Predictive Analysis and Optimisation of Pipelined Wavefront Applications Using Reusable Analytic Models

by

Gihan Ravideva Mudalige

A thesis submitted to the University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

University of Warwick

July 2009

Abstract

Pipelined wavefront computations are an ubiquitous class of high performance parallel algorithms used for the solution of many scientific and engineering applications. In order to aid the design and optimisation of these applications, and to ensure that during procurement platforms are chosen best suited to these codes, there has been considerable research in analysing and evaluating their operational performance.

Wavefront codes exhibit complex computation, communication, synchronisation patterns, and as a result there exist a large variety of such codes and possible optimisations. The problem is compounded by each new generation of high performance computing system, which has often introduced a previously unexplored architectural trait, requiring previous performance models to be rewritten and reevaluated.

In this thesis, we address the performance modelling and optimisation of this class of application, as a whole. This differs from previous studies in which bespoke models are applied to specific applications. The analytic performance models are generalised and reusable, and we demonstrate their application to the predictive analysis and optimisation of pipelined wavefront computations running on modern high performance computing systems.

The performance model is based on the LogGP parameterisation, and uses a small number of input parameters to specify the particular behaviour of most wavefront codes. The new parameters and model equations capture the key structural and behavioural differences among different wavefront application codes, providing a succinct summary of the operations for each application and insights into alternative wavefront application design.

The models are applied to three industry-strength wavefront codes and are validated on several systems including a Cray XT3/XT4 and an InfiniBand commodity cluster. Model predictions show high quantitative accuracy (less than 20% error) for all high performance configurations and excellent qualitative accuracy.

The thesis presents applications, projections and insights for optimisations using the model, which show the utility of reusable analytic models for performance engineering of high performance computing codes. In particular, we demonstrate the use of the model for: (1) evaluating application configuration and resulting performance; (2) evaluating hardware platform issues including platform sizing, configuration; (3) exploring hardware platform design alternatives and system procurement and, (4) considering possible code and algorithmic optimisations.

to my parents and grandmother
with love and gratitude

Acknowledgements

I am indebted to many people for the help, advice, guidance, support and friendship they have provided me during the course of this work. Several have made special contributions, influencing not only my thesis work but also my technical and professional development in research and computer science. It is a privilege to acknowledge them here.

My supervisor, Dr. Stephen Jarvis, first guided me to conduct research in this area, giving me the opportunity to work in the High Performance Systems Group at Warwick and provided a never-ending source of optimism, good-will and guidance. I am truly grateful for his support, advice and encouragement.

Prof. Mary Vernon, my advisor, during the research fellowship year at the University of Wisconsin-Madison has been an inspiration for developing analytic models. I am sincerely indebted to her for giving me the opportunity to work at Madison and for the many long hours of discussions, advice and motivation.

I am grateful to Dr. Daniel Spooner for acting as my second supervisor, particularly for his advice during the early years of my degree.

A special vote of thanks should go to my colleague and fellow labmate, Simon Hammond for his relentless hard work, support and the many hours of discussions during our research collaborations.

I would like to acknowledge and thank Jon Holt, Andy Herdman, Ash Vadgama and Ben Ralston of the Parallel Technology Support team at AWE for providing us with access to the Chimaera benchmark code as well as supporting our research with valuable comments. Additionally I am thankful to Patrick H. Worley for giving us access to the ORNL Cray XT3/XT4 under the PEAC project, Howard Pritchard for his comments on the Cray XT3/XT4 and David Sundram-Stukel for his comments during the development of the reusable analytic model.

It is a pleasure to acknowledge the many members and colleagues of the High Performance Systems Group both past and present including, Dr. Ligang He, Dr. Guang Tan, Dr. David Bacigalupo, Dr. Graham Nudd, Dr Nathan Griffiths, Dr Arshad Jhumka, Dr. Elizabeth Ogston, Dr. Tongcheng Guo, Jonathan Smith, Justin Dyson, Xenu Chen, Peter Wong, Jonathan Byrd, Brian Foley, Paul Isitt, Lei Zhao, James Wen Jun Xue, Adam Chester, Matthew Leeke, Alistair Mills and Mohammed Al Ghamdi.

I would also like to acknowledge the support of my many friends, particularly Dr. Dhammika Widanage and Matthew Higgins for the supportive discussions and guidance during my PhD years.

Finally, my profound appreciation goes to my parents and my grandmother. My father whose perfectionism in his work will be a continuing source of inspiration to me, my mother whose love, support and limitless optimism shatters the daunting perception of even the most impossible undertaking and my grandmother who brought me up during my school days with love and unending compassion. I dedicate this thesis to them, with love and gratitude.

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by myself except where otherwise stated.

Portions of this work have been published in the following publications:

Parts of Chapter 2 in [1] :
S.A. Jarvis, D.P. Spooner, G.R. Mudalige, B.P. Foley, J. Cao, and G.R. Nudd. Performance Evaluation of Parallel and Distributed Systems, chapter *Performance Prediction Techniques for Large-scale Distributed Environments*. Mohamed Ould-Khaoua and Geyong Min Eds. Nova Science, 2005.

Chapters 3, 4 and 5 in [2]:
G.R. Mudalige, M.K. Vernon, and S.A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*. April, 2008. IEEE Computer Society.

Chapter 6 in [3]:
G.R. Mudalige, S.A. Jarvis, D.P. Spooner, and G.R. Nudd. Predictive Performance Analysis of a Parallel Pipelined Synchronous Wavefront Application for Commodity Processor Cluster Systems. In *Proc. IEEE International Conference on Cluster Computing - Cluster2006*, Barcelona, September 2006. IEEE Computer Society.

Parts of Chapter 7 in [4]:
S.D. Hammond, G.R. Mudalige, J.A. Smith, and S.A. Jarvis. Performance prediction and procurement in practise: Assessing the suitability of commodity cluster components for wavefront codes. In *Proc. Performance Engineering Workshop 08 (UKPEW)*, Imperial College, London, July 2008. Also accepted for publication in *IET Software* 2009.

and in [5]:
G.R. Mudalige, S.D. Hammond, J.A. Smith, and S.A. Jarvis. Predictive Analysis and Optimisation of Pipelined Wavefront Computations. In *Proc. 11th Workshop on Advances in Parallel and Distributed Computational Models (APDCM 2009)*, held as part of the *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009)*, Rome, Italy, May 2009. IEEE Computer Society. Also invited for publication in the *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, 2010. Taylor and Francis Publications.

Other work that has been conducted as part of this research has been published in:

[6] S.D. Hammond, G.R. Mudalige, J.A. Smith, and S.A. Jarvis. WARPP - A Tool Kit for Simulating High-Performance Parallel Scientific Codes. *In Proc. 2nd International Conference on Simulation Tools and Techniques (SIMUTools 2009)*, Rome, Italy, March 2009. ACM Press.

[7] S.D. Hammond, J.A. Smith, G.R. Mudalige, and S.A. Jarvis. Predictive Simulation of HPC Applications. In *The IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA 2009)*, Bradford, U.K., 26-29 May 2009. IEEE Computer Society.

Sponsorship and Grants

The research in this thesis was part-sponsored by research grants from:

The University of Warwick Department of Computer Science Fellowship (2004-2005), The Warwick Postgraduate Research Fellowship (WPRF) (2005-2006 and 2007-2008), The Overseas Research Student Fellowship (ORS) by the Government of the U.K. (2004-2006 and 2007-2008), The Warwick, Wisconsin-Madison Research Fellowship (2006-2007) and The National Science Foundation (NSF), U.S. under grant CNS-0435437.

Access to the Chimaera benchmark was provided by the Atomic Weapons Establishment (AWE) U.K. under grants CDK0660 (*The Production of Predictive Models for Future Computing Requirements*) and CDK0724 (*AWE Technical Outreach Program*).

This research work used resources at the National Centre for Computational Sciences (NCCS) at the Oak Ridge National Laboratory (ORNL), which is supported by the Office of Science of the U.S. Department of Energy (DOE) contract DE-ASC05-00OR22725 under the Performance Evaluation and Analysis Consortium (PEAC).

This research also made use of resources at the Centre for Scientific Computing (CSC) at the University of Warwick under the Science Research Investment Fund and Joint Research Equipment Initiative under grant JR00WASTEQ.

Abbreviations

API - Application Programming Interface
ASC - Advanced Simulation and Computing Program
AWE - Atomic Weapons Establishment (U.K)
BLAS - Basic Linear Algebra Subprograms
BSP - Bulk Synchronous Parallel (model)
CFDs - Computational Fluid Dynamics
CHIP^{3}S - Characterisation Instrumentation for Performance Prediction of Parallel Systems
CMP - Chip Multi-Processor
CNL - Compute Node Linux
CRCW - Concurrent Read Concurrent Write
CREW - Concurrent Read Exclusive Write
DMA - Direct Memory Access
EPCC - Edinburgh Parallel Computing Centre
EREW - Exclusive Read Exclusive Write
FFT - Fast Fourier Transforms
FLOPS - Floatingpoint Operations per Second
HMCL - Hardware Model Characterisation Language
HPC - High Performance Computing
HPF - High Performance FORTRAN
IMB - Intel MPI Benchmark
ILP - Instruction level parallelism
LAN - Local Area Network
LANL - Los Alamos National Laboratory (U.S)
LAPACK - Linear Algebra Package
LU - Lower and Upper triangular system solution application benchmark in NPB
MFLOPS - Millions of Floating-Point Operations per Second
MIMD - Multiple Instruction (stream) Multiple Data (stream)
MIPS - Millions of Instructions per Second
MISD - Multiple Instruction (stream) Single Data (stream)
MTU - Maximum Transmission Unit
MPI - Message Passing Interface
MPMD - Multiple Program Single Data
MPP - Massively Parallel Processor
MTU - Maximum Transmission Unit
MVA - Mean Value Analysis
NPB - NASA's Aerodynamic Simulation Parallel Benchmarks
NIC - Network Interface Card
NoW - Network of Workstations

NUMA - Non Uniform Memory Access
ORNL - Oak Ridge National Laboratory (U.S)
PACE - Performance Analysis and Characterisation Environment
PAPI - Performance Application Programming Interface
PBB - Purpose Bases Benchmarks
PGAS - Partitioned Global Address Space
PRAM - Parallel Random Access Machine
PSL - Performance Specification Language
PVM - Parallel Virtual Machine
QoS - Quality of Service
SDR - Single Data Rate
SIMD - Single Instruction (stream) Multiple Data (stream)
SISD - Single Instruction (stream) Single Data (stream)
SMP - Symmetric Multi Processor
SPEC - Standard Performance Evaluation Corporation
SPMD - Single Program Multiple Data
SSOR - Successive Over-Relaxation
UMA - Uniform Memory Access
WAN - Wide Area Network
WarPP - Warwick Performance Prediction toolkit

Notations

C	Number of cores on a CMP(section 4.3.3)
Cx	Number of cores in the x dimension on a CMP (section 4.6)
Cy	Number of cores in the y dimension on a CMP (section 4.6)
G	LogGP parameter: *Gap per byte* defined as the time taken to transmit a byte on to the network (section 2.3.4)
G_{copy}	Gap per byte for a memory copy on-chip (section 4.3.2)
G_{dma}	Gap per byte for a DMA transfer on-chip(section 4.3.2)
$H_{tile}(cells)$	Height of the tile in the z dimension (section 4.2)
it, jt, kt	Number of grid cells in x, y and z dimension for Sweep3D (section 3.1.2)
I	NIC contention (interference) time (section 4.6)
L	LogGP parameter: the upper bound on the *latency* of the network the flight time for a message from one point of the network to another (section 2.3.4)
m	Number of processors along the y dimension of the 2D processor array (section 3.1.2)
mmi	Number of angles solved per sweep step in Sweep3D (section 3.2.2)
mk	Height of the tile in the z dimension for Sweep3D (section 3.2.2)
$MessageSize_{EW}$	Message size (East-West or West-East) (section 4.2)
$MessageSize_{NS}$	Message size (North-South or South-North) (section 4.2)
n	Number of processors along the x dimension of the 2D processor array (section 3.1.2)
n_{diag}	Number of sweeps that completes from corner up to and including the main diagonal (section 4.2)
n_{full}	Number of sweeps that completes fully (from corner to opposite corner) (section 4.2)
$no\_of\_Kblocks$	Number of tiles on a processor stack in the Sweep3D analytic model in [8] (section 4.2)

n_{sweeps}	Total number of sweeps (section 4.2)
N_x, N_y, N_z	Number of grid cells in x, y and z dimension (section 3.1.2)
o	LogGP parameter: the *overhead* time taken by a processor to transmit/receive a message. (section 2.3.4)
o_{c2NIC}	Time to setup a, DMA or other, copy of the message data between kernel memory and the NIC and to prepare or process the message header (section 4.3.1)
o_{copy}	Processing time before and after the message copies on the sender and the receiver on-chip (section 4.3.2)
o_{dma}	Overhead of setting up a DMA or other, copy of the message data between kernel memory and the NIC on-chip (section 4.3.2)
o_h	The processing time for a handshake request or reply, including the time to prepare a new message header (section4.3.1)
o_{init}	Overhead for a message copy between application and kernel (section4.3.1)
P	LogGP parameter: number of processors (section 2.3.4)
P_r	Probability that a processor takes W_r time to complete a block of cells of height H_{tile} (section 7.4.1)
R	Execution time for a single simulation (section 5.2)
$Receive$	Time to obtain a message from the network (section 4.2)
$Send$	Time to release a message to the network (section 4.2)
$StartP_{i,j}$	Time to begin the main computation on processor (i, j) (section 4.2)
$T_{allreduce}$	Time to complete an MPI allreduce operation (section 4.3.3)
$T_{diagfill}$	Time gap between starting a sweep at a corner processor and the first wavefront of that sweep reaching up to the main diagonal processors (section 4.2)
T_{fill}	Time for a wavefront operating on a 1D processor array to arrive from one end processor to the opposite side processor (section 4.5)
$T_{fullfill}$	time gap between starting a sweep at a corner processor and the first wavefront of that sweep reaching the opposite corner processor (section 4.2)
$T_{nonwavefront}$	Time to complete non-wavefront portions (section 4.2)
$Total\_Comm$	End to end communication time (section 4.2)

T_{stack} Time taken by a processor to solve its stack of tiles (section 4.2)

W_g A grid cell computation time (main computation block) (section 4.2)

$W_{g,pre}$ A grid cell computation time (pre-computation block) (section 4.2)

$W_{g,rhs}$ A grid cell computation time (LU RHS computation block) (section 4.7.1)

X Total number of simulations that complete per unit time (throughput) (section 5.2)

η Number of overlapping simultaneous sweeps (section 7.3.2)

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

The use of computational methods is now an essential research methodology that propels modern science, medicine and engineering. The scientific methods of computational modelling and simulation are now as ubiquitous as the traditional theoretical and experimentation approaches to scientific research. Particularly through computational modelling and simulation, domain scientists and researchers in these fields have been able to conceptualise, discover, design and produce more innovative solutions and solve highly complex problems than was ever possible with traditional *paper and pencil* methods. Thus, such computational methods have established themselves as a third pillar of scientific investigation, comparable to experimental and theoretical approaches. In the last decade, the use of these techniques has grown rapidly, due to the increasing availability of efficient and large-scale computational resources.

High Performance Computing (HPC) or its more popularly known designation - *supercomputing* - is concerned with the research, design, development, manufacturing, deployment and usage of such high-end computer systems. It is more specifically characterised by the use of parallel processing for running advanced application programs quickly, efficiently and reliably. HPC systems can be of the form of a high-end server, or a massively parallel *supercomputer* with thousands of processors interconnected by high speed networks or a large distributed system that is spread across a building, city, country or even continents.

As the name implies, *performance* is key to HPC and the goal is to achieve the best performance in a cost-effective and resource efficient manner. Analysis and evaluation of performance as well as its prediction and speculating about future behaviour have therefore been key aspects of HPC. These activities - collectively referred to as *performance engineering* - are particularly invaluable, as these systems and software often require a large investment not only from the end user and owners but also from the designers, developers and vendors. The advantages of performance engineering also extend beyond this, allowing efficient scheduling by anticipating a workload's behaviour prior to execution [9, 10], which in turn allows efficient utilisation of resources and sustainable levels of Quality of Service (QoS) [11]. It has been recognised that performance engineering techniques can be used throughout the lifecycle of a system [9, 12]. For example, at the design stage they can serve to quantify the advantages and disadvantages of different architectural options. When procuring systems, users can utilise performance predictions to compare alternative systems and at the implementation stage predictions based on an implemented prototype can serve as a forecast for the final system [13]. After installation, predicted results can be used to validate whether the installation was successful, and whether the system is configured accurately to obtain optimum system efficiency. Additionally, during maintenance such performance analysis data can indicate faults that affect the system behaviour and also quantify the possible benefits that can be gained by upgrading. On the application side, for example, performance engineering enables application scientists to design near optimal application code by exposing software bottlenecks [14] and hardware facilities that should be exploited to obtain maximum efficiency and resource utility [15].

Due to these significant benefits, performance engineering has been and continues to be an important research area in the field of HPC. But understanding the performance aspect of an HPC application when running on various parallel high performance systems and architectures still remains to be a significantly difficult task. The complexity of the HPC applications and systems, the rapid pace of technology development and the lack of expertise in performance engineering make it highly non-trivial and labour intensive.

The underlying objective of this work is to assist the complex and demanding task of performance engineering for HPC. To this end, the dissertation investigates the performance of a significantly important, non-trivial and ubiquitous class of HPC computations - Pipelined Wavefront Computations - on modern HPC systems. We conduct a performance engineering study of these codes by developing a reusable analytic model, attesting it as a technique that simplifies the task of performance engineering, and then utilise the model for comprehensively assessing the quantitative and qualitative performance of pipelined wavefront computations on modern HPC systems.

1.1 Motivation and Problem Statement

Pipelined Wavefront Computations, originally described as hyperplane methods [16] by Lamport form a major portion (up to 80 % [17]) of the high performance scientific computing workload at institutes such as the Los Alamos National Laboratories (LANL) in the U.S [18] and the Atomic Weapons Establishment (AWE) [19] in the U.K. The communication primitives used in these parallel benchmark codes are the blocking send, receive, and group communication primitives in the Message Passing Interface (MPI) [20]. To aid the design, procurement and optimization decisions of these applications on high-end computers, there has been and continues to be considerable research interest in performance engineering of these codes.

Pipelined Wavefront Computations, generally operates on a 3D grid of data cells (although the algorithm extends equally well on to 1D and 2D grids), where the parallel processing of the computation can be viewed as a wavefront originating from a corner of the data grid propagating to the oppisite corner of the data grid. Figure 1.1 illustrates this operation. Here, a 3D data grid of size $N_x \times N_y \times N_z$ is decomposed onto a 2D array of $m \times n$ processors. Each partition of data cells assigned to a processor can be viewed as a stack of tiles, each of 1 cell high. The data dependency of the cells held in processors results in a sequence of wavefronts (or a sweep) that starts at one of the corner processors, computing over the cells of its topmost or bottommost tile and propagating to the opposite processor's bottommost or topmost tile. The shaded tiles in Figure 1.1 depict three basic steps of the general wavefront operation. It illustrates the tiles that are processed during the nal three wavefronts (or final three sweep steps) - light Gray, then medium Gray, then dark Gray - belonging to a sweep that originated at the bottommost tile on processor (1, 1), ending at the topmost tile on (n, m).

There have been several previous performance engineering studies on pipelined wavefront computations. In [14], Yarrow *et al.* develop analytic models of two different versions of the NAS parallel benchmark [21], LU. LU employs pipelined wavefront computations to

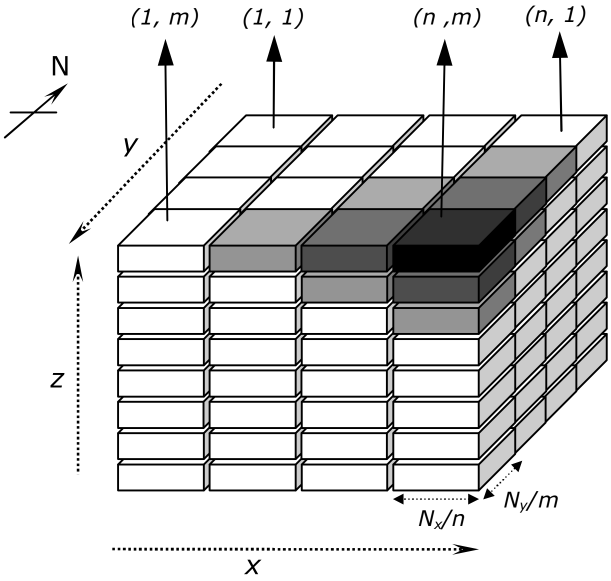


Figure 1.1: Operation of a Wavefront computation

solve a compressible Navier-Stokes equation used in computational fluid dynamics (CFDs). The model elucidates the differences in communications structure and predicts performance with a maximum of 30% error compared with measured LU execution time on an IBM SP system. Sundaram-Stukel and Vernon [8] develop an analytic model based on LogGP [22] for the pipelined wavefront application Sweep3D [23] on an IBM SP/2 system. This model has high accuracy with less than 10% error for up to 128 processors. Mathis *et al.* [24] develop a general analytic performance model for Sweep3D and apply the model to explore two possible alternative domain decompositions. Hoisie *et al.* [17] develop analytic models for a single *sweep* of the wavefront code based on Sweep3D, while Kerbyson *et al.* [25] use the single sweep equation to model the total Sweep3D runtime. In [26], Hoisie *et al.* detail the contention arising in wavefront computations when executed on clusters of SMPs where processors in a node share router links to communicate with processors in other nodes.

Non-analytic performance engineering work on wavefront codes include the Sweep3D simulation model developed and evaluated using PACE [27, 28] and MPI-SIM [29, 30] which was part of the POEMS [31] system. PACE uses a layered performance characterisation method and provides tools that model and evaluate (using discrete-event simulation) the computation and communication activities in an application by linking serial code graphs with a template that models the message passing and concurrency behaviour. MPI-SIM simulates an application by direct execution of computation and parallel discrete even simulation for communication and I/O operations. Other tools of note that have been used to evaluate Sweep3D include Kojak [32] and PTI [33].

Other related work on wavefront codes includes a study on the best methods to present them to the compiler for effective generation of parallel codes [34]. In this work the wavefront algorithm is parallelised using three approaches: message passing, compiler driven automatic parallelisation and programmer defined explicit parallel language features. The work assesses the efficiency of these approaches for parallelising wavefront algorithms.

In spite of all the above work, we believe that three key issues continue to make performance engineering of wavefront applications important:

Firstly, the complex synchronisation pattern of computation and communications in wavefront codes brings with it a considerable amount of structural variations and optimisation possibilities. Each of the previous performance engineering studies are specific to LU or Sweep3D and require significant and unspecified restructuring to apply them to other wavefront codes of interest. As will be seen in the applications that are analysed in this dissertation, each have considerable differences in domain decompositions, operational characteristics, parallel overlappings and optimisations. Therefore one of the key difficulties has been that each new wavefront application has had to be performance engineered from scratch.

Secondly, the above problem was confounded by each generation of HPC systems, introducing a previously unexplored aspect that made previous studies of wavefront computations require a considerable (sometimes almost entire) re-evaluation. For instance, the typical approach to performance engineering of new wavefront applications or evaluating a new HPC system was to modify existing performance studies to reflect the behaviour of the new code or the new systems. However, the modification process is error-prone and thus each new model must be extensively validated. One of the underlying goals of this dissertation is to develop methods that reduce this effort, alleviating the above problems and assisting in future performance engineering work of wavefront applications.

Finally and more importantly, all the previous performance studies on wavefront applications have only addressed (in most cases through customised models) the performance and thus the optimisation of a specific application has not explored or investigated the class of applications as a whole. A considerable motivation for us, therefore, is the open question of what optimisations are possible given this important class of parallel applications. Particularly, one of the interesting questions, is that if the underlying scientific or numerical solution permits, what the optimal software design for this class of application is and how best it can be run on a particular HPC system. Moreover, we want to develop tools that address performance questions posed by application domain scientists who are required to develop their own wavefront codes and need to understand the optimal design possibilities, given their scientific and numerical boundaries. The aim is to develop techniques that not only answer the above questions, but also motivate and expose new questions and at the same time enable us to obtain solutions in a speedy, efficient and low cost manner.

Thus, motivated by the above key open issues, this dissertation develops analytic performance models that are generalised or *reusable* and demonstrate, their use for predictive analysis and optimisation of pipelined wavefront computations running on modern HPC systems. In the remainder of this section, we discuss the incentives for using analytic methods and motivate the idea of reusable performance model development.

1.1.1 Analytic Modelling

An analytic performance model is a mathematical construct that represents key aspects of a computer system and/or program. The analytic model is made up of parameters that abstractly represent the inputs, outputs, and the system. Given the required input parameters, performance predictions are extracted as solutions of this mathematical expression. It is one of the four broad performance engineering methodologies. The others being Benchmarking (or direct measuring/monitoring), Statistical Analysis, and Simulation.

Performance measurement or monitoring is the most direct and straightforward form of performance analysis. In the simplest case, a program of interest (a collection of programs or a workload) is executed on the system of interest and quantitative values such as execution time and floating point operation rates are monitored. The workload plays an important part here. In order to compare systems the workloads executed on them should be compatible for comparison. A benchmark is such a workload that is developed to assess systems. Also related to benchmarking is the use of profiling tools that enables one to instrument a code and extract measures and behaviours of the code as it runs dynamically.

Based on measurements taken on a working system, statistical methods can be used to understand expected behaviour of systems. Statistical methods such as regression and curve fitting are used to make speculations for possible future systems. For example data from a relatively small cluster running an HPC application may provide an initial idea of how the application will scale to a larger cluster made up of similar nodes and communication channels.

In contrast, simulation techniques seek to capture application behaviour by executing a representation of the application's control flow, communication structure and synchronisation behaviour. In effect they model the application running on a real-world system as a computer program, also called a simulator. The events that occur during execution, such as computation, memory access and communications, are represented as events in the simulator. Simulation allows one to model the events of the system at any level of detail from low-level instruction simulations to high-level transactions.

The previous performance engineering work for wavefront applications consists of research, based on all of the above methodologies, where in most cases they employ multiple or a hybrid of these techniques. Especially for analytic and simulation based methods, there is a need to benchmark a system to obtain initial performance behaviour, system parameter values etc. Additionally, the initial insights gained from some basic statistical technique can be an integral part of model/simulator development, testing and validation.

In [35], three requirements that a performance evaluation technique for parallel computing must accomplish are detailed. (1) Provide an understanding of the fundamental principles of parallel program behaviour and their impact on program performance, (2) enable the evaluation of the performance of a particular program on a particular system and thus obtain insights that can suggest potential improvements and (3), predict the impact (on program performance) of design changes in the program or changes to the underlying system or system configuration.

In this dissertation, we use analytic modelling to assert all these requirements in our performance engineering research of wavefront codes. Previous research using analytic modelling, that comprehensively satisfies the above requirements can be found. For example in [36, 37, 38] we see how the development of performance models of several scientific applications yields insights into the parallel program behaviour. In [8, 14] the analytic models expose possible improvements and identify bottlenecks, while in [25] they provide the ability to predict both qualitatively and quantitatively the performance of the application when the underlying system is changed. Analytic models provide the most insights into the various parameters and their interactions. It is possible to use simulation to search the space of parameters for the best possible combination but usually the trade-off between the parameters is not clear [39].

Further motivation for using analytic modelling extensively in this research is its low cost when compared to direct measuring and benchmarking, the speed in which models can be evaluated when compared to simulation, the flexibility when compared to pure statistical methods and the high levels of accuracy that can be attained. During the course of this thesis, the results will show further evidence for the advantages of utilising this methodology. This is particularly true in facilitating the development of reusable performance engineering characterisations for wavefront computations by, (1) enabling one to gain an understanding of which performance optimisations are possible, (2) determining quantitatively how much benefit each possible optimisation is likely to attain, and (3) allowing one to evaluate qualitatively which combination of optimisations are worthwhile.

Although the majority of this work uses analytic models, it should be noted that we supplement and reinforce our findings using both application benchmarking, in the form of actual validations, as well as results obtained by extensive simulations of these codes. Multiple evaluation techniques, including simulation studies, are invaluable when an actual system is unavailable or infeasible for validating the analytic predictions. When making critical decisions such as in the procurement of highly expensive HPC systems, assessment of the suitable system requires guarantees of predicted performance through multiple methodologies.

1.1.2 Reusable Performance Models

Performance Engineering - the understanding, analysing, evaluating and predicting of performance - on HPC systems is a highly complex task. An HPC application's performance on a high-end computer system is determined by many variables including, but not limited to, application algorithm, compiler, compiler optimisations, operating system, memory hierarchy and network. Thus it is difficult to find the optimal convergence point of all these variables. Furthermore, when the requirement to be cost effective and economical in managing the above resources to obtain the best performance per unit cost is included into the evaluation, the undertaking becomes significantly complex. Three additional concerns make this task even more laborious and demanding:

Firstly, the formidable size of HPC application codes demands analysis of many hundreds, thousands and hundreds of thousands of lines of source code, in many cases written using a number of different programming languages and development tools, incorporating many third party libraries and legacy codes. This imposes considerable demand on the performance engineer in terms of time, effort and resources.

Secondly, the pace at which the computing industry changes, not only the hardware and software technologies, but also the practices and sometimes the semantics is difficult to follow and elucidate in terms of their affect on overall performance for a given application. Incorporating new technological developments into previously understood performance characteristics for an application is highly non-trivial as now the performance engineer needs to re-evaluate the new characteristics and examine methods of extrapolating performance profiles. The next generation of development will demand a repeat of this cycle. Thus the performance engineer is forced to pursue a moving target.

Thirdly there is generally a significant lack of performance engineering techniques and expertise. This is compounded very much as a result of the previous concerns. Moreover, the knowledge of performance engineering has not usually been the forté of domain scientists who are most often concerned firstly with solving the scientific or engineering problem and then second in improving the performance of the application.

Therefore, there is a significant need for developing efficient performance engineering methodologies and expertise.

In a recent white paper *The Landscape of Parallel Computing Research: A View From Berkeley* [40] the idea of using a number of computation-communication patterns (called dwarfs) that commonly occur in HPC codes as an abstract set of behavioural patterns was presented. The concept is to not be overly specific to individual applications, implementations, optimisations or hardware platforms, but to use the computation-communication patterns to draw an abstract view of their performance and underlying hardware and system requirements. The white paper [40] argues that these computation-communication "dwarfs" have consistently appeared in parallel applications persisting the significant technological changes that have occurred since the inception of this field. As an example of the existence of ubiquitous computation-communication patterns, the authors of [40] present the existence of numerical libraries such as BLAS [41] and LAPACK [42] which encompass commonly occurring code as *reusable software*.

Inspired by this idea, we believe that commonly occurring computation-communication patterns provide an excellent basis to develop reusable performance characterisations. During the course of this thesis we show the significant benefits and flexibility of developing such characterisations, particularly as an ideal tool that aids exploration of the key open issues that has motivated this research.

1.2 Thesis Contributions

More specifically, the principal contributions of this thesis are as follows.

- We develop a reusable (plug-and-play type) analytic model that reflects the functional behaviour of existing and imaginable wavefront computations that use MPI on a regular orthogonal 3D grid of data. The model, based on LogGP [22], uses a small number of input parameters to specify the particular behaviour of most (if not all) wavefront codes. A key advantage of these parameters is that they are neither complex nor difficult to obtain. The new parameters and model equations capture the key structural and behavioural differences among different wavefront application codes, providing a succinct summary of operation for each application and insights into alternative wavefront application design. To our knowledge, this is the first *plug-and-play* reusable model for wavefront computations and perhaps the first such model for a class of HPC applications with a highly complex computation, communication and synchronisation pattern.

- We extend our reusable models to address issues that arise when performance predicting wavefront applications running on multi-core (CMP) nodes. This includes a more precise model of message contention on multi-core nodes, particularly related to a large Cray XT4 system, giving significant insight into performance bottlenecks on CMP nodes and possible solutions to alleviate them. Additionally, we develop simple extensions to the reusable model so that it can be applied to model wavefront codes based on irregular grids of data, HPC systems with heterogeneous resources, as well as 2D grids of data.

- We apply the reusable plug-and-play model to three different 3D wavefront codes of interest - NPB-LU, ASC Sweep3D and AWE's Chimaera. The resulting model for Chimaera is the first analytic model for this application. Then the models are validated on several systems including a large Cray XT3/XT4 and an InfiniBand commodity cluster. Model Predictions show high quantitative accuracy (less than 20% error) for all high performance configurations of LU, Sweep3D and Chimaera and excellent qualitative accuracy.

- As part of the modelling process and for validation, we develop highly accurate MPI communication models for on-chip and off-chip Send, Receive and All-reduce operations for the Cray XT3/XT4, and an InfiniBand based Cluster. These models provide insights into the MPI protocol operation on the respective networks and contribute to elucidating the contention arising on CMPs particularly on the Cray XT3/XT4. The communications models can be re-used in other analytic models that model applications running on these machines and use the same communication operations.

- Using the model, we present applications, projections and insights for optimisations that shows the utility of reusable analytic models for performance engineering HPC codes. Particularly, we demonstrate its ability to assist in evaluating (1) software configuration performance, (2) hardware platform questions including platform sizing, configuration and a case study that uses the performance engineering insights to assess system procurement decisions, (3) hardware platform design alternatives such as the optimal number of processor cores per node, and (4) optimisations and performance bottlenecks for wavefront computations demonstrating quantitatively and qualitatively the performance improvements from the optimisations. We validate these with the use of a discrete event simulator. The bottleneck analysis shows the most important contributors (hardware and software) to a wavefront code's runtime. To our knowledge, this is the first such extensive optimisation study conducted for wavefront applications.

- Finally, we provide further evidence for the validity of the results from the analytic models for the predicted performance behaviour of wavefront codes, by using a contrasting performance engineering methodology: namely simulation. To that end, we demonstrate an additional advantage of our analytic model development process, in which the insights obtained are used to enhance and drive the development of automation and simulation systems. In this context we show how the PACE [12] system is enhanced to performance predict wavefront computations on modern HPC systems and the development of the new WarPP [43, 6] simulation toolkit and the utility of analytic models to aide its design. We increase the predictive accuracy through coarse grained computation simulation as opposed to fine-grained (instruction level) simulation and in turn enable the simulation of over 100K processing elements in tractable time. We use the WarPP discrete event simulation systems to present the alignment between the analytic model results with that of predictive simulation, demonstrating further validations of the insights gained in this research.

1.3 Thesis Limitations

Computer performance is measured in various forms. In many scientific and engineering problems, it is concerned with the wall clock time taken for an application to run on a computer system or more specifically the time to solution. This could also be interpreted as throughput or the number of solutions per time unit. Other important measures involve, speedup of an application (which is the ratio of the time to solution on a single processor to the time to solution of a parallelised - or enhanced version of the same program), resource usage (such as the amount of memory, disk space used or the rate at which power is consumed) and the rate at which operations are performed (such as transactions, mathematical operations, etc). Additional performance measures include measures related to fault tolerance, reliability and dependability. Finally, all of these criterion are usually viewed in the context of cost and economic viability or more specifically as the performance per unit cost.

This thesis is solely concerned with the performance measures of (1) time to solution as well as throughput, (2) scalability including speedup and efficiency. Performance related to power consumption, system reliability or dependability are not addressed. Additionally, we only discuss performance per unit cost in a qualitative context. However, recent applications of this work and feedback from the sponsors of this research have shown that the techniques developed in this research can be readily used to obtain specific evaluations of cost, for example during the procurement process for an HPC system.

1.4 Thesis Overview

This chapter detailed the underlying goals, open questions and motivations for the research presented in this dissertation. The remainder of this thesis is organised as follows:

Chapter 2 presents a detailed account of (1) the basic concepts, components, terminology and implementations of high performance computing, (2) the basic laws and principles related to performance engineering and (3) the state of the art in performance engineering methodologies for HPC. In general this chapter provides a survey of the literature related to HPC performance engineering and provides a critique of the existing methodologies.

Chapter 3 details the operational behaviour of pipelined wavefront computations as a mandatory preliminary to develop a general *reusable* analytic performance model. We investigate three 3D wavefront codes of interest - NPB-LU, ASC Sweep3D and AWE's Chimaera. We describe in detail the research carried out in support of understanding of the complexities of these computations in terms of both the algorithm and the application execution on a target platform.

Chapter 4 develops the reusable (plug-and-play) analytic model that reflects the functional behaviour of existing and imaginable wavefront computations that use MPI. This chapter also includes details of the development of MPI sub-models that characterise the message passing performance for the Cray XT3/XT4 system (Jaguar) at the Oak Ridge National Laboratory (ORNL) in the US and representative validations of the reusable analytic model applied to NPB-LU, Sweep3D and Chimaera on this system.

Chapter 5 uses the analytic model to investigate speculative performance of wavefront codes on the Cray XT4 system including (1) software configuration performance, (2) hardware platform questions including platform sizing and configuration (3) hardware platform design alternatives such as the optimal number of processor cores per node (4) application bottleneck analysis and (5) application redesigns to alleviate bottlenecks.

Chapter 6 details an alternative investigation of the performance of pipelined wavefront computations through the use of a discrete event simulator. As part of this work, the application of the insights gained through analytic model development for enhancing the predictive accuracy of a discrete event simulator system is presented.

Chapter 7 further demonstrates the utility of the models developed in this research by analysing the performance of several key optimisation possibilities which include model extensions and a comprehensive bottlenecks analysis in the context of answering performance engineering questions to assist system procurement decisions. The accuracy of the results of this chapter is reinforced by detailed predictive simulation.

Finally, Chapter 8 summarises the conclusions of the research, and details several related questions that could be addressed in future research.

2 Performance Analysis and Prediction

2.1 Introduction

In order to provide a clear context and a consistent perspective for this work, in this chapter we present a detailed account of (1) the basic concepts, components, terminology and implementations of high performance computing that are inherently based on parallel computing and parallel programs, (2) the basic laws and principles related to performance engineering and (3) the state of the art in performance engineering methodologies for HPC at the time of this research.

2.2 Parallel Computing and Parallel Programs

Any computing system that makes use of simultaneous execution of some set of instructions on data can be classified as a parallel computer system. Since the inception of the modern digital computer, computer architects have been exploring parallelism and concurrency at many levels as a form of improving efficiency and increasing the throughput of systems. The idea is that large problems can be separated into smaller parts and these smaller parts can be solved simultaneously, or in *parallel*, usually using multiple processing units or entities. In modern HPC systems, high performance is almost always achieved by some parallel computing methodology. Therefore in this dissertation, all systems or applications termed to be in the domain of high performance computing are implicitly taken to be a subset of parallel computing unless stated otherwise.

There are several forms of parallelism. Bit level parallelism - is concerned with increasing the processor word size, which then, for instance, reduces the number of processor clock cycles needed to carry out instruction movements between processor and memory [44], pp15. The shift to 64 bit words from 32 bit computing is an example of increasing bit level parallelism.

Instruction level parallelism (ILP) - re-orders computer instructions such that they can be combined to execute in parallel without changing the result of the program. A well known form of ILP is the use of pipelining where the basic fetch-decode-execute cycle of successive instructions are overlapped by delegating each stage to a separate hardware unit. Other examples for ILP methods are superscalar execution, where multiple execution units are used to process instructions in parallel, out-of-order execution of instructions, speculative execution, in which instructions are executed before being certain of its occurrence and branch prediction which involves for instance, taking a conditional branch to avoid blocking control flow.

Data Parallelism - allows a computer to distribute data across several processors that can then be processed in parallel, for example vector operations. Alternatively, task parallelism

- distributes different tasks or functions across processors to be executed in parallel. When parallelising an application or when writing parallel code, it is data and task parallelism that are dominantly exploited either by a parallelising compiler or a programmer, while bit level and ILP are well exploited by modern compiler technology on single processor architectures.

2.2.1 Parallel Computing Architectures

Considering the systems that are used for parallel computing, a broader and abstract classification was provided in Flynn's taxonomy [45]. Flynn categorises computer architectures based on whether it operates on multiple instructions and/or multiple data. The simplest sequential computer is observed to be a Single Instruction stream, Single Data stream (SISD) system where one instruction stream operates on one data stream at any instance. The second category - Multiple Instruction stream, Single Data stream (MISD) can be viewed as ILP based on the pipelining method. But it can be argued that pipelining changes the data stream at each stage and therefore is not a single data stream. The pure form of MISD has not been explored significantly and as such is not implemented in modern systems. Data Parallelism is classified into the Single Instruction stream, Multiple Data stream (SIMD) where an operation of a single instruction is applied on many data elements, similar to a vector operation. Finally Multiple Instruction stream, Multiple Data stream (MIMD) covers the case where different instruction sets are operated on different (independent) data sets.

While SISD categorises non-parallel systems and MISD represent only a limited level of parallelism or represents a class of architecture that has not been explored or have proved to be note-worthy to solve modern problems, it is SIMD and MIMD architectures that have become the prevalent parallel architectures in modern HPC systems. Even then MIMD machines are by far the most dominant as the SIMD operational model could be easily implemented on MIMD machines. However, specialised SIMD systems do exist for instance in the form of vector machines that are developed solely to solve inherently data parallel problems. The MIMD architecture, on the other hand has been the most popular parallel architecture due to its sufficiency to support all forms of parallel programming models.

MIMD architectures can be further classified based on the distribution of memory hierarchy. Shared memory MIMD machines are the earliest form, where one block of memory is shared between many processing elements. All processors share a global address space and shared data variables are operated on by many processors using consistency mechanisms and synchronisation to arrive at the desired solution. When implementing shared memory systems, the simplest have been the Symmetric Multi Processor (SMP), also called Uniform Memory Access (UMA) where each memory location takes the same amount of time to be accessed by any processor. With the advent of multi-core processor chips (Chip Multi-Processors, CMPs), SMP style memory accessing is used when the processing cores access a shared main memory. But CMPs are mostly used as clusters to implement distributed memory MIMD systems. Alternatively the Non-Uniform Memory Access (NUMA) systems take different times to access different areas of memory. Due to the consistent memory access requirement, SMPs cannot be too large and range from about 2 to 64 processors. NUMA machines, on the other hand can be scaled to larger numbers of processors in the order of 1024.

The distributed memory MIMD implementation, in contrast, is built with each processing element having its own local memory. Processors have to then communicate with each other through a network passing explicit variables during execution. Distributed memory machines scales to larger numbers of processors while at the same time remain relatively cheap. The disadvantage is that the latency to communicate variables is higher than on a shared memory architecture. This latency is dictated by the interconnection network. Based on the level of coupling or distribution of processors, distributed memory MIMD systems, range from the Massively Parallel Processor (MPP) Systems, computer clusters to computer Grids. MPPs are closely coupled systems where a large number of processors, typically from 1K to more than 100K are interconnected by a very high speed network. Each processor works in tight synchrony with each other and the interconnection is highly reliable. Clusters on the other hand share similar characteristics to MPPs but the interconnect is generally slower and there are fewer processors than an MPP machine. A typical example of a low-end cluster is a Network of Workstation (NoW) where several workstations are grouped as a MIMD system, to perform parallel processing utilising a LAN. A popular system architecture that's commonly used in modern HPC is clusters of SMPs, or the more recent version - a cluster of CMPs. Each node in these systems is made up of a SMP or CMP while the nodes are interconnected by a very high speed network. Both MPPs and clusters are usually located in one location such as a single room or building. In contrast Grid computing interconnects systems spread across many buildings or are in different cities, countries or even continents. Processors in such a widely distributed system are much more independent than in MPPs or a cluster. Communication between them is kept to a minimum as the interconnection latency and bandwidth between two different locations on a computer Grid are relatively poor.

2.2.2 Parallel Programming Models and Languages

In addition to the parallel architectures classification, the programs written for these systems can have several parallel programming models [44],pp26-47. These models exist as an abstraction of the above hardware and memory architecture and are not specific to any parallel architecture. Theoretically they can be implemented on any underlying hardware.

The earliest form of parallel programming was achieved through the Shared Memory programming model. Tasks share a common memory space which facilitates communication between tasks. Data consistency and synchronisation is managed by well understood mechanisms such as semaphores, critical sections and locks [46, 47, 48] avoiding, for instance, race conditions, deadlocks and starvation. Parallelism is achieved by a parallelising compiler that uses *clues* provided by the programmer as compiler directives, and automatically generates parallel tasks that can be assigned to multiple processors. The OpenMP consortium [49] has developed the most popular programming specification for the shared memory programming model. Compilers implementing this specification allocate parallel areas of the code such as independent loop iterateration to separate *threads* of execution. The multiple threads or groups of threads are executed simultaneously, where each thread or group is normally assigned to a single processor. The communication between threads is achieved through shared memory. As such, the parallelism is said to be *fine grained* and has low overhead.

The Message Passing programming model uses explicit data movements between tasks (also called processes) that are assigned to processors. The data movements are explicit in the sense that the programmer should specify which information should be conveyed between a specific sender and a receiver. However, in contrast to the shared memory model, there is no shared memory location that is accessible to all processors. Particularly each task or process operates on a separate block of memory. Message passing models have been usually implemented as a set of routines packaged into a library that can be called by sequential programming languages. The programmer is responsible for specifying parallelism. In this sense, message passing parallelism is coarser grained than shared memory parallelism. The most common message passing specification is the Message Passing Interface (MPI) [20] which can be considered the de facto industrial standard for production work. The commonly used MPI bindings are for the Fortran, C and C++ languages while bindings for Java, Perl, Python as well as .NET and MATLAB [50] have been implemented but have not yet been used significantly for production work. The pipelined synchronous wavefront applications investigated in this dissertation are all based on message passing and use MPI. Therefore significant consideration needs to be taken regarding the performance aspects of MPI operation on HPC systems, when developing these models.

Finally, the Data Parallel programming model uses a more strictly regulated form of concurrency where multiple processing units perform an operation on separate elements of data simultaneously. At the end of each parallel execution block an information exchange might occur to re-organise the data. Such a re-organisation is implemented on shared memory or by message passing. Therefore the data parallel model could be considered a more abstract model than shared memory or message passing. Such a programming model is said to implement a Single Program Multiple Data (SPMD) programming model. The data parallel programming model is implemented by languages such as High Performance Fortran (HPF) [51] and the OpenMP shared memory directives. Additionally, message passing programs can be written implementing the strict step-wise data parallel execution model.

2.2.3 Parallel Decompositions

Another related parallel programming concept is the parallel decomposition implemented in a program. The simplest of these is the trivial or embarrassingly parallel decomposition where it involves running the same sequential program on many processors without need for any dependency or communication between the tasks.

The data parallel decomposition takes a large data set and splits it into smaller parts and then distributes them across a set of processors. Each processor may then apply a set of instructions on its local data and communicate with other processors to exchange data that has dependencies in order to arrive at the desired solution. The set of instructions that is executed by all processors can be the same (Single Program Multiple Data - SPMD) or different (Multiple Program Multiple Data - MPMD). When a large data grid is split into regular sub-grids of data the decomposition is termed to be a regular domain decomposition. That is, each processor will work on an equal (or almost equal) number of data elements. If each data point in the global grid requires a similar amount of processing then we say that the grid is regular. If the number of grid points cannot be distributed equally (or almost equally) then we say that the grid is unstructured. In both unstructured and irregular grids, the amount of work done per processor will in most cases have large variations.

The task parallel decomposition (also called Functional decomposition), splits the problem according to different functions and assigns each one to a different processor to be run in parallel. An example of functional decomposition is pipelining where each stage is processed by an independent processing unit and the data flows through the stages to finally produce the desired solution. Other notable parallel decompositions include task farming, divide and conquer and speculative parallelism [52].

2.3 Performance Engineering Methodologies

From a performance analysis point of view, the type of application, programming model or parallel decomposition, which is best suited to be run on the various physical hardware systems varies considerably. For example, message passing programs can be run on both shared memory systems as well as distributed memory systems. On shared memory systems the messages are "passed" as memory copies. On the other hand, shared memory programming models can be implemented on distributed memory systems where each processor will access a virtual shared memory space. But in this case the shared memory program will perform significantly worse on the virtual shared memory than on an actual shared memory system as now communication between tasks/threads/processors are done via a network. Similarly for instance, the performance to cost ratio may dictate that a given data parallel program is best implemented on a MIMD architecture as opposes to a SIMD machine. A trade-off between runtime and scalability may dictate that for a small number of parallel tasks, a shared memory MIMD machine will give better performance for this application. While for larger numbers of tasks, a distributed memory system could appear to be far more cost efficient.

It is these types of trade-offs that a performance engineer must consider. Ultimately, the objective is to identify the architecture that gives the best performance-cost trade-off for a given HPC application. In addition, the insight obtained will enable one to answer more significant design questions such as the programming model and/or the data decomposition best suited for the underlying scientific and numerical algorithm. In this section we give a detailed account of the state of the art and discuss key established laws, methodologies, tools and techniques that aid the performance engineer in addressing these issues.

As noted in Section 1.1.1, performance engineering techniques fall in to four broad categories - benchmarking (or direct measuring/monitoring), statistical analysis, simulation and analytic modelling. Since the inception of parallel computing and then high performance computing in the 1960s and 1970s, there has been a wealth of performance engineering research and studies conducted using techniques based on the above categories.

We begin our discussion of previous performance engineering work with two laws, namely Amdahl's Law [53] and Gustafson's Law [54]. These laws form two of the most significant principles in the design and analysis of parallel computers. This will be followed by the most notable early attempts at developing a framework for designing and developing parallel programs. Then in section 2.4 we return to the discussion of the four categories of performance engineering with detailed examples of significant previous research. We believe that this will provide the necessary definitions for the key terminology used later in the dissertation, and provide a background and a clear perspective to the contributions made in this thesis.

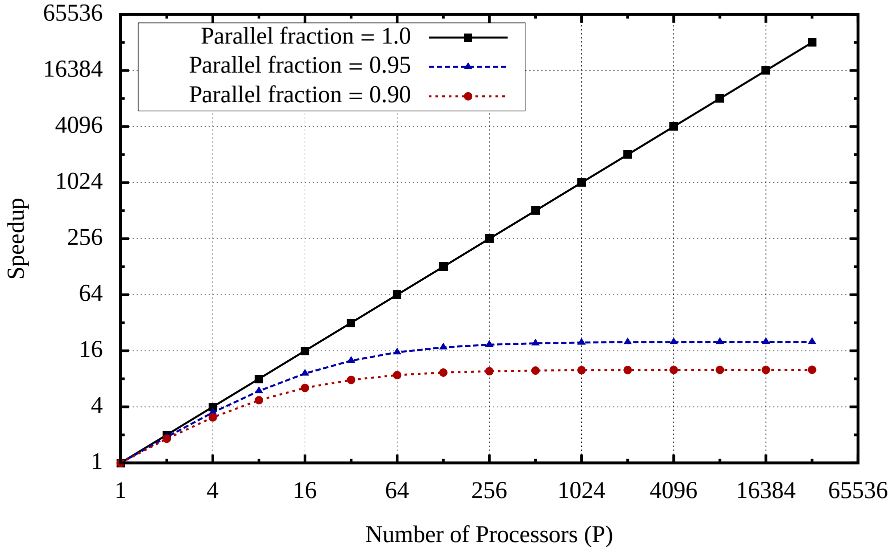


Figure 2.1: Speedups projected by Amdahl's law

2.3.1 Amdahl's Law and Gustafson's Law

In 1967, Amdahl presented a paper [53] that states the maximum limit of the speed improvement that can be achieved in a parallel program. The parallel speed improvement, is usually given as a ratio based on the runtime of the parallel version and the run time of the serial version of the program. This is called the speedup, and is given in (2.3.1).

$$Speedup(n) = \left(\frac{Execution\ time\ of\ the\ serial\ version\ of\ the\ program}{Execution\ time\ of\ the\ parallel\ version\ of\ the\ program\ on\ n\ processors}\right) \tag{2.3.1}$$

A related measure to speedup is parallel efficiency, which is obtained by dividing the speedup by the number of processors.

$$Parallel\ Efficiency(n) = \frac{Speedup(n)}{n} \tag{2.3.2}$$

Any parallel program can be viewed as having two parts: a potentially parallel part and an inherently sequential part. Amdahl's law states that the maximum speedup that can be achieved in a parallel program is limited by the runtime of the sequential portion of the program as given in (2.3.3).

$$Speedup(P) \leq \left(\frac{1}{f_s - \frac{f_p}{P}}\right) \tag{2.3.3}$$

P is the number of processors, $f_s + f_p = 1$ and f_s represent the fraction of the sequential portion of the program. It is apparent that when P goes to infinity - i.e. when the parallel portion can be infinitely parallelisable, the speedup is still limited by the sequential runtime. The conclusion from [53] therefore was that "the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude". Thus when f_p/P is small compared to f_s the bottleneck is the serial portion of the program.

Amdahl's law is not specific to constant problem sizes, although it has sometimes been interpreted over the years in such a way. In this case it assumes that the problem size or the portions of the program that is serial and parallel remain constant. Then (2.3.3) predicts diminishing returns when improving the parallel section's runtime. Theoretically (2.3.3) shows that the maximum speedup that can be achieved on P processors is P - i.e. linear speedup and is obtained when the limit f_s reaches zero. Figure 2.1 shows the diminishing returns projected by Amdahl's law and the ideal case of obtaining linear speedup when the serial part of the program is negligible.

There are, however cases when $superlinear$ speedup is achieved. For example, this can occur when the parallelised code improves the serial execution of the code. In other words the speedup is not only due to the parallelisation (i.e. increase of processors and distributing the work among them). The advantage due to parallelisation still remains to be linear at maximum. A typical example would be the cache effects of large distributed memory systems. In this case a large data set is distributed across P number of processors. As P increases the amount of data assigned to one processor decreases to the point that all of it may fit into the high speed cache memory on the processor. If the reduction in memory access is greater than the sum of the other overheads (such as communication cost increase due to communicating among n processors) one will observe superlinear speedup.

Another metric that's related to speedup, is scalability. Although a rigorous scientific definition for parallel scalability has yet to be defined [55], intuitively in the context of parallel computing, a program that gives improved performance proportional to the increase in the number of processors can be considered to be a scalable program. The level of scalability depends on how close the proportionality is linear (i.e. linear speedup). If linear speedup is achieved we say that the program is highly scalable.

For a fixed problem size, Amdahl's law, presents a view that with the increase of P, the serial portion's influence on the runtime becomes more and more dominant. This gives a very pessimistic view on parallel processing where we see that no amount of parallelising will improve the performance beyond how fast a processor can execute the serial portion of the program.

