
Modelling the Performance of Processors in Heterogeneous Computing Environments

Robert Higgins
B.Sc. (Hon) Computer Science

Thesis presented for the degree of
Doctor of Philosophy in Computer Science
to the
School of Computer Science and Informatics
College of Engineering, Mathematical & Physical Sciences
University College Dublin

Research Supervisor
Dr. Alexey Lastovetsky

October 2010

Contents

1	Introduction	1
1.1	Heterogeneity in High Performance Computing Processors	3
1.2	Describing the Performance of Heterogeneous Processors	5
1.3	Data Partitioning and Task Scheduling for Heterogeneous Pro- cessors	8
1.3.1	Task Scheduling	9
1.3.2	Data Partitioning	10
1.4	Functional Performance Model	12
1.5	Predicting Performance Variance	14
1.6	Structure	15
2	The Band Performance Model	16
2.1	Formulation of the Band Model	17
2.1.1	Load History	19
2.1.2	Applying Load History to Functional Performance Model	22
2.1.3	The detailed Band Performance Model	23
2.2	Workload Distribution with Performance Models	26
2.2.1	Angular Metric	31
2.2.2	Constant Probability Metric	32
2.2.3	Piecewise Probability Metric	35
2.2.4	Adjusted Functional Model	36
2.3	Problem Partitioning Experiments	36
2.3.1	Metric Based Partitioning	39
2.3.2	Calculating Execution Time	43
2.3.3	Experiments	45

2.3.4	Summary of Experiments	56
2.4	Summary	57
3	Optimised Construction of the Band Performance Model	58
3.1	Problem Formulation	59
3.2	Assumptions	63
3.3	Definitions	64
3.4	Speed Band Approximation Algorithm	66
3.5	GBBP Experiments	74
3.6	Summary	78
4	Application to Construct Processor Performance Models	79
4.1	Performance Model Manager	80
4.2	Efficient Construction	80
4.2.1	Band of Performance	80
4.2.2	Model Shape	81
4.2.3	Boundary multi-parameter GBBP	83
4.2.4	Diagonal multi-parameter GBBP	84
4.3	Routine Configuration	84
4.3.1	Flexible Construction	86
4.4	Enabling Access and Use of FPM	91
4.5	Summary	91
5	Functional Performance Models in a GridRPC Environment	93
5.1	SmartGridSolve and PMM	94
5.2	Hydropad and PMM	97
5.3	Models and Experiments	99
5.4	Summary	105
6	Conclusion	106

Appendices

A	PMM User Manual	109
A.1	Introduction	111
A.2	Installation	112
A.2.1	Requirements	112
A.2.2	Compiling & Installing	112
A.3	Configuration	114
A.3.1	General Configuration	114
A.3.2	Load Monitor Configuration	115
A.3.3	Routine Configuration	116
A.4	Building the FPM of a Computation	121
A.4.1	Choosing Parameters of a Routine	122
A.4.2	Writing a Benchmark for PMM	123
A.4.3	Configuring the Benchmark in PMM	126
A.5	Viewing Models	128

List of Figures

1.1	Illustration of inter- and intra- cluster heterogeneity.	3
a	Illustration of a Heterogeneous Network of Computers.	3
b	Illustration of interconnected clusters forming a Heterogeneous Meta-Computer.	3
1.2	Illustration of Heterogeneity at the Chip Level and Internal to a Node.	5
1.3	Examples of the HINT benchmark for various Platforms (from [44]).	9
1.4	Examples of Functional Performance Models for Matrix Multiplication on a variety of platforms and using different multiplication implementations.	13

a	Functional Performance Models of optimised and naive square matrix multiplication on heterogeneous CPUs (<i>hcl05</i> and <i>hcl15</i>). Note: naive and optimised performance are plot on separate y axes.	13
b	Functional Performance Models of sub columns of a 4096×4096 matrix multiplication performed on an NVIDIA GTX285 GPU, ATI 4770 GPU and a 4 CPU Intel Xeon (total of 8 cores).	13
2.1	Load observations and the load averages for various periods calculated from them.	21
a	Load observations which are used as the basis for the calculated load averages, LA	21
b	Calculated load averages using a window of 60 minutes over 120 minutes of load observations.	21
2.2	Maximum and minimum load functions extracted from the calculated load average matrix LA	22
2.3	T_i , the ideal execution time of some problem is adjusted for maximum and minimum expected load by finding the intersection of T' and $l_{min}(t)$ or $l_{max}(t)$	24
2.4	Ideal functional performance model $S_i(N)$ and the simple Band Performance Model derived from it.	24
2.5	T' is intersected with projections of calculated load averages from LA	26
2.6	Predicted execution speeds for a single problem size (on x-axis) are binned and plotted in a histogram which forms as the basis for a probability density function describing the processor's speed variation at that problem size.	27
2.7	Graphical representation of partitioning with constant and functional performance models.	29

a	Single benchmarks represent speed as constant across problem size, the partition of workload with such a model may be graphically represented by a line through the origin intersecting both models.	29
b	Partitioning with functional models is equivalent to finding the same line, through the origin that intersects both models at certain problem sizes, summing to the total workload to be partitioned.	29
2.8	Illustration of performance fluctuations which result in optimality for some distribution of workload.	30
2.9	Illustration of the arc of lines through the origin which indicate that a workload is optimal for some combination of performance fluctuations on a set of processors.	33
2.10	Illustration of a Column-wise Naive Partitioning of a Parallel Matrix Multiplication. B is distributed to all processors, w_0 columns of A are distributed to p_0 and elements in corresponding w_0 columns of C are calculated by p_0	38
2.11	Performance profiles of optimised and naive matrix multiplication methods.	39
a	Functional Performance Models of optimised matrix multiplication on platforms used in experiments.	39
b	Functional Performance Models of naive matrix multiplication on platforms used in experiments.	39
2.12	Fitness of metric based distribution in the generations of a genetic algorithm for 16, 8 and 4 processor problem distributions. Note: fitness scales omitted.	44
a	Evolution of Distribution Fitness Based on Angular Metric and Future Knowledge (16 processors).	44
b	Evolution of Distribution Fitness Based on Constant Probability and Piece-wise Probability Metrics (16 processors).	44
c	Evolution of Distribution Fitness Based on Angular Metric and Future Knowledge (8 processors).	44

d	Evolution of Distribution Fitness Based on Constant Probability and Piece-wise Probability Metrics (8 processors).	44
e	Evolution of Distribution Fitness Based on Angular Metric and Future Knowledge (4 processors).	44
f	Evolution of Distribution Fitness Based on Constant Probability and Piece-wise Probability Metrics (4 processors).	44
2.13	Example performance models and load traces used for experimentation.	46
a	Band Performance Models and Adjusted Functional Models for <i>hcl05</i> and <i>hcl15</i> computing columns of a 4000 square matrix multiplication using an optimised routine.	46
b	Example Load Bands and Historical Load Fluctuation Traces used in two Processor Experimentation with <i>hcl05</i> and <i>hcl15</i> .	46
2.14	Performance of partitioning methods using two heterogeneous processors, <i>hcl05</i> and <i>hcl15</i> . <i>hcl05</i> operates under low external load (average load index of 0.1), with low fluctuations (deviation of 0.05). <i>hcl15</i> operates under a high external load (average of 0.8) and moderate load fluctuation (deviation of 0.3).	50
a	Performance of partition methods when using an optimised computational routine.	50
b	Performance of partition methods when using a naive computational routine.	50
2.15	Performance of partitioning methods using two processors, <i>hcl07</i> and <i>hcl01</i> . Processors have identical speed but heterogeneous memory. Both operate under moderate average levels of external load (0.5). <i>hcl07</i> experiences low load fluctuations (deviation of 0.05) and <i>hcl01</i> experiences high fluctuations (deviation of 0.6).	51
a	Performance of partition methods when using an optimised computational routine.	51

b	Performance of partition methods when using a naive computational routine.	51
2.16	Performance of partitioning methods using 4 processors, <i>hcl05</i> , <i>hcl10</i> , <i>hcl11</i> and <i>hcl15</i> . Processors have varied average external loads and load fluctuations.	52
a	Performance of partition methods when using an optimised computational routine.	52
b	Performance of partition methods when using a naive computational routine.	52
2.17	Performance of partitioning methods using 8 processors, <i>hcl01</i> to <i>hcl08</i> . Processors have varied average external loads and load fluctuations.	53
a	Performance of partition methods when using an optimised computational routine.	53
b	Performance of partition methods when using a naive computational routine.	53
2.18	Performance of partitioning methods using 16 processors, <i>hcl01</i> to <i>hcl16</i> . Processors have varied average external loads and load fluctuations.	54
a	Performance of partition methods when using an optimised computational routine.	54
b	Performance of partition methods when using a naive computational routine.	54
3.1	Using piecewise linear approximation to build speed bands. Circular points are experimentally obtained, square points are calculating using heuristics.	60
a	The speed band of a problem which uses the memory hierarchy inefficiently (using 5 experimental points). . .	60
b	The speed band of a problem which uses the memory hierarchy efficiently (built using 8 experimental points). .	60
3.2	Illustration of how a piece-wise function approximates a band. .	62

a	The speeds $S_{max}(x)$ and $S_{min}(x)$ representing a cut of the real band used to build the piece-wise linear approximation.	62
b	A piece-wise linear approximation built by connecting the cuts.	62
3.3	The non-empty intersectional area of a piece-wise linear function approximation of the real-life speed band is a simply connected surface.	63
3.4	The two general shapes of the real-life band.	65
a	Shape of real-life speed band of a processor for a routine that uses the memory hierarchy inefficiently.	65
b	Shape of a real-life speed band of a processor for a routine that uses the memory hierarchy efficiently.	65
3.5	Illustrations of steps of the Geometric Bisection Building Procedure.	70
a	Initial approximation of the speed band.	70
b	Steps in the approximation of any performance increase at the start of the building procedure.	70
3.6	Further Illustrations of steps of the Geometric Bisection Building Procedure.	71
a	Finalising construction where a region of higher constant performance is detected.	71
b	Finalising construction where a region of lower constant performance is detected.	71
3.7	Construction cannot be finalised in either left or right intervals as the performance is neither constant nor already approximated by the model.	72
3.8	Illustrations of possible shapes of the model when the speed at a new bisection point is already approximated.	73
a	Possible shape of real-life band: $C(x_{b1})$ and regions to the left or right are all accurately approximated, i.e. there is no inflection in the model.	73

b	Possible shape of real-life band: $C(x_{b1})$ is accurately approximated but regions to the left and right are not, i.e. there is inflection in the model.	73
3.9	Experiments showing the constructed Band Approximations realised by the Geometric Bisection Building Procedure for a variety of routines on X1.	76
a	Speed Band for Cholesky Factorisation on X1.	76
b	Speed Band for Naive Matrix Multiplication on X1.	76
c	Speed Band for optimised ATLAS Matrix Multiplication on X1.	76
3.10	Experiments showing the constructed Band Approximations realised by the Geometric Bisection Building Procedure for a variety of routines on X2.	77
a	Speed Band for Cholesky Factorisation on X2.	77
b	Speed Band for Naive Matrix Multiplication on X2.	77
c	Speed Band for optimised ATLAS Matrix Multiplication on X2.	77
4.1	Load observations and the load averages for various periods calculated from them.	82
a	Typical profile of Band Performance Models according to their memory access efficiency.	82
b	Band Performance Model for <code>barmatter</code> routine with points used by GBBP and a naive construction algorithm shown.	82
5.1	Illustration of Scheduling Transactions in GridSolve.	95
5.2	Task Graph of Hydropad Application.	98
5.3	Graph of <code>darkmatter</code> Functional Performance Model with relative performance highlighted.	101
5.4	Detail of problem parameters where paging contributes to sudden performance decrease in <code>darkmatter</code> problem.	102
5.5	Functional Performance Models for the <code>barmatter</code> routine, with points for GBBP and a naive construction method shown.	103

A.1	Example output from the <code>pmm_view</code> utility.	130
a	Example output from <code>pmm_view</code> showing a 2 parameter model for a naive matrix multiplication on a machine with 256MiB of memory which has been constructed with naive method.	130
b	Slice of the previous model where the second parameter is fixed at 3000, as displayed by the <code>pmm_view</code> utility.	130

List of Tables

2.1	Machine specifications for Band Model experiments.	55
3.1	Machine specifications for GBBP experiments.	74
3.2	Results of experimentation with GBBP.	75
a	Speedup of GBBP over naive construction procedure for different applications.	75
b	Number of points used by GBBP in construction of models for different applications.	75
5.1	SmartGridSolve Server Configuration.	100
5.2	Time Spent Construction Functional Performance Models.	104
5.3	Scheduling Improvements with FPMs in SmartGridSolve.	105

Abstract

Accurate performance models of processors are essential for efficient heterogeneous parallel or distributed computing. Characterising the performance of a processor at a particular operation is a challenge. Simple models that are easy to construct and use do not represent enough of the detail of a processor's performance to provide high efficiency in all conditions. However, detailed models are challenging to construct and more difficult to use in a data partitioning algorithm or task scheduler.

In this thesis the construction and use of a detailed performance model is investigated. This model is titled: the Band Performance Model (BPM). It is an evolution of the Functional Performance Model (which expresses the speed of a processor as a function of the size of a specific task that it operates on). The BPM describes the processor speed not as a single valued function but as a function with a range of possible speeds based on a prediction of CPU availability. In this way the model encapsulates inherent variability in the performance of the processor. Variability which is caused by the concurrent execution of other processes.

The Band Performance Model is formulated and a number of methods are described that use the model for partitioning of problems. The efficiency of the partitions is demonstrated. The cost of construction of such a model is addressed by a novel optimised building procedure. This procedure is implemented in a software tool which is demonstrated in detail. The tool is used to build performance models for a non-synthetic distributed application and these models are integrated with a Grid middle-ware, where significant performance increases are shown as a result of their use.

Acknowledgements

I would first like to thank my supervisor, Dr. Alexey Lastovetsky, for his guidance, support and flexibility through the duration of my studies. I would also like to express my gratitude to the School of Computer Science; to its head, Dr. Joe Carthy, for providing assistance towards the completion of my thesis, and to the administrative and support staff who have always been readily available, helpful and a pleasure to seek assistance from.

I am grateful for the funding provided to me by IBM, the Irish Research Council for Science, Engineering and Technology, and the School of Computer Science and Informatics.

To my colleagues of the Heterogeneous Computing Laboratory, Ravi Reddy, Brett Becker, Michele Guidolin, Thomas Brady, Vladimir Rychkov; I thank you for amongst other things: the academic collaboration, good company, encouragement and advice.

I thank my friends and neighbours from Space Science (and elsewhere) for many pleasant, and often extended, lunch-breaks: Gary, Sophie, Suzanne, John and Sinead. Garrath and John from Computer Science for motivation, discussion and especially tea.

Thank you to my family for supporting me constantly. And finally, I thank Gaby for her patience through it all, for giving me reason to finish and helping me get to the end.

Chapter 1

Introduction

Heterogeneous parallel and distributed computing platforms constitute a wide and increasingly popular range of high performance computing resources. Efficiently utilising these platforms is significantly more challenging when compared to using homogeneous computing platforms. Despite this, the current evolution of systems is tending towards more widespread use of heterogeneous processor of various sorts. The challenge of scaling performance of clusters is being met by the development massively parallel streaming multiprocessors, reconfigurable computing and heterogeneous multi-core processors.

The focus of this thesis is on performance models which enable efficient problem partitioning or task scheduling in heterogeneous parallel computing environments. The research conducted builds on the Functional Performance Model, which describes the performance of a processor, executing a specific problem, as a function of the problem's size. This kind of detailed performance model has been found necessary as a general solution to optimally utilising heterogeneous processors. However, though it considers the performance of a processor as variable with problem size, it does not encapsulate a measure of performance variance as a result of external load fluctuations on the processor. Representing the performance variation of non-dedicated processors is important in highly heterogeneous environments such as desktop grids or Networks of Workstations (NOWs).

The Band Performance Model (BPM) described in this thesis represents the

performance of the single valued points from a base FPM with a range possible speeds given by some probability distribution. This represents speed variance due to both changing problem size and external load fluctuations that may occur on the processor. The adjustment is made by taking the execution time of the original point in the FPM and making a prediction of the future average load over a similar period of time. The prediction is based on a history of load fluctuations recorded on the processor. The formulation of the BPM is described and methods to evaluate the distribution of some workload using the BPM are proposed. These evaluation methods are used to choose appropriate workload distributions and the performance of the distributions is compared.

The Functional Performance Model is a generalised solution to the problem of heterogeneity in a parallel computing environment, it makes no assumptions about the problem it is to be used to partition or the processor it is to represent. As a result, to find the FPM of a processor and problem, that problem must be extensively benchmarked. This benchmarking must be carried out on every heterogeneous node in a network, and further, the model cannot be used to represent the speed of any other problem. Such efforts to build the FPM are expensive and limit its application. In order to address this, a novel algorithm, titled Geometric Bisection Building Procedure (GBBP) is introduced. GBBP uses the natural variation in performance to build an approximation of Band Performance Model. It minimises the number of experimental points that must be benchmarked to accurately approximate the shape of the BPM. The Functional Performance Model can be seen as a product of a BPM, where the range of speeds described by the band is reduced to some average single value. In this way, the GBBP algorithm builds the FPM also.

Further, this algorithm is implemented in a GPL Licenced tool. The purpose of the tool is to allow for flexible construction and experimentation with a large set of FPMs and BPMs. It provides a number of construction methods for building models, including GBBP, a framework for configuring the benchmarking of a problem and a simple and flexible interface to the problem benchmark.

The last contribution made in this thesis is the integration of a FPM in the scheduler of a GridRPC system. By providing accurate estimations of execution times based on actual benchmarks, the FPM can significantly improve the perfor-

mance of a task scheduler. This is demonstrated for a challenging non-synthetic application which uses the GridRPC framework.

1.1 Heterogeneity in High Performance Computing Processors

Heterogeneity is an increasingly common attribute of high performance computing resources. Networks of Workstations have provided parallel computing capabilities without the significant cost of acquiring a dedicated cluster resource for many years. The capacity of NOWs to take on problems that previously required expensive vector or massively parallel processors has been demonstrated in [1] and elsewhere. Non-dedicated NOWs were shown to have ample capacity to provide high performance computing power in papers such as [2] and [3] has shown that non-idle, shared systems may be used as processing resources in a parallel computation. Such work has demonstrated that ad-hoc, non-dedicated Heterogeneous Networks of Workstations are viable parallel computing platforms and a large body of research in adapting applications for this platform has been developed.

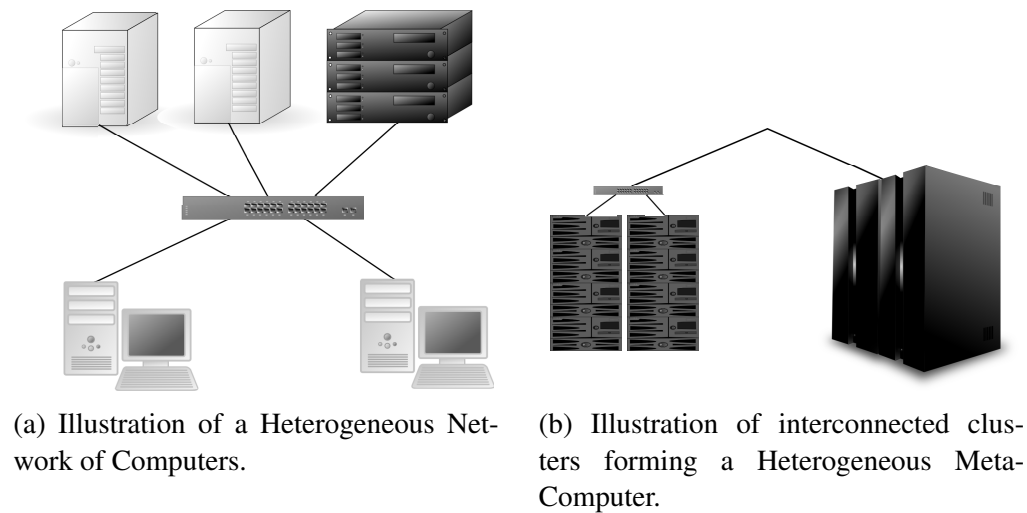


Figure 1.1: Illustration of inter- and intra- cluster heterogeneity.

On a larger scale, inter-connected clusters have formed heterogeneous meta-

computing resources in networks such as NorduGrid [4], Grid'5000 [5], TeraGrid [6], Open Science Grid [7] and so on. Software such as PAX-MPI [8], and more recently MPICH-G2 [9] (built on top of Globus [10]) has enabled applications to pool the resources of many heterogeneous clusters in performing computations. Use of resources in this manner has been studied and found practical, even for tightly coupled applications such as parallel matrix multiplication in [11, 12, 13].

Further, the Cell platform [14] and Hybrid CPU-GPU computing [15, 16] result in a level of processing heterogeneity inside nodes of an otherwise homogeneous cluster. Using the processing power of both CPU and GPU at the same time requires considering the heterogeneous nature of the two resources. In [17] matrix multiplication has been successfully decomposed between a single CPU and GPU, [18] carries out a similar study on the parallelisation of Fast Fourier Transforms on GPU and CPUs, building on this [19] demonstrates a framework for collaborative computation on GPUs and CPUs. In [20] the entire TSUBAME super-computer is benchmarked using LINPACK. This is a highly heterogeneous computing environment comprising of a pair of clusters. Nodes of the first cluster contain Opteron multi-core CPUs, NVIDIA GPUs and Clear-speed accelerators. The nodes of the second cluster are homogeneous, using Xeon multi-core CPUs. The pair of clusters are linked by a 200Gb/s connection and the LINPACK benchmark was run to use all resources available.

Finally, heterogeneous multi-core and heterogeneous system-on-chip (SoC) platforms take heterogeneity in processors to the lowest level of system architecture. As multi-core processors increase core count dramatically, the speed-up they achieve at typical end-user workloads is limited by Amdahl's law [21]. Such workloads often have a low degree of parallelism and suffer as a result. Given the same power envelope and real-estate on a chip, better performance can be achieved by mixing few large, powerful cores (for sequential workloads) with many smaller, less powerful cores for parallel workloads [22, 23]. The result is a set of cores with a single instruction set but heterogeneous performance [24]. Also, SoCs that integrate massively parallel processors and multi-cores on the same die have been developed [25, 26], and reconfigurable Field Programmable Gate Arrays (FPGAs) have been integrated with general purpose processors on

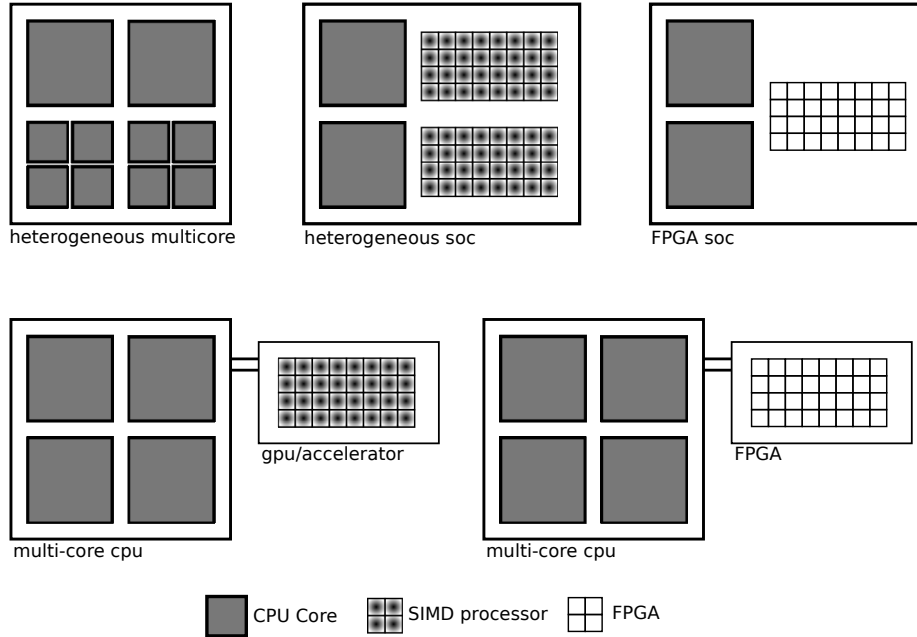


Figure 1.2: Illustration of Heterogeneity at the Chip Level and Internal to a Node.

a single die or package [27, 28, 29].

While not all of these developments are targeted directly at the highly scalable parallel computing domain, since the era of NOWs clusters have been built from a wide array available processors and components. The emerging heterogeneity of these processors, at the very lowest level of a computer’s architecture, will result in more difficult decomposition of parallel workloads on any system using them [30].

1.2 Describing the Performance of Heterogeneous Processors

It is clear that heterogeneity in the high performance computing domain is increasing as new architectures emerge and it is accepted that heterogeneity will be a feature of future high performance computing platforms [31]. Forms that processor heterogeneity have taken and may take in the future are summarised in

the previous section.

The performance of a processor, or entire systems, has been represented in many ways for the purposes of both high level comparison of systems and solving task scheduling or data partitioning problems.

The simplest measure of a processors performance is perhaps the theoretical peak FLOPS (Floating-point Operations Per Second). This is calculated from the number of cycles required by the floating point unit of a processor to execute a floating point calculation. The peak is rarely achieved by any application or benchmark as a result of latencies in fetching data for the floating point unit to operate on.

As an alternative to theoretical estimations, benchmarks may be executed to characterise the performance of a processor. Frequently FLOPS or MIPS (million instructions per second) are measured and characterise the performance of a processor. MIPS is derived from the running time of the Dhrystone benchmark [32], which executes a set of common programming constructs. In order to accurately measure the raw number of instructions that can be issued by the processor, the benchmark defeats compiler optimisations, omits any floating point calculations and fits in the highest level of cache (and entirely in the instruction cache of modern CPUs). FLOPS is frequently calculated using the LINPACK [33] benchmark. This solves a fixed number of dense linear equations, where the dominant operations are floating point (matrix multiplication). The LINPACK benchmark has had to change the size of problem it solves as cache sizes have grown over time. This is an indication of an inherent problem with a single sized benchmark: that they do not measure the processor's speed as it executes on data in each of the many levels of the memory hierarchy. This hierarchy can range from L1-L3, even L4¹ caches, physical RAM and virtual memory. Both MIPS and FLOPS are simplistic representations of a processors performance, they are measured using highly synthetic benchmarks and they often do not relate well with the performance of actual applications. Despite this, FLOPS is frequently used as a comparator of heterogeneous processors.

At the cluster level, a parallel LINPACK benchmark such as High Perfor-

¹IBM eX4 chip-set supports 256MB off-chip L4 cache for Xeon processors, and the IBM z196 processors are attached to an interconnect hub with 96MB of L4 cache

mance LINPACK (HPL) [34] is used to calculate the FLOPS attained by collections of processors solving large sets of linear equations. The highest FLOPS attained by a cluster for an arbitrary sized LINPACK problem (the largest size that fits in memory usually results in the highest speed) is used to rank the fastest super-computers in the in the Top500 [35] list. LINPACK as a benchmark suffers the criticism that it does not stress the communication links of a cluster appropriately, and much like MIPS and FLOPS, that it does not relate to the speed of a platform executing a real problem.

The SPEC [36] suite of benchmarks attempts to address this deficiency for comparison of individual nodes. It is a set of a wide variety of application benchmarks which allow vendors to describe the performance of their systems at a variety of problems. Similar suites, such as the NAS Parallel Benchmarks [37, 38] and DEISA [39] serve the same purpose but for clusters. Such benchmark suites are mostly used to evaluate the performance of systems at certain general types of application. In some cases they have been used to forecast performance of specific applications [40, 41], however they are not used for prediction of performance for task scheduling or data partitioning problems.

One property that is common to the benchmarks mentioned thus far is that they all represent the speed of a processor as single point values. Even for suites of benchmarks, the collection of data points ultimately only reveals a number of single perspectives of a processors performance, not a global view of performance. The reality is that a processor's performance varies with the size of problem it operates on, as a result of differing latencies and bandwidth when accessing the various levels of memory hierarchy. This is key to the problem of partitioning data, as a processors speed may change depending on the amount of work you give it. Also when scheduling a task, the performance of a processor may vary depending on input parameters of the task, and choosing the correct schedule requires knowledge of this relationship.

The STREAM benchmark [42] is focused on the performance of a processor's memory system. It measures a set of various operations on main memory rather than floating point calculations. STREAM2 [43] is an extension which explores the performance of a processor at all levels of the physical memory hierarchy benchmark. It is of interest as it represents the decline in bandwidth

available to feed the processor data as a problem size increases across cache and memory boundaries. This decline is directly related to the decline in speed of a processor when computing a task that is data-starved.

The final benchmark to mention in this section is HINT [44]. This is a scalable benchmark that provides a combination processor performance and the performance of a memory hierarchy in a single representation. HINT is a synthetic benchmark based around a synthetic kernel. The kernel performs a hierarchical integration of some function, with increasing detail as the kernel is iterated. It provides a measurement of Quality Improvement Per Second (QUIPS). This relates to the rate at which the integration becomes more accurate. In the initial iterations, the overall problem size is small as the integration has a low level of detail. The integration quality is improved at a high rate in this initial phase. As execution time increases so too does the overall quality and the amount of data that comprises the problem. This results in a gradual decline in the refinement rate of the integration. The QUIPS function can be plot against overall execution time or bytes of data in the integration and it is shown in Figure 1.3. HINT is interesting as it bears a resemblance to the model of a processor that is developed in this thesis, though HINT was not the starting point of this work.

1.3 Data Partitioning and Task Scheduling for Heterogeneous Processors

In order to utilise various types of heterogeneous processors in parallel, via data partitioning or task scheduling, models of the performance of each processing element must exist. With the nature of processing units becoming more and more diverse, models of performance must have greater detail and must be more general to allow for accurate solution of a data partitioning or task scheduling problem. Existing general performance measures of various types have been summarised in the previous section.

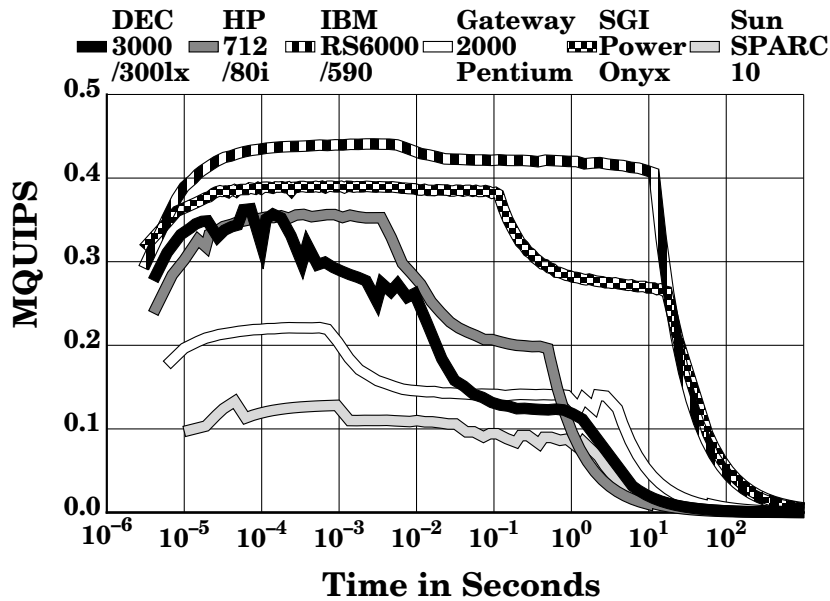


Figure 1.3: Examples of the HINT benchmark for various Platforms (from [44]).

1.3.1 Task Scheduling

In the study of task scheduling algorithms for heterogeneous processors it is assumed that the Expected Time to Compute (ETC) of a task on a node in a network is known [45, 46, 47, 48, 49, 50]. Estimating the ETC is a branch of research itself and it is this area of estimation that we work on.

There are a number of approaches for estimating the ETC. In [51] static analysis of code is used to build the cost function of a procedure. This function is in terms of the procedure's argument sizes and allows for run-time prediction of the procedures cost. Such code analysis has a limited application, to certain languages and architecture types, thus they are not suitable for heterogeneous computing in general [49].

Combining code analysis with platform benchmarking is another approach. The PAWS project [52] provides an environment to evaluate the performance of an application on different hardware without the need to execute it. It characterises the functional elements of a computer by benchmarking those elements (processors, memory, I/O) and characterises the application as a set of procedures in data-flow graph. The execution of the application is simulated on the

functional elements of the computer to provide a prediction of ETC. [53] proposes a framework for characterising an application using a set of base templates and characterising a computer using a set of analytical benchmarks of those base templates. The combination of these elements allows for a prediction of execution time of an application on a computer. In general, the scheme is to benchmark basic operations of the computer and profile the application to identify how it executes that set of basic operations, then predict execution time using both models. Such systems can accurately predict the relative performance of heterogeneous systems but their prediction of actual ETC is not always reliable.

Finally, statistical prediction of ETC based on previous executions of a problem are often used. These methods build a model of performance based on an analysis of previously timed executions, they are expected to sit along side a scheduler and be continually fed with more observations to refine their prediction as new tasks are executed. Examples of such methods may be found in [54, 55, 56]. In [57] a hybrid of analytical benchmarking and statistical prediction is used so that task execution times may be shared between similar systems. FAST [58] performs statistical prediction of ETC seeded with an initial set of task benchmarks. The tasks are benchmarked using an initial set of typical problem sizes. This gives the statistical prediction an initial accuracy. Subsequent scheduled tasks are timed and used to improve the prediction of ETC further.

1.3.2 Data Partitioning

In data partitioning on heterogeneous parallel computing the typical representation of performance that is used is processor weight, relative speed, or normalised speed of the set of processors. These are essentially the same thing. Using this measure, one can proportionally divide a workload between processors and expect that they will complete the computation of their respective work at the same instant. Often data partitioning algorithms assume that the relative speed is provided [59, 60, 11, 61]. Otherwise suggestions are made for the calculation of relative speed, in [62] a system for analysing the performance of different data partitioning methods, it suggests a test program is executed on nodes of the network to provide a FLOPS rating for each node. In [12] a small test calcula-

tion is executed on nodes before decomposition to give a measure of speed. In some cases, no performance model of processors is used and decomposition is achieved through manual tuning and experimentation [19, 20].

The relative performance model ultimately represents the speed of the processor as constant for all problem sizes that may be assigned to it, however it is clear that the performance of a processor is a function of the problem type it computes, the size of data that it executes on it and naturally, the underlying architecture of the processor.

