pages 105 and 108.)

[93] N Nethercote and J Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007. ISSN 0362-1340. (Cited on page 59.)

[94] J Nickolls, I Buck, M Garland, and K Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. ISSN 1542-7730. (Cited on page 10.)

[95] Michael S Noble and Stoyanka Zlateva. Scientific computation with javaspaces. In *High-Performance Computing and Networking*, pages 657–666. Springer, 2001. (Cited on page 9.)

[96] Peter Norvig. Design patterns in dynamic programming. *Object World*, 96(5), 1996. (Cited on page 6.)

[97] Nvidia Corporation. NVIDIA CUDA C Programming Best Practices Guide. Manual Version 2.3, 2009. (Cited on pages 11 and 62.)

[98] J D Owens, M Houston, D Luebke, S Green, J E Stone, and J C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. ISSN 0018-9219. (Cited on page 89.)

[99] S F Parker, S M Bennington, J W Taylor, H Herman, I Silverwood, P Albers, and K Refson. Complete assignment of the vibrational modes of C60 by inelastic neutron scattering spectroscopy and periodic-DFT. *Phys. Chem. Chem. Phys.*, 2011. (Cited on page 75.)

[100] K Parlinski, Z Q Li, and Y Kawazoe. First-Principles Determination of the Soft Mode in Cubic ZrO2. *Phys. Rev. Lett.*, 78(21):4063–4066, May 1997. doi: 10.1103/PhysRevLett.78.4063. (Cited on page 14.)

[101] Krzysztof Parlinski. The PHONON software package, 2003. (Cited on page 14.)

[102] A J Ramirez-Cuesta. aClimax 4.0.1, The new version of the software for analysing and interpreting INS spectra. *Computer Physics Communications*, 157 (3):226–238, March 2004. (Cited on page 14.)

[103] Dennis C Rapaport. *The art of molecular dynamics simulation*. Cambridge University Press, Cambridge, second edition, 2004. (Cited on page 14.)

[104] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007. (Cited on pages 9 and 28.)

[105] Daniel L Roach. *Computational Investigations of Polycrystalline Systems Using Inelastic Neutron Scattering Techniques*. PhD thesis, University of Salford, Salford M5 4WT, UK, 2006. (Cited on pages 14 and 16.)

[106] Daniel L Roach, J D Gale, and D K Ross. Scatter: A New Inelastic Neutron Scattering Simulation Subroutine for GULP. *Neutron News*, 18(3):21–23, 2007. (Cited on pages 14 and 16.)

[107] Daniel L Roach, Brent Heuser, D Keith Ross, Michael T Garba, Gael Baldissin, Julian D Gale, and Douglas L Abernathy. Experimental determination of the Q-dependence of in-plane vibrations in graphite. *In preparation*, 2013. (Cited on page 14.)

[108] Daniel L Roach, D Keith Ross, Julian D Gale, and Jon W Taylor. The interpretation of polycrystalline coherent inelastic neutron scattering from aluminium. *Journal of Applied Crystallography*, 2013. (Cited on pages 14, 16, 51, and 150.)

[109] Thomas G Robertazzi. Ten reasons to use divisible load theory. *Computer*, 36(5): 63–68, 2003. (Cited on page 89.)

[110] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2009. (Cited on page 70.)

[111] SARA. Description of the Huygens system. Web page, Dutch National High Performance Computing and e-Science Support Center, http://www.sara.nl/systems/huygens/description, 2011. (Cited on page 49.)

[112] Robert Sedgewick. *Algorithms in Java*. Addison-Wesley Professional, 2003. (Cited on page 21.)

[113] MD Segall, Philip JD Lindan, MJ al Probert, CJ Pickard, PJ Hasnip, SJ Clark, and MC Payne. First-principles simulation: ideas, illustrations and the castep code. *Journal of Physics: Condensed Matter*, 14(11):2717, 2002. (Cited on page 14.)

[114] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010. (Cited on page 9.)