But if the parallel portion is allowed to grow with the number of processors - i.e. have a growing problem size, then we see benefits of parallel computing in a much more optimistic light. In 1988 Gustafson detailed this in [54] and this is now known as Gustafson's law. If the parallel portion is increased in proportion to the number of processors then the speedup is given by (2.3.4).

$$ScaledSpeedup(P) = \left(\frac{f_s + Pf_p}{f_s + f_p}\right) = f_s + P(1 - f_s) \tag{2.3.4}$$

From (2.3.4) if the serial part f_s reduces compared to the parallel part when the problem size increases, then speedup (or Scaled Speedup in this case) reaches P. For example, the serial portion can be observed to be reducing relative to the parallel portion, when the problem size per processor is set to a constant while increasing the number of processors, i.e. the total problem size increases with the number of processors. This is called *weak scaling* in parallel computing literature, where now the serial portion remains constant due to the fixed problem size per processor. Alternatively, the fixed total problem size being solved by an increasing number of processors is called *strong scaling*. In fact Gustafson's law only brings to light an aspect of Amdahl's law that has been misinterpreted.

2.3.2 Parallel Random Access Machine (PRAM) Model

Early parallel programming efforts were machine specific and had very limited portability [44], pp190. A program was essentially specific to the operational characteristics of the hardware. Therefore the development of parallel programs was limited by not having a general model such as the Von Neumann model [56] for serial computers. One of the earliest efforts of developing a generic model of parallel programming was the Parallel Random Access Machine (PRAM) model [57].

The PRAM model represents parallel computing in which a number of processors P operates in synchrony, can make simultaneous access to a block of shared main memory. The number of processors is unbounded where each processor is ranked as $P_0, P_1, ...$ and so on. Similarly the blocks of memory are unbounded, where any memory location is accessible by any processor in uniform time. Additionally each processor has unbounded local memory, a program counter, and a flag indicating the status of the processor (i.e. processor is running/not running). Communication costs (or memory access times) are considered to be zero.

A processor's cycle of execution consists of (1) read from shared memory (2) do local computation and (3) write to shared memory. This cycle may be executed by any processor concurrently. Multiple processors reading the same shared memory location are allowed but the basic PRAM requires conflict resolution rules to deal with simultaneous reads and writes to the same memory location by different processors or multiple processors writing to the same memory location. The conflict resolution policies have defined four sub-models that extend the basic PRAM - Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), and Concurrent Read Concurrent Write (CRCW). EREW allows only one processor at a time to read from or write to a memory location, while CREW allows multiple reads and only exclusive writes. CRCW has further conflict resolution policies that dictate for instance which processor gets to write to a memory location first, e.g. priority based on rank of processor, validity based on writing modified or unmodified data [58, 59].

The PRAM model exposes the inherent concurrency when designing a parallel application by giving an abstract view of a general parallel machine [44],pp191. That is, it corresponds intuitively to the programmers' view of a parallel computer and hides specific machine and programming details of the real world. For this same reason the many assumptions in the PRAM model are unrealistic in terms of real-world systems. For example communication and data access costs are not zero in real systems and often can dominate a run time. Additionally the uniform memory access time may only be true for small number of processors sharing memory (such as an SMP) while large distributed memory machines that violate this may be the only way to scale to large number of processors. The assumption that processors operate in complete synchrony is also unrealistic. Thus from a performance engineering stand point it fails in providing an accurate performance profile of the program.

Several additional extensions have been made as an effort to make the PRAM more realistic. Papadimitrio and Yannakakis [60] provide a latency PRAM model by accounting for the time for memory accesses, while Local-Memory Parallel Random Access Machine (LPRAM) [61] accounts for communication bandwidth. A similar model, accounting latency and bandwidth is provided by Aggarwal at. el in [62] called the Block Parallel Random Access Machine (BPRAM).

Despite these improvements, the PRAM model remains too abstract to be representative of real systems in a performance engineering context. But it has been widely used in parallel algorithms design and complexity analysis due to its simplicity. As such, the PRAM should be viewed as a model best suited for program and algorithm design.

2.3.3 Bulk Synchronous Parallel (BSP) Model

The BSP model [63] attempts to provide a more realistic parallel machine to the programmer than the PRAM when creating parallel algorithms. A more complete treatment of the BSP model is provided in [64]. We summarise the model here due to its importance to performance engineering as the first model that enables one to realistically estimate the performance of parallel applications particularly based on message passing, running on real hardware.

The BSP model's aims are similar to that of the PRAM in that it details a style of parallel programming developed for general purpose parallelism [65]. A BSP machine consists of a number of processors P, each with their own local memory. The processors are connected via a network. The BSP model forces a parallel program to be developed in a rigorously structured format as shown in Figure 2.2.

A BSP program progresses in a control flow consisting of *super steps*. Each super step consists of (1) computations on local data, (2) necessary communications between processors and (3), a barrier synchronisation that ensures that all communications are completed before the next local computation step [64]. The separation of computation and communications makes for a structured (simple) approach to building a parallel program. At the same time the BSP machine is abstracted from machine specifics and thus a program developed in the BSP style can theoretically be implemented to run on any parallel architecture/machine. The structure of the super-steps is easily predictable analytically for a given architecture. For example a very simple model for the time for a superstep would be the sum of the time for the longest local computation, the upper bound of the time to complete a communication and the time until the end of the barrier synchronisation.

BSP defines several parameters so that the performance (time to solution) of a program can be analytically estimated. The number of maximum incoming or outgoing messages for a processor P_i is given by h_i. This is called an h-relation. A parameter g is defined such that it takes hg time to deliver an h-relation. Thus if a processor sends h messages each of size m bytes then the time for these communications are given by mgh. Therefore g can be roughly thought of as the rate at which data is transmitted over the network and is determined by experimentation/benchmarking for a given parallel machine.

The time to solution of a super step is then given by (2.3.5) where w_i is the local computation time on processor i and l is the time for a barrier synchronisation (also determined

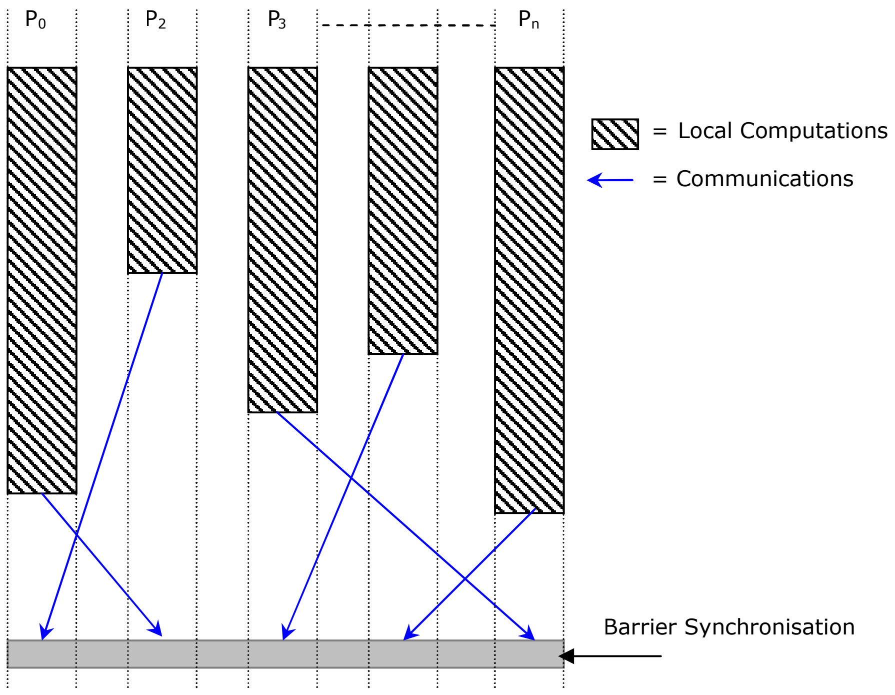


Figure 2.2: A superstep in the BSP model

empirically for a given parallel machine).

$$Time\ of\ a\ Super\ Step = MAX\{w_i\} + MAX\{mh_ig\} + l \tag{2.3.5}$$

The advantages of the BSP model over the PRAM model are that (1) it does not assume that all processors work in synchrony, (2) it accounts for communication overhead, (3) it is more general where the ideal PRAM model is a sub-model (when $g = l = 1$) of BSP.

As BSP is a model of parallel programming, it can be implemented using many programming languages and systems. Message passing libraries such as MPI [20] and PVM [66] can be used to implement a program that adheres to the BSP model. There are also BSP libraries such as the one provided by Oxford BSP Library [67], Green BSP Library [68] and the standardised BSPLib [69], that enables one to write a BSP program using library functions that are called by common HPC languages such as Fortran and C.

The strict super step structure may in some cases be viewed as a disadvantage when optimising a parallel program. For instance the ability to define a subset of processors communicating with each other while another subset computes on local data is difficult to describe within a super step. Also the frequent use of barrier synchronisations can degrade performance in systems. Although many modern HPC systems have implemented highly optimised barrier synchronisations, the super step cost equation (2.3.5) shows that the parallel program should be developed with a minimum number of supersteps so as to reduce the barrier synchronisations between super steps.

In spite of these minor issues, and several others discussed in Section 2.3.4, BSP remains popular when developing parallel algorithms. It is also the first model that enabled designers to obtain a realistic performance estimate of parallel programs.

2.3.4 LogP and LogGP Models

Motivated by several issues in the basic BSP model, Culler *at el.* detailed the LogP model of parallel computing in [70]. As mentioned before, the strict programming structure of the BSP model sacrifices flexibility and some performance (e.g. cost per barrier synchronisation) in favour of structured and predictable/verifiable parallel program design. The LogP model attempts to give better flexibility to the programmer. In addition it allows for optimising communication patterns by allowing more precise scheduling (as oppose to the BSP model, where the length of a super-step depends on the most unfavourable h-relation [70]) and using explicit message passing for synchronisation devoid of the BSP super step structure so that costly barrier synchronisations are avoided. Additionally LogP gives a detailed set of parameters that define abstractly the performance of a system and network. These parameters attempts to, "allow the machine designers to give a concise performance summary of their machine against which algorithms can be evaluated" [70]. LogP is more general than BSP, but it has been shown that LogP and BSP can essentially simulate each other without much performance difference [71].

In the LogP model, the parallel machine is represented as a distributed memory system where P processors (each having their own local memory) communicate with each other using point-to-point messages. The communications network performance is parametrised by the following parameters as detailed in [70]:

- L: the upper bound on the *latency* of the network - the flight time for a message from one point of the network to another.
- o: the *overhead* time taken by a processor to transmit/receive a message. i.e. the time take by a processor to release a message to the network or receive a message from the network.
- g: the *gap* defined as the minimum time between consecutive message transmissions or receptions at a processor. So $1/g$ represents the available per-processor communication bandwidth.
- P: the number of *processors*.

The LogP model assumes that the network has a finite capacity where at most only L/g messages can be in transit from or to any processor. If a processor attempts to exceed this capacity then it will stall until the network can accommodate the messages. The model can be modified by ignoring g if it is less than or close to equal to the overhead o or when the communication pattern of a parallel algorithm is such that it communicates infrequently. But it should be noted that when modelling most modern HPC machines/networks using LogP the g parameter is ignored as they can accept new messages into the network as fast as the processor can produce them [8]. I.e. these machines are said to have balanced networks.

Additionally the LogP model assumes that all messages are of a small size. A basic extension to the LogP model given by LogGP [22] models the case where long messages are involved. In this case G represents the *Gap per byte* defined as the time taken to transmit a byte on to the network. Then $1/G$ gives the available per processor communications bandwidth for long messages. Figure 2.3 gives a graphical view of these parameters.

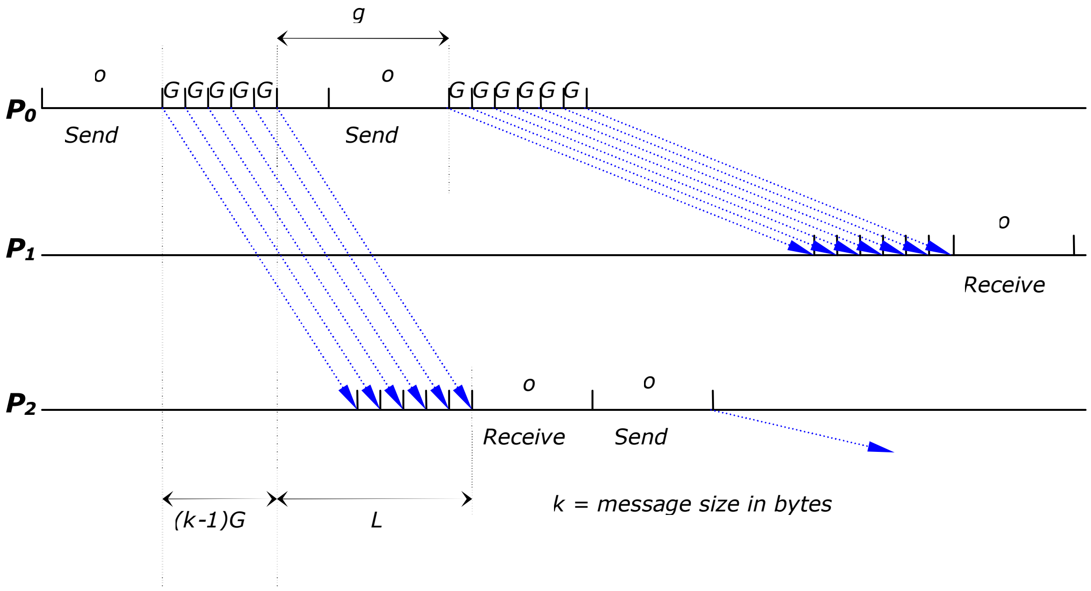


Figure 2.3: LogGP parameters

The LogP/LogGP parameters represent a simple and concise set of parameters aimed at providing a framework for developing parallel algorithms. As such it captures the majority of the parallel program design issues and performance while dropping uncommon or machine specific parameters. But at the same time, the parameters are detailed enough to encourage the design of parallel programs that avoid unrealistic assumptions (such as zero communication costs in the PRAM model) and expose possible optimisation techniques (such as precise scheduling of computation and communications so that communications latency can be hidden). Moreover, many extensions to the basic LogP/LogGP models show (e.g. LoPC [72] and LoGPC [73]) that these basic parameters can be augmented with other techniques such as for example queuing theory and mean value analysis (MVA) to represent more detailed behaviour of parallel programs and systems, particularly when used as a tool for performance analysis and prediction. This extensibility of the LogP approach has been an advantage for modelling complex applications and new technology trends. We note related work in section 2.4.4 and 3.3. The main contributions of this thesis are directly based on the LogGP method for analytic performance engineering.

2.4 Performance Engineering and the HPC Lifecycle

Performance engineering can be used throughout the HPC lifecycle. The HPC lifecycle, viewed as a hardware cycle in the perspective of the machine developers, vendors, end-users and system administrators and as a software lifecycle, viewed in relation to the domain scientists, engineers, programmers and code maintainers are detailed in Figure 2.4 [39]. Figure 2.4 is an extended version of the usual development stages of *specification, design, implementation, testing* and *maintenance*. Each stage may require revisiting a previous stage during real-worls

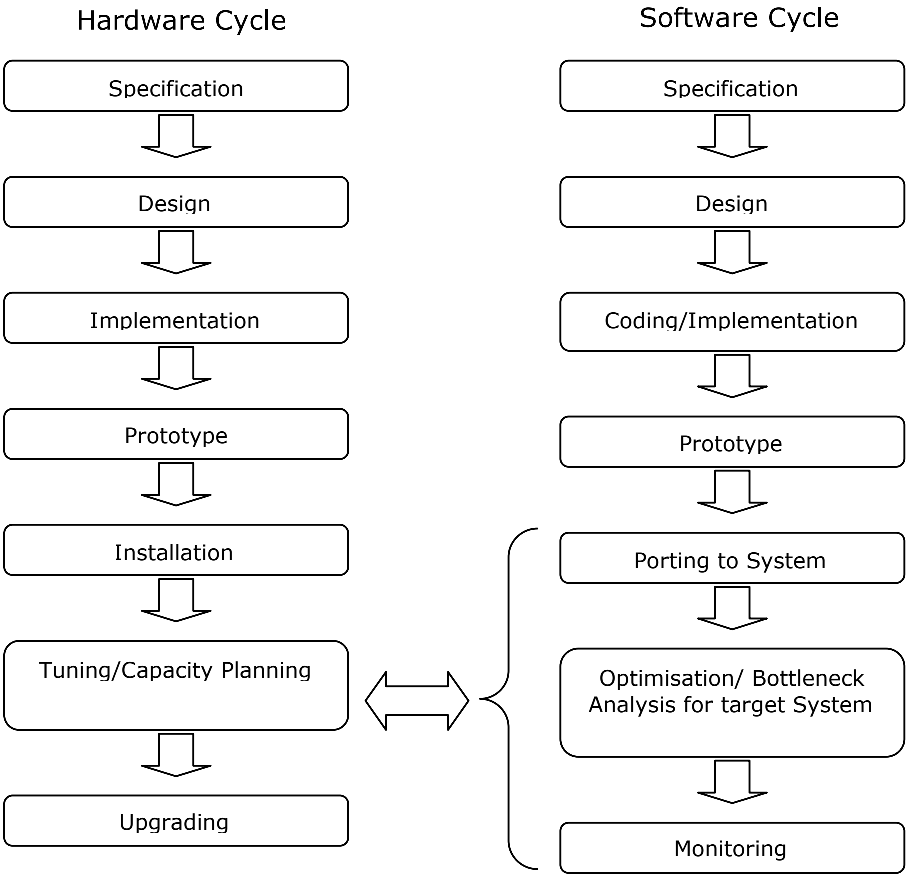


Figure 2.4: Stages in the HPC lifecycle

development. For instance, optimisation of an application may lead to re-designing the software or tuning/capacity planing will lead to a re-evaluation of the system specification.

Generally we can view a relationship of the software cycle with that of the hardware cycle at the post prototype and/or installation stages. For instance, software applications written previously by domain experts need to be ported to the new HPC machine or tested on it, followed by tuning and optimisation of the code. Capacity planning, including system sizing for a given application on the new machine, will allow for determining the best possible application configuration to execute it. Finally monitoring the application execution will give an important indicator of system maintenance requirements and possible upgrades.

Performance engineering can be used at each of the life cycle stages for both the system and the application. For instance, the parallel programming methodologies discussed in the previous sections, such as PRAM, BSP and LogGP, can be used at the specification and design stages of the application as well as providing verifiable models for performance prediction for example at post-procurement stages. Similarly, methodologies such as benchmarking can be used at post prototype or installation stages. In the remainder of this chapter we will discuss the state of the art in the four performance engineering methodologies - benchmarking/profiling, statistical analysis, simulation and analytic modelling - and their relative pros and cons including related research as well as their applicability to each of the activities occurring at both the HPC system and application life cycles.

2.4.1 Benchmarking and Profiling

A benchmark is a test workload that is run on a target platform to measure performance of system components [39]. Analysing the performance of a system or an application based on the direct execution of a benchmark is perhaps the most straightforward form of performance engineering. In the simplest case, you can monitor the performance of target applications by executing them directly on the said platform. Such a real workload has the obvious advantage of representing the actual working/production behaviour of target applications on a target system. But real workloads generally are either inflexible so that they cannot be repeated, consist of sensitive data/operations so they cannot be distributed, to vendors at procurement for instance, or require real-world inputs that may not be available.

Over the years workloads have been developed that represent important algorithmic kernels/operations ubiquitous in HPC applications, kernels that exercise various subsystems of a hardware system or are representative applications of a larger workload that can be used as metrics to measure the performance of a machine and in particular compare machines. Such regulated workloads provide the flexibility needed to carry out the measuring in a controlled manner. Quoting such *benchmark* measures gives a quantitative gauge about performance on a system.

The types of performance measured by benchmarks can range from low-level machine operation counts such as floating point operation rates, mixes of instructions, communication latencies/bandwidths of networks to total time to solution for an application. While the low-level measures indicate the performance of certain sub-units of a system, measures such as time to solution or transactions/completions per unit time, encompass the performance of all the sub-units of a system working together to arrive at a solution.

2.4.1.1 Low-level Benchmarks, Kernels and Microbenchmaks

Low-level benchmarks typically measure the performance of a subsystem of hardware in a machine. The typical important subsystems of an HPC machine are computation, communication, memory and input/output.

There are several quantitative metrics of computation. Processor clock speeds give a partial indication of computation, but do not usually translate to an accurate indication of processor performance due to computation performance being dependent on the memory hierarchy performance as well as processor speeds and superscalar features, not to mention compiler and compiler optimisations. A processor can only compute as fast as the rate at which it can obtain instructions and data from memory. Thus, metrics that gauge the amount of useful instructions performed per unit time are used. Millions of Instructions Per Second (MIPS) capture the throughput at which a processor will complete a mix of instructions, while Millions of Floating-Point Operations per Second (MFLOPS) count the number of floating-point operations. Floating-point instructions are particularly useful in the context of HPC as a majority of scientific and engineering applications are floating-point intensive.

Out of the low-level benchmarks, LINPACK [74, 75] is perhaps the best known for measuring limits of computation performance. LINPACK consists of a number of programs or *kernels* that solve dense systems of linear equations which are typical of an instruction set with a high percentage of floating-point operations [39]. LINPACK makes use of the BLAS (Basic Linear Algebra Subprograms) [41] libraries for the solution and produces a MFLOPS measure of the system. High Performance LINPACK [76] is a portable version of LINPACK popularly used in the top500 supercomputer [77] rankings. Other noteworthy computation intensive low-level benchmarks include the Livermore loops [78], the computation kernels from the HPC Challenge benchmarks [79] - DGEMM and FFT - and the SPEC CPU kernels [80].

There are many benchmarks that measure the communications performance of a parallel computer system. Low-level communication benchmarks (also called Microbenchmarks or low-level probes) usually perform a timed message passing operation between MPI processors. A communication system is usually characterised by its bandwidth and latency. Microbenchmaks, for example, allows one to obtain a measure for the latency and bandwidth of a network when performing purely synthetic point-to-point communications or collective communications. The myriad array of MPI benchmarks including the industry standard Intel MPI benchmark [81] (formerly PALLAS' MPI benchmark) and others such as MPPTest [82], NetPipe [83] and SKaMPI [84] are examples of communication micro-benchmarks. Similarly in shared memory parallel programming, OpenMP benchmarks such as the EPCC OpenMP benchmarks [85] measure the overheads associated with OpenMP directives on a given platform.

The memory performance is particularly important as processor throughput is directly dependent on the sustainable speed at which the memory can supply instructions and data to a processor. During the mid 1980s until about 2004, when processors clock speeds were growing exponentially [86], the memory performance was important due to its need to keep up with processor performance. Now in the multi-core era, memory performance has become important due to the need to feed the increasing number of cores on a chip continually with instructions and data so that useful work is produced. Memory performance is usually measured as a bandwidth (e.g. MBytes/second). Notable microbenchmarks include STREAM [87] that measures the sustainable memory bandwidth through a synthetic benchmark program and RandomAccess from the HPC Challenge benchmarks [79] that measure the rate of integer random updates of memory in Giga Updates per Second (GUPS).

2.4.1.2 Synthetic Benchmarks, Application Benchmarks and Benchmark Suites

While kernels and microbenchmarks give an indication of the performance of a system's subcomponent's performance, they give little indication of how the machine will perform realistically on useful workloads. *Synthetic benchmarks* such as Whetstone [88] and Dhrystone [89] were the earliest attempts at developing an artificial workload to obtain an indication of the total system performance. These replicated the behaviour of real programs based on statistics of operations [90]pp27 derived from programs in the 1970s and 1980s.

Application benchmarks consist of a reduced version of a production code and are derived directly from real workloads. Examples of important application benchmarks include the ASC benchmarks [91] (formally ASCI) developed to asses the performance of HPC machines at the US Department of Energy, NASA's Aerodynamic Simulation Parallel Benchmarks (NPB) [92] that contain codes representative of CFD workloads at NASA and the

SPEC benchmark suite [80] which encompasses a *suite* of benchmarks that also include low-level/microbenchmarks in addition to application benchmarks.

Application benchmarks are vastly superior and advantageous for demonstrating the achievable performance of a target HPC machine. Usually the time to solution (elapsed wall clock time) or the achieved throughput of the system (e.g. transactions completed) is reported at the end of a benchmark run. Furthermore application benchmarks can be designed so that on parallel systems an indication of the application's scalability can be observed. When comparing systems, application benchmarks usually provide rules that limit some optimisations/methods that give higher performance. This can be argued to be sidestepping the real ability of a system to demonstrate higher performance.

A recent evolution in measuring system productivity is the concept of Purpose Bases Benchmarks (PBB) [93]. PBBs try to address shortcomings of traditional benchmarking where the productivity of a system is measured in terms of an activity performed. Traditional benchmarking does not evaluate the system in terms of "useful work done". For example the LINPACK kernels solve the system of equations using less efficient Gaussian elimination and explicitly forbid the use of newer and efficient Strassen methods [94] as the latter invalidates the floating-point operation counts. The authors of PBBs argue that the productivity of the system should be measured by the "amount of progress towards a goal of real human interest" done by a system via the most efficient means possible on the target system. They provide a rigorous definition for a PBB in [93]. Although PBBs are not ubiquitous in current HPC benchmarking studies, it demonstrates an alternative and perhaps more useful assessment of computer productivity and performance.

2.4.1.3 Profiling

While benchmarking gives a relatively concise summary of the performance of a system, profiling gives a view of the program when it is executed. Profiling is concerned with gathering information during one or more runs of a program and presenting the dynamics of its execution which include for example, frequency and duration of function calls, frequency of operation counts within blocks of code, dynamic memory consumption and release, etc. A profiler collects data about the program through various means of code instrumentation, operating system monitors, hardware level monitors including interrupts and performance counters.

The activity of profiling, also sometimes called monitoring, provides several insights into a program's behaviour [39]pp111. It enables one to trace the execution path of a program (sometimes through off-line replay of the profile trace) allowing one to identify most frequently used or time consuming sections of the code, obtain the time spent in various subroutines/modules/operations of the code and assert the interdependence between variables. An important additional use includes assistance in debugging.

Profilers or program monitors can be classified based on measurement method, instrumentation technique and the type of profile output. The two methods of measurement are (1) Tracing - program events of interest will be logged synchronously by the tracing profiler as events occur (e.g. VampirTrace [95]), (2) Sampling - program state will be logged on demand or periodically by a sampling profiler using system timer facilities and performance counters (e.g. gprof [96], PAPI [97, 98]). The former tend to produce large amounts of data, while the latter can give a false picture if adequate sampling is not done.

Instrumenting a program for profiling can be done at the level of source code (before compilation) (e.g. PAPI [97]), at compile time (before linking) (e.g. gprof [96], CrayPat [99]) linking with a profiling library (e.g. VampierTrace [95], Opt [100], Pablo [101]), during runtime (through hooks in a runtime environment, operating system or hardware) (e.g. Paradyn [102]). Profiler suites such as TAU [103, 104] and KOJACK [32] allow one to instrument the application at a combination of the above stated levels.

A profiling output can be (1) a flat profile or (2) a histogram. The former give the summary of the execution by presenting the average behaviour of function calls/frequencies and average time spent in each during the run. The latter gives a detailed history of the execution path as a call graph including the caller and callee routines, times spent on each level and frequency of calling at each level.

Perhaps the best known profiler is the gprof [96] profiler belonging to the UNIX development tool-chain. gprof originally developed for analysing serial code (written in C, Fortran, Pascal or COBOL) provides both a flat profile and a call graph detailing the usage of CPU time. The program information is gathered through sampling. Proprietary parallel profilers such as Portland Group's PGPROF [105] and Cray's Performance Analysis Tool (CrayPAT) [99] serves similar functions to that of gprof and include many additional tools such as graphical tools for parallel programs and are scalable to programs running on large number of processors usually written in MPI and/or OpenMP.

For MPI programs notable tools include public domain profilers Upshot, nupshot, Jumpshot and MPE [106, 107] and the de facto industry standard MPI profilers - Vampire and VampirTrace [95]. VampirTrace, includes a library to be linked to a parallel program which will then trace the message passing events between processors, including message send/receives event ordering, function entry and exit points, message lengths and times. At the end of a profile run the trace file produced by VampirTrace can be analysed through the tools in Vampire including graphical visualisation tools that give timelines, state changes, communication event viewers in addition to the call graph type information about functions and subroutines.

In contrast to the tools mentioned so far, PAPI [97, 98] provides an API that enables access to the performance counters of modern micro processors. Using PAPI, the programmer can observe the dynamic view of the relation between software performance and low-level processor events such as floating-point operations, cache misses etc. The API allows one to manually instrument parts of the program or allows higher level software tools (e.g. TAU [103, 104], KOJACK[32], HPCToolkit[108]) to access performance counters.

The main disadvantage (among others discussed later in Section 2.5) of benchmarking and profiling is that they are limited to reporting the performance of systems/applications "as is". The measures obtained alone cannot be relied on for speculative studies for instance in predicting performance of future systems or imaginable/innovative new systems, or predicting performance of application optimisations. Another disadvantage of profiling is that the tools are sometimes quite invasive and can strongly influence the performance of the application yielding results which are misleading. Particularly, for answering "what if" questions benchmarking/profiling results must be used in conjunction with other performance engineering methodologies. We consider the first of these methods based on statistical analysis, in the next section.

2.4.2 Statistical Analysis

Statistical performance prediction involves applying techniques such as regression/curve-fitting and pattern recognition to speculate on future behaviour. While benchmarking and profiling gives measures of a system's and application's execution, statistical techniques can be used to analyse the gathered data to obtain predictions. The assumption is that the obtained historical data contains adequate information for forecasting performance and that the statistical equations and methods used can replicate this historical trend. This approach is widely used in performance analysis of systems such as computer Grids and for transaction heavy systems (such as e-business applications) where the level of complexity and operation granularity is difficult to predict using other performance analysis techniques.

While applying statistical methods such as regression and curve-fitting to empirical data is an obvious *first attack* technique for almost any performance engineering study, there are several established research projects that solely utilise statistical methods as their main prediction tool. These include, The Network Weather Service [109, 110], research by Dinda *et al.* [111, 112, 113, 114, 115], Vazhkudai [116] and frameworks such as NetLogger [117]. Statistical approaches are particularly ubiquitous in workload characterisation [118, 119, 120, 121] and in supporting application scheduling decisions. Many of the research projects described above are leveraged to improve job scheduling, planning and load balancing on various parallel high performance and distributed systems [122, 123].

The main limitation in statistical methods is that explanatory analysis of a system or application may not be captured or exposed by the analysis. For example scaling behaviour of an application may be approximated by statistical extrapolation, but the reasons for poor/-good scaling (i.e. bottlenecks) may not be understood. Even pure scaling studies based on statistical techniques alone, may prove to give inaccurate results if the application has a complex computation-communication and synchronisation patterns such as the wavefront applications discussed in this thesis. Therefore statistical methods are best used in conjunction with performance analysis based on simulation or analytic methods. In fact as we will show in this thesis, statistical techniques such as regression are an integral part of developing simulation or analytic predictive performance models. The state of the art in simulation techniques for HPC are discussed in the next section followed by analytic modelling in section 2.4.4.

2.4.3 Simulation

In the general sense a computer simulation or *a simulator* is an imitation of an actual system by a computer program. Simulation is a well established performance evaluation technique and has been widely used for performance engineering. Simulation is particularly important when designing a new system, for example new processor designs are extensively simulated before committing to develop a prototype. If the system to be analysed is not available, for instance during design or procurement, a simulation can serve to understand performance, compare alternatives or answer design questions [39]pp393. Even if the system is available, simulation may allow a systems designer to do a much more wide variety of experimentation than is possible with the actual system.

There has been and continues to be considerable research interest in developing simulators for performance engineering. One classification of simulators is based on whether the simulation is that of the system state (discrete-state) or that of system events (discrete-event). The former for example will simulate state changes at every clock cycle. The latter will maintain a time ordered-queue of interesting events (such as communication events, memory accesses) and trigger a simulated system response as each event is processed.

A complementary classification separates simulators based on the generation of events as - stochastic driven, trace driven and execution driven. Stochastic simulation uses a probability distribution to generate the events such as arrival of customers, transactions, operations; such simulators are useful, when detailed information of workloads are not available. In contrast, trace driven simulators require a trace of an actual run of the system. The simulator will then read the trace and simulate the events on the trace so that a performance engineering study can be conducted. Execution driven simulators on the other hand will simulate the application running on the target platform and produce the events that are to be simulated. This ensures that the trace corresponds to the one that would be obtained if that application were actually executed on the architecture being simulated.

Trace driven simulation has been widely used in sequential processor system simulation. But on a multiprocessor system the execution path is not necessarily the same on different architectures or even different runs [124]. Therefore the dominant parallel systems simulators are almost always execution driven. However, there are exceptions such as DIMEMAS [125, 126] that uses a trace from an application, taken at runtime. Using the trace, DIMEMAS can simulate the behaviour of the application on very modest computing resources, such as a workstation. A disadvantage of such a trace, considering any realistic HPC code, is the size of the generated traces, as well as difficulties in scaling to hundreds, thousands and even hundreds of thousands of processing elements for speculative analysis.

On the other hand discrete-event simulators are more efficient due to their focus on simulating only important events, as opposed to the whole system state. One of the first execution-driven (discrete-event) simulators is the Rice Parallel Processing Testbed (RPPT) [127]. In RPPT, a pseudo-concurrent execution of an actual parallel algorithm provides a more accurate and realistic model of the workload execution. RPPT combines sets of instructions between branch, loop or communication instructions and treats them as execution blocks reducing the types of events in the queue. This technique, called direct execution with augmentation, is much more efficient than instruction-level simulation that is found in sequential system simulators. Similarly, PROTEUS [128] and SPAM [124] provide direct execution with augmentation, where the former provides tools for non-intrusive monitoring and repeatability of the simulation, and generation of profiling information similar to prof [129]. The latter gives a simulation kernel and tracing tool that aids the development of execution driven simulators which require a huge amount of development effort. All these simulators are uni-processor host simulators.

PACE [12] and its successor WarPP [43] provide a layered characterisation approach that enables reusing hardware, parallel structures and application models. It uses a different type of execution driven (discrete-event) simulation in that the execution is that of a static characterisation of the application code and parallel structure. In PACE, sequential portions of the code are characterised through micro-statements, each of which are benchmarked on the target platform and included as a hardware model in the simulator. A more detailed discussion of PACE and WarPP is presented in Chapter 6.

To improve efficiency of the simulation, parallel simulators (i.e. simulation running on parallel hosts) were developed. Notable work includes the Wisconsin Wind Tunnel (WWT) [130, 131], MPI-SIM [29, 132] and BigSim [133]. Wisconsin Wind Tunnel is a shared memory architecture simulation engine while MPI-SIM provides a library for the execution driven parallel simulation of MPI programs. MPI-SIM is further discussed in Section 3.3 where its use as part of the POEMS [31] system in performance analysis of wavefront codes is detailed.

Although they are much more efficient than uni-processor host simulators, parallel simulators incur high overhead in managing the non-determinacy of the simulation. For example the message arrival order may need to be artificially preserved/rectified (e.g. through check-pointing, roll-back, forward re-execution) [133] on a parallel simulator due to the fact that there are multiple processors carrying out the simulation. Various synchronisation methods are used to correct this, resulting in high inefficiency for large systems simulations. Some techniques to reduce this overhead are used in simulators such as Parallel PROTEUS [134] and BigSim [133].

The main disadvantages of simulation are the amount of time required to generate simulation results for even the most conservative parallel system [39]. This time becomes even more intractable if there are many "what if" scenarios to be explored. But more importantly, system insights and bottlenecks may not be readily apparent from a simulation study, some times due to too much detail, and require additional effort to understand important performance issues. Nevertheless, simulation has proven to be an invaluable technique that complements the other performance engineering methods including analytical modelling. In some cases, simulation is the only feasible technique to analyse performance of the system.

The simulation studies specifically aimed at pipelined wavefront computations will be discussed in detail in section 3.3, while the enhancements to the PACE [12] simulation system developed by insights gained through analytical modelling and later used for validation of wavefront code optimisations will be detailed in Chapter 6. We detail notable analytic modelling techniques in the next section.

2.4.4 Analytic Modelling

An analytic model is a mathematical construct that represents key aspects of a computer system and/or program. The basis of a good analytic model is to develop expressions that focus on the important parameters and elucidate the behaviour of a system and the application but remove details that do not substantially affect to performance. Amdahl's law [53] and Gustafson's law [54] provide the most basic and important analytic models for parallel computing but have limited ability to elucidate the important behaviours and parameters of a parallel system and an application. We can view these laws as providing boundary values for the performance of the system.

The second type of analytic models used in performance engineering view the system as a network of queues where tasks share system resources such as the CPU, Memory, Network Interface Card (NIC), disks etc. This leads to the system being suitable to be modelled using queueing theory, particularly considering the non-deterministic (or stochastic) behaviour of jobs competing for service. Queueing theory enables one to ascertain the time each job spends in various queues, which then can be combined to predict the response time of the total system [39]. Examples of analytic models developed using queueing theory, including mean value analysis and stochastic models include [135, 136, 137, 138, 139, 140].

Another type of analytic model for parallel systems exploits the deterministic nature of parallel applications. It has been shown that we can assume determinism is a valid and accurate premise for most parallel executions [35]. Here, the author shows how deterministic values for mean task time (e.g. computation time) and communication time can be used, in conjunction with stochastic models, for shared resources or competing jobs to obtain performance predictions with significantly less complex models than pure stochastic models. Additionally, the deterministic assumption implies a unique execution sequence and that the time to solution of a parallel program could be computed by summing up the maximum time spent between synchronising processes on this sequence [35] called the *critical path*. This insight is comparable to the combined time of BSP supersteps in (2.3.5). The time to solution of a parallel program can then be expressed as in (2.4.1) where the components sum the computation, communication, contention and synchronisation on the critical path of the program execution.

$$T_{total} = (T_{computation} + T_{communication} - T_{overlap}) + \\ T_{resource\_contention} + T_{synchronisation} \tag{2.4.1}$$

Any part of computation and/or communication that are performed simultaneously need to be subtracted ($T_{overlap}$). The computation and communication times are assumed to be deterministic, while estimating resource contention and synchronisation delays require sub-models depending on the significance of each portion to the total time to solution. For example, complex sub-models for contention have been developed in [72, 73] and simplified synchronisation costs have been used in [8]; simplified contention models are used in [26] as well as the contention model described in Section 4.6 in this thesis.

One disadvantage of this approach is that for the cases where non-deterministic computation times and communication times are dominant in an application the model given in (2.4.1) becomes inaccurate. One example of such an application is a code that has significant data dependencies in selecting the execution path (e.g. conditional branches, dynamic loop bounds). Such applications are best analysed through pure stochastic models. Nonetheless, the many successes of deterministic models [17, 141, 8, 36, 38, 37, 14] have proved the validity of the deterministic assumption in modelling parallel programs.

The analytic models developed in this thesis are based on developing reusable expressions for applicable components of (2.4.1). We leverage the consistent deterministic behaviour of wavefront codes to use this simplifying model as opposed to developing the model from stochastic principles.

2.4.5 Hybrid and Other Methods

There are several notable projects that provide tools and techniques that use a combination of the four performance engineering methodologies. Examples include POEMS [31] and Prophesy [142]. There are also notable tools that attempt to provide automation in developing models. Examples of such tools include the assertion based framework [143] and several tools provided in the PACE [12] and WarPP [43]simulation toolkits.

The POEMS system could be considered as a collection of techniques that include, several simulators; processor and memory hierarchy models based on SimpleScalar[144], interconnection network models using the PARSEC parallel simulation language[145], large-scale parallel program simulations using the MPI-Sim[29] simulator, analytic models based on LogP, LogGP and LoPC, and a repository of performance data gathered during previous evaluation, modelling and measurement processes. These are used as historical data to estimate the performance of widely used algorithms as functions of system and architectural characteristics and configurations.

Prophesy [142] utilises pure performance modelling as well as components of statistical forecasting. The Prophesy framework consists of a data collection component that inserts automatic instrumentation codes into the application at the level of functions, procedures, loops and branches; central databases that are used to hold the collected performance measures; and a data analysis component that uses the collected data for automated modelling by curve fitting, parametrisation and kernel coupling. Kernel coupling, entails analysing the effect of one kernel on another by finding the ratio of the performance of successive kernels against the performance of executing each kernel independently. The curve fitting method is directly related to statistical forecasting where the empirical data on the database is utilised. Parametrisation requires manual analysis of code with relation to system performance measurements.

The Modelling Assertions (MA) framework [143] provides an API to annotate source code (Fortran and C) that uses MPI. During application execution, these annotations capture the significant events and produces trace files. The MA framework then post-processes the annotations to develop performance models that resemble a high-level representation (control flow) of the annotated code portion's execution and an intermediate file. The intermediate file serves to develop a "symbolic model" that parametrises the application in terms of problem size, number of processors and other user-defined characteristics. The symbolic models generated by the MA framework are compatible with the Matlab and Octave script format and can be evaluated to get performance predictions.

2.5 Discussion

Figure 2.5 lists a categorisation of the performance engineering methodologies discussed in the previous sections. Although we show a clear demarcation between techniques, in practice a combination of techniques is used during a performance engineering study. Particularly, simulation and analytic modelling require some amount of data gathered through benchmarking or profiling as well as use of statistical analysis techniques to obtain accurate results. Additionally, a single technique should not be used alone for performance prediction without validating it with at least one more technique [39]pp32.

Selecting the appropriate performance engineering technique for the relevant systems/application development stage is important. All the techniques and methodologies discussed above are applicable to stages of the HPC lifecycle, yet some are more suitable than others. Table 2.1. lists a summary of the pros and cons of the above methodologies and their relevance to the HPC lifecycle.

Benchmarking and profiling are simple techniques for application by non-performance engineering experts. Profiling in particular, allows one to zoom in on critical

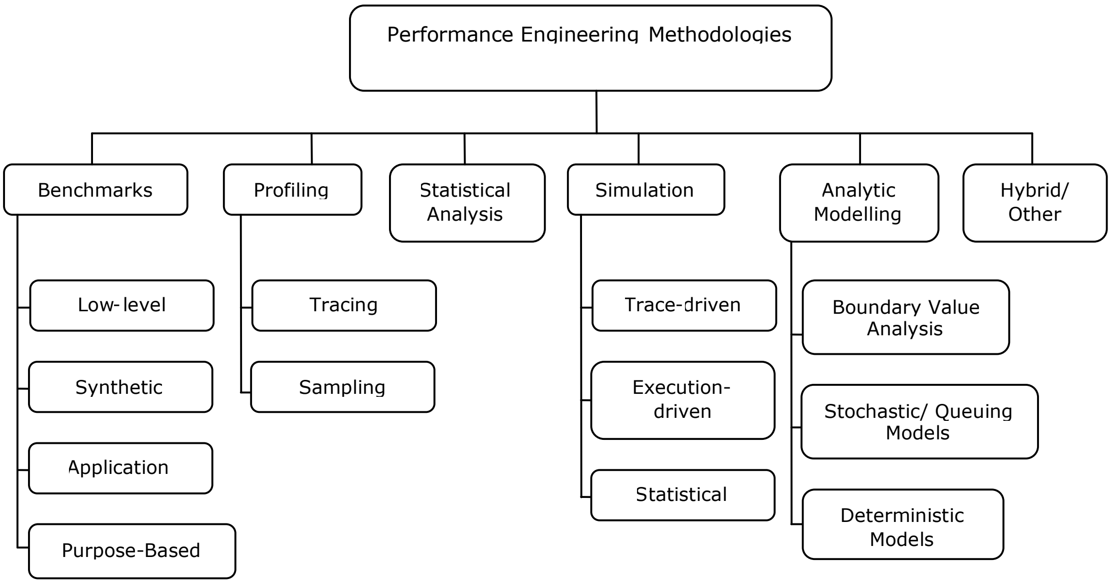


Figure 2.5: Performance Engineering Methodologies

Table 2.1: Pros and cons of performance prediction methodologies

	Benchmarking /Profiling	Statistical Analysis	Simulation	Analytic Modelling
Cost	high	low	low	low
Evaluation Time	high	low	high	low
Effort	low	medium	medium-high	medium-high
Flexibility	low	low	high	high
Portability	depends	high	high	high
Scalability	high	depends	high	high
Accuracy	variable	depends	medium-high	medium-high
Tools	benchmarks/ profilers	mathematics/ statistics	programming/ programmer	analyst
Stage	post prototype	post prototype	any	any

sections of the code and is most useful when conducting a performance engineering study that require an analytic or simulation model of the hotspots of a code. These techniques cost time and money, to run each experiment on a system, particularly an HPC system. The obvious advantage of benchmarking and profiling is that they give actual runtime measures or profiles of the target applications running on the target systems. Thus the values obtained are not speculative compared to other performance engineering methodologies. This in turn can give more certainty when making decisions regarding systems (e.g. for procurement) and applications (e.g. for revising applications). However, many factors such as spurious behaviour of the system, experimental errors, profiler overheads, can distort the conclusions obtained. Also the parameters exposed during a measurement exercise may not represent the real world variable range (e.g. due to inexperience of the benchmarker), making the results accuracy range from very high to almost none [39]. Benchmarking and profiling may not provide accurate information for extrapolating future system performance (i.e. limited flexibility for answering "what if" scenarios), but can provide invaluable or sometimes essential information when developing simulation or analytic models. Furthermore this technique can only be carried out at the post prototype stages in the HPC lifecycle when there is a system and application to measure.

Statistical forecasting depends on the assumption that the obtained historical data contains the required information for forecasting a prediction and that the statistical equations and methods used can replicate this historical behaviour. Thus the accuracy of these methods depends on how far this assumption holds. For small range speculations (e.g. bandwidth change, small range scalability testing) there is high accuracy. For larger speculations (e.g. application optimisations, hardware protocol changes) there may be high errors. The advantage of statistical forecasting is its ability to be used in highly complex systems such as widely heterogeneous systems with dynamic configurations and contention, where other methods of performance prediction are unrealistic. Additionally it can be carried out by non-performance engineering professionals using very modest computational requirements.

Simulation can be done at any level of detail or at any stage of the HPC lifecycle. It requires less simplifying assumptions than analytic models but takes a longer time to evaluate. Simulation may allow one to explore the state space of parameters for optimal performance configuration with comparable amounts of portability and accuracy, but often may not provide the significant level of insights that an analytic model will provide regarding, for instance, the performance trade-offs, operational elucidation and optimisation possibilities. Both techniques will require a comparable amount of effort for development. Analytic models on the other hand, may require an expert's effort to develop. It is the detailed level of insights that analytic models provide that this dissertation uses to obtain the results and research contributions detailed in the next chapters.

The work described in this chapter is by no means exhaustive. However, we believe, that the techniques presented, provide a sufficient perspective to the state-of-the-art in performance engineering at the time of our research.

3 Pipelined Wavefront Computations

Pipelined wavefront computations are ubiquitous in many HPC workloads. This chapter details the operational behaviour of these codes as a mandatory preliminary to developing a general *reusable* analytic performance model. Analytic performance model development is rooted in an analyst's understanding of the complexities of both the algorithm and the application execution on a target platform. Therefore, in this chapter we detail the work carried out in support of the former while at the same time provide a self contained primer of the operation of wavefront computations to the reader.

3.1 Pipelined Wavefront Sweeps

The parallel pipelined wavefront algorithm was originally described by Lamport in [16] as a parallel algorithm to optimise the performance of Fortran DO loops. It has since been used in areas such as computational fluid dynamics (CFD) [14, 146], particle physics [147], parallel iterative solvers [148] and parallel triangular linear equation systems solvers [149]. For simplicity, we begin our discussion of wavefront computations, based on a 2D grid of data points (or data cells[1]). The expansion to a 3D grid of data points is analysed subsequently.

3.1.1 Wavefront Sweeps on 2D Data Grids

Listing 3.1: A simple sequential loop operating on a 2D data array

```
FOR $i = 1, m$ DO
    FOR $j = 1, n$ DO
        $A(i, j) = A(i - 1, j) + A(i, j - 1)$
    END FOR
END FOR
```

Consider the loop in listing 3.1. It details a simple loop for a 2D array of data points of size $m \times n$, where each point is indexed by $(i, j) : i \in [1, m]$ and $j \in [1, n]$. The loop needs to be parallelised so that it produces the same results as its sequential execution. Examining the data dependencies of this loop, we see that any data point (i, j) requires both $(i - 1, j)$ and $(i, j - 1)$ to be computed before, its own computation. Thus any parallelisation must adhere to this dependence, if we are to arrive at the same numerical solution. The pipelined wavefront algorithm was introduced by Lamport in [16] as a solution to parallelise such loops. The parallelisation basically involves decomposing the 2D grid of data such that a column of

[1] note that we use grid points, data points and data cells interchangably in this thesis.

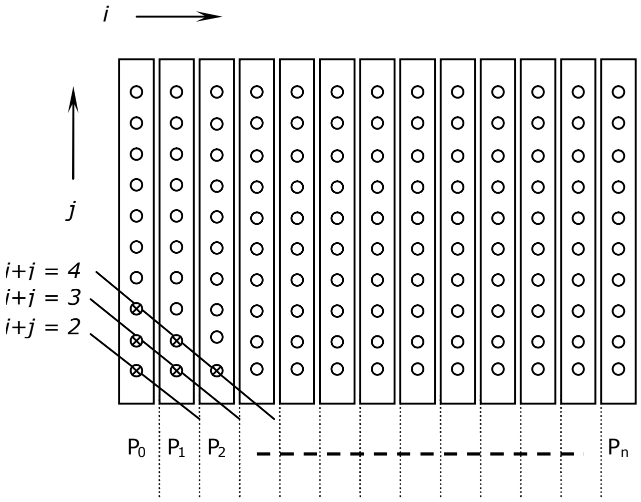


Figure 3.1: A 2D pipelined wavefront operation on a 1D processor array

elements in the n direction (or alternatively m) is assigned to a separate processor as in Figure 3.1. Then the loop body is concurrently executed for all points lying on the diagonal *line* defined by $i + j = CONST$. After each concurrent iteration, $CONST$ is incremented, until all the grid points are solved. We denote this parallelised loop in listing 3.2, where the sequential loop is re-written with modified index variables $g = i + j$ and $h = i$. Thus the parallelisation basically performs the same computation as listing 3.1 but in a different order.

Listing 3.2: Parallelised loop for a 2D data array using pipelined wavefronts

```
DO CONCURRENTLY ON EACH PROCESSOR
FOR $g = 2, n + m$ DO
        $A(h, g - h) = A(h - 1, g - h) + A(h, g - h - 1)$
END FOR
```

The first three iterations of the loop in listing 3.2 are shown in Figure 3.1. The grid point computed at each step is marked by a $\otimes$, while grid points yet to be processed are denoted by a $\circ$. Then, the motion of the computation during the execution of this concurrent loop can be viewed as a *wavefront* that *sweeps* from one corner of the 2D grid to the opposite corner. The processors that split the i direction can be viewed as stages of a pipeline, where each element belonging to a processor's column of grid points are processed at each step of the pipeline.

By carefully examining the parallelised loop in listing 3.2 and Figure 3.1 we see that for each iteration, the processors will perform the loop body computation without violating the data dependency rules we observed in the sequential loop. That is, to compute $A(i, j)$ a processor requires $A(i - 1, j)$ and $A(i, j - 1)$. However, only $A(i - 1, j)$ will be obtained by an off processor communication from the neighbouring *upstream* processor. In other words, we see that at each step of the wavefront, a processor will require data from one up-stream neighbour processor.

Listing 3.3: A simple sequential loop operating on a 3D data array

```
FOR $k = 1, l$ DO
    FOR $i = 1, m$ DO
        FOR $j = 1, n$ DO
            $A(i, j, k) = A(i-1, j, k) + A(i, j-1, k) + A(i, j, k-1)$
        END FOR
    END FOR
END FOR
```

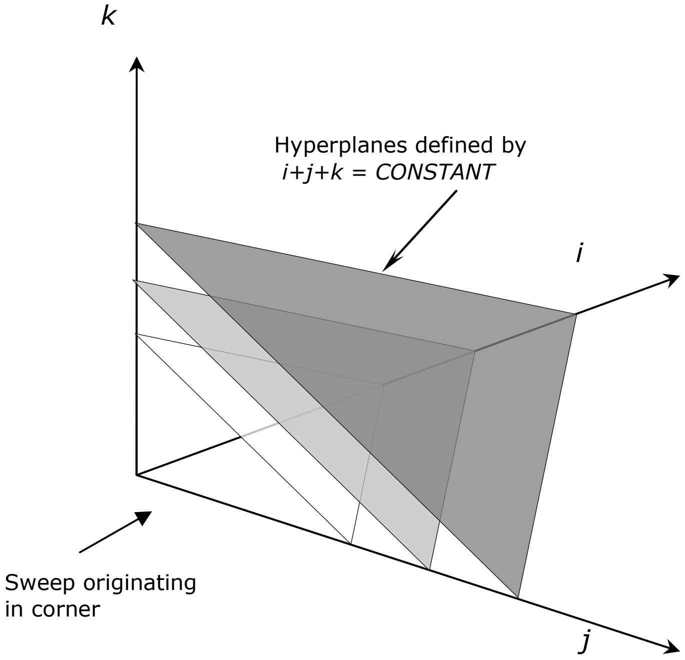


Figure 3.2: Hyperplanes on a 3D grid of data

3.1.2 Wavefront Sweeps on 3D Data Grids

To expand the above parallelisation to a 3D grid of data of size $m \times n \times l$, we consider parallelising the loop in listing 3.3. Now there are up to three near neighbour data dependencies to compute a grid point. To apply the wavefront parallelisation, the 3D data grid is first decomposed on to a 2D array of processors, where the column of cells defined by the same (i, j) coordinates are assigned to a single processor. Thus, the cells in the k direction are held within a single processor. The parallelisation then involves executing concurrently the points on a *plane* defined by $i + j + k = CONST$ (see Figure 3.2). The resulting parallelisation is given in listing 3.4. Similar to the 2D case, the new loop index variables are obtained by setting $f = i + j + k$, $g = k$ and $h = j$. In [16], the plane defined by $i + j + k = CONST$ is termed a *hyperplane* and the algorithm is thus called the hyperplane method.

The key characteristic of the above algorithm, therefore, is that the grid points along

Listing 3.4: Parallelised loop for a 3D data array using pipelined wavefronts

```
DO CONCURRENTLY ON EACH PROCESSOR
    FOR $f = 3, m + n + l$ DO
            $A(f - g - h, h, g) = A(f - g - h - 1, h, g) + A(f - g - h, h - 1, g)+$
                                  $A(f - g - h, h, g - 1)$
    END FOR
```

the plane defined by $i + j + k = CONST$ are computed concurrently and that this *hyperplane* progresses through the 3D grid as a wavefront. The number of data dependencies, for a given point (i, j) could be up to three near neighbour upstream points given by: $(i - 1, j, k) \rightarrow (i, j, k), (i, j - 1, k) \rightarrow (i, j, k)$ and $(i, j, k - 1) \rightarrow (i, j, k)$. Although in the above explanation we have assumed that each processor will hold a stack of grid points or cells of size $1 \times 1 \times l$, in reality the available number of processors are always far less than $m \times n \times 1$. Thus a data decomposition method for cells needs to be considered when implementing wavefront computations for any realistic parallel machine. Next, we describe such a decomposition and define the operation of a general pipelined wavefront sweep.