Firstly, a processor's speed is dependent on the type of problem it solves, for this reason benchmark suites attempt to encompass as many types of problem possible and estimators of ETC attempt to identify the type of problem by code analysis. In heterogeneous data partitioning deriving relative speed from some fixed benchmark or measure of processor performance is inadequate, this has been identified in [63]. The mpC language [64] addresses this problem by allowing the programmer to define a small characteristic benchmark routine to measure the relative speeds of processors before work is partitioned. These speeds are input to a user defined model of the parallel computation to help the programmer to optimally utilise the heterogeneous resources. HeteroMPI [65] implements the same techniques in mpC, along with a data partitioning API, as an extension to MPI.

Second, the processor's speed is a function of the problem size it operates on. This has been demonstrated by the HINT benchmark [44] and in [66], where a performance model for a mixed in-core and out-of-core problem is described. Describing the variation in performance between paging and not-paging is not only an issue for out-of-core processing. As processor architectures become more heterogeneous, their performance as problem size increases becomes more difficult to approximate and this necessitates a more detailed general model of performance than relative speed provides.

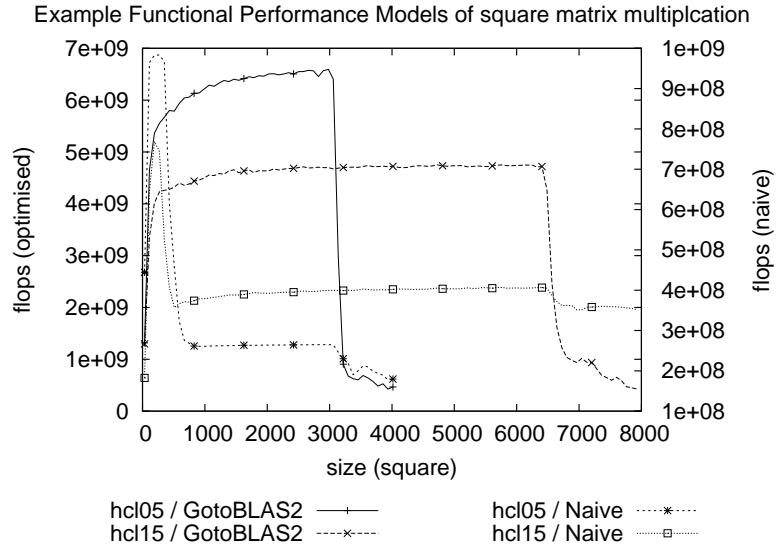
In order to address these issues the Functional Performance Model has been proposed in [67]. This model describes the speed of a processor at a specific task in terms of the input parameters to that task. The construction of this model and an extension to it are the subject of this thesis.

1.4 Functional Performance Model

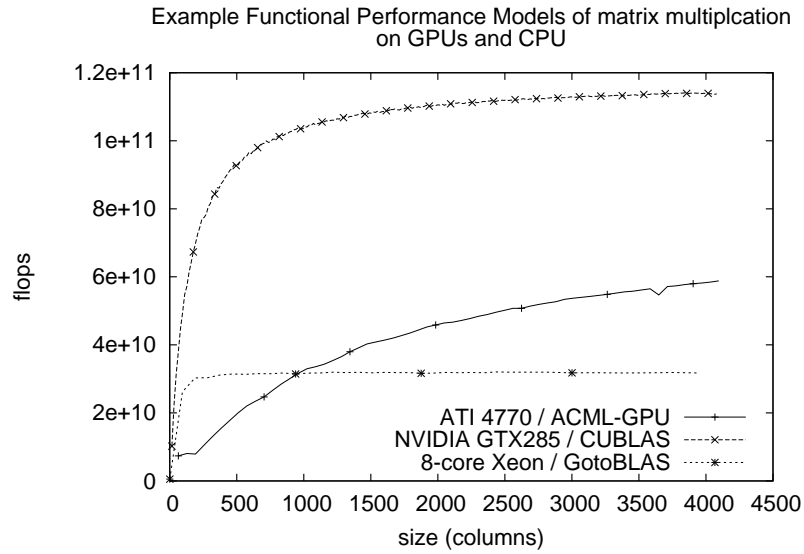
The Functional Performance Model (FPM) is a model that accurately describes the speed of a processor, at a specific problem, as a function of the size of problem it computes. Performance varies with problem size as the processor may speed up through more efficient use of its caching hierarchy, or slow down through greater use of deeper levels of the memory hierarchy. The extent of these changes is dependent on the implementation of the problem that is to be computed. An inefficiently implemented computation will slow down continually as the data it operates on is fetched from deeper levels of the memory hierarchy, while an optimised computation may have a relatively flat performance function, until paging occurs. Further, the computation on widely different architectures may result in differently shaped functions.

Examples of the FPM for a matrix multiplication are given in Figure 1.4, they illustrate the wide variation in the performance functions of different matrix multiplication methods on various platforms. Figure 1.4a shows naive and optimised multiplication on two different platforms *hcl05* and *hcl15* (specifications may be viewed in Chapter 2, Table 2.1, page 55). Optimised multiplication is performed by GotoBLAS2. First, one can see that the naive method varies across the full range of problem sizes, while the optimised method has a constant performance profile for a significant proportion of its profile, until paging occurs. Figure 1.4b shows FPMs for multiplication on CPU and GPUs. CUBLAS [68] and AMCL-GPU [69] libraries are used to perform matrix multiplication on a NVIDIA GTX285 and an ATI 4770. Both of these are compute capable consumer grade GPUs. The CPU used is a four processor machine, totaling eight Xeon cores, listed as *yeats* in the specification table referenced above. Comparing the GPUs and CPU profiles, there is significant difference in how the performance of each type of processor increases and then plateaus as problem size gets larger. The CPU is almost instantaneously reaching its peak, while the GPUs have a more gradual climb.

Accurately partitioning a problem or scheduling a task on such platforms requires detailed knowledge of the performance of the platforms. The FPM is a general solution to providing this performance information. Application of



(a) Functional Performance Models of optimised and naive square matrix multiplication on heterogeneous CPUs (*hcl05* and *hcl15*). Note: naive and optimised performance are plot on separate y axes.



(b) Functional Performance Models of sub columns of a 4096×4096 matrix multiplication performed on an NVIDIA GTX285 GPU, ATI 4770 GPU and a 4 CPU Intel Xeon (total of 8 cores).

Figure 1.4: Examples of Functional Performance Models for Matrix Multiplication on a variety of platforms and using different multiplication implementations.

the FPM in various scenarios has resulted in increased performance when using heterogeneous resources. This has been demonstrated for data partitioning [67], task scheduling [70] and dynamic loop scheduling [71]. Constructing the FPM has been described in [72, 70].

1.5 Predicting Performance Variance

Much work exists in the area of predicting processor performance variability. Statistics provided by the operating system kernel and their ability to describe CPU availability have been analysed at length in [73], the average queue length of processes executing or waiting to execute on the CPU, was found to be more efficient at representing the current performance of the CPU than a percentage availability measure. Average queue length is often reported by UNIX kernels as the “load average”, with one, five and fifteen minute periods. In this study a four second averaging period was found best but one minute periods also performed well.

In [74] various load descriptions are used to balance tasks in a system, and again the average queue length was found to be a better predictor of a CPUs current performance. Both these studies focused on instantaneous load descriptions of processors. This is because the tasks to be assigned to those processors are some short lived distributed processes, and future load fluctuations are not of concern.

The Network Weather Service (NWS) [75] is a reliable predictor of short and medium term load CPU availability. It uses a combination of load statistics gathered from continuous measurement of load average, CPU availability and the results of a probing process which are input to an autocorrelation model to provide accurate performance forecasts on non-dedicated platforms. The WINNER system [76], also uses a hybrid of measures to predict CPU availability in its performance model. NWS itself is a component of a number of larger systems such as Globus [77] and Condor [78].

In [79, 80] load traces are used to evaluate various linear time series models for the prediction of future load between a one and thirty second interval. Here, the traces are used both for fitting the linear time series models and verifying the

accuracy of the predictions based on those models. They show that load traces can be reliably used to predict future load in the time windows they chose to study.

Studies such as [81] propose a method to balance a workload across a set of non-dedicated platforms based on their average predicted execution performance. They have shown that non-dedicated platforms do have the capacity to provide high performance in parallel computing provided their accurate measures of external utilisation and that the utilisation does not have very high variance.

1.6 Structure

This thesis proceeds as follows: The Band Performance Model is introduced and formalised in Chapter 2. Experimentation with the Band Model in a data partitioning problem is presented. In Chapter 3 an algorithm for the optimised construction of both Band and Functional Performance Models is presented. The optimised construction is shown to be significantly faster than a naive method, and this is key to improving the practicality of the FPM and BPM. Chapter 4 presents a tool that implements this optimised algorithm and a number of other utilities for building, using and viewing FPMs and BPMs. This tool is titled the Performance Model Manager (PMM). Chapter 5 describes the integration of detailed performance models with the task scheduler of SmartGridSolve [82], and the resulting speed-up achieved in a real-life application. After the conclusion in Chapter 6, the user manual of the PMM tool is attached as an appendix.

Chapter 2

The Band Performance Model

Networks of Workstations (NOWs) provide high performance parallel computing capabilities without the significant cost of acquiring dedicated cluster resources. NOWs are typically built from a wide variety of nodes that are *in situ* on a campus, in an office or belonging to some general purpose network. The degree of heterogeneity in NOWs is often high and applies to many properties of the nodes. Their processing capacity may vary greatly and will be determined by features such as: memory bus, memory capacity, CPU clock, CPU architecture, cores, and so on. As described in Chapter 1, the Functional Performance Model (FPM) accurately predicts the performance of heterogeneous processors allowing optimal distribution of workloads between dedicated heterogeneous processors.

However, some nodes may not be fully dedicated to the computational tasks of the NOW, they may have varying levels of integration with the wider network to which they belong. Desktops, servers and other computers may be volunteered to the NOW, offering their spare resources to use for computational tasks. They will experience load fluctuations according to their primary role. As a result their processor availability and subsequent performance may vary. The Band Performance Model (BPM) which is presented in this chapter is an extension to the FPM where the variability in processor speed is represented as a range of performance rather than as the single-valued Functional Performance Model.

The BPM is expressed as a function of problem size in the same way that the

FPM describes performance. However, it differs in that at any particular problem size the speed of the processor is represented by a band of performance rather than some discrete value. This band can be a simple maximum and minimum predicted performance or a more complex probability density function describing the performance variation. For problems of small size, with short execution time, the band of a processor experiencing a large fluctuation in processor availability will have wide properties. For larger problems with longer running execution times, the processor availability will approach some constant level as load fluctuations average out, consequently the band will narrow.

Representing this processor availability in a model provides greater detail in the description of performance and allows for improved problem partitioning and task scheduling. In this chapter the formulation of the Band Performance Model is described. Methods to use the BPM in the partitioning of data for a parallel matrix multiplication problem are presented and these methods are compared to the Functional Performance Model as well as a Constant Performance Model. The comparison is done in a simulated execution environment which is also described in depth.

2.1 Formulation of the Band Model

The Band Performance Model is a product of two inputs. A Functional Performance Model that describes the ideal speed of a processor executing a problem with no external processes contending for the CPU, and some prediction of future external processor load that is provided at runtime, when a problem is to be partitioned or scheduled.

The FPM represents execution speed as operations per second. It is agnostic to the type of operation, but for most scientific problems the speed is represented as Floating Point Operations per Second (FLOPS). It is denoted as: $S_i(x)$, where x is the size of the problem which the processor executes and the subscript i indicates the description of ideal speed. The number of operations required to compute a problem of size x is its complexity, $C(x)$. Using complexity, the ideal

execution time, $T_i(x)$ and ideal execution speed can be derived from one another:

$$S_i(x) = C(x)/T_i(x)$$

$$T_i(x) = C(x)/S_i(x)$$

Practically, $S_i(x)$ is a piece-wise linear approximation of the real speed function, each point in the piece-wise approximation corresponds to a benchmark of the problem at a particular problem size. The benchmarking of a problem and construction of FPM and BPMs is a subject described in Chapter 3. For this description it is assumed that the approximation of $S_i(x)$ has already been constructed.

The second input to the formulation of the Band Performance Model is a prediction of processor availability. The basis for this prediction is the UNIX *load average*.

The load average is a statistic provided by UNIX-based and UNIX-like operating systems. It is a measure of “the number of processes in the system run queue averaged over various periods of time”¹. The operating system maintains load averages with time periods over one, five and fifteen minutes. The averages are exponentially damped, and samples of the run queue length are taken at a fixed frequency. The system queue length is also exposed to the user and may be used to calculate averages manually over shorter periods. Load averages are widely reported by utilities such as `w` and `uptime`, system calls such as `getloadavg()` and the `/proc` file-system on Linux.

Its use as an estimate of processor availability is well researched. In [73] queue length is shown to be a better measure of availability for load balancing than percent CPU utilisation. The one minute UNIX load average performed well, with a four second average of the queue length performing best. In [74] a number of different system parameters were used to describe processor load in a load balancing system. It was shown that for balancing the execution of short tasks, the queue length was the best individual description of load on a system. In their experiments the load average performed poorly, but this is likely as a result

¹From the `getloadavg` man page entry, section 3 (library calls), Linux Programmers Manual.

of the type of task they were balancing, which had short execution times. In [76] a hybrid availability measure used by the WINNER system was described. It combined both load average data and CPU usage to improve accuracy and they presented experiments showing how closely their availability measure tracked actual CPU availability. In [75] again, UNIX load average is used in a prediction of CPU availability in the Network Weather System. In this case it is combined with CPU usage and a probing process in a autocorrelating forecasting model. They show accurate short and medium term predictions of CPU availability using their methods.

The Band Performance Model uses a history of load average observations to generate two types of CPU availability predictions, first a simple maximum and minimum processor availability measure is used to generate the simple BPM. Second, a more detailed probability density function describing processor availability is used to generate a Probabilistic Band Performance Model. In both cases, the predictions from the load history are used to adjust the ideal speed function S_i so that it represents a range of predicted speeds.

The investigation of the BPM does not touch on the improved CPU availability measures that have been presented in other research, however they could be used to generate the band model in the same manner that will be described.

2.1.1 Load History

A history of load fluctuations is kept by sampling the one minute load average at a fixed interval of Δ time units (logically every minute). The recorded one minute load average at the $n - th$ sample is denoted as l_n . A load average for a period of t minutes can then be calculated by averaging a uninterrupted sequence of $\frac{t}{\Delta}$ historical load observations. Given a history of load averages, H , containing h observations:

$$H = l_1, l_2, \dots, l_h$$

Load averages over periods $\Delta, 2\Delta, \dots, h$ may be calculated. The Δ period load averages are simply the individual load observations l_i , the 2Δ period load averages would be the average of pairs of load observations l_i, l_{i+1} where $i = 1 \dots h$.

The periods for which load averages are calculated is limited by a window

from Δ to w time units, $2w < h$. For example, if $w = 60$ and $\Delta = 1$ (both minutes), then the 1 minute period load averages will be the 60 most recent load observations in H :

$$LA_1 = l_1, l_2, \dots, l_w$$

The 2 minute period load averages, will be calculated from the 61 most recent load observations:

$$LA_2 = \frac{l_1 + l_2}{2}, \frac{l_2 + l_3}{2}, \dots, \frac{l_w + l_{w+1}}{2}$$

The w , 60 minute, period load averages require $2w - 1$, or 119 observations to calculate,

$$LA_w = \frac{\sum_{k=1}^{60} l_k}{60}, \frac{\sum_{k=2}^{61} l_k}{60}, \dots, \frac{\sum_{k=w}^{2w} l_k}{60}$$

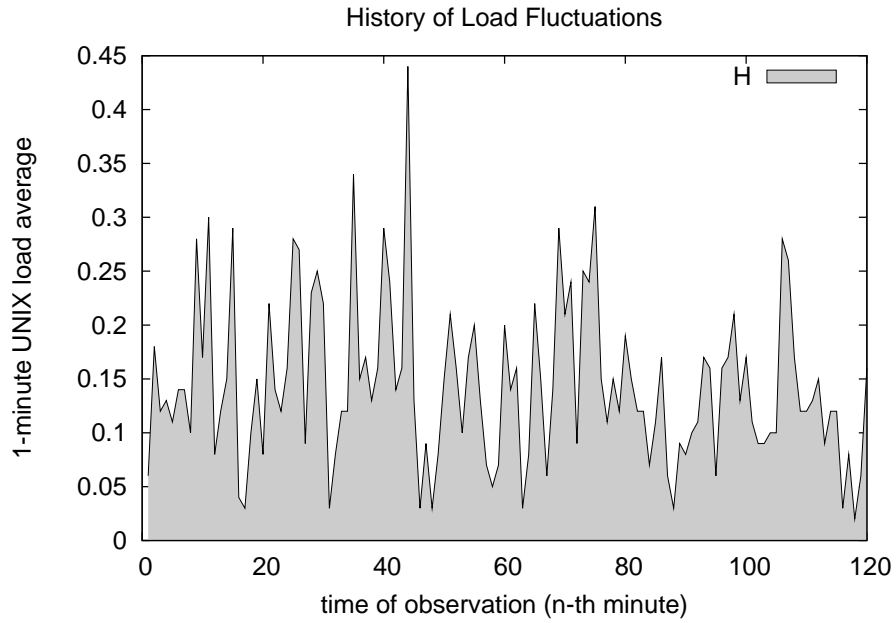
A calculated load average is defined as $l_{i,j}$, where i is the period of the average and j is the position in the sequence of calculated averages. A matrix of all calculated load averages is defined as follows:

$$LA = \begin{pmatrix} l_{1,1} & \cdot & \cdot & l_{1,w} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ l_{w,1} & \cdot & \cdot & l_{w,w} \end{pmatrix}$$

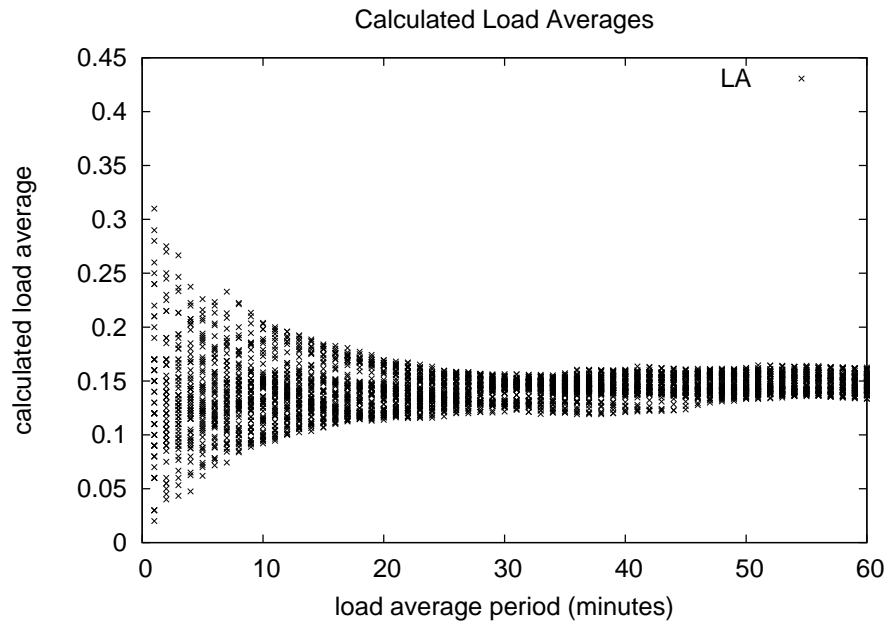
where:

$$l_{i,j} = \frac{\sum_{k=j}^{i+j-1} l_k}{i} \text{ where } i = 1 \dots w; j = 1 \dots w \text{ and } 2w - 1 \leq h$$

Using this method, the matrix LA has an equal number of load averages for every average period, thus it provides information on historical performance variation at every period. Also, shorter periods only use the most recent observations, while longer average periods use a deeper history of observations. Figure 2.1 shows an example of a history of load observations and the load averages calculated from that history. The calculated averages are used as an availability measure to apply to the ideal functional performance model, $S_i(x)$, to create the Band Performance Model.



(a) Load observations which are used as the basis for the calculated load averages, LA .



(b) Calculated load averages using a window of 60 minutes over 120 minutes of load observations.

Figure 2.1: Load observations and the load averages for various periods calculated from them.

2.1.2 Applying Load History to Functional Performance Model

Two piecewise linear functions that represent maximum and minimum expected load over increasing time periods (t) can be extracted from the load average matrix LA (Figure 2.2). The points in the load functions are defined as:

$$l_{max}(t) = \max_{j=1}^w(LA_{t,j})$$

$$l_{min}(t) = \min_{j=1}^w(LA_{t,j})$$

where $t = 1 \dots w$. I.e. the maximum and minimum of each row of the matrix define points in the piece-wise functions. $S_i(x)$, the ideal functional performance model of some task may be adjusted using l_{min} and l_{max} to form a band rather than a single valued function. $S_i(x)$ is piece-wise continuous function, consisting of a series of points, connected by line-segments. Converting this to a band involves converting each point to the maximum and minimum predicted level of performance. A load average, l , describes the number of processes executing or

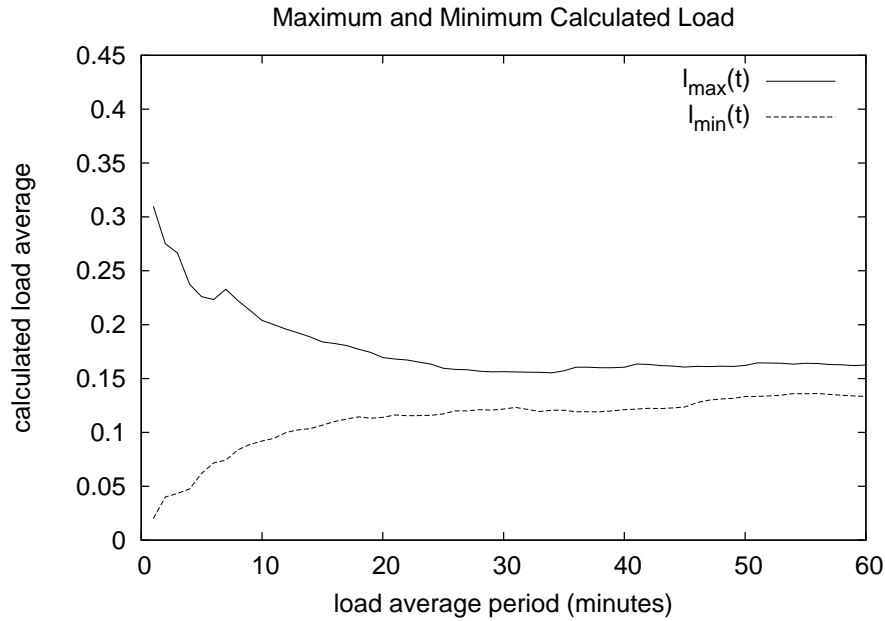


Figure 2.2: Maximum and minimum load functions extracted from the calculated load average matrix LA .

waiting to execute on the CPU over a certain period. The availability of the CPU over the same time period may be derived from the load average and this can then be used to calculate the actual speed of a point in $S_i(x)$. Introducing a single threaded computational task (i.e. the task that $S_i(x)$ describes) to a processor will add 1 unit to the current load. Now a total of $l + 1$ processes will be executing or waiting to execute. We assume that all processes will receive a fair share of the CPU time, i.e. the processor's availability, is divided evenly between the $l + 1$ processes. The proportion of CPU time that the computation task will receive, the availability a , is defined as follows:

$$a(l) = 1 \times \frac{1}{(1 + l)}$$

So a task of size x_p that executes under ideal conditions with a speed of $S_i(x_p)$, utilising 100% of the CPU, will execute at $a(l) \times S_i(x_p)$ under some load l . Similarly, the execution time of the task under a load, T' , is equal to $T_i(x_p)/a(l)$.

For every point in $S_i(x)$, $T_i(x)$ is also known. This execution time may be used to look up the maximum and minimum predicted load over such a time period in $l_{max}(t)$ and $l_{min}(t)$. However, applying these predicted loads would increase the execution time to T' , and the predicted load over the new duration may be altered. So, to adjust a point in $S_i(x)$ using $l_{min}(t)$ and $l_{max}(t)$, the intersection between execution time adjusted for load, $T'(l)$, and the load functions must be found. Both of these functions have the same, though reversed, axes (time and load). Though the intersection between them may be found analytically, in practice it is carried out geometrically due to the piece-wise nature of the load functions. This is illustrated in Figure 2.3. This action is carried out for every point in the ideal functional performance model. Ideal execution times are converted to maximum and minimum predicted execution times and finally to pairs of executions speeds which form the simple Band Performance Model (Figure 2.4).

2.1.3 The detailed Band Performance Model

The matrix of calculated load averages, LA , describes the historical distribution of load over increasing time periods. The simple Band Performance Model only

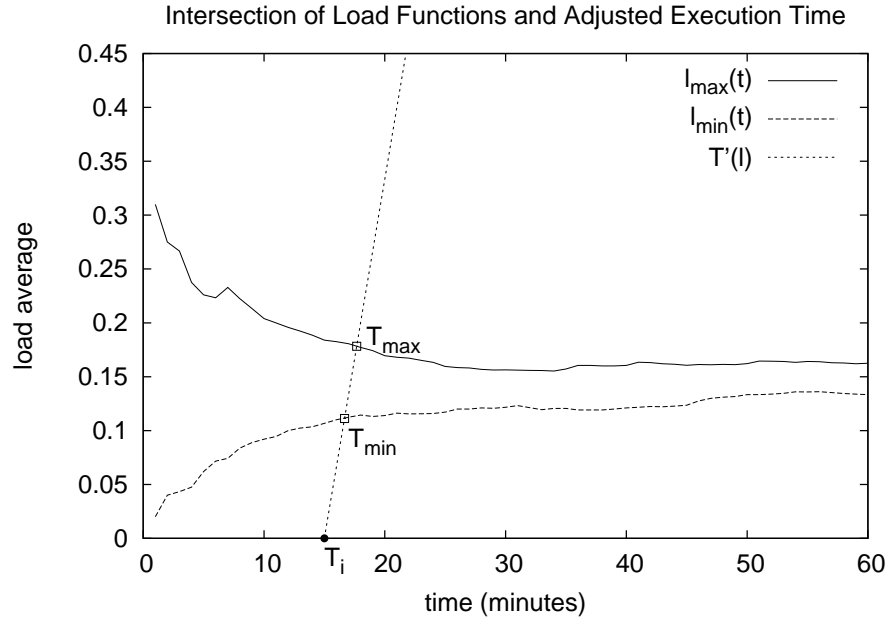


Figure 2.3: T_i , the ideal execution time of some problem is adjusted for maximum and minimum expected load by finding the intersection of T' and $l_{min}(t)$ or $l_{max}(t)$.

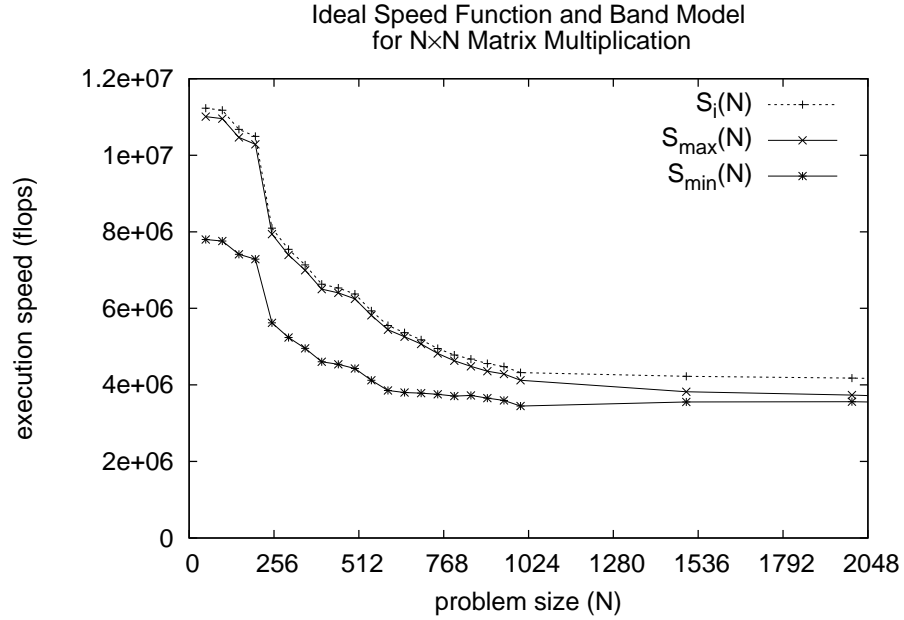


Figure 2.4: Ideal functional performance model $S_i(N)$ and the simple Band Performance Model derived from it.

uses the maximum and minimum loads from each time period, as represented by the piece-wise linear functions l_{max} and l_{min} . Using every load average for a relevant time period in LA would allow for a detailed Band Performance Model where the speed of a execution is not a simple interval, from minimum to maximum, but some distribution of speeds in that range.

$T'(l)$, the execution time of a problem, adjusted for load, is intersected with the load functions $l_{max}(t)$ and $l_{min}(t)$ as before. This defines a period window, T_{min} to T_{max} , over which load averages from LA should be examined. The corresponding rows of LA that are used to find the distribution of execution times are defined by the integer division of the period window by load observation period Δ , i.e.:

$$rows = \lfloor \frac{T_{min}}{\Delta} \rfloor \dots \lfloor \frac{T_{max}}{\Delta} \rfloor$$

The load averages in these rows of LA are distinct points, not interconnected piece-wise functions as l_{max} or l_{min} . In order to find the distribution of execution times and speeds $T'(l)$ must be intersected with line segments that represent these point load averages. To create a line segment from a load average l , it is projected, from its position at its particular time period δ , to the time period $\delta + 1$. This projection is not simply horizontal, but adjusted for the change in load band width between δ and $\delta + 1$. A line segment AB is defined by two points A and B . A is simply the load average (l, δ) , B is the $\delta + 1$ projection, positioned in the range of $l_{min}(\delta + 1)$ to $l_{max}(\delta + 1)$, relative to the position of l between $l_{min}(\delta)$ and $l_{max}(\delta)$:

$$\begin{aligned} A &= (l, \delta) \\ B &= \left(l_{min}(\delta + 1) + (l - l_{min}(\delta)) \times \frac{l_{max}(\delta + 1) - l_{min}(\delta + 1)}{l_{max}(\delta) - l_{min}(\delta)}, \delta + 1 \right) \end{aligned}$$

Line segments are created for every load average in the *rows* of LA . Then each segment is tested for intersection with $T'(l)$. This is illustrated in Figure 2.5. The collection of all intersection points reveals the distribution of predicted execution times. These execution times are converted to speeds and histogram is built to represent the distribution of execution speeds. From the histogram, a piece-wise

probability density function is formed, illustrated in Figure 2.6. This gives a probability based estimation of the performance of a processor for any given problem size and is the final representation of the detailed Band Performance Model.

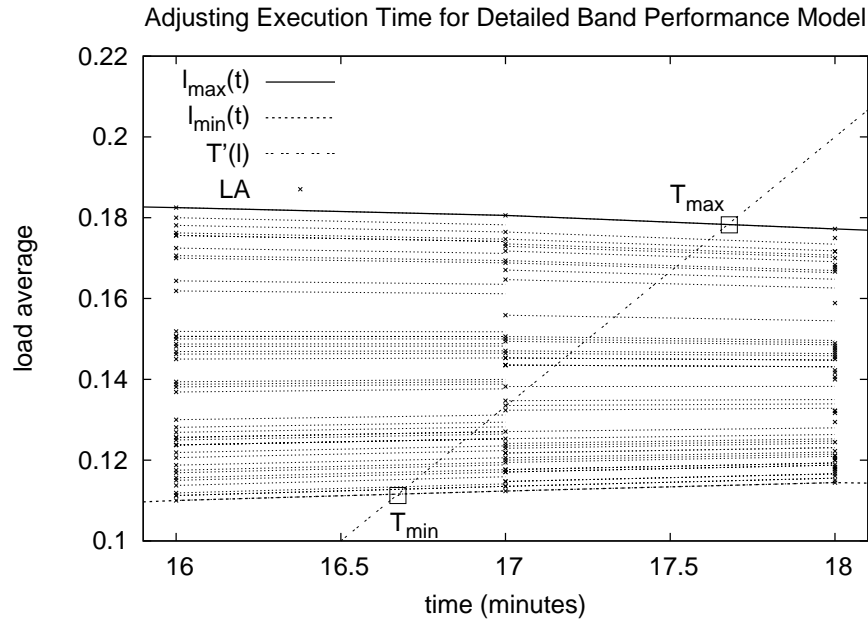


Figure 2.5: T' is intersected with projections of calculated load averages from LA .

2.2 Workload Distribution with Performance Models

Having formulated the simple and detailed Band Performance Models, methods to use these models to solve the data partitioning problem must be conceived. The simplest solution to the data partitioning problem for heterogeneous processors is the use of a constant performance model, where the speed of a processor is represented by a single benchmark of some kind. Given a set of processors and benchmarks, the total speed of the set and the speed of a single processor relative to the entire set may be calculated. A problem is then partitioned between pro-

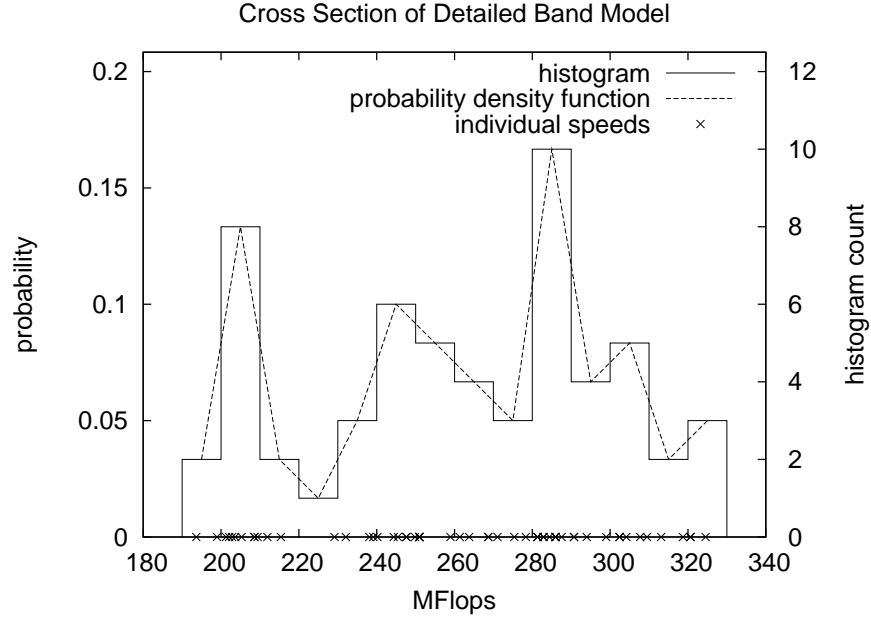


Figure 2.6: Predicted execution speeds for a single problem size (on x-axis) are binned and plotted in a histogram which forms as the basis for a probability density function describing the processor's speed variation at that problem size.

processors such that each receives a chunk that is proportional to its relative speed.

