Jan Christian Meyer

Performance Modeling of Heterogeneous Systems

Thesis for the degree of Philosophiae Doctor

Trondheim, November 2012

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Computer and Information Science

ISBN 978-82-471-4015-4 (printed ver.)
ISBN 978-82-471-4016-1 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2012:344

Printed by NTNU-trykk

For Adrien and Martin Manó,
my beloved family
who keep me synchronized.

Abstract

As the complexity of parallel computers grows, constraints posed by the construction of larger systems require both greater, and increasingly non-linear, parameter sets to model their behavior realistically. These heterogeneous characteristics create a trade-off between the complexity and accuracy of performance models, creating challenges in utilizing them for design decisions.

In this thesis, we take a bottom-up approach to realistically model software and hardware interactions, by composing system models from simpler, linear models, which allow parts of the analysis to be automated. We associate empirically benchmarked platform performance metrics with the core elements in a variant of bulk-synchronous execution, aiming to quantify application performance, and associated potential for computation and communication overlap on SMP clusters.

The original bulk-synchronous performance model is introduced, and we identify areas of computation and communication where its abstractions impede realistic models of contemporary hardware. These are addressed independently, using experimental evidence to develop a representation collecting computation kernel characteristics and pairwise communications in matrices, to combine into a system model. As bulk-synchronous execution strongly depends on periodic, global synchronization, we develop a cost model for it by combining latency measurements with a parametric representation of signalling patterns, and experimentally verify the resulting predictions for three common algorithms.

We describe a design to implement the BSPLib programming interface, combining threads and message-passing parallelism to achieve overlap on commodity cluster platforms, implementing its one-sided communication primitives using out-of-band control messages. We augment and validate the cost model of one adapted synchronization algorithm with the corresponding bandwidth requirement, completing a framework for modeling BSPLib program performance.

Finally, we test the utility of this framework as a proof-of-concept for guiding software performance adaptations, using two cases. First, we use the latency terms to automatically generate synchronization operations, using model predictions to generate customized patterns with respect to platform topology, showing that the resulting algorithms equal or outperform the system defaults. Second, the strong scaling characteristics of a 5-point stencil code is compared for three implementations. Experiments show the performance overhead of our implementation, but also its capability for predicting program cost, including parameter values to optimize for balanced overlapping of computation and communication.

Preface

This thesis is submitted to Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree Philosophiæ Doctor.

The work herein was performed at the Department of Computer and Information Science, NTNU, Trondheim, under the supervision of Associate Professor Anne C. Elster.

The work was financed by the Faculty of Information Technology, Mathematics and Electrical Engineering.

Acknowledgements

Pursuing a single project for such an extended period is a rare opportunity. First and foremost I wish to thank my advisor, Anne C. Elster, for affording me the possibility, and for all her tireless work in helping me through it. I also extend my gratitude to co-advisors Lasse Natvig and Jørn Aslak Amundsen, who have offered very helpful feedback and advice along the way.

Many heartfelt thanks go to fellow Ph.D. students Thorvald Natvig, Rune Erlend Jensen, Magnus Jahre, Asbjørn Djupdal, Marius Grannæs, Dragana Laketić, Nils Grimsmo and Truls Amundsen Bjørklund, and everyone else along the corridor, who made it a memorable time for reasons beyond research work.

Magnus Lie Hetland and Peter Henry Hughes both deserve my sincere thanks, for pointing me directly to key insights, at particular times when I was all out of ideas.

Finally, I am most deeply grateful to the near and dear: Adrien and our son Martin Manó, our parents Sissel and Jan, Marika and Vince, my brother Bjørn, and my incomparable friend Egil Albertsen. You have all patiently and thoughtfully suffered my absence, supported, listened, discussed, commented, sympathized, sacrificed, inspired, and forgiven a thousand broken promises. Here you have what kept me for so long.

Jan Christian Meyer
Trondheim, November 2012

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The main aim of this thesis is to derive performance models of complex program and platform interactions which admit automated support for performance tuning. Recent generations of parallel computers are composed of subsystems with highly variable, nonlinear performance properties, which make them challenging to model accurately. Maintaining an illusion of a large memory with uniform access cost is already impossible in most cases, and the communication costs of distributed memory systems are influenced by many factors. The sustainable computational rate of a processor is tied to the memory access patterns of programs, making it variable even on systems composed of identical processors. Models expressed as small sets of linear parameters do not reflect these heterogeneous characteristics, but increasing the level of model detail detracts from both clarity and generality.

In order to maximize the efficiency of parallel programs, it is necessary to chart the sustainable load of component subsystems, leading to the consideration of how far communication and computation can be overlapped. Because the cost of communication is fundamentally dependent on the distance between the communicating parties, balancing the two is essential in order to permit system scale to grow without diminishing the utility of the added computational resources.

The method described in this thesis approaches system model complexity based on the existence of effective linear models of individual subsystems. Assuming the distribution of a known, finite workload onto these subsystems, the composition of overall system behavior from a heterogeneous collection of subsystems can be automated. This permits system models to incorporate a great number of parameter values without requiring an analyst to manually manipulate them all.

1.1 Modeling Challenges And Scope

Ideally, a performance model should be simple, general, and provide strong predictions. These objectives conflict with the need to capture system complexity, as it requires structural information about both system and program to be taken into account. The resulting trade-offs make it unrealistic to search for a single, correct approach to all systems. This section outlines the choices we make to produce a usable framework for deriving performance models for systems with heterogeneous performance parameters.

In this thesis, we have chosen to model heterogeneous systems by extending the Bulk-Synchronous Parallel (BSP) model [95]. Its purpose is to form a *bridging* model, combining aspects of parallel computation from several levels of abstraction.

Figure 1.1 illustrates how BSP unifies algorithmic, programmatic, processing and performance models in terms of a few, global concepts. In order to adapt it to heterogeneous systems, we adjust the processing model to permit performance models of greater detail. BSP is chosen because it makes it possible to exchange these parts without violating the semantics of the algorithmic and programmatic components, allowing this thesis to utilize and complement existing programs and algorithms in the body of related work.

Our objective is to keep component models sufficiently simple, so that they can be represented in a uniform manner programmatically. Linear models of subsystems are appropriate for this purpose, because of the relative simplicity of manipulating large systems of linear equations in software. Such an approach carries two significant limitations. One is that it requires a bound on the time interval for which the model should be valid, in order to derive the amount of work delegated to each subsystem. The other lies in the assumption that a piecewise linear description of global behavior can be obtained from a subsystem decomposition.

Bulk-synchronous execution inherently partitions computation into bounded intervals, restricting our scope of study to synchronized or loosely synchronized algorithms. A 1996 technical report by Fox [36] estimates that this accounts for 90% of parallelized problems in scientific computing. Assessing the accuracy of that number is beyond the scope of this thesis, but we argue that synchronized computation is an important area of study, while noting that our approach is poorly suited to asynchronous algorithms.

Nonlinear subsystem models are not addressed in this thesis because composing an overall system of nonlinear equations greatly complicates automatic manipulation, detracting from its effectiveness for hiding model complexity. Computation rate is nonlinearly related to the data traffic caused by problem specific properties. This is approached by treating such functions as piecewise linear, and decomposing them into a discontinuous set of linear models. While this works in the practical cases investigated here, it increases the amount of manual labor involved in modeling.