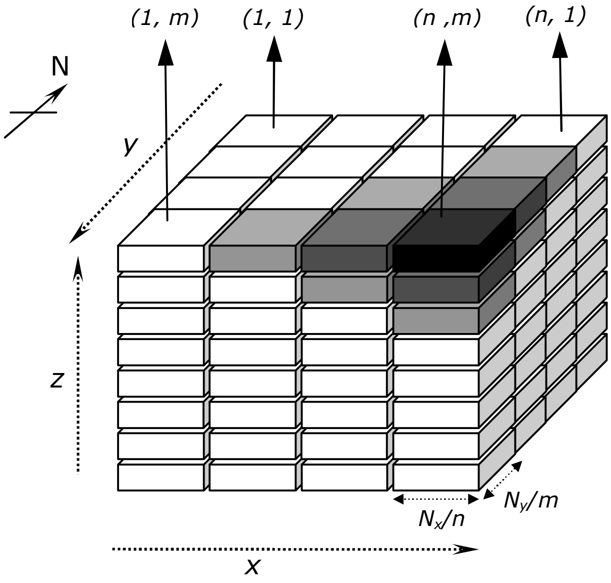


Figure 3.3: 3D data grid mapping on to a 2D processor array

Consider a 3D discretised grid of data cells, with dimensions denoted by x, y and z as in Figure 3.3. The total number of cells is given by $N_x \times N_y \times N_z$. The 3D data grid is partitioned and mapped onto a 2D $m \times n$ array of processors, such that each processor is assigned a stack of data cells of size $N_x/n \times N_y/m \times N_z$ as also depicted in the figure. Note that m and n are the number of processors in the y and x dimensions respectively. A processor is indexed as (i, j), where i is the horizontal position (column number or x dimension) and j is the vertical position (row number or y dimension) respectively. Now, each partition of data cells assigned to a processor can be viewed as a stack of *tiles*, each of 1 cell high.

The data dependency of the cells held in processors results in a sequence of wavefronts

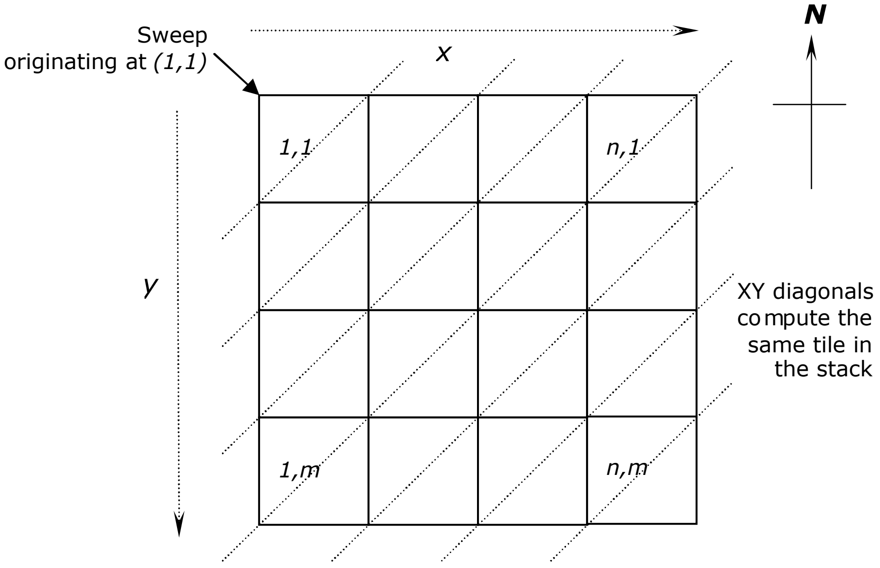


Figure 3.4: Pipelined wavefronts on the 2D processor array

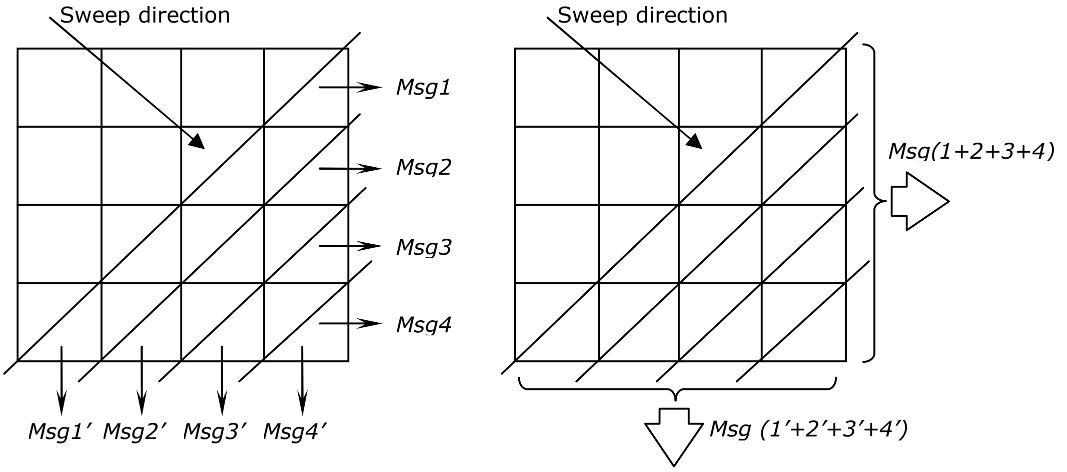
(or a *sweep*) that starts at one of the corner processors, computing over the cells of its topmost or bottommost tile. For example, consider the case where processor $(1, 1)$ in Figure 3.3 begins by computing the results for its bottommost tile. At the end of a calculation by the processor, it sends the respective boundary values (i.e., the new values for the data cells at the edge of the tile) to processors $(1, 2)$ and $(2, 1)$. After receiving those values, processors $(1, 2)$ and $(2, 1)$ each compute values for their bottommost tiles, while processor $(1, 1)$ computes values for the next tile in its stack. Each processor sends boundary values to its east and south neighbours and then computes new values for the next tile in its stack and so forth, until all the tiles have been processed. This creates a series of "wavefronts", as illustrated in Figure 3.4, since the processors along each $x - y$ diagonal are all processing the tile at the same position in their respective stacks. The sequence of wavefronts - or *sweep* - ends when the processor (n, m) at the opposite corner finishes processing it's top-most tile - that is, the tile at the opposite corner of the 3-D grid. The shaded tiles in Figure 3.3 depict the tiles that are processed during the final three wavefronts (or final three sweep steps) - light Gray, then medium Gray, then dark Gray - belonging to a sweep that originated at the bottommost tile on processor $(1, 1)$, ending at the topmost tile on (n, m). The general algorithm implementing one sweep of the wavefront computation that progresses from one corner processor to the opposite corner is given in listing 3.5.

Note that the algorithm in listing 3.5 is performed on each processor concurrently. A processor can only proceed with its computation when it has received upstream boundary values. Until then a processor will be in a blocked state waiting for the RECEIVEs to be satisfied. Thus, communications are done per tile, and *not* for each cell across the processor boundaries. If the true hyperplane algorithm operation had been implemented on the 3D grid of data, then each cell along a boundary of a processor will have had to communicate its value to the corresponding downstream cell in the neighbouring processor (in each direction) using a separate message. This is depicted in Figure 3.5(a) where the cells belonging to a single processor is

Listing 3.5: General pipelined wavefront algorithm

```
FOR EACH TILE DO
          RECEIVE FROM WEST
          RECEIVE FROM NORTH
                    COMPUTE (CELLS IN TILE)
          SEND TO EAST
          SEND TO SOUTH
END FOR
```

illustrated. In this case, each cell generates a message (Msg). The earlier versions of the NAS-LU benchmark had such an implementation which has been shown to be highly inefficient [14] due to the large number of small messages generated. By contrast we assume the agglomerated messaging style depicted in Figure 3.5(b), in the general wavefront algorithm as well as throughout the rest of this work. Now, only a single message is sent to the corresponding downstream processor in each direction, considerably reducing the communication costs.



(a) Messages sent for every cell across the processor boundaries (b) Only a single message is sent at each processor boundary

Figure 3.5: Fine-grained messaging and agglomerated messaging

There are many variations and structural augmentations to the general wavefront algorithm. Examples include multiple - overlapping and/or simultaneous - sweeps, wavefront operating on irregular (unstructured) data grids, variations on the computation done per sweep step, etc. It is the goal of this dissertation to develop a reusable performance model to capture a wide range of these variations, culminating in a model to investigate the performance implications and possible optimisations for each variation when executed on modern HPC systems. In the next section, we investigate three, real world wavefront applications that contain several of these structural differences. Particularly we explore the behaviour of three significant wavefront benchmark codes - NAS Parallel Benchmark suite's LU, Sweep3D from the ASC benchmarks and AWE's Chimaera, as a starting point to understand real world implementations of pipelined wavefront computations on modern HPC systems.

3.2 Pipelined Wavefront Applications

LU from NASA's aerodynamic simulation parallel benchmark (NPB-LU) [21, 92, 146, 14], Sweep3D [23] from the ASC benchmarks [91] and Chimaera from the U.K. AWE are all important scientific benchmark codes. LU represents a compressible Navier-Stokes equation solver used in computational fluid dynamics (CFD), Sweep3D developed by the Los Alamos National Laboratory (LANL) in the U.S. represents particle transport applications that make up 50-80% of the computations that run on their high performance systems [17], while Chimaera is a particle transport benchmark from the Atomic Weapons Establishment (AWE) in the U.K., representative of a major portion of their workload and also used for the procurement of their high performance systems.

Both Sweep3D and Chimaera are particle transport codes, while LU belongs to the solution of a CFD problem. Therefore we investigate briefly the numerical solutions underlying wavefront computations applied to CFDs and particle transport codes with the use of previously published work and explore in detail the variations and structural differences of each code compared with the general wavefront operation introduced in the previous section.

3.2.1 NPB - LU

The NAS parallel benchmark's LU is a simplified compressible Navier-Stokes equation solver [14]. A more complete treatment of the numerical solution of LU is detailed in [146, 150]. In this section we briefly develop the solution in order to identify the application of the hyperplane parallel algorithm and subsequently discuss its computation-communication behaviour.

LU uses the well known Gauss-Seidel relaxation scheme with successive over-relaxation (SSOR) for solving discretised and linearised equations. The solution is obtained through an iterative process. Consider the $(n+1)^{th}$ time step of a dicretised linear system of equations given by:

$$U^{(n+1)} = U^{(n)} + \Delta U^{(n)} \tag{3.2.1}$$

The solution at each time step requires computing $\Delta U^{(n)}$ given by:

$$\mathcal{K}^{(n)} \Delta U^{(n)} = R^{(n)} \tag{3.2.2}$$

where $\Delta U^{(n)}$ and $R^{(n)}$ are vectors of length $N = 5 \times (N_x - 2) \times (N_y - 2) \times (N_z - 2)$ each, $\mathcal{K}$ is a sparse matrix of size $N \times N \times N$ with each of its elements denoted by (i, j, k) being a 5×5 sub matrix. The equation associated with each (i, j, k) point in (3.2.2) can be then represented by:

$$\begin{aligned} \mathcal{A}_{i,j,k}\Delta U_{i,j,k-1} + \mathcal{B}_{i,j,k}\Delta U_{i,j-1,k} + \mathcal{C}_{i,j,k}\Delta U_{i-1,j,k} + \\ \mathcal{D}_{i,j,k}\Delta U_{i,j,k} + \\ \mathcal{E}_{i,j,k}\Delta U_{i+1,j,k} + \mathcal{F}_{i,j,k}\Delta U_{i,j+1,k} + \mathcal{G}_{i,j,k}\Delta U_{i,j,k+1} &= R_{i,j,k} \end{aligned} \tag{3.2.3}$$

where, $i \in [2, N_x - 1], j \in [2, N_y - 1]$ and $k \in [2, N_z - 1]$. The coefficients $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}$ and $\mathcal{G}$ are submatrices of size 5x5. As it can be seen from (3.2.3) there are data dependencies in all three dimensions similar to the dependencies for a grid point in listing 3.3.

The solution to (3.2.1) is computed through a SSOR scheme for faster convergence with the use of an over-relaxation factor $\omega \in (0, 2)$ such that:

$$U^{(n+1)} = U^{(n)} + \left(\frac{1}{\omega(2-\omega)}\right) \Delta U^{(n)} \tag{3.2.4}$$

To perform the SSOR operation (3.2.2) is rearranged such that the calculation is carried out via a solution of a regular sparse, block lower(L) and upper(U) triangular system, giving rise to the name LU.
By setting:

$$\mathcal{K}^{(n)} = (\mathcal{D}^{(n)} + \omega\mathcal{Y}^{(n)} + \omega\mathcal{Z}^{(n)}) \tag{3.2.5}$$

where, $\mathcal{D}^{(n)}$ is the diagonal matrix, $\mathcal{Y}^{(n)}$ is the lower triangular matrix and $\mathcal{Z}^{(n)}$ is the upper triangular matrix of $\mathcal{K}$, (3.2.2) can be rewritten as:

$$\begin{aligned} \mathcal{K}^{(n)}\Delta U^{(n)} &= (\mathcal{D}^{(n)} + \omega\mathcal{Y}^{(n)} + \omega\mathcal{Z}^{(n)})\Delta U^{(n)} \\ &= R^{(n)} \end{aligned} \tag{3.2.6}$$

Rearranging (3.2.6):

$$\begin{aligned} \left[\mathcal{D}^{(n)} + \omega\mathcal{Y}^{(n)} + \omega\mathcal{Z}^{(n)}\right]\Delta U^{(n)} &= \\ \left[\mathcal{D}^{(n)} + \omega\mathcal{Y}^{(n)} + \omega\mathcal{Z}^{(n)} + \omega^2\mathcal{Y}^{(n)}(\mathcal{D}^{(n)})^{-1}\mathcal{Z}^{(n)}\right]\Delta U^{(n)} &= \\ \left[\mathcal{D}^{(n)} + \omega\mathcal{Y}^{(n)}\right]\left[I + \omega(\mathcal{D}^{(n)})^{-1}\mathcal{Z}^{(n)}\right]\Delta U^{(n)} &= R^{(n)} \end{aligned} \tag{3.2.7}$$

Thus the lower triangular solution is given by:

$$\left[\mathcal{D}^{(n)} + \omega\mathcal{Y}^{(n)}\right]\Delta\acute{U} = R^{(n)} \tag{3.2.8}$$

and the upper triangular solution is given by:

$$\left[I + \omega(\mathcal{D}^{(n)})^{-1}\mathcal{Z}^{(n)}\right]\Delta U^{(n)} = \Delta\acute{U} \tag{3.2.9}$$

Therefore in order for a solution, an iteration of LU proceeds by:

1. Computing the right-hand-side vector $R^{(n)}$
2. Computing the lower triangular solution : (3.2.8)
3. Computing the upper triangular solution : (3.2.9)
4. Updating the solution : (3.2.4)

In LU, step (1) is computed using a parallel stencil computation where each data point in the data grid will obtain its four nearest neighbour's values and sum them, simultaneously. The majority of the computation is spent in steps (2) and (3). The equation solved at a grid point (i, j, k) during step (2) can be derived from (3.2.3) to produce the lower triangular system solver:

$$\omega\left[\mathcal{A}_{i,j,k}\Delta U_{i,j,k-1} + \mathcal{B}_{i,j,k}\Delta U_{i,j-1,k}\right] + \mathcal{C}_{i,j,k}\Delta U_{i-1,j,k} + \mathcal{D}_{i,j,k}\Delta U_{i,j,k} = R_{i,j,k} \tag{3.2.10}$$

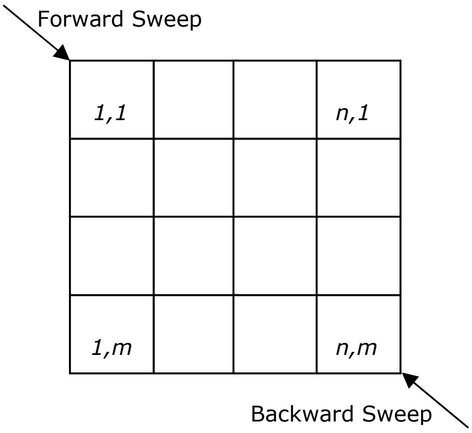


Figure 3.6: LU pipelined wavefront operation on the 2D processor array

This translates to the sequential algorithm given by listing 3.6. This has the same form as 3.3 and thus can be solved using the hyperplane method, resulting in listing 3.7, a wavefront computation.

Listing 3.6: LU sequential algorithm

```
FOR $k = 2, N_z$ DO
  FOR $j = 2, N_y$ DO
    FOR $i = 2, N_x$ DO
      $\Delta U_{i,j,k} = \mathcal{D}^{-1}_{i,j,k} \left[ R_{i,j,k} - \omega \left( \mathcal{A}_{i,j,k} \Delta U_{i,j,k-1} + \mathcal{B}_{i,j,k} \Delta U_{i,j-1,k} \right) + \mathcal{C}_{i,j,k} \Delta U_{i-1,j,k} \right]$
    END FOR
  END FOR
END FOR
```

Listing 3.7: LU parallel algorithm

```
DO CONCURRENTLY ON EACH PROCESSOR
FOR $i + j + k = 6, N_x + N_y + N_z - 3$ DO
      $\Delta U_{i,j,k} = \mathcal{D}^{-1}_{i,j,k} \left[ R_{i,j,k} - \omega \left( \mathcal{A}_{i,j,k} \Delta U_{i,j,k-1} + \mathcal{B}_{i,j,k} \Delta U_{i,j-1,k} \right) + \mathcal{C}_{i,j,k} \Delta U_{i-1,j,k} \right]$
END FOR
```

A similar solution is applied at step (3) for solving the upper triangular solution. Thus an LU iteration includes two wavefront sweeps. The operation of the wavefronts on the 3D grid of data is illustrated in Figure 3.6. The 3D data grid of size $N_x \times N_y \times N_z$ is decomposed as a 2D array of processors by continually halving the domain in the x and y dimensions. Thus LU is only designed to run on a number of processors which is a power of 2. The z dimension is held within a processor and as before, can be viewed as a stack of tiles each having a height of one cell.

The two sweeps in LU propagate in the opposite directions. The forward sweep starts at the bottommost tile on processor $(1, 1)$ and spreads through the 3D data grid ending on the topmost tile of processor (n, m). At the end of this forward sweep, a backward sweep begins at the topmost tile of processor (n, m) and propagates towards the processor $(1, 1)$ ending at its bottommost tile.

Listing 3.8: The pipelined wavefront algorithm in LU

```
FOR EACH TILE DO
        PRE-COMPUTE CELLS IN TILE
        RECEIVE FROM WEST
        RECEIVE FROM NORTH
                COMPUTE (CELLS IN TILE)
        SEND TO EAST
        SEND TO SOUTH
END FOR
```

Compared to listing 3.5, an additional algorithmic difference in LU is that a pre-computation block is performed before a processor posts its receives. This is illustrated in listing 3.8. More specifically, during the forward sweep, for each tile performing the L.H.S. expression of (3.2.8), the LU splitting of the matrix is performed [14] and then relaxed. Similarly during the backward sweep, for each tile performing the L.H.S. expression of (3.2.9), the LU splitting of the matrix is performed [14] and then relaxed. The LU splitting does not require any boundary data exchanges between processors and therefore can be performed before posting receives.

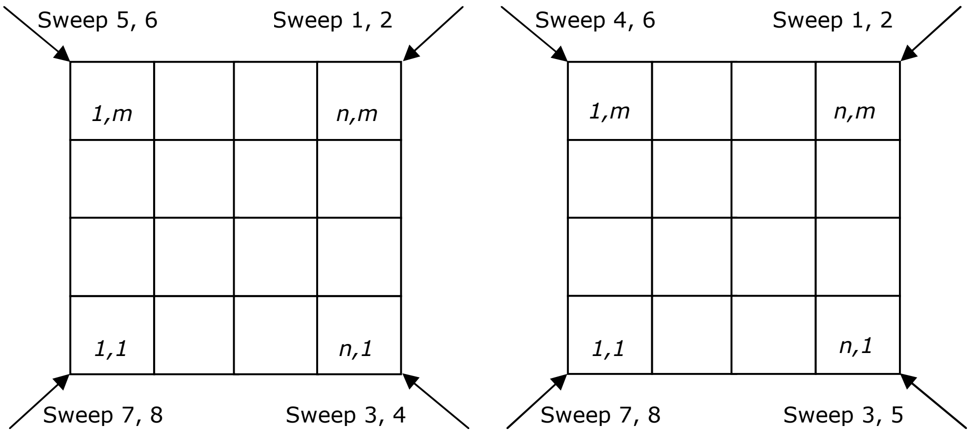
There are various performance implications of this additional pre-computation which will be analysed in Chapter 7. A general re-usable performance model will be required to capture this behaviour and the ability to predict the aggregate performance of multiple sweeps. Additionally we see that non-wavefront portions of the code, such as the computation of the R.H.S .vector $R^{(n)}$ should be modelled separately if we are to obtain a model for the whole application, particularly to predict the total time to solution. We find further structural and behavioural variations in Sweep3D and Chimaera both of which, in contrast to LU, solve particle transport computations using wavefront algorithms.

3.2.2 Sweep3D and Chimaera

Both Sweep3D and Chimaera implement solutions for particle transport simulations. Particle transport codes model the travel of particles such as neutrons and photons through a background medium. Sweep3D and Chimaera represent workloads of production applications that run on HPC systems at the U.S. Dept. of Energy (DOE) and the U.K. Atomic Weapons Establishment (AWE) respectively. The numerical solution on which Sweep3D is based has been well published [147, 151] while Chimaera solves a similar problem albeit exhibiting several differences in parallel computational operation. A detailed analysis of the numerical solution involved in particle transport problems is beyond the scope of this work. However a brief look at the problem solved by Sweep3D gives us a view of the problem parameters involved. Particle transport codes are based on the solution to the time-independent, multigroup Boltzmann transport equation given by:

$$\Omega.\nabla\psi(r,E,\Omega)+\sigma(r,E)\psi(r,E,\Omega)=\int_{\infty}^{0}\sigma(r,E'\rightarrow E)\int_{S^2}\psi(r,E',\Omega')d\Omega'dE'+q(r,E,\Omega) \tag{3.2.11}$$

The quantity to be found is the flux ψ of a particle at a spatial point r, moving in the direction of $\Omega \in S^2$ with energy $E \in (0,\infty)$. To solve the problem these variables are discretised. In particular, (1) the energy E of a particle is restricted to a set of finite subintervals between some maximum and minimum interval giving a set of *energy groups*, (2) the angular-direction Ω is discretised to a set of angles and (3) the spatial domain D, where $r \in D$, is assumed to be a 3D rectangular space and is partitioned into a cartesian grid of cells. Thus the flux ψ needs to be solved per energy group, per angle and per cell. The angles can be solved independently as there are no data dependencies between angles. However, there is a dependence between energy groups making the solution over energy groups sequential. That is, the 3D cube of cells needs to be solved for one energy group completely before solving for the next.



(a) Sweep3D

(b) Chimaera

Figure 3.7: Sweep3D and Chimaera pipelined wavefront operation on the 2D processor array

The solution for the flux of cell (i,j,k) for an angle per energy group requires the cell face values of three upstream near neighbour cells. The solution of cell (i,j,k) will then give the flux at its cell centre and the outward flux of the three remaining faces. Thus, the computation can only start from a corner of the 3D cube of data given boundary conditions (vacuum or reflective). It is clear that, due to this data dependence, the computation needs to proceed as a wavefront sweep from one corner to the opposite corner. As there are eight corners (or Octants) in the cube, the solution involves sweeps starting from all eight corners.

Similar to LU, Sweep3D and Chimaera use a 2D domain decomposition of the 3D data grid resulting in processors holding stacks of tiles of size $N_x \times N_y \times 1$ (denoted by it, jt, kt in the application). The eight corners at which sweeps originate in 3D can be viewed as being reduced to four corners on the 2D processor array. Thus two sweeps originate from each corner processor. Figure 3.7 details the order in which sweeps are performed for Sweep3D and Chimaera respectively.

Further differences to the general wavefront operation can be observed in Sweep3D. Firstly an input parameter (mk) called a *k-block* allows the user to set the number of tiles solved per sweep step. We can view this as increasing the thickness of a tile from 1 to k-block size ($1 \leqslant k\_blocksize \leqslant N_z$). Similarly, another input parameter (mmi) called an $angle\_block$, ($1 \leqslant angle\_blocksize \leqslant \Omega$) allows setting the number of angles solved per sweep step. For the version of Chimaera investigated in this research both the k-block and the angle-block were fixed.

Finally, due to boundary conditions, the order in which sweeps are performed allow for some sweeps to be overlapped. In Sweep3D, processor (n, m) begins the first tile of the second sweep immediately after it finishes the last tile of the first sweep, and the third sweep begins when processor $(n, 1)$ has completed its stack of tiles in the second sweep. In Sweep3D, the fourth sweep begins as soon as processor $(n, 1)$ has finished its tiles in the third sweep, but in Chimaera the fourth sweep does not begin until processor $(1, m)$ at the opposite corner finishes the third sweep.

Listing 3.9: The pipelined wavefront algorithm in Sweep3D

```
FOR EACH ENERGY GROUP ∈ E DO
    FOR EACH OCTANT DO
        FOR EACH ANGLE–BLOCK DO
            FOR EACH K–BLOCK DO
                RECEIVE FROM WEST
                RECEIVE FROM NORTH
                     COMPUTE (CELLS FOR EVERY ANGLE
                     IN ANGLE BLOCK AND IN K–BLOCK)
                SEND TO EAST
                SEND TO SOUTH
            END FOR
        END FOR
     END FOR
END FOR
```

Listing 3.10: The pipelined wavefront algorithm in Chimaera

```
FOR EACH ENERGY GROUP ∈ E DO
    FOR EACH OCTANT DO
        FOR EACH TILE DO
                RECEIVE FROM WEST
                RECEIVE FROM NORTH
                     COMPUTE (CELLS FOR EVERY ANGLE IN TILE)
                SEND TO EAST
                SEND TO SOUTH
        END FOR
     END FOR
END FOR
```

Listings 3.9 and 3.10 illustrate the basic algorithm solved in Sweep3D and Chimaera separately. The message send/receive directions denoted are for a sweep originating from the top left corner processor (e.g. sweeps 5 or 6 for Sweep3D or sweeps 4 or 6 for Chimaera). Note that the octant ordering and overlappings are not differentiated. Both 3.9 and 3.10 represent what we call the sweep or wavefront portion of one iteration of these particle transport benchmarks. Over 95% of the total runtime for Sweep3D and Chimaera is spent on these wavefront operations. Sweep3D only solves 1 energy group while Chimaera solves for 16 energy groups. Furthermore, Sweep3D performs only 12 iterations (in its default setting), while Chimaera iterates until a convergence criterion is satisfied. For the version of Chimaera we investigated it takes between 100-500 iterations for convergence. On top of a multi-group solution of a single time step of Sweep3D and Chimaera, realistic particle transport codes include time-dependence with 1000s of time steps [23]. This further increases the total number of iterations.

As it can be seen, LU, Sweep3D and Chimaera have considerable differences in performing pipelined wavefront sweeps. Further differences in wavefront codes include (but are not limited to) (1) multiple simultaneous sweeps [148], (2) pipelined wavefront computations on irregular grids of data such as particle transport codes on unstructured meshes [37], and (3) variations on data distributions [148]. Add to this the myriad array of software and machine configurations tunable when running these codes on HPC systems, we have a complex set of variables/behaviours that determine the performance of pipelined wavefront computations. In the next section we detail significant previous work that explored this performance engineering problem setting the stage for our contributions in this dissertation.

3.3 Related Work

One of the earliest performance analysis studies on wavefront computations was a paper by Qin *et al.* [149]. It details two types of data distributions on 2D arrays of data and provides analytic expressions for the runtime of the wavefront algorithm on these decompositions. The two partitionings are (1) a column-wise domain decomposition where one dimension of the 2D data grid is partitioned across a number of processors (similar to the decomposition detailed in Section 3.1 but with processors holding multiple columns) and (2) a partitioning that aims to reduce the idle processor time at the beginning and end of a sweep. The expressions are parametrised in terms of problem size, machine parameters such as number of processors, and communication and computation performance. The performance model is then used to compute the optimal partition sizes and the results are validated on a transputer system. Results show that the column partitioning gives the fastest runtime.

Joubert *et al.* in [148] provide a performance analysis on 3D wavefront computations in the form of algorithms used in parallel iterative solvers. It includes a detailed discussion on a cost analysis of this algorithm, parametrised based on problem size, problem parameters and machine parameters such as computation, and communication performance. It also notes several optimisations for the code including the idea of the use of an optimal k-blocking, performing simultaneous sweeps and a modified data distribution. But neither the cost model nor the optimisations discussed are qualitatively or quantitatively validated on a real-world HPC platform, nor are they validated using some other performance engineering method (such as simulation). Thus the accuracy of the models are unknown. Nevertheless, the analysis presented is generic and not specific to any concrete application so as to be applicable in general to parallel iterative solvers on 3D grids of data.

Both Qin [149] and Joubert's [148] give an algorithmic analysis with only the former providing quantitative validations, albeit on an older parallel system with limited parallelism.

Performance engineering a concrete application model for a larger HPC system was detailed in [14] by Yarrow *et al.* In this work, the authors develop an analytic model for the time to solution as a sum of computation and communication costs for two versions of LU: version 2.0 and 2.3.

LU version 2.0 performs a fine grained wavefront operation. That is, the wavefront progresses by processing cells in a diagonal order and, at the points where the wavefront has to cross over to the next processor, a message has to be communicated between processors (see Figure 3.5(a)). Thus for each cell at a processor boundary, a message will be communicated resulting in a large number of small messages during a wavefront sweep. [14] uses a network latency and bandwidth model to account for the communication performance and characterises the computation cost per processor through its floating point operation speed (FLOPS/Sec). In LU version 2.3, messages per cell at the inter-processor boundary are agglomerated similar to Figure 3.5(b). That is, instead of computing a tile in a diagonal order within a processor, they are computed in a standard column-row order (i.e. canonical order). Therefore a communication occurs only after all the cells in the current tile have been computed. Thus messages sent in version 2.3 are significantly smaller and larger than in version 2.0.

The analytic models for the two versions of LU show that version 2.3 performs significantly better due to the considerably smaller number of messages sent. This in turn had reduced the cost incurred for each message startup. The models are validated on up to 128 processors on an IBM SP System with predictive errors of up to 30%. Furthermore the analytic model is customised to only LU and is not reusable to provide performance predictions on other wavefront codes. Nevertheless, one of the significant results of this work is a demonstration of the affect of network latency on wavefront code performance. In this dissertation we provide a more precise quantitative and qualitative analysis of the influence of network latency in addition to analysing other bottlenecks due to computation performance, network bandwidth and idle processors.

Sundaram-Stukel and Vernon [8] develop a LogGP model for Sweep3D on an IBM SP/2. The model elucidates the operation of wavefronts in Sweep3D by capturing computations, communications and synchronisation delays incurred when wavefronts are executed on the target system. Validations show predicted total execution time has less than 10% error for up to 128 processors and all problem sizes reported. As part of the validation, this work also develops LogGP MPI communication models and measured LogGP parameter values for the IBM SP/2 system. Finally [8] shows performance projections for two problem sizes of interest for the ASCI program [152], the 1-billion cell problem and the 20 million cell problem. The projected performance is for up to 27,000 processors which at that time was considered to be the expected size of near future HPC machines. Even with such a large number of processors, problem configurations of interest to ASCI goals i.e. the 1-billion cells problem and 20 million cells problem with 30 energy groups and 10000 time steps - was shown to be impractical with the current algorithm. The synchronization costs on the SP/2 were shown to be the principal factor limiting scalability of the application. The most significant limitation of [8] is that it is customised for Sweep3D and is not readily restructurable for other wavefront codes.

Several papers [17, 25, 153, 37] from the performance architecture laboratory at the Los Alamos National Laboratory (LANL) in the U.S. detail analytic models for Sweep3D. Hoisie *et al.* in [17] develops a model for one sweep of Sweep3D by developing analytic expressions for computation and communication costs. It gives validations of Sweep3D on up to 500 processors on three different HPC systems - an IBM RS/6000, a SGI Origin 2000 and a Cray T3E with low predictive errors (below 10%). This work also provides projections of execution time to solve the ASCI problem sizes of interest.

An analysis of the single sweep model when operating on a system made out of a cluster of SMP nodes is detailed in [26]. This explores the contention arising in wavefront computations at SMP boundaries where processors in a node share router links to communicate with processors in other nodes, and develops a criterion for the minimum number of router links per SMP node to get no contention. Although the final Sweep3D model is not detailed, validations of the model on an SGI Origin 2000 system are presented.

Related work by Kerbyson *et al.* [25] uses the single sweep model in [17] to give a model for the total runtime of Sweep3D. [25] uses the model to speculate the runtime of Sweep3D (without any validations) on a vector processor based HPC system (The Earth Simulator [154, 155]) and an Alpha Server System. It concludes that Sweep3D performs significantly better on the EarthSimulator given a higher achieved computation performance per processor. This Sweep3D model attempts to incorporate the effects of contention when executing on a cluster of SMPs. A concern to be noted is that this contention model inaccurately predicts communication time resulting in a reduction to zero, when the number of links per SMP node increases.

In [153] Hoisie *et al.* provide further validations of the Sweep3D model on three leading HPC systems [25], Blue Gene/L (IBM), Red Storm (similar to the Cray XT3) and ASC Purple (IBM Power 5). A comparison of these three systems is presented with model predictions having less than 25% error. However, modifications to predict the contention effects on CMPs in the RedStorm system are not detailed. RedStorm consists of CMP nodes similar to the Cray XT4 system used in our validations. We detail a more precise contention model for the Cray XT4 in Section 4.6 of this dissertation. Finally an analytic model for a wavefront application operating on an irregular grid of data is developed in [37]. The model is validated with typical errors of approximately 10% on a 64 node Alpha Server system and a 32 node Intel Itanium-2 cluster.

A similar model to that of the Los Alamos models is developed in [24] by Mathis *et al.* using customised equations for computation and communication delays in wavefront execution time. The model makes an original contribution by analysing the runtime for one sweep on three different domain decompositions: KBA [147], Volumetric and Hybrid. Furthermore it explores the case of multiple simultaneous sweeps in all decompositions. This work does not provide validations on any realistic HPC system, but shows analytically the optimal scenarios for each of the three domain decompositions.

The notable simulation-based models for pipelined wavefront codes are the MPI-SIM [29] simulation model for the POEMS [31] project and the simulation models developed using the PACE [27] system and the successor WarPP toolkit [43].

The POEMS system as noted in Section 2.4.5 is a collection of performance engineering tools and methodologies. Parts of [31] detail the use of MPI-SIM [29] and SimpleScalar [144] simulators to analyse Sweep3D. The simulation is done through direct execution driven discrete event simulation, where the computation per sweep step is simulated per instruction by SimpleScaler, and the MPI communication events are handled by calls to MPI-SIM. The results from the simulation study in [31] highly correlate with the predictions for Sweep3D from the analytic models based on LogGP (which are also part of the tools in POEMS that were separately developed in [8]). These include speculative results for prediction on an IBM SP/2 system on large processor counts in the order of $10,000$. At the time of this study a real system of that size was not available for direct validation, but the authors demonstrate validating the simulation and analytic model results against each other.

PACE has similarities to the POEMS simulation work for wavefont applications, in that direct execution of computation and communication events are carried out to obtain predictions. We defer the discussion of related work conducted using PACE [27] for wavefront applications to a more complete discourse in Chapter 6

All the analytic models detailed above sum the critical path computation and communication times to obtain the total execution time on a given parallel architecture. Some have been demonstrated to be highly accurate for a given platform of interest. However, in addition to the various shortcomings of each, these models are customised for predicting performance of a single application, in most cases either for Sweep3D or LU. Therefore, they require significant and unspecified restructuring in order to be applied to any other existing or imaginable pipelined wavefront application. We believe that this significantly limits their applicability as a reusable model, particularly for optimising wavefront codes and for speculative analysis. We encountered such a situation when modelling Chimaera, which had no published performance models or studies, prior to this research. A similar argument can be made regarding the unspecified structuring required in the Sweep3D simulation models [31, 27] if it were to be generalised to predict performance of any wavefront application. In the next chapter we present the development of a reusable, *plug-and-play* analytic model to address this open issue. The extensive utility of this model for speculative analysis, design and optimisation of pipelined wavefront computations are detailed subsequently in Chapter 5 and Chapter 7.

4 A Plug-and-Play Reusable Analytic Model

The algorithmic operation of pipelined wavefront computations are discussed in the previous chapter. In particular, the description of a general wavefront algorithm and its operation on a 2D array of processors, as well as definitions for a pipelined wavefront, a wavefront sweep and operation of a sweep step were discussed. Three real-world applications that use pipelined wavefront computations were investigated, including their significant structural differences and resulting variations to the basic wavefront algorithm. Finally, we surveyed previous performance engineering work related to these applications identifying several key research papers that developed customised performance models for Sweep3D and LU. Despite this previous research, we believe that all previous performance studies on wavefront codes lack the ability to provide a comprehensive understanding of the performance of wavefront algorithms on modern HPC systems. In particular they fail to address possible structural differences and variations, and in turn fail to provide consistent insights for possible optimisations. In this chapter, we present the development of a reusable, *plug-and-play* analytic model, based on the LogGP [22] parametrisation, to address these open issues, forming the first key contribution of this dissertation.

The specific sections of this chapter can be summarised as follows: We begin in section 4.1, by outlining a simple set of parameters to capture the significant structural and behavioural differences between pipelined wavefront codes. Then, section 4.2 develops the basic reusable analytic model equations for pipelined wavefront computations on regular 3D orthogonal grids of data. The basic reusable model assumes for simplicity that each processor core executing the computation has a dedicated network interface card (NIC) and main memory; i.e. each node is a single non-CMP processor. Next, in Section 4.3 we look at one of the main HPC systems that were used to validate this model - the Oak Ridge National Laboratory (ORNL) Cray XT3/XT4 (Jaguar) system. This section also includes MPI sub-models developed using LogGP that characterise the message passing performance on the Jaguar machine. Next, a detailed analysis of the issues regarding measuring computation performance is explored in Section 4.4, followed by several key extensions to the basic model - (1) that allow the user to apply the model to 2D regular orthogonal grids of data in section 4.5 and (2) nodes with multiple cores (CMPs) in Section 4.6. Finally in Section 4.7, validations of the reusable model applied to NPB-LU, Sweep3D and Chimaera on up to 8000 processors of the ORNL Cray XT3/XT4 are presented.

4.1 Application Parameters

We begin by declaring a basic set of parameters aimed at capturing the functional, behavioural and structural differences of pipelined wavefront codes. Table 4.1 details these parameters. We

Table 4.1: Plug-and-Play Reusable Model Application Parameters

Parameter	LU	Sweep3D	Chimaera
N_x, N_y, N_z	$Inputsize$	$Inputsize$	$Inputsize$
W_g	$measured$	$measured$	$measured$
$W_{g,pre}$	$measured$	0	0
$H_{tile}(cells)$	1	$mk \times mmi/mmo$	1
n_{sweeps}	2	8	8
n_{full}	2	2	4
n_{diag}	0	2	2
$T_{nonwavefront}$	$T_{stencil} + \delta_h$	$2T_{allreduce} + \delta_h$	$T_{allreduce} + \delta_h$
$MessageSize_{EW}$ (Bytes)	$40N_y/m$	$8H_{tile} \times \#angles$ $\times N_y/m$	$8H_{tile} \times \#angles$ $\times N_y/m$
$MessageSize_{NS}$ (Bytes)	$40N_x/n$	$8H_{tile} \times \#angles$ $\times N_x/n$	$8H_{tile} \times \#angles$ $\times N_x/n$

Parameter	Description
N_x, N_y, N_z	Number of grid cells in x, y and z dimension
W_g	A grid cell computation time (main computation block)
$W_{g,pre}$	A grid cell computation time (pre-computation block)
$H_{tile}(cells)$	Height of the tile in the z dimension
n_{sweeps}	Total number of sweeps
n_{full}	Number of sweeps that completes fully (from corner to opposite corner)
n_{diag}	Number of sweeps that completes from corner up to and including the main diagonal
$T_{nonwavefront}$	Time to complete non-wavefront portions
$MessageSize_{EW}$	Message size (East-West or West-East)
$MessageSize_{NS}$	Message size (North-South or South-North)

derive these based on the differences in LU, Sweep3D and Chimaera and by assuming that the wavefront codes operate on a 3D regular orthogonal grid of data cells. The number of cells in each dimension is then given by N_x, N_y and N_z.

Recall from Figure 3.6 and Figure 3.7 that LU, Sweep3D and Chimaera have different number of sweeps as well as a different structure to the sweeps. We specify the number of sweeps using a general parameter n_{sweeps}. LU only has 2 sweeps - forward and backward, while Sweep3D and Chimaera have 8 sweeps each one originating from one of the 8 corners of the 3D data cube.

In the case of LU, sweep 1 must completely finish executing on all processors before sweep 2 can begin, and sweep 2 must also completely finish before the iteration ends. In Sweep3D, sweep 4 must completely finish before sweep 5 begins and sweep 8 must complete before the iteration ends. However, as shown in Figure 3.7(a), sweep 2 in Sweep3D can begin as soon as the corner processor (n, m) finishes its stack of tiles for sweep 1, and sweep 3 can begin as soon as the stack of tiles for sweep 2 has been processed by the main diagonal processor $(n, 1)$. Note that while sweep 3 is starting up in Sweep3D, sweep 2 is finishing its last few wavefronts. Chimaera has some similarities and some differences in how soon each sweep follows the previous sweep, as shown in Figure 3.7(b) and noted in Section 3.2.2. Other wavefront applications may have other structures for their sweeps.

To capture the relevant behaviour of a wide range of possible sweep structures, we define two new parameters namely, n_{full} and n_{diag} as given in Table 4.1. n_{full} specifies the number of sweeps that must fully complete (i.e. from one corner to the opposite corner) before the next sweep begins, while n_{diag} specifies the number of sweeps that must complete up to and including the main diagonal of the 2D processor array. As will be explained in the next section, all other sweeps (e.g., sweep 2 in Sweep3d) only need to complete on the processor where the sweep originates before the next sweep begins. The values of n_{full} and n_{diag} for each application in Table 4.1 are derived from the corresponding sweep structure in Figure 3.6 and Figure 3.7.

For Chimaera, all the tiles on processor (n, m) for that sweep must be processed, before sweep 2 starts. Sweep 3 can only start at $(n, 1)$ after the delay between processor (n, m) finishing its last tile for sweep 2 and processor $(n, 1)$ finishing its last tile for sweep 2. The number of times this delay, which is modelled later, occurs is given by n_{diag}. Similarly, considering Sweep3D, sweep 5 at processor $(1, m)$ can only start after the delay between processor $(n, 1)$ and processor $(1, m)$ finishing their last tiles for sweep 4. In this case the delay can be characterised as occurring from one corner to the opposite corner. Thus, we denote the number of times this delay, which is also modelled later, occurs by n_{full}.

Further differences between the three codes are captured using parameters as follows: First, LU performs a pre-calculation before performing the MPI receives, while Sweep3D and Chimaera do not. We use $W_{g,pre}$ (Table 4.1) to specify the computation per grid point that occurs before the receives, and set this parameter to zero if no computation is performed.

Second, both Sweep3D and Chimaera have a number of angles to be computed for each data cell. In Sweep3D this is defined by parameter mmo set through an input file, while Chimaera has a similar parameter setting. LU does not have any notion of multiple independent angles to be solved per cell - i.e. it has a fixed amount of work to be performed per cell. To increase/decrease the amount of parallel work done, Sweep3D uses an input parameter called mmi that is also set via the input file to the benchmark, defining the number of angles to be computed before sending boundary values to the near neighbours. This angle block is not defined in Chimaera. Common values for mmi in the Sweep3D benchmark are 6 and 3. In the model developed in this research, we use mmi and mmo to compute an effective value of the height of the tile, as described in the next section.

Third, Sweep3D has a parameter (mk) that defines the height of a tile (in terms of the number of grid cells) also called a k-block. We define a new parameter H_{tile} in our model inputs in Table 4.1. LU and Chimaera each have a fixed tile height equal to one cell. Sweep3D computes mmi of the angles in the tile before sending the boundary values, and then computes another mmi of the angles. In terms of total code execution time, this is the same as computing all of the angles for a tile of height $H_{tile} = mk \times mmi/mmo$, as shown in the table. Note that this implies that W_g is the measured total computation time for all angles in a cell or in the case of LU the total computation time for the fixed amount of work per cell.

Parameter $T_{nonwavefront}$ is the execution time for the operations performed between iterations. For instance LU performs a four-point stencil computation after the 2 sweeps in each iteration, while Sweep3D performs two all-reduce operations. The model of stencil execution time ($T_{stencil}$) is detailed later in Section 4.7 of this chapter. Similarly the time for the MPI collective operation Allreduce is modelled later. The wavefront portions of these codes take over 95% of their parallel runtime. The remaining time is spent in collectives such as the all-reduce operations or in the case of LU, in the four-point stencil computation. We denote the other negligible times by δ_h for completeness.

The size of messages exchanged depends on the underlying data structures exchanged during the wavefront operation. LU communicates five values (each of 8 bytes) for every cell at a processor boundary, while both Sweep3D and Chimaera message sizes depends on the total number of angles solved per cell and the thickness of a tile (H_{tile}). The subscripts EW denotes message size in the horizontal or East-West direction and NS denotes message size in the vertical or North-South direction with regard to the directional conventions set in Figure 3.3.

Note that a wide range of different wavefront application behaviours are captured in the simple and small set of application parameters in Table 4.1. In particular, the parameters can be used to specify various amounts of work before and after the boundary values are received, a range of tile height, an arbitrary number of sweeps per iteration, a wide range of sweep precedence structures including those in the three benchmarks, and a general processing time between iterations. Hence these application parameters support the evaluation of LU, Sweep3D, Chimaera, other possible wavefront applications, and many if not all possible application code design changes.

The parameters are more complete than previous parameters for Sweep3D or LU because they include both the sweep structure and the computations that are performed at the end of each iteration. As discussed in Section 4.7, the reusable model accurately computes execution times for each wavefront code from these application parameters. Hence, the parameter values provide a succinct summary of the key differences among wavefront codes with respect to measured application performance. Furthermore, as we will show in the development of the model, and its application to LU, Sweep3D and Chimaera (and their various extensions) the above parameter set is truly general and extremely re-usable.

4.2 Reusable Model : Single Core

The typical approach to developing a predictive model for a new wavefront code is to modify an existing model to reflect the different behaviours in the new code. The model modification process is however error-prone, and thus each new model must be extensively validated. To reduce such development and validation costs, we choose to build in the impact of the various possible behaviours to a model, relying on the input parameters developed in the previous section to specify the appropriate features for each application. The idea is analogous to reusable software, which is popular due to reduced software development and testing costs. To this end we develop the basic reusable model for pipelined wavefront applications building on a previous customised model for Sweep3D from [8].

We note that in the basic model, it is assumed that each MPI process is mapped on to a single node with a single processor. More specifically the processor is assumed to be a single non-CMP processor with a dedicated NIC and dedicated main memory. In this case all MPI communications are off node. Also note that the basic model is for wavefront applications operating on regular orthogonal grids of data. As such the model takes into account the symmetry in which sweeps are performed. In this case, the execution time of a sweep starting at a corner processor and progressing to the opposite corner processor is the same regardless of the corner at which the sweep originates. Furthermore the assumption of a regular grid implies that the computation requirement of each grid cell is homogeneous, or has very little variance. We discuss computation variance in more detail in Section 4.4.

Equations (4.2.1) to (4.2.5) provide the accurate LogGP model for Sweep3D from [8], and serve as a starting point as well as a useful comparison and contrast for our plug-and-play reusable model for a wide variety of wavefront applications. The specific terms in each equation are described in detail in [8]. Here we simply focus on the overall structure of the model and note that the terms in each equation reflect the sequencing of the operations in the code. (4.2.1) models the time to compute each set of mmi (out of mmo) angles for a tile. The parameters it and jt define the x and y dimensions of the tile. Furthermore, W_g in this previous model is the computation time for one angle of one data cell.

$$W_{i,j} = W_g \times mmi \times mk \times jt \times it \tag{4.2.1}$$

As noted in Table 4.1 we model this differently by defining W_g as the total execution time of one data cell. In the case of Sweep3D or Chimaera this accounts for the time to compute all the angles in a cell. mk defines the number of cells in the z dimension completed per sweep step. The total execution time for a sweep is the same regardless of which corner it originates from. The model computes execution times for a sweep that starts from the upper left corner in the processor grid using the processor indexing in Figure 4.1, and then applies portions of the sweep time to the appropriate actual sweeps in the code. (4.2.2) defines the time at which the sweep starts on any given (i, j) processor in the grid.

$$\begin{aligned} StartP_{i,j} = max(&StartP_{i-1,j} + W_{i-1,j} + Total\_Comm + Receive, \\ &StartP_{i,j-1} + W_{i,j-1} + Send + Total\_Comm) \end{aligned} \tag{4.2.2}$$

(4.2.3) computes the time until the corner processor on the main diagonal completes its stack of tiles in the sweep and (4.2.4) computes the time until the sweep completely finishes on processor (n, m). (4.2.5) sums the total time to execute the 8 sweeps.

$$\begin{aligned} Time_{5,6} = StartP_{1,m} + 2[(W_{1,m} + Send_E + Receive_N + (m-1)L) \times \\ no\_of\_Kblocks \times mmo/mmi] \end{aligned} \tag{4.2.3}$$

$$\begin{gathered} Time_{7,8} = StartP_{n-1,m} + \\ 2[(W_{n-1,m} + Send_E + Receive_W + Receive_N + (m-1)L + (n-2)L) \times \\ no\_of\_Kblocks \times mmo/mmi] + Receive_W + W_{n,m} \end{gathered} \tag{4.2.4}$$

$$T = 2(Time_{5,6} + Time_{7,8}) \tag{4.2.5}$$

The specific terms in each equation are described in detail in [8]. Here we simply focus on the overall structure of the model and note that the terms in each equation reflect the sequencing of the operations in the code. The terms $(m-1)L$ in (4.2.3) and $(m-1)L+(n-2)L$ in (4.2.4) model machine specific synchronisation costs that were observed on the IBM SP/2 on which this model was validated. These were speculated to be due to back-propagation of MPI message passing handshakes on that machine. We omit the synchronisation terms in the development of the re-usable model, noting that these previous or other synchronisation

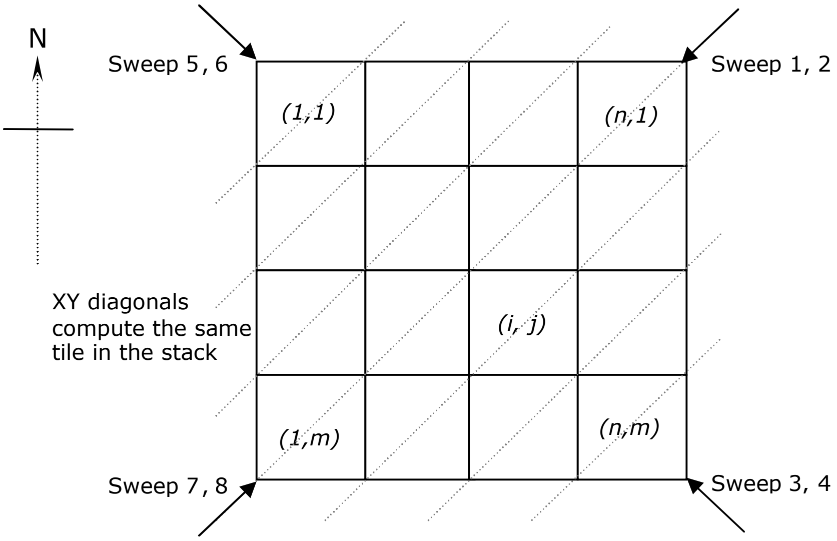


Figure 4.1: Pipelined wavefront operation on a 2D processor array

terms can be incorporated in the re-usable model for other architectures, as needed.

Drawing from the insights of this model, our goal is to obtain a general expression for the critical path time of one iteration of a wavefront application where this expression can be restructured using the application parameter values in Table 4.1. To this end we breakdown the critical path time to components that can be combined with the application parameters. Firstly we develop a general expression for the amount of computation done per sweep step. For Sweep3D this is given by (4.2.1).

As described in Chapter 3, during a sweep step a processor potentially does a pre-computation on its block of cells, waits for boundary values from up to two neighbouring processors, performs the main computation on its own block of cells and then passes boundary values to up to two downstream neighbour processors. We use the model parameters detailed in Table 4.1 to model the computation work done per sweep step before and after boundary values are received by (4.2.6) and (4.2.7).

$$W_{pre} = W_{g,pre} \times H_{tile} \times N_x/n \times N_y/m \tag{4.2.6}$$

$$W = W_g \times H_{tile} \times N_x/n \times N_y/m \tag{4.2.7}$$

$H_{tile} \times N_x/n \times N_y/m$ gives the number of cells to be computed in a sweep step by a processor, while the time to compute a cell as mentioned in section 4.1 is given by $W_{g,pre}$ and W_g for the pre-computation and main computation respectively. Considering the value of H_{tile} for Sweep3D as noted in Table 4.1, equations (4.2.6) and (4.2.7) also illustrate the simplicity of incorporating the mmi parameter into H_{tile}.

Next, we recall that wavefronts progress through the 2D processor array using the diagonals as stages of a pipeline. Therefore we identify that the critical path time consists of a combination of the following components:

- the time for a wavefront starting at one corner to reach the opposite corner processor;

- the time for a wavefront starting at one corner to reach the processors at the main diagonal on the 2D processor array;
- the time taken by a processor to compute all of its tiles for a given sweep;
- the time for any non-wavefront code executions that occur before or after sweeps.