Given a set of n processors, $P = p_0, p_1, \dots, p_n$, the speed of the j -th processor, as represented by a single benchmark, is given by some value: S_j . A workload of W may be partitioned between processors such that each processor P_j , is given an amount of work: w_j , defined as:

$$w_j = W \times \frac{S_j}{\sum_{k=1}^n S_k}$$

If S_T is the summation of executions speeds $\sum_{k=1}^n S_k$, we can also say that the solution to the problem partitioning is finding a distribution where there is an equal ratio of speed and problem size across all processors, i.e.

$$\frac{S_T}{W} = \frac{S_j}{w_j} = \frac{S_0}{w_0} = \frac{S_1}{w_1} \dots \frac{S_n}{w_n}$$

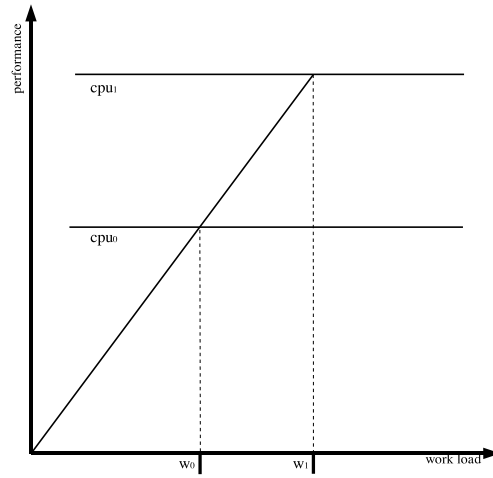
Calculating S_T and w_j is trivial when the model of performance is based on a sin-

gle benchmark as the speed of a processor is represented as constant regardless of the problem size it executes. This relative division of workload can also be expressed graphically as in Figure 2.7a, where a line from the origin intersects two constant performance models. Dividing some total workload into perfectly relative portions.

The Functional Performance Model describes performance in a more realistic way, not as a constant, but as varying with problem size, i.e. the speed of the j -th processor is defined by $S_j(x)$. Partitioning with such a model has been described in [67]. It has been demonstrated that the optimal partitioning of a problem occurs when a line projected from the origin intersects the functional performance models. The summation of the x components of the intersection points (i.e. problem sizes) gives the total workload for which the distribution is perfect. Conversely, finding the perfect distribution of some workload corresponds to finding a line through the origin which intersects the functional models such that the x components sum to the required workload. An illustration of this is shown in Figure 2.7b. An efficient algorithm to achieve this through a bisection search is presented in [67].

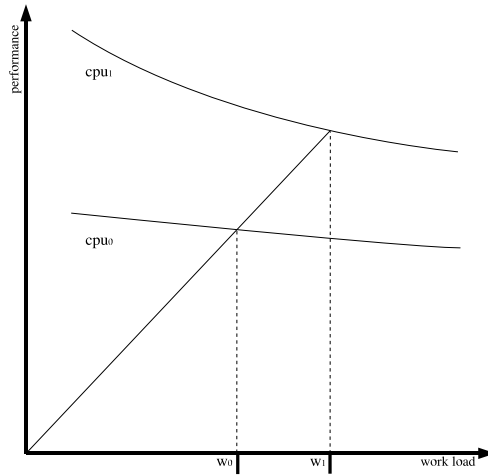
The Band Performance Model extends the FPM so that performance is no longer a single function but a region between two maximum and minimum functions, $S_{max}(x)$ and $S_{min}(x)$. This region reflects the possible performance variations on the processors. Given a particular distribution of work, there may be many combinations of performance variations for which the distribution is optimal. There may exist more than one line that passes through the origin and intersects the performance bands of a set of processors at the work attributed to that processor. Such a pair of performance levels for two processors and a particular workload distribution is illustrated in Figure 2.8. The notion that a workload partition may be optimal for certain levels of performance described by the band is the basis for the proposed methods of determining the best workload partition using Band Performance Models.

Problem Partition using a Performance Benchmark



(a) Single benchmarks represent speed as constant across problem size, the partition of workload with such a model may be graphically represented by a line through the origin intersecting both models.

Problem Partition using a Performance Function



(b) Partitioning with functional models is equivalent to finding the same line, through the origin that intersects both models at certain problem sizes, summing to the total workload to be partitioned.

Figure 2.7: Graphical representation of partitioning with constant and functional performance models.

Problem Partition using a Performance Band

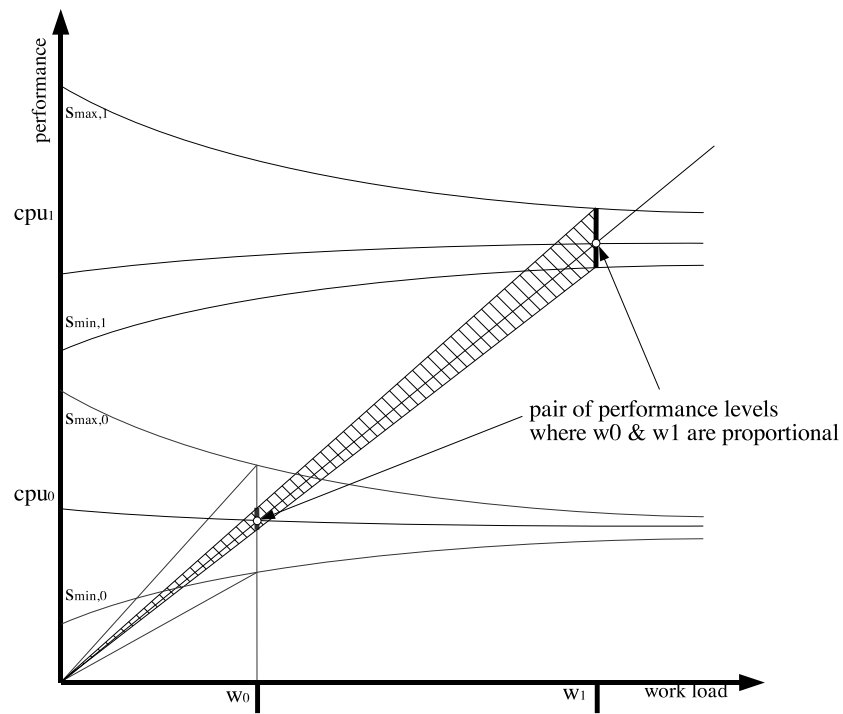


Figure 2.8: Illustration of performance fluctuations which result in optimality for some distribution of workload.

2.2.1 Angular Metric

As described previously, the graphical expression of partitioning with FPMs is to find some line that intersects the FPMs of the processors and the origin, such that the sum of the intersection points is equal to the workload that is to be partitioned. Partitioning with Band Models can also be approached as an extension of partitioning with FPMs. The single line through the origin that solves the partitioning problem using a FPM is expanded to an angular segment of lines, passing through the origin and through the performance band of each processor at the corresponding workload. Each line in this angular segment corresponds to a set of performance levels in the bands of the processors for which the distribution is optimal. Maximising the size of the angular segment will maximise the number of performance combinations for which the distribution will be optimal. Should all performance levels in the bands have equal probability, a distribution which maximises the angular segment should have a high probability of being optimal.

For a set of n processors, P . A distribution of a total workload W is given by:

$$W = \sum_{j=1}^n w_j$$

For each processor p_j , the workload assigned to it w_j may execute at a range of speeds, as described by the band model, from $S_{max_j}(w_j)$ to $S_{min_j}(w_j)$. An angle θ_j is created with vertex at the origin and lines, extending to the maximum and minimum speeds $(w_j, S_{max_j}(w_j))$ and $(w_j, S_{min_j}(w_j))$, as described by a processors Band Model. Where the set of angles overlap, there is what is titled a “common angle” α . Lines within this common angle correspond to a workload distribution that is optimal for some particular combination of speed variations across the processors in the set.

The size of the common angle can be calculated from the angle of incidence with the x axis that is made by the lines which create the set of θ_j angles. If the incidence angles of lines extending to the maximum and minimum performance

functions of some processor p_j are defined as:

$$\begin{aligned} I_{max_j} &= \arctan\left(\frac{S_{max_j}(w_j)}{w_j}\right) \\ I_{min_j} &= \arctan\left(\frac{S_{min_j}(w_j)}{w_j}\right) \end{aligned}$$

Then the size of the common angle may be defined as the subtraction of the largest incidence angle of lines intersecting the minimum performance functions, from the smallest incidence angle of lines intersecting the maximum performance functions. I.e.

$$|\alpha| = \max_{j=1}^n(I_{min_j}) - \min_{j=1}^n(I_{max_j})$$

Where the angles of incidence that make up α are:

$$\begin{aligned} I_{\alpha_0} &= \max_{j=1}^n(I_{min_j}) \\ I_{\alpha_1} &= \min_{j=1}^n(I_{max_j}) \end{aligned}$$

The calculation of the common angle is illustrated in Figure 2.9. Finding a distribution with the widest common angle will maximise possible performance variance combinations where the distribution remains optimal.

2.2.2 Constant Probability Metric

Instead of maximising the common angle α , the range of performance levels that the common angle covers over each performance band can be considered as a metric to maximise when finding a good workload distribution. The magnitude of α represents the volume of possible perfectly balanced performance fluctuations for a particular distribution, but α does not describe whether these performance fluctuations cover a wide range of the overall performance fluctuations of the processors.

The Constant Probability Metric rates a workload distribution using the percentage of each performance band that is covered by the common angle α . The

Calculating the Distribution Arc

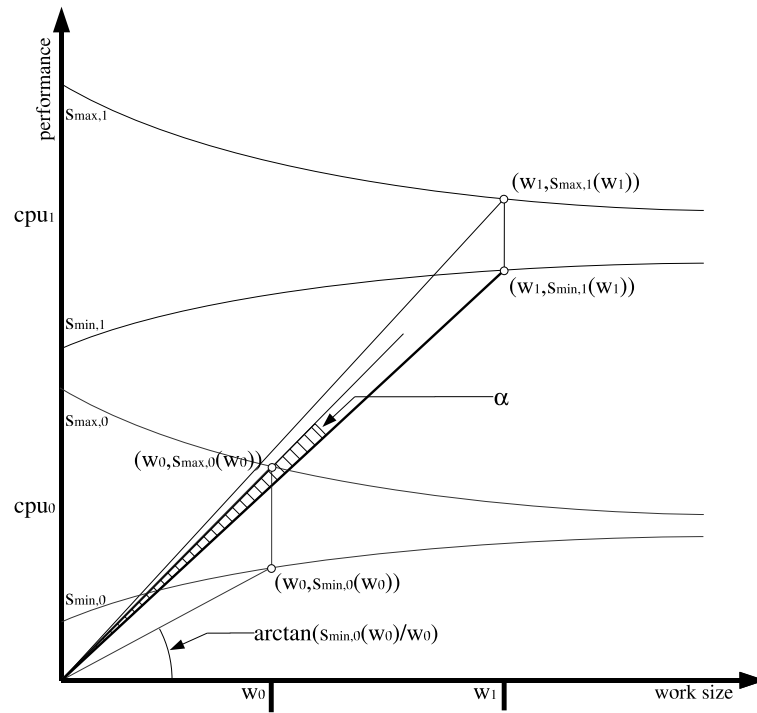


Figure 2.9: Illustration of the arc of lines through the origin which indicate that a workload is optimal for some combination of performance fluctuations on a set of processors.

performance fluctuations of each processor are given a constant equal probability of occurring. The probability of the fluctuations that occur inside the common angle are used to calculate an overall probability of the distribution being balanced.

A distribution of work, w_j , on a processor p_j will have a predicted performance variance given by a cut, $C_j(w_j)$, of the performance band. The cut is the vertical section from $(w_j, S_{min_j}(w_j))$ to $(w_j, S_{max_j}(w_j))$. The sides of the common angle α , if it exists, will intersect the cut a processor's performance band at two points, u_j and v_j , which may be calculated from the incidence angles of the two sides, $I_{\alpha_{max}}$ and $I_{\alpha_{min}}$:

$$u_j = w_j \times \tan(I_{\alpha_0})$$

$$v_j = w_j \times \tan(I_{\alpha_1})$$

The actual future performance is assumed to be inside the cut $C_j(w_j)$. Within the cut an even and constant probability is assigned to each performance level between the cut limits of S_{max_j} and S_{min_j} . On finding the intersection between the common angle and $C_j(w_j)$, given by values u_j and v_j , the probability of the performance being at some level inside this intersection is given by:

$$\phi_j = (v_j - u_j) / |C_j(w_j)|$$

Essentially, ϕ_j is the percentage of the cut that is intersected by the common angle. Given a set of processors and a workload distribution, the probability of the performance variance of each processor occurring within the limits of the common angle may be calculated, and the overall probability of these performance levels occurring on all processors is given by:

$$\Phi = \prod_{1 \leq j \leq n} \phi_j$$

Maximising Φ is the second method proposed for finding a good distribution of workload using Band Performance Models.

2.2.3 Piecewise Probability Metric

The detailed Band Performance Model represents the variance in processor speed not just as an interval with even probability, but as a series of discrete speed predictions. Using the detailed Band Performance Model, and the common angle, α as calculated described previously, a probability metric that is more accurate than the constant one may be calculated.

A distribution of work assigned to the j -th processor, p_j , creates cuts its performance band as before. The cut of the band, $C_j(w)$, is defined as the vertical interval from $(w_j, S_{min_j}(w_j))$ to $(w_j, S_{max_j}(w_j))$. Following the procedure described in section 2.1.3, a series of individual performance predictions can be calculated between the points that define the cut. These predictions are gathered into a probability density function $f_j(x)$ (Figure 2.6) which is used to represent the speed of the processor executing the workload w_j . As with the constant probability metric described previously, the common angle α intersects the cut of the j -th processor at values u_j and v_j . The probability distribution function may be integrated between these points to give a more accurate measure of the probability that the performance level will occur between the points made by the common angle intersection.

ϕ_j , the probability of the performance of a processor p_j being inside the common angle α is defined as:

$$\phi_j = \int_{u_j}^{v_j} f_j(x) dx$$

On finding ϕ_j for each processor and a given workload distribution, Φ , the overall probability that the distribution will encounter some performance fluctuations where it results in perfect balance, is again, the product of the each independent processor's probability measure ϕ_j .

$$\Phi = \prod_{1 \leq j \leq n} \phi_j$$

The probability metric is a more reliable metric for the distribution as it encapsulates the greatest detail about the actual performance fluctuations of each

processor in the network.

No algorithms have been designed for finding the best distribution based on the metrics provided for Band Performance Model. Instead their use is investigated by way of exhaustive search for small solution spaces and genetic algorithms for larger solution spaces. This is presented Section 2.3.1.

2.2.4 Adjusted Functional Model

The final method of partitioning with a Band Model to be presented is where the band itself is only used to distil a Functional Performance Model. Here, all predictions of execution speed from the detailed Band Performance Model described in section 2.1.3 are gathered and averaged to give single valued mean execution speed for each problem size that makes up the piece-wise linear speed function.

The novelty of the adjustment is that it is that each point in the functional model is not adjusted by a single common prediction of future load fluctuation but by a dynamic one which predicts based on the time period of the execution for a particular problem size.

A median of the maximum and minimum speed functions, $S_{max}(x)$ and $S_{min}(x)$ which define the boundaries of the simple Band Model was also considered, but intuitively this does not render as accurate a prediction as the mean performance from the detailed Band Performance Model.

Partitioning a problem with a set of functional models is by way of a bisection search algorithm which was presented in [67]. This algorithm finds the line through the origin which intersects each performance model such that the problem size components of those intersection points sum to the total workload to be distributed.

2.3 Problem Partitioning Experiments

In order to investigate the utility of the Band Performance Model in addressing the problem of data partitioning, a simulated execution environment has been established. This consists of two parts: functional performance models and load

fluctuation traces. The functional performance model provides the optimal execution speed and execution time for any problem size assigned to a processor. The load history trace can be used to predict the actual execution time of a problem on a processor. Load traces are often used to evaluate the performance of a scheduling method or load prediction algorithm [83, 80]. Repeatable experiments may be conducted as follows:

- View the load trace from some time t .
- Utilise the load observations preceding t to build the Band Model, detailed Band Model and Adjusted Functional Model.
- Partition a problem using these models:
 - In the case of Band Models, find the optimal partition using exhaustive search or genetic algorithm
 - In the case of Functional Model, use existing algorithm.
- Evaluate the partitions by simulating their execution:
 - Find the optimal execution time of each sub-problem of the partitioned problem using the optimal functional performance model.
 - Adjust the optimal execution time using load observations in the trace from the time t onwards, giving the simulated execution time of the sub-problem.
 - Find the longest execution time of the sub-problems and return this as the execution time of the partitioned problem.

The problem to be partitioned in the set of experiments is multiple repetitions of a naively parallelised square matrix multiplication. The repetitions are carried only to extend the overall execution time of the problem, they are not parallelised. The parallelisation comes from a naive distribution of data. In the calculation of $C = A \times B$, B is held by all processors, and columns of C and A are distributed between processors (see Figure 2.10). The calculation of the distributed columns of C requires no interprocess communication as each processor

will have the relevant columns of A and all of B . The simulation environment does not need to consider the time spent on communications as a result. The workload to distribute is based on a single parameter, W , the number of columns in a matrix.

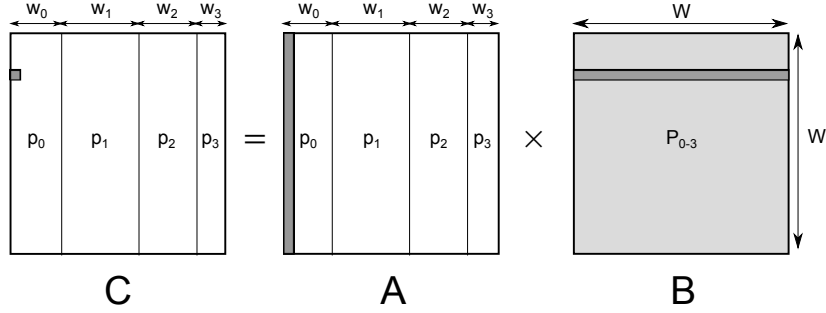
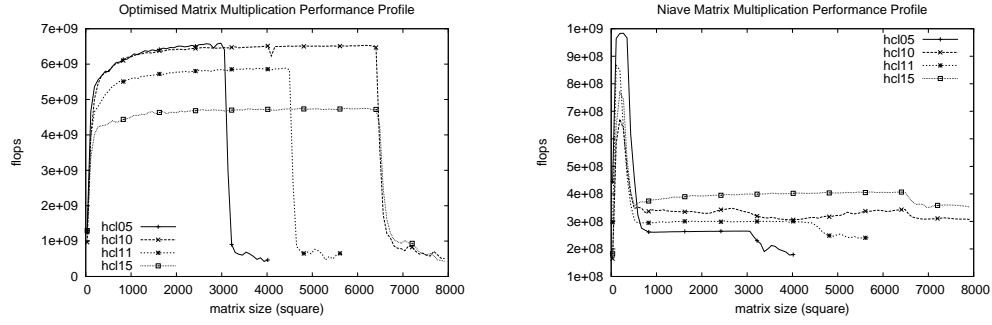


Figure 2.10: Illustration of a Column-wise Naive Partitioning of a Parallel Matrix Multiplication. B is distributed to all processors, w_0 columns of A are distributed to p_0 and elements in corresponding w_0 columns of C are calculated by p_0 .

The partitioned matrix multiplication is performed using either an optimised BLAS library (GotoBLAS2 [84]) or a naively implemented matrix multiplication method. The shape of the performance models of these two implementations differs greatly. The highly tuned algorithm has a relatively flat performance model regardless of problem size (in the region before paging). This flat part may be approximated accurately by an appropriately sized single benchmark, but the user must be aware of the profile first and choose the size correctly. The naive matrix multiplication has a performance model with many decreases in execution speed, particularly as the problem size reaches the boundaries of different levels of a processor's memory hierarchy. This kind of performance profile can not be represented using a single benchmark.

Experiments using the optimised multiplication method relate to partitioning of highly tuned codes, such as those provided in various high performance computing libraries. The naive implementation relates to partitioning of a user written parallel code, where expertise in hardware specific or automated optimisation may not be available. The code will have a complex performance profile which cannot be accurately approximated by a single benchmark.



(a) Functional Performance Models of optimised matrix multiplication on platforms used in experiments.

(b) Functional Performance Models of naive matrix multiplication on platforms used in experiments.

Figure 2.11: Performance profiles of optimised and naive matrix multiplication methods.

Each of the metrics described in 2.2 are used to partition a workload, then these partitions are evaluated and compared with an optimal partition, which is found using future knowledge of performance fluctuations. Before the experimental results are presented in subsection 2.3.3, the method of finding partitions based on metrics (subsection 2.3.1), and the method of evaluating those partitions (subsection 2.3.2) are described.

2.3.1 Metric Based Partitioning

In order to find the best distribution indicated by each metric, angular, constant probabilistic or piece-wise probabilistic, a search of all possible distributions must be made. The purpose of these experiments is to investigate the suitability of the Band Performance Model for workload distribution, rather than to present methods to use the Band Model for distribution. Efficiency in finding the best distribution is not considered. Finding a distribution using each metric is done either using an exhaustive search or an evolutionary meta-heuristic.

Exhaustive Search

An exhaustive search involves evaluating the metric for all possible distributions of a workload between n processors. This is equivalent to iterating through a

restricted set of compositions of an integer. The restrictions put in place are that the composition must have n components and that each component, or processor, must have a non-zero value. A simple recursive algorithm was implemented to achieve this, illustrated in Algorithm 1.

Algorithm 1 ConstrainedCompositions(I, N, IDX, R, C)

Input: I , integer to decompose
Input: N , number of components to decompose to
Input: IDX , index variable (initially 0)
Input: R , remainder of I left to be decomposed (initially I)
Input: $C[N]$, array to store composition of I

```

    if  $\text{IDX} + 1 = K$  then
         $C[\text{IDX}] \leftarrow R$ 
        print  $C[\text{IDX}]$ 
    else
        for  $V = 1$  to  $R$  do
             $C[\text{IDX}] \leftarrow V$ 
            ConstrainedCompositions( $I, N, \text{IDX} + 1, R - V, C$ )
        end for
    end if

```

The algorithm used implements tail recursion and so is relatively fast. There are many pre-existing algorithms to calculate compositions which may be more efficient or faster (e.g. [85, 86, 87]), however the efficiency of generating compositions is not important. The number of possible compositions, i.e. the number of possible distributions of a workload, increases rapidly as the number of components (processors), and the integer (workload) to decompose increase. The number of compositions of an integer I in n parts, (where zero is excluded from the compositions) is given by the binomial coefficient ([88]):

$$\text{comps} = \binom{I-1}{n-1}$$

It should be clear that it is only possible to evaluate the metric of every possible workload distribution for small numbers of processors and workloads. In experiments the exhaustive search was used for 2 and 3 processor configurations only.

Genetic Algorithm

For larger configurations an Evolutionary Computation technique has been used to search through the problem space of possible workload distributions, specifically, a Genetic Algorithm (GA) [89]. GAs are often suitable for multi-dimensional global search problems particularly where the search space is not well known. Given the variety of metrics to test, the GA is seen as a useful investigative method.

Algorithm 2 Genetic Algorithm

```

Initialise(Population)
Evalutate(Population)
while not TerminateCondition do
    newGeneration := Select(Population)
    newGeneration := Crossover(newGeneration)
    newGeneration := Mutate(newGeneration)
    Evaluate(newGeneration)
    Population := Survive(newGeneration, Population)
end while

```

A workload distribution is represented as a member of a population in a GA. Each member has a certain fitness, in this case given by the metric, the population is evolved using elements from natural evolution such as crossover, mutation and natural selection. This results in the average fitness of the population increasing over iterations of the evolutionary cycle, and a good coverage of the entire solution space by way of random mutations. The general pattern of a GA is shown in Algorithm 2.

In order to use a GA to search the solution space of possible workload partitions a number of core functionalities are implemented as follows:

- The encoding of a workload distribution as a member of the population of the GA is as an array of integers describing the distribution. Each integer is a “chromosome” of the member’s representation in the GA. The distribution is seen both as the actual decomposition of a particular workload, and as a relative weighting of work to assign to each processor. This relative weighting is necessary for crossover to function.

- Evaluation of the fitness of member of the population is by way of the metrics described in the previous section. A metric will give a score to each population member which is used in the selection process.
- The Selection process is by tournament, there is no specific modification to the selection process to deal with the application domain. Tournament players are randomly selected from the population and their ranks (provided by metrics) are used to choose a winner. Pairs of winners are used as parents in crossover.
- Crossover is one of the main ways a genetic algorithm traverses the search space so it is important that it is implemented properly. The general method is to splice “chromosomes” between two parents together, to create two children, each with opposing halves of their parents. In this application, a single chromosome would be the work load assigned to a specific processor. And the entire chromosome string is the overall workload distribution.

Given two parent workload distributions, $da = da_0, da_1, \dots, da_n$ and $db = db_0, db_1, \dots, db_n$, a splice point is selected randomly at a processor p_s in the distribution. The first child will be generated from da_0, da_1, \dots, da_s and $db_{s+1}, db_{s+2}, \dots, db_n$ distribution sub-parts. The second child will have the opposite sub-parts db_0, db_1, \dots, db_s and $da_{s+1}, da_{s+2}, \dots, da_n$.

At this point the children will have distributions that may sum to more or less than the total amount of work to be distributed. A fixing function must be applied. The fixing function takes a distribution and finds the relative work assigned to each processor by that distribution, even if it distributes more than the target amount of work. Then the relative amounts are used to re-distribute the total required workload between the processors. This maintains the main theme of crossover, where characteristics from two parents are divided and passed on to two children.

- Mutation is achieved by taking random members of the new generation of children created by crossover and altering them in some random way. For this application, a random set of processors are chosen from the workload distribution, between these processors work is added to some and removed

from others, while maintaining the total amount of work distributed in the mutation.

- Survival is simple Darwinian survival, where the population is trimmed to a certain size by removing the least fit members.

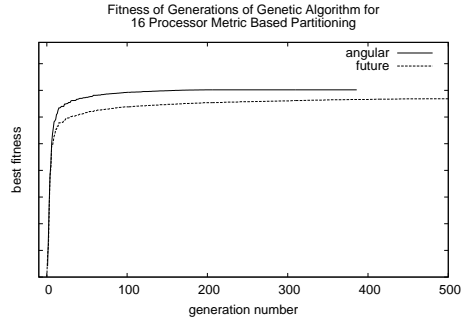
After a set number of generations has been exceeded, or when the population converges consistently for a number of generations, the Genetic Algorithm terminates, returning the distribution with the highest metric in its final population. The evolution of the fitness of this distribution, using each metric and for four, eight and sixteen processor configurations, are shown in Figure 2.12. These graphs show how the fitness of the best partition improves in each generation of the Genetic Algorithm. It can be seen that the GA rapidly approaches the fittest distribution in all cases. In Figures 2.12a and 2.12b, where there are sixteen processors to divide the workload between, the approach is slightly slower, but the solution space the GA searches is also much larger. The scales of the various metrics are omitted as they do not relate to each-other.

The distribution returned by the GA has been verified as having the highest metric of all possible distributions through a number of small scale exhaustive searches. Though it cannot be stated that this method always returns the best solution, using the GA enables experimentation with large numbers of processors which would otherwise be impossible.

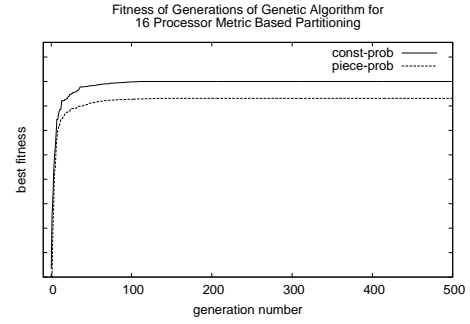
2.3.2 Calculating Execution Time

In order to evaluate the performance of a workload distribution created by a particular metric, the execution time of that distribution must be calculated. Using load traces, this can be achieved in the following manner:

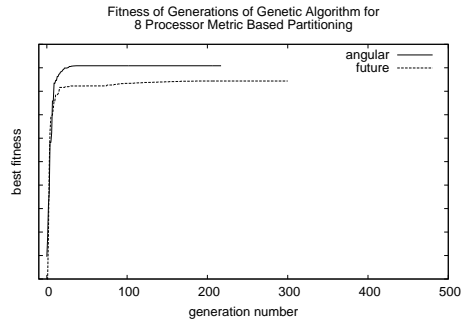
A load function for future load fluctuations from t onwards is created. It represents the future load at periods of $1\delta, 2\delta, 3\delta$ and so on. The execution time adjusted for load T' is intersected with this function in much the same way as it is intersected with historical load functions in the creation of the Band Performance Model (see Section 2.1.2). The resulting intersection point gives the execution time of the problem under the future load fluctuations.



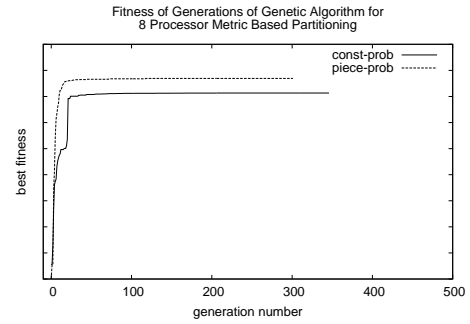
(a) Evolution of Distribution Fitness Based on Angular Metric and Future Knowledge (16 processors).



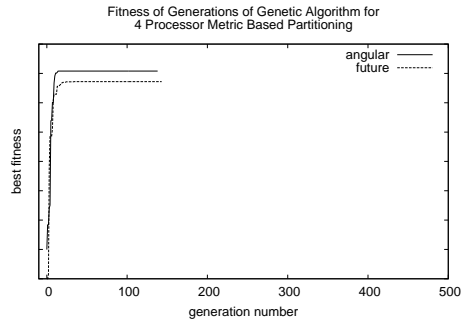
(b) Evolution of Distribution Fitness Based on Constant Probability and Piece-wise Probability Metrics (16 processors).



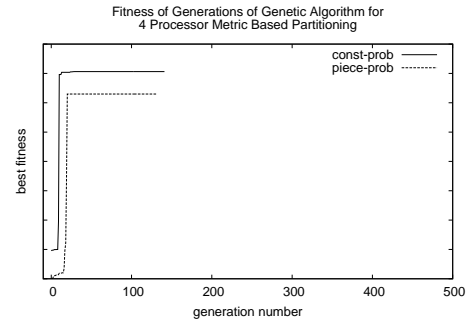
(c) Evolution of Distribution Fitness Based on Angular Metric and Future Knowledge (8 processors).



(d) Evolution of Distribution Fitness Based on Constant Probability and Piece-wise Probability Metrics (8 processors).



(e) Evolution of Distribution Fitness Based on Angular Metric and Future Knowledge (4 processors).



(f) Evolution of Distribution Fitness Based on Constant Probability and Piece-wise Probability Metrics (4 processors).

Figure 2.12: Fitness of metric based distribution in the generations of a genetic algorithm for 16, 8 and 4 processor problem distributions. Note: fitness scales omitted.

This method of predicting an execution time given a load trace is similar to that used in [80]. They represent availability of a processor as a continuous function of time based on a recorded load trace. The execution time of a task with a known ideal execution time ($t_{nominal}$), is given by the integral of an availability function, which should equal $t_{nominal}$ at some t_{exec} , the resulting execution time under the load trace. In the experiments presented here the load fluctuations are not integrated under, but instead their amalgamated effects over time are calculated, this amalgamation is intersected with T' . The result is identical, but the approach is slightly different.

This is the basic simulation environment in which experiments are conducted. Given a history of past load fluctuations and a trace of future load fluctuations, the simple and detailed Band Models may be created and used to partition some problem, which then has its execution simulated so that we may evaluate the performance of the distribution.

The clear drawback of this method is that it assumes the load observations perfectly describe the availability of the processor. Between CPU bound tasks, the relationship between load and processor availability is linear but this breaks down when IO-bound tasks share the processor. However, in order to maintain repeatability of experiments, the simulated environment is more convenient. It also has the advantage of allowing the evaluation of the models under different conditions without attempting to manufacture repeatable workloads. Further, the limitation does not prevent the comparison of different partitioning methods and metrics.

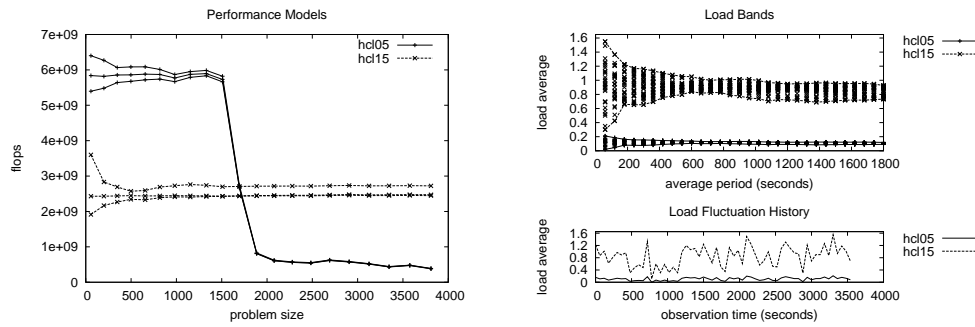
2.3.3 Experiments

A total of 16 processors are used in the following experiments. Each processor is represented by a Functional Performance Model which has been built on a real platform under zero external load, i.e. the model describes the peak performance of the processor for all possible partitioned amounts of work. The platforms used to obtain these models are listed in Table 2.1, *hcl01* to *hcl16* are included in the experiments.