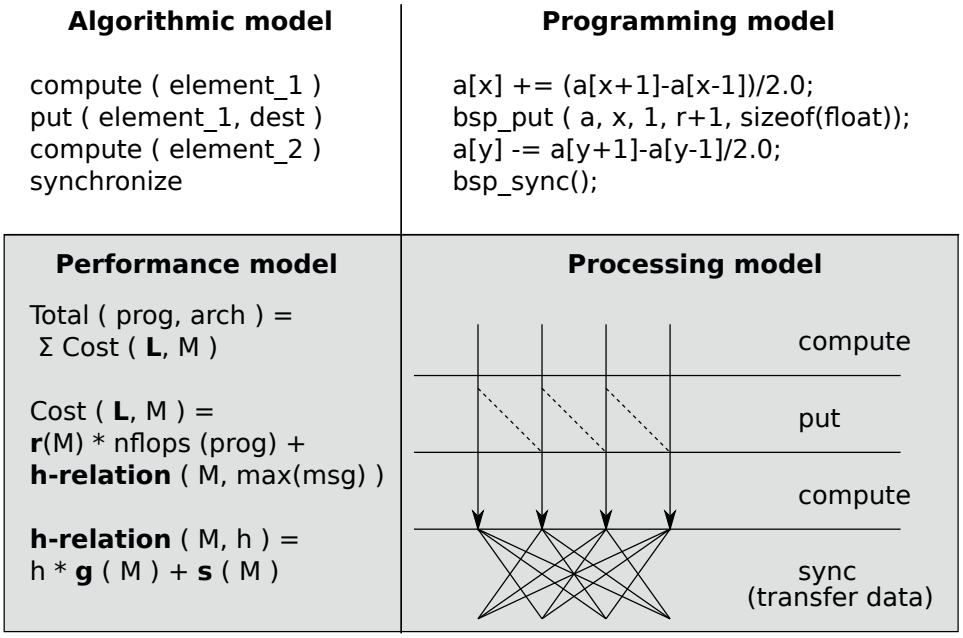

Figure 1.1: Aspects of the BSP model

Algorithmic *and* **programming** *models represent the abstractions presented for program design purposes, as a set of fundamental operations and their corresponding programming language support. The* **processing** *model shows a synchronized* superstep *where communication is effected by a total exchange at the end. It serves as a shared abstraction to both software and hardware architecture. The* **performance** *model attaches cost functions to key elements of programs and platforms. Specifically,* **L** *is the periodicity of the program,* **h** *is the maximal amount of data communicated between a pair of parallel processes,* **r** *is the rate of computation,* **g** *is the throughput of the communication infrastructure, and* **s** *is the latency. L is written as a function of the program, to reflect variations in the amount of work in a given superstep. M is not explicitly acknowledged in the original notation, but is introduced here to acknowledge the platform-dependency of cost functions. This notation will be significantly altered with our model refinements, but is stated here to clarify connections with related work.*

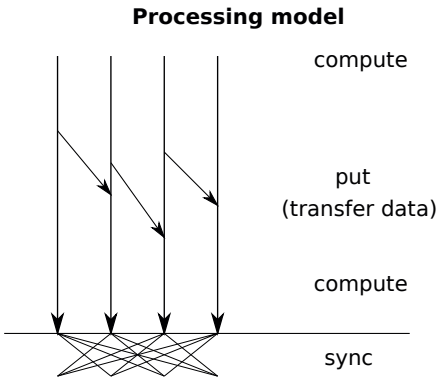

Figure 1.2: Alternative processing model

1.2 Framework Outline

Figure 1.2 illustrates our modified processing model, which retains the BSP semantics that effects of communication are not observed until after synchronization/total exchange. The change amounts to initiating communication as early as permissible, decoupling the cost of communication from synchronization. This reduces the amount of communication required at synchronization time, and holds the potential for mitigating interconnect contention, as communication may happen at the individual process' discretion when the message is ready.

Accounting for background communication, as well as a heterogeneous performance model, Figure 1.3 shows an overview of our approach for modeling application behavior at the system level. It is structured according to what Barker *et al.* [14] name the "fundamental equation of modeling", given in Equation 1.1.

$$T_{total} = T_{compute} + T_{communicate} - T_{overlap} \tag{1.1}$$

Selecting the computational *superstep* [95] as the model unit of work implies a division of computation and communication time totals into non-maskable and maskable parts. Equations 1.2 and 1.3 express the non-maskable time as the difference of total requirement T and a maskable part T'.

$$T_{comm-nonmaskable} = T_{comm} - T'_{comm} \tag{1.2}$$

$$T_{comp-nonmaskable} = T_{comp} - T'_{comp} \tag{1.3}$$

This allows the right hand side of Equation 1.1 to be restated as Equation 1.4, with total time representing the superstep cost.

$$T_{total} =$$

$$(T_{comp} - T'_{comp}) + (T_{comm} - T'_{comm}) + \max(T'_{comp}, T'_{comm}) + T_{sync} \tag{1.4}$$

This formulation indicates that a *superstep* consists of some sequentially dependent work, some which can be overlapped (bounding total time to the greatest requirement), and the synchronization cost of a semantic fence to mark the completion of both.

The approach proceeds bottom-up, in the 3 stages shown in Figure 1.3:

1. Approximate T_{comp}, T_{comm} and T_{sync} separately
2. Combine the approximations in a linear system which describes collective behavior
3. Derive a system-level model of execution

Details concerning each of these stages are developed in subsequent chapters of this thesis. The underlying goal is to manage the complexity of the resulting system-level model by selecting simplifications to facilitate automatic model manipulation. Thus, the point of modeling system behavior as (potentially large) linear systems in Stage 2 is to admit heterogeneous collections of subsystem performance characteristics.

An important feature to note in Figure 1.3 is that the starting point of Stage 1 is a separation of program and platform characteristics. The purpose of initially considering these in isolation is to consider their representation as parameters by Stage 2, so that a model of one may be applied to several instances of the other. In particular, there is strong focus on keeping the topology of the communication infrastructure parametric. This requires coupling the physical locality of a process to its position in the logical layout of a program. Locality is observed to be an important factor in determining the cost of communication, implying that accurate modeling requires its impact to be kept under strict control.

1.3 Research Questions

The main research question of this thesis is
How can automation support the analysis of interactions between a parallel algorithm and the executing platform when both show heterogeneous performance characteristics?

Addressing this question breaks into more specific research questions, which pertain to the requirements of adapting the approach in Figure 1.3 to a particular system:

RQ1 How can the computation and communication requirements of a program be coupled to an independent profile of the executing platform?

RQ2 How can the impact of synchronization on program performance be determined?

RQ3 Which constraints govern the accuracy of performance predictions produced using the developed framework?

RQ4 How suitable is the framework for the purposes of automatic application performance tuning?