We obtain general reusable analytic expressions for these components as follows:

Recall that in LU, the pre-computation block can be performed by a processor without any boundary values from near neighbours. Therefore at the beginning of a sweep all the processors can perform the pre-computation block simultaneously. As soon as this block is computed, all processors will be in a blocked state for boundary value receives from upstream neighbours. Only one corner processor (depending on which direction the sweep is propagating from) can start its main computation. For instance a sweep originating from processor $(1, 1)$ in Figure 4.1 will take W_{pre} time to begin the main computation for the first tile for that sweep. If we define $StartP_{i,j}$ as the time it takes for a processor (i, j) to begin its main computation for its first tile for a given sweep, then $StartP_{1,1}$ is given by (4.2.8).

$$StartP_{1,1} = W_{pre} \tag{4.2.8}$$

From Figure 4.1 we can see that a sweep progression from a corner can be viewed as wavefronts propagating from diagonal to diagonal of the 2D processor array. The first wavefront of a sweep originating form a corner processor (say processor $(1, 1)$) to arrive at a processor (i, j) has to have completed the main computation for the first tile of all the processors on all the previous diagonals up to the diagonal to which processor (i, j) belongs. We model this in (4.2.9) using a recursive expression similar to (4.2.2) used in [8].

$$\begin{aligned} StartP_{i,j} = max(&StartP_{i-1,j} + W_{i-1,j} + Total\_Comm_E + Receive_N, \\ &StartP_{i,j-1} + W_{i,j-1} + Send_E + Total\_Comm_S) \end{aligned} \tag{4.2.9}$$

The term $Total\_Comm$ accounts for the time to perform a near neighbour node-to-node message communication, while $Send$ and $Receive$ account for the time taken by a processor to release a message to the network and for the time taken by a processor to acquire a message from the network respectively. The subscripts denote the direction of message passing. These costs are dependent on message size as well as the properties of the underlying HPC machine. Thus, to obtain predictions for a pipelined wavefront application running on an actual HPC system, the values for $Total\_Comm, Send$ and $Receive$ should be separately modelled. Later in section 4.3 we develop MPI sub-model for these terms for the ORNL Cray XT3/XT4 which is one of the main validation platforms used in our work. Further models are developed for an InfiniBand network in Chapter 7.

Note in equation (4.2.9), the first term on the right corresponds to the case where the message from the West is the last to arrive at processor (i, j). In this case the message from the North has already arrived, but cannot be received until the West message is completely received. The second term corresponds to the case where a message from the North arrives last. In this case, processor $(i, j-1)$ does a send to its East before it sends to its South processor (i.e. to processor (i, j)). The $Total\_Comm$ in this case is for the end to end communication between processors $(i, j-1)$ and (i, j).

Using (4.2.9) we can model the time for the main computation of the first wavefront that originated on processor $(1, 1)$ to arrive at processors $(1, m)$ and (n, m) as (4.2.10) and (4.2.11) respectively.

$$T_{diagfill} = StartP_{1,m} \tag{4.2.10}$$

$$T_{fullfill} = StartP_{n,m} \tag{4.2.11}$$

The former is the time gap between starting a sweep at $(1, 1)$ and the first wavefront of that sweep reaching up to the main diagonal processors. The latter is the time gap between starting a sweep at $(1, 1)$ and the first wavefront of that sweep reaching the opposite corner of the 2D processor array. W_{pre} does not appear in equations (4.2.10) and (4.2.11) because the parallel pre-computation for the first tile is accounted for in equation (4.2.8).

More importantly, we also note that (4.2.10) is equivalent to the time gap between completing the final tile belonging to a sweep that originated on processor $(1, 1)$ and the final tile of the same sweep on processor $(1, m)$. Given that there are n_{diag} number of such gaps we can model the total time spent in these gaps as $n_{diag}T_{diagfill}$. Similarly (4.2.11) is equivalent to the time gap between completing the final tile of a sweep that originated on processor $(1, 1)$ and the final tile on processor (n, m). Then $n_{full}T_{fullfill}$ gives the total time spent in such gaps. These gaps lie on the critical path of a wavefront application's execution in addition to the time taken by a processor to solve all of the cells in its domain, which we model next.

The time taken by a processor to solve all of the cells assigned to it, that is to process all of its stack of tiles, is equivalent to performing N_z/H_{tile} number of steps each consisting of (1) a pre computation, (2) a main computation and (3) four communication operations: two sends and two receives. We model this as in (4.2.12).

$$T_{stack} = (Receive_W + Receive_N + W + Send_E + Send_S + W_{pre})N_z/H_{tile} - W_{pre} \tag{4.2.12}$$

The processor at the corner, or on the boundaries of the of the 2D processor array will not perform all four communication operations. However, all processors compute their tiles at the same rate due to the blocking nature of the MPI sends and receives that result from the data dependency between processors. Therefore, even if the corner and border processors complete faster due to sending fewer messages, they still will be blocked by waiting for the other inner processors to complete all four communication operations. The per-tile processing time in (4.2.12) includes W_{pre} for each tile. The total number of tiles that need to be computed is given by N_z/H_{tile}. The subtracted W_{pre} is an adjustment for the final tile in the stack.

Now that we have obtained general reusable analytic expressions for the components of the critical path time of a pipelined wavefront computation, we combine them with the application parameters in 4.1 to form an equation that gives the runtime for one iteration of a general pipelined wavefront application:

$$Time\ per\ iteration = n_{diag}T_{diagfill} + n_{full}T_{fullfill} + n_{sweeps}T_{stack} + T_{nonwavefront} \tag{4.2.13}$$

Equation (4.2.13) provides the time for one iteration of a wavefront computation by

(1) combining the appropriate number of terms for sweeps that must complete at the main diagonal or at the opposite corner before the next sweep can begin, and (2) by adding the term for non-wavefront computations that occur at the end of the iteration or possibly between the sweeps. Note that (4.2.13) provides for an infinite variety of sweep sequences, while the model inputs n_{full} and n_{diag} are the key measures of the sweep precedence structure for a given application. In section 4.7 we detail the application of the reusable model to formulate concrete analytic models for LU, Sweep3D and Chimaera, further demonstrating the reasoning behind the use of n_{full} and n_{diag} and the development of the model equations.

Table 4.2: Plug-and-play LogGP Model: One Core Per Node, on 3D Data Grids

$W_{pre} = W_{g,pre} \times H_{tile} \times N_x/n \times N_y/m$	4.2.6
$W = W_g \times H_{tile} \times N_x/n \times N_y/m$	4.2.7
$StartP_{1,1} = W_{pre}$	4.2.8
$StartP_{i,j} = max(StartP_{i-1,j} + W_{i-1,j} + Total\_Comm_E + Receive_N,$ $StartP_{i,j-1} + W_{i,j-1} + Send_E + Total\_Comm_S)$	4.2.9
$T_{diagfill} = StartP_{1,m}$	4.2.10
$T_{fullfill} = StartP_{n,m}$	4.2.11
$T_{stack} = (Receive_W + Receive_N + W + Send_E + Send_S + W_{pre})N_z/H_{tile} - W_{pre}$	4.2.12
$Time\ per\ iteration = n_{diag}T_{diagfill} + n_{full}T_{fullfill} + n_{sweeps}T_{stack} + T_{nonwavefront}$	4.2.13

Expression	Description
m	Number of processors along the y dimension of the 2D processor array
n	Number of processors along the x dimension of the 2D processor array
$StartP_{i,j}$	Time to begin the main computation on processor (i, j)
$Total\_Comm$	End to end communication time
$Send$	Time to release a message to the network
$Receive$	Time to obtain a message from the network
$T_{diagfill}$	Time gap between starting a sweep at a corner processor and the first wavefront of that sweep reaching up to the main diagonal processors
$T_{fullfill}$	time gap between starting a sweep at a corner processor and the first wavefront of that sweep reaching the opposite corner processor
T_{stack}	Time taken by a processor to solve its stack of tiles
$T_{nonwavefront}$	Time taken by non-wavefront portions of the code

The new re-usable model (as summarised in Table 4.2 is significantly more versatile than previous models of specific wavefront codes, yet it has a similarly small number of intuitive equations. As detailed in Section 4.7, the re-usable model is also highly accurate and comparable to the previous customised models of the Sweep3D code [17, 25, 8].

4.3 The Cray XT3/XT4 and MPI Communications Performance

The development of the reusable model in the previous section was not specific to any HPC system. The only key assumption was that the system on which the application was running is an HPC system with nodes containing only a single processor core per node where one

MPI process is mapped to one node. In order to obtain qualitative and quantitative measures for a specific pipelined wavefront application running on a platform of interest, the application dependent parameters and the machine dependent parameter values should be known. These parameters applied to the reusable model will result in a specific analytic model for the given application's execution on the target HPC system. The application parameters for LU, Sweep3D and Chimaera have already been briefly detailed in Table 4.1 with a more detailed derivation of parameters for each application to follow in Section 4.7. In this section we concentrate on the machine dependent parameters for one of the main validation systems used in this research - the Oak Ridge National Laboratory (ORNL) Cray XT3/XT4 (Jaguar). The machine dependent parameters required for the reusable model are the computation performance ($W_g, W_{g,pre}$) and the MPI communication performance ($Total\_Comm, Send$ and $Receive$). We first focus on the communication performance of Jaguar. The issues related to evaluating the compute performance will be discussed in Section 4.4.

The Jaguar system at ORNL, during the time of this research comprised of two partitions - one consisting of a Cray XT3 and another consisting of a Cray XT4[1]. Table 4.3 details the key system specifications of Jaguar as used in this research [156, 157].

Table 4.3: The ORNL Jaguar : System Details

	XT3	XT4
Processor nodes	2.6 GHz dual-core AMD Opteron	
Memory per processor	4 GB	
Number of processor nodes	5,212	6,296
Memory bandwidth	6.4 GB/s	10.6-12.8 GB/s
Interconnect router	Cray SeaStar	Cray SeaStar2
Bi-directional Interconnect Bandwidth	7.6 GB/s (peak), 4 GB/s (sustained)	7.6 GB/s (peak), 6 GB/s (sustained)
Memory speed	400 MHz	667 MHz
Operating System	Catamount micro-kernel	

Each compute node in Jaguar consists of a dual-core AMD Opteron processor with 64KB L1 instruction cache, 64KB L1 data cache and, 1 MB L2 cache per processor. The processor is connected via a HyperTransport [158] link to a Cray SeaStar [157] chip, which is in turn connected to a 3-D torus network. This interconnection between nodes facilitates efficient mapping of MPI processes in wavefront applications and implies near-neighbour send and receive operations. The main differences between the XT3 and the XT4 are the memory bandwidth, memory speed, and sustained interconnect bandwidth.

Each of $Total\_Comm, Send$ and $Receive$ are functions of the application specific message sizes as detailed in Table 4.1 used to communicate near neighbour data between processors. Thus we require models for these terms parametrised by message size. Such sub-models are a key component of any analytic model based on the LogGP parameter set [8, 36, 25].

To our knowledge, Cray XT3/XT4 MPI communication models have not previously been reported in the literature. Hence, we derive these models below. Note that we confirmed the basic operation of the XT3/XT4 MPI implementations with system architects. To

[1] The XT4 at ORNL has since been upgraded to Quad-Core nodes, running Compute Node Linux (CNL) as well as various other upgrades to the execution environment.

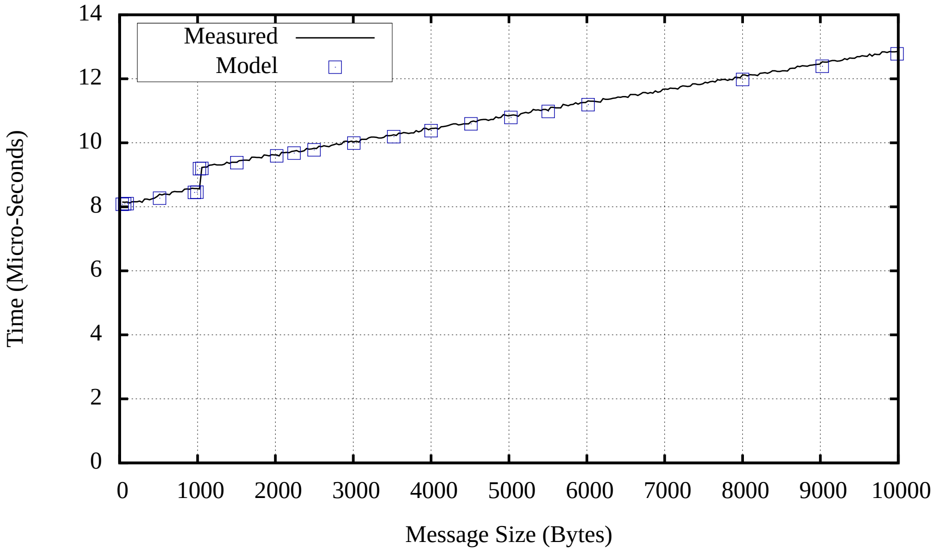


Figure 4.2: Measured and modelled Cray XT4 off-node MPI end-to-end communication times

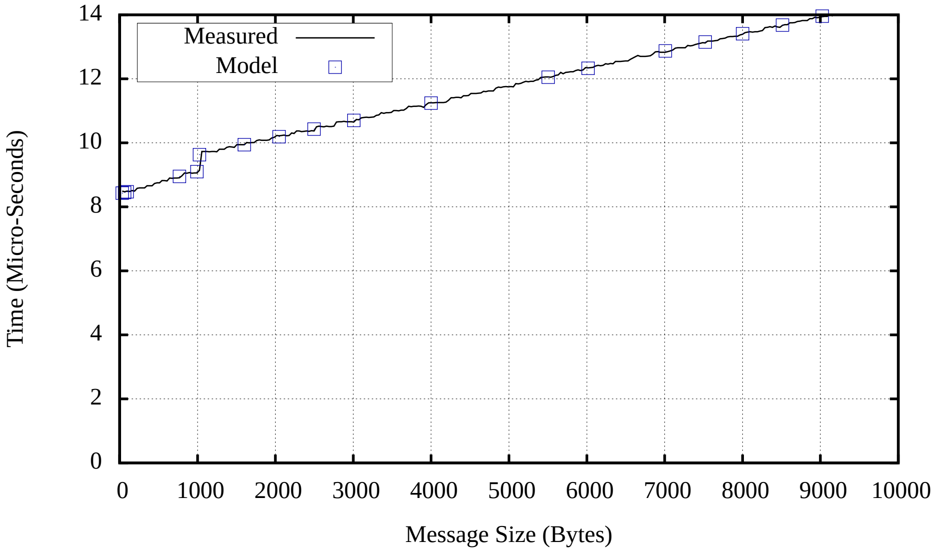


Figure 4.3: Measured and modelled Cray XT3 off-node MPI end-to-end communication times

our knowledge, the models in Section 4.3.2 are also the first validated LogGP models of *on-chip* MPI send/receive for any platform. The LogGP communication models derived below can be used for any application that uses MPI primitives. They also yield insights into the implementation as well as quantitative values of the end-to-end communication latency (L), the processing overhead (o) at the sender and receiver, and the per-byte transmission cost (G). Thus, these models are valuable in their own right. Note that in modern architectures, a node can transmit a new message as soon as a previous message transmission is complete, and thus the gap parameter, g, is equal to zero.

4.3.1 MPI Send/Receive: Off-node

Figure 4.2 plots one half of the round-trip time for a ping-pong message exchange between two nearest neighbour nodes in the XT4, as a function of the size of the message that is transmitted back and forth. These results were obtained using the Intel MPI Benchmark (IMB) version 3.0 [81]. In the IMB Ping-Pong benchmark, each node posts a receive immediately after completing a send, and thus there is very low variance in the measured round-trip times. The solid line connects the measured values, while the points in squares are the highly accurate values predicted by model equations (4.3.1) and (4.3.2) in Table 4.5. The model error was observed to be less than 5%. Similar results obtained through running the IMB on two XT3 nodes are given in Figure 4.3. As it can be seen the XT4 has a minor quantitative performance improvement over the XT3 but has almost no qualitative difference. Thus the qualitative reasoning during model development remains the same for both these systems although we present it as a model for the XT4 communication operations. We observed a similar result for the on-chip MPI performance for the XT3 and XT4. The majority of the subsequent application model validations were also done only for the XT4 system partition[1].

The measured communication time increases linearly with message size up to 1024 bytes and again after 1025 bytes. For all messages larger than 1025 bytes, the sender first sends a short message requesting a reply when the receive has been posted, and waits for a reply before sending the body of the message. This handshake adds a fixed delay, h, equal to the time to send 1025 Bytes minus the time to send 1024 Bytes, in the case that the receive is already posted.

The slopes of the curves before and after the 1024 byte message size are approximately equal. This is the per-byte transmission cost of *'gap per byte'* (G). Note that G is the sum of the per-byte costs for each of the copy operations at the sender and receiver and that $1/G$ yields an inter-node bandwidth of 2.5 GBytes/sec. Also note that the off-node cost per byte is the same for all message sizes. Thus the method used to copy the message data between application and kernel memory is the same for all message sizes, as is the method used to copy the message from kernel memory to the Network Interface Card (NIC). As direct memory access (DMA) methods have higher performance than byte-to-byte copies for large messages, it is likely that DMA methods are used to copy the message data between each of these message buffers. This assumption is not needed in the model derivation below, but is a useful hypothesis. Regardless of the actual method employed, since all message sizes are copied from one buffer to another using the same method, the processing overhead (o) before and after message copies at the sender should be the same for all message sizes. Likewise, the processing overhead at the receiver should be the same for all message sizes.

As in previous LogGP models of communication on various parallel platforms, we assume during model development that the processing overhead on the sender is approximately the same as on the receiver. The intuitive reasons for this assumption are that the sender and receiver perform the same number of message copy operations, and that the DMA setup and other significant processing costs before and after the copy operations can be expected to be about the same at each end. Validations of the model are needed to test the validity of this assumption as well as all other abstractions in the model.

[1] The LU validations were done using the XT3 system as at the time of validation in Jaguar only consisted of an XT3 partition

For messages smaller than 1025 Bytes, (4.3.1) models the total time to send a message, including the message processing time (o) at each end and the end-to-end latency (L) between the two processors.

$$Total\_Comm_{\leq 1KB,offchip} = o + Message\_size \times G + L + o \tag{4.3.1}$$

For message sizes larger than 1024 Bytes, we partition the message processing time at the sender and receiver. Specifically, let h denote the total time for a handshake, obtained as the difference in transmission times for 1025 and 1024 Bytes. Let $o = o_{init} + o_{c2NIC}$ where o_{init} denotes the overhead for the copy between application and kernel, and o_{c2NIC} denotes the time to setup a, DMA or other, copy of the message data between kernel memory and the NIC and to prepare or process the message header. Also let o_h denote the processing time for a handshake request or reply, including the time to prepare a new message header. Using these processing overheads, we model the total time to send a message larger than 1024 bytes using the following sequence of times:

$$Total\_Comm_{>1KB,offchip} = o + h + Message\_size \times G + L + o \tag{4.3.2}$$

where $h = L + o_h + L + o_h$. The detailed overheads enable the model to reflect the order in which the overheads occur and are also needed in the development of the on-chip model in the next section.

For a given message size, (4.3.1) has two unknowns, o and L, whereas (4.3.2) has three unknowns, o, L and o_h. Using the equation for h to solve for the three unknowns leads to an infeasible solution, indicating that the o values in the equation for h are smaller than the overhead o for the message transmission. We surmise that the handshake latency is a NIC-to-NIC latency plus one or more NIC accesses by the processor at each end, whereas the message transmission includes an additional latency between NIC and processor memory where the message data buffers are located.

Assuming o_h is negligible, we use the measured handshake time (h) in (4.3.2). In this case, for a given message size less than one kilobyte and another given message size larger than one kilobyte, we solve (4.3.1) and (4.3.2) simultaneously to derive values of o and L, given in Table 4.4. These values provide the predicted communication time plotted in Figure 4.2, indicating that the model is highly accurate. Note that the off-node parameters in Table 4.4 are one or two orders of magnitude lower than the values for the IBM SP2 in [8], which are: $G = 0.07$ μsec/byte, $L = 23$ μsec and $o = 23$ μsec. Thus the Cray XT4 communication hardware and software are highly optimised.

The reusable model also requires sub-models of the time for the sending processor to execute the *Send* and for the receiving processor to complete the *Receive*. These are easy to derive from the total time to send the message. From the above equations we find that for message sizes less than 1024 Bytes:

$$\begin{aligned} Send_{\leq 1KB,offchip} &= o & (4.3.3)\\ Receive_{\leq 1KB,offchip} &= o + L & (4.3.4) \end{aligned}$$

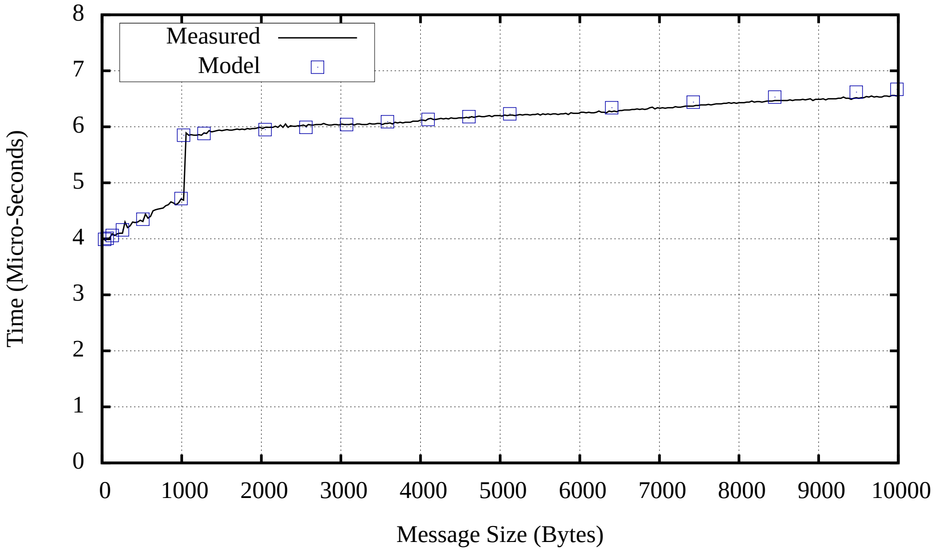


Figure 4.4: Measured and modelled Cray XT4 on-chip MPI end-to-end communication times

and for message sizes greater than 1024 Bytes:

$$Send_{>1KB,offchip} = o + h \quad (4.3.5)$$

$$Receive_{>1KB,offchip} = L + Message\_size \times G + L + o \quad (4.3.6)$$

4.3.2 MPI Send/Receive: On-chip

Figure 4.4 provides half the round-trip in the ping-pong MPI communication benchmark as a function of message size, in the case that the sender and receiver are on the same dual-core chip. We again observe a significant increase in transmission time for message size equal to 1025 Bytes. However, it is unlikely that a handshake operation with the other on-chip core could account for the magnitude of this increase. Instead, note that the slope of the curve for message sizes below 1024 Bytes is larger than the slope for larger messages. These two per-byte transmission costs - G_{copy} and G_{dma} - are given in Table 4.4, with subscripts which denote that it is likely that the larger messages are transmitted using a DMA operation. Since a slower message copy is used for messages smaller than 1025 Bytes, it is likely that the fixed increase at 1025 Bytes is due to the DMA setup cost.

Using the notation in the previous section, and assuming $L = 0$ for on-chip message transmission, we model the total time to send a message smaller than 1025 Bytes as follows:

$$Total\_Comm_{\leq 1KB,onchip} = o_{copy} + Message\_size \times G_{copy} + o_{copy} \quad (4.3.7)$$

Note that o_{copy} is the processing time before and after the message copies on the sender and the receiver, while G_{copy} is the total time per Byte to copy the data from one application buffer to the other. For message sizes larger than 1024 bytes, we let $o = o_{copy} + o_{dma}$ and model the total end-to-end message communication time as:

$$Total\_Comm_{>1KB,onchip} = o + Message\_size \times G_{dma} + o_{copy} \tag{4.3.8}$$

We solve (4.3.7) and (4.3.8) to obtain values for o and o_{copy} which are given in Table 4.4. The value of o is approximately the same as in the off-node model, which greatly increases our confidence in these communication models. A further observation that gives us more confidence about the validity of the models is that the per-byte cost to move the data from sender to receiver is lower on-chip than off-node for all message sizes. Similar to the off-node model the models for $Send$ and $Receive$ are given by the following for message sizes less than 1024 Bytes:

$$Send_{\leq 1KB,onchip} = o_{copy} \tag{4.3.9}$$

$$Receive_{\leq 1KB,onchip} = o_{copy} \tag{4.3.10}$$

and for message sizes larger than 1024 Bytes:

$$Send_{>1KB,onchip} = o_{init} + o_{dma} \tag{4.3.11}$$

$$Receive_{>1KB,onchip} = Message\_size \times G_{dma} + o_{init} \tag{4.3.12}$$

Table 4.4: XT4 Communication Parameters

Off-node	Value	On-chip	Value
G	0.0004 μs/byte	G_{copy}	0.000764 μs/byte
		G_{dma}	0.000091 μs/byte
L	0.36 μs	o	3.77 μs
o	3.85 μs	o_{copy}	1.98μs

Table 4.5: LogGP Model of XT4 MPI Communication

(a) Off-Node Communication Model	
$Total\_Comm_{\leq 1KB,offchip} = o + Message\_size \times G + L + o$	(4.3.1)
$Total\_Comm_{>1KB,offchip} = o + h + Message\_size \times G + L + o$ where $h = L + o_h + L + o_h$	(4.3.2)
$Send_{\leq 1KB,offchip} = o, Receive_{\leq 1KB,offchip} = o + L$	(4.3.3),(4.3.4)
$Send_{>1KB,offchip} = o + h$	(4.3.5)
$Receive_{>1KB,offchip} = L + o + Message\_size \times G + L + o$	(4.3.6)
(b) On-Chip Communication Model	
$Total\_Comm_{\leq 1KB,onchip} = o_{copy} + Message\_size \times G_{copy} + o_{copy}$	(4.3.7)
$Total\_Comm_{>1KB,onchip} = o + Message\_size \times G_{dma} + o_{copy}$	(4.3.8)
$Send_{\leq 1KB,onchip} = o_{copy}, Receive_{\leq 1KB,onchip} = o_{copy}$	(4.3.9),(4.3.10)
$Send_{>1KB,onchip} = o = o_{copy} + o_{dma}$	(4.3.11)
$Receive_{>1KB,offchip} = Message\_size \times G_{dma} + o_{copy}$	(4.3.12)

4.3.3 MPI Allreduce

For completeness, this section develops and validates very simple models of the time to perform an MPI allreduce operation on the Cray XT4. These operations appear in the nonwavefront portions of each iteration in Sweep3D and Chimaera. The performance models can also be applied to other applications that perform such operations. Validation of these simple models indicates that the very abstract models capture the relevant costs, providing insight into the component processing and communication delays in the critical path of this group communications primitive. Furthermore, it gives confidence in our analysis for developing extensions to the basic reusable model for wavefront codes running on CMP nodes as detailed in Section 4.6. In particular the allreduce models act as a sanity check for the existence of contention costs on CMP nodes on the XT4 during wavefront operation.

We note that the MPI allreduce on the XT4 is performed using a binary tree as in Figure 4.5, more specifically called a butterfly pattern. Thus for P processors (where P is a power of 2 number of processors) the longest communication path is $log_2(P)$. Thus we model the total time for single-core nodes as:

$$T_{allreduce} = log_2(P) \times Total\_Comm \tag{4.3.13}$$

and the total time when all-reduce executes on a CMP with C cores per node as:

$$\begin{aligned} T_{allreduce} = {} & [log_2(P) - log_2(C)] \times C \times Total\_Comm_{offchip} + \\ & log_2(C) \times C \times Total\_Comm_{onchip} \end{aligned} \tag{4.3.14}$$

Note that the latter equation is derived assuming that the intra-node operations are serialised on a shared bus. If the CMP node has greater parallelism in the interconnect among the multiple cores, the extra factor of C in each term should be deleted or replaced by another value that suitably represents the interference among the intra-node messages. Figure 4.5 and Figure 4.6 illustrates the allreduce operation on dual-core and quad-core CMP nodes respectively for a total of 8 cores. As can be seen from these figures, the first and second terms in (4.3.14) represent the time to complete the number of stages in which the communications are performed off-node and on-chip (i.e. intra-node) respectively. For a non-power of two number of processors the binary tree algorithm requires additional correction steps to communicate with all the processors [159]. Such corrective steps could be easily incorporated to the above models. As the power of two number of processors are the high performance configurations we predominantly used in our wavefront application model validations, we have omitted the analysis of non-power of two processor model for the MPI allreduce on the Cray XT4.

Table 4.6: Validations for the LogGP MPI allreduce model on a Cray XT4

Number of Processors(P)	Single core per node			Two cores per node		
	Measured (μs)	Predicted (μs)	Error (%)	Measured (μs)	Predicted (μs)	Error (%)
4	15.4	16.3	5.51	24.3	24.5	-0.92
16	29.5	32.6	10.44	56.9	57.0	-0.28
64	46.7	48.9	4.62	89.4	88.5	1.03
256	60.6	65.1	7.40	122.0	121.0	1.18
1024	76.1	81.4	6.92	152.0	152.0	1.88

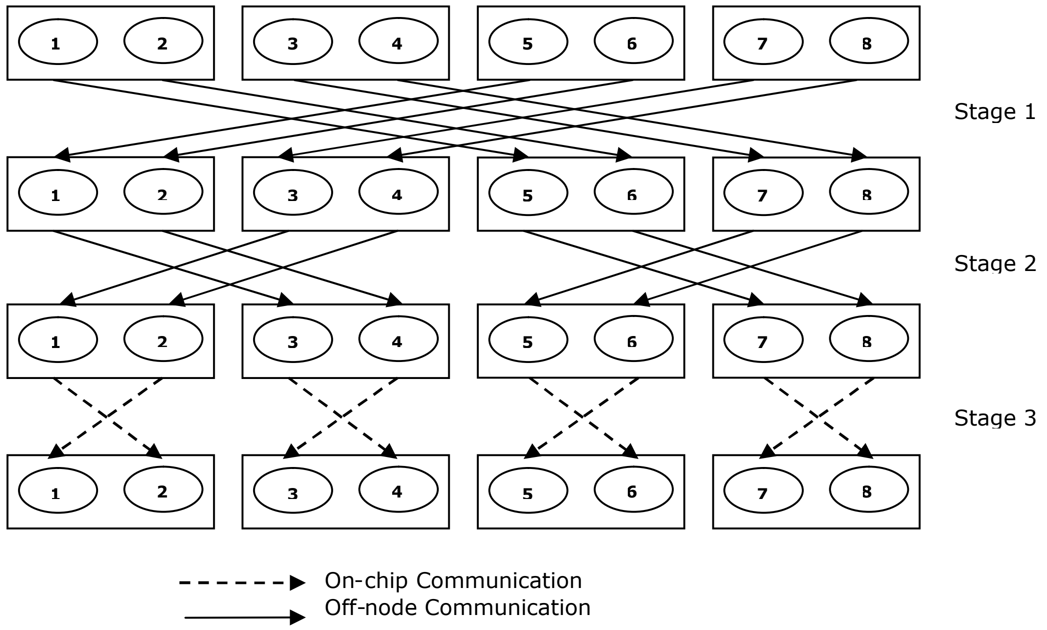


Figure 4.5: MPI allreduce operation on dual-core nodes

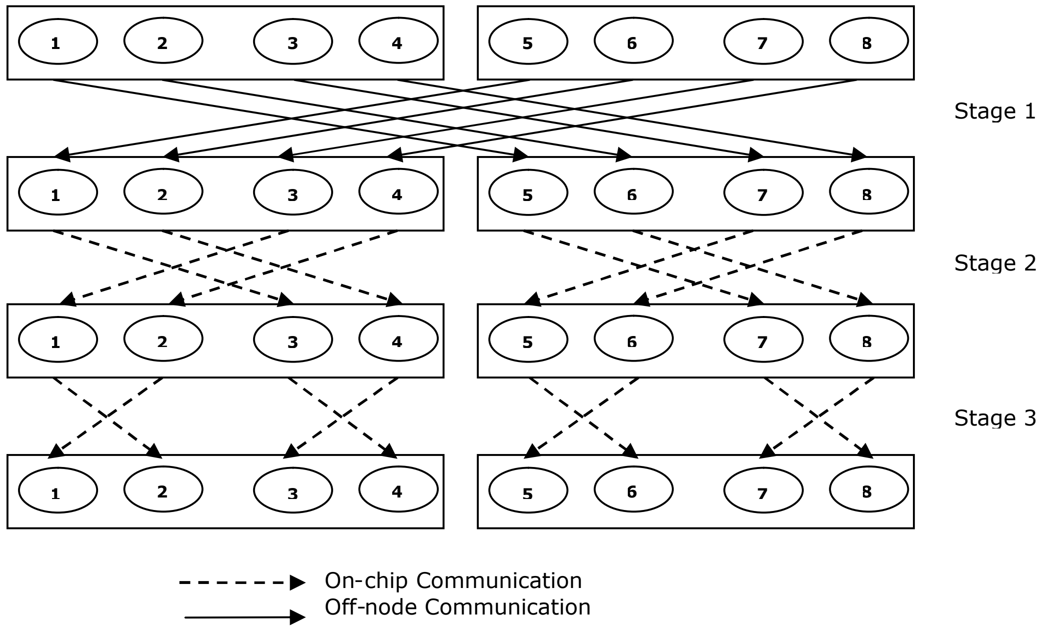


Figure 4.6: MPI allreduce operation on quad-core nodes

Validations of both all-reduce models on the Cray XT4 are provided in Table 4.6. P is the total number of cores performing the allreduce. Note that in the single core per node result, the P processors are each mapped to different nodes. The measured value is the total time for 1000 executions of the all-reduce operation divided by 1000. Note that the largest observed error in the model estimates is approximately 10%.

4.4 Measuring Computation Performance

As mentioned before, compute performance (captured by $W_{g,pre}$ and W_g) is one of the two machine dependent parameters in Table 4.1. An accurate analysis of the issues related to measuring the compute performance parameters is required at this point, in order to ascertain (1) the variance of the observed measures in general and (2) the increase (or decrease) of the measured parameter values, when running an MPI process on all cores/processors per node (given a node made of a CMP processor or a SMP) as oppose to running only one MPI process per node. The latter is an implication related to extending the basic reusable model to the Cray XT4's dual-core nodes which we detail in the next section.

First we note that in order to obtain accurate timings, directly measuring the time to compute a single cell is not practical. Existing timer routines are not sufficiently accurate to measure such an extremely small cost. Our observation for the three applications investigated in this thesis has been that $W_{g,pre}$ and W_g are in the order of one tenth of a micro second, while the resolution of the commonly used timers are approximately 1 or 2 micro seconds. Additionally the overhead associated with calling such a timer routine introduces large amounts of perturbation if each cell is measured individually. Therefore we measure the time to compute a block of tiles per sweep step (i.e. W, or W_{pre}) directly and derive W_g and $W_{g,pre}$ by dividing by the appropriate number of cells. An example of such a code instrumentation used for measuring W is given in listing 4.1.

Listing 4.1: Timer instrumentation of a wavefront code

```
 double precision t2,t3,et2,et3,tot_W,avg_W,tot_WW,avg_WW,Var
 integer iter

FOR EACH OCTANT DO
        FOR EACH TILE DO
            RECEIVE FROM WEST
            RECEIVE FROM SOUTH
                    call timers(t1,et1)
                            COMPUTE (CELLS IN TILE)
                    call timers(t2,et2)
                    iter   = iter   + 1
                    tot_W  = tot_W  + et2-et1
                    tot_WW = tot_WW + (et2-et1)*(et2-et1)
            SEND TO EAST
            SEND TO NORTH
        END FOR
END FOR

 avg_W  = tot_W/iter
 avg_WW = tot_WW/iter
 Var    = avg_WW - (avg_W*avg_W)
 print*,'iteration:',its,'myid: ',myid,' W: ',avg_W
 print*,'iteration:',its,'myid: ',myid,' Var: ',Var
```

Second, we note that the W_{pre} and W should be measured when the application executes on at least four cores, or the number of cores that share a cache or other memory resource at a given node, whichever is larger. Four cores are required regardless of the number of cores per node, so that the code path executed is approximately the same as that which will be executed for larger configurations.

The third consideration is the variance of $W_{g,pre}$ and W_g. One of the assumptions of the basic reusable model is that the data grid cells have comparable or homogeneous computational costs, that is, the cells belong to a regular grid of data. Such an assumption holds for LU, Sweep3D and Chimaera due to the very low variance (less than 0.1 coefficient of variation) observed in measured W_{pre} and W values on the systems used during this research. Nevertheless, we note here the several observations when considering the variance in computation performance in these three applications.

Firstly we observed that the computation cost per cell may vary with respect to the grid size assigned per processor. Particularly, the grid size per processor varies when conducting a weak scaling study where a constant total problem size is continually divided among an increasing number of processors. This can be attributed to cache misses and other effects [8]. Therefore in the validations presented in Section 4.7 we measured W_g and $W_{g,pre}$ for each per-processor grid size for LU[1] and Sweep3D. In both these applications the per processor problem size could be arbitrarily set using an input configuration file, making it possible to easily obtain W_g and $W_{g,pre}$ for any per processor grid size. For instance if a 1000^3 grid point problem is to be solved using a 100×100 2D processor array then an almost exact estimate of the per cell compute time can be obtained by measuring the application running a total problem size of $20 \times 20 \times 1000$ on a 2×2, 2D processor array.

In the case of Chimaera such a measure was impractical to obtain due to it being configured to solve a problem only in strong scaling mode. Each Chimaera problem size of interest (60^3, 120^3 and 240^3) consists of an input file that specifies the properties of each cell. Thus different input files are required to obtain a mixture of cells with comparable properties to be run on a 2×2 processor array. Furthermore, as strong scaling results in differant per processor grid sizes, each case requires a different input file. Therefore we used an estimated W_g based on a small number of processors (e.g. 16, 32, 64, 128, 256, 1024) when speculating performance on larger numbers (> 4000) of processors. As can be seen from the validation in Section 4.7, due to the low variance of compute performance per cell in Chimaera, the predictive accuracy of the model is high even with errors introduced by this estimate. Nevertheless, we note that a more application specific model for W and W_{pre} can easily be used, replacing the r.h.s in equations (4.2.6) and (4.2.7), possibly providing higher model accuracies.

The computation performance per cell may also vary based on the iteration. For instance, Sweep3D contains an extra calculation called fixups for the final 5 iterations in a total of 12 iterations. In our validations for Sweep3D, W_g was measured separately for the iterations without fixups and for iterations with fixups. As the model is for a single iteration of a wavefront application such differences between iterations could be easily accommodated.

If the computation per cell has high variance across the data cells, we classify such a data grid as an irregular or unstructured grid of data. The model extensions required to deal with wavefront application executions on such a data grid is detailed in Chapter 7, Section 7.4.

We also note that model error due to parameter variance (including computation pa-

[1] For weak scaling, LU required a modification to a configuration file and recompiling with different problem sizes as oppose to using its standard problem sizes - A,B,C and D.

rameters) on a machine can be estimated using the measured variance of a parameter and propagating the variance through the model as detailed in Appendix E. This allows to obtain the standard deviation of the final runtime of an application predicted by the model. It provides a confidence interval for the runtimes, where the user is given an approximate set of bounds for the runtime. Due to the very low variance of measured parameter values on the systems used in this research and to present the qualitative trend of the predictions with clarity we have omitted these bound lines in the results presented in this thesis.

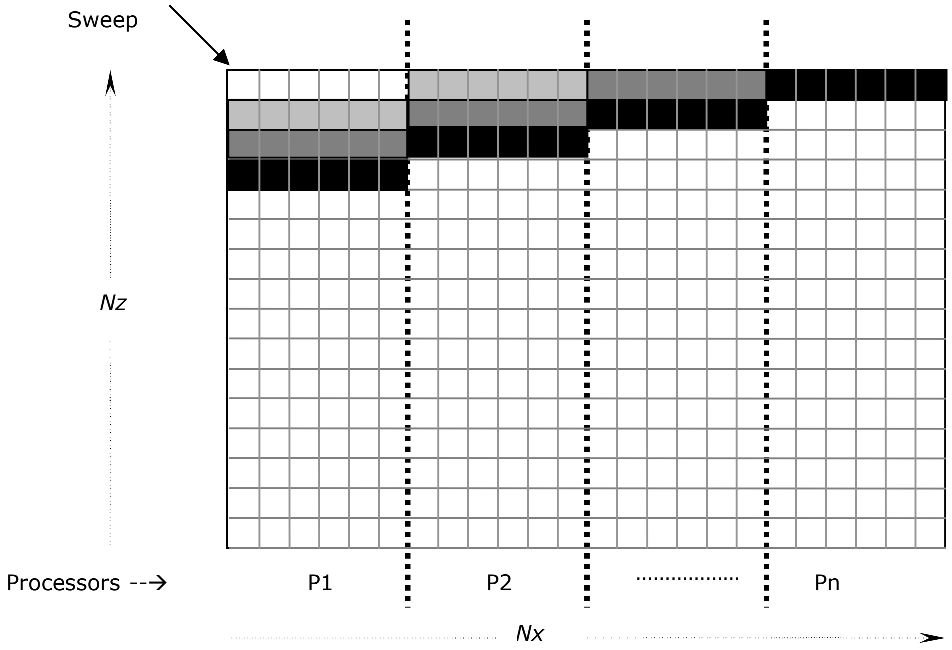


Figure 4.7: Wavefront operation on a 2D data grid

4.5 Deriving a Model for 2D Regular Orthogonal Grids

Table 4.7: Plug-and-play LogGP Model for Wavefront Codes on 2D Data Grids

$W_{pre} = W_{g,pre} \times H_{tile} \times N_x/n$	(4.5.1)
$W = W_g \times H_{tile} \times N_x/n$	(4.5.2)
$StartP_1 = W_{pre}$	(4.5.3)
$StartP_i = StartP_{i-1} + W_{i-1} + Total\_Comm_E$	(4.5.4)
$T_{fill} = StartP_n$	(4.5.5)
$T_{stack} = (Receive_W + W + Send_E + W_{pre})N_z/H_{tile} - W_{pre}$	(4.5.6)
$Time\, per\, iteration = n_{full}T_{fill} + n_{sweeps}T_{stack} + T_{nonwavefront}$	(4.5.7)

The reusable model in Table 4.2 is for modelling pipelined wavefront applications on 3D regular orthogonal grids of data. By ignoring the terms related to either the x or y dimension, this model reduces to a simpler model that can be applied to 2D regular orthogonal grids of data. This is given in Table 4.7 for a 2D grid of data depicted in Figure 4.7. Note that in this case the 2D domain is decomposed on to a 1D array of processors and thus there is only one pipeline fill term, T_{fill} replacing (4.2.10) and (4.2.11). Similarly the four communications operations in (4.2.12) are reduced to two operations per sweep step. The shaded strips in Figure 4.7 depict the initial wavefronts that originates at the top left corner propagating towards the opposite corner.

4.6 Extending the Reusable Model to CMP Nodes on the XT4

The reusable model so far assumed that one MPI process is mapped on to one non-SMP node or a non-CMP processor. This assumption was made to simplify the development of the basic model by sidestepping the analysis and inclusion of any additional time costs that may occur due to shared resources such as a NIC or main memory. In this section we extend the basic reusable model to be applicable to the Cray XT4, when an MPI process is mapped on to each of the cores in the XT4's dual core nodes; That is, in this case there are two MPI processes executing on one XT4 node.

In addition to the issues discussed in Section 4.4 regarding the variance of W_g and $W_{g,pre}$ during execution on multi-core processors, two additional extensions to the basic reusable model are required. First (4.2.9) needs to be modified to specify which of the MPI send and receive operations are on-chip and which are off-node. Second, message contention at shared node resources needs to be accounted for, in (4.2.12). Note that all the communications costs in (4.2.12) should be off-node costs, because the processing of the stack of tiles is limited by the slowest communication costs in each iteration. This follows from the fact that wavefronts progress through the 2D processor grid, using the diagonals as stages of a pipeline. The repeating rate of a pipeline is determined by the repeating rate of the slowest stage after the pipeline is full.

Let the wavefront application be mapped to the multi-core nodes such that the cores at each node form a $C_x \times C_y$ rectangular sub-array in the $m \times n$ processor grid. (See Figure 4.8 where the data grid is mapped on to four nodes each with $C_x \times C_y$ cores). In this case, the off-node communications occur at the edge of the rectangle. Let (i, j) again denote the location of each core in the processor grid where the processor indices start from 1 in both directions. Using this notation, Table 4.8 provides the required modification to (4.2.9) for on-chip communication between two cores on the same node. For example, the $Send_E$ operation in equation (4.2.9) occurs between cores $(i, j-1)$ and $(i+1, j-1)$. This will be off-node if the core $(i, j-1)$ is at the right edge of the $C_x \times C_y$ subarray (i.e., if $i \; mod \; C_x = 0$) and will otherwise be on-chip. The remaining rules are derived in a similar manner.

For message contention, we note that the primary message contention on the Cray XT4 will occur during the DMA transfer of message data from kernel memory to the NIC via the shared bus. Once the message data is in the NIC memory, there should be very little contention since messages are travelling in one direction only between any two nodes and because the NIC has a separate port for each destination node [157]. The time to transmit a message on the bus can be derived from the measured communication primitives in Section 4.3. For each message interference a value of I is added to the appropriate *Send* or *Receive* operation, as specified in Table 4.8.

The number of interference, I values added is derived from the observation of which

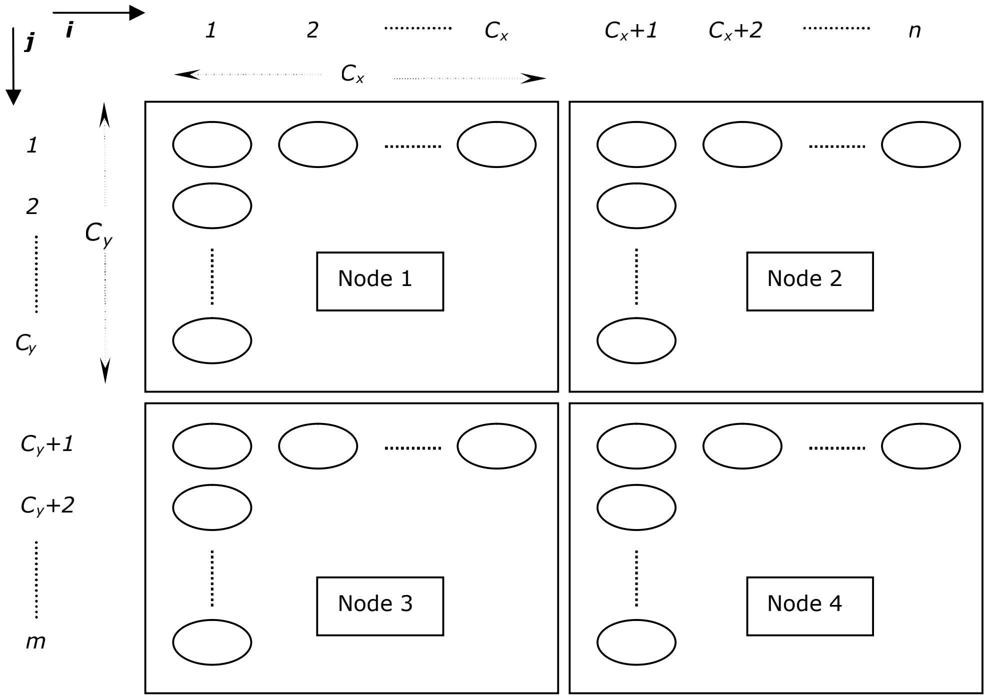


Figure 4.8: Wavefront application mapped to multi-core nodes

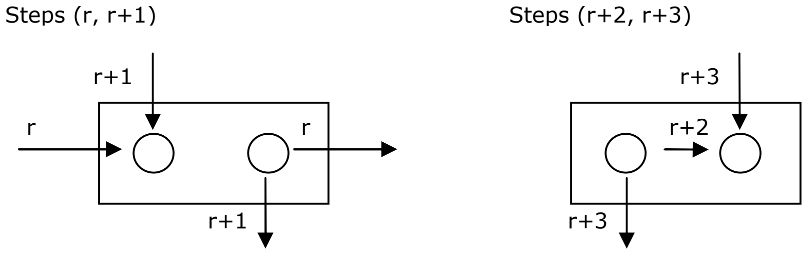


Figure 4.9: Wavefront operation and collisions on dual core nodes

send and receive operations from cores within a single node have the potential to occur at the same time. In other words, we observe which communication operations have the potential to *collide* during wavefront operation. For dual-core and quad-core nodes these *collisions* can be illustrated as in Figure 4.9 and Figure 4.10. The communication operations that occur at the same step are denoted with the same number. As can be seen, for dual-core nodes the communication operations to the south and from the north contend for the shared bus via the NIC[1]. We add an extra I to the north-south communications to account for this. Similarly for quad-core nodes we account for both a north-south and an east west contention by adding an extra I to each send and receive. The derivation of the number of I values added including the illustrations of collisions for, dual, quad, eight and sixteen core nodes is given in Appendix A.

[1] Note that if the dual-core node orientation is changed from the orientation in Figure 4.9 - horizontal cores - by 90 degrees to vertical cores then the contention occurs between the east-west message send and receives.

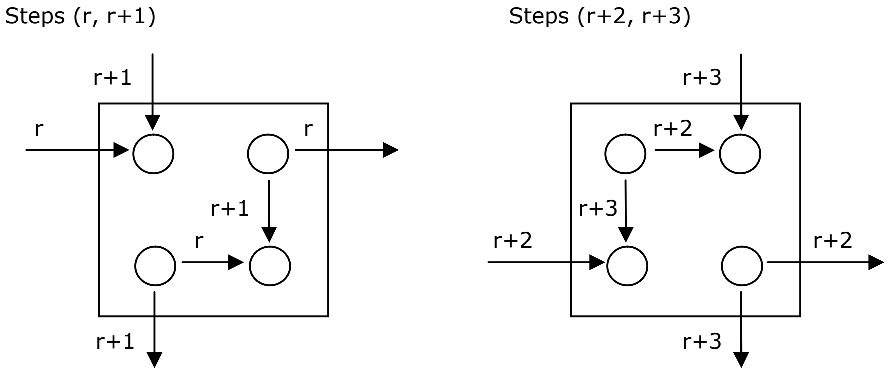


Figure 4.10: Wavefront operation and collisions on quad core nodes

Table 4.8: Re-usable Model Extensions for CMP Nodes

Modifications to Equation (4.2.9)
For $C_x \times C_y$ cores per node, all communication are off-node except the following:
$i\ mod\ C_x \neq 0\ \&\ C_x \neq 1 : Send_E = Send_{onchip,E}$
$i\ mod\ C_x \neq 1\ \&\ C_x \neq 1 : Total\_Comm_E = Total\_Comm_{onchip,E}$
$j\ mod\ C_y \neq 1\ \&\ C_y \neq 1 : Receive_N = Receive_{onchip,N}$
$j\ mod\ C_y \neq 0\ \&\ C_y \neq 1 : Total\_Comm_S = Total\_Comm_{onchip,S}$

Modifications to Equation (4.2.12)
For CMPs with a shared bus to memory
let $I = (o_{dma} + Message\_size \times G_{dma})$
1×2 cores/node : add I to $Receive_N$ and $Send_S$
2×2 cores/node : add I to each $Send$ and $Receive$
2×4 cores/node : add $9I$ to (4.2.12)
4×4 cores/node : add $18I$to (4.2.12)

It should be noted that the additional number of I values added to sends and receives approximate the probable contention and that exact values are difficult to measure accurately (and validate) due to the small magnitude of the communication parameters on the Cray XT4. Additionally, the assumption that the XT4 node architecture will remain unchanged will not hold due to additional innovative hardware and software optimisations and features on future XT4 node architectures. However, we believe that the models presented here capture almost all the important qualitative and quantitative performance factors of wavefront operations on CMP nodes on the XT4. This follows from our practise of developing analytic models to capture abstractly details that matter, but not more than is needed. Additionally the method presented in developing these terms can be easily applied to other node architectures to obtain modified contention terms. In Chapter 5 we further extend this analysis to show how the model gives us insights into the probable qualitative limitations of the CMP-based node architectures in general, as well as specifically on the XT4 at the time of our study, and possible solutions to alleviate such issues.

4.7 Model Validations

Having extended the model to CMP nodes on the XT4, we are now in a position to apply the analytic models to the three concrete applications - LU, Sweep3D and Chimaera. This allows us to validate the predictions from the models on the Cray XT4 system. The following subsections present validations for each application. We first discuss the reasoning behind the derivations of the concrete models by applying the reusable model, followed by quantitative validations.

4.7.1 NPB - LU

For LU, the application parameters in Table 4.1 applied to the reusable model results in the top level analytic model given by (4.7.1).

$$Time\ per\ iteration = 2T_{fullfill} + 2T_{stack} + T_{nonwavefront} \tag{4.7.1}$$

We observed minor differences in the average W_g and $W_{g,pre}$ times between the forward and backward sweeps of LU. Thus, to increase prediction accuracies we measured these times for both forward and backward sweeps separately, and applied them to form a separate model for each. The total sweep time was given by the sum of these two models. The $T_{nonwavefront}$ term for LU consists of a four-point stencil computation which occurs at the end of the two sweeps, for computing the right-hand-side vector as detailed in Section 3.2.1. We have modelled this based on the time to compute the four-point stencil operation plus the time for exchanging boundary values during this operation in (4.7.2).

$$T_{nonwavefront} = (N_x/m) \times (N_y/n) \times N_z \times W_{g,rhs} + 4Total\_Comm \tag{4.7.2}$$

The message size for $Total\_Comm$ is given by $(N_x/n) \times N_z \times 80$ for north-south and $(N_y/m) \times N_z \times 80$ bytes for east-west communications, where each processor will send two rows of a boundary to its nearest neighbour. Each row consists of cells that contain 5 double-precision floating-point values, making the total message size per near neighbour communication 80 bytes (given that a double is stored as an 8 byte value).

Table 4.9 and Table 4.10 provide the total runtime of LU on the Cray XT3 compared with the runtime predicted by the model in (4.7.1), for two weak-scaling problem sizes - 64^3 and 102^3 cells per processor. Note that the model validations are done on the XT3 due to the ORNL Jaguar system only consisting of a Cray XT3 partition during the time of this validation. Moreover, the model used here is the basic (non-CMP) reusable model and the validations mapped one MPI process to one XT3 node, which preserves the conditions required to obtain accurate predictions from the basic model. In this case we used off-node communication parameter values of $o = 4.038\mu Sec$, $L = 0.33\mu Sec$ and $G \approx 0.0005\mu Sec/Byte$, which we obtained from the model in Table 4.5 using the XT3 node-to-node communication benchmark data.