[115] Brian T Smith, James M Boyle, Jack Dongarra, Burton S Garbow, Yasuhiko Ikebe, Virginia C Klema, and Cleve B Moler. *Matrix Eigensystem Routines - {EISPACK} Guide*, volume 6 of *LNCS*. Springer-Verlag, 1976. ISBN 3-540-07546-1. (Cited on page 19.)

[116] G L Squires. *Introduction to the theory of thermal neutron scattering*. Cambridge Univ. Press, 1978. (Cited on pages 12, 13, and 57.)

[117] K. Steiglitz, M.L. Honig, and L.M. Cohen. A computational market model based on individual action. *Market-Based Control*, pages 1–27, 1996. (Cited on pages 102 and 127.)

[118] Leandros Tassiulas. Linear complexity algorithms for maximum throughput in radio networks and input queued switches. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 533–539. Ieee, 1998. (Cited on page 107.)

[119] Leandros Tassiulas and Anthony Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *Automatic Control, IEEE Transactions on*, 37(12):1936–1948, 1992. (Cited on page 107.)

[120] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(Suppl 12): S1, 2010. (Cited on page 9.)

[121] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer, 2002. (Cited on pages 9 and 109.)

[122] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009. (Cited on page 9.)

[123] S Tomov, J Dongarra, and M Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010. ISSN 0167-8191. (Cited on pages 11 and 57.)

[124] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with {GPU} accelerators. In *IPDPS 2010 Workshops*, pages 1–8, Atlanta, April 2010. IEEE. (Cited on pages 11, 19, and 57.)

[125] F J Torres, P Ugliengo, B Civalleri, A Terentyev, and C Pisani. A review of the computational studies of proton- and metal-exchanged chabazites as media for molecular hydrogen storage performed with the {CRYSTAL} code. *International Journal of Hydrogen Energy*, 33(2):746–754, 2008. ISSN 0360-3199. doi: DOI:10.1016/j.ijhydene.2007.09.039. (Cited on page 14.)

[126] Sain-Zee Ueng, Melvin Lathara, Sara Baghsorkhi, and Wen-Mei Hwu. {CUDA-Lite}: Reducing {GPU} Programming Complexity. In *Languages and Compilers for Parallel Computing (revised papers)*, volume 5335 of *Lecture Notes in Computer Science*, pages 1–15, Edmonton, July 2008. Springer-Verlag. ISBN 978-3-540-89739-2. doi: 10.1007/978-3-540-89740-8_1. (Cited on page 11.)

[127] L Van Hove. Correlations in space and time and Born approximation scattering in systems of interacting particles. *Physical Review*, 95(1):249, 1954. (Cited on page 12.)

[128] F Vázquez, J J Fernández, and E M Garzón. A new approach for sparse matrix vector product on {NVIDIA GPUs}. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011. ISSN 1532-0634. (Cited on page 57.)

[129] William von Hagen. *The Definitive Guide to GCC, Second Edition (Definitive Guide)*. Apress, Berkely, CA, USA, 2006. ISBN 1590595858. (Cited on page 32.)

[130] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. Kepler+ hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, page 12. ACM, 2009. (Cited on page 9.)

[131] M.P. Wellman, W.E. Walsh, P.R. Wurman, and J.K. MacKie-Mason. Auction protocols for decentralized scheduling. *Games and Economic Behavior*, 35(1):271–303, 2001. (Cited on page 102.)

[132] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998. (Cited on pages 10 and 11.)

[133] J Wilkinson and C Reinsch. *Linear Algebra*, volume 2 of *Handbook for Automatic Computation*. Springer-Verlag, 1971. (Cited on page 19.)

[134] Jeremiah Willcock, Jaakko Järvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine. Lambda expressions and closures for c+. 2006. (Cited on page 6.)

[135] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007. (Cited on page 9.)

[136] B.P.C. Yen and OQ Wu. Internet scheduling environment with market-driven agents. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 34(2):281–289, 2004. (Cited on page 102.)

[137] Jia Yu and Rajkumar Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 119–128. IEEE, 2004. (Cited on page 9.)

[138] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010. (Cited on page 9.)

[139] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012. (Cited on page 9.)