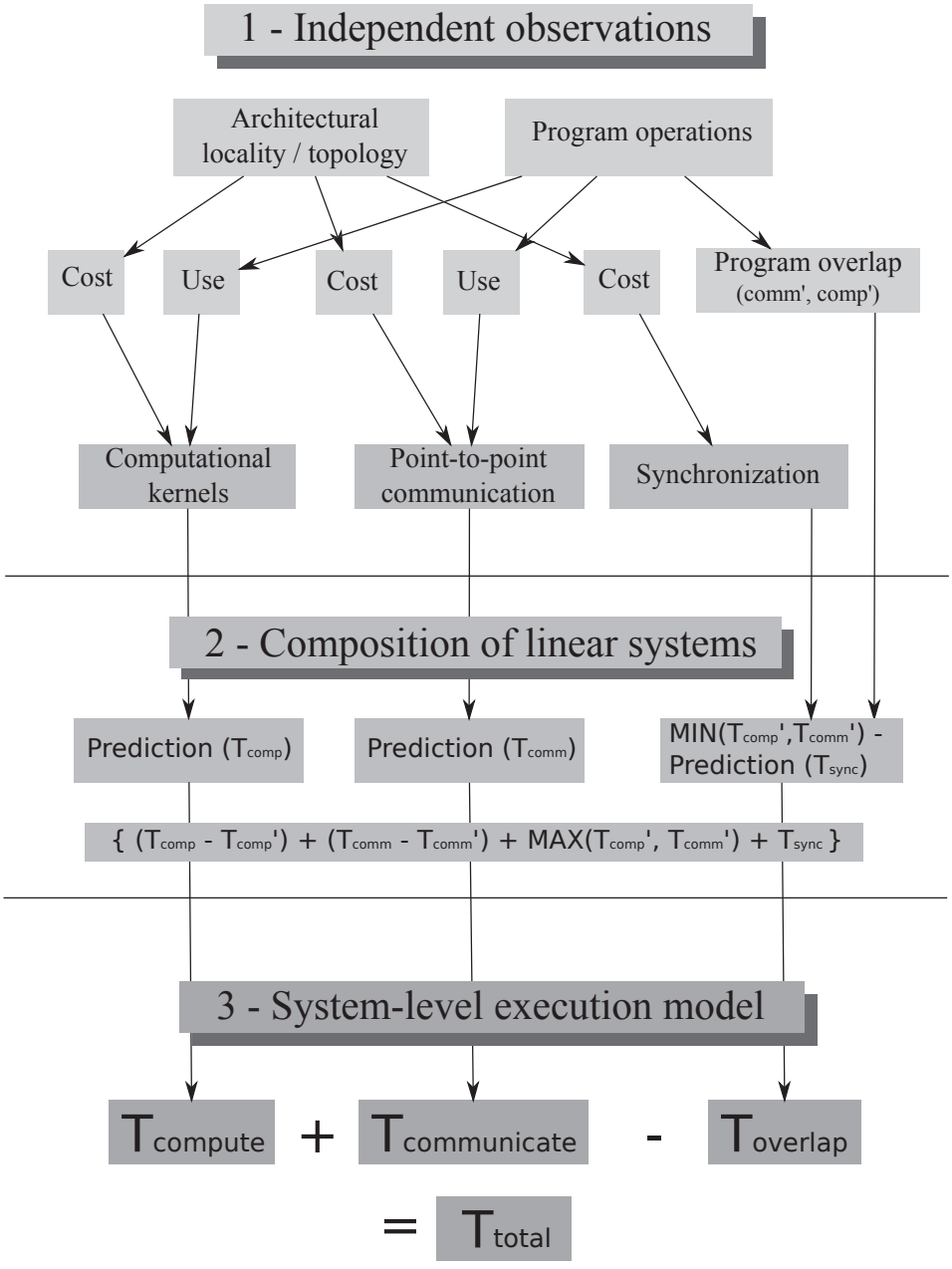

Figure 1.3: Outline of the Proposed Framework

1.4 Contributions

Several variations over models found in the body of related work are employed throughout this thesis. Its novelty lies in coupling application modeling techniques for pairwise communication, collective behavior, and computation rates with programming and processing models adapted from a more theoretical approach. Strong focus on experimental validation is maintained throughout, to ensure that the developed framework is practically applicable. The remainder of this section gives further details of our contributions.

1.4.1 A Bulk-Synchronous Programming Tool

The BSP model is already endowed with a programming interface specification. The work presented in this thesis constructs an implementation of this interface, modifying the processing model to employ asynchronous communication. The corresponding experimental work provides evidence that the combination is a simple and effective means to identify and exploit an algorithm's potential for computation/communication overlap, to the extent made feasible by the target platform.

1.4.2 A Modeling Framework to Capture Overlap Using Linear Systems

The tradition of modeling homogeneous parallel systems in terms of constants or piecewise linear functions grows in complexity when applied to systems where performance parameters are greater in number and range. The presented framework approaches the composition of subsystems by expanding these into matrices containing individual or pairwise performance parameters, and deriving overlap as a collective property of the resulting linear systems.

This approach retains the favorable property of aggregating system models by composition of subsystems, without concealing all structural information. It is shows robustness in the face of parameter values varying by several orders of magnitude.

1.4.3 Benchmarks for Commodity SMP Clusters

To validate the framework on COTS systems of multi-chip, multi-core compute nodes, it is tested on commodity Linux clusters with variable topology and configuration. Because the model relies on empirical data in order to characterize the performance impact of deploying an algorithm on a given platform, these clusters are benchmarked to produce their key parameters. The method of obtaining these benchmarks is described, as great care must be taken in order to give the stability and accuracy necessary for validation.

1.4.4 A Method for Automatic Barrier Adaptation

Because synchronization costs are central to the framework, a detailed cost model of barrier synchronization is developed, including a general matrix representation of arbitrary barrier communication patterns. Since our predictions are shown through experiments to be very accurate, they can be used to produce customized barrier implementations of superior performance to those provided by available system libraries.

1.4.5 A Method for Determining Application Overlap

A system's ability to mask communication by simultaneous computation is of great and growing significance to sustained, scalable performance. Its magnitude is, however, composed of both algorithmic dependencies and the architectural facilities for exploiting it. The implementation and instrumentation necessary to realize this potential and estimate its effectiveness, can require a significant amount of application restructuring. Bulk-synchronous execution semantics allow overlap to be automatically exploited by following the simple programming rule of committing communication as early as possible. The effectiveness of a model derived from our framework is illustrated by its correct identification of parameter values for optimal overlap in a simple application.

1.5 Structure of the Thesis

The structure of this thesis follows the stages of Figure 1.3, through development, testing, and application of a corresponding programming library and performance model. After initial considerations of computation speed, communication model terms are approached with a view to vertical integration. Communication cost is first estimated as a function of topological distance, to estimate synchronization cost. Communication patterns of several synchronization strategies are encoded as application requirements in a reduced model without cost functions for computation and synchronization. This is developed into a synchronization cost function which can be integrated with computation, and the implementation of a corresponding run-time library is described. Finally, the model is applied to automatic analysis of synchronization patterns and a small application program. Results demonstrate that the independent components can be integrated in a model which supports program optimization.

The remaining chapters are structured as follows:

Chapter 2 surveys a spectrum of parallel platforms and programming models with respect to heterogeneity and scale, to establish terminology and place the test system class in a greater context.

Chapter 3 introduces the background to motivate the framework's construction, and presents its basic terms, with emphasis on the relationship between bulk synchronicity and the fundamental equation of modeling.

Chapter 4 describes the challenges posed to stable metrics of computational rate imposed by the memory hierarchy of contemporary platforms, and shows the assumptions and methods applied to the test systems.

Chapter 5 describes the expression of the model's communication startup cost components, which provide accurate performance predictions for the cost of several synchronization algorithms applicable to the test systems.

Chapter 6 describes the techniques employed to create an implementation of the *BSPlib* programming interface which utilizes an application's potential for overlap, and exposes its magnitude on a given target platform. Attention is focused on a particular synchronization algorithm, extending its cost function to include a minimal data payload. This allows it to function as a special case of a total exchange collective, which establishes the synchronization cost estimate required by the framework.

Chapter 7 demonstrates the applicability of the framework to fully automate the construction of generic synchronization algorithms, by examining the model's prediction of their interaction with independently captured architectural profiles.

Chapter 8 introduces a simple finite difference application, and compares performance expectations to the results obtained by studying it with the developed model.

Chapter 9 draws conclusions, and outlines the potential for exploring the framework approach as a tool for guiding manual and automatic performance tuning.

Chapter 2

Scalability and Heterogeneity

Much of the complexity and diversity of parallel computing is due to how the requirements scaling solutions to ever greater problems conflict with those of implementing such solutions using a uniform set of resources. The resulting trade-offs manifest themselves both in hardware and software design. In order to provide an appropriate context for our research, this chapter presents a brief, qualitative survey of how this relationship is reflected in a range of systems.

Section 2.1 defines the distinguishing characteristics of scalability and heterogeneity for the purposes of this discussion. Section 2.2 applies these characteristics to classes of parallel hardware, while Section 2.3 addresses characteristics of programming model classes. Finally, Section 2.4 describes the context of our work, including how other models have contributed influential points.

2.1 Terminology

Since no commonly accepted definitions of heterogeneity and scalability exist, the scope of both terms are defined in the following sections, where they will be related to both the construction of parallel computer platforms, as well as programming models.