Table 4.9: LU Model Validation on Jaguar (Cray XT3) - 64^3 cells per processor

NPE	n	m	Prediction (Sec)	Execution (Sec)	Error (%)	Comput (Sec)	Comm (Sec)
4	2	2	172.08	176.79	-2.67	171.33	0.74
8	4	2	174.36	184.03	-5.25	173.6	0.76
16	4	4	176.65	185.16	-4.6	175.87	0.77
32	8	4	181.22	192.54	-5.88	180.41	0.81
64	8	8	185.79	194.75	-4.6	184.95	0.84
128	16	8	194.93	204.18	-4.53	194.03	0.9
256	16	16	204.06	213.58	-4.46	203.1	0.96
512	32	16	222.35	232.42	-4.33	221.26	1.09
1024	32	32	240.62	251.31	-4.25	239.41	1.21
2048	64	32	277.19	289.1	-4.12	275.72	1.47

Table 4.10: LU Model Validation on Jaguar (Cray XT3) - 102^3 cells per processor

NPE	n	m	Prediction (Sec)	Execution (Sec)	Error (%)	Comput (Sec)	Comm (Sec)
4	2	2	729.32	744.72	-2.07	728.05	1.27
8	4	2	734.55	752.29	-2.36	733.26	1.29
16	4	4	739.78	763.91	-3.16	738.48	1.31
32	8	4	750.24	774.6	-3.14	748.9	1.34
64	8	8	760.7	785.75	-3.19	759.33	1.37
128	16	8	781.62	807.07	-3.15	780.18	1.45
256	16	16	802.53	831.63	-3.5	801.03	1.51
512	32	16	844.38	870.68	-3.02	842.72	1.65
1024	32	32	886.2	913.22	-2.96	884.42	1.78
2048	64	32	969.89	1018.75	-4.8	967.82	2.06

4.7.2 Sweep3D

The top level model for Sweep3D is given in (4.7.3). In this case the critical path time can be elucidated as follows using the processor indexing in (see Figure 3.7):

$$Time\ per\ iteration = 2T_{diagfill} + 2T_{fullfill} + 8T_{stack} + T_{nonwavefront} \qquad (4.7.3)$$

The sweeps start at the top right corner processor, (n, m), then eight sweeps are performed (two per corner) to end back at processor, (n, m). The critical path consists of the sum of the following steps:

1. Time for all the tiles on processor (n, m) to complete sweeps 1 and 2: $2 \times T_{stack}$
2. Time gap between sweep 2 ending at processor (n, m) and sweep 3 starting at processor $(n, 1) : 1 \times T_{diagfill}$
3. Time for all the tiles on processor $(n, 1)$ to complete sweeps 3 and 4: $2 \times T_{stack}$
4. Time gap between sweep 4 ending at processor $(n, 1)$ and sweep 5 starting at processor $(1, m) : 1 \times T_{fullfill}$
5. Time for all the tiles on processor $(1, m)$ to complete sweeps 5 and 6: $2 \times T_{stack}$

6. Time gap between sweep 6 ending at processor $(1, m)$ and sweep 7 starting at processor $(1, 1) : 1 \times T_{diagfill}$

7. Time for all the tiles on processor $(1, 1)$ to complete sweeps 7 and 8: $2 \times T_{stack}$

8. Time gap between sweep 8 ending at processor $(n, 1)$ and sweep 8 ending on processor $(n, m) : 1 \times T_{fullfill}$

9. Time to perform any non-wavefront computations

The $T_{nonwavefront}$ term for Sweep3D consists of two all-reduce operations plus several other operations that we have ignored due to their negligible contribution to the total runtime. Table 4.11 and Table 4.12 provides the total runtime of the sweep portion of Sweep3D on the Cray XT4 compared against the runtime predicted by the model in (4.7.3), for two strong-scaling problem sizes - the 1 billion cell problem and the 20 million cell problem. The model used here includes the CMP extensions (for dual core nodes) described in Section 4.6. Sweep3D has fix-up calculations for the final 5 iterations. Therefore we have measured the W times for both types of iterations (i.e. iterations with and without fix-up calculations).

Table 4.11: Sweep3D Model Validation on Jaguar (Cray XT4) - 1000^3 total problem size, $H_{tile} = 2, mmi = 6$

NPE	n	m	Nx/n	Ny/m	Prediction	Execution	Error	Compute	Comm
					(Sec)	(Sec)	(%)	(Sec)	(Sec)
1K	32	32	32	32	36.01	38.56	-6.62	34.99	1.03
2K	64	32	16	32	21.78	24.98	-12.81	20.78	1.00
4K	64	64	16	16	11.78	13.36	-11.83	10.79	0.99
8K	128	64	8	16	7.34	8.43	-12.87	6.42	0.92

Table 4.12: Sweep3D Model Validation on Jaguar (Cray XT4) - 20×10^6 total problem size, $H_{tile} = 2, mmi = 6$

NPE	n	m	Nx/n	Ny/m	Prediction	Execution	Error	Compute	Comm
					(Sec)	(Sec)	(%)	(Sec)	(Sec)
1K	32	32	9	9	1.23	1.38	-10.68	0.99	0.24
2K	64	32	5	9	0.97	1.11	-13.05	0.72	0.25
4K	64	64	5	5	0.73	0.94	-22.41	0.47	0.26
8K	128	64	3	5	0.68	0.90	-23.85	0.41	0.27

4.7.3 Chimaera

Using the application parameter values in Table 4.1, we can form the top level model for Chimaera as given by (4.7.4).

$$Time\ per\ iteration = 2T_{diagfill} + 4T_{fullfill} + 8T_{stack} + T_{nonwavefront} \tag{4.7.4}$$

Similar to Sweep3D, in Chimaera the sweeps start at the top right corner processor, (n, m), then eight sweeps are performed (two per corner) to end back at processor, (n, m). The critical path consists of:

1. Time for all the tiles on processor (n, m) to complete sweep 1 and 2: $2 \times T_{stack}$
2. Time gap between sweep 2 ending at processor (n, m) and sweep 3 starting at processor $(n, 1)$: $1 \times T_{diagfill}$
3. Time for all the tiles on processor $(n, 1)$ to complete sweep 3: $1 \times T_{stack}$
4. Time gap between sweep 3 ending at processor $(n, 1)$ and sweep 4 starting at processor $(1, m)$: $1 \times T_{fullfill}$
5. Time for all the tiles on processor $(1, m)$ to complete sweep 4: $1 \times T_{stack}$
6. Time gap between sweep 4 ending at processor $(1, m)$ and sweep 5 starting at processor $(n, 1)$: $1 \times T_{fullfill}$
7. Time for all the tiles on processor $(n, 1)$ to complete sweep 5: $1 \times T_{stack}$
8. Time gap between sweep 5 ending at processor $(n, 1)$ and sweep 6 starting on processor $(1, m)$: $1 \times T_{fullfill}$
9. Time for all the tiles on processor $(1, m)$ to complete sweep 6: $1 \times T_{stack}$
10. Time gap between sweep 6 ending at processor $(1, m)$ and sweep 7 starting on processor $(1, 1)$: $1 \times T_{diagfill}$
11. Time for all the tiles on processor $(1, 1)$ to complete sweep 7 and 8: $2 \times T_{stack}$
12. Time gap between sweep 8 ending at processor $(n, 1)$ and sweep 8 ending on processor (n, m): $1 \times T_{fullfill}$
13. Time to perform any non-wavefront computations

Table 4.13 provides the total runtime of the sweep portion of Chimaera on the Cray XT4 compared to the runtime predicted by the model equation (4.7.4), for a strong-scaling problem size of 240^3 cells. The number of iterations is determined by a convergence criterion. For this problem size, convergence is achieved at 419 iterations.

Table 4.13: Chimaera Model Validation on Jaguar (Cray XT4) - 240^3 total problem size

NPE	n	m	Nx/n	Ny/m	Prediction (Sec)	Execution (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
256	16	16	15	15	320.35	365.15	-12.27	304.11	16.25
1024	32	32	7	8	104.09	110.43	-5.74	89.87	14.22
4096	64	64	4	4	43.33	54.13	-19.96	27.86	15.47

4.7.4 Discussion on Validation Results

The validations for LU, Sweep3D and Chimaera provide excellent qualitative and quantitative accuracy. More specifically, we have observed that the model predicts with a maximum error of 20% when the computation time significantly (over 50%) dominates the total run time. We classify such configurations as high performance configurations, as in such cases the significant majority of the runtime is devoted to performing useful computations (i.e. solving the problem) as opposed to communications. Therefore, non-high performance configurations are of less practical interest as suitable configurations for production runs. In these latter cases, the abstract communication and contention model leads to somewhat larger errors (that is, in the order of 20-25%).

The validations for LU have less than 10% error due to the large per processor problem size where the computation time far outweights the communication time. However, the Sweep3D 20 million cell problem and the Chimaera 240^3 problem show that, when the problem size per processor is small, the model produces larger errors. When the problem size per node is small, the communication dominates the total execution time. We attribute the errors occurring in such configurations, as well as the general trend of the model under prediction to the following:

1. The model assumes perfect processor allocation, i.e. logical near neighbour processors will be allocated to physical near neighbour processors on the Cray XT4. The processor allocation scheduler of the XT4 (PBS) attempts to cluster processors of the same job to near-neighbour processors. However, because Jaguar is a shared machine, the scheduler will deviate from this 'ideal' processor allocation.

2. The communication models are too abstract to capture small message performance accurately. For example, overheads at routers are not accounted for in the models. The latency (L) value will increase if a message has to do multiple hops in order to arrive at a logical neighbour processor. Such multiple hops will occur due to issue 1 above.

3. The communication model does not capture the increased L for Cray XT4's cabinet-to-cabinet communication time.

4. Experimental errors during measuring computation, particularly with small per processor problems, cause the model to underestimate the W_g and $W_{g,pre}$ times resulting in under predictions. In our experiments we see that the loop overhead for the x and y dimensions in a tile becomes more significant as the tile size decrease. But this is unaccounted for when using (4.2.6) and (4.2.6) to estimate W_g and $W_{g,pre}$ at large machine scale.

5. The variance caused by inhomogeneous cells may introduce runtime variances. For example Chimaera consists of minor variations in the computation times of different cells.

6. The measured actual application runtimes were performed while the system was busy with other jobs, thus there is network contention as well as other perturbations including system noise. The model does not account for such perturbations.

In spite of these discrepancies, the model has proven to be accurate for a variety of problem sizes and configurations on not only the XT4 (as shown in the above results) but also several other HPC systems. More model validations are detailed in Chapter 5, 6, 7 and Appendix B. These validations include more results from the Cray XT4, as well as an Intel Xeon/InfiniBand cluster. The level of accuracy is more than sufficient to conduct various speculative studies and investigate optimisations for any given existing or imaginary wavefront code. Chapters 5 and 7 present a comprehensive investigation of these issues.

4.8 Summary

To our knowledge, plug-and-play performance models - in which the user only needs to specify a few input parameter values in order to obtain performance predictions for application codes with different behaviour - has not previously been developed. An open question addressed in this research is whether building in the various possible behaviours leads to a more complex set of equations, possibly negating the advantages of the model generality. The models developed in this chapter show that for the varied behaviours in wavefront applications, it has been possible to construct a set of equations that are as simple as the equations that are tailored to a given application. This was an unanticipated result that may not hold for other classes of applications. However, the results are encouraging for this important class of application and may provide an incentive to extend the study more widely.

5 Wavefront Application and Platform Design

A key advantage of a reusable performance model for wavefront computations is the ease with which it can be applied to model various wavefront applications. Particularly, the utility of a good performance model is to provide predictive insights into the performance issues and behaviours of these applications in an efficient, accurate and low-cost manner. As such, in this chapter we apply the plug-and-play wavefront application model to illustrate its utility to investigate performance of wavefront codes. We use the Cray XT4 system as the target platform for our analysis. More specifically, we evaluate application design and configuration in section 5.1, hardware platform procurement questions such as platform sizing and configuration in section 5.2, hardware platform design alternatives, particularly the number of cores per node in section 5.3, application bottlenecks in section 5.4 and finally a possible application re-design to alleviate one of the bottlenecks in section 5.5.

We illustrate the performance model led analysis for the two particle transport benchmarks, Chimaera and Sweep3D, noting that the model can be applied in a similar manner to LU or any other wavefront benchmark or production code of interest. The applications illustrate the versatility of the analytic model in supporting the rapid evaluation of a number of system configuration and design alternatives. Throughout the results, the evaluation for Chimaera is done with a problem size of 240^3 cells, which is a large current cubic problem size available as part of the benchmark. We evaluate Sweep3D with two problem sizes of interest to LANL [17]: 1 billion cells and 20 million cells. Unless otherwise noted, for both problem sizes of Sweep3D, we set the number of angles, mmo, to six. The Chimaera code requires 419 iterations to complete a time step for the problem provided with the benchmark. Unless otherwise stated, in this chapter we set the number of iterations per time step in Sweep3D to 120 which we anticipate will be more representative of many actual particle transport simulations than the default value of 12.

5.1 Application Design: H_{tile}

As shown in the basic application parameters (Table 4.1) and in previous studies of Sweep3D, the number of cells computed per sweep step is a key configuration parameter. The number of cells computed per sweep step is determined by (1) the total problem size and the total number of processors and (2) the height of a tile set by the H_{tile} parameter. In this section we discuss the quantitative and qualitative affects of the latter, leaving the former to be discussed in the next section as a platform sizing and configuration issue.

A larger value of H_{tile} leads to a larger ratio of computation to communication, as shown in equations (4.2.6) and (4.2.7). This leads to longer pipeline fill times as shown in equations (4.2.8) and (4.2.9), but also to lower communication costs because the communica-

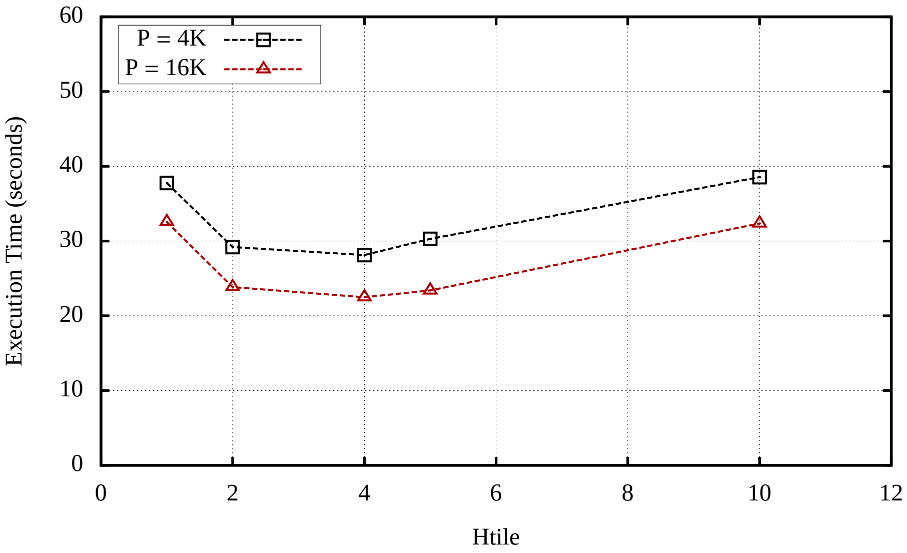


Figure 5.1: Execution time vs. H_{tile}: Sweep3D 20 Million cell problem

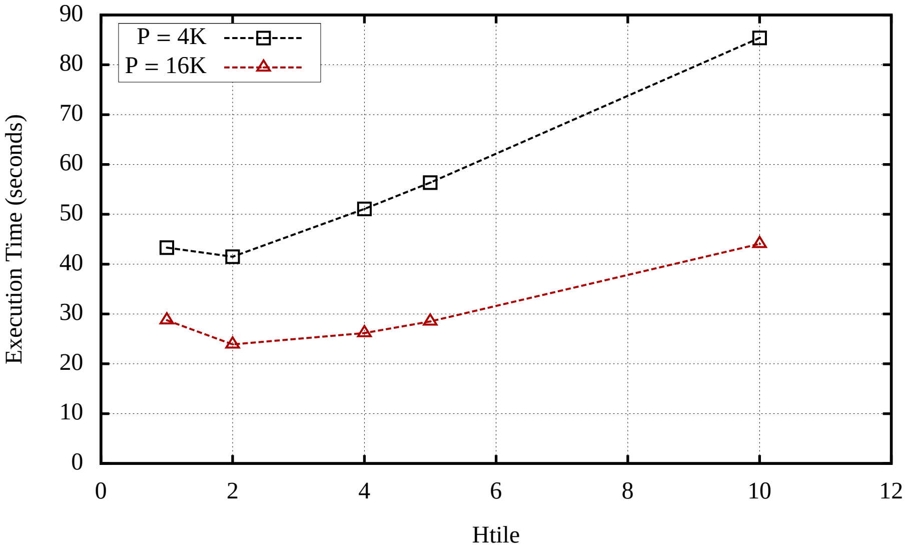


Figure 5.2: Execution time vs. H_{tile}: Chimaera $240 \times 240 \times 240$ cell problem

tion overhead (o) and latency (L) occurs less frequently as shown in equation (4.2.12). A key software configuration question that is easily addressed using the model is the value of H_{tile} to use for a given application code, problem size, and number of processors, in order to achieve minimum execution times. Figure 5.1 and Figure 5.2 show the execution time per time step[1] vs H_{tile} for Sweep3D and Chimaera on the 20 million and 240^3 problem sizes, respectively. For Chimaera we illustrate the speculative case assuming that the tile size can be modified.

For each benchmark and problem size, we provide a curve for a small system configuration (4096 processors) and the maximum number the problem can practically run on (16K processors). In each case, H_{tile} in the range of 2, 4 or 5 minimises the execution time. Results for Sweep3D with the 10^9 problem size on 4K - 128K processors (up to 32K processors shown

[1] for comparison in this section we consider one time step to consist of 480 iterations for Sweep3D and 419 iterations for Chimaera.

in Figure 5.3), also show that H_{tile} in the range of 2 to 5 minimises execution time. In contrast, previous work evaluating Sweep3D on the IBM SP/2 that has higher communication overhead and latency, found H_{tile} in the range of 5 to 10 (i.e., $mk = 10$ and $mmi/mmo = 0.5$ or 1) minimised execution time [8].

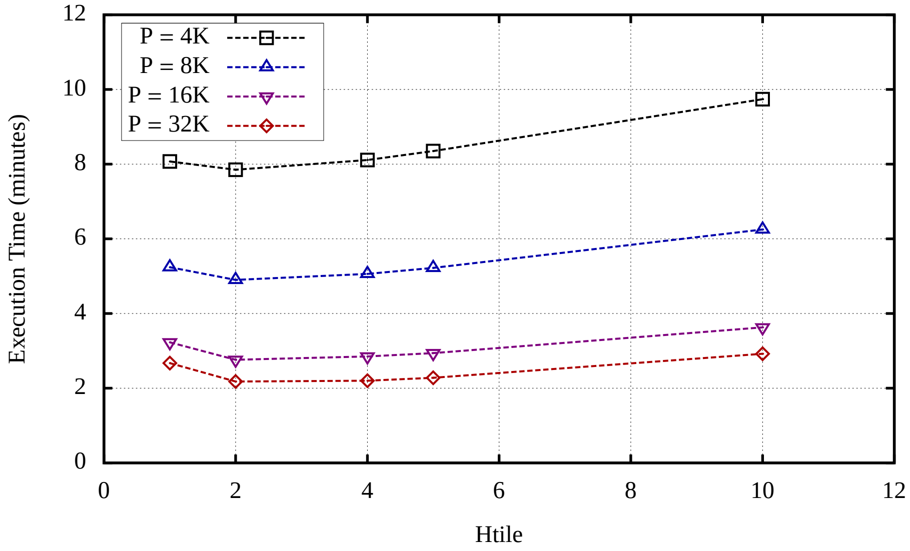


Figure 5.3: Execution time vs. H_{tile}: Sweep3D 1 Billion cell problem

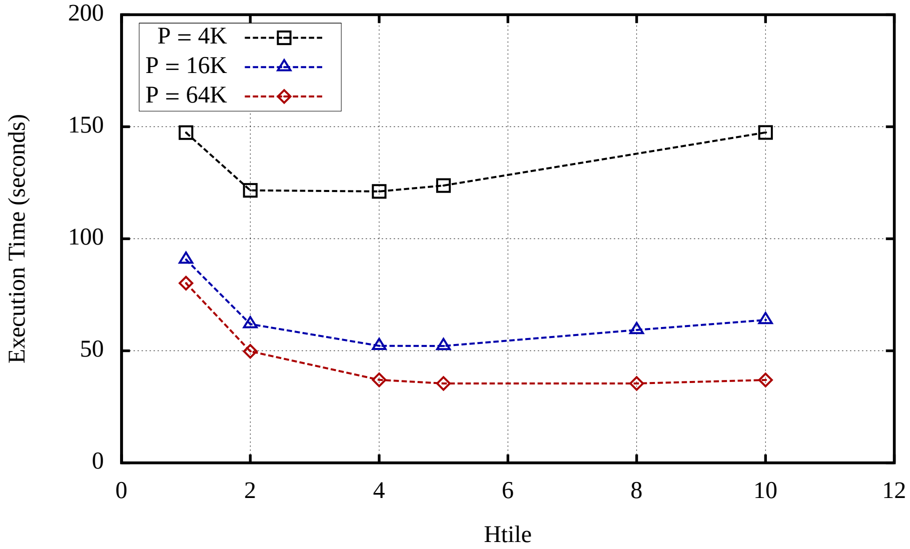


Figure 5.4: Execution time vs. H_{tile}: Chimaera $240 \times 240 \times 960$ cell problem

Figure 5.4, shows the qualitative effects of H_{tile} when the z dimension is larger than the other dimensions. In this case H_{tile} = 2, 4 or 5 provides an increasing reduction in runtime with a maximum of just over 50% (compared with H_{tile} = 1) on 64K processors. Our results illustrate the ability of the model to rapidly evaluate software design modifications in order to determine whether the implementation effort is justified. We use H_{tile} = 2 for the results in the remainder of this Chapter, noting that in some cases the execution time will be slightly lower if H_{tile} is set to 4 or 5. One further point of interest from Figure 5.1 and Figure 5.2 is that on 16K processors the execution time for one iteration of Sweep3D with a problem size of 20 million cells (with 480 iterations to complete a time step) is very similar to the execution time of Chimaera with a problem size of 240^3 cells (requiring 419 iterations). These two benchmarks perform different processing. For example, Sweep3D computes six angles while Chimaera computes ten angles. Of interest is that the codes have qualitatively similar processing costs.

5.2 Platform Sizing and Configuration

For a given particle transport problem size of interest, increasing the number of processors decreases execution time, but with diminishing returns. Model results in Figure 5.5 illustrate this for Sweep3D for the 10^9 problem. In this figure, we show the execution time for 10^4 time steps, when the code uses both processors in each dual-core XT4 node. We assume a Sweep3D type production code simulating 30 energy groups [17], which implies a 30-fold increase in execution time compared with the execution time for a single energy group[2]. The measured values illustrated for Sweep3D are the results from multiplying the actual Sweep3D benchmark run, performed on the Cray XT4, by 30 and 10^4.

These values of interest to LANL are used to illustrate the system sizing question in the context of production problems of interest to the organisations that own the benchmarks. The related model results for Chimaera 240^3 are illustrated in Figure 5.6. In this case we have kept the default number of energy groups solved at 16 but have used an H_{tile} of 2 due to better performance as predicted in the previous section.

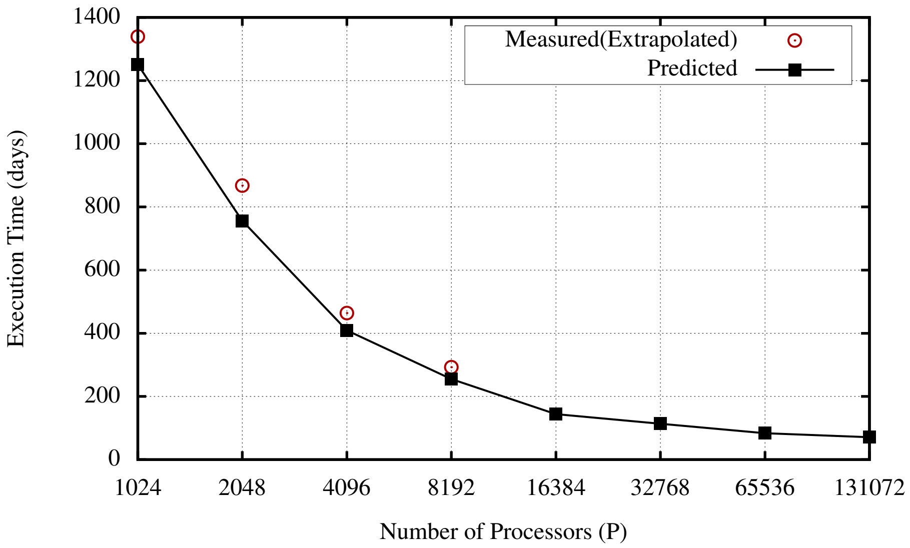


Figure 5.5: Execution time vs. System size: Sweep3D Billion cell problem, 10^4 time steps, 30 energy groups, 120 iterations, $H_{tile} = 2$

Figures 5.5 and 5.6 also provide measured code execution times for the numbers of nodes that are available in the ORNL Cray XT4. Note that we obtained an error in the order of 10% in the predicted execution times. These results illustrate that the projected execution times are qualitatively correct and sufficiently accurate to support accurate decisions concerning

[2] recall that Sweep3D as a benchmark solves only 1 energy group, while Chimaera solves 16.

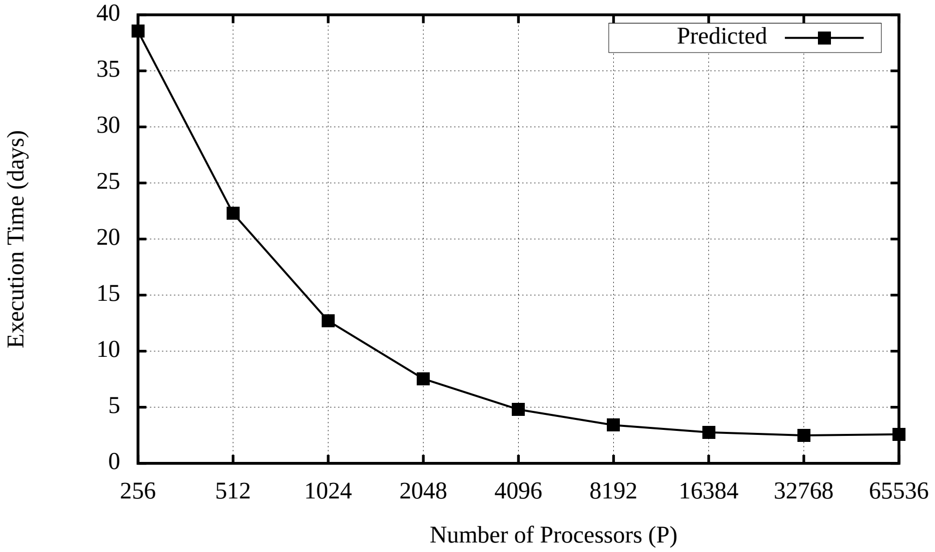


Figure 5.6: Execution time vs. System size: Chimaera 240^3 cell problem, 10^4 time steps, 16 energy groups, 419 iterations, $H_{tile} = 2$

how many processors should be allocated to a given particle transport simulation. For both Sweep3D and Chimaera (as shown in Figure 5.5 and Figure 5.6 respectively) the trade-off in execution time versus the number of nodes is complex. For Sweep3D there are diminishing but perhaps still significant returns as the number of processors increases beyond 16K. A given user requiring nearly the minimum possible execution time may determine that the desired system size is 64K or 128K cores. On the other hand, due to the diminishing returns from 32K processors to 64K processors, another user may want to trade-off the execution time of one problem on 64K processors against solving two 1 billion cell problems simultaneously, each on half of the 64K processors.

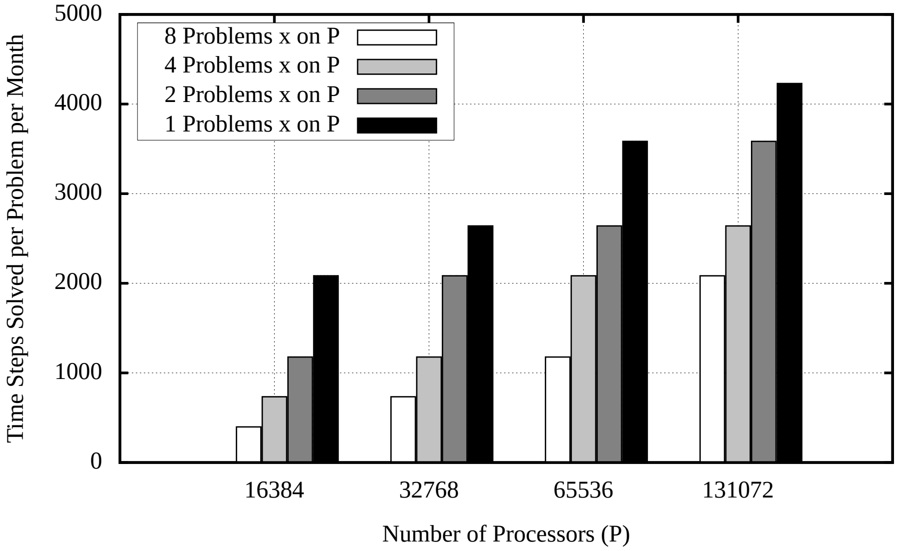


Figure 5.7: Throughput vs. Partition Size (Sweep3D 10^9 Cells, 10^4 time steps, 30 energy groups 120 iterations, H_{tile} = 2)

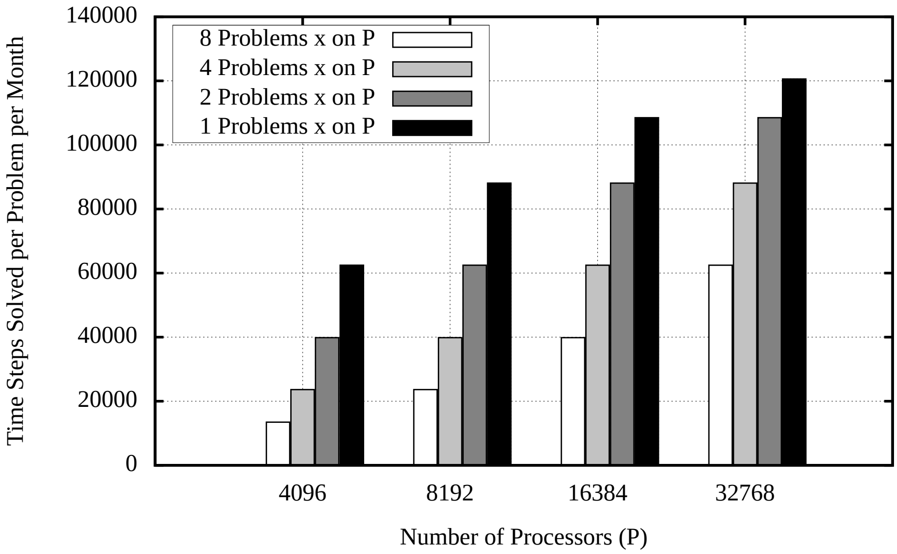


Figure 5.8: Throughput vs. Partition Size (Chimaera 240^3 Cells, 10^4 time steps, 16 energy groups 419 iterations, H_{tile} = 2)

We provide results for evaluating this trade-off in Figure 5.7 and Figure 5.8. The black bars provide the number of time steps solved per month when a single problem executes on the given number of processors. The other bars in the figures show the number of time steps completed per month by each of 2, 4, or 8 particle transport simulations that are executed in parallel on equal-size partitions of the given number of processors. For example, the dark gray bars show the number of time steps solved per month in each of two problems solved when the given number of processors is partitioned in half. Note that when two 1 billion cell Sweep3D problems each run on half of 32K processors, just over 2000 time steps are solved per month in each of the problems. This means that approximately five months or 150 days are required to execute 10,000 time steps. Figure 5.7 also shows that approximately 150 days are required to simulate the 10,000 time steps on 16K processors. Hence, Figure 5.7 is another way to view the performance vs system size. In the case of 128K processors in Figure 5.7, two parallel simulations execute at approximately 7/8 the rate of a single simulation, providing perhaps an attractive alternative for some users. Similarly for Chimaera, we see from Figure 5.8, that two 240^3 problems on on 32K processors complete in approximately 11/12 the rate of a single run on the same number of processors. The results in Figure 5.7 and Figure 5.8 illustrate that a given site may want to consider the total number of simulations that need to be run when making procurement decisions or when allocating system resources to particle transport simulations.

It is desirable to achieve a good trade-off between minimising the execution time for a single simulation (R) by running it on as many processors as possible, and maximising the total number of simulations that complete per unit time (X) by partitioning the available processors so that simulations run in parallel. It is possible to quantify this trade-off, as illustrated in Figure 5.9 for the 1 billion cell problem of Sweep3D. Two curves are plotted as a function of partition size for parallel simulations on 128K cores. When the partition size is 32K cores, four 1 billion cell simulations are run in parallel. The lower curve is the value of R/X, the ratio of time to complete each 1 billion particle simulation divided by the number of simulations that

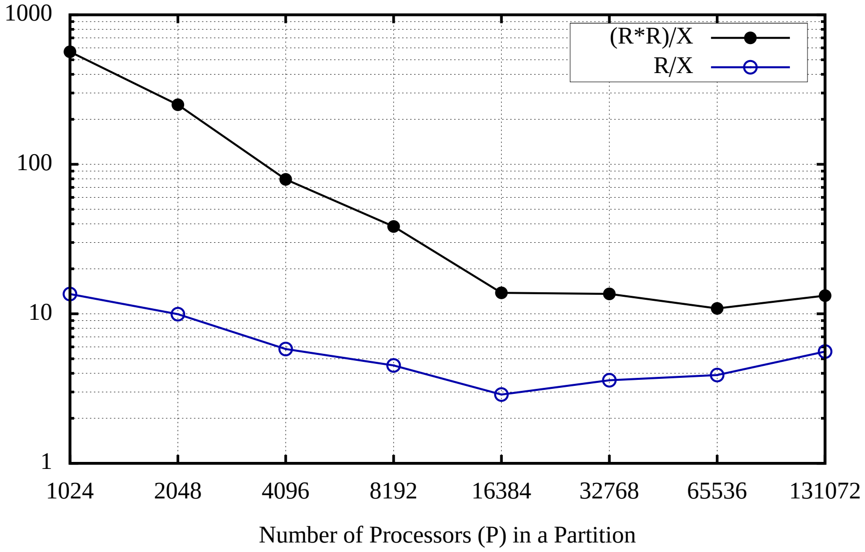


Figure 5.9: Optimising Partition Size (Sweep3D 1 Billion Cells, Total number of available processors = 128K)

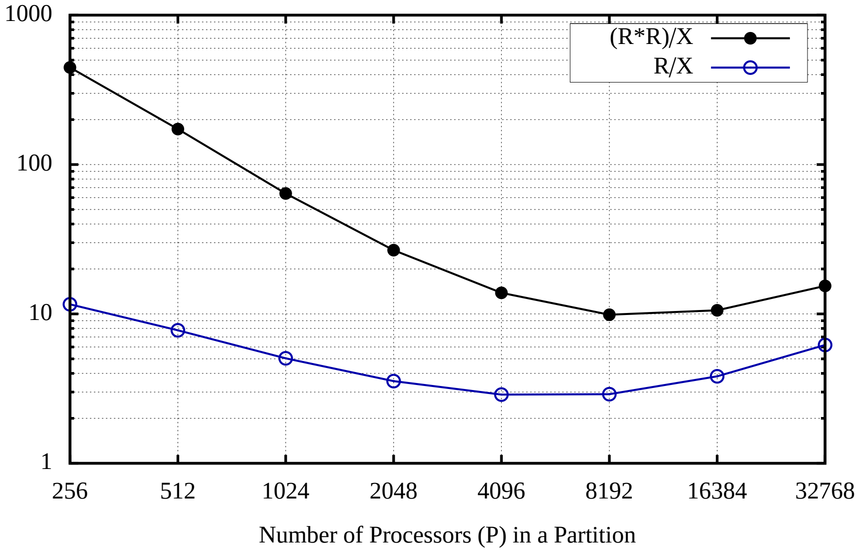


Figure 5.10: Optimising Partition Size (Chimaera 240^3 Cells, Total number of available processors = 32K)

complete per time R. This ratio is minimised when the partition size is 16K processors and thus 8 simulations are run in parallel. The upper curve is R^2/X, which places greater emphasis on minimising the execution time for each simulation, and is optimised at 64K processors per simulation. A given site or user can compare these optimised partitions with the results in Figure 5.5 and Figure 5.7 to arrive at a decision about how to configure the system.

Similarly for Chimaera, we select a total of 32K processor cores and partition it to run multiple problems simultaneously. As illustrated in Figure 5.10, for the 240^3 problem the optimum number of processors to run can be determined by the minimum values of the two curves. In this case if the objective is to obtain better throughput then the partition size is 4K processors. If the time to solution (run time) is more important than the partition size should be set to 8K processors on a total of 32K processors.

5.3 Platform Design: Multi-core Nodes

We next examine the platform design issue of how many cores per node would be desirable for the important class of large particle transport simulations that make up a large fraction of the workload at places such as LANL and AWE. These results are obtained using the model extensions provided in Table 4.8, which assume a shared bus architecture within each node but can easily be modified for other node architectures. Results are provided here to illustrate the utility of the model in providing insights into the question of interest.

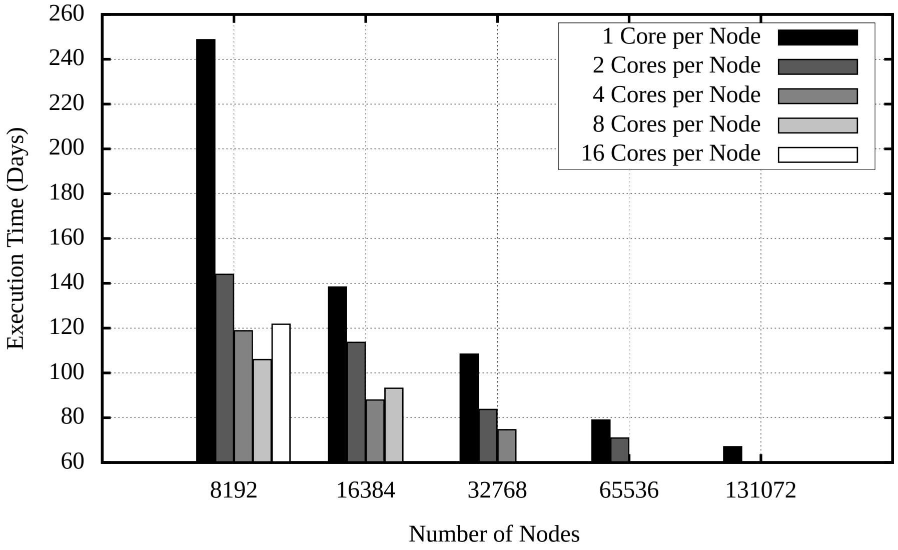


Figure 5.11: Execution time on multi-core nodes (Sweep3D 1 Billion Cells, 10^4 time steps, 30 energy groups, 120 iterations, H_{tile} = 2)

Figure 5.11 provides the execution time for a 1 billion particle transport simulation versus the number of nodes on the platform, and for various possible numbers of cores per node ranging from one core per node to sixteen cores per node. Because there are diminishing returns when the simulation runs on increasing numbers of nodes (with one core per node) there are also diminishing returns for increasing the number of cores per node. Note also that in these results, two cores on a given number of nodes (e.g. 64K nodes) provide slightly better execution time than four cores on half the nodes (e.g. 32K nodes) due to the shared bus architecture.

If the target execution time is approximately the execution time on 64K single-core nodes, then the figure shows that this performance can be nearly achieved with 32K dual-core nodes or 16K quad-core nodes. An 8K-node system with 16 cores per node has the same total number of cores as 32K quad-core nodes (and thus twice as many cores as a system with 32K dual-core nodes), but execution time is degraded due to contention for the shared bus. However, if the 16-core node is provisioned with a separate shared bus, shared memory, and NIC for each group of 4 cores, then the execution time on the system with 8192 nodes would be the same as the execution time for the 32K quad-core nodes. This is perhaps an even more viable multi-core design for particle transport simulations. Similar results for Chimaera are illustrated in Figure 5.12. These results once more illustrate the value of the model in examining various system design and configuration questions.

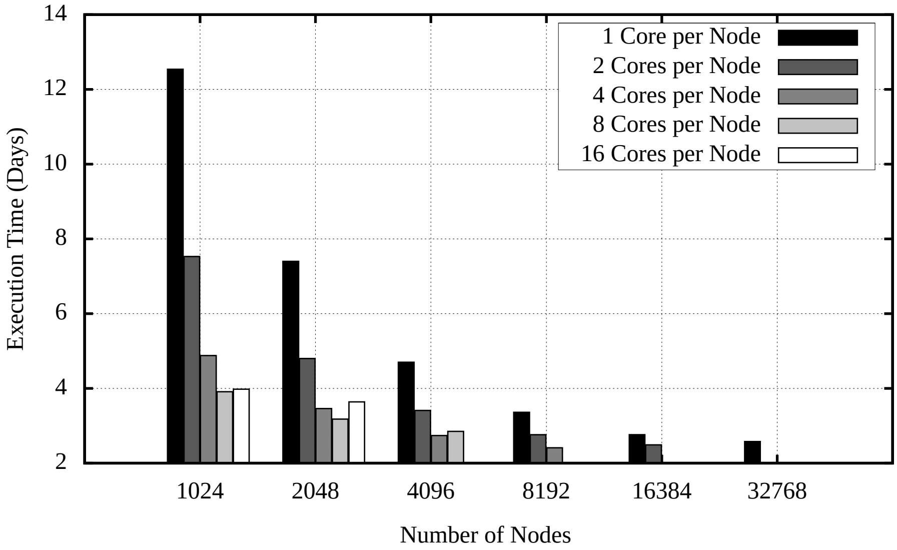


Figure 5.12: Execution time on multi-core nodes (Chimaera 240^3 Cells,10^4 time steps, 16 energy groups 419 iterations, H_{tile} = 2)

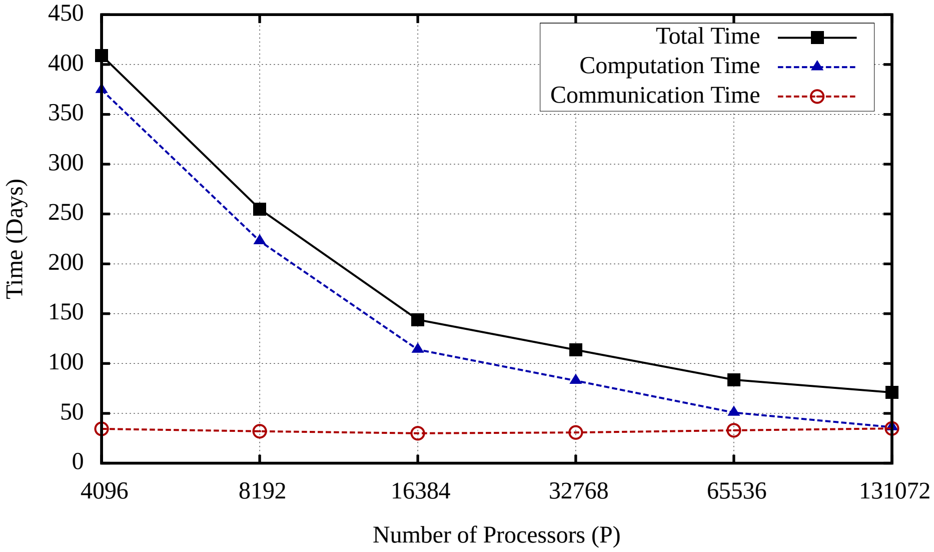


Figure 5.13: Computation and communications cost breakdown (Sweep3D 1 Billion Cells, 10^4 time steps, 30 energy groups, 120 iterations, H_{tile} = 2)

5.4 Application Bottlenecks

In the next set of experiments, we illustrate the use of the model to understand application bottlenecks which are not readily measured when running or simulating the actual code. Figure 5.13 and Figure 5.14 provide the total execution time for the 1 billion cell problem for Sweep3D

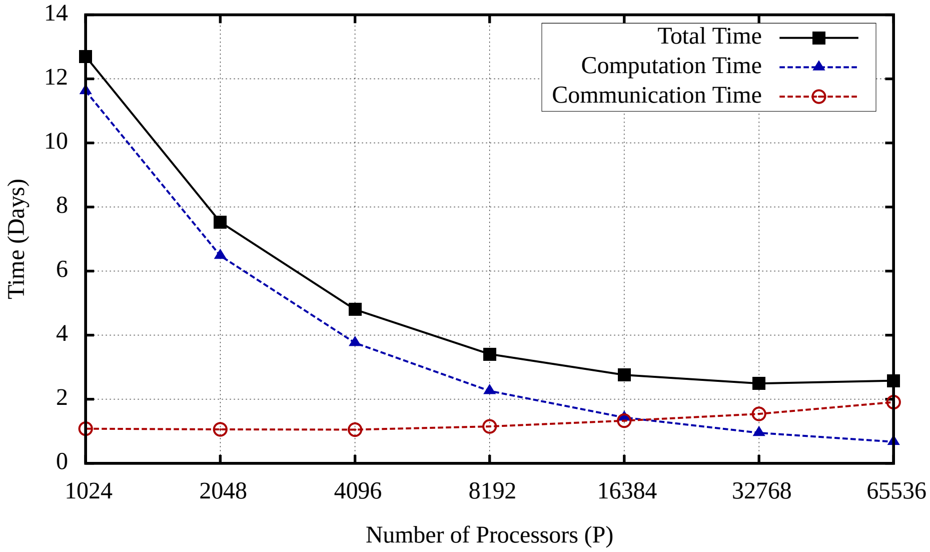


Figure 5.14: Computation and communications cost breakdown (Chimaera 240^3 Cells, 10^4 time steps, 16 energy groups 419 iterations, H_{tile} = 2)

and the 240^3 problem for Chimaera, as well as the breakdown of total critical path time into computation and communication components, as a function of the number of processors on the Cray XT4. The communication component of the total execution time is derived from the $Send$, $Receive$ and $TotalComm$ execution time terms in the model (including the contention during communications). The computation component is the rest of the total execution time. Note that the point at which communication dominates the total execution time is the point at which increasing the number of processors provides greatly diminished reduction in the total execution time. Since communication of the boundary values is required for the simulations, the only opportunity for improving the observed communication bottleneck is to further improve the inter-node communication efficiency. The model can also be used by system architects to project execution times for such communication improvements.

Further insights as to which communication component requires optimising when running wavefront codes can be investigated. For example Figure 5.15 and Figure 5.16 detail the total communication time and contribution of communication bandwidth and latency (i.e. overhead o, and network latency L) as projected by the model for Sweep3D and Chimaera respectively. As can be seen, the majority of the communication time for the Cray XT4 is due to the message start up costs (i.e. processor overhead o) and network latency L, while the contribution of bandwidth to the critical path of execution remains relatively small. Thus, reducing the communication overheads on the XT4 will significantly reduce the communication time spent during a wavefront operation. By contrast an increase in bandwidth would have considerably less effect.

We further examine components that contribute to the total wavefront execution time by considering an alternative time breakdown. In this case we investigate the time spent in pipeline fill and steady state. Recall that considering a single sweep, the pipeline fill time is the time between the first processor starting the sweep and the last processor starting the same sweep. In terms of the model, this is the time contributed by the terms in (4.2.8), (4.2.10) and (4.2.11). The steady state time is the time spent processing the stack of tiles given by the

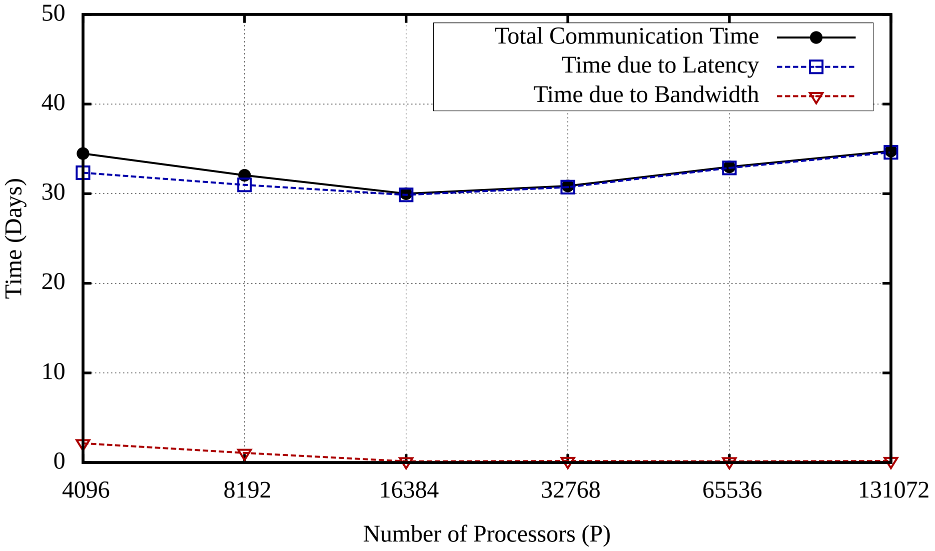


Figure 5.15: Communications cost breakdown (Sweep3D 1 Billion Cells, 10^4 time steps, 30 energy groups, 120 iterations, H_{tile} = 2)

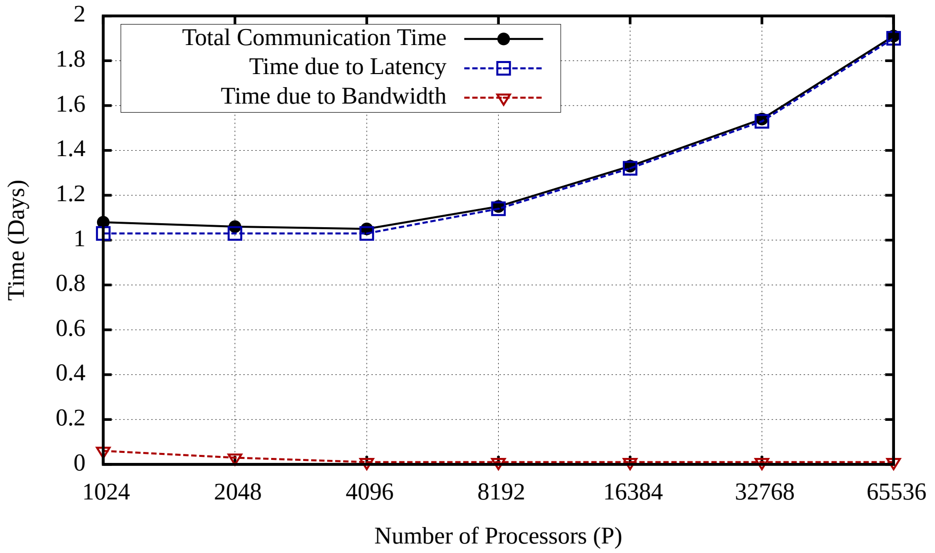


Figure 5.16: Communications cost breakdown (Chimaera 240^3 Cells,10^4 time steps, 16 energy groups 419 iterations, H_{tile} = 2)

expression in (4.2.12). Figure 5.17 and Figure 5.18 give the times spent during pipeline fill and steady state operation of Sweep3D and Chimaera respectively.

Note that in these graphs the problem is solved in strong scaling mode and thus although the pipeline length increases with the increasing number of processors, the computation time taken by a processor per sweep step (as well as the message size) reduces. Alternatively in Figure 5.19 the problem size per processor is kept constant, resulting in an increasing cost for pipeline fill due to the increasing number of processors. The steady state cost remains constant. Thus a possible optimisation could look at methods to reduce the cost due to pipeline fill. Such an optimisation is discussed in the next section.