Load traces were generated with a random Gaussian distribution and having

a target mean and standard deviation. In experiments this is used to apply certain characteristics to the load fluctuations experience by processors, either highly variable, with a large deviation, or stable with a low deviation. The performance of the partitioning metrics could then be evaluated under different circumstances by choosing sets of load traces to experiment with. Example performance models obtained from the platforms and load traces generated for an experiment are shown in Figure 2.13.

In each experiment square matrices of various sizes are partitioned between selected platforms and the partitions are evaluated. For each matrix size, the partition and simulated execution is made from multiple different points in the load trace and an average execution time is returned. The execution time of partitions are compared with a best case execution time, which is calculated using future knowledge of load fluctuations. The performance of each partition method, relative to the best case, is plot against the total size of the partitioned matrix.



(a) Band Performance Models and Adjusted Functional Models for *hcl05* and *hcl15* computing columns of a 4000 square matrix multiplication using an optimised routine.

(b) Example Load Bands and Historical Load Fluctuation Traces used in two Processor Experimentation with *hcl05* and *hcl15*.

Figure 2.13: Example performance models and load traces used for experimentation.

Two Processor Experiments

The first set of experiments use the platforms *hcl05* and *hcl15*. These processors have differing speeds and amounts of memory. *hcl05* uses an external load trace

with a average level of 0.1, i.e. it is busy 10% of the time, and a low level of fluctuation about this average. *hcl15* has a higher external load average of 0.8 and the fluctuation of the external load is moderate.

The results are shown in Figure 2.14. One can see through-out most of the results in this section that the single benchmark distribution method performs close to the optimal for small problem sizes. This is because the simulated execution of partitions is based on a load trace with a resolution of one minute. The execution time of small problems may be under one minute. The single benchmark uses an instantaneous measure of speed to partition the workload and if the execution occurs entirely within the same time-step as the measurement, it will have perfect accuracy.

The functional and band model based partition methods do not perform as well as the single benchmark in this region because their representation of processor speed is based on the history of load fluctuations rather than an instantaneous measure. For such short running problems, the average historical performance may not result in as accurate a representation as the instantaneous processor speed. Some adjustment to the formulation of the band model could be made to account for this, e.g. by using only few most recent load observations to build the band for short running problem sizes.

Moving beyond small problem sizes, the single benchmark performs well when an optimised calculation method is used. This is because the performance function of the optimised method is very flat, and a single benchmark represents speed as constant for all problem sizes. However at a certain problem size paging occurs and the performance of the single benchmark partition dives. When viewing the results for the naive calculation method one can see that the single benchmark never performs well, this is because the real profile of the naive calculation is far from flat, with many decreasing steps at memory hierarchy boundaries.

The functional and band model based partition methods perform well in both the naive and optimised calculation experiments. The constant probability metric has inconsistent performance, being poor for small problem sizes but having the best performance for larger problem sizes. The adjusted functional model and piece-wise probabilistic partitioning methods follow each-other closely. All

methods are within 5% of each-other.

The second set of two processor experiments uses platforms *hcl07* and *hcl01*. Both these machines have identical processors but differing memory configurations. The external load traces used apply the same level of average load to both processors, 0.5, or 50% utilisation, but *hcl07* experiences almost no fluctuations while *hcl01* has a high standard deviation of 0.6 from the mean of 0.5.

The results in Figure 2.15 show that the overall performance of the metrics is worse than in the previous case where the deviation was lower, but they still achieve greater than 80% of the optimal speed. Most of the same observations from the previous experiment can be made here too, in this case however the constant probability metric performs consistently better than other partitioning methods

Four Processor Experiments

The four processor experiments use machines *hcl05*, *hcl10*, *hcl11* and *hcl15*. Each machine has its own external load trace with various properties. *hcl05* and *hcl11* both have low average external load, and low load fluctuations. *hcl10* and *hcl15* have high average external load and moderate load fluctuations. The results of the experiments are shown in Figure 2.16.

The performance of the partitioning methods is similar to the performance in the two processor experiments. The constant probability metric is inconsistent performing worst of all for some problem sizes and best for others. The adjusted functional model, piece-wise probabilistic metric and angular metric all perform consistently and reach more than 90% of the optimal speed.

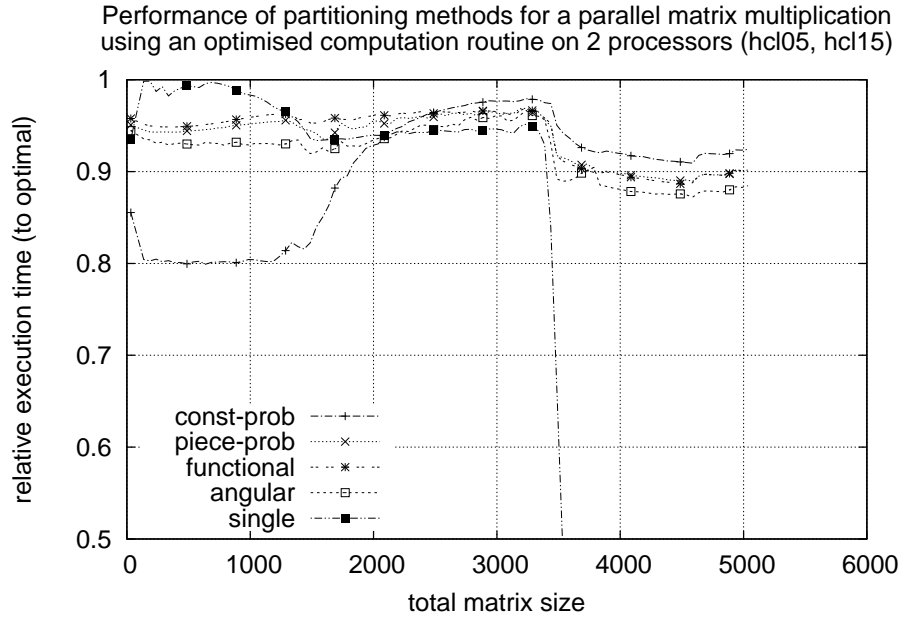
Eight Processor Experiments

The eight processor experiments use machines *hcl01* to *hcl08* with differing external load traces. The performance is the same as the pattern seen in the four and two processor experiments (Figure 2.17). The performance of the single benchmark partitioning method declines consistently until paging occurs and then it drops severely. Of the functional and band model based partitioning methods, the constant probability metric often has the peak performance but it is also poor

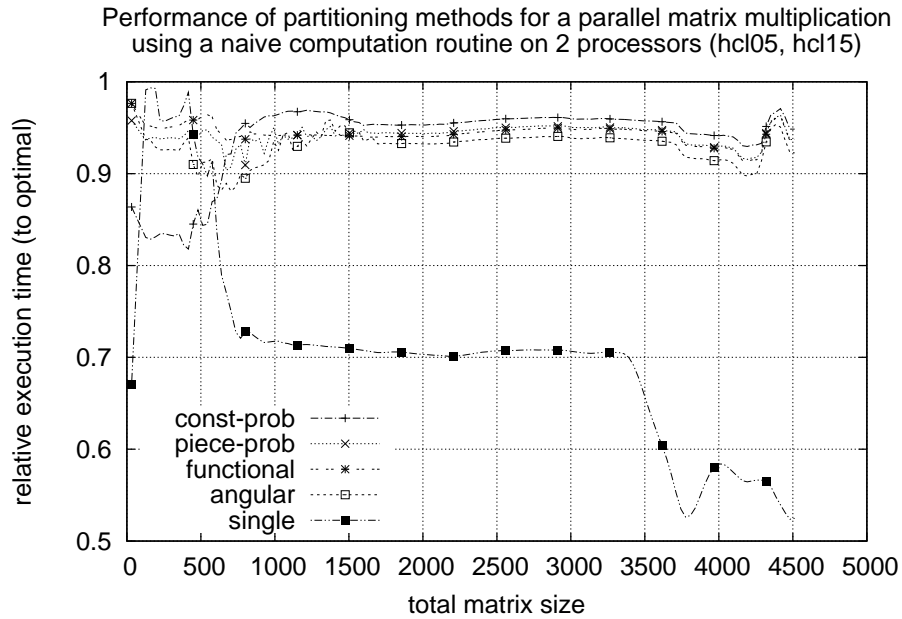
for small problem sizes. The other methods all perform consistently.

Sixteen Processor Experiments

Finally, the sixteen processor experiments use machines *hcl01* to *hcl16* with a wide range of external load traces. The results (Figure 2.18) show lower performance than has been seen before for small problem sizes. With a greater number of processors contributing to the total computation, the execution time for small problem sizes becomes very short, and this may factor in the performance of the band model based metrics. As problem size increases the pattern from previous experiments emerges. The adjusted functional model performs consistently for all problem sizes and the single benchmark is also as expected, performing well for the optimised calculation method until paging occurs and performing poorly for the naive calculation method.

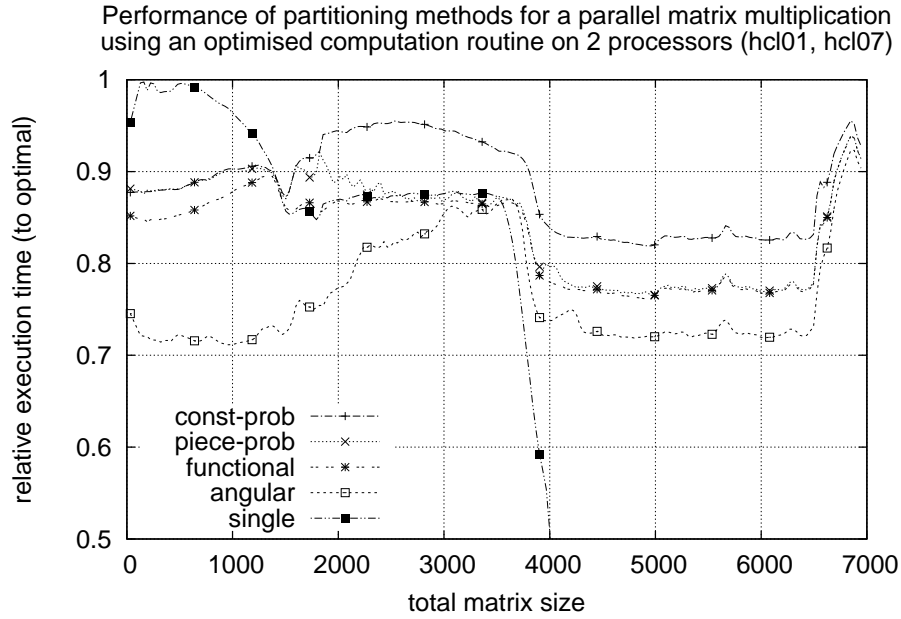


(a) Performance of partition methods when using an optimised computational routine.

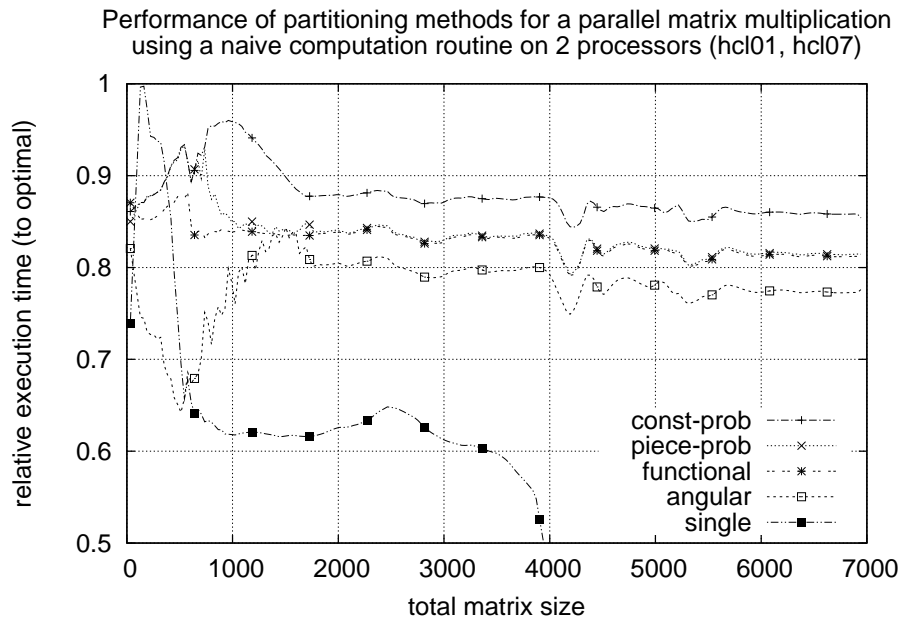


(b) Performance of partition methods when using a naive computational routine.

Figure 2.14: Performance of partitioning methods using two heterogeneous processors, *hcl05* and *hcl15*. *hcl05* operates under low external load (average load index of 0.1), with low fluctuations (deviation of 0.05). *hcl15* operates under a high external load (average of 0.8) and moderate load fluctuation (deviation of 0.3).



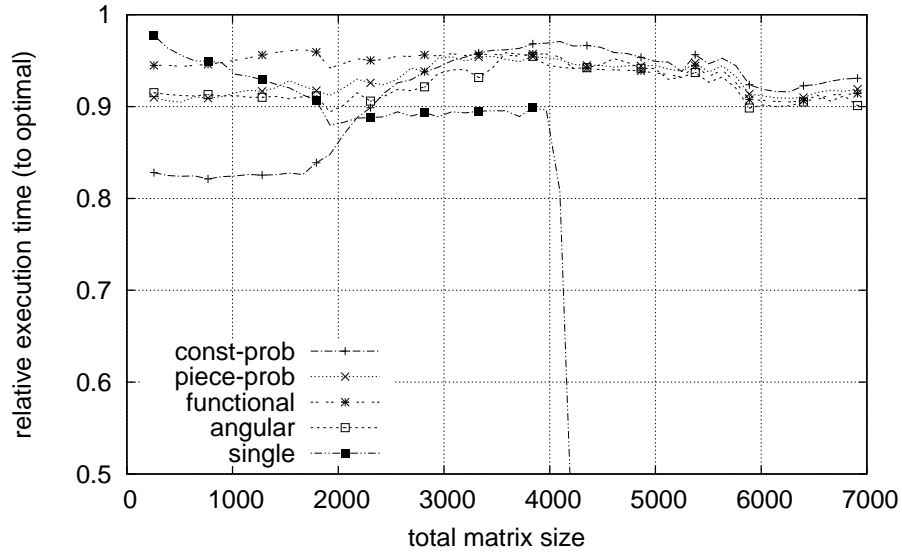
(a) Performance of partition methods when using an optimised computational routine.



(b) Performance of partition methods when using a naive computational routine.

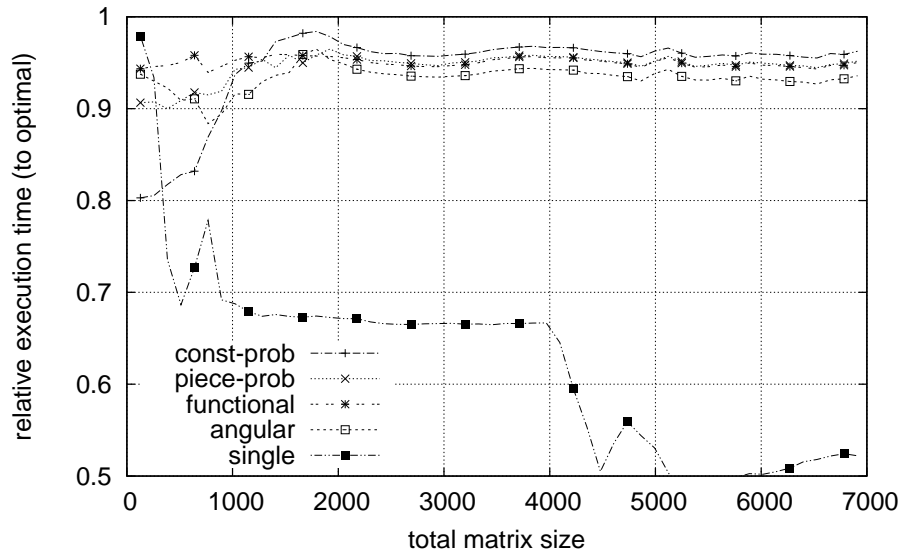
Figure 2.15: Performance of partitioning methods using two processors, *hcl07* and *hcl01*. Processors have identical speed but heterogeneous memory. Both operate under moderate average levels of external load (0.5). *hcl07* experiences low load fluctuations (deviation of 0.05) and *hcl01* experiences high fluctuations (deviation of 0.6).

Performance of partitioning methods for a parallel matrix multiplication using an optimised computation routine on 4 processors (hcl05, hcl10, hcl11, hcl15)



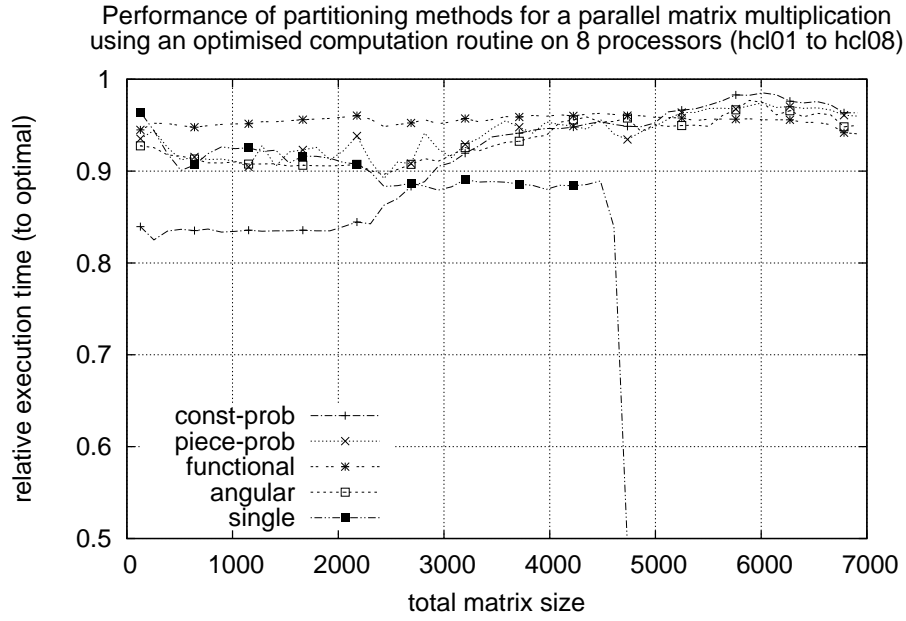
(a) Performance of partition methods when using an optimised computational routine.

Performance of partitioning methods for a parallel matrix multiplication using a naive computation routine on 4 processors (hcl05, hcl10, hcl11, hcl15)

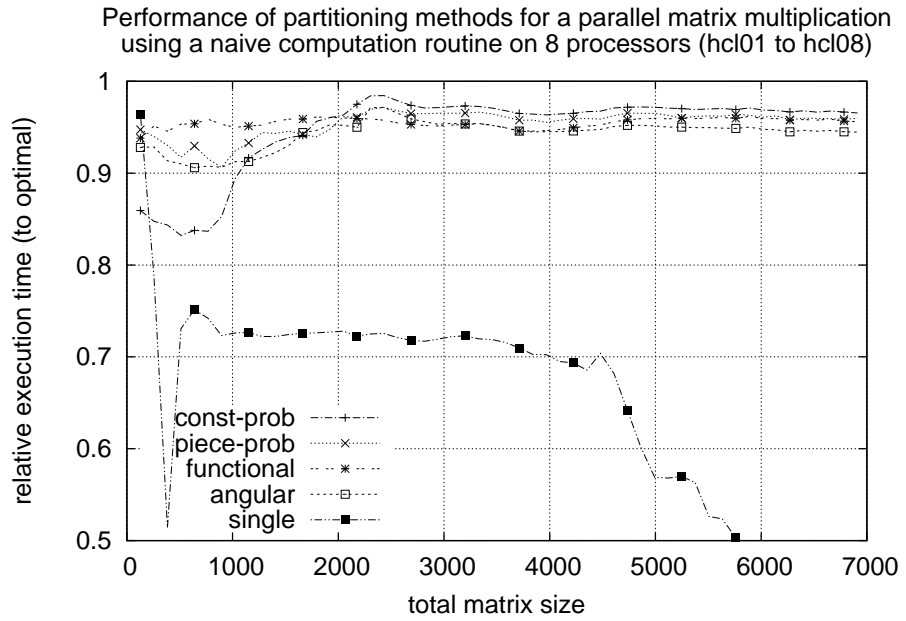


(b) Performance of partition methods when using a naive computational routine.

Figure 2.16: Performance of partitioning methods using 4 processors, *hcl05*, *hcl10*, *hcl11* and *hcl15*. Processors have varied average external loads and load fluctuations.

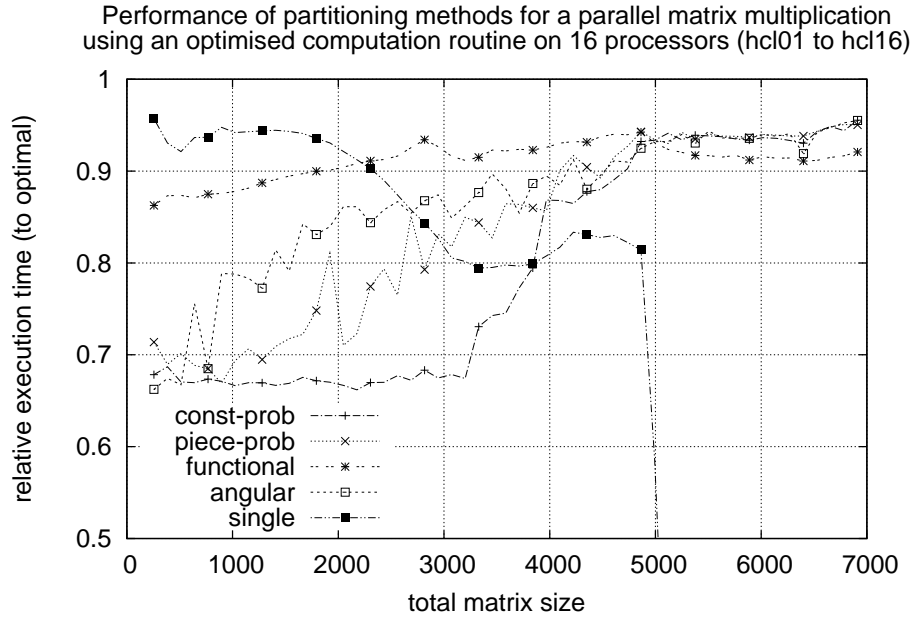


(a) Performance of partition methods when using an optimised computational routine.

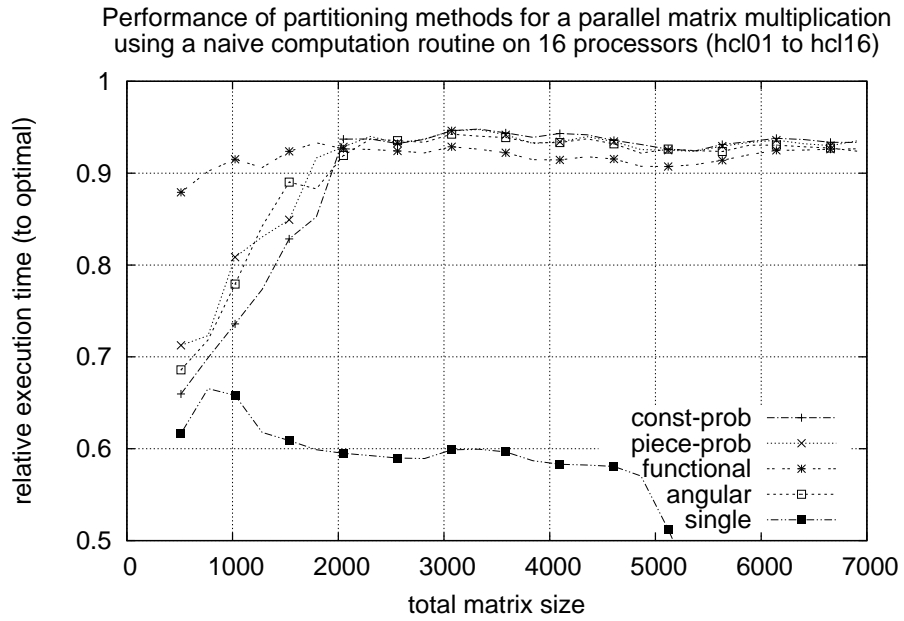


(b) Performance of partition methods when using a naive computational routine.

Figure 2.17: Performance of partitioning methods using 8 processors, *hcl01* to *hcl08*. Processors have varied average external loads and load fluctuations.



(a) Performance of partition methods when using an optimised computational routine.



(b) Performance of partition methods when using a naive computational routine.

Figure 2.18: Performance of partitioning methods using 16 processors, *hcl01* to *hcl16*. Processors have varied average external loads and load fluctuations.

Table 2.1: Machine specifications for Band Model experiments.

Name	Processor	CPUs : Cores	Clock (Ghz)	32 / 64 bit	L1 [I+D] / L2 / L3 Cache Per Processor (KiB)	Bus (Mhz)	Memory (MiB)
hcl01	Pentium 4 (Prescott)	1	3.4	32	16+16 / 1024 / 0	800	1024
hcl02	Pentium 4 (Prescott)	1	3.4	32	16+16 / 1024 / 0	800	1024
hcl03	Pentium 4 (Prescott)	1	3.4	32	16+16 / 1024 / 0	800	1024
hcl04	Pentium 4 (Prescott)	1	3.4	32	16+16 / 1024 / 0	800	1024
hcl05	Xeon (Irwindale)	1	3.6	64	16+16 / 2048 / 0	800	256
hcl06	Xeon (Irwindale)	1	3.0	64	16+16 / 2048 / 0	800	256
hcl07	Pentium 4 550 (Prescott)	1	3.4	32	16+16 / 1024 / 0	800	256
hcl08	Pentium 4 550 (Prescott)	1	3.4	32	16+16 / 1024 / 0	800	256
hcl09	Opteron 265	1:2	1.8	64	2x64+64 / 2x1024 / 0	1000	1024
hcl10	Opteron 265	1:2	1.8	64	2x64+64 / 2x1024 / 0	1000	1024
hcl11	Pentium 4 551 (Prescott)	1	3.2	64	16+16 / 1024 / 0	800	512
hcl12	Pentium 4 551 (Prescott)	1	3.4	64	16+16 / 1024 / 0	800	512
hcl13	Celeron D 331 (Prescott)	1	2.9	64	16+16 / 256 / 0	533	1024
hcl14	Xeon (Nocona)	1	3.4	64	16+16 / 1024 / 0	800	1024
hcl15	Xeon (Nocona)	1	2.8	64	16+16 / 1024 / 0	800	1024
hcl16	Xeon (Irwindale)	1	3.6	64	16+16 / 2048 / 0	800	1024
years	Xeon MP 7110M (Tulsa)	4:8	2.6	64	2x16+16 / 2x1024 / 4096	800	32768

2.3.4 Summary of Experiments

The set of results presented shows that, as expected, the single benchmark is not appropriate for partitioning problems where the performance profile of the computation is not flat and consistent, either because paging occurs or because the computation has not been optimised. Also, for long running computations, the single benchmark, taken at beginning of runtime will not be sufficient to achieve optimal balance.

The functional and band based methods perform well for a generalised problem where the performance profile may or may not be flat. They also accurately represent the variance in performance at single points in the profile. Of the methods using these models for problem partitioning, the adjusted functional model and piece-wise probabilistic metric have the best results. The constant probability metric is too inconsistent when partitioning small problem sizes and the angular metric performs slightly worse than the functional or piece-wise probabilistic metric.

The adjusted functional model and piece-wise probabilistic metric both perform consistently well across all experiments. They also have almost equal overall performance. The key difference between these is that partitioning with the piece-wise probabilistic method is performed using a heuristic search while partitioning problems using the functional is by an efficient algorithm presented in [67]. This algorithm is guaranteed to converge on the solution of the partitioning problem and has a worst case complexity of $O(p^3 \times \log_2(n))$ (where p is number of processors and n is the total problem size to partition). Finding a partition based on the adjusted functional model is far simpler than finding one using the piece-wise probabilistic metric. As the performance of both partitions is so close, the adjusted Functional Performance Model is proposed as the best representation of the speed variance of non-dedicated processors. This does not discard the Band Performance Model as the adjusted model is a product of the band.

2.4 Summary

In this chapter the Band Performance Model has been introduced. This model extends the Functional Performance Model by adding a prediction of processor performance variance to it.

The Functional Performance Model may also be considered as a product of the Band Model, where the performance variance is averaged to a single valued function. This model is described as an adjusted Functional Performance Model.

The formulation of a simple and detailed Band Performance Model has been described and methods of evaluating the “goodness” of a problem distribution using these models has been proposed. A Genetic Algorithm has been implemented to solve the problem of partitioning a workload while maximising the various goodness metric proposed.

Experiments simulating the execution time of problem distributions using a variety of model types have concluded that though the simple and detailed Band Performance Models result in good load balance, the improvement over an adjusted functional model is marginal at best.

As a result, algorithms for using the Band Models to partition problems have not been developed beyond the implementation of the genetic algorithm and the adjusted Functional Performance Model is suggested as the best representation of processor speed on platforms with load fluctuations.

Chapter 3

Optimised Construction of the Band Performance Model

This chapter presents an optimised procedure for building a piece-wise linear approximation of the Band Performance Model (BPM). The Model represents the variation in execution speed of a problem on a processor, across a range of different problem sizes. The formulation of the Band Performance Model has been described in Chapter 2. Briefly, it is a product of two inputs:

- a history of load observations which describe fluctuations in processor availability over various time periods. This history is used to predict maximum and minimum load over time periods t in two piece-wise linear functions: $l_{min}(t)$ and $l_{max}(t)$.
- a piece-wise linear function representing ideal execution speed as a function of problem size $S_i(x)$, and ideal execution time $T_i(x)$.

The history of load observations may be obtained at almost no cost and is not a limiting factor in the practical use of a Band Performance Model. However the ideal speed function is expensive to discover. Each end-point of the line segments that make up the piece-wise linear function corresponds to an experimentally obtained execution speed. Obtaining the speeds requires the actual execution of the routine computing a specific problem size. A detailed model may require many of these points and the duration of experiments may be impractically long.

Example speed band approximations, showing the number of experimental points required for the approximation, are illustrated in Figure 3.1. The procedure presented in this chapter minimises the number of such points that must be measured to build an accurate approximation of the Band. The resultant approximation gives the speed of the processor for any problem size with an accuracy that is within the inherent deviation of the processor's performance. For simplicity in problem partitioning and task scheduling, a Functional Performance Model (FPM) may be placed through the middle of the Band, and this FPM will also have a degree of accuracy that corresponds with the known performance fluctuations of the processor.

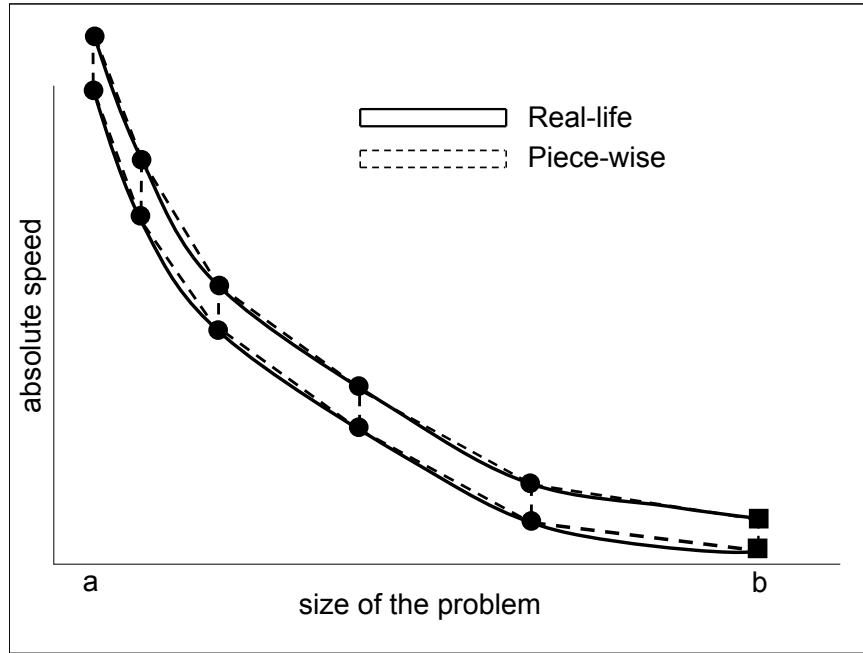
In this chapter the problem of optimally approximating the Band model is formalised and an algorithm that attempts to solve this problem is described. Experimental results from an implementation of this algorithm are presented to show the speed-up in construction time that is achieved.

3.1 Problem Formulation

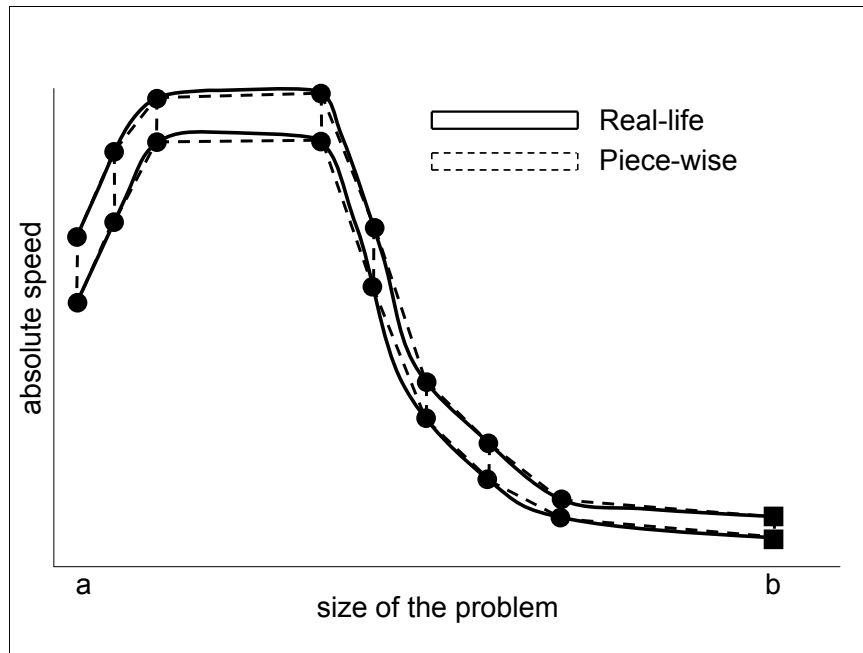
For a specific routine in a real-life situation, the performance demonstrated by the processor is characterised by a band representing the speed of the processor across various problem sizes. The width of such a band describes the level of fluctuation in the speed due to changes in external load.