2.1.1 Architectural Scalability

According to Hennessy and Patterson[43], scalability was long considered a property which could be built into an architectural design. Their further discussion of multiprocessor systems offers no succinct updated view, but it indicates that difficulties stem from increased requirements to grow interprocessor communication networks. Hwang and Xu[49] divide scalability into *resource*, *application* and *technology* scalability, further specified in

terms of various properties such as machine size, software scalability and heterogeneity scalability. They list of four design principles for scalability, which are

1. The principle of independence
2. The principle of balanced design
3. The principle of design for scalability
4. The principle of latency hiding

Parallel hardware scalability will be discussed in terms of these four principles. The principle of independence states that dependencies between system components should be minimized. The principle of balanced design states that any performance bottleneck should be minimized. The principle of design for scalability states that scalability should be acknowledged from the beginning of a design process, reflected in *overdesign*, *i.e.* features which anticipate future extensions, and *backward compatibility* for the sake of downscaling. The principle of latency hiding refers to exposing the potential for exploiting simultaneous execution and communication, to conceal startup cost. Patterson's treatment of the topic [82] states that "In the time that bandwidth doubles, latency improves by no more than a factor of 1.2 to 1.4", and reasons that this trend can be expected to continue. Accordingly, future designs should invent further techniques like caching, replication and prediction, to reduce the impact of latency at the expense of other resources.

2.1.2 Architectural Heterogeneity

Our classification of architectural heterogeneity will take the high-level view that the defining characteristics of parallel computers are the processing and communication facilities, with the latter encompassing the effects of hierarchical memory subsystems. The sources of heterogeneity in a parallel architecture are thus the degree of variability in the range of their processing element designs, and in the elements which are employed to transport data to and from them.

2.1.3 Programmatic Scalability

McCool's survey of scalable programming models [67] focuses on the conceptual mapping between architectural and programming model aspects. It makes an essential point by discriminating between processing and programming models, in order to separate issues of programmability and execution efficiency. Programming models are defined as a programmer's abstract view of software logic, whereas processing models are the associated cost considerations by which performance trade-offs can be evaluated. He identifies three central characteristics of a scalable programming model:

1. Simplicity
2. Expressiveness
3. Safety

Simplicity is the property of affording common programming tasks with little programming effort. *Expressiveness* is the property of allowing succinct statements of solutions within a problem domain. *Safety* refers to facilities which protect programmers from making common mistakes. It is interesting to note that the utility of these properties is not to provide efficient execution, but to restrict program complexity. Our discussion of programmatic scalability will adopt these three parameters as evaluation criteria for programming models, leaving efficiency to be considered as an aspect of program interaction with architectural parameters.

2.1.4 Programmatic Heterogeneity

Classifying heterogeneity exclusively with respect to programming models is prone to become a statement of subjective opinion. The term is often used either with respect to hardware only, or when considering systems which integrate hardware and software. This may stem from the fact that different programming models feature different concepts, making it difficult to compare them systematically. Moreover, programmers' mental model of the operations at their disposal is highly subjective, *e.g.* it is a well known problem that programs which are obvious to their author can be incomprehensible to another reader, making it difficult to reach any consensus on how many levels of abstraction it involves. Still, addressing the heterogeneity of programming has some merit, witnessed by how heterogeneous architectures inspire programmers to develop multi-model solutions [31, 85].

Although the number of abstractions or entities identified in a programming model is not a perfect map of every idea the model may present to a programmer, it enables structured reasoning on how many concepts are considered independent in its specification. The number of *available* abstractions is obviously not in direct relation to the number actually utilized in any given program. However, our discussion of programmatic heterogeneity will use it as a measure of the potential for variability in application behavior, in order to provide an ordered relation between different models.

2.2 Architectural Map

Figure 2.1 gives an overview of several kinds of systems, related to each other by the criteria identified in Sections 2.1.1 and 2.1.2. Since creating an exhaustive, fine-grained taxonomy of parallel systems is vulnerable to rapid technology changes, systems are grouped into approximate categories. Section 2.2.1 discusses many-core systems. Section 2.2.2 discusses systems augmented with special-purpose accelerators, Section 2.2.3 discusses distributed shared memory systems, Section 2.2.4 discusses distributed memory systems, and finally, Section 2.2.5 discusses computational grids.

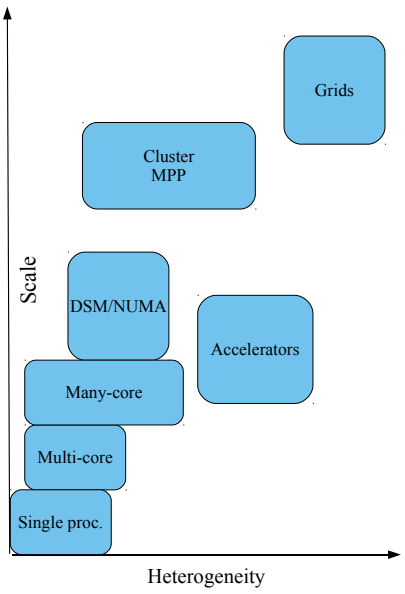

Figure 2.1: Map of architectural scalability and heterogeneity

2.2.1 Multi-core and Many-core Systems

Multi-core processors of autonomous units have yet to reach the same scale as large systems, but core counts are rapidly growing. Asanovic *et al.* [11] introduce the term *many-core* as an extension of this development, noting that the number of cores per chip can be expected to double with each silicon generation. Accordingly, we place the *multi-core* category in the low range of scalability, referring to the present generation of processors with relatively small numbers of processing cores. The computational cores of these processors are typically identical, but their overall performance becomes heterogeneous due to variable access cost and contention effects observable on a chip level [72].

Beyond mass-market processors, the literature describes systems which warrant discussion as many-core designs. The Niagara processor [56] provides 8 units capable of 4-way thread execution. While the cost of numerical operations render this design less interesting for scientific computing purposes, we note that its degree of heterogeneity is reflected in the attention devoted to thread scheduling. The Larrabee architecture [88] promised up to 48 processing units on a chip. The architectural description argues its general programmability, as cores are based on x86 designs. Brief treatment is given to applications in physics as well as video processing, but applied benchmarks primarily address gaming applications and graphics rendering. While the Larrabee architecture has been put aside, a more recent Intel press release [50] indicates that processors of similar on-chip parallelism and core design may target high-performance computing.

These systems are designed primarily for scalability, driven by the exploitation of chip area as replicated cores. As witnessed by the application sensitive design decisions made with Sun's Niagara and Intel's Larrabee, however, the balance of these designs is obviously biased towards restricting application dependent bottlenecks. Replicating cores mandates a uniform interface, so core internals must be encapsulated, corresponding to the principle of independent design. Achieving latency hiding still requires careful programming or specialized workloads.

Towards increased heterogeneity, Kumar *et al.* [59] point out that system-on-chip (SoC) designs apply increased transistor counts to make single-chip computing platforms from a mixture of integrated devices, but that these systems mostly allocate distinct tasks for the various subsystems. Deciding on the complexity of the replicated core in a multi-core processor is a trade-off not only because of variable application requirements, but also because of similar variability in the requirements between phases of a single application. Thus, general-purpose chip multiprocessors have both power and throughput advantages to gain from heterogeneous design. Amdahl's historic contribution [8] calls for improvements in parallel processing to be matched by a corresponding improvement in sequential processing. Woo and Lee [101] examine this with respect to many-core computing, comparing constant power budget configurations of a small number of large cores, a large number of small cores, and a large core coupled with a greater number of small ones. Their analytical model indicates that the latter combination is more energy efficient, indicating that chip-level heterogeneity may mitigate the effect of Amdahl's law. The Cell BE [42, 58] represents a commercially available heterogeneous on-chip multiprocessor with general purpose processing capabilities, consisting of a modified Power4 core, and eight *synergistic processing elements*, which are optimized for executing single-precision floating point operations using 128-bit wide vector instructions.