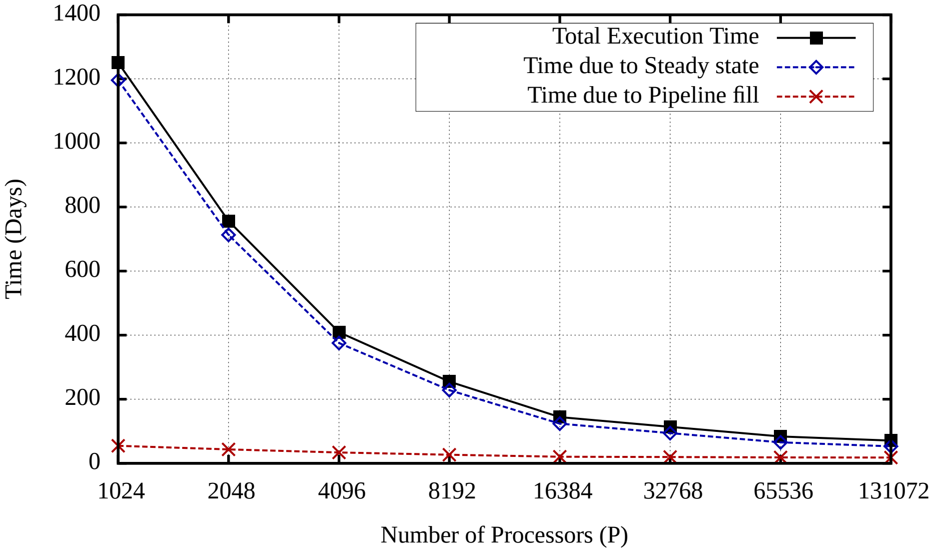


Figure 5.17: Pipeline fill and steady state cost breakdown (Sweep3D 1 Billion Cells, 10^4 time steps, 30 energy groups, 120 iterations, H_{tile} = 2)

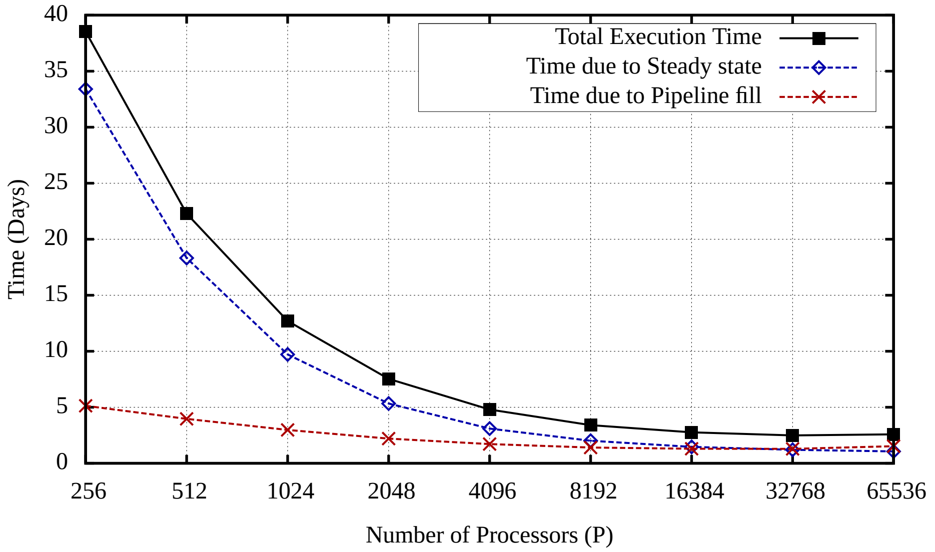


Figure 5.18: Pipeline fill and steady state cost breakdown (Chimaera 240^3 Cells,10^4 time steps, 16 energy groups 419 iterations, H_{tile} = 2)

5.5 Sweep Structure Re-design

The pipeline fill overhead might be reduced by the following Sweep3D re-design. Instead of performing all eight sweeps for the first energy group and iterating to convergence before solving the next energy group, we could pipeline the solution of the energy groups by performing the first two sweeps for all 30 energy groups followed by sweeps 3 and 4 for all 30 energy groups, and so forth. Pipelining the energy groups might require more iterations to reach convergence. We can project the execution time assuming that no additional iterations are needed, by modifying the model input parameters. In other words, we set $n_{sweep} = 240$ so that a total of 240 sweeps are required per iteration, with $n_{diag} = 2$ and $n_{full} = 2$. The

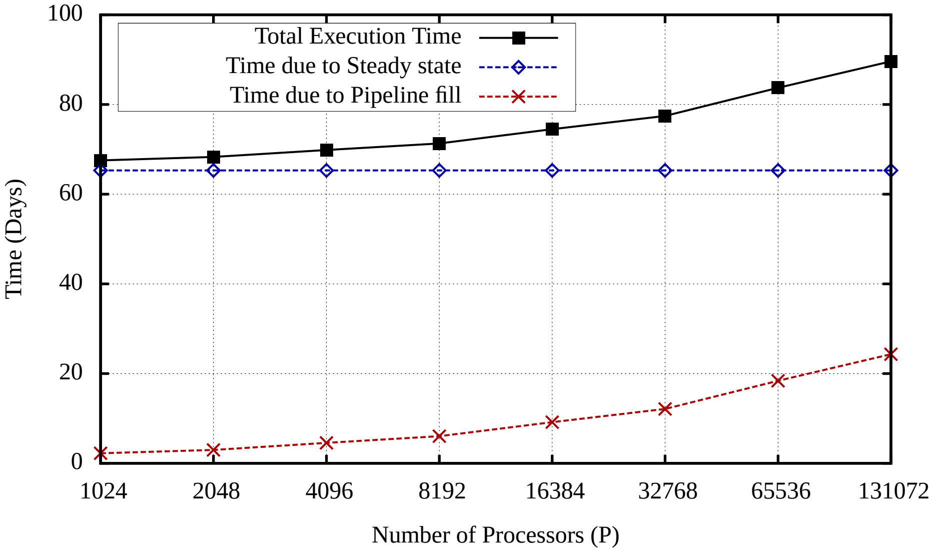


Figure 5.19: Pipeline fill and steady state cost breakdown (Sweep3D $4 \times 4 \times 1000$ Cells per processor, 10^4 time steps, 30 energy groups, 120 iterations, H_{tile} = 2)

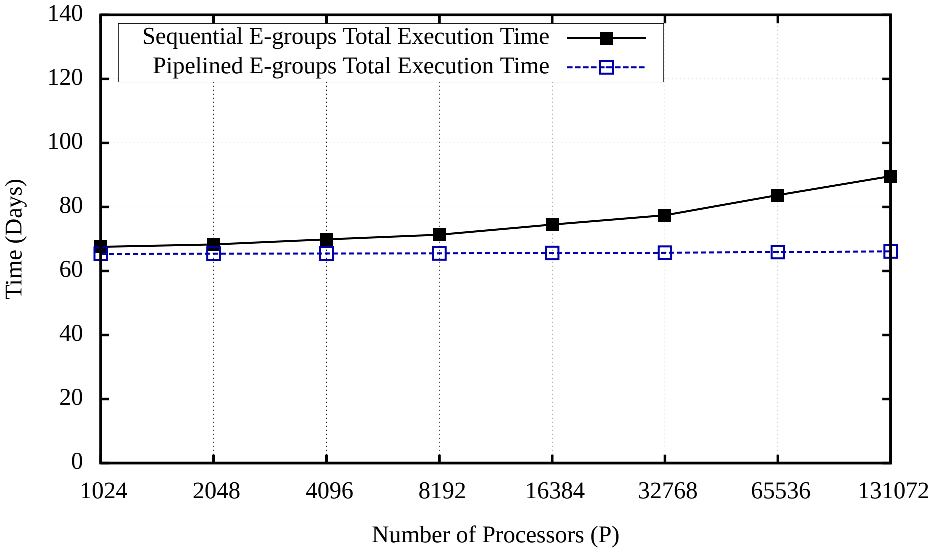


Figure 5.20: Sweep structure redesign - pipelining 30 energy groups (Sweep3D $4 \times 4 \times 1000$ Cells per processor, 10^4 time steps, 30 energy groups, 120 iterations, H_{tile} = 2)

projected execution time with these parameters is also given in the figure, showing that nearly the entire pipeline fill overhead is eliminated. The projections can be made for an increased number of iterations to reach convergence, if the user can provide knowledgeable estimates of this increase. Again these results illustrate how the model can be used to rapidly gain insight into software bottlenecks and the impact of possible software modifications, in order to determine where implementation effort might profitably be placed.

Figure 5.20 shows the speculated gains of this optimisation. The total time for the optimised version of the wavefront code for 30 energy groups are now almost the same as the steady state time for the original code. That is, the sweep structure redesign has managed to eliminate all the pipeline fill times (except for the first one) by pipelining all 30 energy groups.

5.6 Summary

The analysis presented in this chapter demonstrated the utility of the reusable analytic model to provide efficient assessments of software/hardware design decisions and optimisations for running pipelined wavefront applications. The results show that the optimum number of tiles to be solved per sweep step for Sweep3D and Chimaera are in the range of 2 to 5, given the number of angles solved (6 for Sweep3D and 10 for Chimaera) on about 4K to 128K number of processors. Furthermore we see that the number of tiles solved can be tuned to obtain considerable performance benefits if the Z dimension is larger than the other two dimension of the 3D data cube.

We also see that increasing the number of processors gives diminishing returns. The model can be used to decide the optimum number of processors to run a given problem size based on the time to solution and return for investment. An alternative platform sizing criterion based on system throughput was also presented. Additionally we assess the optimum number of cores per node based on the XT4 shared bus node architecture to be 4 cores per node. The results also show that wavefront code performance is dominated by computation time for small number of processors, while on large number of processors communication time begins to dominate the runtime. The model provides quantitative values of the amount of computation and communication that contribute to the critical path. It also predicts that the major contributors to the communication times are the network latency and message processing overhead. A final analysis based on the breakdown in terms of the pipeline fill time and the steady state times, provided a possible optimisation for particle transport codes.

The chapter specifically investigated and projected performance for a system based on the ORNL Cray XT4 system. Further optimisations and analysis of wavefront applications are discussed in Chapter 7 in which we base our findings on an InfiniBand-based commodity cluster system. Before looking at these further optimisations we explore the performance of wavefront codes using an alternate performance prediction methodology, namely simulation, in the next chapter.

6 Wavefront Simulation Models

So far our performance engineering efforts have been mainly based on analytic methods. In this chapter we investigate the performance of pipelined wavefront computations using an alternate performance engineering technique based on simulation. As mentioned in Chapter 2, simulation techniques are considered a significant and complementary alternative to analytic methods. Thus we are motivated to further probe the performance characteristics of wavefront codes using this methodology. More specifically we approach the research detailed in this chapter due to the following key questions and incentives. We are motivated to (1) explore an alternative performance engineering technique applied to wavefront codes, (2) investigate further insights (if any) that can be gained using simulation, (3) explore if the insights from the analytic methods can be used in improving the accuracy of simulation models (and vice-versa) and (4) ascertain whether the analytic model validation results gained are agreeable with the results from simulation.

To explore the above issues, we employ a contemporary simulation methodology based on the layered characterisation of parallel applications introduced in the Performance Analysis and Characterisation Environment (PACE) [12]. The PACE toolset and its subsequent derivations and re-developments (JPACE [160] and WarPP[4, 43, 6]) have been in development over a number of years at the University of Warwick's Department of Computer Science. The layered characterisation techniques used in these experimental simulation systems have been, and continue to be, motivated by developing automation tools for performance engineering HPC applications [4] and supporting application scheduling on large parallel and distributed systems [11].

We begin by a detailed description of the PACE layered characterisation methodology in section 6.1, as a preliminary to illustrate the contrasting process involved in simulation as opposed to analytic modelling. We further examine this process by details of a customised simulation model for the Sweep3D application in section 6.2. Next, in section 6.3 we discuss limitations of the simulation system followed by a key improvement that was motivated by analytic techniques. This enhancement forms the first main contribution from our research using simulation methods. Finally in section 6.4, we show the complementary use of simulation techniques by cross validating our predictions from the reusable analytic model with that of predictions from the simulation models.

6.1 The PACE Discrete Event Simulation System

PACE (Performance Analysis and Characterisation Environment) [12] is an implementation of a layered performance characterisation method which was first introduced in [161]. The layered characterisation aims to encompass all aspects of a system including a software execution graph, the parallelisation strategy and the system's resources and architecture. PACE is largely based on independent application and resource modelling, an overview of which can be found in Figure 6.1. PACE consists of a static source code analyser called '*capp*', which extracts the control flow of the application and the frequency of performance-critical operations (op-codes). *Capp* is used to extract the operation of a serial kernel in terms of C language micro-characterisations (*clc*s). The core of the PACE system is a Performance Specification Language (PSL) named CHIP^{3}S (Characterisation Instrumentation for Performance Prediction of Parallel Systems) and a related compiler. The PSL provides a description of the application and its parallelisation in an intuitive language syntax. The resource modelling is supported by a Hardware Modelling and Configuration Language (HMCL), which provides a description of the computation and communication resource performance of a system. HMCL scripts consist of hardware resource performance values obtained by processor and MPI benchmarks. Once both the application and resource models are created, they can be combined as inputs to the PACE evaluation engine to obtain predictions of execution time within seconds. An important aspect of this process is the ability to reuse the models with different resource or application models.

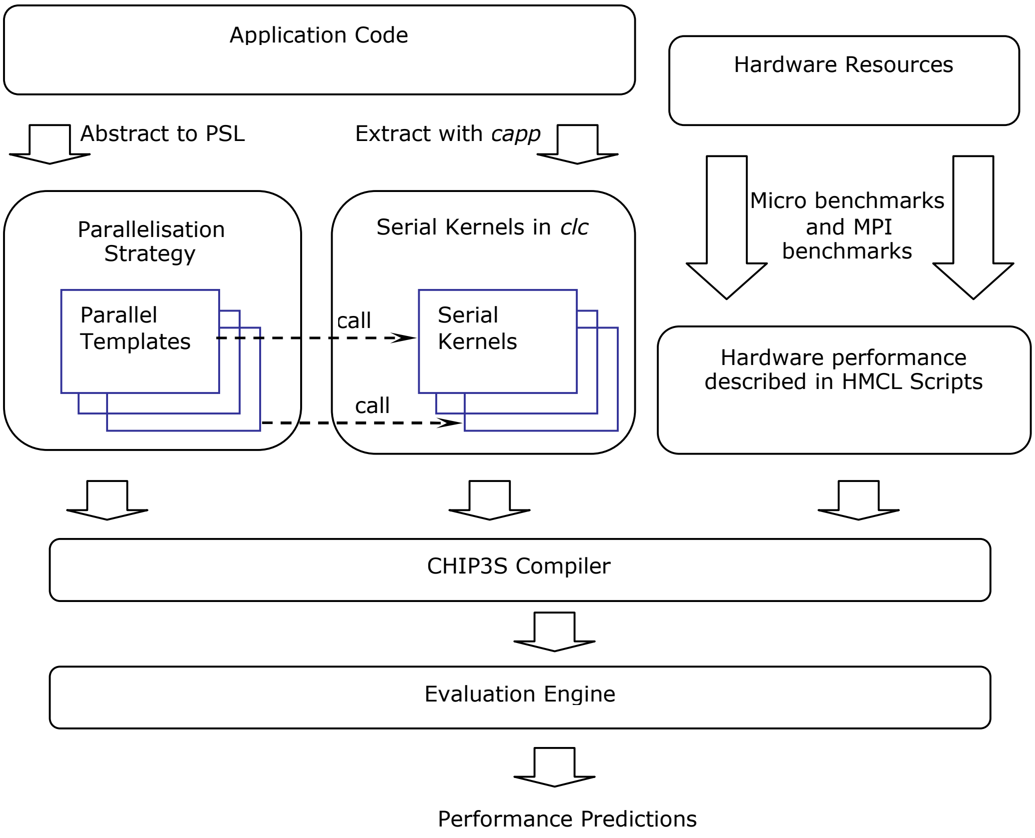


Figure 6.1: Overview of the PACE simulator and toolset

The PACE evaluation engine is a discrete event simulator that executes the compiled models to produce a simulation of events. The evaluation engine tracks the occurrence of

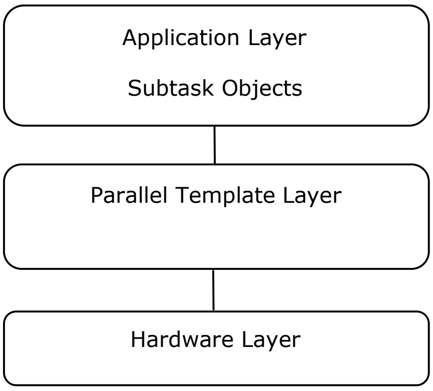


Figure 6.2: Layers in a PACE model

events to produce quantitative estimates of the runtime of the application modelled. Once a model is built, it can be used to investigate speculative analysis of the application such as its scaling behaviour. A PACE model of an application running on a target HPC platform is built by considering a layered or *modular* characterisation of the application, its parallelisation strategy and the hardware resources that execute the code. Thus each layer is modelled separately. Figure 6.2 illustrates this modular hierarchy of a PACE model.

When developing a simulation model of a parallel application for execution on the PACE evaluation engine, the application is characterised at the three main levels of (1) application layer, (2) parallel template layer and (3) hardware layer. The application layer consists of a characterisation of the serial portions of the parallel application code (more specifically called subtask objects). This characterisation is that of a control flow of the computation performed serially in the form of a software execution graph. In other words, these characterisations capture the serial events of the application to be simulated. The parallel template on the other hand describes the parallelisation strategy of the application model. It provides the inter-process messaging and denotes the times at which each subtask object is evaluated in the discrete event simulation. Thus, the parallel template defines the parallel events and their synchronisation. Finally the hardware layer provides a timing characterisation of the hardware resources that are used for execution of the application. Therefore, this layer provides a platform dependent set of timings for computation and communications.

The PACE system illustrated in Figure 6.1 can also be viewed as providing a set of tools to facilitate the development of predictive simulation models. The source code analyser, *capp* takes a C or Fortran program code and performs a static code analysis to generate the control flow (cflow) of serial kernels. As we will show in the next section, this process is largely automated and requires only minor modifications by a performance engineer. The parallelisation strategy is described in the CHIP^{3}S performance specification language. Finally tools and scripts are also provided to extract hardware characterisation models from micro-benchmarks of computation and communication. The next section demonstrates the development of a PACE model for Sweep3D to illustrate the capabilities of the system.

6.2 A PACE Model for Sweep3D

In contrast to the process of developing analytic models, when developing simulation models (such as a PACE model) the process involved aims to simulate the events that occur during the execution of the program code. Thus the program code is directly used to identify (either manually or automatically) the events that are to be simulated. In this section we provide a step by step analysis of developing the PACE simulation model for the Sweep3D application. The model was originally detailed in [162]. Our contributions are discussed in the next section.

The Sweep3D benchmark consists of the following key sections: (1) read input configuration file and perform the domain decomposition (2) perform wavefront sweeps and (3) perform collective operations at the end of each iteration. The runtime reported in the benchmark only considers (2) and (3) of these and as such our simulation model will be directed towards modelling only the subroutines in these sections. The application layer of the model acts as a starting point of developing the simulation model by declaring the subtask objects that consists of the serial portions (or events) of the application in addition to several other CHIP^{3}S specifics that aid the simulation of these sections. Part of the *sweep3d* application layer object's PSL description is detailed in listing 6.1.

Listing 6.1: Application object:sweep3d

```
application sweep3d {

include hardware;
include source;
include sweep;
include fixed;
include flux_err;

var numeric: npe_i = 1,
...............
link {
hardware: Nproc = npe_i * npe_j;
sweep:  it = it,
...............
}
option {hrduse = "IntelP31266"; }
proc exec min
        var x, y;
        {       if (x > y) return y;
                else return x;
        }
        ...............
proc exec init {
        var numeric: i, tmp;
        if (isct == 0) nm=1;
        else if (isct == 1) nm=4;

        it = it_g / npe_i ;
        jt = jt_g / npe_j + 1 ;
        if( mk > kt ) mk = kt;
        ..................
        for (i=1;i<=tmp;i=i+1){
           for(mi=1;mi<=mmi;mi=mi+1){
               ndiag = ndiag+max (min(i,min(jt, min(nk, jt+nk-i))),0);
           }
193             }
194             (*get average of ndiag*)
195             ndiag = ndiag/tmp;
196             for( i=1;i<=-epsi;i=i+1){
197                     call source;
198                     call sweep;
199                     call fixed;
200                     call flux_err;
201             }
202         }
203     }
```

The initial declarations consist of *include* statements, *var* external variable declarations, *link* statements and *options* statements. A complete description of the meaning and application of these statements are given in [12, 163]. The *include* statements declare other objects that are referenced by this object, the *var* declares externally (by user at evaluation time) modifiable variables, the *link* statements enable variables in other referenced objects to be modified by this object and the *options* statements define a set of default values. In this example the default hardware model to be used is set to an Intel Pentium 3 (model 1266) which is detailed in listing 6.4. The *proc exec* statements declare a subroutine or a function. The evaluation of the model starts from the procedure *init* which calls the evaluation of the four subtask objects one after the other for 12 iterations (lines 196 - 200) (depending on the *epsi* convergence variable defined in the input file that details the problem size). It can be seen that procedures directly implement the control flow of the application. Thus, evaluation of the model means that these statements are directly executed (in a similar fashion to a set of C code statements). By coding the control flow of the application, run-time values that decide loop iterations that cannot be determined before run-time are automatically calculated. Such variables include the *ndiag* value (calculated dynamically in lines 189 - 193), which directly determines the per cell work modelled in the *sweep* subtask object. In order to establish the complex relationship that determines the value of *ndiag*, we have used the average value resulting from the actual C code implementation of the application.

Listing 6.1 shows the calls to subtask objects in lines 197 to 200. Subtask objects detail the control flow of the serial computation portions in the application as well as declare the parallel template objects that call them. The structure of the model's subtask objects are similar to the application objects, but additionally contain C language characterisation (*clc*) descriptions which define the control-flow of serial computation. Part of the *sweep* subtask object can be found in listing 6.2. The include file `sweep.x` consists of the *clc* descriptions representing the core computation units of the application. The source code analysis parser *capp* provides an automated procedure to obtain these descriptions. Appendix C contains part of *sweep.x* which is an example of the output from *capp*. It contains the *clc* description named `work()` which is a computation denoted in the sweep subtask object.

Listing 6.2: Subtask object:sweep

```
1     subtask sweep {
2     include hardware;
3     include pipeline;
4
5     var numeric:
6             it = 26,
      ...............
30    link {
31    pipeline:
32            Tx_sweep_init = sweep_init(),
33            Tx_octant = octant(),
      ...............
40            Tx_work = work(),
      ...............
52    }
54    proc exec init {
      ...............
67    }
      ...............
      ...............
69    #include ”sweep.x”
70    }
```

As it can be seen from cflow `work()` in Appendix C, the control flow is described in terms of abstract op-codes that provides the computational cost as determined by a static analysis of the actual code. For example the AILG operation stands for the time taken for an addition of two local integer variables, the result of which is stored as a global variable. Similarly, LFOR accounts for the overhead of a for loop and ARD1 accounts for an array access consisting of double precision floating-point values. A more complete discourse on the various *clc* op-codes are given in [164].

When obtaining a prediction using simulation, unlike control flow statements, the *clc* instructions are not executed, but are accumulated depending on the number of loop counts and branch probabilities to give a time for each serial computation described by the *clc*. A subtask object includes the parallel template used when it is evaluated. In the case of *sweep* in listing 6.2 the related parallel template is *pipeline* (line 31). The branches are assigned a probability score and loops are given an average iteration count that can be calculated from profiles of the execution of the application and data analysis. The procedure *work* represents the bulk of the computation. The point at which it is evaluated from within the *pipeline* parallel template object can be seen at line 260 in listing 6.3.

Listing 6.3: Parallel template object:pipeline

```
1       #include <mpidefs.h>
2       partmp pipeline {
        ..............
168     proc exec init {
169     var numeric: phase,
        ..............
        ..............
```

```
  for( phase = 1; phase <= 8; phase = phase + 1)
  {        ..............
    for( i = 1; i <= mmo; i = i + 1 )
    {  ..............
       for( j = 1; j <= kb; j = j + 1 )
       {  ..............
          for( x = 1; x <= npe_i; x = x + 1 )
          for( y = 1; y <= npe_j; y = y + 1 )
          {   ..............
              ew_rcv = Get_ew_rcv( phase, x, y );
              if( ew_rcv != 0 )
              { step mpirecv { confdev ew_rcv, myid, nib*8; } }
              else { step cpu on myid { confdev Tx_else_ew_rcv; } }
          }
               ..............
          for( x = 1; x <= npe_i; x = x + 1 )
          for( y = 1; y <= npe_j; y = y + 1 )
          {   ..............
              ns_rcv = Get_ns_rcv( phase, x, y );
              ..............
          }

          step cpu {
             confdev Tx_work;
          }

          for( x = 1; x <= npe_i; x = x + 1 )
          for( y = 1; y <= npe_j; y = y + 1 )
          {   ..............
              ew_snd = Get_ew_snd( phase, x, y );
              if( ew_snd != 0 )
              { step mpisend { confdev myid, ew_snd, nib*8; } }
              else { step cpu on myid { confdev Tx_else_ew_snd; } }
          }

          for( x = 1; x <= npe_i; x = x + 1 )
          for( y = 1; y <= npe_j; y = y + 1 )
          {   ..............
              ns_snd = Get_ns_snd( phase, x, y );
              ..............
          }
       }
       ..............
    }
  }
 }
}
```

The pipelined wavefront structure of Sweep3D is modelled in the parallel template layer. The core template implementing this is the *pipeline* parallel template object. The structure of this template has been derived directly from the *sweep* function found in the application; it should be noted therefore, that a level of understanding of the application is required to extract the parallel decomposition. Nevertheless, due to the intuitive syntax of the PSL scripts, this process is straightforward for an engineer with some understanding of the application.

The communication resource usage and the communication pattern is also described in the parallel template layer. The *init* procedure describes the start of the *per octant, per angle block, per k-plane block* loop in lines 194 to 211. For each iteration of this loop, MPI receives are posted (line 222) followed by the per processor work (line 260). Next the MPI sends are executed by sending outbound cell face values (line 269). In addition to the *init* procedure, *pipeline* defines and makes use of several procedures such as *Get_ew_rcv* (line 220). The serial kernel code characterised in the related subtask object are called from the parallel template (e.g. *Tx_work* at line 260, *Tx_else_ew_rcv* at line 223).

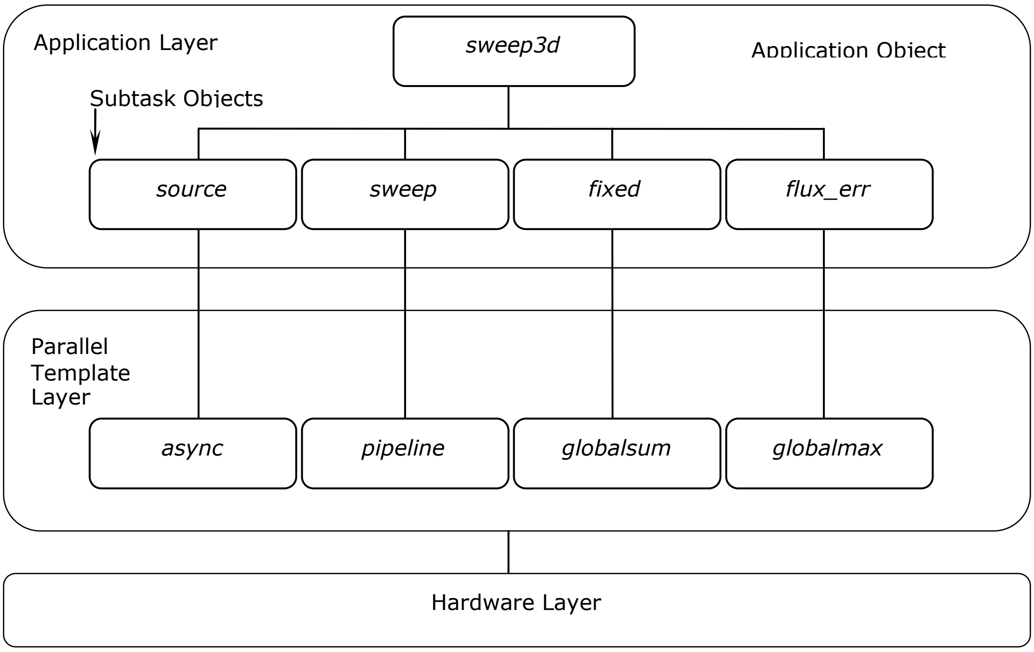


Figure 6.3: Layered objects for PACE Sweep3D model

The full layered object diagram for the simulation model for Sweep3D is detailed in Figure 6.3. It includes the above discussed *sweep3d* application layer object, *sweep* subtask object and *pipeline* parallel template object in addition to the other components (subtask and parallel template layer objects) of the full Sweep3D simulation model. The *globalsum* and *globalmax* parallel template objects implement the parallelisation strategy of MPI allreduce primitives with sum and max operations respectively, while the *async* object implements a sequential template that has no communications. The subtask objects *fixed* and *flux_err* makes use of the allreduce parallelisation strategy implemented in the *globalsum* and *globalmax* parallel templates to model the collective operations at the end of each Sweep3D iteration. The remaining subtask object, *source* implements a fixed computation cost that occurs at the beginning of an iteration.

The final layer of the model, the hardware layer, gives resource usage cost for a target platform. This consists of the computational resource usage for the op-codes that make up the *clc*s in the subtask layer, memory resource usage (not used in this model) such as cache access times [165] and the communication network resource usages as used in the parallel template layer. The hardware layer models will be linked according to the target platform that is being evaluated. Listing 6.4, details the hardware layer resource model for an Intel Pentium 3 2-way SMP cluster with a Myrinet2000 Interconnect. The *clc* section consists of time costs in micro seconds for a set of about 170 op-codes. These timings are obtained by simple synthetic micro benchmarks run on the target processor. The *mpi* section denotes the parameters representing the message passing performance of the systems' interconnect. The parameters A to E describe an equation of the form:

$$Transfer\ time\ of\ x\ bytes = \begin{cases} B + Cx, & \text{for } x \leq A \\ D + Ex, & \text{for } x \geq A \end{cases} \tag{6.2.1}$$

where x is the size of a message in bytes. This is simply a curve fit for a set of data points. There are three sets of A to E parameters in listing 6.4, representing the gradient and intercept for the above equation for MPI send times, MPI receive times and ping-pong times respectively. Parameter A represents a message size where communication characteristics of the interconnect display different gradients. The data points for this regression are obtained using an MPI benchmark program that carries out timed MPI sends, receives and ping-pongs for increasing message sizes. This simple communication resource model has proved to be sufficient for the communication behaviour exhibited by an application such as Sweep3D. This could be attributed to the one way blocking sends and receives that dominate the application. If, on the other hand, a large number of collective communications are to be modelled, then a more detailed communication resource model and benchmark procedure may be required.

Listing 6.4: Hardware model for a Pentium 3 2-way SMP Myrinet2000 cluster

```
config IntelP31266 {
hardware {
   Tclk = 1 / 1266,
   Desc = "Intel P3 1266MHz,2GB",
   Desc = "PC, Intel P4 1266Mhz, 2GB RAM, Linux 2.4.21-37.0.1.ELsmp",
   Source = "CSC IBM cluster";
}
clc {
     SISL = 0.000637512,
     SISG = 0.000636529,
     SILL = 0.000637146,
     ................
     ................
}

mpi {
     DD_COMM_A = 1024,
     DD_COMM_B = 10.7866,
     DD_COMM_C = 0.0158239,
     DD_COMM_D = 41.7131,
     DD_COMM_E = 0.0858768,
          DD_TSEND_A = 23552,
          DD_TSEND_B = 5.3193,
          DD_TSEND_C = 0.00352455,
          DD_TSEND_D = −209.632,
          DD_TSEND_E = 0.0404188,
          DD_TRECV_A = 1024,
          DD_TRECV_B = 9.61369,
          DD_TRECV_C = 0.00175511,
          DD_TRECV_D = 15.8287,
          DD_TRECV_E = 0.00202664;
        }
    }
```

6.3 Enhancing the Predictive Accuracy of PACE for Modern HPC Systems

The Sweep3D simulation model developed in the previous section has several limitations when predicting performance of Sweep3D on modern HPC systems. Firstly we recall that the serial computation model in a PACE model is characterised via static source code analysis. Such a characterisation was observed to underestimate the effect of several important optimisations on modern processors. Particular examples include, compiler optimisations, such as instruction scheduling, out of order or speculative execution, the myriad array of super scalar features of a processor, such as multiple operation pipelines, on-the-fly optimisations, and the effect of highly specialised memory hierarchy. The work in [162], on which the model explained in the previous section is based, relies on a set of opcode benchmarks, which when combined with the tally of opcodes produced by the *capp* source code analyser, allow summative results to be calculated; these results then form the basis for performance predictions. This method was acceptable for processors available at the time. Producing accurate predictions based on this very fine-grained benchmarking relied on the processor executing the code exactly as it was or with very little modification when the *capp* tool did the static source code analysis. This assumption does not hold for modern processor systems and compilers where it under estimates run-time hardware/compiler performance optimisations when the application is actually executed. Predictions based on this approach in some cases, such as on an AMD Opteron 2-way SMP cluster, gave a prediction error as large as 50%.

Secondly, the op-code characterisation of computation means that a simulation is evaluated at a per instruction level, which takes a considerable amount of simulation time. Thus the simulation system at the time did not scale to simulate more than about 64 processors. But as modern HPC systems generally have over 10K (and even over 100K) processors, improving the system to predict application run times on such large systems has become increasingly important.

An alternative approach and the one that is adopted in this research, draws on the insights gained through the analytic model development process, as a solution to the above issues. We use profiling to obtain a coarser level measurement of the achieved performance of the serial source code of the application by using a method similar to the one described in section 4.4. The time for each serial computation block is measured during a run of the code on a small number of processors (the larger of the number of cores/processors per node or four). These times can then be used in conjunction with loop iteration times and conditional statement probabilities (also obtained via runtime monitoring of the code) and static code analysis to develop a control flow that is more representative of the actual execution of the code. We used the PAPI [97] library to monitor and count the operations on serial kernels. Thus we were able to verify that the actual number of floating point operations that are executed by a processor are accurately represented in the subtask object. Listing 6.5 illustrates the modified characterisation of a serial kernel named *work* used in the *sweep* subtask object. The original subtask object is included in Appendix C.

Listing 6.5: Modified clc for the serial computation *work* from subtask object *sweep*

```
168 proc cflow work {
169  loop(<is clc,LFOR>,jt+nk-1+mmi-1){
172   loop (<is clc, LFOR>, ndiag){
174     compute <is clc,2*MFDG>;
176       loop (<is clc,LFOR>,nm-1){
178          loop (<is clc,LFOR>,it){
179           compute <is clc,2*MFDG>;
180         }
182       }
183       case (<is clc,IFBR>){
184         (-ifixups)/(-epsi):
185         loop(<is clc,LFOR>,it{
186            compute <is clc,19*MFDG>;
187         }
188         1-((-ifixups)/(-epsi)):
190         loop (<is clc,LFOR>,it){
191            compute <is clc,19*MFDG>;
193         }
              ...............
              ...............
205        case (<is clc,IFBR>){
206        do_dsa:
             loop (<is clc,LFOR>,it){
208            compute <is clc,6*MFDG>;
209                   }
210               }
212           }
213        }
214    }
```

In order to be compatible with the PACE evaluation engine we express the time per serial computation as an achieved floating-point operation rate while at the same time express the *clc* descriptions in terms of floating-point operations. The mnemonic MFDG represents a floating point operation. Although only the floating-point operations characterise the serial computation, the achieved rate subsumes the times to compute other operations (such as for example for loop overheads, LFOR and conditional branch overheads, IFBR). Due to Sweep3D (as well as many other HPC codes) being double precision floating point intensive, using this operation as a core measure of the time is justified. The hardware level model for a target platform will then not only be specific to that platform but specific to the application (in this case Sweep3D) as well. As we are using the actual application execution (measured on a small number of processors), the need for the use of inaccurate synthetic benchmarks to measure op-code performance on a processor can be eliminated.

Table 6.1: Model Validation Systems

	Pentium 3 Cluster	AMD Opteron Cluster	SGI Altix
Processor	Intel P3 1.4GHz	Opteron 2GHz	Intel Itanium-2 1.6 GHz
Processors/Node	2	2	56
Memory/Node	2GB	2GB	112GB
Interconnect	Myrinet 2000	Gigabit Ethernet	SGI NUMA link-4
Operating System	RedHat Linux 7.2	RedHat Linux (kernel 2.6)	RedHat Enterprise Linux AS 3.0
Achieved Floating-point operation rate	110 MFLOPS	350 MFLOPS	225 MFLOPS
Compilers used	GNU C compiler 2.96	GNU C compiler 3.4.4	Intel C compiler 8.1
Total Processors	128	32	56

We validate the enhanced simulation model on three representative HPC systems. The three clusters used here are chosen so as to validate the model for a variety of representative architectures[1]. This includes an Intel Pentium 3 cluster validation of the Sweep3D model (Table 6.2) running on a cluster of commodity processors comprising of a traditional x86 Intel architecture with a Myrinet 2000 interconnect. An AMD Opteron cluster validation (Table 6.3) investigates the performance on the x86_64 AMD architecture interconnected by a Gigabit Ethernet network. Both of these systems are SMP clusters consisting of 2 processors per SMP node. Finally the SGI Altix system allows the exploration of performance on a genuinely shared memory system with up to 56 processors comprising of Intel Itanium2 (IA-64) processors (Table 6.4). The key system specifications of these three platforms are detailed in Table 6.1. For each case the problem size consists of 50^3 cells per processor with weak scalability. The k-blocking factor (*mk*) is kept constant at a value of 10.

Table 6.2: Sweep3D simulation model validations on an Intel Pentium-3 2-way SMP cluster with a Myrinet 2000 interconnect

Data Size	Num. of PEs	2D Proc. Array	Execution (sec)	PACE Prediction (sec)	Error (%)
100x100x50	4	2x2	26.54	28.59	-7.72
200x200x50	16	4x4	32.28	32.78	-1.55
200x400x50	32	4x8	35.89	38.09	-6.13
400x400x50	64	8x8	40.03	40.75	-1.8
400x500x50	80	8x10	43.09	43.4	-0.73
450x500x50	90	9x10	43.7	44.07	-0.85
500x500x50	100	10x10	44.37	44.73	-0.81
400x700x50	112	8x14	46.32	48.71	-5.16

As can be seen from the validations, the accuracy of the predictions is over 90% on the above systems. Furthermore, run times of the application on large processor configurations can now be speculated in tractable time [3]. Note that the model errors are positive for the Altix validations due to using an estimated achieved floating-point operation rate. At the time

[1] The Cray XT3/XT4 used in the previous chapter was not available for access during the time of this research.

Table 6.3: Sweep3D simulation model validations on an AMD Opteron 2-way SMP cluster interconnected by a Gigabit Ethernet

Data Size	Num. of PEs	2D Proc. Array	Execution (sec)	PACE Prediction (sec)	Error (%)
100x100x50	4	2x2	8.98	9.69	-7.9
150x150x50	9	3x3	9.94	10.54	-6
150x200x50	12	3x4	10.57	11.07	-4.7
200x200x50	16	4x4	10.77	11.33	-5.22
200x250x50	20	4x5	11.18	11.85	-5.97
200x300x50	24	4x6	11.95	12.38	-3.59
250x250x50	25	5x5	11.73	12.11	-3.24
250x300x50	30	5x6	12.07	12.64	-4.68

Table 6.4: Sweep3D simulation model validations on an SGI Altix Intel Itanium-2 56-way SMP

Data Size	Num. of PEs	2D Proc. Array	Execution (sec)	PACE Prediction (sec)	Error (%)
100x100x50	4	2x2	14.66	13.95	4.81
200x200x50	16	4x4	17.31	15.91	8.09
200x250x50	20	4x5	17.57	16.55	5.82
200x300x50	24	4x6	18.29	17.2	5.98
200x400x50	32	4x8	19.83	18.48	6.79
300x400x50	48	6x8	20.54	19.19	6.57
350x350x50	49	7x7	19.95	18.81	5.71
250x500x50	50	5x10	21.56	20.1	6.76
350x400x50	56	7x8	21.04	19.46	7.51

of this validation PAPI was not available on the SGI Altix system to accurately ascertain the number of floating point operations executed by the Intel Itanium-2 processors.

Contrasting the process of obtaining prediction through simulation to that of the use of the reusable analytic model, shows that the main potential advantage of a simulation system such as PACE is its promise of automation. Particularly, the high effort and time for development of predictive models may be reduced with the use of tools such as *capp* and the use of a PSL to directly extract simulation events. But the PACE tools (and the subsequent WarPP tool kit described in the next section) as well as other contemporary simulation tools are yet to demonstrate significant support for fully automated performance model generation that can be used on a wide range of complex parallel applications. For example, developing the parallel template for the Sweep3D application required a level of understanding of the application by a performance engineer.

We find that the depth of insights gained are specifically directed towards building a PACE simulation model and obtaining a runtime prediction value, as opposed to gaining a concrete understanding of the complexities of a parallel application. On the other hand, we feel that a deeper more exact level of understanding comes naturally when developing analytic models. Thus we argue that the advantage of a simulation system such as PACE should be viewed as a support tool for initial evaluation of an unknown parallel application and not as a technique to obtain the full picture in one attempt. For sure as mentioned in Chapter 2, simulation techniques may be built at a greater level of detail, but in the context of predictive models for applications such as Sweep3D, such details may obscure the important performance aspects of the application. Nevertheless, an accurate simulation model provides an excellent cross validation technique. This second advantage is explored in the next section.

6.4 The WarPP Simulation Toolkit

The WarPP simulation toolkit [4, 43] draws on the insights from PACE and attempts to deliver tools and a simulation systems that (1) reduces the time required to construct an initial performance model and (2) further improves predictive accuracy and simulator performance for predictions for large processor configurations. More specifically it is aimed at supporting performance engineering studies of applications executing on massively parallel processor machines (MPPs) which may contain multi-core, multi-processor nodes each having complex performance properties. In this final section we briefly discuss the WarPP toolkit and present cross-validations of the analytic model predictions with that of simulation and actual runtime measures for the Chimaera particle transport application running on a Intel-InfiniBand HPC cluster. The underlying objective is to present the agreeability of the results from the analytic model when compared to that of the simulation results.

The WarPP tools provide a C-like scripting language (a Performance Specification Language - PSL) that can be used to develop a simulation model by hand (user), by automated code analysis or by automated generation of trace-based profiles. The simulation system is a discrete event simulator similar to PACE but provides several significant developments. Firstly, its smallest unit of computation granularity is represented by the execution of an entire "basic block" drawing from the research insights detailed in the previous section.

Secondly, the simulator has the ability to incorporate hybrid models into one single simulation. For example, parts of the model can be made by the use of traditional PSL like statements or analytic sub-models or trace/profiler-based models.

Thirdly, WarPP supports multi-layer network models. A network in an MPP system could be made of a high bandwidth, low latency core-to-core bus within a single processor, interprocessor communications networks found within multi-processor nodes and node-to-node interconnects such as Gigabit Ethernet, InfiniBand and the Cray SeaStar. Each sub-network is described in simulation as a set of latency, bandwidth pairs or a "profile" mapped to a range of the message size space, with the topology of the system being encoded by an assignment of MPI rank pairs for each profile. Finally, recent tools being developed as part of the WarPP project attempt to provide a system for rapid initial model construction and parameterisation in order to reduce the effort of the performance engineers [7]. One approach examined is trace-based simulation model developments. More details of the WarPP tools can be found in [43].

Table 6.6 and Table 6.7 detail validations of the reusable analytic model against predictions from the WarPP simulation model and actual run times from the system for the Chimaera 240^3 problem. The target HPC system in this case is an Intel Xeon cluster interconnected by an Infiniband interconnect at the Warwick Centre for Scientific Computing (hereafter referred to as CSC-Francesca). The key system specifications are noted in Table 6.5.

Table 6.5: Intel InfiniBand (CSC-Francesca) Cluster - Key Specifications

Processor	Dual-Intel Xeon 5160
	3 GHz
Cores/Processor	2
Processors/Node	2
Memory/Node	8GB
Interconnect	QLogic InfiniBand 4X, SDR
Operating System	SUSE Linux Enterprise Server 10
Achieved Floating-point operation rate	110 MFLOPS
Compilers used	Intel C/Fortran Compiler Suite 10
Total Cores	960

Table 6.6: Chimaera Model Validation on a Intel Xeon-InfiniBand cluster - 120^3 total problem size

NPE	Nx/n	Ny/m	Analytic Prediction (Sec)	Execution (Sec)	WARPP Simulation Predictions (Sec)	Analytic Error (%)	Simulation Error (%)
32	15	30	88.9	107.18	89.58	-17.05	-16.42
64	15	15	47.25	56.72	48.75	-16.69	-14.05
96	7.5	20	35.43	40.89	33.8	-13.36	-17.34
128	7.5	15	30.28	32.56	28.98	-7.01	-11

Table 6.7: Chimaera Model Validation on a Intel Xeon-InfiniBand luster - 240^3 total problem size

NPE	Nx/n	Ny/m	Analytic Prediction (Sec)	Execution (Sec)	WARPP Simulation Predictions (Sec)	Analytic Error (%)	Simulation Error (%)
81	27	27	324.35	342.33	330.46	-5.25	-3.47
96	15	40	268.57	297.03	277.56	-9.58	-6.55
100	24	24	259.37	278.37	248.32	-6.82	-10.79
128	15	30	205.74	225.65	207.18	-8.82	-8.19
169	19	19	167.83	174.35	177.09	-3.74	1.57
256	15	15	108.81	129.65	117.98	-16.08	-9
512	8	15	63.08		66.64		
1024	8	8	37.99		37.29		
2048	4	8	23.27		22.49		
4096	4	4	15.6		15.59		
8192	2	4	11.86		11.57		
16384	2	2	10.15		10.11		

The network models used in these validations are based on simple linear equations parametrised by message size similar to the models developed for the Cray XT3/XT4 systems in the previous chapters. For simplicity we use a latency/bandwidth model for this network without loosing predictive accuracy. The communication time for a message of length x bytes can be modelled as $t_{send}(x) = (1/B)x + n_l$ with the bandwidth (B) and latency (n_l) associated with the appropriate region for x. The time for a receive is modelled by: $t_{recv}(x) = (1/B)x$ since the receiver does not experience the latency required to establish the connection but must spend at least the actual transmission time in a locked state accepting data from the network interconnect. Note the difference between the definitions of n_l and the LogGP parameter L, where the former is a combination of both the message processing overhead (o) and network latency (L) based on the MPI messaging protocol. The separation of n_l in terms of o and L was not possible with the communications profiles of CSC-Francesca. However, predictive accuracy remains at a higher level, even with such an abstract communications model. Table 6.8 details the network parameters used in the validations. Note that the network benchmarking is partitioned into two regions by message size. The point at which the split in network performance occurs is 2048 bytes, indicating that the InfiniBand management system may be configured for a maximum transmission unit (MTU) size of 2Kbytes.

Table 6.8: InfiniBand network model parameters

Network Profile	Message Size (Bytes)	n_l (μSec)	B (GBytes/s)
on-chip (core to core)	≥ 0	0.655	2.70
off-processor (processor to processor)	< 2048 ≥ 2048	0.69 0.91	2.80 3.83
off-node (node to node)	< 2048 ≥ 2048	2.64 3.63	0.46 0.73

We have not observed contention on the CMP nodes of CSC-Francesca machine that is similar to the Cray XT3/XT4. Thus we have ignored these costs (if any) in our predictions. As can be seen from Table 6.6 and Table 6.7, there is good agreement between the analytic model and simulation model predictions. This further increases our confidence in the accuracy of the models. Additionally, the results show that the model is equally accurate for systems other than the Cray XT3/XT4 on which the reusable analytic model was initially developed.

6.5 Summary

This chapter has described the research work that used simulation to investigate the performance properties of wavefront applications. Developing predictive simulation models was presented in contrast to the analytic model analysis of wavefront codes given in the previous chapters. The PACE simulation system was used to model the Sweep3D application. An enhancement to the model development was also presented where coarser grained computation timings are used to characterise computation performance of serial kernels. This has shown to give higher predictive accuracy and scalability when modelling applications for modern HPC systems. This key improvement was subsequently implemented in the WarPP toolkit enabling the system to model over 100K processors with less than 20% predictive errors in tractable time.

We also demonstrated the high level of agreement between the analytic and simulation predictions for wavefront codes (specifically using the Chimaera application on a modern HPC system). The strong correlation between the analytic and simulation results means that when a real system is not available for validation the simulation results can be equally well used as a form of validation of our analytic models (or vice-versa). This enables us to cross compare the results from the analytic model for additional confidence. As a result, we use WarPP as an alternate validation for several optimisations developed in the next chapter.

7 Optimisations and System Procurement

7.1 Introduction

In this final contribution chapter we return to the use of the reusable analytic models to address one of the key open questions that motivated this research. The objective is to investigate the bottlenecks that affect the operation and the possible optimisations for the parallel pipelined wavefront algorithm in terms of computation, communication and synchronisation behaviour. Recall that in Chapter 5 section 5.5, one possible optimisation based on the particle transport codes (Sweep3D and Chimaera) was presented. In this chapter we explore further the utility of the analytic models in understanding such possibilities for near optimal code design. More specifically, we show how the models provide significant insight to: (1) identify the qualitative and quantitative benefits of several key possible optimisations, including an analysis for wavefronts operating on irregular/unstructured data grids as well as heterogeneous systems; (2) bottleneck analysis of the algorithm in support of system procurement. The results from the WarPP simulator are used for validations of several optimisations and provide insights when an actual system is not available for executing the code. The target HPC system in these optimisations is the Intel Xeon/InfiniBand cluster at the Warwick Centre for Scientific Computing (CSC-Francesca), unless otherwise stated. For ease of reference, the analytic model from Chapter 4 is reproduced below.

Table 4.2 Plug-and-play LogGP Model: One Core Per Node, on 3D Data Grids

$W_{pre} = W_{g,pre} \times H_{tile} \times N_x/n \times N_y/m$	4.2.6
$W = W_g \times H_{tile} \times N_x/n \times N_y/m$	4.2.7
$StartP_{1,1} = W_{pre}$	4.2.8
$StartP_{i,j} = max(StartP_{i-1,j} + W_{i-1,j} + Total\_Comm_E + Receive_N,$ $StartP_{i,j-1} + W_{i,j-1} + Send_E + Total\_Comm_S)$	4.2.9
$T_{diagfill} = StartP_{1,m}$	4.2.10
$T_{fullfill} = StartP_{n,m}$	4.2.11
$T_{stack} = (Receive_W + Receive_N + W + Send_E + Send_S + W_{pre})N_z/H_{tile} - W_{pre}$	4.2.12
$Time\ per\ iteration = n_{diag}T_{diagfill} + n_{full}T_{fullfill} + n_{sweeps}T_{stack} + T_{nonwavefront}$	4.2.13

7.2 Shifting Computation Costs

We begin by investigating an optimisation inspired by a key difference between NPB-LU and the two particle transport codes (Sweep3D and Chimaera) that was used in the development of the reusable analytic model in previous chapters. Recall from listings 3.8, 3.9 and 3.10 one of the main contrasting features of LU was that the computation time per sweep step consists

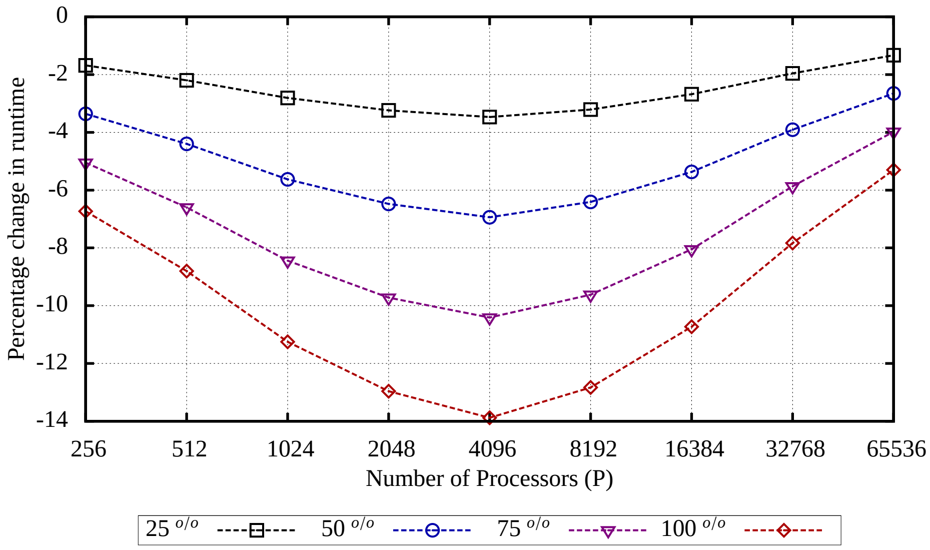


Figure 7.1: Optimisation by shifting computation costs to pre-computation - strong scaling (Speculative Chimaera type application, 240x240x240 Cells, 1 time step, 16 energy groups 419 iterations, H_{tile} = 1)

of a pre-computation (i.e. a computation before the MPI receives) and a main computation. The analytic model accounts for these by modelling them in W_{pre} and W respectively. In this section we investigate the quantitative and qualitative affects of completing part of the main computation during the pre-computation block. More specifically, in an application such as Chimaera where there is no pre-computation, we speculate as to the affects of completing parts of the main computation in an artificial pre-computation block. However, the amount that can be shifted to the pre-computation block depends on the underlying mathematics solved, but in this optimisation we are interested in the speculative case where there are no such limitations.

The pre-computation occurs before receives are posted and therefore does not require any boundary values from near neighbour processors. This in turn means that all processors can compute the pre-computation simultaneously. Thus we investigate how much savings (if any) can be gained by re-structuring the computation in the pipelined wavefront code. Predictions for this optimisation can be easily obtained without any extensions to the reusable analytic model. If we consider the computation per sweep step (W) in a Chimaera-type wavefront application, then we simply multiply the W in (4.2.9) and (4.2.12) by a factor of α where $0 \leq \alpha \leq 1$, while replacing W_{pre} in (4.2.8) and (4.2.12) by $(1 - \alpha)W$ to obtain predictions.

For instance Figure 7.1 presents the model predictions for the case with $\alpha = 0.25, 0.5, 0.75, 1$, representing a 25, 50, 75 and 100 percent shift of computation on to the pre-computation block when using strong scaling. It should be noted that shifting 100% of the computation is presented as a boundary case where in reality such an instance may not be viable. In strong scaling, the number of cells computed by a processor reduces as the number of processors increases. This has the effect of reducing the benefits of the optimisation as predicted by (4.2.9). But as the length of the pipeline increases the time spent in pipeline fill increases with the number of processors. The effect of these contradicting benefits results in a run time reduction of approximately 13% on 2K to 8K processors. From Figure 7.2 we see the contribution of the optimisation clearly increasing as the number of processors increases

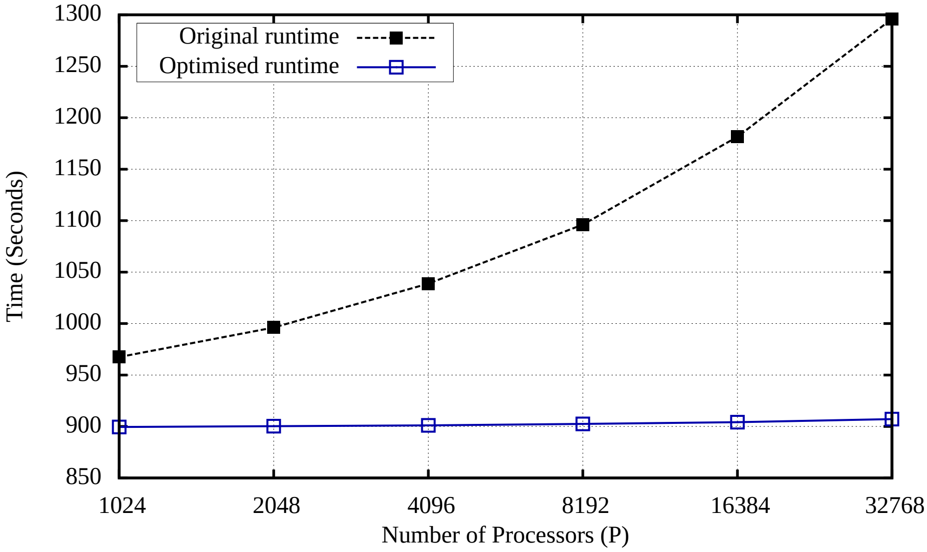


Figure 7.2: Optimisation by shifting 100% of computation costs to pre-computation - weak scaling (Speculative Chimaera type application, 8x8x1000 Cells/PE, 1 time step, 16 energy groups 419 iterations, H_{tile} = 1)

in the case of weak scaling. In this case the length of the pipeline (given by the number of x-y diagonals) increases with the number of processors, while the computation time per sweep step remains constant.

Thus, we can conclude that when developing wavefront codes, it is significantly more beneficial to structure the computation such that as much of the computation as possible is performed during the pre-computation block.

7.3 Multiple Simultaneous Sweeps

The wavefront applications investigated so far have all had a specific ordering in which the multiple sweeps were executed. LU's two sweeps occur one after the other as a forward and a backward sweep without any overlapping. Sweep3D and Chimaera perform eight sweeps with two sweeps overlapping up to a maximum of half of the pipeline fill time (as detailed in chapter 3). We have not yet explored the case when the wavefront sweeps are fully overlapped. More specifically, this entails performing all 8 sweeps beginning at the same time at their respective corners of the 3D data grid and sweeping across to the opposite corner. In this section we extend the reusable model to predict the performance of such an application.