The problem is to experimentally find an approximation of the speed band of the processor that can represent the real-life band with sufficient accuracy but spend a minimal amount of experimental time in the building process. One such approximation is a piece-wise linear function which accurately represents the real-life speed band with a finite number of points.

The piece-wise linear functional approximation of the speed band is built using a set of experimentally obtained points for different problem sizes. The problem size, x , is equivalent to the amount of data stored and processed by the routine. This is usually governed by the input parameters of the routine. To obtain an experimental point of a particular problem size the routine is executed with the parameters corresponding to the problem size. The execution time is measured under ideal conditions, with no other contending processes to give



(a) The speed band of a problem which uses the memory hierarchy inefficiently (using 5 experimental points).



(b) The speed band of a problem which uses the memory hierarchy efficiently (built using 8 experimental points).

Figure 3.1: Using piecewise linear approximation to build speed bands. Circular points are experimentally obtained, square points are calculating using heuristics.

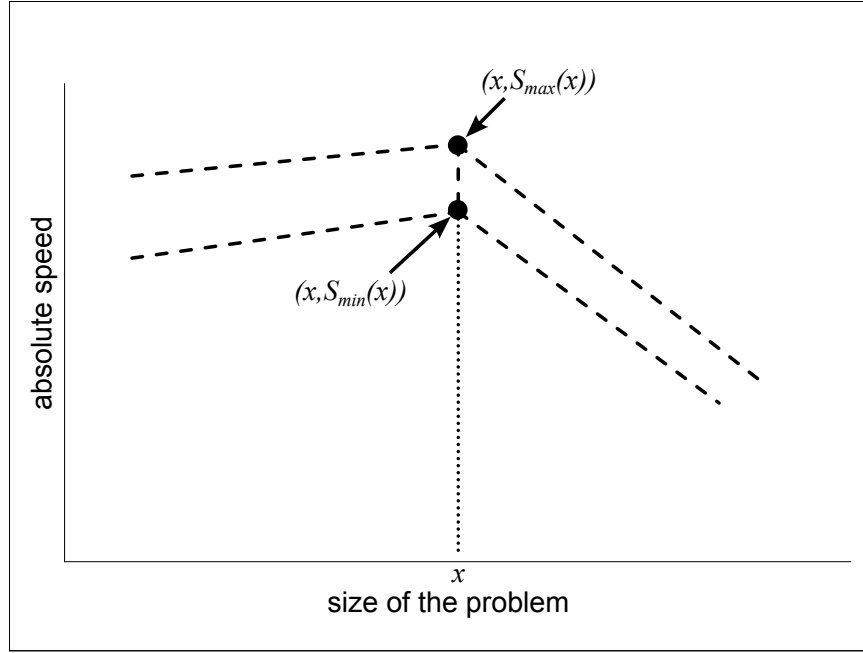
$T_i(x)$. The ideal speed $S_i(x)$ is equal to the volume of computations divided by the $T_i(x)$. The volume of computations may be given by the complexity of the problem.

A prediction of the maximum and minimum external load that would occur during the execution period of the problem size x are extracted from a record of historical load fluctuations. The exact manner is described in Chapter 2. These predictions are applied to the ideal speed resulting in maximum and minimum predicted speeds of execution, $S_{max}(x)$ and $S_{min}(x)$. In this manner, the experimental point is converted to a range of speeds, connected by a vertical line, from $(x, S_{max}(x))$ to $(x, S_{min}(x))$. This line is denoted as the “cut” of the real-life speed band (see Figure 3.2a). The difference between $S_{min}(x)$ and $S_{max}(x)$ represents the level of fluctuation in speed due to changes in external load during the execution of a problem of size x .

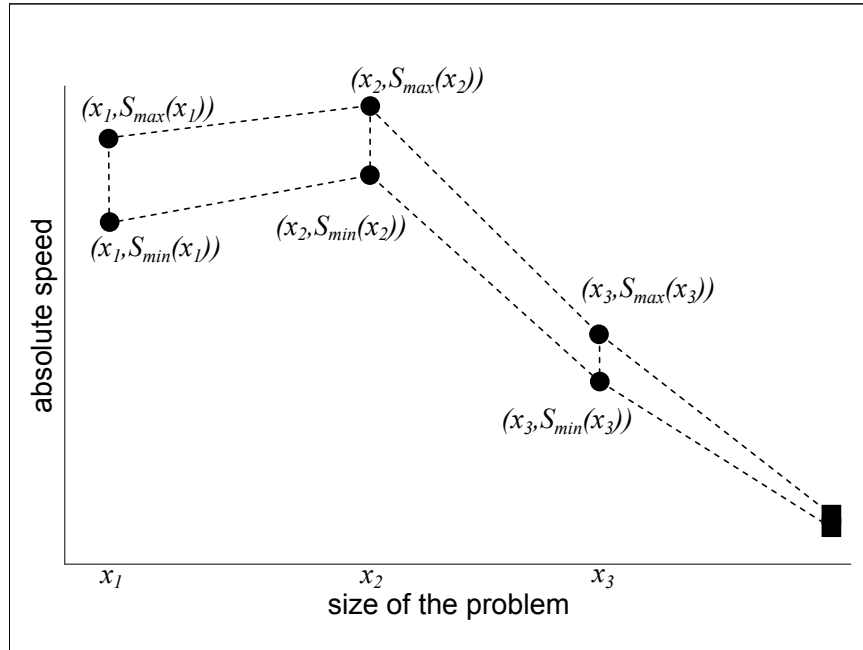
The piece-wise linear approximation is obtained by connecting the cuts derived from the experimentally obtained execution speeds as illustrated in Figure 3.2b. The problem of building the approximation is to find a set of experimental points that can represent the speed band with sufficient accuracy and at the same time spend minimum experimental time building the approximation.

Formally, the problem is defined as follows:

- Given the functions $l_{min}(t)$ and $l_{max}(t)$ (that are functions of time, characterising the level of fluctuation in load over time).
- obtain a set of n experimental points representing the piece-wise linear function approximation of the speed band of a processor, each point representing a cut given by $(x_j, S_{max}(x_j))$ and $(x_j, S_{min}(x_j))$, where x_j is the size of the problem and $S_{max}(x_j)$ and $S_{min}(x_j)$ are speeds calculated based on the functions $l_{min}(t)$, $l_{max}(t)$ and the ideal speed $S_i(x_j)$ at the j th point, such that:
 - The non-empty intersectional area of the piece-wise linear function approximation with the real-life speed band is a simply connected surface. (A surface is said to be connected if a path can be drawn from every point contained within its boundaries to every other point,



(a) The speeds $S_{\max}(x)$ and $S_{\min}(x)$ representing a cut of the real band used to build the piece-wise linear approximation.



(b) A piece-wise linear approximation built by connecting the cuts.

Figure 3.2: Illustration of how a piece-wise function approximates a band.

a topological space is simply connected if its path connected and it has no holes.) This is illustrated in Figure 3.3.

- The sum of times, $\sum_{j=1}^n T_j$, is minimal, where T_j is the experimental time used to obtain point j .

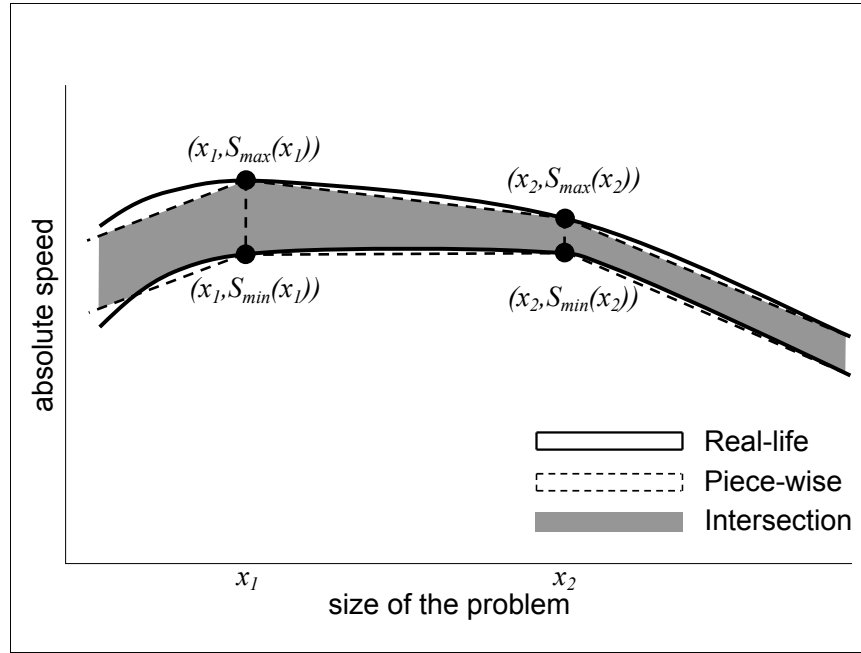


Figure 3.3: The non-empty intersectional area of a piece-wise linear function approximation of the real-life speed band is a simply connected surface.

3.2 Assumptions

The procedure presented operates under a number of reasonable assumptions on the shape of the real-life band that it approximates. They are as follows:

1. The upper and lower curves of the speed band are continuous functions of the problem size.
2. The permissible shapes of the real-life speed band are:
 - (a) The upper and lower curve are both a non-increasing function of the size of the problem for all problem sizes (i.e. Figure 3.4a).

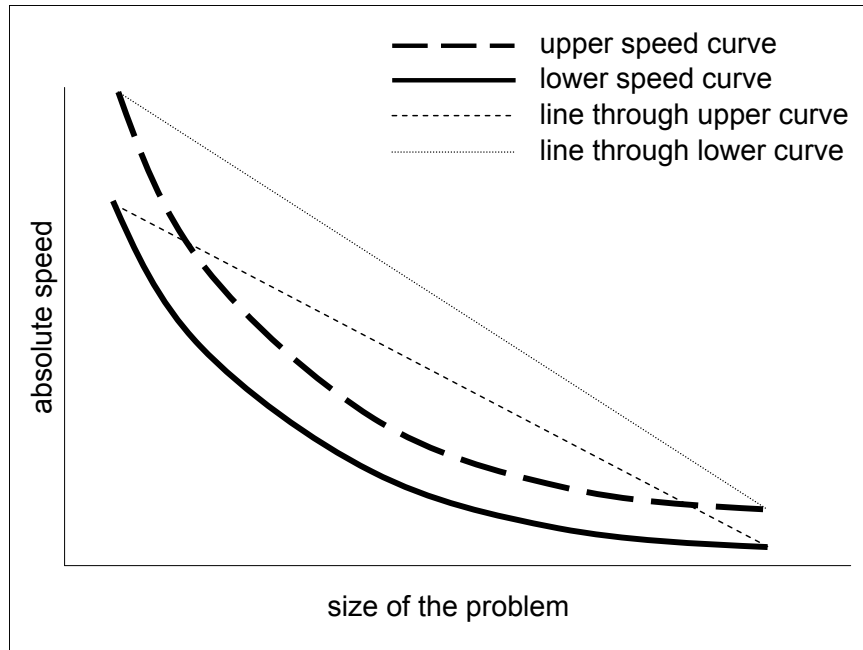
- (b) The upper and lower curve are both an increasing function of the problem size, followed by a non-increasing function of the problem size (i.e. Figure 3.4b).
- 3. A straight line intersects the upper curve of the real-life speed band in no more than one point between its endpoints. Also, a straight line intersects the lower curve of the real-life speed band in no more than one point between its end points. This is illustrated for routines which implement inefficient memory access, as per assumption 2a in Figure 3.4a, and for routines which implement efficient memory access, as per assumption 2b in Figure 3.4b.
- 4. The width of the real-life speed band, representing the level of speed fluctuations due to changes in load, decreases as the problem size, and consequently problem execution time, increases.

Experiments with diverse scientific routines and heterogeneous platforms presented in [90] have shown that the above assumptions are realistic and that the speed band of the processor can be approximated accurately by a band that satisfies these assumptions.

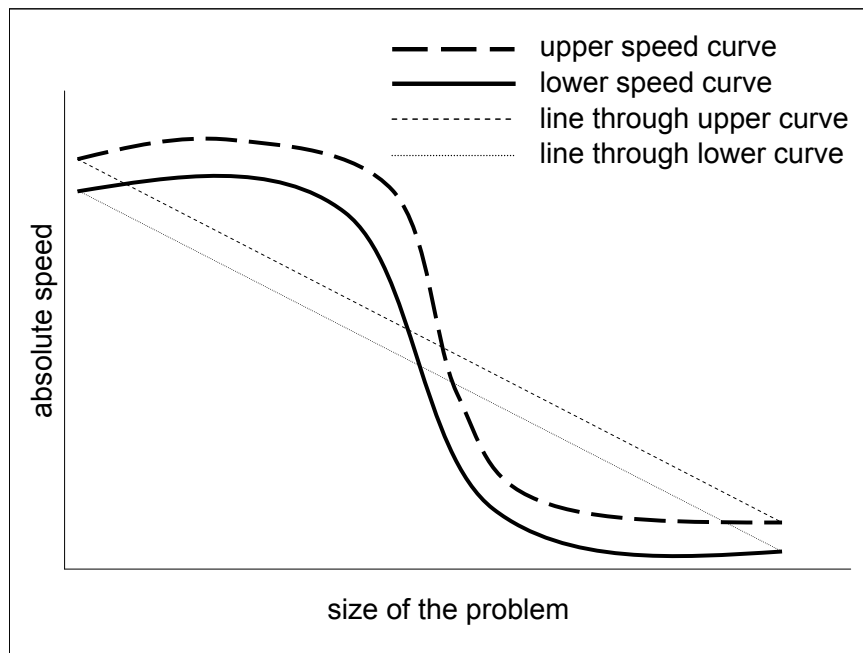
There is also a constraint on the environment that the model is used in. The effects on the performance of a processor caused by several users running heavy computational tasks simultaneously is not considered. It is assumed that only one computational task will be executed at a time, but that this task may execute in the presence of other programs performing computations and communications of a relatively light-weight nature. For example, desktop applications such as email clients, browsers, editors and so on. The Band Model can approximate the fluctuation in processor performance due to such tasks and the scenario fits the notion of a “Desktop Cluster”.

3.3 Definitions

Before presenting the procedure to build a piece-wise linear function approximation of the speed band of a processor, some objects, operations and relations



(a) Shape of real-life speed band of a processor for a routine that uses the memory hierarchy inefficiently.



(b) Shape of a real-life speed band of a processor for a routine that uses the memory hierarchy efficiently.

Figure 3.4: The two general shapes of the real-life band.

must be defined.

1. The cut $C(x)$ is a line segment connecting the points $(x, S_{min}(x))$ and $(x, S_{max}(x))$.
2. The “interval” $I(x)$ is the horizontal projection of the interval $S_{min}(x), S_{max}(x)$ for a problem size x .
3. $I(x) \leq I(y)$ if and only if $S_{max}(x) \leq S_{max}(y)$ and $S_{min}(x) \leq S_{min}(y)$
4. $I(x) \cap I(y)$ represents the intersection between the intervals $(S_{min}(x), S_{max}(x))$ and $(S_{min}(y), S_{max}(y))$. If $I(x) \cap I(y) = \emptyset$ where \emptyset represents an empty set with no elements, then the intervals are disjoint. If $I(x) \cap I(y) = I(y)$ then the interval $(S_{min}(x), S_{max}(x))$ contains the interval $(S_{min}(y), S_{max}(y))$, that is, $S_{max}(x) \geq S_{max}(y)$ and $S_{min}(x) \leq S_{min}(y)$.
5. $I(x) = I(y)$ if and only if $I(x) \leq I(y)$ and $I(y) \leq I(x)$

3.4 Speed Band Approximation Algorithm

The algorithm, titled Geometric Bisection Building Procedure (GBBP), is presented in the following steps, and illustrated in Figures 3.5, 3.6 and 3.8.

Step 1: Initialisation We select an interval of problem sizes, $[a, b]$, where a is some small size and b is a problem size large enough to make the speed of the processor practically zero. The problem size a should be sufficiently small that the problem fits in one of the upper levels of the memory hierarchy of the computer (L1 Cache for instance) and b may be equivalent to the maximum amount of memory that the computer will allow to be allocated on the heap. The speeds of execution at a , $S_{min}(a)$ and $S_{max}(a)$ are obtained experimentally, while the speed at b is set at 0. The initial approximation is a band connecting the cuts $C(a)$ and $C(b)$ as illustrated in Figure 3.5a.

Step 2: Approximation of the increasing part of the speed band The band cut at $2a$ is obtained experimentally.

If $I(2a) \leq I(a)$ or $I(2a) \cap I(a) = I(2a)$ (i.e. the increasing section of the band ends at point a), then the current trapezoidal approximation is replaced by two trapezoidal connected bands, the first connecting cuts $C(a)$ and $C(2a)$ and the second connecting cuts $C(2a)$ and $C(b)$. Variable x_{left} is set to $2a$ and x_{right} is set to b and the algorithm proceeds to Step 3, where the non-increasing portion of the band will be approximated.

If $I(a) < I(2a)$ (i.e. the speed band is increasing with problem size) then Step 2 is applied recursively to pairs $(ka, (k+1)a)$ until $I((k+1)a) \leq I(ka)$ or $I((k+1)a) \cap I(ka) = I((k+1)a)$. Then, the current trapezoidal approximation of the speed band in the interval $[ka, b]$ is replaced by two connected bands, the first connecting cuts $C(ka)$ and $C((k+1)a)$, the second connecting the cuts $C((k+1)a)$ and $C(b)$. The x_{left} is set to $(k+1)a$, x_{right} is set to b and the algorithm proceeds to Step 3. This is illustrated in Figure 3.5b.

It should be noted that the experimental time take to obtain cuts of the speed band at sizes $\{a, 2a, \dots, (k+1)a\}$ is relatively small (in the order of seconds) compared with the time to measure large problem sizes (minutes to hours).

Step 3: Approximation of the non-increasing section The interval from $[x_{left}, x_{right}]$ is bisected at x_{b1} into sub-intervals $[x_{left}, x_{b1}]$ and $[x_{b1}, x_{right}]$ of equal length. The current approximation, given by the band connecting cuts $C(x_{left})$ and $C(x_{right})$ is intersected with a line $x = x_{b1}$ giving an approximation of the band cut at x_{b1} : $C'(x_{b1})$. The actual cut at $C(x_{b1})$ is also obtained, experimentally, at problem size x_{b1} .

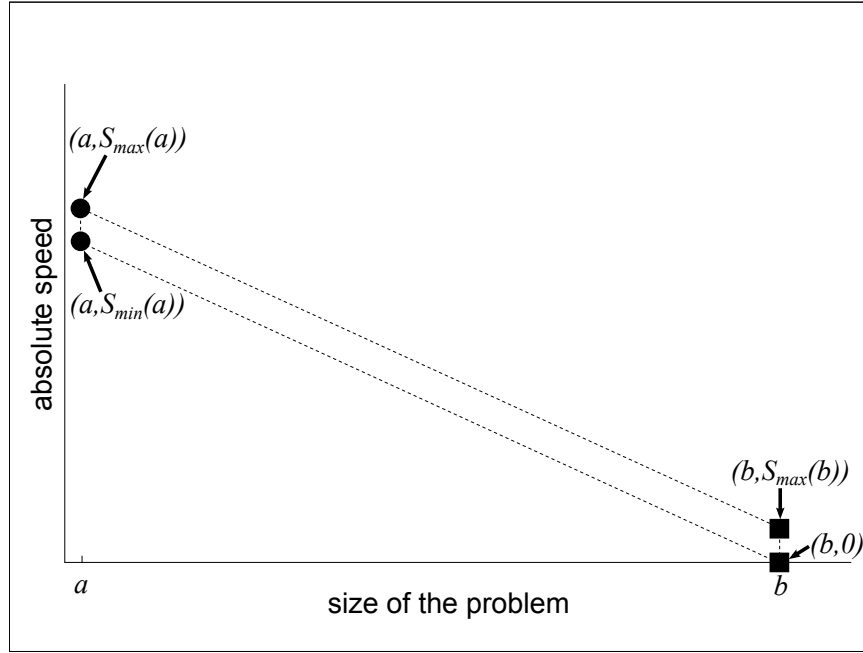
- If $I(x_{left}) \cap I(x_{b1}) \neq \emptyset$, the current trapezoidal approximation of the speed band is replaced by two connected bands, the first connecting cuts $C(x_{left})$ and $C(x_{b1})$, the second connecting cuts $C(x_{b1})$ and $C(x_{right})$. This is illustrated in Figure 3.6a. Construction of the band in the interval $[x_{left}, x_{b1}]$ is halted and Step 3 is applied recursively to the interval $[x_{b1}, x_{right}]$.
- If $I(x_{left}) \cap I(x_{b1}) = \emptyset$ and $I(x_{b1}) \cap I(x_{right}) \neq \emptyset$, the current trape-

zoidal approximation of the speed band is replaced by two connected bands, the first connecting cuts $C(x_{left})$ and $C(x_{b1})$, the second connecting cuts $C(x_{b1})$ and $C(x_{right})$. This is illustrated in Figure 3.6b. Construction of the band in the interval $[x_{b1}, x_{right}]$ is halted and Step 3 is applied recursively to the interval $[x_{left}, x_{b1}]$.

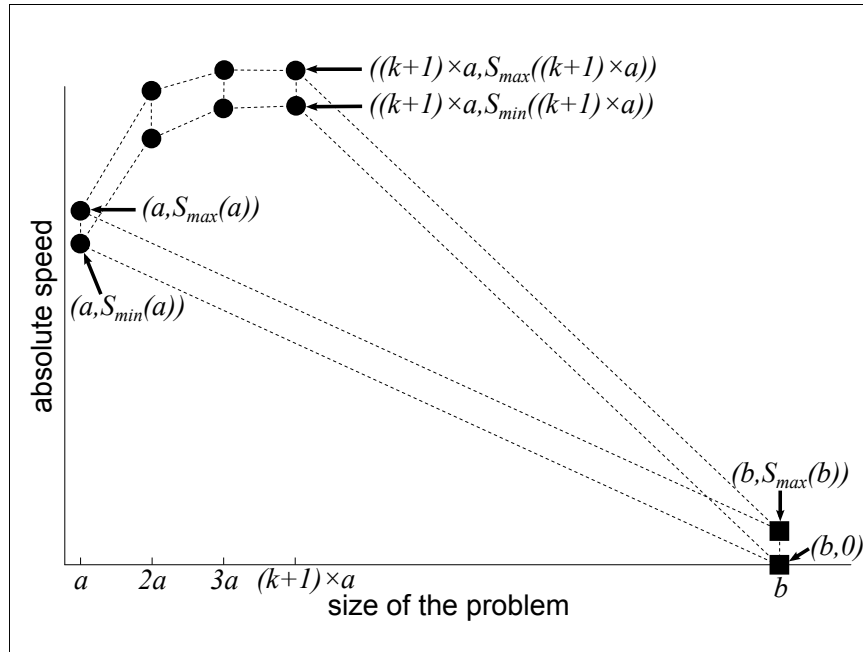
- If $I(x_{left}) \cap I(x_{b1}) = \emptyset$ and $I(x_{b1}) \cap I(x_{right}) = \emptyset$ and $I'(x_{b1}) < I(x_{b1})$ (that is $I'(x_{b1}) \leq I(x_{b1})$ and $I'(x_{b1}) \cap I(x_{b1}) = \emptyset$), the current trapezoidal approximation of the speed band is replaced by two connected bands, the first connecting cuts $C(x_{left})$ and $C(x_{b1})$, the second connecting cuts $C(x_{b1})$ and $C(x_{right})$. This is illustrated in Figure 3.7. Step 3 is applied recursively to both intervals, $[x_{left}, x_{b1}]$ and $[x_{b1}, x_{right}]$.
- If $I(x_{left}) \cap I(x_{b1}) = \emptyset$ and $I(x_{b1}) \cap I(x_{right}) = \emptyset$ and $I'(x_{b1}) \cap I(x_{b1}) \neq \emptyset$, then there are two possible forms of the real life band that the algorithm must discover. These are illustrated in Figures 3.8a and 3.8b. In order to determine which shape the band takes, the intervals $[x_{left}, x_{b1}]$ and $[x_{b1}, x_{right}]$ are tested as follows:
 - First, interval $[x_{left}, x_{b1}]$ is bisected at point x_{b2} and the cut $C(x_{b2})$ is obtained experimentally. The current approximation of the cut $C'(x_{b2})$ is also obtained from an intersection of the vertical line: $x = x_{b2}$ and the trapezoidal approximation of the band.
 - * If $I(x_{b2}) \cap I'(x_{b2}) \neq \emptyset$, then the real-life band corresponds to the shape in Figure 3.8a. Therefore, construction of the band in the interval $[x_{left}, x_{b1}]$ is halted. The current trapezoidal approximation is replaced by two connected bands, the first connecting cuts $C(x_{left})$ and $C(x_{b2})$, the second connecting cuts $C(x_{b2})$ and $C(x_{b1})$. Since the cut at x_{b2} has been obtained it is included in the model, even though the previous approximation was adequate at this point.
 - * If $I(x_{b2}) \cap I'(x_{b2}) = \emptyset$, then the real-life band corresponds to the shape in Figure 3.8b. Therefore, construction is continued by recursively applying Step 3 on intervals $[x_{left}, x_{b2}]$ and $[x_{b2}, x_{b1}]$.

- Similarly, interval $[x_{b1}, x_{right}]$ is bisected at point x_{b3} and the cut $C(x_{b3})$ is obtained experimentally. The current approximation of the cut $C'(x_{b3})$ is also obtained from an intersection of a line $x = x_{b3}$ and the trapezoidal approximation of the band.
 - * If $I(x_{b3}) \cap I'(x_{b3}) \neq \emptyset$, then the real-life band corresponds to the shape in Figure 3.8a. Therefore, construction of the band in the interval $[x_{b1}, x_{right}]$ is halted. The current trapezoidal approximation is replaced by two connected bands, the first connecting cuts $C(x_{b1})$ and $C(x_{b3})$, the second connecting cuts $C(x_{b3})$ and $C(x_{right})$. Since the cut at x_{b3} has been obtained it is included in the model, even though the previous approximation was adequate at this point.
 - * If $I(x_{b3}) \cap I'(x_{b3}) = \emptyset$, then the real-life band corresponds to the shape in Figure 3.8b. Therefore, construction is continued by recursively applying Step 3 on intervals $[x_{b1}, x_{b3}]$ and $[x_{b3}, x_{right}]$.

This completes the definition of GBBP. The algorithm terminates as all construction intervals are finalised. Illustrations of the completed band approximations are shown in Figure 3.1.

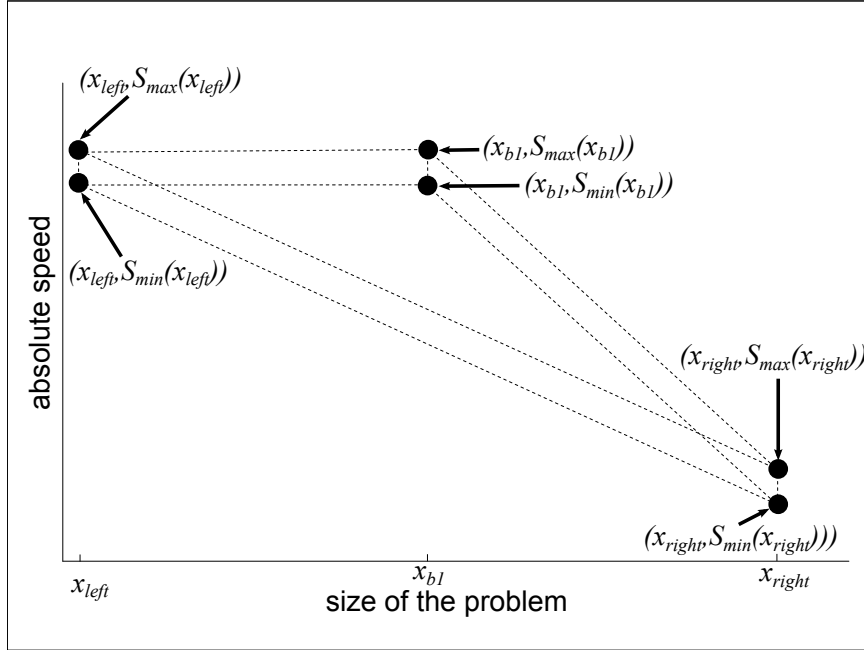


(a) Initial approximation of the speed band.

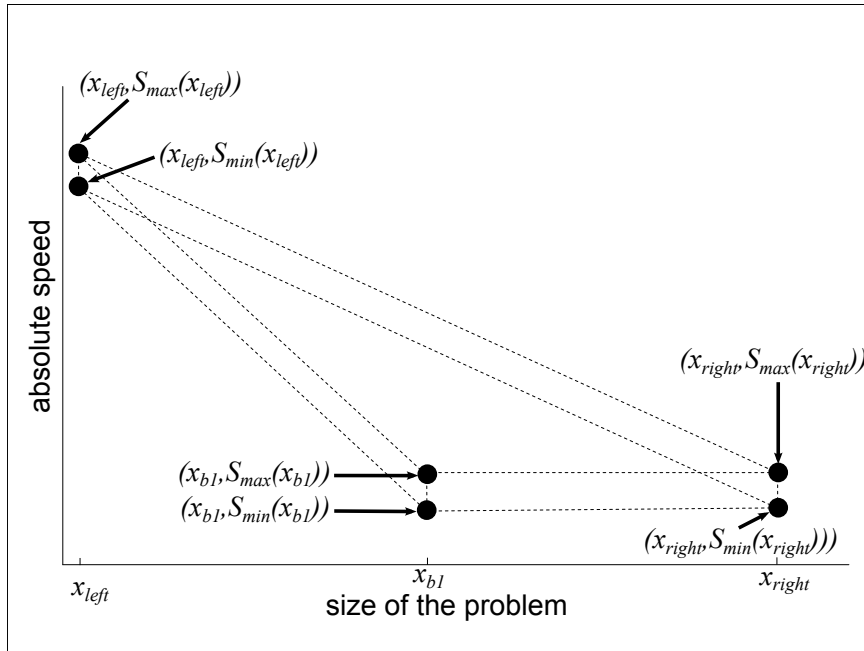


(b) Steps in the approximation of any performance increase at the start of the building procedure.

Figure 3.5: Illustrations of steps of the Geometric Bisection Building Procedure.



(a) Finalising construction where a region of higher constant performance is detected.



(b) Finalising construction where a region of lower constant performance is detected.

Figure 3.6: Further Illustrations of steps of the Geometric Bisection Building Procedure.

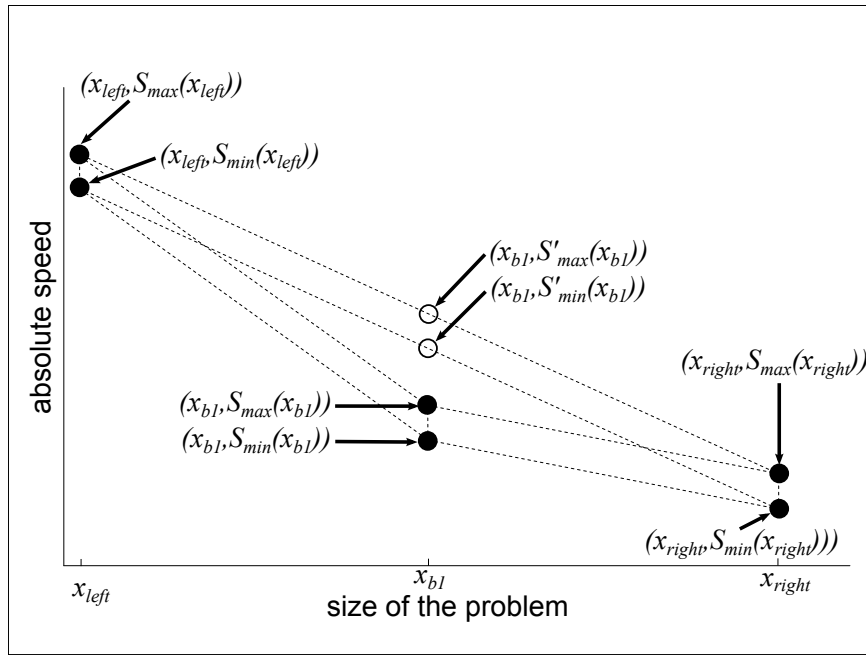
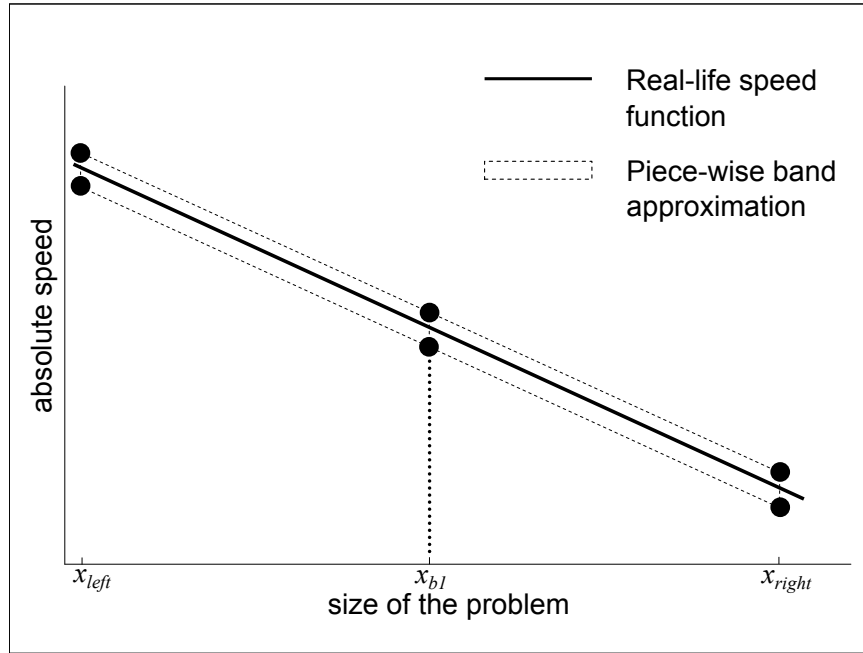
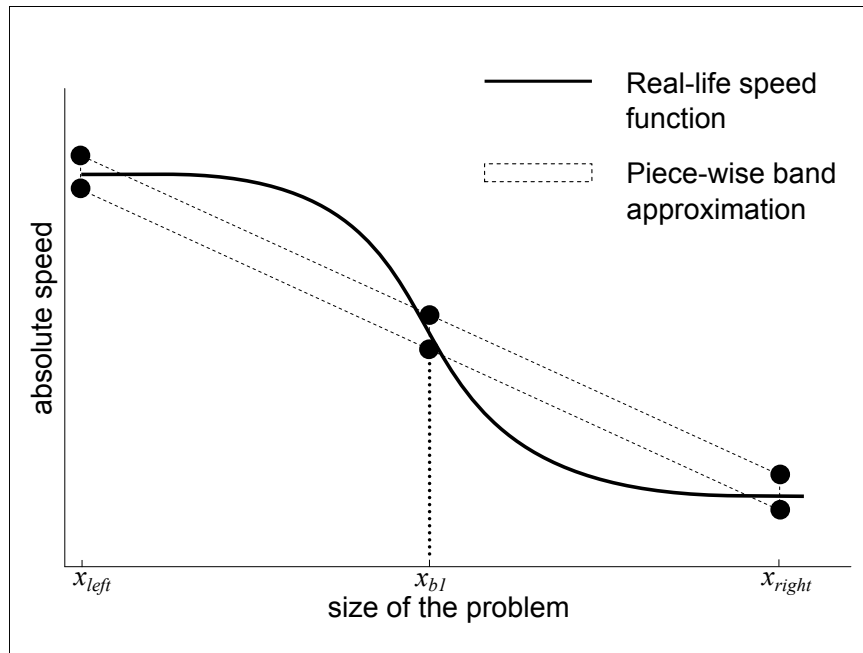


Figure 3.7: Construction cannot be finalised in either left or right intervals as the performance is neither constant nor already approximated by the model.