With tightly coupled cores, the heterogeneous features of the communications subsystem are limited in such systems. Although the heterogeneity of application-specific designs will be dependent on the nature of the application, both the Cell and the projections of Woo and Lee consider designs composed from at most two core designs. While the rapid growth in number of cores per die may lead to designs of more variable on-chip components, contemporary systems remain in the low range of architectural heterogeneity.

2.2.2 GPGPU and Special Purpose Accelerators

Improvements in scalability on the single system level invites application-specific accelerator designs which leverage the potential increase in core count to integrate large numbers of simplified cores on a chip. Such accelerators include recent generations of graphics processors featuring in the hundreds of reduced cores [78], with improvements in general programmability. Designs like the HC-1 [24] utilize FPGA units to let users specify application-specific core capabilities. McCool[67] surveys programming models with a view towards scaling into massive parallelism on chip using graphics processing units, arguing that these are the consumer processors which currently feature the greatest amount of explicit parallelism, and that general purpose processors are likely to follow their development.

Without assessing the accuracy of that prediction, the present generation of accelerated systems carries the distinction that the co-processors are designed quite differently from their host processors. While damaging to subsystem independence, this clearly caters to the principles of balance and design for scalability. Furthermore, modern graphics processors integrate scheduling hardware specialized to facilitate latency hiding by context switches. The manner in which these systems achieve scalability is tightly linked to their wider range of core designs, making them an excellent example of the trade-off between the two aspects.

2.2.3 Distributed Shared Memory Systems

Laudon and Lenoski [63] argue that the scalability of the design they describe is chiefly based on the modularity of its architecture. They also present latency measurements, and descriptions of data and process migration mechanisms to maintain low communication overheads. This work appears to address scalability by arguing that it is embedded in the design, as it includes considerations of how this single, modular architecture can be applied across a range of system sizes, providing a tailored entry price point for dedicated systems with known performance parameters. The first three scalability principles in Section 2.1.1 are evidently considered, and the requirements on interconnection technology are mentioned in a section which bounds total system size to 1024 processors. Two points about this work deserve particular attention, to illuminate later discussion. The first is that the mention of "ccNUMA" in the title acknowledges the significance of a heterogeneous system property, *i.e.* the cost of memory access. The second is that the distributed directory scheme which implements this nonuniform communication may inherently restrict the number of processors which can be added. In their discussion of cache coherency by distributed directories, Hennessy and Patterson[43] state that the amount of information required by a straightforward directory implementation is proportional to the product of the number of memory blocks and the number of processors, and that this becomes a significant overhead for processor counts around 200. Some suggestions are made regarding how this limitation can be reduced by restricting the information stored in the directory, such as tagging a memory block as relevant for a *group* of processors. In the general case, this would imply a hierarchical approach, which is likely to increase the heterogeneity of memory access costs.

Anderson *et al.*[9] describe the CRAY T3E, another distributed shared memory architecture which makes claims to scalability. This article devotes much attention to programming techniques, but communication facilities are also described, and the principle of latency hiding features prominently in the discussion of how remote memory access has extensive support for pipelining and prefetching. Large-scale installations are described as a torus topology of up to 2048 processors. Remote memory access is facilitated through a large number of registers which bypass local memory cache to request transfer from remote locations in a given range of the shared address space. Latency figures for remote load operations are given as a small range of values, indicating that global memory access is non-uniform on this architecture as well, even though effort is made to describe how this effect can be masked by software.

Common to these architectures is that the operating frequency of their processors is relatively low compared to contemporary units. According to Lusk and Chan, "the fastest machines now virtually all consist of multi-core nodes connected by a high speed network" [66], which shows that the significance of memory access cost has grown since the design of these systems. The limits of distributed shared memory architecture scalability are still of of interest, witnessed by the existence of projects like Blue Waters [34], but as such systems are exceptional cases, and exceedingly challenging to construct, we consider distributed shared memory platforms in the low thousands of processing units here.

Distributed shared memory systems are not commonly discussed in the context of heterogeneity, although considerable attention is devoted to the non-uniform cost of memory access. Performance-wise, heterogeneity is limited to similar nonuniform access costs of hierarchical memory as with multi-core systems, but with a greater number of stages leading to greater variability. With communication being the only source of heterogeneity, this class is placed in the low end of the heterogeneity spectrum.

2.2.4 Distributed Memory Systems

The present generation of supercomputers is dominated by architectures featuring distributed memory and supporting the message-passing paradigm of software design. The June 2012 top 500 supercomputer list [94] sorted by architecture share shows that compute clusters and MPP architectures together compose the entire list.

Given the common creation of compute clusters from components-off-the-shelf (COTS), architectural descriptions tend to be scarce. The architecture of the cluster as such is of little academic interest beyond the description of its components. Nevertheless, some constructions like the PACS-CS [21] are documented, due to a measure of novelty in board-level construction and interconnect design. This design aims to leverage the cost-efficiency of COTS while addressing shortcomings of conventional clusters by introducing tailored solutions to improve bisection bandwidth and sustained performance figures. The use of COTS requires independent component designs, and consideration of the growing network bandwidth matches the principles of balance and design for scalability, showing that this system observes the first 3 principles. It is built to a scale of 2560 processing cores.

Although similar to clusters in the sense of supporting parallelism as the joint operation of compute units without shared resources, MPP systems feature more integrated designs between the computational units and the interconnection network. This caters more to the principles of balance and design for scalability than to independence. A prime example from this system class is the Blue Gene/L architecture [6], which features dual-processor computational nodes in a torus topology. One processor is mainly intended to cater to communication operations, but can also be explicitly programmed to perform computation. The processors operate at a relatively low frequency for the sake of power efficiency. With an appropriate interconnection network, the design can scale to 65536 computational nodes.

A more recent example is Roadrunner [13], which was the first system to achieve sustained petaflop performance with standard benchmarks. The composition of this platform breaks

down into 17 *compute units*, which are each in turn composed from 180 triblade *compute nodes*, consisting of one blade of two dual-core Opteron processors and two blades of two PowerXCell 8i processors, a version of the Cell BE with extended support for double-precision floating point operations. The full system thus contains a mixture of processing elements on the order of tens of thousands in number, with a variety of interconnection technologies on the various levels of locality. Quoting Barker *et al.* [13], "An implication of Roadrunner's deep communication hierarchy [...] is that the performance of a hybrid application is critically dependent upon the application's ability to exploit spatial and temporal locality". While the architecture's hierarchical decomposition indicates a modular design built for scalability, this limitation suggests an imbalance in the relative costs of computation and communication, as well as challenges in latency hiding.

The heterogeneity of distributed memory systems spans a great range in our classification, reflecting that it is a function both of the variability in the interconnection between component subsystems, and the heterogeneity of their internal architecture.

At the low end of the range, a common class of cluster systems is composed from multicore processors, or even multiple such processors interconnected on multiprocessor boards, creating the same amount of heterogeneity as distributed shared memory systems, with one or more additional stages of interconnection. In the middle of the range we find systems which are classified as heterogeneous cluster systems due to being composed of component systems of variable processing capacity, as well as a nonuniform interconnection network [77]. At the high end, we find systems which differ in the mixture of processing elements within each component subsystem, but to a lesser extent in the variability of these subsystems [13].

2.2.5 Grids

At the far end of the architectural scalability spectrum, we find *computational grids*. These grids aim to interconnect computational resources on an abstraction level which transcends single systems, in order to provide computation as a transparent service independent of the site of program execution.

According to Foster and Kesselman [35], the scale of a computational grid should be considered along with its intended application, *e.g.* grids for distributed supercomputing are likely to be differently dimensioned compared to throughput-oriented grids which aim to increase the utilization of otherwise idle computers.