The motivation for this optimisation is again to reduce the pipeline fill times. When executed simultaneously, the eight sweeps are expected to only have a single pipeline fill time. We investigate this optimisation in the following two forms for a typical modern HPC system consisting of CMP nodes:

1. Each sweep is computed by separate processor cores simultaneously.

2. All cores compute all the sweeps simultaneously.

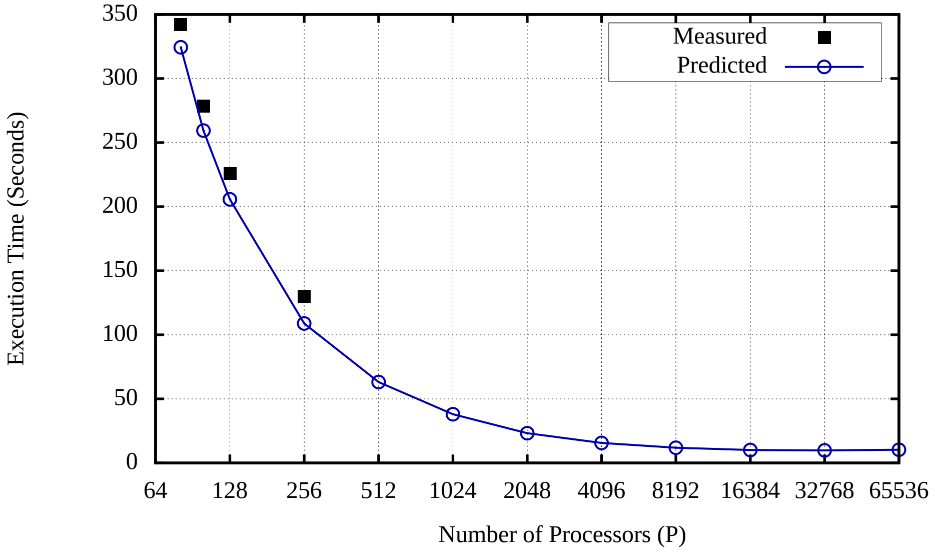


Figure 7.3: Chimaera Model Validation on a Intel Xeon-InfiniBand cluster - 240^3 total problem size, H_{tile} = 1)

7.3.1 Multiple Simultaneous Sweeps on Separate Cores

Modern HPC systems consist of nodes with multiple processing units. Nodes with multicore processors are part and parcel of such systems. Thus the availability of high processor counts may enable us to assign separate sweeps to separate processing elements on a node. For example in a Chimaera type application the runtime flat-lines after about 4K processors (see Figure 7.3). Thus in a system with say 16K processor cores, assigning one fourth of them to compute only two sweeps may provide a better trade-off in performance if the sweeps are computed on 4K processors using simultaneous sweeps. There is only a minor degradation for the computation performance per tile, but due to simultaneous sweeps the pipeline fill time is reduced.

Consider the case where a total problem size of 240^3 is solved for a Chimaera type application with the eight sweeps computing simultaneously, beginning from the four corners of the 2D processor array. Assuming that a node consists of a single quad-core processor we assign two sweeps to be computed by a single core on each node. Figure 7.4 depicts such a distribution of work. In this case the time to solution can be predicted simply by considering the time to compute a single sweep from one corner of the processor array to the opposite corner. If we assume that the time to compute a block of cells of height H_{tile} for a single sweep on a single processor is W, due to a core processing two sweeps we model the computation time per sweep step to be double the computation time for a single sweep step. Similarly, the message size was also doubled to account for larger messages being sent per sweep step. We compare the predictions with those of the original Chimaera runtime predictions in Figure 7.5.

The results show that the benefits of performing multiple wavefronts increase for large processor counts. This is directly attributable to the savings from longer pipeline fill lengths. It should be noted that any contention due to multiple wavefronts crossing node boundaries have been ignored in the model. In a system such as the Cray XT4 used in the previous chapters, we anticipate communication contention on the node due to sharing communication re-

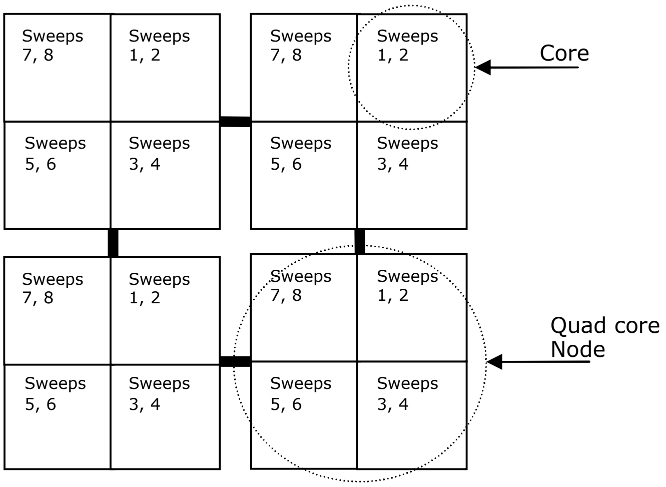


Figure 7.4: Multiple simultaneous sweeps on separate cores

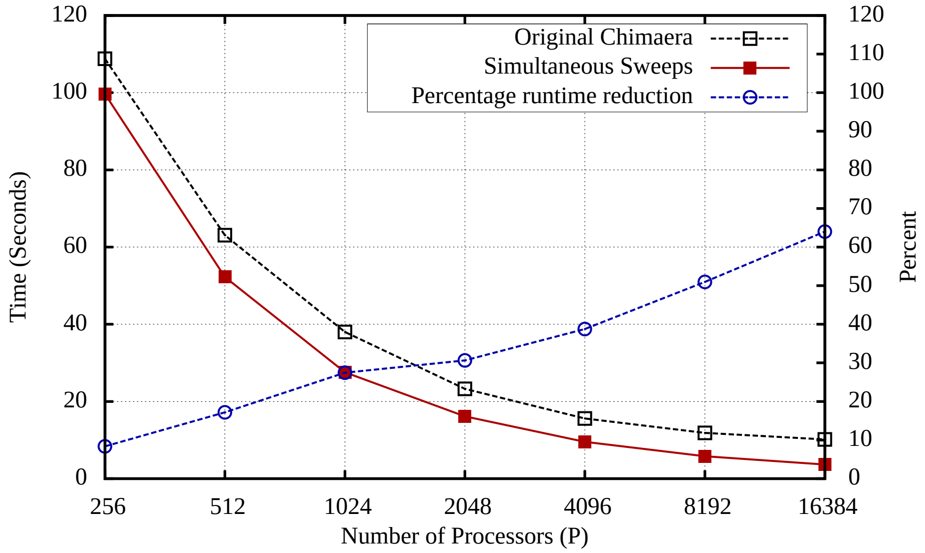


Figure 7.5: Simultaneous Sweeps on Separate cores (Speculative Chimaera type application, 240x240x240 Cells, 1 time step, 16 energy groups 419 iterations, H_{tile} = 1)

sources such as the single bus to RAM and the NIC. Although a speculative analysis of such a case can be given for the XT4 with the CMP communication models developed in Chapter 4, a concrete application that uses multiple simultaneous sweeps will be required to further explore this issue.

7.3.2 Multiple Simultaneous Sweeps on All Cores

In the second case, we assign all 8 sweeps to be simultaneously processed, beginning at each corner of the 2D processor grid, by all the processor cores. In this case, during the operation

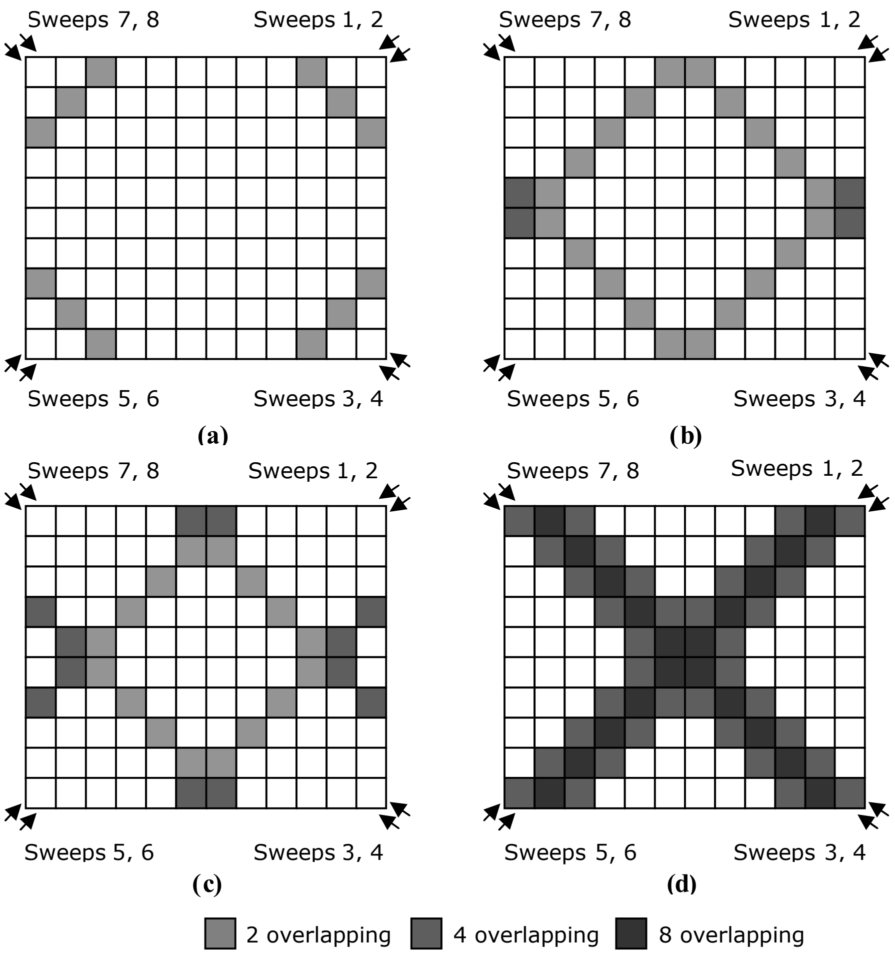


Figure 7.6: Simultaneous multiple wavefronts overlapping steps

of multiple sweeps at any time, a processor may compute a minimum of 2 sweeps up to a maximum of 8 sweeps. This is illustrated in Figure 7.6, where based on the progression of the first wavefront of each sweep there are (a) 2 sweeps, (b) and (c) 4 sweeps, or (d) 8 sweeps being computed by some processor (i, j).

If we assume that the time to compute a block of cells of height H_{tile} for a single sweep on a single processor is W and the time to compute η simultaneous sweeps on a single processor is ηW, then the critical path time to complete all 8 sweeps depends on the steps during which each sweep starts to overlap with any other sweep. We assume here that the communications primitives used are non-blocking MPI sends. Therefore, the extensions required to model multiple sweeps will at least require finding the steps during which η varies from 2, 4 to 8.

Consider a 3D data grid decomposed on to a 2D processor array of size $m \times n$. At the beginning of a sweep each processor will be computing 2 sweeps (one originating from the top of the 3D cube and one from the bottom). Assuming $n > m$, after $m/2$ steps, sweeps 1, 2 and 3, 4 as well as sweeps 5, 6 and 7, 8 overlap. If $m > n$ then sweeps 1, 2 and 5, 6 as well as 3, 4 and 7, 8 overlap first. But in both cases this will amount to a maximum of four sweep overlaps. Finally, as in Figure 7.6 (c), after $(n+m)/2$ steps, all sweeps overlap. As the overlappings occur during the initial pipeline fill stages of the wavefront operation we increase the computation time per sweep step from $2W$ to $4W$ and $8W$ after $m/2$ and $(n+m)/2$ respectively, as in (7.3.1) and (7.3.2).

$$StartP_{1,1} = \eta W_{pre} \tag{7.3.1}$$

$$\begin{aligned} StartP_{i,j} = max(&StartP_{i-1,j} + \eta(W_{i-1,j} + \frac{1}{2}(Total\_Comm_E + Receive_N)), \\ &StartP_{i,j-1} + \eta(W_{i,j-1} + \frac{1}{2}(Send_E + Total\_Comm_S))) \end{aligned} \tag{7.3.2}$$

$$\eta = \begin{cases} 2, & \text{default} \\ 4, & \text{if } m/2 < i+j \text{ or } n/2 < i+j \\ 8, & \text{if } i+j \geq (m+n)/2 \end{cases}$$

As there are 2 sweeps originating from one corner of the 2D processor array, we again assume that the communication operations pack boundary values for both these sweeps into one MPI message with double the message length. After the maximum overlapping is achieved, wavefronts complete each step with $\eta = 8$ in a steady state operation until the pipeline empty stages. During the pipeline empty stages, the reverse of pipeline fill occurs, where the overlapping number of sweeps per processor reduces from 8 to 4 and finally to 2.

$$\begin{aligned} T_{stack} = \eta(\frac{1}{2}(Receive_W + Receive_N) + W + \frac{1}{2}(Send_E + Send_S) + W_{pre}) \\ (N_z/H_{tile} - (m+n-1)) - W_{pre} \end{aligned} \tag{7.3.3}$$

$$Time\ per\ iteration = 2n_{full}T_{fullfill} + n_{sweeps}T_{stack} + T_{nonwavefront} \tag{7.3.4}$$

The time for a pipeline empty is equivalent to the time for a pipeline fill. We use the multiplier 2 in the first term of (7.3.4) to accounts for this. The number of steps $(m+n-1)$ for pipeline empty is subtracted in (7.3.3). The communication terms in the above equations are also multiplied by η, to account for the worst case sequentialising of communications when multiple wavefronts are processed. Although we assume MPI non-blocking primitives, we believe contention on the NICs may have the effect of serialising messages. If the number of tiles in the z dimension is less than the number of pipeline fill stages given by $N_z/H_{tile} - (m+n-1)$ then T_{stack} reduces to 0. This is due to the fact that in such a setup the maximum overlapping of all sweeps only occurs briefly during the pipeline fill stages, and that cost is included in (7.3.2). It is also important to note that such a configuration is not efficient, as many processors will remain idle during the pipeline fill and empty stages.

Previous related work on models for multiple wavefront sweeps has been published in [24] and [37]. The former uses the maximum number of multiple wavefronts (i.e. 8) crossing the boundaries of a processor and neglects the dynamic overlapping during pipeline fill and empty. The latter documents the use of multiple simultaneous sweeps in the solution of an unstructured grid particle transport application without specifics of the model extensions. Thus

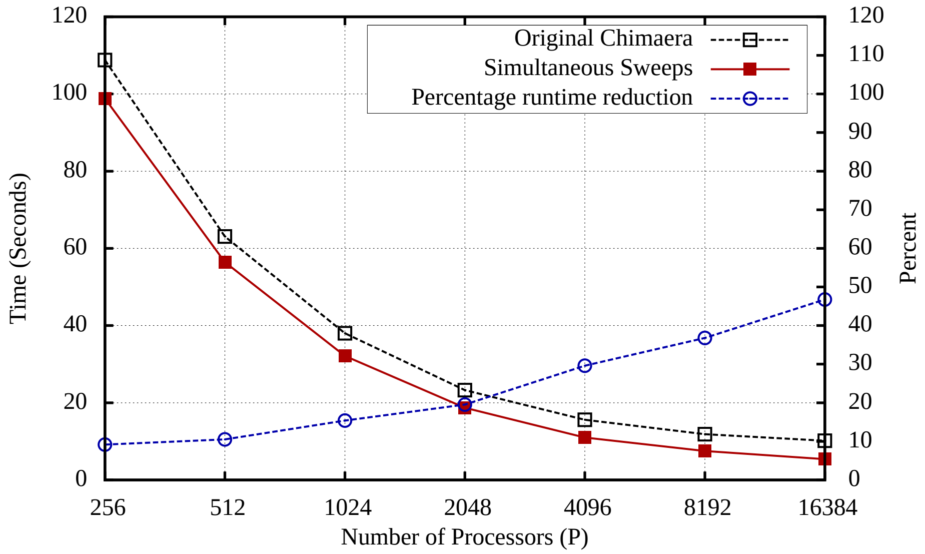


Figure 7.7: Simultaneous sweeps on all cores (Speculative Chimaera type application, 240x240x240 Cells, 1 time step, 16 energy groups 419 iterations, H_{tile}= 1)

we believe that the model detailed in this section provides a more precise and equally simpler model to predict the performance of multiple simultaneous sweeps. Moreover, the extension also shows the re-usability of the wavefront model. Figure 7.7 details the predictions from this model.

Particle transport codes such as Chimaera use a convergence criterion to halt the execution of application. The authors of Chimaera have noted that the convergence of the application depends very much on the order in which the sweeps are performed on the 3D grid of data. Thus, they say that the number of iterations for convergence will increase when simultaneous multiple sweeps are executed. But as the benefits of performing multiple simultaneous sweeps keeps increasing at large scale, we see that there is a trade-off that can be explored for this specific application. A further disadvantage of multiple simultaneous sweeps would be the increased requirement for memory and memory bandwidth between processor cores and main memory. More memory will be required for holding intermediate steps of the computation performed by all 8 sweeps. This will however this will be less of a problem on smaller tile sizes. A high memory bandwidth is needed to continually feed the cores with data to perform the computations for all 8 sweeps per sweep step. Thus having a node architecture with high memory bandwidth (such as the Cray XT3/XT4) will be advantageous.

7.4 Model Extensions for Heterogeneous Resources and Irregular/Unstructured Grids

The analytic reusable model in Chapter 4 assumes that the wavefront application operates on a regular orthogonal grid of data. Additionally the model implicitly assumes that the computational resources and the communication network is homogeneous. This implies that the computation per sweep step, and each of the near neighbour communication costs remain the same (or have very low variance) across the 2D processor array. In this section we develop simple extensions to the model to account for (1) heterogeneous computation and communication resources and (2) high variance across computation requirements - i.e. irregular/unstructured data grids.

Both irregular/unstructured data grids and heterogeneous compute nodes increases the variance of the time to compute a block of cells on a processor. Therefore we explore both of these cases, by considering the variance of W based on the processor indices (i, j). More specifically, when a 3D data grid is assigned to a 2D processor array (as in Figure 4.1), let the time to compute a block of cells of height H_{tile} assigned to a processor (i, j) be given by $W_{i,j}$. If the 3D data grid is irregular/unstructured and/or the compute resources are heterogeneous, then we can assume considerable differences between $W_{i,j}$ and $W_{i',j'}$ where $i \neq i'$ and $j \neq j'$. We consider three cases based on the heterogeneity of the compute and communication resources and the computational requirements of the 3D data grid, to motivate the model extensions as follows:

1. Each cell in the 3D data grid has similar computational requirements (i.e. homogeneous cells), the 3D grid is structured (i.e. is 3D orthogonal), but the computational resources (i.e. processors) are heterogeneous.

2. Each cell in the 3D data grid has different computational requirements (i.e. heterogeneous cells), the 3D gird is structured, and the computational resources are homogeneous.

3. Each cell in the 3D data grid has similar computational requirements (i.e. homogeneous cells), the 3D grid is unstructured and the computational resources are homogeneous.

The first case is simply when a structured regular orthogonal grid of data is solved, using different types of processors. In the second case, the 3D data grid is assumed to consist of cells that have different computational requirements. For instance, the number of floating-point operations to be solved per cell may vary on each cell across the grid. In the final case, an unstructured grid is considered, where the division of work across processors will vary as now the 3D grid is not orthogonal. We explore each case in the following subsections.

7.4.1 Homogeneous Cells, Structured Grid and Heterogeneous Resources

Recall that the basic reusable model was developed considering the critical path time for wavefront execution, and that the diagonals of the 2D processor array were considered as stages of a pipeline. Thus when there are heterogeneous $W_{i,j}$ times, the critical path will be determined by the processors that have the largest $W_{i,j}$ costs on the path of a given wavefront. More specifically, considering a wavefront that begins at the corner processor $(1, 1)$ and ends at the opposite corner processor (n, m), the critical path consists of the time to fill the stages of the pipeline and repeats at the rate of the slowest stage. Thus only 2 considerations and modifications to the basic reusable model are required in the case of heterogeneous resources computing a regular orthogonal grid of data.

The first of these is finding a method to obtain the worst case path cost from processor $(1, 1)$ to (n, m) considering a single sweep. This can be easily obtained by examining the processors in the 2D array as edges of a directed graph with $W_{i,j}$ values as costs to traverse the graph. Now the problem reduces to finding the path with the highest costs that connect $(1, 1)$ and (n, m). From a practical point of view this will require measuring the average computation per tile on each processor. The critical path time for pipeline fill can then be computed using the recursive expression in (4.2.9), where the maximum cost for each stage is already considered.

However, when using the model for speculative studies, measuring $W_{i,j}$ for each processor is not practical. For instance, if we are speculating on the scalability of the application for a system that has a larger number of processors, than is actually available. More specifically, consider the following example: Let a 3D wavefront application be assigned to a 2D processor array with a total number of processors $n \times m$, where the pool of processors takes a range of times to compute a block of cells of height H_{tile}. Let this range be given by $W_1, W_2, W_3, .., W_r, .., W_{max}$. Additionally, let the probability that a processor takes W_r time to complete a block of cells of height H_{tile} be given by P_r, where $\sum_{r=1}^{max} P_r = 1$. To account for the variance of computation, we modify the computation time W in (4.2.9) by replacing it with the averaging expression in (7.4.1).

$$W_{avg} = \sum_{r=1}^{max} P_r W_r \tag{7.4.1}$$

As can be seen, we are using simple averaging to account for the variance in computation time. We show that such approximations will provide adequately accurate qualitative speculations. But, if any other more detailed distribution function is known for the variance of W (for example if the probabilities that P_r changes with the number of processors, and this function is known) then that could be easily included in the (4.2.9) expression.

The second consideration is that the time for the steady state wavefront operation in T_{stack} should be set to the rate of the longest $W_{i,j}$. Assuming W_{max} is the largest time to compute a block of cells we replace W in (4.2.12) by W_{max}.

Table 7.1: Predictions for a system with heterogeneous processors (Chimaera 240x240x240 Cells, 1 time step, 16 energy groups 419 iterations, H_{tile} = 1)

Number of Processors	Simulation Model Predictions (sec)	Analytic Model Predictions (sec)	Difference (%)
256	206.78	202.21	-2.21
512	114.63	113.1	-1.34
1024	68.99	64.83	-6.03
2048	38.9	36.8	-5.4
4096	24.65	22.45	-8.93
8192	18.07	15.33	-15.12
16384	13.1	11.93	-8.94
32768	12.37	10.68	-13.69
65536	11.42	10.78	-5.66

Table 7.1 gives representative validations of this model comparing predictions from the model, to those from the WarPP simulation models. The scenario explored in this validation is for a pool of processors that take $0.5W, W$ and $2W$ times to compute a tile. The probability that a processor will take $0.5W, W$ or $2W$ is : 0.5, 0.13 and 0.37 respectively. This example was inspired by a typical procurement practice with a limited budget, where an organisation

might already have a pool of processors that compute a tile in $2W$ time, but intend to expand its system by procuring processors that compute a tile in W and $0.5W$ during the course of two upgrade cycles. Thus our model is able to provide predictions for the capability of the system after the upgrades.

Another scenario that can make use of such a probability distribution may occur when running a wavefront application on a wide area network (WAN). The network may have several partitions, each located at a different site. Each site might be made up of homogeneous processors while different sites have different processors. Now, assuming the job scheduler can select any processor from any site with equal probability to execute a parallel job, we can easily determine the probabilities of processor distribution as above. If a different scheduling policy is used, then this can also easily be incorporated into the probability distribution. For example, a workload characterisation of the systems in the WAN may provide a statistical measure that takes into account the typical availability of resources and the behaviour of the scheduler in such scenarios.

Similarly, for wavefront applications running on heterogeneous communication resources, the critical path time is limited by the worst case communication link. Thus, we use the maximum cost for communicating in (4.2.12), while a similar averaging cost can be used for the communication terms in (4.2.9). To simplify the measurement process and the model, we note that if in a wavefront code the messages sent are sufficiently small, then we can ignore the contribution of bandwidth if the sum of message overhead and network latency is large. I.e. in such a case we characterise the network performance by the value given by the intercept of graphs such as Figure 4.2. Our experience has been that for high performance configurations of a given wavefront code, the messages sent are sufficiently small to satisfy the above assumption. We see the importance of latency compared to bandwidth in Figure 5.16 and give further proof of this later in this chapter.

7.4.2 Heterogeneous Cells, Structured Grid and Homogeneous Resources

In this case, each cell has different computational requirements. This essentially makes the computation time of each tile computed per sweep step different, even within a single processor. Thus the steady state terms in (4.2.12) need to take in to account the maximum computation and communication costs for each sweep step. If we denote the set of all processors by $\forall P$ and the set of all communication links by $\forall l$ then we can rewrite (4.2.12) to account for heterogeneous cells as follows:

$$\begin{aligned} T_{stack} = \sum_{step=1}^{N_z/H_{tile}} \{ & \max_{\forall l}(Receive_W) + \max_{\forall l}(Send_E) + \\ & \max_{\forall l}(Receive_N) + \max_{\forall l}(Send_S) + \max_{\forall P}(W_{i,j} + W_{pre,i,j})\} \end{aligned} \tag{7.4.2}$$

In other words, the critical path now consists of the maximum computation block and communication link at each sweep step of the wavefront code. No modifications are needed to (4.2.9) as the maximum is already considered. Unfortunately, knowing the maximum before execution of the code may be difficult in practice. This limits our ability to make speculative studies using the model. From historical data however, the compute requirements for each cell may be obtainable. For instance it may be possible to build a map of the 3D cells by detailing each cell's floating-point operation count. Then by using a processor's achieved floating point operation rate (for computing particle transport codes for instance) we can obtain the compute time per tile on each processor for each tile given the size of H_{tile}.

7.4.3 Homogeneous Cells, Unstructured Grid and Homogeneous Resources

In the case where the data grid is unstructured (i.e. each processor computes a different number of cells) the critical path time of the wavefront application will again be limited by the worst case compute block and worst case communication link. The model given in the previous section can therefore still be used to model this case. Assuming that the computational requirements of each cell are homogeneous, the variation in the computation time per sweep step occurs due to processors not computing an equal number of cells. In other words there is a load imbalance. In unstructured data girds, the assignment of cells to each processor is usually handled by a mesh partitioner [166, 167]. Thus, if the partitioning of the grid is known, the computational time per sweep step could be computed for each tile on each processor. Again, some historical knowledge of the code's runtime behaviour and computation, communication requirements as needed, but once a map of the number of cells assigned to each processor is built, (4.2.9) and (7.4.2) can be easily used to obtain predictions.

7.5 System Procurement and Bottleneck Analysis

Identifying key issues that affect the performance of a workload is particularly important for application optimisation as well as for HPC procurement, operation, maintenance and upgrading. In this section we assess several system procurement questions with regard to the Chimaera benchmark, using the analytic model. Furthermore we attempt to shed light on the features of an HPC system that are best suited to running wavefront codes.

7.5.1 Larger Problem Sizes

The decision to purchase a new HPC system or upgrade an existing one may often be due to an increase in problem size. For example an increased resolution of the 3D grid may be needed to obtain higher accuracies for the problem solved. The computation/communication requirements of such scenarios for wavefront applications can be easily explored using the analytic models developed in this research. Figure 7.8 details the expected parallel efficiency of solving larger problem sizes of the Chimaera application on a system similar to the CSC-Francesca system with an increasing number of processors.

Recall that parallel efficiency (given by (2.3.2)) can be used as a measure of scalability of the application on a target system. There is a decline in parallel efficiency when increasing the number of processors. This can be attributed to the time spent in communication, which proportionally increases relative to computation as the problem size per processor reduces. Moreover, the pipeline fill increases with the larger number of processors, also contributing to increased communication. An organisation might prefer a parallel efficiency of 50% as

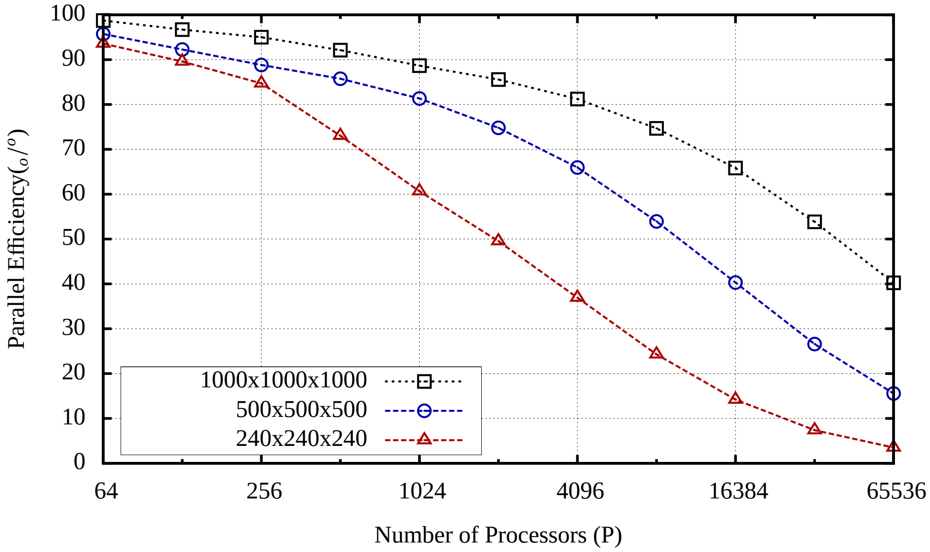


Figure 7.8: Parallel efficiency of larger problem sizes (Chimaera, 1 time step, 16 energy groups 419 iterations)

a very basic metric of evaluating the number of processors a user should choose during a Chimaera run. This half way mark may be considered as the point where enough useful (computation) work is done with an acceptable execution time. In other words at the lower end, say at less than 30% efficiency, communication time makes up a substantial propotion of the execution time. At the high end, although more useful work is done, the time to solution is unacceptably large. Depending on the system-wide number of jobs to be run simultaneously, at 50% efficiency, an approximate processor core count for procurement may be obtained. It can be seen that the scalability increases with larger problem sizes, requiring more processor cores to reach the 50% mark.

7.5.2 Computation, Latency and Bandwidth

Recall that in Chapter 5, Figures 5.13, 5.14, 5.15 and 5.16, detailed the breakdown of computation/communication as well as the contributions of latency and bandwidth to the critical path of a wavefront code's execution, on the Cray XT4. In this section we further explore the runtime trade-offs when each of these components is changed. Similar to all the previous results in this chapter, we base our analysis on an Intel/Xeon, Infiniband cluster. Figure 7.9, presents the predicted change in runtime from using processor cores with 10%, 20% and 50% higher performance. We see that in small processor configurations (i.e. towards the left of the graph) the improvement on computation significantly reduces the runtime. When the total runtime becomes more dependent on the communication performance on larger processor configurations, the computing benefit diminish.

Similarly, the affects of changing latency and bandwidth are presented in Figure 7.10 and Figure 7.11. We see that on larger processor configurations, the bottleneck is due to the latency of the network. The model predicts that improvements to the bandwidth of the network contribute less to the critical path. However, we suspect that lower bandwidth manifests itself as network contention, degrading the performance of a wavefront code. It can be shown that

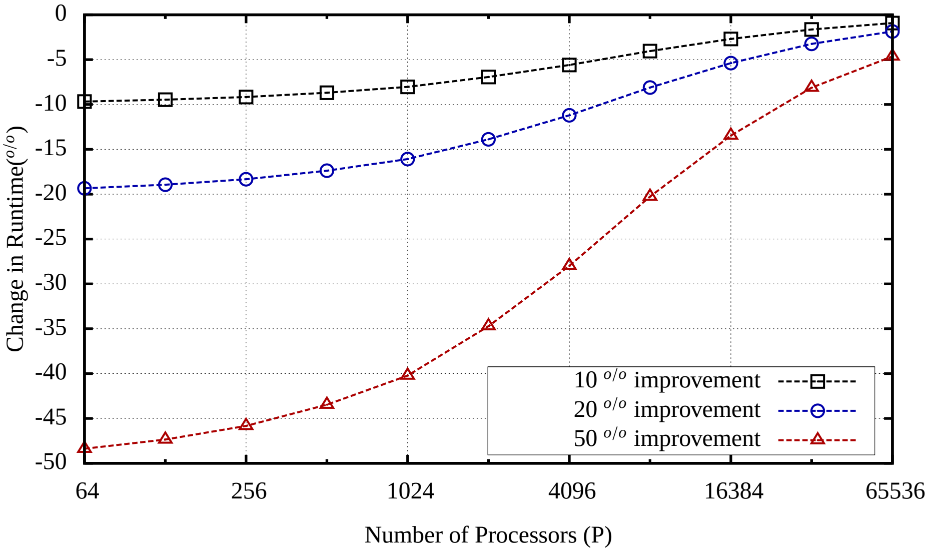


Figure 7.9: Change in runtime due to improved computation performance (Chimaera 240x240x240, 1 time step, 16 energy groups 419 iterations)

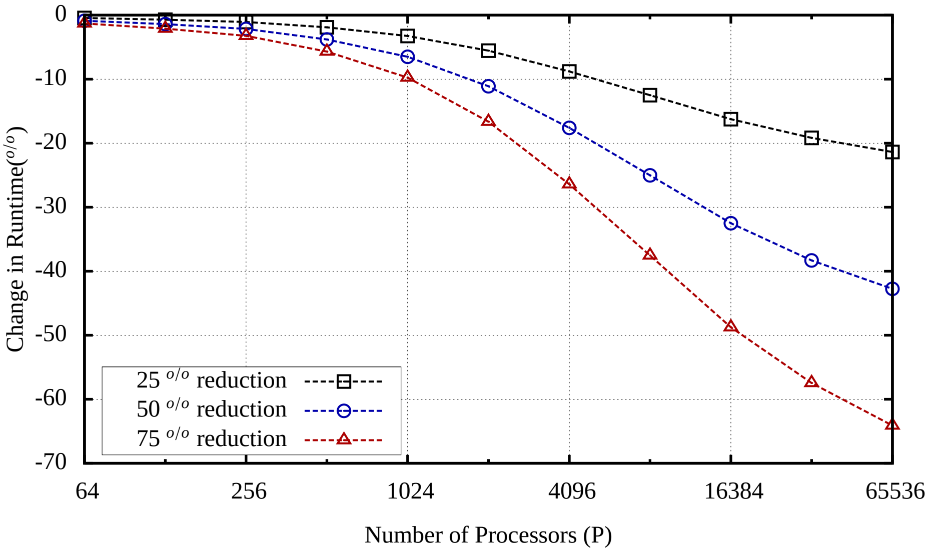


Figure 7.10: Change in runtime due to reduced network latency (Chimaera 240x240x240, 1 time step, 16 energy groups 419 iterations)

the message size and the frequency of injecting messages to the NIC by a processor connected to a 3D torus network such as the one found in a CrayXT4 will not saturate the network. Thus we can safely ignore any network contention on such a system.

For an InfiniBand network, given the number of ports on a switch (288 in the switch in CSC-Francesca), such a claim require further investigation for wavefront codes, given insights regarding contention explored in recent works [168]. The message injection rate of a node running four MPI tasks from a wavefront code such as Chimaera will also not saturate the switch. However, depending on the number of NICs per node, the topology of the network and the number of available ports on the switch, multiple switch levels and the number of cores or processors in a node, there may be a possibility of network contention. These issues

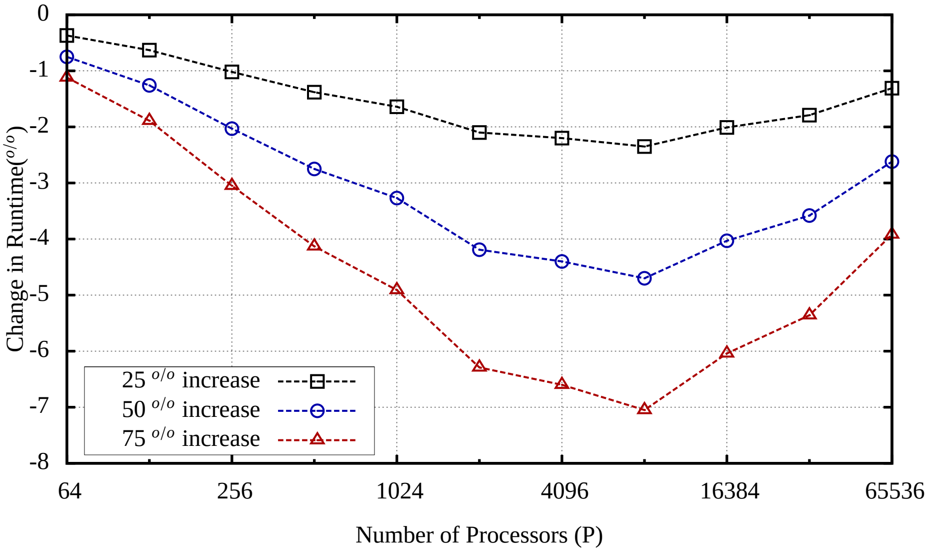


Figure 7.11: Change in runtime due to increased network bandwidth (Chimaera 240x240x240, 1 time step, 16 energy groups 419 iterations)

are currently being investigated as future work. A further problem with multiple switches is the increased physical wire time and possible router overheads. These costs degrade the network latency even if the possible contention issues are ignored. Given the choice to buy two different versions of InfiniBand networks, one having two or four times more bandwidth than the other and thus being considerably more expensive, while both has the same latency, the model shows that for wavefront codes it is better value for money to purchase the lower bandwidth InfiniBand network. Alternatively, investing in a lower latency network will provide better performance.

7.6 Summary

The optimisations detailed in this chapter, as well as in Chapter 5, are by no means exhaustive. There may be other variations and different HPC hardware/software platforms that yet provide further insights for those systems and configurations. However, we believe that the predictive reusable models developed in this thesis and the insights gained in wavefront operations will serve as a significant aid to understanding any existing or future unexplored variations.

8 Conclusions and Future Work

Pipelined wavefront computations are an important and ubiquitous class of parallel applications. Computational solutions for a number of scientific and engineering problems are based on these codes. To aid the design, configuration and optimisation for these applications, as well as to assist in the procurement of HPC systems that are to execute these codes, there has been, and continues to be, significant research in understanding their performance. Despite previous research, a number of key open questions remain.

Each of the previous performance engineering studies were specific to the two wavefront applications NPB-LU and Sweep3D. These studies require significant and unspecified restructuring to apply them to other wavefront applications of interest. The complex synchronisation patterns of computation and communication in these codes presents significant amount of variation and optimisation possibilities. Therefore, a key difficulty has been that each new application of this type has had to be re-analysed and re-modelled. This problem is compounded due to the fast development rate of HPC hardware and software systems. As a result, previously unexplored systems' performance issues were difficult to incorporate into existing performance models. Previous approaches of modifying existing performance studies to reflect the behaviour of new code on new systems was error-prone and required extensive validation. One of the underlying goals of this research therefore was to develop methods that reduce this effort, and in so doing alleviate some of the above problems, and therefore assist in future performance engineering work of wavefront applications.

A final key motivation was to address the lack of a comprehensive exploration of the performance and optimisation possibilities of the applications that belong to the pipelined wavefront class of applications. That is, given the underlying scientific or numerical solution, what is the optimum software design for this class of applications and how should it best be run on a particular HPC system. The aim was to develop techniques that not only answer these questions, but also motivate and expose new questions and at the same time enable us to obtain solutions in an efficient and low cost manner.

8.1 Contributions and Conclusions

Motivated by the above key issues, the first part of this thesis presented the development of a reusable analytic model to predict the runtime and scaling behaviour of pipelined wavefront computations. As this class of applications is ubiquitous in modern HPC workloads, our approach was to abstract the commonly occurring computation-communication patterns and develop a reusable performance characterisation. The goal was to demonstrate the benefits of developing reusable performance models to reduce the demanding task of performance engineering.

The first contribution of this dissertation is the formulation and development of a reusable - *plug-and-play* - analytic model based on LogGP for the predictive performance analysis of pipelined wavefront computations. The model enables the prediction of the runtime and scaling behaviour of different message-passing-based wavefront applications running on modern parallel platforms. A key feature of the model is that it requires only a few input parameters to project performance for wavefront computations with a range of variations. Furthermore, the parameters are simple and are not difficult to obtain. A given set of parameter values succinctly describe the operation of a wavefront code, allowing the use of these values as a concise summary that describes the configuration and variations of a given application. As the name implies, the parameters can be used in a plug-and-play fashion to obtain models for a variety of existing and speculative wavefront codes.

The reusable model uses expressions that capture the various sections of the critical path of the operation of wavefront codes. These expressions can be used as building blocks to elucidate the critical path of execution. The model is abstract in order to capture the performance details that matter in general. Due to its reusable nature, its flexibility can be extended to capture more specific performance behaviours for a given application and a given HPC system. The extensions explored in this dissertation include predictive models for performance of wavefront codes (1) on the Cray XT4 which has CMP node architectures, (2) on heterogeneous resources, and (3) on irregular/unstructured data grids. Other extensions include variations in computation per sweep step, differences in iterations, multiple sweeps (both serial, overlapped and simultaneous) and operation of wavefronts on a 2D data grid.

Further contributions due to model development and analysis include (1) the highly accurate MPI send, receive and all-reduce communication models that have been developed for the Cray XT3/XT4 and (2) the model extensions that capture the contention issues during wavefront operation on CMP nodes on the Cray XT3/XT4. The former develops models for both node-to-node (off-chip) as well as core-to-core (on-chip) communications. These models are themselves reusable in other applications that use MPI communications primitives. The CMP extensions provide a first look at the limitations of CMP architectures when processor cores are sharing resources such as single bus to RAM, shared DMA memory access controllers and shared NIC. On the CrayXT3/XT4 system at the time of our research, we modelled the contention caused by a shared bus on nodes with a dual-core processor. The model was then used to extrapolate further models that capture the behaviour of 4, 8 and 16 core processors.

The reusable model was applied to predict the performance of three important benchmarks that use wavefront computations - NPB-LU, Sweep3D and Chimaera. Each model was validated on up to 8K processors on the Cray XT3/XT4 at ORNL as well as up to 256 processors on a smaller commodity cluster based on Intel Xeon processors and an InfiniBand interconnect. Results show excellent qualitative accuracy. Quantitative accuracy for all high performance configurations was over 85%. Further validations were carried out against a discrete event simulator showing high agreement of predictions for up to 65K processors. This level of accuracy has given us great confidence in the ability of the model to be used in various speculative studies to explore "what if" scenarios.

Experience from the analytic model development also provided a significant insight into increasing the accuracy and scalability of the PACE discrete event simulation system. The techniques that were subsequently deployed by the WarPP simulation system which has enabled us to model wavefront computations running on over 100K processor systems with predictive errors below 15%.

In the second part of this dissertation, the analytic model was used to conduct an extensive investigation into the performance behaviour of wavefront applications. In particular the model was used to obtain projections and insights for optimisations showing the significant utility of the reusable analytic model. Specific evaluations include (1) software configuration performance, (2) hardware platform questions including platform sizing, configuration and a case study that uses the performance engineering insights to assess system procurement decisions, (3) hardware platform design alternatives such as the optimal number of processor cores per node, and (4) optimisations and performance bottlenecks for wavefront computations demonstrating quantitatively and qualitatively the performance improvements from the optimisations.

8.2 Future Work

Plug-and-play performance models, where the user only needs to specify a few input parameter values in order to obtain performance predictions for application codes with different behaviour, have not been previously explored. Furthermore, the fact that the model developed here is for a significantly complex class of parallel applications is also novel. An open question addressed in this research is whether building in the various possible behaviours leads to a more complex set of equations, possibly negating the advantages of the model generality. Our work has shown that it has been possible to generate a set of equations that is as simple as the equations that are tailored to a given application. This unanticipated result may or may not hold for other classes of applications. However, the results are an incentive to extend this study more widely and use reusable performance characterisations for other important classes of applications.

There are several further areas for future work. These falls into two distinct categories: (1) Extensions to the reusable model, and (2) future work on wavefront computations. The first involves several further validations and extensions to the reusable model, particularly when applying it to study more specific performance issues. The latter forms a set of longer term ideas that have been identified due to the work in this thesis in conjunction with the current and future anticipated HPC research needs.

8.2.1 Further Validations and Model Extensions

A collection of model extensions and issues discussed in Chapter 7 were analysed in a speculative manner due to the unavailability of applications to validate the predictions. These include wavefront operation on irregular and/or unstructured grids of data, heterogeneous resources and multiple simultaneous wavefronts. The model predictions have given us insights as to the performance behaviour of these deployments. The models should be applied to specific applications to further ascertain our insights. The case of using heterogeneous resources is of further interest for investigating wavefront behaviour on widely distributed networks. Owing to the tightly coupled and frequent communication pattern of wavefront codes, it has been only used in low-latency and high bandwidth networks, particularly MPPSs, SMPs and tightly coupled clusters. It would be appropriate to investigate the behaviour of these codes on a more loosely coupled network such as a Condor [169] pool. Condor is a middleware that enables implementation of a Computer Grid [170]. In such a case the computation and communications resources would be heterogeneous. Such systems in most cases are made up of workstations spread across an organisation's sites and scavenge idle nodes to run codes via Condor. Thus, these systems are significantly cheaper and may present an organisation with an almost free parallel computational resource. Thus, predicting the performance of wavefront codes running on such systems may provide insights that are significant.

The reusable model's ability to be extended can be investigated with several more specific performance issues. One of these will be investigating and modelling network contention arising in very low bandwidth networks. The analytic models predict a very low contribution arising from the network bandwidth. Most modern HPC systems have more than enough bandwidth to support the rate of message injection by the nodes to the network without saturating it. An interesting issue would be to analyse the case where the network is saturated or is close to saturation. We anticipate that such a scenario will require a model which accounts for network contention. Modelling such contention will be specific to the network and application and will be very useful in understanding bottlenecks on such systems.

More application specific models can be easily incorporated into the reusable model. An example would be a sub-model that parametrises the memory performance of a node such that computation per sweep step can be predicted. Such sub-models are inherently application-specific but may provide further insights into optimisation possibilities.

8.2.2 Future Work on Wavefront Computations

With the advent of multi-core processors, and the increasing availability of parallel systems, there has been a growing need for parallel programming languages, and programming methodologies which explicitly support parallelism. The current dominant parallel programming standards - MPI and OpenMP, both implement parallelism on top of sequential programming languages such as Fortran and C. Moreover, a huge amount of programmer effort and time is required to develop applications using them. To ease this burden, as well as to develop languages and supporting methodologies that explicitly facilitate writing programs in parallel, there are several key on-going research and development efforts - Unified Parallel C [171, 172], Co-array Fortran [173], Titanium [174], Fortress [175], Chapel [176] and X10 [177]. All of these languages use, the Partitioned Global Address Space (PGAS) model to achieve parallelism. In the PGAS model, a global memory address space is assumed where it is logically partitioned and each partition is allocated to each processor. Thus a local processor (or a thread) has an *affinity* to its partition of memory, but other processors (threads) can access this memory by addressing both the processor and the memory location without explicit messaging as in message passing [178]. Additionally, the explicit partitioning of data is also not required as is the synchronisation of data. Thus the aim is to free the programmer to solve the domain problem, rather than worry about parallelism at a low level.

Wavefront computations have yet to be implemented using languages that adhere to this new PGAS model. The performance properties will clearly require a comprehensive study, but it would be interesting to see the performance relationships of such an implementation with the existing applications based on the message passing paradigm. The PGAS languages may ease the process of parallel programming. Whether an application written in these languages results in higher performance, and what bottlenecks will need to be overcome to achieve this, remains to be seen.

Similar to these emerging parallel programming languages, new hardware and systems architectures have shown their use as viable HPC platforms. Examples include computational units that make use of Graphics Processor Units (GPUs) or Vector elements based processors such as the IBM Cell [179] and Field Programmable Gate Array based solvers (FPGAs). These innovative hardware platforms are now being increasingly incorporated into traditional HPC systems as accelerator units that promise speedy solutions for particular types of computations. Developing performance models for wavefront applications running on such platforms will be an important and interesting study. We hope that the insights gained from the reusable wavefront model will be advantageous for such work and that it will enable further extensions to the models to capture such specialist hardware.

∼∼∼∼∼

Traditional scientific research, based on experimental and theoretical approaches have been greatly enhanced by modern computer systems. With the increasing availability of parallel systems, the use of computational methods and HPC has enjoyed rapid growth in the last decade. Programming in parallel and the understanding the related parallel performance behaviour of applications on modern HPC systems continues to present many challenges. Nevertheless, methods, tools and techniques to address these challenges appear to be attainable. We look forward to the significant insights resulting from performance engineering studies and to the advancement of parallel programming and high performance computing design.