(a) Possible shape of real-life band: $C(x_{b1})$ and regions to the left or right are all accurately approximated, i.e. there is no inflection in the model.



(b) Possible shape of real-life band: $C(x_{b1})$ is accurately approximated but regions to the left and right are not, i.e. there is inflection in the model.

Figure 3.8: Illustrations of possible shapes of the model when the speed at a new bisection point is already approximated.

Table 3.1: Machine specifications for GBBP experiments.

	Arch.	CPU (MHz)	Total Memory (KiB)	Available Memory (KiB)	Cache (KiB)	Problem size end points $a:b$	
						Matrix Multi- plication (naive & ATLAS)	Cholesky Factorisa- tion
X1	Linux / Intel Xeon	1977	1033908	460368	512	100:13000	100:19500
X2	SunOS / Ultra- SPARC Ili	440	524288	401408	2048	100:7000	100:13000

3.5 GBBP Experiments

Two systems: a 2GHz Pentium Xeon Linux workstation with 1GB of RAM (X1) and a 440MHz UltraSPARC Ili Solaris workstation with 512MB of RAM (X2), are used in experiments with the Geometric Bisection Building Procedure. Each machine is integrated in the departmental network and operates with roughly 400MB of available memory, the rest being consumed by operating system and user processes. These extra processes perform basic computations and communications associated with email clients, browser sessions, text editing and audio use. Their specifications can be found in Table 3.1.

Three applications are used to demonstrate the efficiency of GBBP in the construction of the piece-wise linear function approximation of the speed band of a processor. The first application is Cholesky Factorisation of a dense square matrix employing the LAPACK [91] routine **dpotrf**. The second is a matrix-matrix multiplication of two dense matrices using memory hierarchy inefficiently. The third application is a matrix-matrix multiplication of two dense matrices using the level-3 BLAS routine **dgemm** [92], supplied by the Automatically Tuned Linear Algebra Software package (ATLAS) [93]. This routine uses the memory hierarchy efficiently.

CPU	Naive Matrix Multiplication	ATLAS Matrix Multiplication	Cholesky Factorisation
	Speed Up (<i>Number of Points used by GBBP</i>)		
X1	5.9 (5)	8.5 (7)	6.5 (19)
X2	5.7 (5)	5.7 (5)	15 (8)

(a) Speedup of GBBP over naive construction procedure for different applications.

CPU	Naive Matrix Multiplication	ATLAS Matrix Multiplication	Cholesky Factorisation
X1	5	7	19
X2	5	5	8

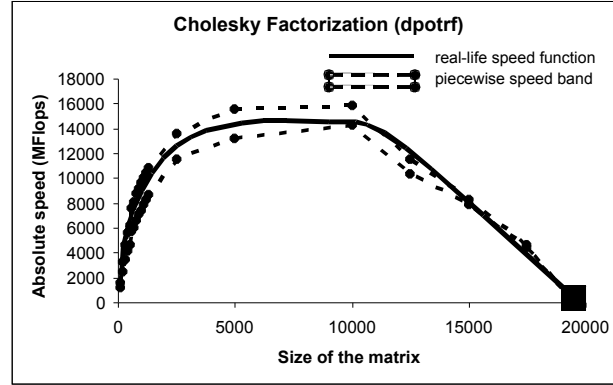
(b) Number of points used by GBBP in construction of models for different applications.

Table 3.2: Results of experimentation with GBBP.

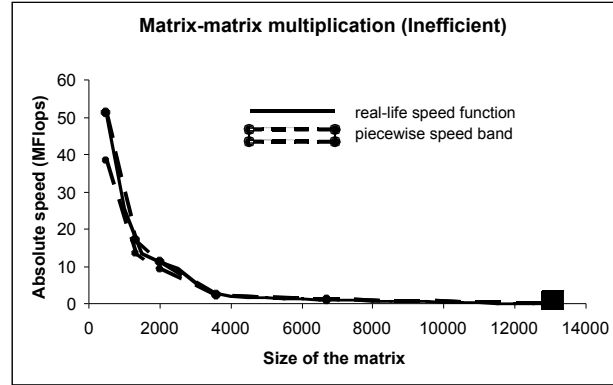
Figures 3.9 and 3.10 show the real life speed function and piece-wise linear function approximation of the speed band for processors X1 and X2 on a selection of the demonstration applications. Each point in the approximations is obtained by execution of the application for the problem size at that point. The absolute speed of the processor for this problem size is found by dividing the volume of computations by the measured execution time. For the applications described the volume of computations is given by their floating point complexity.

Table 3.2a shows the speed-up of GBBP over a naive construction procedure. The naive procedure divides the interval $[a, b]$ of problem sizes into n equal points, the application is then executed for each of the problem sizes $\{a, (a + (b - a)/n), (a + 2(b - a)/n), \dots, b\}$, to obtain the experimental points used in building the piece-wise linear function approximation of the speed band. In the experiments 20 points were used to achieve a reasonable accuracy over the range $[a, b]$. The speed-up calculated is equal to the ratio of experimental time taken to build the band approximation using the naive procedure over the time taken using GBBP.

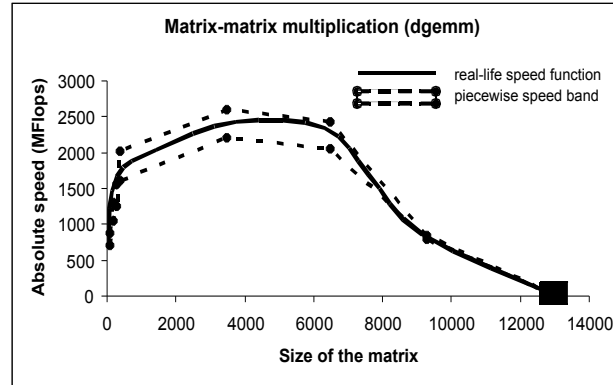
In Table 3.2b, for processor X1 and the Cholesky Factorisation routine, it can be seen that the number of experimentation points required by GBBP was close to the naive construction method (19 versus 20), however the execution speed up remained high. Figure 3.9a reveals that a large number of experimental



(a) Speed Band for Cholesky Factorisation on X1.

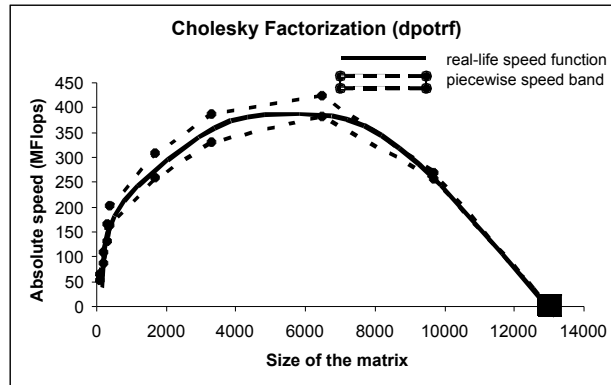


(b) Speed Band for Naive Matrix Multiplication on X1.

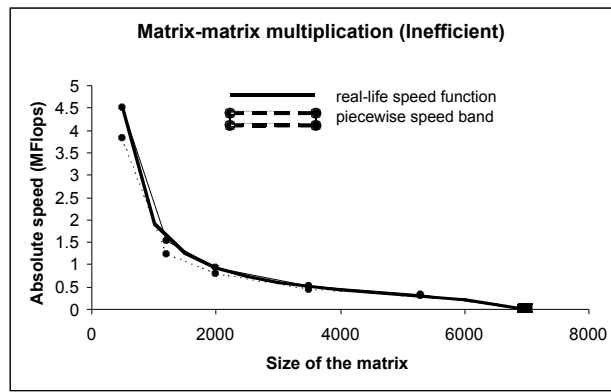


(c) Speed Band for optimised ATLAS Matrix Multiplication on X1.

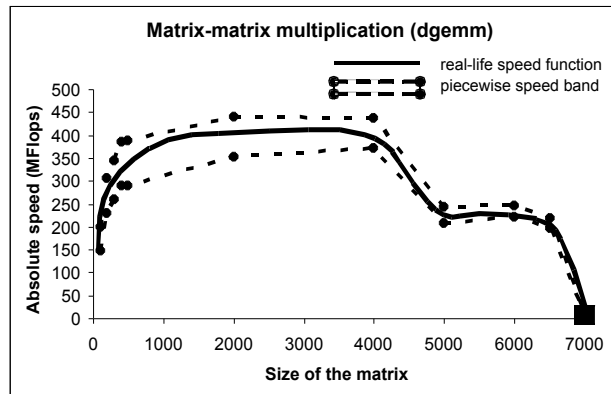
Figure 3.9: Experiments showing the constructed Band Approximations realised by the Geometric Bisection Building Procedure for a variety of routines on X1.



(a) Speed Band for Cholesky Factorisation on X2.



(b) Speed Band for Naive Matrix Multiplication on X2.



(c) Speed Band for optimised ATLAS Matrix Multiplication on X2.

Figure 3.10: Experiments showing the constructed Band Approximations realised by the Geometric Bisection Building Procedure for a variety of routines on X2.

points were taken for very small problem sizes in the second step of GBBP where it tests for a performance increase in the band. These experimental points have no significant impact on the overall construction time as their running time is relatively small.

3.6 Summary

The piece-wise linear functional approximation of the speed band of a processor may be practically built using the algorithm presented (GBBP). Its use results in a dramatic speed up in construction time against a naive method. This has been demonstrated for a variety of routines with differing band profiles, both optimised and naive matrix multiplication as well as Cholesky Factorisation. The algorithm presented uses more experimental points in regions of the band profile where significant change occurs and fewer experimental points where the profile may easily be approximated. Overall it approximates the profile of the band with a high degree of accuracy using fewer points than a naive method.

Chapter 4

Application to Construct Processor Performance Models

This chapter presents a tool, the Performance Model Manager (PMM), which addresses the complexity of the construction and management of a set of Functional and Band Performance Models on a computing server.

PMM [94] is a freely available open source piece of software distributed under the GPL. It implements the Geometric Bisection Building Procedure described in Chapter 3, optimising the construction of Functional and Band Performance Models. The manual for PMM is attached as Appendix A of this thesis.

It allows for the definition of a large number of dynamic routines, for which it will construct detailed performance models. Their construction may be managed by flexible policies implemented in PMM which aim to minimise the impact of the building procedure on the platform.

Finally, it provides visualisation of, and interfaces to access, constructed models so that they may be understood and used in other systems.

In following sections the implementation of the GBBP construction method is briefly described, how a routine is configured for construction by the manager is shown and the management of model construction and interfaces to access constructed models are explained.

4.1 Performance Model Manager

PMM is a tool that has been developed to address issues surrounding the construction, maintenance and use of Functional Performance Models in a variety of parallel computing environments. It consists of three main features. Firstly it implements the Geometric Bisection Building Procedure, described in Chapter 3, for multi-parameter FPMs, optimizing the construction of a routine's performance model. It permits a large number of routines to have their construction managed by implementing a flexible benchmarking scheduler, suitable for use where a queuing system does not exist. Finally it provides access to the models in a variety of ways, allowing feedback from actual executions and providing tools to use the models in scheduling decisions.

4.2 Efficient Construction

Unoptimised construction of the FPM for a large set of routines installed on a large number of servers is infeasible due to the time and resources that would be consumed. A novel algorithm that optimises the construction has been described in Chapter 3, titled Geometric Bisection Building Procedure (GBBP).

Single parameter GBBP optimisations are made possible by using the natural variation in performance (due to a server's external load fluctuations) and assumptions on the shape of a FPM (that it may initially be increasing, but is then decreasing and monotonic). Examples of the shape of models that fit these assumptions are shown in Figure 4.1a.

4.2.1 Band of Performance

The performance of a server in a non-dedicated environment is variable and a performance model for such a server must not be static. A FPM can be considered as a single possible level of performance that a server may return under certain load conditions. When those load conditions are variable, the FPM becomes a band of performance levels rather than a single function.

GBBP finds a piecewise approximation of this band (illustrated in Figure

4.1a). The approximation is constructed in such a way that its intersection with the real-life performance band forms a simply connected surface. That is, the approximation intersects the real-life band across the entire problem size range leaving no gaps. This ensures the accuracy of the model while allowing the formulation of optimizations that do not violate the constraint. A history of load fluctuations, which does not include load from any heavy computational processes that have been scheduled for execution on the processor, is recorded. This history can be used to predict the maximal and minimal expected load fluctuations that a problem may encounter on execution. Benchmarks made during the construction of the model are adjusted by the loads they are predicted to encounter. The result is a maximal and minimal speed for every problem size and these form the band model.

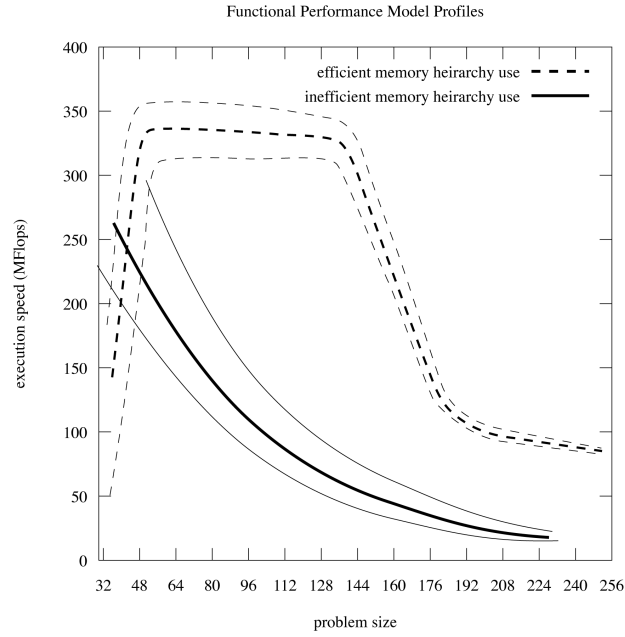
The functional model is then extracted from the band using the average speed between the maximum and minimum limits of the band. The band itself is not used in scheduling. As shown in Chapter 2, it provides limited benefit while adding a great deal of overhead. Its main purpose is in enabling the optimization of construction and providing a dynamically adjusted functional model which describes performance under the average load condition.

4.2.2 Model Shape

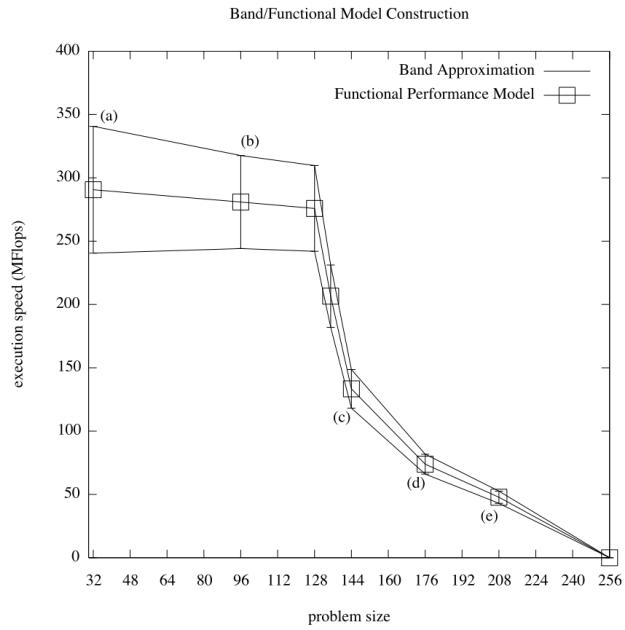
The optimization of construction is based on assumptions on the shape of the model. These are: that the performance may initially increase and that after any initial increase it will be decreasing and monotonic. The initial performance increase is discovered through a series of short benchmarks for small problem sizes. These are inexpensive.

Once non-increasing, the problem size range is recursively bisected. The performance at each bisection point is found through experimentation. At each point, an attempt is made to determine if further construction is required or if sufficient detail has been resolved in the model. Identifying where benchmarks are no longer needed minimises the construction time.

For instance, in Figure 4.1b, when the model at benchmark points (*a*) and (*b*) is examined, we find that the vertical component of the band at (*a*) contains



(a) Typical profile of Band Performance Models according to their memory access efficiency.



(b) Band Performance Model for `barmatter` routine with points used by GBBP and a naive construction algorithm shown.

Figure 4.1: Load observations and the load averages for various periods calculated from them.

(*b*) entirely. As a result of the assumptions on shape (that it is monotonic), construction between these points can cease without violating the simply connected property of the real/model intersection.

Further, in the case of benchmarks (*c*), (*d*) and (*e*) we can see that (*e*) was previously approximated by the segment joining (*c*) to (*d*) and, before that, by the segment joining (*c*) to the endpoint. Again, no further benchmarks are required in the intervals between these points, as it has been shown that the model adequately approximates the real band in these regions. A full description of the GBBP algorithm can be found in Chapter 3

GBBP as described only applies to performance models that are a function of a single parameter. In some cases it may not be possible to combine the parameters of a problem into a single measure of problem size. For these cases a multi-parameter model must be constructed. A pair of heuristic methods for multi-parameter GBBP, each based extension of the single parameter algorithm are implemented in PMM. The reliability of the approximation is not as rigorous as for single parameter models.

Multi-parameter construction is exponentially more expensive than single parameter construction, so it is vital that any parameters which do not contribute to the problem size or speed of execution are not included in the construction of the multi-parameter model.

Single parameter GBBP requires that a minimum and maximum problem size are defined. This is also the case for multi-parameter methods, each parameter must have a defined maximum and minimum. The two methods implemented are as follows:

4.2.3 Boundary multi-parameter GBBP

For each routine parameter, GBBP is applied in constructing a Bands where that parameter is variable, and all other parameters are fixed at their minimum values.

The resultant band models form the boundaries of the full multi-parameter model. New construction intervals are projected parallel to each parameter-axis and through the points along the boundary band models forming a grid. GBBP is applied to each of these intervals to complete the approximation of the multi-

parameter model.

4.2.4 Diagonal multi-parameter GBBP

In this heuristic, a virtual construction interval created between a point where all parameters have their minimum values, a so-called origin of the parameter space, and a point where all parameters have their maximum values. The Geometric Bisection Building Procedure is applied along this interval to create a performance band.

On completion of this diagonal band model, new construction intervals are projected parallel to each parameter axis and through the experimental points that make up the diagonal band, forming a grid. GBBP is applied to each of the new construction intervals to complete the approximation of the multi-parameter model.

The advantage in this method is that the overall problem size grows consistently in the initial diagonal construction interval. This results in a reasonable initial approximation of the shape of the model. The initial boundary models in the Boundary method may not have the expected shape as the problem size does not always grow significantly if just one parameter is increased.

4.3 Routine Configuration

PMM provides the developer of a routine with a framework for using GBBP to construct a routine's FPM. An interface to a call a routine with a particular set of parameters must exist so that PMM may execute benchmarks at points determined by GBBP.

To realise this interface the routine developer must provide a benchmarking binary that executes a call to the routine on behalf of PMM. This binary must follow a set of rules regarding the input that it accepts and the output it returns to PMM. In certain specific environment it could be conceivable to execute routines dynamically without this requirement, provided their function call has been described in some machine parse-able way. However, there would be no facility to pass intelligible data for the routine to process. This would require a user define

actual code to initialise data as well as a detailed description of the function.

Such efforts would amount to no less effort than what is proposed: that the user implements a small benchmarking binary to execute a routine. Ultimately the proposed method is a far simpler and more flexible solution to the interfacing between the routine and PMM.

In order to allow the PMM to execute benchmarks of a routine at points as requested by GBBP, the benchmark binary of the routine that the developer provides must:

- accept an ordered list of command line parameters that define the size of the input parameters to the routine
- dynamically allocate input and output data structures according to the input arguments
- initialise input parameters with data that is intelligible to the routine call, and permits normal execution
- place calls to the PMM timer functions directly before and after execution of the routine
- terminate and return normally on successful execution

An example of a routine benchmark is shown in Listing 4.1. The calls to PMM timing code are highlighted on lines 27 and 32. Also shown are constructors, destructors and the function that formats and prints the measured benchmarking information (line 34), which is parsed by PMM. The routine that is benchmarked by this example is an N-Body simulation of dark matter. It comes from an application that will be described in Chapter 5.

The `darkmatter` routine acts on two large 3-dimensional matrices. Both these matrices must be cubic and as a result we only need to build the FPM in terms of two parameters, the size of a single side of each matrix, N_x and N_p . This is an important optimization that the routine developer can enable us to use, as greater numbers of parameters results in far longer construction time. There is also a constraint that N_p is less than N_x . Parameter constraints may be configured in the description provided to PMM, or by the benchmarking binary itself, as is

the case here. When the parameters are not valid it returns a defined code to PMM on line 15 of Listing 4.1.

In the configuration of PMM the routine developer specifies the parameters to pass to the benchmarking binary, the order that they appear in the function call, the range of each parameter (over which the model is to be built) and a path to the binary itself. An example configuration in XML is shown in Listing 4.2.

4.3.1 Flexible Construction

PMM can construct models in a number of modes. Invoked from the command line in an interactive mode, PMM can construct all models it has been configured with at that instant. This provides for accurate model construction before a server is enabled as part of some computing cluster. For a large number of routines, this may be is a lengthy process that could occupy a machine for an unacceptable amount of time. For situations where a server cannot be removed from use for the duration of the model construction process, PMM can be started as a daemon process. In this mode the construction of FPMs could be less intrusive. Time constraints, system conditions and routine priorities can be applied to manage the building process with a maximum level of flexibility provided to the system administrator.

Time constraints limit the periods when models are permitted to be constructed. Three constraints have been implemented:

now construct a model as soon as possible with no time limit on benchmark execution

until allow construction of a model up until a certain time, at which point, end construction or allow another time constraint to take over

periodic construct a model in specific time intervals, which can be defined by the minute of an hour, the hour of a day, day of week, etc.

Along with each time constraints are halting conditions. These are monitored conditions that can prevent benchmarking. We make a number of conditions available to monitor as well as test for the existence of a halt-file. The halt-file

Listing 4.1: Example Benchmark Code

```
1 #include <pmm_util.h>
2 #include <"hydropad_bench.h">
3
4 int main(int argc, char **argv) {
5
6     /* declare variables */
7     global_data *gb;
8     int nx, np;
9     struct pmm_timer *t;
10    long complexity;
11
12    parse_args(argc, argv, nx, np);
13
14    if (nx < np)
15        return PMM_INVALID_PARAM;
16
17    /* allocate and initialise data */
18    allocate_gb(gb);
19    gb->nx = gb->ny = gb->nz = nx;
20    gb->np=np;
21    initialize_gb(gb);
22
23    complexity = (long) pow(nx,3.0);
24
25    t = pmm_timer_init("dark", complexity); /* init timer
        */
26
27    pmm_timer_start(t); /* start timer */
28
29    /* execute routine */
30    darkmatter(gb->nx, gb->ny, gb->nz, gb->np, .....);
31
32    pmm_timer_stop(t); /* stop timer */
33
34    pmm_timer_result(t); /* get timing result */
35    pmm_timer_destroy(t); /* destroy timer */
36
37    free(gb);
38
39    return EXIT_SUCCESS;
40 }
```

Listing 4.2: Example configuration of darkmatter routine in PMM

```
1 <routine>
2   <name>darkmatter</name>
3   <exe_path>/usr/lib/pmm/darkmatter</exe_path>
4   <model_path>/var/pmm/darkmatter_model</model_path>
5   <parameters>
6     <param>
7       <name>nx</name>
8       <order>0</order>
9       <min>32</min>
10      <max>256</max>
11    </param>
12    <param>
13      <name>np</name>
14      <order>0</order>
15      <min>32</min>
16      <max>256</max>
17    </param>
18  </parameters>
19
20  <priority>30</priority>
21
22  <benchmarking_policies>
23    <policy>
24      <time_constraint type=now/>
25      <condition type="user_login"/>
26      <condition type="halt_file">
27        <halt_path>/tmp/.pmm_halt</halt_path>
28      </condition>
29    </policy>
30  </benchmarking_policies>
31
32  <construction>
33    <method>gbbp</method>
34  </construction>
35 </routine>
```

allows an administrator to add any halting condition they wish via an external program that creates and removes the file. The conditions implemented are:

user login Halt construction if a user is logged into the machine.

load threshold Halt construction when load is above a threshold, this condition is not monitored while a benchmark is being executed.

process detection Halt the construction if a particular process is detected.

user process detection Halt the construction if any process that does not belong to an exclusion list of users is detected.

halt file As described previously, a specific file is tested for existence and construction is halted on that basis.

Finally we allow each routine to have a construction priority. Problems with a higher priority are constructed to completion before the construction of other routines is begun. Problems with the same priority are scheduled based on their level of completion, always choosing to benchmark a routine that is less complete first.

All constraints on construction can be applied system wide, to all routines configured in PMM, but specific routines can have specific constraints applied to them, which override the system wide configuration. For example, the general timing policy may be that benchmarks are only executed on weekends, but some high priority routine may have a less limiting constraint allowing it's benchmarks to be executed during weekdays provided there are no processes detected that relating to a cluster resource user.

When halting conditions are encountered, no benchmarks will start executing until the conditions have cleared. However, if a benchmark is already executing a decision must be made as to whether to allow it to complete or signal it to halt. The action to take is a configurable option. If the halting strategy is to interrupt executing benchmarks and the time constraints / halting conditions are very limiting, lengthy benchmarks may never be run to completion. Consequently some models may never be completely constructed. To mitigate this issue the scheduler takes a number of actions:

1. Benchmarks of a large size are added to the rear of a routines benchmark queue.
2. Interrupted benchmarks are moved to the rear of a routines benchmark queue.
3. Repeatedly interrupted routines have their priority reduced.

Though none of these actions prevent this issue entirely, they do delay the point at which it would interfere with FPM construction. Ultimately it is for the administrator to decide how to un-constrain the construction so it may complete.

The design of benchmarking scheduler is trivial. As it has a periodic duty to check the halting conditions, this fixed loop can also be used to schedule new benchmarks. The algorithm is defined in Algorithm 3.

Algorithm 3 Scheduling loop of the Performance Model Manager

```
while 1 do
  update system conditions
  if a benchmark  $b$  is currently executing then
    if  $b$ 's execution-policy is no longer satisfied by the system conditions
      then
        halt  $b$ , if halt-able
      end if
    else
      if the global execution policy is satisfied then
        execute a benchmark on a routine with the highest priority
      else
        for all routines with a defined execution policy do
          if the routine's execution policy is satisfied then
            add the routine to an executable list
          end if
        end for
        execute benchmark on the routine with the highest priority on the executable routine list
      end if
    end if
    sleep
  end while
```

4.4 Enabling Access and Use of FPM

PMM provides external programs with access to models in two manners. First, direct access to the FPMs is available via the file system. The models are stored in a structured format using XML. The PMM API provides methods to locate and parse the FPMs stored on a system into data structures. The API also provides accessor methods to look up an execution time approximated by the model, given a particular set of routine parameters. New points in the FPM can be added to the model when using files, but only if the models are not in the process of being constructed by PMM.

When running as a daemon, the manager can service requests for models via socket instead. It accepts the submission of benchmark timing via socket also, which may come from actual executions of a routine that have had timing code inserted. If construction is ongoing when an actual execution time is submitted, the submission can be processed by the GBBP algorithm and can aid in further minimizing construction time. A set of methods is provided for conveniently opening a socket to PMM and sending or receiving data, in the form of individual benchmarks or whole models.

Further, a plotting tool is provided which interfaces with the widely available Gnuplot [95] application. The tool allows for visualisation of both complete models and models in construction, as well as live visualisation of the construction process. Single and two parameter models can be viewed natively, and higher dimensional models can be viewed by defining slices, i.e. fixing parameters at certain values so that no more than two free parameters remain.

4.5 Summary

This chapter presented the Performance Model Manager tool, which has been designed to enable the construction and use of Band and Functional Performance Models. Its goals are to build the FPM in the most efficient manner possible and to minimise the disruption to a running server. To these ends, it implements the Geometric Bisection Building Procedure and it allows the user to apply a flexible set of constraints on the benchmarking procedure.

The configuration of PMM and how it benchmarks a routine in order to construct the routine's FPM has been described briefly, a complete manual can be found in the Appendix [A](#). FPMs enable more efficient parallel computing in heterogeneous environments, it is important that a tool such as PMM exists so that their use can be conveniently realised.

Chapter 5

Functional Performance Models in a GridRPC Environment

Grid Computing often utilises highly heterogeneous networks of computers. Efficient high performance computing on Grids can only be achieved when accurate models of the performance of compute nodes are available to the scheduling middle-ware. The performance of a processor executing a routine is determined not only by the physical characteristics of the processor, but also by the nature of the routine's core algorithm and the size of the problem it is requested to compute (determined by input parameters). When scheduling the execution of a remote routine on a Grid, it is important to make an accurate estimation of the routine's execution time on the available heterogeneous processors.

In GridRPC [96] systems, many scheduling decisions are based on an estimation of execution time of the routine. The Minimum Completion Time heuristic [48] as implemented in NetSolve [97] or the Historical Trace Manager [98] heuristics implemented in GridSolve [99] both rely on the accuracy of this estimation. Presently, this is provided by the combination of a simple LINPACK [33] style benchmark, which measures operations per second of the processor, and a measure of the complexity of a given routine. This kind of estimation assumes that the core algorithm of the routine is similar to the benchmark code used, which is not necessarily the case. A routine may be more or less suited to a particular processor as a result of the underlying architecture of that pro-

cessor. This difference may not be represented in the benchmark. Further, the single benchmark does not account for variance in processor speed as problem size increases. It describes the speed as constant. Should the speed of processors decreases with problem size, it assumes they will do so at the same rate and that their relative speeds remain constant. This is never quite true, especially where paging occurs on processors at different sizes of problem. In any case, advanced task schedulers require accurate measures of a task's execution time on all processors rather than an accurate ranking of the processors.

The Functional Performance Model (FPM) is an excellent candidate for making a more accurate estimation of a routine's execution time. It is a routine specific, realistic, experimentally obtained model of the actual execution speed of a routine expressed as a piece-wise linear function of the problem size. The problem size is given by its input parameters. These properties of the FPM address both issues with the current estimations used in GridSolve. However, the tasks of construction, management and use of a set of functional models are not trivial. The previous chapter presented a tool which addresses these issues: the Performance Model Manager (PMM). In this chapter the integration of Functional Performance Models, provided by PMM, with an extended GridRPC system: SmartGridSolve [82], is described.

5.1 SmartGridSolve and PMM

The steps involved in a GridRPC call using GridSolve are illustrated in 5.1. There are three actors: the client, agent and server. Servers execute routines on behalf of the client. The agent maintains a list of registered GridRPC servers that it may offer to clients. Each server communicates to the agent the routines it can solve and periodically sends an up-to-date performance index for the server. When the client makes a GridRPC call it first communicates with the agent, sending a description of the routine it wishes to have solved. In return, the client receives an ordered list of servers ready to service the request. It then selects a server and sends a request to solve the routine directly to that server. The list of servers sent to a client is ordered using GridSolve's scheduler on the agent. Amongst other things, the scheduler uses a servers score to decide how to order

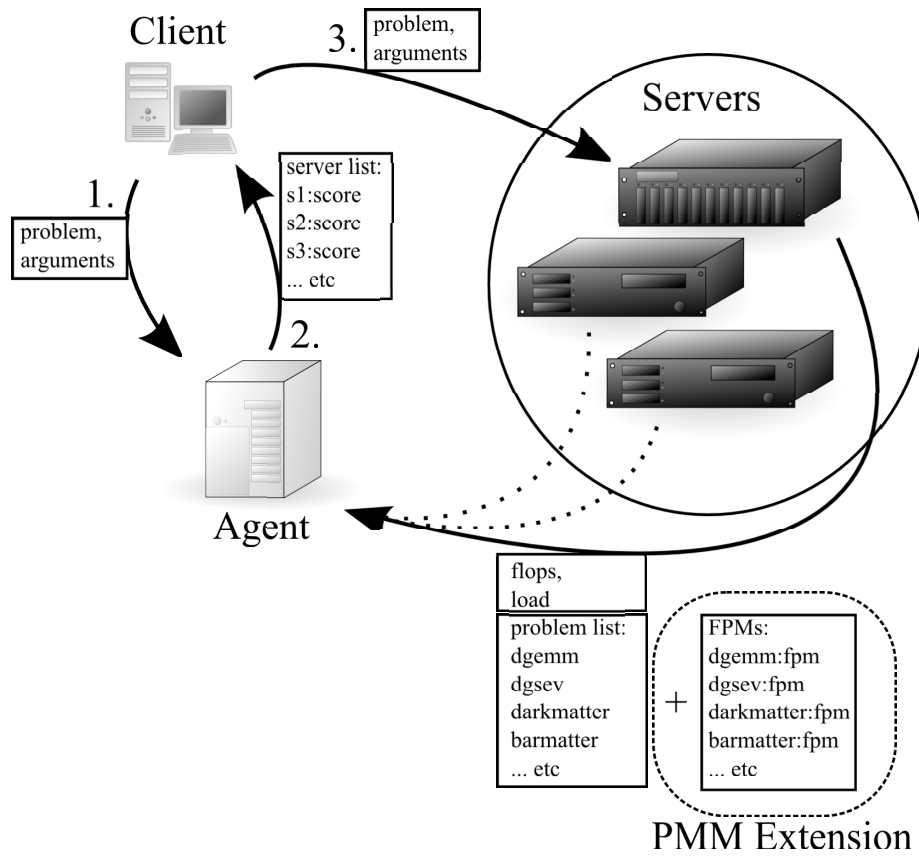


Figure 5.1: Illustration of Scheduling Transactions in GridSolve.

the server list. For a given routine request, the agent calculates a score for each server that has the ability to compute the routine. The score is a representation of the time that a server would require to execute the routine. This is calculated using two components, a measure of the routine's complexity and a measure of the server's speed.