Both of these categories still contain some of the world's largest computational resources. In the case of distributed supercomputing, the Enabling Grid for E-sciencE grid infrastructure provides access to a number of processors in the hundreds-of-thousands order of magnitude [104]. The Folding@Home network is a loosely coupled system for harvesting spare computational power from the idle time of generic desktop computers. At the time of writing, the number of active donor processors also number in the hundreds of thousands [60].

While the largest grid applications are massively parallel, they also bear the distinguishing

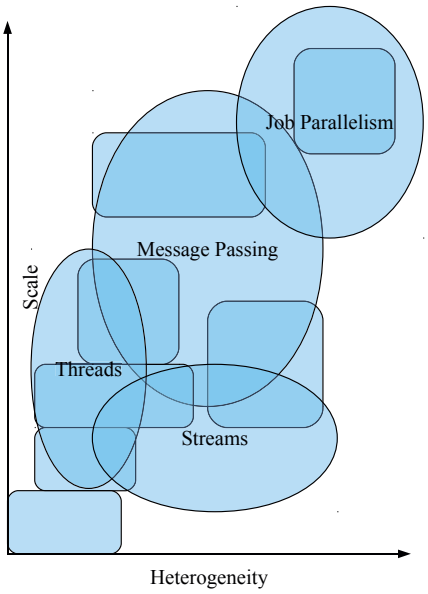

Figure 2.2: Map of programming model scalability and heterogeneity

characteristic that they are mostly programmed using job-level parallelism, reducing the programmatic difficulty of exploiting the available resources to a pure scheduling problem. An exception to this can be found in the work of Allen *et al.* [5], which details the application of grid-enabled communication libraries to obtain performance measurements on a combination of one 1024-processor, one 256-processor and two 128-processor systems. Although this configuration is restricted to a resource consumption two orders of magnitude smaller than the greater challenges tackled in a grid context, it is interesting with respect to the principles of scalability to note that the experiences collected pay meticulous attention to the issues of load balance and masking communication cost in order to achieve the reported performance.

As computational grids cannot be restricted to any particular category of component subsystems or interconnection technology, it is difficult to discuss them in terms of their architectural properties. In light of the above examples, however, it is fitting to categorize them as extreme cases of both scalability and heterogeneity.

2.3 Programming Model Map

Based on the criteria identified in Sections 2.1.3 and 2.1.4, a high-level taxonomy of programming models can be identified in a manner similar to that of architectures. A division of programming models into *thread*, *stream*, *message passing* and *job parallelism* cate-

gories is shown in Figure 2.2, superimposed on the architectural map from Figure 2.1. This section argues their placement, beginning with threaded models in Section 2.3.1. The discussion then proceeds in order of scalability, covering stream parallelism in Section 2.3.2, and message passing in Section 2.3.3, before discussing the corner cases of hybrid models and job level parallelism in Section 2.3.4.

2.3.1 Thread Parallelism

Quoting Akhter and Roberts [3], "Scalability is the challenge of making efficient use of a larger number of threads when software is run on more-capable systems". This definition is too thread-centric to generically capture scalability, even within high-performance programming, but it reflects a common perception of the challenges which come with many-core general processors. A variety of threading models are available, some are language neutral such as POSIX threads [3] or OpenMP [100], while others are related to specific environments such as *e.g.* Java or Perl. All facilitate multiple concurrent instruction streams using a shared address space, restricting threads to shared memory systems.

Explicit threading provides detailed control of fine-grained synchronization. Common model features are either mutual exclusion primitives in the form of locks and semaphores, or language extensions for marking critical sections. While utilizing such features makes programming simpler than using explicit locking, this type of synchronization is still contains a number of pitfalls which lead to common programming mistakes. In this sense, explicit threading is neither simple nor safe. Expressiveness is naturally tied to the facilities of the language featuring the threading model, but the management of synchronization leads to at least some programming unrelated to the problem domain.

OpenMP provides a simple set of directives which can be applied to imperative programming constructs such as loops and sequential blocks. These allow the programmer to guarantee that sections are free of dependencies, automating thread management and identification of synchronization requirements. This improves simplicity and safety of threading with little interference in expressiveness. However, hiding the cost of thread management from the programmer conceals performance parameters which are critical to scalability.

The class of architectures supporting threaded programs spans a wide range of scales, encompassing installations from single-core *hyperthreading* processors with limited support for concurrency, to large systems with distributed shared memory, which may execute thousands of threads. This large footprint in the architectural landscape makes threading a convenient model for inclusion in hybrid programming approaches, such as those described by Rabenseifner [85] and Barker *et al.* [13].

Thread programming is a lightweight approach to parallelism, as threads have only a small local workspace within a shared program state. The shared program state provides implicit communication, as any thread can read values modified by another, requiring concurrent execution to control cases where it results in nondeterminism. Thus, thread control can be provided by a small set of constructs, placing it low in the heterogeneity spectrum.

The POSIX thread interface focuses on synchronization primitives, providing mutual exclusion by explicit locking of memory locations, signal delivery between threads, and waiting for thread completion. Collective operations include barrier synchronization primitives and broadcast signals, but higher-order operations require explicit programming. OpenMP also provides mutual exclusion, but at a higher level of abstraction, as critical sections can be marked in the code without explicit locking procedures. Like POSIX threads, OpenMP also provides signalling and barrier synchronization, and collective operations are extended to include reductions. The task of spawning threads is abstracted almost completely, with brief mnemonics to indicate independent program sections. Finally, some scheduling parameters can be controlled, and a high-resolution timer is provided for profiling. This raises level of abstraction relative to POSIX thread programs, but also complicates cost analysis, *e.g.* by introducing barriers which are implicit in the code.

2.3.2 Stream Parallelism

Streaming models are connected to systems which integrate accelerators and commodity processors. Large computational demands are divided into independent data streams, and the computation is expressed as a *kernel*, which is a small algorithm for processing a segment at the head of a stream. This choice of program unit is similar to a thread, by representing a light-weight invocation of a function on a small amount of data. However, kernels have restricted communication and synchronization facilities. The performance gap between memory access and computation requires a level of *numerical intensity*, *i.e.*, each element fetched must undergo a number of computational operations in order to amortize the cost of fetching it.

Extensions of traditional paradigms require that functions can address program global state: if the programming model does not express this, it requires automatic detection of the data set a function acts on. Detecting this requires complex data-flow analysis [2], and the exploitable benefit must be conservative, even when successful. Without guarantees on locality of reference, program translation leads to bursts of read-modify-write sequences [67], which is bad for pipelining and numerical intensity.

Stream processing explicitly recognizes that a kernel consumes a restricted number of data elements. This comes at the expense of some amount of programmability, leading to streams often being embedded in conventional languages. An early example of this approach can be found in the software system of the Imagine stream processor[52], which separates the *StreamC* and *KernelC* extensions to the C programming language to take advantage of a dedicated processor architecture. Approaches which leverage graphics processors for general purpose computation are presently undergoing rapid development. Starting from a library approach using OpenGL, Adinetz [1] summarizes how its inconvenience for general computations led to a wealth of higher-level approaches, such as CUDA [78], Cg and Sh [67]. Developments like the Imagine architecture have led to similar ideas also in the embedded applications space [86]. Less GPU-centric approaches such as the programming toolchain of the Cell Broadband Engine [58] and OpenCL [41] also exist.

The specific designs of such approaches change quickly, as seen from the 2006 press re-

lease regarding the AMD Close-to-Metal technology [7] and its subsequent abandonment in 2008 [97]. For the purposes our discussion, the defining characteristic of all these approaches is the emphasis on computational kernels, and that with the exception of OpenCL, they address problems of a scale which is solvable on a small number of processing units each featuring a high number of parallel components.

Programming models which couple existing languages with architecture-specific extensions go far with respect to expressiveness, assuming their use within appropriate application domains. Simplicity and safety are improved by the move from adapting problem descriptions for graphics pipelines to more general tools. Still, programming these models still requires special training even for trained programmers, indicating that performance concerns still receive more attention than simplicity and safety so far.