Bibliography

[1] S.A. Jarvis, D.P. Spooner, G.R. Mudalige, B.P. Foley, J.Cao, and G.R. Nudd. *Performance Evaluation of Parallel and Distributed Systems*, chapter Performance Prediction Techniques for Large-scale Distributed Environments. Mohamed Ould-Khaoua and Geyong Min Eds. Nova Science, 2005. v

[2] G.R. Mudalige, M.K. Vernon, and S.A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, April, 2008. v

[3] G.R. Mudalige, S.A. Jarvis, D.P. Spooner, and G.R. Nudd. Predictive Performance Analysis of a Parallel Pipelined Synchronous Wavefront Application for Commodity Processor Cluster Systems. In *Proc. IEEE International Conference on Cluster Computing - Cluster2006*, Barcelona, September 2006. IEEE Computer Society. v, 105

[4] S.D. Hammond, G.R. Mudalige, J.A. Smith, and S.A. Jarvis. Performance Prediction and Procurement in Practise: Assessing the Suitability of Commodity Cluster Components for Wavefront Codes. In *Proc. Performance Engineering Workshop '08 (UKPEW)*, Imperial College, London, July 2008. v, 94, 107

[5] G.R. Mudalige, S.D. Hammond, J.A. Smith, and S.A. Jarvis. Predictive Analysis and Optimisation of Pipelined Wavefront Computations. In *Proc. 11th Workshop on Advances in Parallel and Distributed Computational Models (APDCM 2009), 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009)*, Rome, Italy, May 2009. IEEE Computer Society. v

[6] S.D. Hammond, G.R. Mudalige, J.A. Smith, and S.A. Jarvis. WARPP - A Tool Kit for Simulating High-Performance Parallel Scientific Codes. In *Proc. 2nd International Conference on Simulation Tools and Techniques (SIMUTools'09)*, Rome, Italy, March 2009. ACM Press. vi, 8, 94

[7] S.D. Hammond, J.A. Smith, G.R. Mudalige, and S.A. Jarvis. Predictive Simulation of HPC Applications. In *The IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA 2009)*, Bradford, U.K., 26-29 May. IEEE Computer Society. vi, 107

[8] D. Sundaram-Stukel and M.K. Vernon. Predictive Analysis of a Wavefront Application Using LogGP. In *PPoPP '99: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 141–150. ACM Press, 1999. x, 3, 5, 21, 31, 48, 50, 54, 55, 57, 59, 60, 63, 69, 82

[9] D.J. Kerbyson, A. Hoisie, and H.J. Wasserman. Modelling the Performance of a Large-Scale Systems. In S.A. Jarvis, editor, *Proceedings of 19th Annual U.K Performance Engineering Workshop*, pages 2–14, University of Warwick, U.K, July 2003. WARWICKPRINT. 1

[10] S.A. Jarvis, D.P. Spooner, H.N. Lim-Choi-Keung, J. Cao, S. Saini, and G.R. Nudd. Performance Prediction and its Use in Parallel and Distributed Computing Systems. *Future Gener. Comput. Syst.*, 22(7):745–754, 2006. 1

[11] D.P. Spooner. *Performance-based Middleware for Grid Computing*. PhD thesis, University of Warwick, Department of Computer Science, 2005. 1, 94

[12] G.R. Nudd, D.J. Kerbyson, E. Papaefstathiou, S.C. Perry, J.S. Harper, and D.V. Wilcox. PACE: A Toolset for the Performance Prediction of Parallel and Distributed Systems. *Int. Journal of High Performance Computing Applications*, 14(3):228–251, Fall 2000. 1, 8, 29, 30, 31, 94, 98

[13] D.J. Kerbyson, A. Hoisie, and H.J. Wasserman. Use of Predictive Performance Modeling During Large-Scale Systems Installation. In *1st Int. Workshop on Hardware/Software Support for Parallel and Distributed Scientific and Engineering Computing(SPDEC-02)*, Charlottesville, September 2002. 1

[14] M. Yarrow and R. Van der Wijngaart. Communication Improvement for the LU NAS Parallel Benchmark: A Model for Efficient Parallel Relaxation Schemes. Technical Report NAS- 97-032, NASA Ames Research Center, November 1997. 2, 5, 31, 35, 40, 41, 44, 48

[15] F. Petrini, G. Fossum, J. Fernndez, A.L. Varbanescu, M. Kistler, and M. Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10. IEEE, 2007. 2

[16] L. Lamport. The Parallel Execution of DO Loops. *Commun. ACM*, 17(2):83–93, 1974. 2, 35, 37

[17] A. Hoisie, H. Lubeck, and H.J. Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parellel Architectures using Multidimentional Wavefront Applications. *Int. J of High Performance Computing Applications*, 14(4):330–346, Winter,2000. 2, 3, 31, 41, 49, 59, 80, 83

[18] Los Alamos National Laboratory LANL. `http://www.lanl.gov/`. 2

[19] The Atomic Weapons Establishment (AWE). `http://www.awe.co.uk/`. 2

[20] The Message Passing Interface (MPI). `http://www-unix.mcs.anl.gov/mpi/`. 2, 14, 20

[21] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, D. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991. 2, 41

[22] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997. 3, 7, 21, 51

[23] Sweep3d. The ASCI Sweep3d Benchmark.
`http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`. 3, 41, 47

[24] M.M. Mathis, N.M. Amato, and M.L. Adams. A General Performance Model for Parallel Sweeps on Orthogonal Grids for Particle Transport Calculations. Technical report, Texas A&M University, 2000. 3, 49, 117

[25] D.J. Kerbyson, A. Hoisie, and H.J. Wasserman. A Comparison Between the Earth Simulator and Alphaserver Systems using Predictive Application Performance Models. *Computer Architecture News (ACM)*, December 2002. 3, 5, 49, 59, 60

[26] A. Hoisie, O. Lubeck, H.J. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 219. IEEE Computer Society, 2000. 3, 31, 49

[27] J. Cao, D.J. Kerbyson, E. Papaefstathiou, and G.R. Nudd. Performance Modeling of Parallel and Distributed Computing Using PACE. In *19th IEEE Int. Performance, Computing and Communications Conf(IPCCC)*, page 485492, Phoenix, AZ, USA, Feb. 3, 49, 50

[28] J. Cao. *Agent-based Resource Management for Grid Computing*. PhD thesis, University of Warwick, 2001. 3

[29] S. Prakash and R.L. Bagrodia. MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 467–474, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. 3, 30, 32, 49, 50

[30] R. Bagrodia, E. Deeljman, S. Docy, and T. Phan. Performance Prediction of Large Parallel Applications Using Parallel Simulations. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 151–162, New York, NY, USA, 1999. ACM. 3

[31] V.S. Adve, R. Bagrodia, J.C. Browne, E. Deelman, A. Dube, E.N. Houstis, J.R. Rice, R. Sakellariou, D. Sundaram-Stukel, P.J. Teller, and M.K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Trans. Softw. Eng.*, 26(11):1027–1048, 2000. 3, 30, 31, 49, 50

[32] F. Wolf and B. Mohr. KOJAK - A Tool set for Automatic Performance Analysis of Parallel Applications. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, volume 2790 of *Lecture Notes in Computer Science*, pages 1301–1304, Klagenfurt, Austria, August 2003. Springer. Demonstrations of Parallel and Distributed Computing. 3, 27

[33] E. Papaefstathiou. Design of a Performance Technology Infrastructure to Support the Construction of Responsive Software. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 96–104, New York, NY, USA, 2000. ACM. 3

[34] E.C. Lewis. *Achieving Robust Performance in Parallel Programming Languages*. PhD thesis, University of Washington, 2001. 3

[35] V.S. Adve. *Analyzing the Behavior and Performance of Parallel Programs*. PhD thesis, University of Wisconsin - Madison, October 1993. 5, 31

[36] D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, H.J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modelling of a Large-Scale Application. In *Proceedings of SuperComputing*, Denver, 2001. 5, 31, 60

[37] M.M. Mathis and D.J. Kerbyson. Performance Modeling of Unstructured Mesh Particle Transport Computations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, NM, April 2004. 5, 31, 47, 49, 117

[38] M.M. Mathis, D.J. Kerbyson, and A. Hoisie. A Performance Model of Nondeterministic Particle Transport on Large-Scale Systems. In *Proc. Computational Science - ICCS 2003, LNCS*, volume 2659, pages 905–915. Springer-Verlag, 2003. 5, 31

[39] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement,Simulation, and Modeling*. Wiley-Interscience, New York, NY, April 1991. 6, 22, 24, 25, 26, 28, 30, 32, 33

[40] 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, and K.A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. 7

[41] BLAS. Basic Linear Algebra Subprograms. `http://www.netlib.org/blas/`. 7, 25

[42] LAPACK. Linear Algebra PACKage. `http://www.netlib.org/lapack/`. 7

[43] The WARwick Performance Prediction Toolkit (WarPP). `http://go.warwick.ac.uk/ep/pg/csrcbc/research/wppt/`. 8, 29, 31, 49, 94, 107

[44] D. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, August 1998. ISBN 1558603433, pp. 15,26-27,190. 11, 13, 18

[45] M.J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972. 12

[46] E.W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9):569, 1965. 13

[47] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent Control With "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971. 13

[48] E.W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138, 1971. 13

[49] The OpenMP Specification for Parallel Programming
. `http://www.openmp.org/`. 13

[50] MATLAB - the Language of Technical Computing.
`http://www.mathworks.com/products/matlab/`. 14

[51] High Performance Fortran (HPF). `http://hpff.rice.edu/`. 14

[52] L.M.Silvay and R.Buyya. *High Performance Cluster Computing: Programming and Applications*, chapter 2. Prentice Hall PTR, NJ, USA,, 1999. 15

[53] G.M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In *American Federation of Information Processing Societies*, volume 30, pages 483–485, 1967. 15, 16, 17, 30

[54] J.L. Gustafson. Re-evaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, 1988. 15, 17, 30

[55] M.D. Hill. What is Scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990. 17

[56] J. von Newmann. First Draft of a Report on the EDVAC. Technical report, 1945. 18

[57] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of Computing*, pages 114–118, New York, NY, USA, 1978. ACM. 18

[58] B. Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. 18

[59] T.J. Harris. A Survey of PRAM Simulation Techniques. *ACM Comput. Surv.*, 26(2):187–206, 1994. 18

[60] C. Papadimitriou and M. Yannakakis. Towards an Architecture-independent Analysis of Parallel Algorithms. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 510–513, New York, NY, USA, 1988. ACM. 19

[61] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computations. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 11–21, New York, NY, USA, 1989. ACM. 19

[62] A. Aggarwal, A.K. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theor. Comput. Sci.*, 71(1):3–28, 1990. 19

[63] L.G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990. 19

[64] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997. 19

[65] W.F. McColl. General Purpose Parallel Computing. In A M Gibbons and P Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 1993. 19

[66] The parallel virtual machine
. http://www.csm.ornl.gov/pvm/. 20

[67] R. Miller. A library for bulk synchronous parallel programming. In *BCS Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, pages 100–108. BCS, December 1993. 20

[68] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards Efficiency and Portability: Programming with the BSP Model. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, 1996. 20

[69] J.M.D. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A Proposal for a BSP Worldwide Standard. BSP Worldwide Standard
. http://www.bsp-worldwide.org/, April 1996. 20

[70] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993. 21

[71] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *SPAA '96: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, New York, NY, USA, 1996. ACM. 21

[72] M. Frank, A. Agarwal, and M.K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Principles Practice of Parallel Programming*, pages 276–287, 1997. 22, 31

[73] C. A. Moritz and M.I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):404–415, 2001. 22, 31

[74] J.J Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, Dept.of Computer Science University of Tennesee Knoxvill and Computer Science and Mathematics Division Oak Ridge National Laboratory, November 6 2004. 24

[75] Linpack.
http://www.netlib.org/linpack/. 24

[76] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.
http://www.netlib.org/benchmark/hpl/. 25

[77] Top 500 Supercomputing Sites.
http://www.top500.org/. 25

[78] F.H. McMahon. The Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range. Technical report, Lawrence Livermore National Laboratory, 1986. 25

[79] HPCC the HPC Challenge Benchmarks.
http://icl.cs.utk.edu/hpcc/. 25

[80] Standard Performance Evaluation Corporation. http://www.spec.org/benchmarks.html. 25, 26

[81] Intel MPI Benchmarks.
http://www.intel.com/software/products/cluster/mpi/mpi_benchmarks_lic.htm. 25, 62

[82] Benchmakking MPICH.
http://www-unix.mcs.anl.gov/mpi/mpptest/. 25

[83] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A Network Protocol Independent Performace Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, 1996. 25

[84] Special Karlsruher MPI Benchmark.
http://liinwww.ira.uka.de/~skampi/. 25

[85] EPCC openMP Microbenchmarks.
http://www.epcc.ed.ac.uk/research/openmp/. 25

[86] H. Sutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal
http://www.ddj.com/, March 2005. 25

[87] STREAM Sustainable Memory Bandwidth in High Performance Computers.
http://www.cs.virginia.edu/stream/. 25

[88] H. J. Curnow and B. A. Wichmann. A Synthetic Benchmark. *Computer Journal*, 19(1), 1976. 25

[89] R.P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Commun. ACM*, 27(10):1013–1030, 1984. 25

[90] D.A. Patterson and J.L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2003. 25

[91] ASC the Advanced Simulation and Computing Program.
http://http://www.lanl.gov/asc/. 25, 41

[92] NAS Parallel Benchmark.
http://www.nas.nasa.gov/Resources/Software/npb.html. 25, 41

[93] J. Gustafson. Purpose-Based Benchmarks. *International Journal of High Performance Computing Applications*, 18(4):475–487, 2004. 26

[94] V. Strassen. Gaussian Elimination is not Optimal. *Numer. Math*, 13:354–356, 1969. 26

[95] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, 1996. 26, 27

[96] S.L. Graham, P.B. Kessler, and M.K. McKusick. gprof: a Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982. 26, 27

[97] Performance Application Programming Interface.
http://icl.cs.utk.edu/papi/. 26, 27, 104

[98] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000. 26, 27

[99] CrayPat.
http://docs.cray.com/. 27

[100] Using OPT - A White Paper.
http://www.allinea.com/downloads/OPTWhite.pdf. 27

[101] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. An Overview of the Pablo Performance Analysis Environment. Technical report, University of Illinois, Department of Computer, 1992. 27

[102] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995. 27

[103] TAU - Tuning and Analysis Utilities.
http://www.cs.uoregon.edu/research/tau/. 27

[104] S.S. Shende and A.D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006. 27

[105] PGPROF - Performance Profiler.
http://www.pgroup.com/products/pgprof.htm. 27

[106] V. Herrarte and E. Lusk. Studying Parallel Program Behavior with `upshot`. Technical Report ANL–91/15, Argonne National Laboratory, 1991. 27

[107] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward Scalable Performance Visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999. 27

[108] HPCToolkit.
http://hipersoft.cs.rice.edu/hpctoolkit/. 27

[109] R. Wolski, N.T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999. 28

[110] R. Wolski. Experiences with Predicting Resource Performance On-line in Computational Grid Settings. *SIGMETRICS Perform. Eval. Rev.*, 30(4):41–49, 2003. 28

[111] P.A. Dinda. The Statistical Properties of Host Load. *Sci. Program.*, 7(3-4):211–229, 1999. 28

[112] P.A. Dinda. *Resource Signal Prediction and its Application to Real-time Scheduling Advisors*. PhD thesis, Pittsburgh, PA, USA, 2000. Chair-David R. O'Hallaron. 28

[113] P.A. Dinda. Online Prediction of the Running Time of Tasks. *hpdc*, 00:0383, 2001. 28

[114] P.A. Dinda. A Prediction-Based Real-Time Scheduling Advisor. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 35, Washington, DC, USA, 2002. IEEE Computer Society. 28

[115] P.A. Dinda. Design, Implementation, and Performance of an Extensible Toolkit for Resource Prediction in Distributed Systems. *IEEE Trans. Parallel Distrib. Syst.*, 17(2):160–173, 2006. 28

[116] S. Vazhkudai and J.M. Schopf. Using Regression Techniques to Predict Large Data Transfers. *Int. J. High Perform. Comput. Appl.*, 17(3):249–268, 2003. 28

[117] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A Toolkit for Distributed System Performance Analysis. In *MASCOTS*, pages 267–273, 2000. 28

[118] S. Chiang and M.K. Vernon. Characteristics of a Large Shared Memory Production Workload. *Lecture Notes in Computer Science*, 2221:159, 2001. 28

[119] D.G Feitelson and B. Nitzberg. Job Characteristics of a Production Pparallel Scientific Workload on the NASA Ames iPSC/860. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 337–360. Springer, 1995. 28

[120] S. Hotovy. Workload Evolution on the Cornell Theory Center IBM SP2. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 27–40. Springer-Verlag, 1996. 28

[121] K. Windisch, V. Lo, R. Moore, D. Feitelson, and B. Nitzberg. A Comparison of Workload Traces From Two Production Parallel Machines. In *6th Symp. Frontiers Massively Parallel Comput.*, pages 319–326, 1996. 28

[122] J.M. Schopf and F. Berman. Stochastic Scheduling. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 48, New York, NY, USA, 1999. ACM. 28

[123] F.D. Berman, R. Wolski, S. Figueira, J.M Schopf, and G. Shao. Application-level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, Washington, DC, USA, 1996. IEEE Computer Society. 28

[124] A. Gefflaut and P. Joubert. SPAM: A Multiprocessor Execution-Driven Simulation Kernel. *Int. Journal in Computer Simulation*, 6(1):69, 1996. 29

[125] S. Girona and J. Labarta. Sensitivity of Performance Prediction of Message Passing Programs. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, Nevada, USA, July 1999. 29

[126] J. Labarta, S. Girona, and T. Cortes. Analyzing Scheduling Policies Using DIMEMAS. *Parallel Comput.*, 23(1-2):23–34, 1997. 29

[127] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice Parallel Processing Testbed. *SIGMETRICS Perform. Eval. Rev.*, 16(1):4–11, 1988. 29

[128] E.A. Brewer, C. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. In *Measurement and Modeling of Computer Systems*, pages 247–248, 1992. 29

[129] Digital Equipment Corporation. *prof(1). Ultrix 4.0 General Information*. Vol. 3B (Commands(1): M-Z). 29

[130] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Measurement and Modeling of Computer Systems*, pages 48–60, 1993. 30

[131] S.S. Mukherjee, S.K. Reinhardt, B. Falsafi, M. Litzkow, M.D. Hill, D.A. Wood, S. Huss-Lederman, and J.R. Larus. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8(4):12–20, 2000. 30

[132] S. Prakash, E. Deelman, and R. Bagrodia. Asynchronous Parallel Simulation of Parallel Programs. *IEEE Trans. Softw. Eng.*, 26(5):385–400, 2000. 30

[133] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L.V. Kalé. Simulation-based Performance Prediction for Large Parallel Machines. *Int. J. Parallel Program.*, 33(2):183–207, 2005. 30

[134] U. Legedza and W.E. Weihl. Reducing Synchronization Overhead in Parallel Simulation. In *Workshop on Parallel and Distributed Simulation*, pages 86–95, 1996. 30

[135] R. Berry and K.M. Chandy. Performance Models of Token Ring Local Area Networks. In *SIGMETRICS '83: Proceedings of the 1983 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 266–274, New York, NY, USA, 1983. ACM. 31

[136] V.S. Adve and M.K. Vernon. Performance Analysis of Mesh Interconnection Networks with Deterministic Routing. *IEEE Transactions on Parallel and Distributed Systems*, 05(3):225–246, 1994. 31

[137] D.J. Sorin, V.S. Pai, S.V. Adve, M.K. Vernon, and D.A. Wood. Analytic Evaluation of Shared-memory Systems with ILP Processors. *SIGARCH Comput. Archit. News*, 26(3):380–391, 1998. 31

[138] M. Chiang and G.S. Sohi. Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput-Oriented Environment. *IEEE Trans. Comput.*, 41(3):297–317, 1992. 31

[139] M.K. Vernon, E.D. Lazowska, and J.Zahorjan. An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-consistency Protocols. *SIGARCH Comput. Archit. News*, 16(2):308–315, 1988. 31

[140] A.G. Greenberg, I. Mitrani, and L. Rudolph. Analysis of Snooping Caches. In *Performance '87: Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 345–361, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co. 31

[141] D.J. Kerbyson, S.D. Pautz, and A. Hoisie. Performance Modelling of Deterministic Transport Computations. In *Performance Analysis and Grid Computing*, Kluwer, 2003. 31

[142] V. Taylor, X. Wu, and R. Stevens. Prophesy: an Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications. *SIGMETRICS Perform. Eval. Rev.*, 30(4):13–18, 2003. 31, 32

[143] S.R. Alam and J.S. Vetter. Hierarchical Model Validation of Symbolic Performance Models of Scientific Kernels. In *Euro-Par*, pages 65–77, 2006. 31, 32

[144] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, 1997. 31, 50

[145] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H.Y. Song. Parsec: A Parallel Simulation Environment for Complex Systems. *Computer*, 31(10):77–85, 1998. 32

[146] E. Barszcz, R.A. Fatoohi, V.Venkatakrishnan, and S.K. Weeratunga. Solution of Regular,Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors. Technical Report RNR-93-007, NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035, April 1993. 35, 41

[147] K.R. Koch, R.S. Baker, and R.E. Alcouffe. Solution of the First-Order form of the 3D Discrete Ordinates Equation on a Massively Parallel Processor. *Transactions of the American Nuclear Society*, 65:198–199, 1992. Annual Meeting, Boston, MA. 35, 44, 49

[148] W. Joubert, T. Oppe, R. Janardhan, and W. Dearholt. Fully Parallel Global M/ILU Preconditioning For 3-D Structured Problems. 35, 47, 48

[149] J. Qin and T. Chan. Performance Analysis in Parallel Triangular Solve. In *IEEE Second International Conference on Algorithms and Architectures for Parallel Processing*, pages 405–412. IEEE Computer Society, 1996. 35, 47, 48

[150] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035, January 1991. 41

[151] M.R. Dorr and C.H. Still. Concurrent Source Iteration in the Solution of Three-Dimensional Multigroup Discrete Ordinates Nutron Transport Equations. Technical Report UCRL-JC-116694 Rev 1, Lawrence Livermore National Laboatory, Livermore, CA, May 1995. 44

[152] Accelerated Strategic Computing Initiative (ASCI) Statement of Work, C6939RFP6-3X. http://www.llnl.gov/asci_rfp, February 12 1996. 48

[153] A. Hoisie, G. Johnson, D.J. Kerbyson, M. Lang, and S. Pakin. A Performance Comparison Through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/l, Redstorm, and Purple. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 74, New York, NY, USA, 2006. ACM. 49

[154] S. Kitawaki and M. Yokokawa. Earth Simulator Running. Int.Supercomputing Conference, Heidelberg, June 2002. 49

[155] T. Sato. Can the Earth Simulator Change the Way Humans Think? Keynote address, Int. Conf. Supercomputing, New York, June 2002. 49

[156] Cray XT3 Data Sheet. http://www.cray.com/products/xt3. 60

[157] Cray XT4 Data Sheet. http://www.cray.com/products/xt4. 60, 71

[158] HyperTransport Consortium. http://www.hypertransport.org/. 60

[159] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. In *10th European PVM/MPI Users Group Meeting*, Oct 2003. 66

[160] J.D. Turner. *A Dynamic Prediction and Monitoring Framework for Distributed Applications*. PhD thesis, University of Warwick, Department of Computer Science, 2003. 94

[161] E. Papaefstathiou, D.J. Kerbyson, and G.R. Nudd. A Layered Approach to Parallel Software Performance Prediction: A Case Study. In *Proc. of Massively Parallel Processing Applications and Development*, 1994. 94

[162] J. Cao, D.J. Kerbyson, E. Papaefstathiou, and G.R. Nudd. Modelling of ASCI High Performance Applications Using PACE. In *Proc. UK Performance Engineering Workshop (UKPEW'99)*, pages 413–424, Bristol, July. 97, 103

[163] E. Papaefstathiou, D.J. Kerbyson, G.R. Nudd, T.J. Atherton, and J.S. Harper. An Introduction to the Layered Characterisation for High Performance Systems, December 5, 1997. Research Report CS-RR-335,University of Warwick, Dept. of Computer Science. 98

[164] E. Papaefstathiou. *A Framework for Characterising Parallel Systems for Performance Evaluation*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, U.K, 1995. 99

[165] J.S. Harper. *Analytic Cache Modelling of Numerical Programs*. PhD thesis, University of Warwick, Department of Computer Science, Sept, 1999. 102

[166] G. Karypis and V. Kumar. METIS 4.0: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Technical report, Department of Computer Science, University of Minnesota. 122

[167] The METIS home page. http://www.cs.umn.edu/~metis. 122

[168] G. Johnson, D.J. Kerbyson, and M. Lang. Optimization of Infiniband for Scientific Applications. In *Workshop on Large-Scale Parallel Processing (LSPP), IEEE/ACM Int. Parallel and Distributed Processing Symposium (IPDPS)*, Miami, FL, April 2008. 124

[169] Condor: High Throughput Computing. http://www.cs.wisc.edu/condor/. 129

[170] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998. 129

[171] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, May 2005. ISBN: 0-471-22048-5. 129

[172] Berkeley UPC - Unified Parallel C. http://upc.lbl.gov/. 129

[173] Co-Array Fortran. http://www.co-array.org/. 129

[174] Titanium. http://titanium.cs.berkeley.edu/. 129

[175] Project Fortress. http://projectfortress.sun.com/. 129

[176] Chapel - The Cascade High-Productivity Language. http://chapel.cs.washington.edu/. 129

[177] The X10 Programming Language. www.research.ibm.com/x10/. 129

[178] J.V. Ashby. New Languages for High Performance, High Productivity Computing. Technical Report RALTR2007012, Computational Science and Engineering Department, STFC Rutherford Appleton Laboratory, 2007. 129

[179] The Cell project at IBM research. http://www.research.ibm.com/cell/. 130

A Modelling Contention on CMPs

The following analysis illustrates the derivation of contention terms in Table 4.8. The CMP processors considered here are specific to the Cray XT4 system at the time of this research. As such the interference term I is machine dependant. But the pattern of wavefront operation is machine independent, thus providing insights to a possible bottleneck limiting performance during wavefront application execution on modern CMP processors.

Consider the contention experienced by messages sent by the shaded core. Assuming that there is a 0.5 probability that two messages will collide we can estimate the contention experienced by messages sent by the shaded core as follows:

A.1 Dual Core CMP

- at step r: one extra communication might contend with 0.5 probability with the horizontal communication done by the shaded core: add 0.5I
- at step $r + 1$: one extra communication might contend with 0.5 probability with the vertical communication done by the shaded core: add 0.5I
- at step $r + 2$: no extra communication contend with the horizontal communication done by the shaded core.
- at step $r + 3$: one extra communication might contend with 0.5 probability with the vertical communication done by the shaded core: add 0.5I

Thus a total of $1.5I$ terms should be added where $0.75I$ per $Send_S$ and $Receive_N$. Considering a worst case contention, we use the approximate value of adding one I per $Send_S$ and $Receive_N$

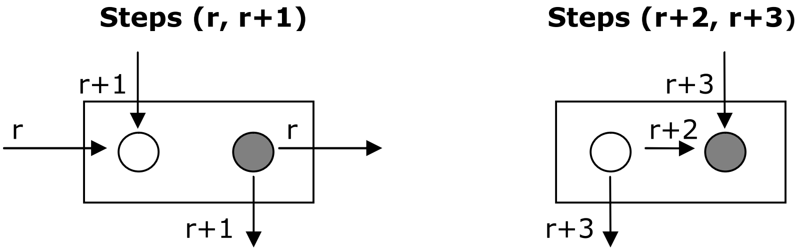


Figure A.1: Wavefront operation and collisions on dual core nodes

A.2 Quad Core CMP

- at step r: 2 extra communications might contend with 0.5 probability with the horizontal communication done by the shaded core: add I
- at step $r+1$: 2 extra communications might contend with 0.5 probability with the vertical communication done by the shaded core: add I
- at step $r + 2$: 2 extra communications might contend with 0.5 probability with the horizontal communication done by the shaded core: add I
- at step $r+3$: 2 extra communications might contend with 0.5 probability with the vertical communication done by the shaded core: add I

$Total = 4I$: i.e. one I per $Send$ and $Receive$

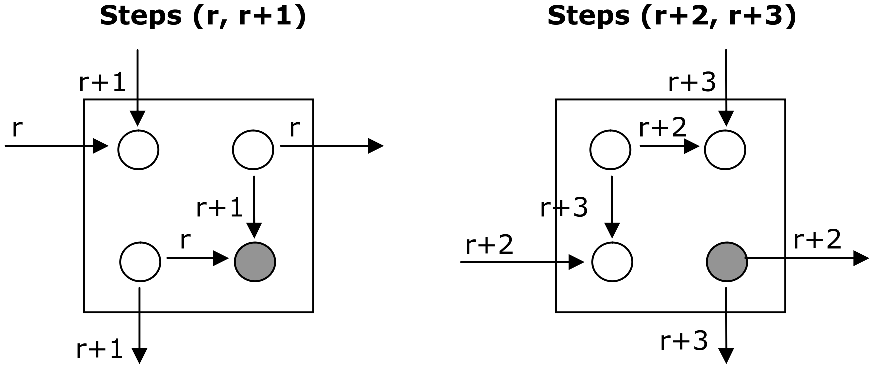


Figure A.2: Wavefront operation and collisions on quad core nodes

A.3 8 Core CMP

- at step r: 5 extra communications might contend with 0.5 probability with the horizontal communication done by the shaded core: add $2.5I$
- at step $r+1$: 4 extra communications might contend with 0.5 probability with the vertical communication done by the shaded core: add $2I$
- at step $r + 2$: 5 extra communications might contend with 0.5 probability with the horizontal communication done by the shaded core: add $2.5I$
- at step $r+3$: 4 extra communications might contend with 0.5 probability with the vertical communication done by the shaded core: add $2I$

$Total = 9I$

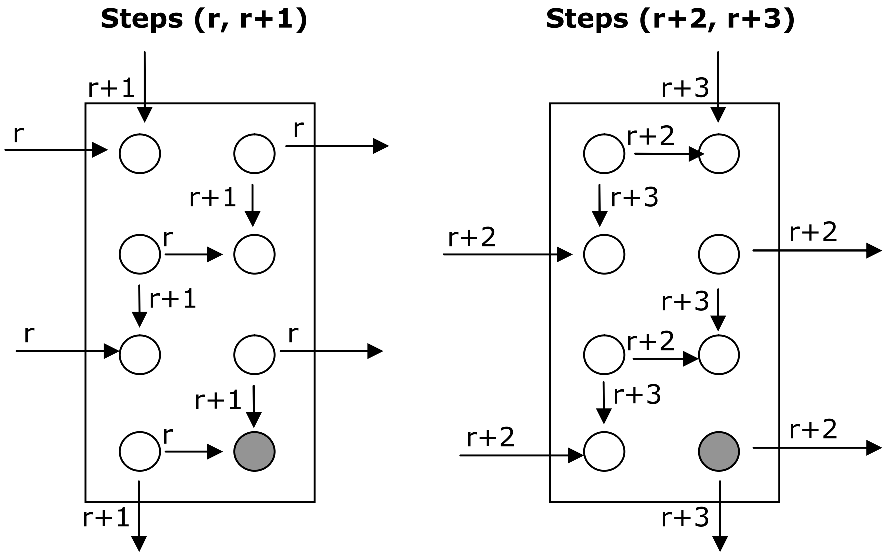


Figure A.3: Wavefront operation and collisions on 8 core nodes

A.4 16 Core CMP

- at step r: 9 extra communications might contend with 0.5 probability with the horizontal communication done by the shaded core: add $4.5I$
- at step $r+1$: 9 extra communications might contend with 0.5 probability with the vertical communication done by the shaded core: add $4.5I$
- at step $r + 2$: 9 extra communications might contend with 0.5 probability with the horizontal communication done by the shaded core: add $4.5I$

- at step $r+3$: 9 extra communications might contend with 0.5 probability with the vertical communication done by the shaded core: add $4.5I$

$Total = 18I$

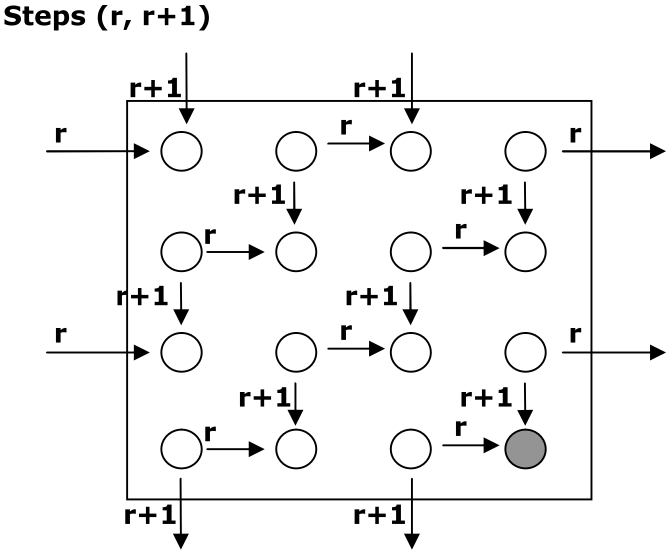


Figure A.4: Wavefront operation and collisions on 16 core nodes

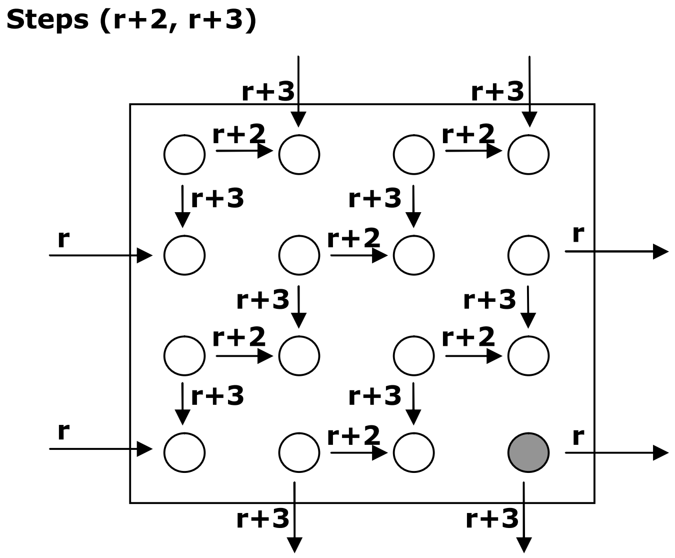


Figure A.5: Wavefront operation and collisions on quad core nodes

B Model Validations

B.1 Chimaera Validations

Table B.1: Chimaera Model Validation on Jaguar (Cray XT4) - 60^3 problem size, $H_{tile} = 1$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	W (Sec)	W_g (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
16	4	4	15	15	$2.09E^{-4}$	$9.29E^{-7}$	48.71	55.27	-11.87	44.67	4.05
64	8	8	8	8	$5.19E^{-5}$	$9.23E^{-7}$	17.22	17.52	-1.72	13.61	3.60

Table B.2: Chimaera Model Validation on Jaguar (Cray XT4) - 120^3 problem size, $H_{tile} = 1$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	W (Sec)	W_g (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
64	8	8	15	15	$1.95E^{-4}$	$8.65E^{-7}$	92.15	109.1	-15.54	83.99	8.15
256	16	16	8	8	$4.87E^{-5}$	$8.66E^{-7}$	33.04	34.9	-5.35	25.78	7.26
1K	32	32	4	4	$1.22E^{-5}$	$8.66E^{-7}$	15.44	18.54	-16.73	7.37	8.07

Table B.3: Chimaera Model Validation on Jaguar (Cray XT4) - 240^3 problem size, $H_{tile} = 1$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	W (Sec)	W_g (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
256	16	16	15	15	$3.47E^{-4}$	$1.54E^{-6}$	320.35	365.15	-12.27	304.11	16.25
1K	32	32	8	8	$8.16E^{-5}$	$1.45E^{-6}$	104.09	110.43	-5.74	89.87	14.22
4K	64	64	4	4	$2.04E^{-5}$	$1.45E^{-6}$	43.33	54.13	-19.96	27.86	15.47

Table B.4: Chimaera Model Validation on a Intel Xeon-InfiniBand cluster - 120^3 problem size

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	W (Sec)	W_g (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
32	8	4	15	30	$1.98E^{-4}$	$4.40E^{-7}$	88.90	107.18	-17.05	83.46	5.44
64	8	8	15	15	$1.00E^{-4}$	$4.44E^{-7}$	47.25	56.72	-16.69	42.73	4.53
96	16	6	7.5	20	$6.64E^{-5}$	$4.43E^{-7}$	35.43	40.89	-13.36	30.97	4.45
128	16	8	7.5	15	$5.52E^{-5}$	$4.91E^{-7}$	30.28	32.56	-7.01	26.17	4.11

Table B.5: Chimaera Model Validation on a Intel Xeon-InfiniBand luster - 240^3 problem size

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	W (Sec)	W_g (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
81	9	9	27	27	$3.73E^{-4}$	$5.1E^{-7}$	324.35	342.33	-5.25	312.17	12.18
96	16	6	15	40	$3.05E^{-4}$	$5.1E^{-7}$	268.57	297.03	-9.58	256.7	11.87
100	10	10	24	24	$2.94E^{-4}$	$5.1E^{-7}$	259.37	278.37	-6.82	247.89	11.49
128	16	8	15	30	$2.30E^{-4}$	$5.1E^{-7}$	205.74	225.65	-8.82	194.82	10.92
169	13	13	19	19	$1.84E^{-4}$	$5.1E^{-7}$	167.83	174.35	-3.74	157.68	10.15
256	16	16	15	15	$1.15E^{-4}$	$5.1E^{-7}$	108.81	129.65	-16.08	99.72	9.09

B.2 Sweep3D Validations

$W_{g,nf}$ - no fixups,$W_{g,f}$ - with fixups

Table B.6: Sweep3D Model Validation on Jaguar (Cray XT4) - 1000^3 total problem size, $H_{tile} = 2, mmi = 6$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	$W_{g,nf}$ (Sec)	$W_{g,f}$ (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
1K	32	32	32	32	$3.16E^{-7}$	$3.69E^{-7}$	36.01	38.56	-6.62	34.99	1.03
2K	64	32	16	32	$3.71E^{-7}$	$4.26E^{-7}$	21.78	24.98	-12.81	20.78	1.00
4K	64	64	16	16	$3.71E^{-7}$	$4.26E^{-7}$	11.78	13.36	-11.83	10.79	0.99
8K	128	64	8	16	$4.14E^{-7}$	$5.00E^{-7}$	7.34	8.43	-12.87	6.42	0.92

Table B.7: Sweep3D Model Validation on Jaguar (Cray XT4) - 20×10^6 total problem size, $H_{tile} = 2, mmi = 6$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	$W_{g,nf}$ (Sec)	$W_{g,f}$ (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
1K	32	32	9	9	$3.58E^{-7}$	$3.89E^{-7}$	1.23	1.38	-10.68	0.99	0.24
2K	64	32	5	9	$4.33E^{-7}$	$4.78E^{-7}$	0.97	1.11	-13.05	0.72	0.25
4K	64	64	5	5	$4.40E^{-7}$	$5.00E^{-7}$	0.73	0.94	-22.41	0.47	0.26
8K	128	64	3	5	$6.00E^{-7}$	$6.33E^{-7}$	0.68	0.90	-23.85	0.41	0.27

Table B.8: Sweep3D Model Validation on Jaguar (Cray XT4) - $5 \times 5 \times 400$ per processor problem size, $H_{tile} = 5$, $mmi = 6$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	$W_{g,nf}$ (Sec)	$W_{g,f}$ (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
4	2	2	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.62	0.59	4.81	0.47	0.15
8	4	2	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.62	0.62	0.06	0.48	0.15
16	4	4	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.63	0.67	-6.1	0.48	0.15
32	8	4	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.64	0.67	-4.79	0.49	0.15
64	8	8	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.65	0.7	-6.16	0.5	0.16
128	16	8	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.67	0.71	-6.47	0.51	0.16
256	16	16	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.7	0.77	-9.17	0.53	0.16
1K	32	32	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.78	0.84	-7.35	0.6	0.18
2K	64	32	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.84	0.95	-11.49	0.65	0.19
4K	64	64	5	5	$4.64E^{-7}$	$5.19E^{-7}$	0.96	1.07	-10.95	0.74	0.21

Table B.9: Sweep3D Model Validation on Jaguar (Cray XT4) - $14 \times 14 \times 255$ per processor problem size, $H_{tile} = 2.5$, $mmi = 6$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	$W_{g,nf}$ (Sec)	$W_{g,f}$ (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
4	2	2	14	14	$3.61E^{-7}$	$3.69E^{-7}$	1.99	1.97	1.17	1.8	0.19
16	4	4	14	14	$3.61E^{-7}$	$3.69E^{-7}$	2.02	2.07	-2.54	1.83	0.2
64	8	8	14	14	$3.61E^{-7}$	$3.69E^{-7}$	2.08	2.19	-5.31	1.88	0.2
256	16	16	14	14	$3.61E^{-7}$	$3.69E^{-7}$	2.19	2.31	-5.27	1.98	0.21
1K	32	32	14	14	$3.61E^{-7}$	$3.69E^{-7}$	2.41	2.68	-10.22	2.19	0.22

Table B.10: Sweep3D Model Validation on Jaguar (Cray XT4) - $20 \times 20 \times 1000$ per processor problem size, $H_{tile} = 5$, $mmi = 6$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	$W_{g,nf}$ (Sec)	$W_{g,f}$ (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
4	2	2	20	20	$3.91E^{-7}$	$4.35E^{-7}$	16.29	16.55	-1.58	15.87	0.43
16	4	4	20	20	$3.91E^{-7}$	$4.35E^{-7}$	16.41	16.75	-2.01	15.98	0.43
64	8	8	20	20	$3.91E^{-7}$	$4.35E^{-7}$	16.65	16.96	-1.85	16.22	0.43
256	16	16	20	20	$3.91E^{-7}$	$4.35E^{-7}$	17.13	17.87	-4.14	16.69	0.44
1K	32	32	20	20	$3.91E^{-7}$	$4.35E^{-7}$	18.09	19.52	-7.33	17.63	0.46

Table B.11: Sweep3D Model Validation on Jaguar (Cray XT4) - $45 \times 45 \times 1000$ per processor problem size, $H_{tile} = 5$, $mmi = 6$

NPE	n	m	$\frac{N_x}{n}$	$\frac{N_y}{m}$	$W_{g,nf}$ (Sec)	$W_{g,f}$ (Sec)	Pred (Sec)	Exec (Sec)	Error (%)	Compute (Sec)	Comm (Sec)
32	8	4	45	45	$3.32E^{-7}$	$3.63E^{-7}$	68.77	69.28	-0.73	68.25	0.52
256	16	16	45	45	$3.32E^{-7}$	$3.63E^{-7}$	71.47	74.23	-3.72	70.93	0.54
512	32	16	45	45	$3.32E^{-7}$	$3.63E^{-7}$	72.82	75.08	-3.01	72.27	0.54
1K	32	32	45	45	$3.32E^{-7}$	$3.63E^{-7}$	75.51	78.33	-3.6	74.96	0.55
2K	64	32	45	45	$3.32E^{-7}$	$3.63E^{-7}$	78.2	80.46	-2.81	77.64	0.57

C cflow `work` from *sweep.x*

Listing C.1: sweep.x

```
(*
 * CHIP3S
 * Application Characterisation Tool
 * Source  : sweep.c
 * RUV Type: clc
 *)
...............
...............
     proc cflow work {  (* Defined at sweep.c:697 *)
[697]    compute <is clc, FCAL>;
[741]    case (<is clc, IFBR>) {
         do_dsa:
[743]        compute <is clc, AILL, TILL, SILL>;
[745]        loop (<is clc, LFOR>, mmi) {
[745]            compute <is clc, CMLL, AILL, TILL, SILL>;
[749]            loop (<is clc, LFOR>, nk) {
[749]                compute <is clc, CMLL, AILL>;
[751]                compute <is clc, AILL>;
[751]                call cflow sign;
[751]                compute <is clc, TILL, SILL>;
[753]                loop (<is clc, LFOR>, it) {
[753]                    compute <is clc, CMLL, 3*ARD3, ARD1, MFDL, AFDL
                            , TFDL, INLL>;
                    }
[749]                compute <is clc, INLL>;
                }
[745]            compute <is clc, INLL>;
            }
         }
[765]    compute <is clc, SILL>;
[765]    loop (<is clc, LFOR>, mmi) {
[765]        compute <is clc, CMLL, ARL1, SILL, INLL>;
         }
[768]    compute <is clc, SILL>;
[768]    loop (<is clc, LFOR>, jt+nk−1+mmi−1) {
[768]        compute <is clc, 4*AILL, CMLL, SILL, TILL>;
[772]        loop (<is clc, LFOR>, mmi−1) {
[772]            compute <is clc, CMLL, 3*ARL1, 2*TILL, AILL, INLL>;
             }
[777]        compute <is clc, 2*AILL>;
[777]        call cflow min;
[777]        call cflow min;
[777]        call cflow min;
[777]        call cflow max;
[777]        compute <is clc, 2*ARL1, 2*TILL, AILL, 2*SILL>;
[800]        loop (<is clc, LFOR>, ndiag) {
[800]            compute <is clc, CMLL, TILL, SILL>;
```

```
[804]         loop (<is clc, LFOR>, mmi−1) {
[804]             compute <is clc, 2*AILL, CMLL, ARL1, TILL, INLL>;
              }
[811]         compute <is clc, 2*TILL, 3*AILL>;
[813]         call cflow min;
[813]         compute <is clc, AILL>;
[813]         call cflow sign;
[813]         compute <is clc, TILL, 3*AILL>;
[814]         call cflow max;
[814]         compute <is clc, AILL>;
[814]         call cflow sign;
[814]         compute <is clc, 3*TILL, 2*AILL, ABSI, 5*ARD1, 2*MFDL
                  , 4*TFDL, ARD3, SILL>;
[840]         loop (<is clc, LFOR>, it) {
[840]             compute <is clc, CMLL, ARD3, ARD1, TFDL, INLL>;
              }
[842]         compute <is clc, SILL>;
[842]         loop (<is clc, LFOR>, nm−1) {
[842]             compute <is clc, CMLL, SILL>;
[844]             loop (<is clc, LFOR>, it) {
[844]                 compute <is clc, CMLL, 2*ARD1, 2*ARD3, MFDL, AFDL
                          , TFDL, INLL>;
                  }
[842]             compute <is clc, INLL>;
              }
[848]         case (<is clc, IFBR>) {
              (−ifixups)/(−epsi):
[855]             compute <is clc, TILL>;
[855]             loop (<is clc, LFOR>, it) {
[855]                 compute <is clc, 4*CMLL, 3*ANDL, 8*ARD1, 8*MFDL
                          , 9*TFDL, 7*ARD3, 9*AFDL, DFDL, AILL, TILL>;
                  }
              1−((−ifixups)/(−epsi)):
[881]             compute <is clc, TILL>;
[881]             loop (<is clc, LFOR>, it) {
[881]                 compute <is clc, 4*CMLL, 3*ANDL, 7*ARD1, 8*MFDL
                          , 8*TFDL, 5*ARD3, 9*AFDL, DFDL, SILL, CMDL>;
[902]                 case (<is clc, IFBR>) {
                      0.5:
[904]                     compute <is clc, 2*AFDL, 4*TFDL, DFDL, 3*MFDL
                              , ARD1, SFDL, CMDL>;
[910]                     case (<is clc, IFBR>) {
                          0.5:
[910]                         compute <is clc, ARD1, MFDL, ARD3, AFDL
                                  , TFDL>;
                          }
[912]                     compute <is clc, CMDL>;
[912]                     case (<is clc, IFBR>) {
                          0.5:
[912]                         compute <is clc, ARD1, MFDL, ARD3, AFDL
                                  , TFDL>;
                          }
[913]                     compute <is clc, SILL>;
                      }
[916]                 compute <is clc, CMDL>;
[916]                 case (<is clc, IFBR>) {
                      0.5:
[918]                     compute <is clc, 2*AFDL, 4*TFDL, DFDL, 3*MFDL
                              , ARD3, ARD1, SFDL, CMDL>;
[924]                     case (<is clc, IFBR>) {
```

```
                            0.5:
[924]                           compute <is clc, ARD1, MFDL, ARD3, AFDL
                                    , TFDL>;
                            }
[926]                       compute <is clc, CMDL>;
[926]                       case (<is clc, IFBR>) {
                            0.5:
[926]                           compute <is clc, ARD1, MFDL, AFDL, TFDL>;
                            }
[927]                       compute <is clc, SILL>;
                        }
[931]                   compute <is clc, CMDL>;
[931]                   case (<is clc, IFBR>) {
                        0.5:
[933]                       compute <is clc, 2*AFDL, 4*TFDL, DFDL, 3*MFDL
                                , ARD3, ARD1, SFDL, CMDL>;
[939]                       case (<is clc, IFBR>) {
                            0.5:
[939]                           compute <is clc, ARD1, MFDL, AFDL, TFDL>;
                            }
[941]                       compute <is clc, CMDL>;
[941]                       case (<is clc, IFBR>) {
                            0.5:
[941]                           compute <is clc, ARD1, MFDL, ARD3, AFDL
                                    , TFDL>;
                            }
[942]                       compute <is clc, SILL>;
                        }
[945]                   compute <is clc, 4*TFDL, ARD1, 2*ARD3, 2*AILL
                            , 2*TILL>;
                    }
                }
[956]           compute <is clc, SILL>;
[956]           loop (<is clc, LFOR>, it) {
[956]               compute <is clc, CMLL, 2*ARD3, 2*ARD1, MFDL, AFDL
                        , TFDL, INLL>;
                }
[959]           compute <is clc, SILL>;
[959]           loop (<is clc, LFOR>, nm-1) {
[959]               compute <is clc, CMLL, SILL>;
[961]               loop (<is clc, LFOR>, it) {
[961]                   compute <is clc, CMLL, 3*ARD3, 2*ARD1, 2*MFDL
                            , AFDL, TFDL, INLL>;
                    }
[959]               compute <is clc, INLL>;
                }
[967]           case (<is clc, IFBR>) {
                do_dsa:
[970]               compute <is clc, SILL>;
[970]               loop (<is clc, LFOR>, it) {
[970]                   compute <is clc, CMLL, 8*ARD3, 4*ARD1, 3*MFDL
                            , 3*AFDL, 3*TFDL, INLL>;
                    }
                }
[981]           compute <is clc, ARD3, TFDL, INLL>;
[987]           compute <is clc, 2*POL1, AILL, TILL, INLL>;
            }
        }
    } (* End of work *)
.............
```

D Wavefront Model and Extensions

D.1 Model Parameters

Table 4.1 Plug-and-Play Reusable Model Application Parameters

Parameter	LU	Sweep3D	Chimaera
N_x, N_y, N_z	$Inputsize$	$Inputsize$	$Inputsize$
W_g	$measured$	$measured$	$measured$
$W_{g,pre}$	$measured$	0	0
$H_{tile}(cells)$	1	$mk \times mmi/mmo$	1
n_{sweeps}	2	8	8
n_{full}	2	2	4
n_{diag}	0	2	2
$T_{nonwavefront}$	$T_{stencil} + \delta_h$	$2T_{allreduce} + \delta_h$	$T_{allreduce} + \delta_h$
$MessageSize_{EW}$ (Bytes)	$40N_y/m$	$8H_{tile} \times \#angles$ $\times N_y/m$	$8H_{tile} \times \#angles$ $\times N_y/m$
$MessageSize_{NS}$ (Bytes)	$40N_x/m$	$8H_{tile} \times \#angles$ $\times N_x/m$	$8H_{tile} \times \#angles$ $\times N_x/m$

D.2 Single Core Model

Table 4.2 Plug-and-play LogGP Model: One Core Per Node, on 3D Data Grids

$W_{pre} = W_{g,pre} \times H_{tile} \times N_x/n \times N_y/m$	(4.2.6)
$W = W_g \times H_{tile} \times N_x/n \times N_y/m$	(4.2.7)
$StartP_{1,1} = W_{pre}$	(4.2.8)
$StartP_{i,j} = max(StartP_{i-1,j} + W_{i-1,j} + Total\_comm_E + Receive_N,$ $StartP_{i,j-1} + W_{i,j-1} + Send_E + Total\_Comm_S)$	(4.2.9)
$T_{diagfill} = StartP_{1,m}$	(4.2.10)
$T_{fullfill} = StartP_{n,m}$	(4.2.11)
$T_{stack} = (Receive_W + Receive_N + W + Send_E + Send_S + W_{pre})N_z/H_{tile} - W_{pre}$	(4.2.12)
$Time\ per\ iteration = n_{diag}T_{diagfill} + n_{full}T_{fullfill} + n_{sweeps}T_{stack} + T_{nonwavefront}$	(4.2.13)

D.3 2D Model

Table 4.7 Plug-and-play LogGP Model for Wavefront Codes on 2D Data Grids

$W_{pre} = W_{g,pre} \times H_{tile} \times N_x/n$	(4.5.1)
$W = W_g \times H_{tile} \times N_x/n$	(4.5.2)
$StartP_1 = W_{pre}$	(4.5.3)
$StartP_i = StartP_{i-1} + W_{i-1} + Total\_comm_E$	(4.5.4)
$T_{fill} = StartP_n$	(4.5.5)
$T_{stack} = (Receive_W + W + Send_E + W_{pre})N_z/H_{tile} - W_{pre}$	(4.5.6)
$Time\ per\ iteration = n_{full}T_{fill} + n_{sweeps}T_{stack} + T_{nonwavefront}$	(4.5.7)

D.4 Extensions for Cray XT3/XT4 CMP Nodes

Table 4.8 Re-usable Model Extensions for CMP Nodes

Modifications to Equation (4.2.9)
For $C_x \times C_y$ cores per node, all communication are off-node except the following: $i\ mod\ C_x \neq 0\ \&\ C_x \neq 1 : Send_E = Send_{onchip,E}$ $i\ mod\ C_x \neq 1\ \&\ C_x \neq 1 : Total\_comm_E = Total\_comm_{onchip,E}$ $j\ mod\ C_y \neq 1\ \&\ C_y \neq 1 : Receive_N = Receive_{onchip,N}$ $j\ mod\ C_y \neq 0\ \&\ C_y \neq 1 : Total\_comm_S = Total\_comm_{onchip,S}$
Modifications to Equation (4.2.12)
For CMPs with a shared bus to memory let $I = (o_{dma} + Message\_size \times G_{dma})$ 1×2 cores/node : add I to $Receive_N$ and $Send_S$ 2×2 cores/node : add I to each $Send$ and $Receive$ 2×4 cores/node : add $9I$ to (4.2.12) 4×4 cores/node : add $18I$to (4.2.12)

D.5 Model Extensions for Simultaneous Multiple Wavefronts

Modifications to Equation (4.2.8)

$$StartP_{1,1} = \eta W_{pre} \tag{7.3.1}$$

Modifications to Equation (4.2.9)

$$\begin{aligned} StartP_{i,j} = max(StartP_{i-1,j} + \eta(W_{i-1,j} + \frac{1}{2}(Total\_comm_E + Receive_N)),\\ StartP_{i,j-1} + \eta(W_{i,j-1} + \frac{1}{2}(Send_E + Total\_Comm_S))) \end{aligned} \tag{7.3.2}$$

$$\eta = \begin{cases} 2, & \text{default} \\ 4, & \text{if } m/2 < i+j \text{ or } n/2 < i+j \\ 8, & \text{if } i+j \geq (m+n)/2 \end{cases}$$

Modifications to Equation (4.2.12)

$$\begin{aligned} T_{stack} = \eta(\frac{1}{2}(Receive_W + Receive_N) + W + \frac{1}{2}(Send_E + Send_S) + W_{pre})\\ (N_z/H_{tile} - (m+n-1)) - W_{pre} \end{aligned} \tag{7.3.3}$$

Modifications to Equation (4.2.13)

$$Time\ per\ iteration = 2n_{full}T_{fullfill} + n_{sweeps}T_{stack} + T_{nonwavefront} \tag{7.3.4}$$

D.6 Model Extensions for Heterogeneous Resources

Let pool of processors with heterogeneous processing times be in the range of $W_1, W_2, W_3, .., W_r, .., W_{max}$ with probability that a processor takes W_r time to complete a block of cells of height H_{tile} given by P_r where $\sum_{r=1}^{max} P_r = 1$.

Modifications to Equation (4.2.7): set W to

$$W_{avg} = \sum_{r=1}^{max} P_r W_r \qquad (7.4.1)$$

Modifications to Equation (4.2.12): set W to W_{max}

D.7 Model Extensions for Irregular/Unstructured Grids

Let the set of all processors be denoted by $\forall P$ and the set of all communication links by $\forall l$.

Modifications to Equation (4.2.12)

$$T_{stack} = \sum_{step=1}^{N_z/H_{tile}} \{\max_{\forall l}(Receive_W) + \max_{\forall l}(Send_E) + \max_{\forall l}(Receive_N) + \max_{\forall l}(Send_S) + \max_{\forall P}(W_{i,j} + W_{pre,i,j})\} \qquad (7.4.2)$$

E Model Parameter Error Propagation

E.1 General Case

As summing that, $Send = \alpha Total\_Comm$ and $Receive = \beta Total\_Comm$, where $0 \leq \alpha, \beta \leq 1$, from (4.2.13), the total runtime for $iter$ iterations of a wavefront application is given by:

$$Total_{iter} = f(W, W_{pre}, Total\_Comm) \tag{E.1.1}$$

Then the variance of f is given by:

$$\sigma_f{}^2 = \left(\frac{\partial f}{\partial W}\sigma_W\right)^2 + \left(\frac{\partial f}{\partial W_{pre}}\sigma_{W_{pre}}\right)^2 + \left(\frac{\partial f}{\partial Total\_Comm}\sigma_{Total\_Comm}\right)^2 \tag{E.1.2}$$

E.2 Error Model for Chimaera

The total runtime (assuming runtime spent in the non-wavefront portions to be negligible) is given by:

$$f = iter(n_{diag}StartP_{1,m} + n_{full}StartP_{n,m} + n_{sweep}T_{stack}) \tag{E.2.1}$$

$$StartP_{1,m} = StartP_{1,1} + (m-1)[W + (1+\alpha)Total\_Comm] \tag{E.2.2}$$

$$\begin{aligned} StartP_{n,m} = {} & StartP_{1,1} + (n-1)[W + (1+\beta)Total\_Comm] + \\ & (m-1)[W + (1+\alpha)Total\_Comm] \end{aligned} \tag{E.2.3}$$

$$T_{stack} = [W + 2(\alpha+\beta)Total\_Comm]\frac{N_z}{H_{tile}} \tag{E.2.4}$$

As W_{pre} for Chimaera is zero, the partial differentials w.r.t W and $Total\_Comm$ is then given by:

$$\frac{\partial StartP_{1,m}}{\partial W} = (m-1) \tag{E.2.5}$$

$$\frac{\partial StartP_{n,m}}{\partial W} = (m+n-2) \tag{E.2.6}$$

$$\frac{\partial T_{stack}}{\partial W} = \frac{N_z}{H_{tile}} \tag{E.2.7}$$

$$\frac{\partial StartP_{1,m}}{\partial Total\_Comm} = (m-1)(1+\alpha) \tag{E.2.8}$$

$$\frac{\partial StartP_{n,m}}{\partial Total\_Comm} = (n-1)(1+\beta)+(m-1)(1+\alpha) \tag{E.2.9}$$

$$\frac{\partial T_{stack}}{\partial Total\_Comm} = 2(\alpha+\beta)\frac{N_z}{H_{tile}} \tag{E.2.10}$$

Components of (E.1.2)

$$\frac{\partial f}{\partial W} = iter[n_{diag}(m-1)+n_{full}(m+n-2)+n_{sweep}\frac{N_z}{H_{tile}}] \tag{E.2.11}$$

$$\begin{aligned}\frac{\partial f}{\partial Total\_Comm} &= iter[n_{diag}(m-1)(1+\alpha)+n_{full}(n-1)(1+\beta)+ \\ &\quad n_{full}(m-1)(1+\alpha)+2n_{sweep}(\alpha+\beta)\frac{N_z}{H_{tile}}]\end{aligned} \tag{E.2.12}$$

For Chimaera $n_{diag}=2, n_{full}=4$ and $n_{sweep}=8$ then

$$\begin{aligned}{\sigma_f}^2 &= iter^2[(6m+4n-10+8\frac{N_z}{H_{tile}})\sigma_W]^2+ \\ &\quad iter^2[(6(m-1)(1+\alpha)+4(n-1)(1+\beta)+ \\ &\quad 16(\alpha+\beta)\frac{N_z}{H_{tile}})\sigma_{Total\_Comm}]^2\end{aligned} \tag{E.2.13}$$

The values of α and β depends on the communication model for the target system.