The routine's complexity is set by the routine developer during its configuration in GridSolve. It is a function of the scalar arguments of the routine, which are known to the agent when it is calculating a server's score. The speed of a server is measured in floating point operations per second (FLOPS) using a LINPACK type benchmark. The routine complexity divided by the server's FLOPS gives the server's score.

As previously mentioned, a single benchmark can be a poor representation of a processors speed when the routine being executed is not similar to the benchmarked routine or when the processor uses a different memory hierarchy to the benchmark.

The Functional Performance Model overcomes both of these issues. It models the performance of each specific routine for a range of input sizes, not at only a single point and not using a characteristic application, but the actual routine itself. It is an experimental model, which can render accurate approximations of execution time.

Integration of the FPM in GridSolve requires no fundamental modifications to the GridSolve system design. Server scores are a rough estimation of execution time of a routine with a given set of arguments; the FPM provides exactly this, so from the scheduler perspective, no changes are made. No changes are required on the client side either.

The "Smart" extension to GridSolve (SmartGridSolve [82]) uses the same mechanism to retrieve estimations of routine execution time. It has the ability to schedule groups of parallel routines in a single mapping. When scheduling a group of routines the scores of the routines on all available servers are input to a scheduling algorithm. Inaccuracy in the estimation of execution times severely limits the ability of a scheduler in its search for an optimal mapping.

Functionality is added on the server and agent via compile flags set during the configuration of the GridSolve. Where previously, the server would commu-

nicate to the agent a list of installed routines at start up, it now must also provide the agent with FPMs for those routines. The server retrieves the FPMs either directly from the PMM daemon via socket or from the file system. The server also submits timing from actual executions to PMM.

Modification to the agent is only in networking code to receive models from the server and in calculating a server's score using the FPM. Common socket code can be used in the server to agent and server to PMM communications. Apart from extending the networking protocol between the agent and server, the majority of the required code permitting the use of FPM in GridSolve and Smart-GridSolve exists in PMM's shared library. No modification to the scheduler is necessary.

5.2 Hydropad and PMM

Hydropad [100] is a simulation of the evolution of clusters of galaxies in a universe that is comprised of baryonic matter and dark matter. The core loop of this simulation models the internal interactions of baryonic matter and dark matter, separately and in parallel, while their mutual interaction is modeled in a sequential gravitational calculation. The structure of the application is illustrated in 5.2. A GridRPC version ([101, 102]) of this application has been implemented to demonstrate the performance of the Smart extension to GridSolve. Each task in the graph is implemented as a remote procedure call. As a result of data dependencies between time-steps it is not possible to unroll the loop, which limits the level of task parallelism. Further, the volumes of data that must be communicated by the tasks are high. These properties make it particularly challenging for a GridRPC middle-ware to achieve high performance when running the application. It is for this reason that Hydropad is a good application to examine the performance of GridSolve and the benefit of using FPMs in GridSolve. The data manipulated by the simulation are three-dimensional cubic matrices that describe the particles in the system (in terms of position, pressure, density, etc.). The number of particles in the system is defined by N_p . The accuracy of the overall simulation is determined by the number of cells which the simulation space is divided into. These cells are in a cubic grid structure, the size of which is given

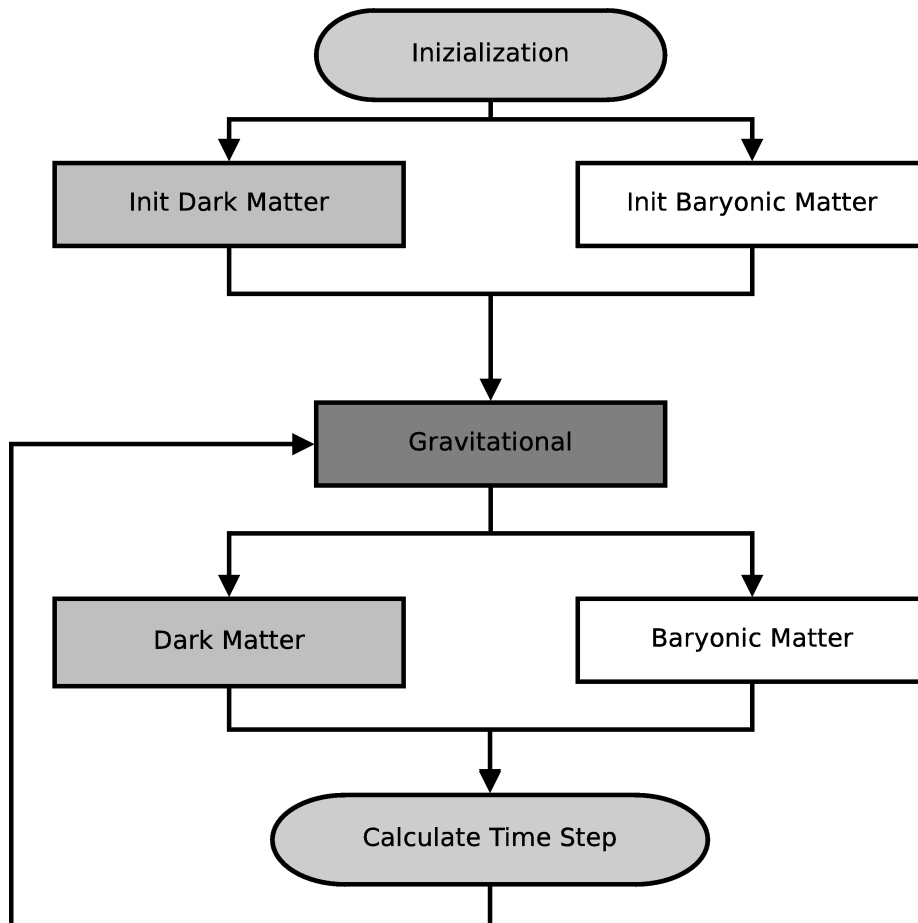


Figure 5.2: Task Graph of Hydropad Application.

by N_x .

The major computational routines in Hydropad are those contained in the main loop. The dark matter routine, `darkmatter`, is a Particle-Mesh N-Body algorithm with a complexity of $O(N_p)$. The Baryonic matter, `barmatter`, routine is a Piecewise Parabolic Method with a complexity of $O(N_x)$.

In the context of Functional Performance Models, both of these routines are interesting ones. The volumes of data they operate on are different. `darkmatter` takes as input parameters both the particles in the system, specified by N_p and the cells of the grid, specified by N_x . `barmatter` only operates on the cells of the grid structure. Despite this, `barmatter` is computationally more intensive. When executing these tasks on a two of heterogeneous machines it is important to note the volumes of data and the memory available to each processor. A simple performance model will map the computationally large `barmatter` routine to the fastest server. However, if the slower server does not have enough memory to compute the `darkmatter` routine without paging, it may be that overall, the tasks would be executed more quickly if `barmatter` is mapped to the slower server. This is counterintuitive when the only performance information available is a single benchmark.

FPMs for Hydropad routines have been built using PMM. As can be seen in the task graph, there are a number of routines that must be executed prior to execute `barmatter` or `darkmatter` routines. These are associated with the initialization of the data structures and the calculation of gravitational fields. In the benchmarking binary for a particular target routine, timing functions can be added around any of the routines that the target is dependent on. As such, only two benchmarking binaries were required in building the models for Hydropad, as adequate data for the FPMs of initialization and gravitational routines could be retrieved from the benchmarks of `darkmatter` and `barmatter`.

5.3 Models and Experiments

This section presents the FPMs constructed using PMM for the Hydropad application and experimental results in the speed up achieved through using the FPMs in GridSolve, with the Smart extension. As task parallelism is limited only two

Table 5.1: SmartGridSolve Server Configuration.

Name	Type	MFLOPs	Memory
<i>hcl10</i>	1.8GHz Opteron	693.85	1024 MiB
<i>hcl05</i>	3.6GHz Xeon	481.68	256 MiB

servers were used in experiments, their configuration is listed in Table 5.1. All timed results were remote computations, totally independent of the client.

Experiments were carried out to illustrate the benefit of using FPMs when scheduling a group of tasks. For simplicity experiments are focused on a single iteration of the main loop in Hydropad, the parallel routines: `darkmatter` and `barmatter`. Figure 5.3 shows FPMs for the `darkmatter` routine, which are in terms of two parameters N_p and N_x . The models for both servers in the experimental setup are displayed. The change in their relative performance as parameters increase in size is illustrated at the base of the graph. It is clear that paging begins on *hcl05* before *hcl10* and that while the relative performance is fairly constant for smaller problem sizes, it changes dramatically when paging starts. At the maximum problem parameters allocate-able by *hcl05*, it is computing at a rate that is twelve times slower than *hcl10*, when before it was just slightly slower. This is a property of *hcl05*s performance that is not represented by a single benchmark. Figure 5.4 reveals greater detail in the region of paging for the `darkmatter` task. Figure 5.5 shows the functional performance models for the `barmatter` task. Again, the differing amounts of available memory on the servers results in performance degradation at different values of N_x . Benchmarking points for a naive construction method are also displayed to illustrate the reduction in the number of benchmarks that are required to build the FPM using GBBP versus a naive method.

The time spent executing GBBP and naive benchmarks is shown in Table 5.2. The speed up achieved by GBBP makes the construction of FPMs a more practical task. In one case GBBP did not achieve a large speed up, the `barmatter` task on *hcl10*. This is because an artificial limit was placed on the range of the input value N_x . Had the model been constructed across all allocate-able problem sizes, the Geometric Bisection Building Procedure would have shown a consistent speedup.

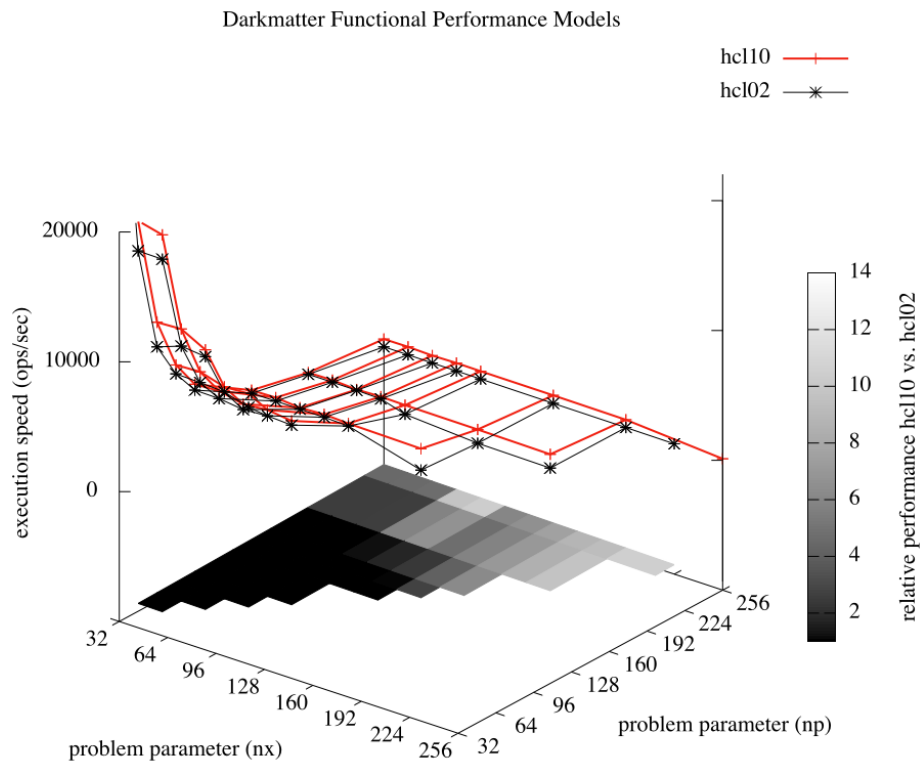


Figure 5.3: Graph of `darkmatter` Functional Performance Model with relative performance highlighted.

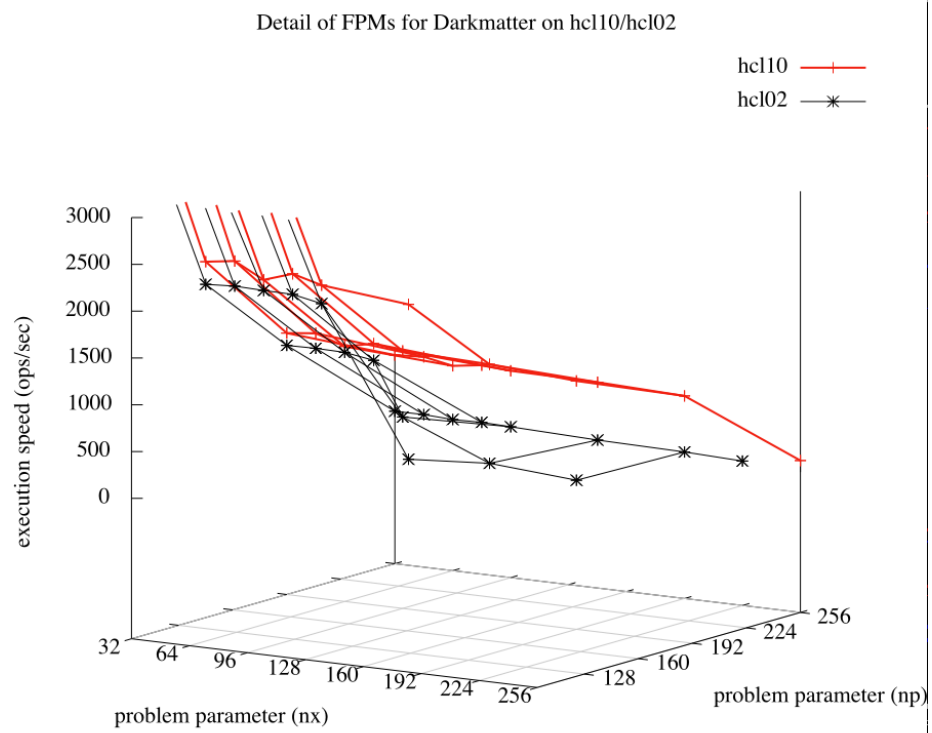


Figure 5.4: Detail of problem parameters where paging contributes to sudden performance decrease in darkmatter problem.

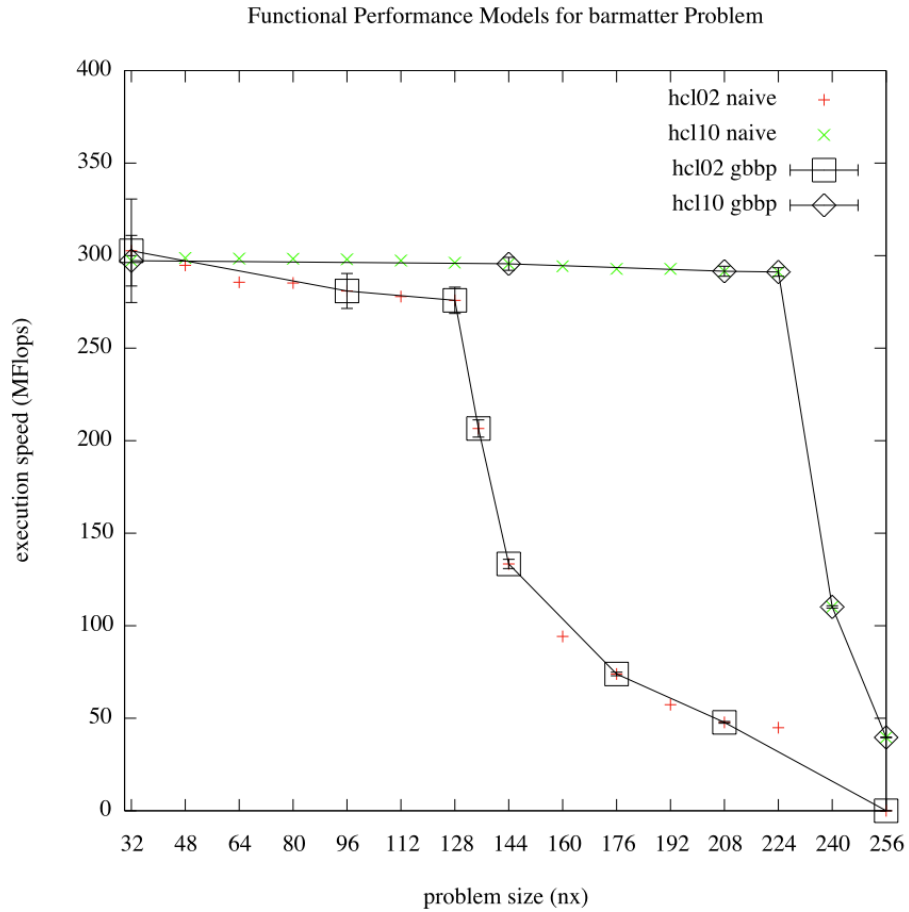


Figure 5.5: Functional Performance Models for the `barmatter` routine, with points for GBBP and a naive construction method shown.

Table 5.2: Time Spent Construction Functional Performance Models.

Task / Machine	Naive Points	Naive Time	GBBP Points	GBBP Time	Speed-up Factor
darkmatter / hcl05	76	6854s	36	2292s	2.99
darkmatter / hcl10	80	1704s	36	598s	2.85
barmatter / hcl05	14	8792s	7	3687s	2.38
barmatter / hcl10	15	8262s	6	7444s	1.11

Finally, Table 5.3 shows the results of a set of experiments on the scheduling accuracy of SmartGridSolve. A single iteration of the main loop was timed for a set of input parameters. First the scheduler was provided with the standard LINPACK type single benchmark that is made available by GridSolve. As `barmatter` is the most computationally intensive problem, the scheduler assigned it to the faster processor, `hcl10`, in all experiments. The `darkmatter` problem was executed in parallel on `hcl05`. However, when the amount of data `darkmatter` operates on exceeds the available physical memory on `hcl05`, it begins to slow. At this point it would be more efficient to assign the computationally intensive `barmatter` to the slower server, as it would be able to solve this problem without paging. The scheduler is not able to make this decision when the performance model of the processor does not represent the change in speed at different levels of the memory hierarchy. When the scheduler uses FPMs in its decision-making the results are much better. When no paging occurs, it schedules in exactly the same way as before, but as `hcl05` begins to page it is assigned the routine with the smaller memory footprint. This permits a more optimal scheduling with much greater overall performance. It also allows larger problem sizes to be executed. Previously, after the number of particles, N_p exceeded 256 the `darkmatter` routine failed to execute on `hcl05` as it could not allocate enough virtual memory.

Table 5.3: Scheduling Improvements with FPMs in SmartGridSolve.

N_p	N_x	Iteration Time: single benchmark	Iteration Time: GBBP	FPM Speed-up Factor
96	96	26.33s	26.17s	1.01
128	96	25.10s	24.91s	1.01
160	96	25.45s	24.59s	1.03
192	96	41.48s	29.63s	1.40
216	96	123.78s	27.98s	4.42
256	96	n/a	39.02s	n/a
288	96	n/a	51.98s	n/a
320	96	n/a	362.57s	n/a

5.4 Summary

This chapter has presented the integration of Functional Performance Models provided by the Performance Model Manager, with a GridRPC middle-ware, SmartGridSolve. This integration has been described and shown to have minimal impact on the general architecture of SmartGridSolve.

Hydropad, a scientific application which has been reworked as a general benchmark of GridRPC systems has been described in the context of Functional Performance Models. PMM has been used to build the models of key tasks in the Hydropad application, these models have then been used in SmartGridSolve to map task to servers. The efficiently constructed models have been presented and the improvement in execution of Hydropad, using a variety of input data sizes is shown. By providing accurate estimations of execution time of various task on the nodes of the Grid, the FPM allows SmartGridSolve to assign these tasks for optimal overall execution speed. It also allows execution of larger problem sizes than the standard GridSolve performance models do by describing the performance drop where paging occurs.

Chapter 6

Conclusion

This thesis has introduced the Band Performance Model to the domain of problem partitioning and task scheduling in heterogeneous parallel and distributed computing environments. This model extends the functional performance model by adding a prediction of processor performance variance to it. It is a highly detailed representation of a processors speed at a specific task.

Though it was conceived as an extension of the Functional Performance Model, the FPM may also be considered a product of the Band Model, where the performance variance represented by the band is averaged to a single valued function. The Band Model represents the uncertainty in the performance of the processor, and the adjusted functional model is just an average level of predicted performance.

In Chapter 2, the formulation of a simple and detailed Band Performance Model has been described and a number of methods of evaluating the “goodness” of a problem distribution using these models have been proposed. Each of these methods results in a metric which may be maximised to find a best distribution of some problem. These metrics have been maximised by exhaustive search and a genetic algorithm. Experiments were conducted which simulated the execution time of the best problem distributions rendered by a variety of model types. The number of processors to partition across, and load fluctuation on those processors was varied to reveal the overall performance of the different model types. The conclusion of those experiments is that though the simple and

detailed Band Performance Models result in good load balance, the improvement over an adjusted functional model is marginal at best. As a result, algorithms for using the Band Models to partition problems has not been developed beyond the implementation of the genetic based solver. This result however, allows us to focus on the derivation of a Functional Performance Model from the Band, which is investigated further in this thesis.

Chapter 3 presents an algorithm (GBBP) that optimises the construction of the piece-wise linear functional approximation of the Band Performance Model. The importance of this is that it allows the practical construction of both Band and adjusted Functional Performance Models. Its use results in a dramatic speed up in construction time against a naive method. This has been demonstrated for a variety of routines with differing band profiles, both optimised and naive matrix multiplication as well as Cholesky Factorisation. The algorithm presented uses more experimental points in critical regions of the performance profile, but overall approximates the profile with the same, or greater, degree of accuracy using less experimental points.

In order to demonstrate and make available Functional Performance Models, a tool, the Performance Model Manager tool has been implemented. This tool is designed to enable the construction and use of Band and Functional Performance Models. Its goals are to build the FPM in the most efficient manner possible and to minimise the disruption to a running server. To these ends, it implements the Geometric Bisection Building Procedure and it allows the user to utilise a flexible set of constraints on the benchmarking procedure. It provides access to the models for use in task scheduling and problem partitioning with other softwares, as well as visualisation of constructed models and the construction process.

Chapter 4 has described the configuration of PMM and how it benchmarks a routine in order to construct the routine's FPM, a complete manual can be found in the Appendix A. FPMs can enable more efficient parallel computing in heterogeneous environments, it is important that a tool such as PMM exists so that their use may be conveniently realised.

Finally, Chapter 5 presented the integration of Functional Performance Models provided by the Performance Model Manager, with a GridRPC middle-ware

(SmartGridSolve) and a non-synthetic application that uses this middle-ware (Hydropad). This integration has been described and shown to have minimal impact on the general architecture of SmartGridSolve. This shows that the FPM is suitable for use in a task scheduling environment.

Hydropad, a scientific application which has been reworked as a challenging benchmark for GridRPC systems, has been described in the context of Functional Performance Models. PMM has been used to build the models of key tasks in the Hydropad application, these models have then been used in SmartGridSolve to map task to servers. The efficiently constructed models have been presented and the improvement in execution of Hydropad, using a variety of input data sizes is shown. The FPM allows SmartGridSolve to assign tasks for optimal execution speed and allows execution of larger problem sizes than the standard GridSolve performance models do.

We conclude that Functional and Band Performance Models have wide and practical application in heterogeneous computing environments. Their use should be pursued in middle-wares and programming frameworks that target heterogeneous platforms. They can be a first port-of-call for when attempting to tune an algorithm to a fixed platform, or integrated in heterogeneous parallel libraries to provide efficient balancing of workloads in users applications, across dynamic resources.

As computing architectures evolve, from Hybrid CPU-GPU computing to Heterogeneous Multi-core, the requirement for balancing workloads between diverse heterogeneous processors will become stronger and more challenging. The Functional Performance Model is a general solution that can reliably solve this problem.

Appendix A

PMM User Manual

Performance Model Manager
User Manual

A.1 Introduction

Performance Model Manager (PMM) is an open source GNU Public Licenced tool for experimentation in the use of Functional Performance Models (FPMs). An FPM is a detailed description of the speed of a computational routine in terms of the routine's input parameters. PMM focuses on addressing issues surrounding the construction, maintenance and use of FPMs. To this end, it has three main features:

- It implements the Geometric Bisection Building Procedure for multi-parameter FPMs, optimizing the construction of a routine's performance model.
- It permits the construction of models for a large number of routines by implementing a flexible benchmarking scheduler.
- It provides access to the models in a variety of ways, so that they may be visualised, used to make scheduling decisions or partition problems.

Construction of the FPM of some general computational procedure is supported by requiring that the user provides a benchmarking executable which behaves according to a simple protocol.

An example implementation of such a “benchmark binary” is given in Section [A.4.2](#) along with details of its configuration within PMM. Further examples are also included in the source distribution.

Models can be constructed on demand or in the background by the `pmmcd` daemon. Construction of multiple models can be scheduled according to a variety of policies. They will be constructed in turn according to their priority and scheduling criteria.

Access to models is available via an API which will be documented in a future release of PMM, at present only viewing of models is described, via a plotting program: `pmm_view`.

This manual continues to Section [A.2](#) where compilation and installation is described. Then the PMM configuration file is described in Section [A.3](#). Section [A.4](#) provides notes on how to write a benchmark binary and configure it as a “routine” to be modelled within PMM.

A.2 Installation

A.2.1 Requirements

PMM is developed for the Linux platform but may also compile on other POSIX operating systems. The following softwares are required to install PMM:

- GNU Make
- GCC compiler suite (tested with 4.x series only)
- libxml2 2.6.0 or greater
- Gnuplot

The following are optional but enable certain features:

- Octave (2.9.14 or greater) is required for multi-parameter model construction
- muParser is required for definition of parameter constraints
- PAPI (4.0.0 tested) is required for higher resolution timing and automatic complexity calculation
- GNU Scientific Library is required for analysis and comparison of models, as well as for certain example routines
- GotoBLAS2 and/or ATLAS are required for further example routines
- LAPACK is required for further example routines

A.2.2 Compiling & Installing

Installation of PMM uses a hierarchy of directories under a certain prefix, by default `/usr`. If this is not desirable the build should be configured with the `--prefix=<dir>` option. A typical installation follows:

```
$ tar -xzf pmm-0.0.1.tar.gz
$ cd pmm-0.0.1
$ ./configure --prefix=$PWD/install
$ make && make install
```

Configuration options to note:

- `--enable-debug` enable debugging messages and flags
- `--disable-octave` disable use of octave and multi-parameter model support
- `--without-muparser` disable use of muParser libraries for definition of parameter constraint formula
- `--without-gsl` disable compilation and installation of components that depend on GSL (routine benchmarks and model analysis tools)
- `--with-gotoblas2 [=path]` enable compilation and installation of demonstration GotoBLAS2 routines with an optional specification of the GotoBLAS2 installation path
- `--with-atlas [=path]` enable compilation and installation of demonstration ATLAS routines with an optional specification of the ATLAS installation path
- `--with-lapack [=path]` enable compilation and installation of demonstration LAPACK routines with an optional specification of the LAPACK installation path
- `--with-papi [=path]` enable use of PAPI with optional specification of PAPI installation path

Further options can be viewed by running `./configure --help`.

After installation, the PMM daemon is started by executing the `pmmd` binary and the PMM viewer program is run via the `pmm_view` binary.

A.3 Configuration

PMM is distributed with a default configuration which will be installed under:

```
<prefix>/etc/pmmd.conf[.sample]
```

This can serve as a template for a user's own configuration and contains sane values for all options. If example routine benchmarks are built, the sample configuration will also describe those routines.

The configuration file has an hierarchical XML structure. Configuration is described between `<config>` root element tags. Under this, the load monitor facility is described by a `<load_monitor>` element, and each routine for which a model is to be built, is described by a `<routine>` element.

In the following sections, each element in the configuration file is described. If an element has a default value, it need not be explicitly set in the configuration file, on the other hand, some options must be set. This information, along with the type of the expected element value (string, integer, etc.) and what exactly the element describes is detailed below.

A.3.1 General Configuration

The following elements (which can be seen in context in Listing [A.1](#)) define some general application configurable options and come directly under the `<config>` tags:

- `<main_sleep_period>` (*integer, default:1*) The benchmark scheduler checks the system state ever n seconds and this period may be configured here. The default value is suitable and this variable is made configurable mostly for developmental purposes.
- `<model_write_time_threshold>` (*integer, default:60*) When benchmarking problems of small size, which execute quickly, the manager may become overloaded by writing the model to disk after each execution. This option allows us to configure how often the model will be saved to disk, i.e. after a total of n seconds has been spent benchmarking a particular model

it will be written to disk. The default value is suitable and this variable is made configurable mostly for developmental purposes.

- `<model_write_execs_threshold>` (*integer, default:10*) This option serves the same purpose as the previous one, except that it specifies the number of benchmark executions that must occur before it is written to disk. It may take hundreds of small benchmarks exceed the time threshold (above), so this second threshold allows us to write based on execution frequency as well. The default value is suitable and this variable is made configurable mostly for developmental purposes.

Listing A.1: Basic Configuration

```
1 <?xml version="1.0"?>
2 <config>
3   <main_sleep_period>1</main_sleep_period>
4   <model_write_time_threshold>60
5     </model_write_time_threshold>
6   <model_write_execs_threshold>20
7     </model_write_execs_threshold>
8
9   <load_monitor>
10     <load_path>/usr/var/pmm/loadhistory</load_path>
11     <write_period>60</write_period>
12     <history_size>60</history_size>
13   </load_monitor>
14
15   ....
16 </config>
```

A.3.2 Load Monitor Configuration

The load monitoring facility is described by a `<load_monitor>` element, this has the following children:

- `<load_path>` (*string, required*) path to a file where load observations are recorded

- `<write_period>` (*integer, default:360*) frequency with which to save the load file to disk (in seconds)
- `<history_size>` (*integer, default:60*) number of load observations to store

A.3.3 Routine Configuration

Each routine is described by a `<routine>` element. Routines have detailed descriptions of the parameters to be passed to them and the construction method that should be used to build their models. An example routine configuration can be seen in Listing [A.2](#), it describes a 2-parameter routine. First, the general options are set using the following child elements:

- `<name>` (*string, required*) The routine name.
- `<exe_path>` (*string, required*) The path to the benchmarking executable
- `<model_path>` (*string, required*) The path to the file where the performance model of the routine will be saved.
- `<exec_args>` (*string, optional*) The value of this parameter will be passed to the benchmarking binary before any arguments which define the problem size that is to be benchmarked. This allows the user to implement say one binary that executes a number of routines based on the initial arguments that are passed.
- `<priority>` (*integer, default:0*) The construction priority this routine has (logically, the higher the value, the higher the priority)

The parameters of a routine are described by a `<parameters>` element. These are the parameters that will be passed to the benchmark binary which ultimately executes the routine which is being modeled. The parameters passed to the benchmark are those that influence the volume of computations or the speed at which the computations are carried out. This is further described in Section [A.4](#). The first child of the `<parameters>` element must be the number of parameter descriptions which will follow:

- `<n_p>` (*integer, required*) number of parameters which the benchmark accepts

Following that, each parameter is described by a `<param>` element. The `<param>` element has a number of child elements which are:

- `<order>` (*integer, required*) The order in which this parameter should be passed to the benchmark binary.
- `<name>` (*string, required*) The name of this parameter.
- `<min>` (*integer, required*) The minimum value this parameter may have. If modelling the performance of the processor while operating in cache only is *not* important, this should be set so the overall problem size is large enough to occupy main memory.
- `<max>` (*integer, required*) The maximum value this parameter may have. This should be large enough to induce significant paging.
- `<stride>` (*integer, default:1*) The stride with which this parameter should be incremented. Stride influences the climbing phase of optimised construction (where successive benchmarks are incremented in size by this value) as well as naive construction (where all points on the stride between min and max are benchmarked). A reasonable value for stride would be, for example, 1/100th of the range between max and min. If stride is too low, excessive time may be spent building a model.
- `<offset>` (*integer, default:0*) Offset for this parameter. If required, you can specify that a parameter value must always be a certain offset from zero.
- `<fuzzy_max>` (*boolean, default:false*) This specifies that the maximum parameter size defined is not a true max and speed at this maximum should be measured.

In normal circumstances the FPM is constructed across a complete range of problem sizes, from small to so large that speed is effectively zero. The

maximum parameter value will be so large that it induces heavy paging. Speed at this maximum is not measured, but assumed to be zero. If this is *not* the case, and the maximum parameter size will not induce heavy paging, `<fuzzy_max>` must be set to *true* for the GBBP algorithm to complete successfully.

Following definition of parameters, a parameter constraint may be defined. This is a formula which uses standard mathematical syntax and the variable names defined above, along with maximum and minimum values for the constraint. The formula will be evaluated and the maximum and minimum values will define the parameter space in which the model is constructed. By way of the parameter constraint, the user may define the memory complexity of a routine and limit the construction of the model to between certain memory footprint sizes. The constraint is defined by `<param_constraint>` which has a number of children:

- `<formula>` (*string, required*) This is the formula which describes the parameter constraint. Any syntax and mathematical operation supported by the muParser library is valid. The variable terms of the formula must be named in the parameter definitions.
- `<min>` (*integer, optional*) This is the minimum value the formula may evaluate to. Combinations of parameters which evaluate to less than this value will not be included in the model construction. If this is not defined, the normal start points which the parameters are described with will be used to limit the model construction region.
- `<max>` (*integer, optional*) This is the maximum value the formula may evaluate to. Combinations of parameters which evaluate to greater than this value will not be included in the model construction. If this is not defined, the normal start points which the parameters are described with will be used to limit the model construction region.