The necessity of expressing statements about streams makes these models more heterogeneous than threading models. As an example, NVIDIAs CUDA model [78] features not only synchronization and the stream abstraction, but also has facilities for event handling, management of various kinds of memory corresponding to the type of processor, and distinguishes between graphics devices and host processors. Taken to an extreme, the two languages of the Imagine system [52] separates statements regarding the computation of a kernel and the scheduling of streams unto execution devices into two disjoint language extensions. Although this approach is less common in other models, it provides a poignant example of the need for varied constructs in a stream programming context.

2.3.3 Message Passing

Message passing found utility before the emergence of true concurrency in commodity computing: computation as an exchange of messages is central to the SmallTalk and Erlang languages, amongst others. According to Kay [53], it was motivated by a desire to "find a better module scheme for complex systems[...]". This not only for improves simplicity and expressiveness, but also fits the modularity principle of scalable design.

Erlang originates as a control language for distributed systems in telecommunications, where operating a large number of devices is business critical. Armstrong summarizes scalability as the requirement that “adding a new machine should be a simple operation that does not require large changes to the application architecture” [10], emphasizing encapsulation over performance. A different perspective is offered by Gropp, Lusk and Skjellum in their description of MPI: “Scalability analysis is the estimation of the computation and communication requirements of a particular problem and the mathematical study of how these requirements change as the problem size and/or number of processes changes” [40].

The advantage of message-passing models is that they specify both the locality and size of shared data structures, admitting execution on distributed memory platforms. This mirrors their application on large machines, as shared memory architectures of similar scale are difficult to construct. Distributed memory models are also suited for large-scale execution because the cost of communication is explicitly acknowledged in code, simplifying performance analysis on growing interconnects. However, message passing requires nontrivial code just to orchestrate the execution of the program, adversely affecting expressiveness.

Simplicity and safety are improved by hiding the management of both message buffer locations and transfer from the programmer.

Message passing models are very flexible with respect to their execution platform, both in terms of heterogeneous communication and computation facilities. This is witnessed by the popularity of using messages as an overarching communication mechanism in hybrid models, coupling it with either thread level parallelism [26, 85], or dedicated accelerator approaches [33, 84]. This flexibility requires message passing schemes to adapt to a wide range of parameters, as different characteristics dominate performance on different platforms. This is reflected in MPI, which supports abstractions for point-to-point messaging using different modes, collective operations from barriers to total exchanges, a derived datatype system, grouping of processes and process topologies. These abstractions are defined to be implementable in terms of a few basic operations, but subsets of the provided functionality can be customized for particular systems. Another point is that the abstractions are independent, creating a large number of combinations available to the programmer. For these reasons, message-passing is classified as most heterogeneous among our model categories.

2.3.4 Hybrid Models and Job-level Parallelism

The cases of hybrid programming [85] and coarse-grained parallelism in the form of job scheduling, are difficult to classify as programming models on their own, as they consist of coupled models and extremely restricted models, respectively.

Hybrid programming models will obviously offer the combined variety of features present in all component models, thus increasing the level of heterogeneity. Placing all possible combinations of the three surveyed categories of models in an order would add little to their discussion, suffice to say that our notion of heterogeneity matches the use of hybrid models to program hybrid (*i.e.* heterogeneous) architectures.

Job control languages and schedulers leveraged to exploit job-level parallelism are not usually considered programming tools, due to the fact that their primary task is to define a mappings between sets of programs and resources. Language features, if at all present, are mostly restricted to text substitution, as well as simple conditional and iteration constructs, rendering them extremely inconvenient for applications beyond describing resource requirements and sequencing the execution other programs.

In spite of such solutions being almost devoid of common programming model constructs, they are mentioned in this discussion because the extremely coarse-grained abstractions they support make them suitable for arranging parallel execution of independent tasks on extremely large scales. This means that it is of little interest to classify the features of the languages themselves in a programming model context, but it is necessary to classify them with respect to scalability for the simple reason that their applications have historically become future targets for detailed programming models. The process images which form the executing environment for threads as well as a unit of parallel computation in modern operating systems, has its origins in the scheduling of multiprogrammed workloads in third-generation operating systems [92]. Over a shorter time span, Bode *et al.* [20]

measured job scheduling throughput on a 64-node cluster platform in 2000, while a 2008 work by Scogland *et al.* [87] discusses thread scheduling on multicore processors, naming vendors with specific processor designs which will feature 64-way threading on a chip.

It is interesting to note that grid scheduling is subject to task-level scheduling [30], which may suggest that present throughput computing tasks, such as executing masses of *e.g.* XRSL [80] batch jobs (which contain no program logic), foreshadow the scale of operation which will require programming model support in the near future.

2.4 Research Context

Effective performance modeling must provide a bridge between the programmatic and architectural aspects of scale and heterogeneity, and the great variations in parallel systems and programming models create a challenging trade-off between generality and accuracy. Our study focuses on clustered, distributed memory systems of shared memory subsystems. This choice is made partly to cover a reasonable range of systems while maintaining portability, so that results will be comparable, and also because such clusters are in widespread use, due to their simple and relatively inexpensive construction.

Several approaches to modeling this class of systems exist already, ranging from purely theoretical approaches, through performance models which attach to programming constructs, to performance studies of particular applications.

Proposing a general theory of modeling and simulation, Zeigler *et al.* [103] start from distributed systems, and classify their components as systems specified by discrete time (DTSS) and differential equations (DESS). Their central proposal is to describe heterogeneous systems in terms of component subsystems which are *closed under composition*. This means that a model of their composition into a greater system can be derived from the component subsystems without alterations to the terms of the model, thereby hiding subsystem detail. Hierarchical models derived in this manner are very robust to the integration of further components, but the framework for model construction is driven by encapsulating subsystems using *coordinator* facilities which may represent substantial bottlenecks when realized in software.

The family of PRAM models [23] provide a simple, abstract parallel machine for deriving asymptotic complexity bounds on parallel algorithms. Although this does not provide realistic models of actual machines, it does provide SPMD style programming, and admits analysis of algorithmic aspects, serving as a starting point for efforts to add programmability and realism [54].

Systems which require explicit communication primitives invite models which divide cost into communication and computation. The Hockney [47] model partitions communication cost into latency and message size as a benchmark of the executing platform, leaving the application of parameter values in software implicit. The LogP model of Culler *et al.* [27] discriminates between the latency of a message, its initialization cost, and the minimal gap between successive messages in order to approach empirical validation. Alexandrov

et al. [4] propose the LogGP model, adding a linear cost increase per transmitted byte, increasing model emphasis on message lengths. Bosque *et al.* [22] further extend it to the HLogGP model to account for heterogeneity in the parallel platform. Valiant's BSP model [95] unifies the measurement of platform parameters with program analysis, providing a coherent set of parameters to describe their interaction. This has been extended both in refinements of the theoretical side [38, 93, 96] and practical efforts to realize it, from realizing the programming primitives in library form [45, 46, 55, 102], to describing transformations of programs into other programming models [19, 90]. Bilardi *et al.* [18] compare the LogP and BSP models, establishing that they can mutually simulate one another with small overheads in asymptotic terms, and noting that BSP provides a more convenient programming abstraction.

The convenience of reasoning about the BSP programming abstraction has invited several proposals for model extensions. The E-BSP model [51] refines the original modeling of communication phases as *h-relations*, adding considerations of instances where unbalanced communication is favorable. It also adds a notion of network proximity, allowing the impact of network topologies such as linear arrays and meshes to be assessed. More recent efforts have focused on hierarchical decomposition, which naturally captures cost variation due to locality in fat tree topologies, as well as those due to the hierarchical memory systems internal to the multiprocessor subsystems which they are often composed from. Thorough justifications for this are given both by Bilardi *et. al.* [17], and Valiant [96].