Directives for the construction method must be described by a `<construction>` element. It has the following child elements:

- `<method>` (*string, default:gbbp*) The construction method, this element may have the following values:
 - *gbbp* - the Geometric Bisection Building Procedure will be used to select benchmark points, minimising the number of points required to accurately estimate the model. In multi-parameter models, the diagonal construction method will be used.
 - *gbbp_naive* - the Geometric Bisection Building Procedure will be used to select benchmark points, minimising the number of points required to accurately estimate the model. In multi-parameter models, the boundary construction method will be used.
 - *naive* - all possible points between the parameter ranges will be benchmarked sequentially
 - *naive_bisect* - all possible points between parameter ranges will be benchmarked using a simple bisection algorithm
 - *rand* - points between the parameter ranges will be selected at random
- `<min_sample_num>` (*integer, default:1*) Specify the minimum number of benchmarks to be taken at a single point in the model. Once this is met, the point will be considered as measured and the construction method will proceed to the next point of its choosing.
- `<min_sample_time>` (*integer, default:0*) Specify the minimum number of seconds that should be spent in the benchmarking of a single point before it is considered as measured. I.e. if set to 60 seconds, a benchmark taking 20 seconds will be measured 3 times.

Finally, priority and scheduling policy may be specified. When multiple routines are configured in PMM priorities allow the user to specify which models will be built first. Scheduling policies allow the user limit the execution of benchmarks to certain time periods or certain system conditions.

- <priority> (*integer, default:0*) Priority of construction for the routine. Higher priority routines will have their models constructed before lower ones
- <condition> (*string, default:now*) Condition under which benchmarking of a routine is permitted.
 - *now* - construction is permitted at all times
 - *idle* - construction is only permitted when the observed 5 minute load average is less than 0.10 (note: the act of benchmarking will influence the load average of the system. After the benchmark is complete, PMM will probably have to wait 5 minutes before the next execution can occur)
 - *nousers* - construction is only permitted when no users are logged into the system. Logged in users would be those reported by utilities such as `w`, `who`, `users` and so on.

Listing A.2: Routine Configuration Example

```
1 <routine>
2   <name>dgemm2</name>
3   <exe_path>/usr/local/lib/pmm/dgemm2</exe_path>
4   <model_path>/usr/local/var/pmm/dgemm2_model</model_path>
5   <parameters>
6     <n_p>2</n_p>
7     <param>
8       <order>0</order>
9       <name>m</name>
10      <min>32</min>
11      <max>4096</max>
12      <stride>32</stride>
13      <offset>0</offset>
14    </param>
15    <param>
16      <order>1</order>
17      <name>n</name>
18      <min>32</min>
19      <max>4096</max>
20      <stride>32</stride>
21      <offset>0</offset>
22    </param>
23  </parameters>
24  <construction>
25    <method>gbbp</method>
26    <min_sample_num>5</min_sample_num>
27    <min_sample_time>120</min_sample_time>
28  </construction>
29  <condition>now</condition>
30  <priority>30</priority>
31 </routine>
```

A.4 Building the FPM of a Computation

This section outlines what a user must do to have PMM build the FPM of some computation. The computation may be a library subroutine, a code fragment or an entire process. Throughout this document this computation will be referred

to as a *routine*. The users routine must be wrapped in a benchmarking binary or script which should behave in a specific manner:

- It must accept arguments from the command line which define the volume of computations it must carry out.
- It must execute and time the computation that is to be modeled. Execution may be via a script or compiled binary, written in any language, and the details of how it perform or times the computation do not concern the PMM tool. If the benchmark is written in C/C++, PMM provides some utilities to aid this in a shared library, `libpmm`.
- It must output timing and volume of computations (complexity) in a standard manner. `libpmm` also supports this.

Implementation of this benchmark is a task left to the user. The following sections describe how to choose input parameters, write the benchmark and configure PMM to build a FPM for the routine.

A.4.1 Choosing Parameters of a Routine

The first step a user must take is to identify the parameters of the routine which effect the volume of computations it must carry out or the speed at which those computations are carried out at. Typically, the volume of computations would be floating point operation count, however PMM is agnostic to the type of computations the routine carries out, and the volume may be expressed as the user wishes. The performance model that we build will be expressed in terms of the chosen parameters. Throughout this section we will refer to an example of a square matrix multiplication. In this scenario, there is only one parameter that effects the volume of computations, N , the length of a matrix side in the multiplication.

For a more general case, were two matrices of sizes $N \times K$ and $K \times M$ are multiplied and the result stored in an third matrix of size $N \times M$, then the volume of computations would depend on three parameters, N , M and K .

A general purpose matrix multiplication routine usually has other associated parameters defining transpositions of the input data and other coefficients. These

however do not contribute significantly to the computational complexity of the routine and they should not be considered as parameters of the model in the PMM framework. It is important to note that building an FPM which is in terms of more than one parameter is very intensive as the number of points required to accurately approximate scales exponentially with the number of parameters the model is in terms of. Any parameters of a routine that can be excluded from the functional performance model should be.

A.4.2 Writing a Benchmark for PMM

For PMM to build the performance model of a routine, it must be able to execute benchmarks of that routine for various problem sizes. As previously stated, the problem's size is determined by the parameters which effect the computational complexity of the routine, and the performance model is a function of these parameters.

PMM must be provided with an executable which carries out a benchmark with given input parameters. The user must write this executable so that it behaves in a specified way. PMM is distributed with the source of a number of example benchmarks, here we will list one and reference it as the required behaviours are described below. Listing [A.3](#) shows an example benchmark for a square matrix multiplication routine. The multiplication is provided by the Gnu Scientific Library. Note in a square matrix multiplication, the volume of computations is determined by the size of one side of matrices to be multiplied.

The benchmark code behaves in the following manner:

- The executable must accept a number of parameters on the command line. These parameters will also be described in the configuration entry for the routine. As of version 0.0.1 parameters can only have integer types. (Lines 17-23)
- Based on the parameters passed on the command line, the benchmark must initialise memory and data structures that are to be passed to the routine. If the computation that is to be modelled is just a simple code fragment,

no allocation of memory that occurs within the code fragment should be done in this initialisation phase. (Lines 29-35)

- The benchmark must start a timer, either using timers provided by the PMM shared library, libpmm, or using his own methods (Lines 38-41)
- The benchmark must execute the routine directly after timing is initiated (Line 44)
- the benchmark must stop timing directly after the routine has finished (Line 47)
- the benchmark must print on a single line, to `stdout`, the seconds and microseconds, separated by a single space, elapsed during the routine execution. This can be done using a function provided by libpmm or the users own method. (Line 50)
- the benchmark must print on a new line, the volume of computations made by the routine (typically the number of floating point operations carried out). Long long integers are supported. (Lines 26,38,50)
- on successful completion of the above operations, the benchmark should terminate and return successful exit status, PMM expects this to be equivalent to `EXIT_SUCCESS` as defined by the C standard. (Line 59)

Listing [A.3](#) shows an example benchmark for a square matrix multiplication routine provided by GSL. The in-line comments refer to each of the points made above

Listing A.3: Square Matrix Multiplication Benchmark

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include "pmm_util.h"
4 #include <gsl/gsl_blas.h>
5
6 #define NARGS 1
7
```

```
8 int main(int argc, char **argv) {
9
10     /* declare variables */
11     gsl_matrix *A, *B, *C;
12     double arg;
13     size_t n;
14     long long int c;
15
16     /* parse arguments */
17     if(argc != NARGS+1) {
18         return PMM_EXIT_ARGFAIL;
19     }
20     if(sscanf(argv[1], "%lf", &arg) == 0) {
21         return PMM_EXIT_ARGPARSEFAIL;
22     }
23     n = (size_t)arg;
24
25     /* calculate complexity */
26     c = 2*n*n*(long long int)n;
27
28     /* initialise data */
29     A = gsl_matrix_alloc(n, n);
30     B = gsl_matrix_alloc(n, n);
31     C = gsl_matrix_alloc(n, n);
32
33     gsl_matrix_set_all(A, 2.5);
34     gsl_matrix_set_all(B, 4.9);
35     gsl_matrix_set_zero(C);
36
37     /* initialise timer */
38     pmm_timer_init(c);
39
40     /* start timer */
41     pmm_timer_start();
42
43     /* execute routine */
44     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, A,
45                   B, 0.0, C);
46
47     /* stop timer */
48     pmm_timer_stop();
49
50     /* get timing result */
```

```
50         pmm_timer_result();
51
52         /* destroy timer */
53         pmm_timer_destroy();
54
55         gsl_matrix_free(A);
56         gsl_matrix_free(B);
57         gsl_matrix_free(C);
58
59         return PMM_EXIT_SUCCESS;
60     }
```

A.4.3 Configuring the Benchmark in PMM

Now that the benchmark code is written, the construction of the model must be configured in PMM. This involves indicating the location of the benchmarking binary, describing the maximum and minimum allowable values of N , the steps at which N may be incremented and the construction priority of the model.

In this example, we assume the model will be created for a processor with 1024MiB of memory. We choose a starting value of 32, a stride of 32 and a maximum value of 8196.

Square matrix multiplication operates on 3 matrices of size $N \times N$, each element of the matrix is a double with a size of 8 bytes. Knowing this we choose a starting value that of reasonable computational size, yet still small enough to fit in the L1 cache or certainly L2 cache of modern machines. $N = 32$ results in a problem size that will occupy less than 32KiB of memory. The stride is chosen as 32 so that the problem size will quickly increase beyond the size of the CPU caches and force use of main memory. The maximum size of 8192 has a memory footprint of over 1.5GiB, this ensures that at the maximum problem size the machine will be paging heavily and we can assume it has zero speed at this point.

To illustrate the parameter constraint we effectively limit the construction parameters in the same way by defining the formula as $3 \times 8 \times N^2$ and the max and min as 32KiB and ~ 1.5 GiB.

Listing A.4: Configuration of PMM for a simple Matrix Multiplication Routine

```
1 <?xml version="1.0"?>
2
3 <config>
4     <load_monitor>
5         <load_path>/var/pmm/loadhistory</load_path>
6     </load_monitor>
7
8
9     <routine>
10         <name>square_dgemm</name>
11         <exe_path>/usr/lib/pmm/square_dgemm</
12             exe_path>
13         <model_path>/var/pmm/
14             dgemm_ben_5000_icc_mkl</model_path>
15         <parameters>
16             <n_p>1</n_p>
17             <param>
18                 <order>0</order>
19                 <name>N</name>
20                 <start>32</start>
21                 <end>8196</end>
22             <stride>32</stride>
23             </param>
24
25         <param_constraint>
26             <formula>3*8*N*N</formula>
27             <min>32768 </min>
28             <max>1610612736</max>
29         </param_constraint>
30         </parameters>
31
32         <condition>now</condition>
33         <construction>
34             <method>gbbp</method>
35         </construction>
36     </routine>
37 </config>
```

The construction may be started instantly by executing `pmmcd` with the build only option `-b`. This will construct the model in the foreground and terminate when finished.

A.5 Viewing Models

During the construction or at the end of the process the model may be viewed using the `pmm_view` tool. This is a simple program that access the model file and plots them via a Gnuplot front-end.

`pmm_view` has the following command line options:

- `-a` Plot averages where there are multiple measured speeds at the same points
- `-c <file>` Specify a configuration file from which a routine will be loaded
- `-f <model file>` Specify a model file which will be loaded and plot, multiple instances of the `-f` option will result in multiple plots being placed on the same axes.
- `-h` Print help
- `-i` Plot construction intervals along the base of the axes (these illustrate the GBBP construction process).
- `-I` Enter a Gnuplot terminal mode after the plot has been made (this allows the user to manipulate the intervals of Gnuplot and adjust the display of a model to their choosing
- `-l` List routines in the specified or default configuration file
- `-m` Plot maximum speeds where there are multiple measured speeds at the same points
- `-o <output file>` Save the plot to the file specified, the format of the plot is determined by the suffix of the output file. Suffixes of `.png` and `.eps` or `.ps` are supported. More advanced output options of Gnuplot may be accessed by specify the `-I` option to enter interactive mode.
- `-P` Create coloured plots using the palette option of Gnuplot

- `-r <routine name>` Specify the name of a routine from the configuration file who's model will be loaded and plot, multiple instances of the `-r` option will result in multiple plots being placed on the same axes.
- `-s <slice spec>` This option allows the user to specify a slice of a multi-parameter model that is to be plot. For example, if there are two parameters to the model and the user wishes to specify that the parameter with index 0 is fixed with a value of 256, resulting in a plot of a function that is in terms of the other parameter, the argument would be: `-s p0:256`. `pmm_view` supports plotting with respect to a maximum of two parameters. For high dimensional models, many slices may be specified so that the plot is reduced to being in terms of two parameters or less.
- `-S <style>` Plot using particular Gnuplot style, such as points, linespoints or dots.
- `-w <wait time>` Re-plot the specified models every `<wait time>` seconds. If the model file stored on disk has not changed in the intervening period, the model will not be redrawn as an efficiency measure.

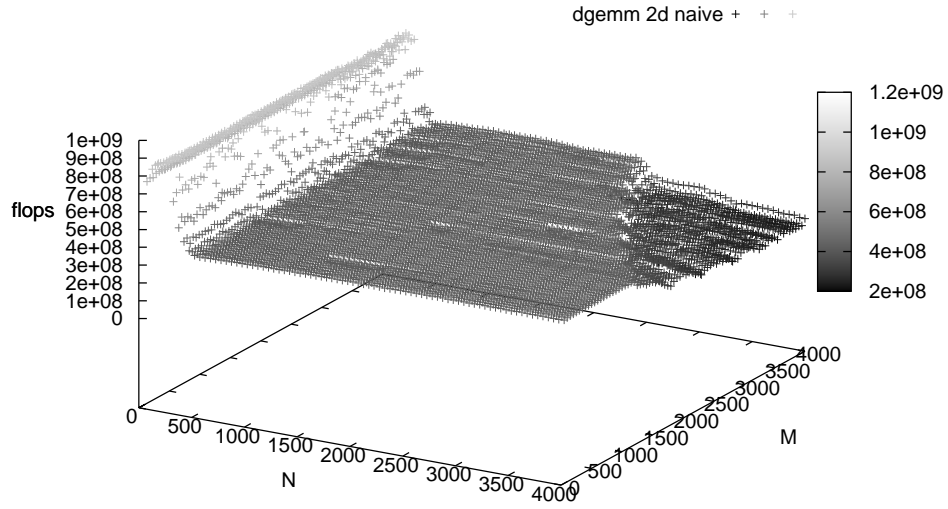
Example output from two executions of `pmm_view` are shown in Figure A.1. A.1a shows the result of the following execution, which demonstrates the calling syntax of the utility:

```
pmm_view -f 2d_model_file -P -S points
```

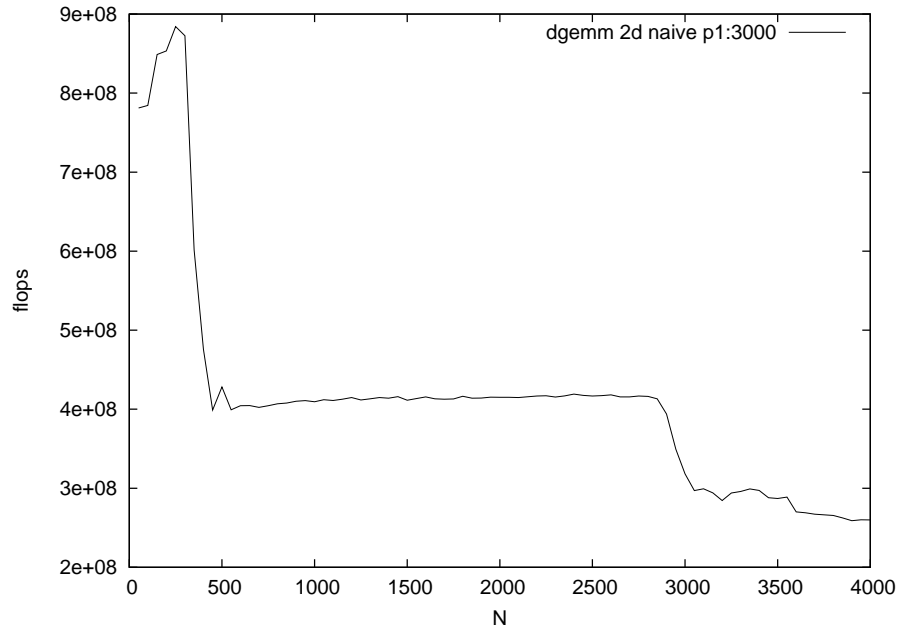
Figure A.1b shows the results of isolating a slice of the first model using the `-s <slice spec>` option. The command in this instance was:

```
pmm_view -f 2d_model_file -s p1:3000 -S lines
```

This instructed the utility to isolate points from the 2 parameter model where the second parameter was equal to 3000.



(a) Example output from `pmm_view` showing a 2 parameter model for a naive matrix multiplication on a machine with 256MiB of memory which has been constructed with naive method.



(b) Slice of the previous model where the second parameter is fixed at 3000, as displayed by the `pmm_view` utility.

Figure A.1: Example output from the `pmm_view` utility.

Bibliography

- [1] T. E. Anderson, D. E. Culler, and D. A. Patterson, “A case for NOW (Networks of workstations),” *IEEE Micro*, vol. 15, no. 1, pp. 54–64, 1995.
- [2] M. W. Mutka and M. Livny, “The available capacity of a privately owned workstation environment,” *Performance Evaluation*, vol. 12, no. 4, pp. 269–284, 1991.
- [3] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, “The interaction of parallel and sequential workloads on a network of workstations,” in *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems - SIGMETRICS '95/PERFORMANCE '95*, (Ottawa, Ontario, Canada), pp. 267–278, 1995.
- [4] P. Eerola, B. Konya, O. Smirnova, T. Ekelof, M. Ellert, J. Hansen, J. Nielsen, A. Waananen, A. Konstantinov, J. Herrala, M. Tuisku, T. Myklebust, F. Ould-Saada, and B. Vinter, “The nordugrid production grid infrastructure, status and plans,” in *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, pp. 158–165, 2003.
- [5] R. Bolze, F. Cappello, E. Caron, M. Dayd, F. Desprez, E. Jeannot, Y. Jgou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E. Talbi, and I. Touche, “Grid’5000: A large scale and highly reconfigurable experimental grid testbed,” *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

- [6] “The TeraGrid project.” <https://www.teragrid.org/>, 2010.
- [7] “Open science grid (OSG).” <http://www.opensciencegrid.org/>, 2010.
- [8] E. Gabriel, M. Resch, T. Beisel, and R. Keller, “Distributed computing in a heterogeneous computing environment,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (V. Alexandrov and J. Dongarra, eds.), vol. 1497 of *Lecture Notes in Computer Science*, pp. 180–187, Springer Berlin / Heidelberg, 1998. 10.1007/BFb0056574.
- [9] N. T. Karonis, B. Toonen, and I. Foster, “MPICH-G2: a grid-enabled implementation of the message passing interface,” *Journal of Parallel and Distributed Computing*, vol. 63, pp. 551–563, May 2003.
- [10] I. Foster, “Globus toolkit version 4: Software for Service-Oriented systems,” *Journal of Computer Science and Technology*, vol. 21, no. 4, pp. 513–520, 2006.
- [11] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, “Matrix multiplication on heterogeneous platforms,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 10, pp. 1033–1051, 2001.
- [12] B. Becker and A. Lastovetsky, “Matrix multiplication on two interconnected processors,” in *Cluster Computing, 2006 IEEE International Conference on*, pp. 1–9, 2006.
- [13] B. Becker and A. Lastovetsky, “Towards data partitioning for parallel computing on three interconnected clusters,” in *Parallel and Distributed Computing, 2007. ISPDC '07. Sixth International Symposium on*, p. 39, 2007.
- [14] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, “Cell broadband engine architecture and its first implementation: a performance view,” *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 559–572, 2007.
- [15] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

- [16] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for gpus: stream computing on graphics hardware,” in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, (New York, NY, USA), pp. 777–786, ACM, 2004.
- [17] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba, “Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment,” in *Proceedings of the 7th international conference on High performance computing for computational science, VECPAR'06*, (Berlin, Heidelberg), p. 305318, Springer-Verlag, 2007.
- [18] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, “An efficient, model-based CPU-GPU heterogeneous FFT library,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–10, 2008.
- [19] A. Ilic and L. Sousa, “Collaborative execution environment for heterogeneous parallel systems,” in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, 2010.
- [20] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama, “Linpack evaluation on a supercomputer with heterogeneous accelerators,” in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–8, 2010.
- [21] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, (Atlantic City, New Jersey), p. 483, 1967.
- [22] M. Hill and M. Marty, “Amdahl’s law in the multicore era,” *Computer*, vol. 41, no. 7, pp. 33–38, 2008.
- [23] M. Annavaram, E. Grochowski, and J. Shen, “Mitigating amdahl’s law through EPI throttling,” in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pp. 298–309, 2005.

- [24] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, “Single-isa heterogeneous multi-core architectures for multithreaded workload performance,” *SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 64, 2004.
- [25] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, “Cell broadband engine architecture and its first implementation - a performance view,” *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.
- [26] H. Kondo, M. Nakajima, N. Masui, S. Otani, N. Okumura, Y. Takata, T. Nasu, H. Takata, T. Higuchi, M. Sakugawa, H. Fujiwara, K. Ishida, K. Ishimi, S. Kaneko, T. Itoh, M. Sato, O. Yamamoto, and K. Arimoto, “Design and implementation of a configurable heterogeneous multicore SoC with nine CPUs and two matrix processors,” *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 4, pp. 892–901, 2008.
- [27] Xilinx Inc., “Virtex-II pro platform FPGA handbook,” Jan. 2002.
- [28] Triscend Corp., “Triscend a7 datasheet,” 2001.
- [29] Intel Corp., “Intel atom processor E6x5C Series-Based platform for embedded computing,” 2010.
- [30] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, “The impact of performance asymmetry in emerging multicore architectures,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 506–517, 2005.
- [31] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero, “The international exascale software project: a call to cooperative action by the global High-Performance community,” *International Journal of High Performance Computing Applications*, vol. 23, pp. 309–322, Nov. 2009.
- [32] R. P. Weicker, “Dhrystone: a synthetic systems programming benchmark,” *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

- [33] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK benchmark: Past, present, and future,” *Concurrency and Computation: Practice and Experience*, vol. 15, p. 820, 2003.
- [34] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, “HPL - a portable implementation of the High-Performance linpack benchmark for Distributed-Memory computers,” 2008.
- [35] “TOP500 supercomputing sites.” <http://www.top500.org/>, 2010.
- [36] “SPEC - standard performance evaluation corporation.” <http://www.spec.org/>, 2010.
- [37] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatesh, and S. Weeratunga, “The nas parallel benchmarks,” *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63 –73, 1991.
- [38] D. Bailey, T. Harris, W. Saphir, R. V. der Wijngaart, A. Woo, and M. Yarrow, “The NAS parallel benchmarks 2.0,” Tech. Rep. NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.
- [39] “The DEISA benchmarking suite.” <http://www.deisa.eu/science/benchmarking>, 2010.
- [40] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere, “Performance prediction based on inherent program similarity,” in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques - PACT '06*, (Seattle, Washington, USA), p. 114, 2006.
- [41] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu, “Performance projection of HPC applications using SPEC CFP2006 benchmarks,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, 2009.

- [42] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, p. 1925, Dec. 1995.
- [43] "STREAM2 home page." <http://www.cs.virginia.edu/stream/stream2/>, 2010.
- [44] J. Gustafson and Q. Snell, "HINT: a new way to measure computer performance," in *System Sciences, 1995. Vol. II. Proceedings of the Twenty-Eighth Hawaii International Conference on*, vol. 2, pp. 392–401 vol.2, 1995.
- [45] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," pp. 96–97, Apr. 1996.
- [46] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *Concurrency, IEEE*, vol. 6, no. 3, pp. 42–50, 1998.
- [47] N. Lopez-Benitez and J. Hyon, "Simulation of task graph systems in heterogeneous computing environments," in *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pp. 112–124, 1999.
- [48] M. Maheswaran, S. Ali, H. Siegal, D. Hensgen, and R. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pp. 30–44, 1999.
- [49] S. Ali, H. Siegel, M. Maheswaran, and D. Hensgen, "Task execution time modeling for heterogeneous computing systems," in *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pp. 185–199, 2000.
- [50] J. Kim, S. Shivle, H. Siegel, A. Maciejewski, T. Braun, M. Schneider, S. Tideman, R. Chitta, R. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, and S. Yellampalli, "Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines,"

- in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, p. 15 pp., 2003.
- [51] B. Reistad and D. K. Gifford, "Static dependent costs for estimating execution time," *SIGPLAN Lisp Pointers*, vol. VII, p. 6578, July 1994.
- [52] D. Pease, A. Ghafoor, I. Ahmad, D. Andrews, K. Foudil-Bey, T. Karpinski, M. Mikki, and M. Zerrouki, "PAWS: a performance evaluation tool for parallel computing systems," *Computer*, vol. 24, no. 1, pp. 18–29, 1991.
- [53] J. Yang, I. Ahmad, and A. Ghafoor, "Estimation of execution times on heterogeneous supercomputer architectures," in *Parallel Processing, 1993. ICPP 1993. International Conference on*, vol. 1, pp. 219–226, 1993.
- [54] M. Devarakonda and R. Iyer, "Predictability of process resource usage: a measurement-based study on UNIX," *Software Engineering, IEEE Transactions on*, vol. 15, no. 12, pp. 1579–1586, 1989.
- [55] M. A. Iverson, F. Ozguner, and G. J. Follen, "Run-Time statistical estimation of task execution times for heterogeneous distributed computing," in *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing, HPDC '96*, (Washington, DC, USA), p. 263, IEEE Computer Society, 1996.
- [56] R. Freund, T. Kidd, D. Hensgen, and L. Moore, "SmartNet: a scheduling framework for heterogeneous computing," in *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings. Second International Symposium on*, pp. 514–521, 1996.
- [57] M. Iverson, F. Ozguner, and L. Potter, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," in *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pp. 99–111, 1999.

- [58] E. Caron, F. Desprez, M. Quinson, and F. Suter, “Performance Evaluation of Linear Algebra Routines,” *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 373–390, Fall 2004.
- [59] P. E. Crandall and M. J. Quinn, “Problem decomposition for Non-Uniformity and processor heterogeneity,” *Journal of the Brazilian Computer Society*, vol. 2, pp. 13–23, July 1995.
- [60] M. Kaddoura, S. Ranka, and A. Wang, “Array decompositions for nonuniform computational environments,” *Journal of Parallel and Distributed Computing*, vol. 36, pp. 91–105, Aug. 1996.
- [61] E. Dovolnov, A. Kalinov, and S. Klimov, “Natural block data decomposition for heterogeneous clusters,” in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, p. 10 pp., 2003.
- [62] P. E. Crandall and M. J. Quinn, “A partitioning advisory system for networked Data-Parallel processing,” *Concurrency: Practice and Experience*, vol. 7, pp. 479–495, Aug. 1995.
- [63] M. J. Zaki, W. Li, and M. Cierniak, “Performance impact of processor and memory heterogeneity in a network of machines,” in *4th Heterogeneous Computing Workshop, also TR574*, 1995.
- [64] A. Lastovetsky, “mpC: A Multi-Paradigm Programming Language for Massively Parallel Computers,” *ACM SIGPLAN Notices*, vol. 31, pp. 13–20, 1996.
- [65] A. Lastovetsky and R. Reddy, “HeteroMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers,” *Journal of Parallel and Distributed Computing*, vol. 66, pp. 197–220, 2006.
- [66] M. Drozdowski and P. Wolniewicz, “Out-of-core divisible load processing,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 14, no. 10, pp. 1048–1056, 2003.

- [67] A. Lastovetsky and R. Reddy, “Data partitioning with a realistic performance model of networks of heterogeneous computers,” in *In International Parallel and Distributed Processing Symposium IPDPS2004. IEEE Computer*, pp. 26—30, 2004.
- [68] “CUDA toolkit 3.0 (March 2010).” <http://developer.nvidia.com/>, 2010.
- [69] “AMD core math library for graphic processors (ACML-GPU) | AMD developer central.” <http://developer.amd.com/gpu/acmlgpu/pages/default.aspx>, 2010.
- [70] R. Higgins and A. Lastovetsky, “Managing the Construction and Use of Functional Performance Models in a Grid Environment,” in *The 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.
- [71] D. Clarke, A. Lastovetsky, and V. Rychkov, “Dynamic Load Balancing of Parallel Computational Iterative Routines on Platforms with Memory Heterogeneity,” in *Europar 2010 / Heteropar 2010*, 2010.
- [72] A. Lastovetsky, R. Reddy, and R. Higgins, “Building the functional performance model of a processor,” in *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006)*, (Dijon, France), ACM, 2006.
- [73] D. Ferrari and S. Zhou, “An Empirical Investigation of Load Indices for Load Balancing Applications,” in *Performance '87: Proceedings of the 12th International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pp. 515–528, 1987.
- [74] T. Kunz, “The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme,” *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 725–730, 1991.
- [75] R. Wolski, N. T. Spring, and J. Hayes, “Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid,” in *Proceed-*

- ings of the 8th IEEE High Performance Distributed Computing Conference (HPDC8)*, 1999.
- [76] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo, “Scheduling Parallel Applications in Networks of Mixed Uniprocessor/Multiprocessor Workstations,” in *Proceedings of ISCA 11th Conference on Parallel and Distributed Computing*, 1998.
- [77] I. Foster, “Globus toolkit version 4: Software for service-oriented systems,” *Journal of Computer Science and Technology*, vol. 21, pp. 513–520, 2006. 10.1007/s11390-006-0513-y.
- [78] M. Litzkow, M. Livny, and M. Mutka, “Condor-a hunter of idle workstations,” pp. 104–111, jun. 1988.
- [79] P. A. Dinda, “The statistical properties of host load,” *Scientific Programming*, vol. 7, no. 3/4, p. 211, 1999.
- [80] P. A. Dinda and D. R. O’Hallaron, “An evaluation of linear models for host load prediction,” in *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pp. 87–96, 1999.
- [81] L. Gong, X. Sun, and E. Watson, “Performance modeling and prediction of nondedicated network computing,” *Computers, IEEE Transactions on*, vol. 51, no. 9, pp. 1041–1055, 2002.
- [82] T. Brady, J. Dongarra, M. Guidolin, A. Lastovetsky, and K. Seymour, “SmartGridRPC: The new RPC model for high performance Grid computing,” *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 2467–2487, 2010.
- [83] S. Zhou, “A trace-driven simulation study of dynamic load balancing,” *Software Engineering, IEEE Transactions on*, vol. 14, no. 9, pp. 1327–1341, 1988.
- [84] K. Goto, “GotoBLAS2.” <http://www.tacc.utexas.edu/tacc-projects/gotoblas2/>, 2010.

- [85] D. E. Knuth, *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, Jan. 2011.
- [86] J. Kelleher and B. O’Sullivan, “Generating all partitions: A comparison of two encodings,” *0909.2331*, Sept. 2009.
- [87] S. G. Akl and I. Stojmenovic, “Parallel algorithms for generating integer partitions and decompositions,” *The Journal of Combinatorial Mathematics and Combinatorial Computing*, vol. 13, pp. 107–120, Apr. 1993.
- [88] P. A. MacMahon, “Memoir on the theory of the compositions of numbers,” *Philosophical Transactions of the Royal Society of London. (A.)*, vol. 184, pp. 835–901, Jan. 1893.
- [89] J. H. Holland, *Adaptation in natural and artificial systems*. Cambridge, MA, USA: MIT Press, 1992.
- [90] A. Lastovetsky and J. Twamley, “Towards a Realistic Performance Model for Networks of Heterogeneous Computers,” in *The 2004 IFIP International Symposium on High Performance Computational Science and Engineering (HPCSE-04)*, pp. 39–58, Springer, 2005.
- [91] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov, and D. Sorensen, “LAPACK Users’ Guide, Release 3.0,” 1999.
- [92] J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, “A set of level 3 basic linear algebra subprograms,” *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, 1990.
- [93] R. C. Whaley, A. Petitet, and J. J. Dongarra, “Automated empirical optimizations of software and the ATLAS project,” *Parallel Computing*, vol. 27, pp. 3–35, Jan. 2001.
- [94] R. Higgins and A. Lastovetsky, “PMM: Performance Model Manager.” <http://hcl.ucd.ie/project/pmm>.
- [95] T. Williams and C. Kelley, “Gnuplot.” <http://www.gnuplot.info>.

- [96] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. A. Lee, and H. Casanova, "Overview of GridRPC: A Remote Procedure Call API for Grid Computing," in *GRID* (M. Parashar, ed.), vol. 2536 of *Lecture Notes in Computer Science*, pp. 274–278, Springer, 2002.
- [97] J. Dongarra, "NetSolve: a network server for solving computational science problems," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, pp. 212–223, 1997.
- [98] Y. Caniou and E. Jeannot, "Multi-Criteria scheduling heuristics for GridRPC systems, in "international journal of high performance computing applications," *Journal of High Performance Computing Applications*, vol. 20, pp. 61–76, 2005.
- [99] A. Yarkhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra, "Recent developments in GridSolve," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 20, p. 2006, 2006.
- [100] C. Gheller, O. Pantano, and L. Moscardini, "A cosmological hydrodynamic code based on the piecewise parabolic method," *Monthly Notices of the Royal Astronomical Society*, vol. 295, p. 519, Apr. 1998.
- [101] M. Guidolin and A. Lastovetsky, "Grid-Enabled Hydropad: a Scientific Application for Benchmarking GridRPC-Based Programming Systems," in *The 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.
- [102] A. Lastovetsky, T. Brady, and M. Guidolin, "Experiments with Smart-GridSolve: achieving higher performance by improving the GridRPC model," in *The 9th IEEE/ACM International Conference on Grid Computing*, (Tsukuba, Japan), 2008.