The influence of these works is visible in our approach, as its essential purpose is coupling performance models to the constructs of an associated programming model. To this end we adopt the BSP view of program execution with only the minor adaption of introducing overlap to the alternating phases of computation and communication. We also replace the classical approach of explicitly deriving performance predictions from small parameter sets, with one of programmatically producing estimates from comparatively large sets. In particular, pairwise performance parameter values are extended into matrices of all pairs, which is identified by Lastovetsky *et al.* [61] as a straightforward method to account for heterogeneity. As a consequence of this, processor locality within a system topology is treated as an implicit property of the platform parameters, rather than explicitly acknowledged through revising the terms with which algorithms are specified.

A common element to these approaches is the acknowledgement that structural information about the computing platform is essential to performance. This presents a trade-off, as the accuracy obtainable by acknowledging it in programs is detrimental to their portability, as is noted by Bilardi *et al.* [16]. Although our work is restricted to observing subsystem features as heterogeneous properties in a single context, we note that it is complementary to a hierarchical decomposition within the granularity of its subsystems. A fully integrated combination featuring a distinction between local and global subsystem behavior is beyond the scope of this thesis, but part of the work produced a paper which demonstrates that our approach can benefit algorithms which are constructed to be aware of hierarchical platform structure [75]. It appears in extended form as Chapter 7 of this thesis.

Chapter 3

Modeling Framework

The tension between scalability and heterogeneity makes it challenging to select appropriate abstractions which capture architecture and algorithm interactions. Attaching performance metrics to programming model features must conceal details of their implementation, but accurate cost estimation depend on them. This chapter identifies architectural and programmatic elements which combine to permit both abstract reasoning and specific predictions about hardware/software interactions.

Section 3.1 gives an example of the modeling approach commonly associated with BSP. Section 3.2 outlines our model revisions, and compares them to Section 3.1. Subsequent sections relate model extensions to the terms of Equation 1.1 in order: Section 3.3 discusses computational requirements and resources, Section 3.4 addresses communication, and Section 3.5 describes overlap.

3.1 Original BSP Performance Model

To highlight our model changes, it is useful to examine how BSP originally models the interaction of a program and platform. As Valiant's paper [95] states the model very generally, we will instead follow the notation of Bisseling [19], who accompanies parameter descriptions with corresponding benchmark code.

Performance is captured by a set of 4 scalar parameters: p is the level of parallelism, h represents the communication requirement in a step, g is the throughput of the interconnect, and l is the *synchronization cost*. The properties h and g are associated with *h-relations*, which are stages of communication where each pair of processors exchange a message of size at most h. This captures an upper bound on the cost of all communication committed during a superstep, without loss of generality. Communication semantics do not require effects to be visible until after synchronization, so a collective, total exchange always suffices to transmit buffered communication between any pair of processes. The conceptual router which models this is fully connected, but may be implemented using networks of

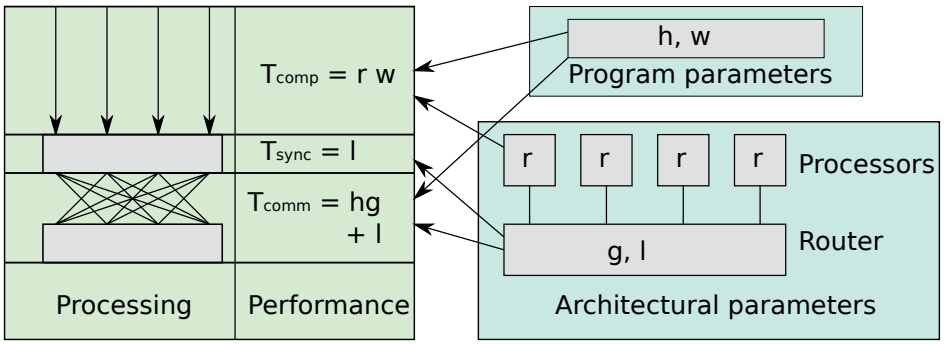

Figure 3.1: Relationship between component BSP models
The figure gives a schematic illustration of how architectural performance parameters are coupled to program requirements, producing a performance model directly related to supersteps in the processing model.

lower connectivity. The performance model assumes that the interconnect will operate close to its capacity during communication steps, and the processing model accordingly gathers all communication for collective transmission.

Having values for p, h, g, and l, the cost of executing a program is expressed in Equations 3.1, 3.2 and 3.3.

$$h = max\{h_s, h_r\} \tag{3.1}$$

$$T_{comm}(h) = hg + l \tag{3.2}$$

$$T_{comp}(w) = w + l \tag{3.3}$$

In Equation 3.1, h_s and h_r represent the maximum number of machine words sent and received by any processor. In Equation 3.3, w represents the maximal amount of work assigned to any processor in a step, measured in floating point operations (*flop*). The value of l is the cost of establishing that all processors have arrived at the next stage. While this value may be smaller for computation than for communication steps, it is considered the same for the sake of simplicity [19].

Bisseling's text [19] focuses on the BSPEdupack software, which it presents with full source program listings. This makes a natural entry point for practical testing, given an implementation of the BSPlib standard it employs. A suitable implementation is *BSPonMPI* [91], as it can utilize distributed memory architectures by using MPI for data transport.

BSPEdupack contains the benchmark *bspbench*, and an example computation *bspinprod* which we use as a preliminary test. The *bspbench* program obtains measurements of the 3 machine parameters for a given level of parallelism. It first measures computational rate by timing a growing series of L1 BLAS DAXPY [64] computations on problem sizes of up to 1024 elements, finding the linear regression line of least square errors, and estimating computation rate by the gradient term. This number is stated in terms of $\frac{flop}{s}$, and relates other values to time as *flop* equivalents. Router throughput and latency are found as the

Table 3.1: BSPBench parameter values for 8-way 2x4 core cluster

P	r	g	l
08	991.695	105.4	30575.7
16	984.713	373.6	631365.8
24	972.553	369.7	1450059.5
32	961.875	89.5	1771331.3
40	968.230	67.5	2500077.3
48	958.886	228.6	3026802.1
56	935.523	521.2	3419705.8
64	944.005	1326.5	3972859.4

gradient and intercept of a similar regression line, obtained from growing *h-relations* from 0 through 255. Double-precision floating point numbers are considered machine words.

The *bspinprod* program computes the inner product of two vectors, in two computation steps and one communication step. Two vectors are allocated in a distributed fashion, assigning N values to p processors for a local problem size of $n = \frac{N}{p}$. Results are reported in strong scaling mode, using $N = 10^8$ elements while growing p. The first computation step finds p local sums of products, for a total workload given in Equation 3.4.

$$comp_1 = \frac{N}{p} \cdot 2flop \tag{3.4}$$

The first communication step scatters the local sum to all participating processes. With single scalar sums, this step is a 1-relation, reducing Equation 3.2 to Equation 3.5.

$$comm = (1 \cdot g + l)flop \tag{3.5}$$

The second computation step accumulates local sums on all processors, yielding Equation 3.6, and the total cost in Equation 3.7.

$$comp_2 = pflop \tag{3.6}$$

$$T_{total} = \frac{(\frac{N}{p} \cdot 2 + l + g + l + p)[flop]}{r[\frac{flop}{s}]} \tag{3.7}$$

The *bspinprod* program was chosen because it applies the same numerical kernel as the benchmark program. This is done to create a comparable computational rate without extensions for heterogeneity. The only modification made to the programs is that the measured cost of *bspinprod* is a median value of 100 repetitions, to reduce warmup effects and background noise.

Table 3.1 and Figure 3.1 report experimentally obtained and theoretically predicted values from experiments run on node multiples of 8 processor cores, ranging from 1 through 8 nodes. The most notable feature of Figure 3.1 is that the theoretical estimates of execution time deviate from actual execution time by 5 orders of magnitude. A second point is

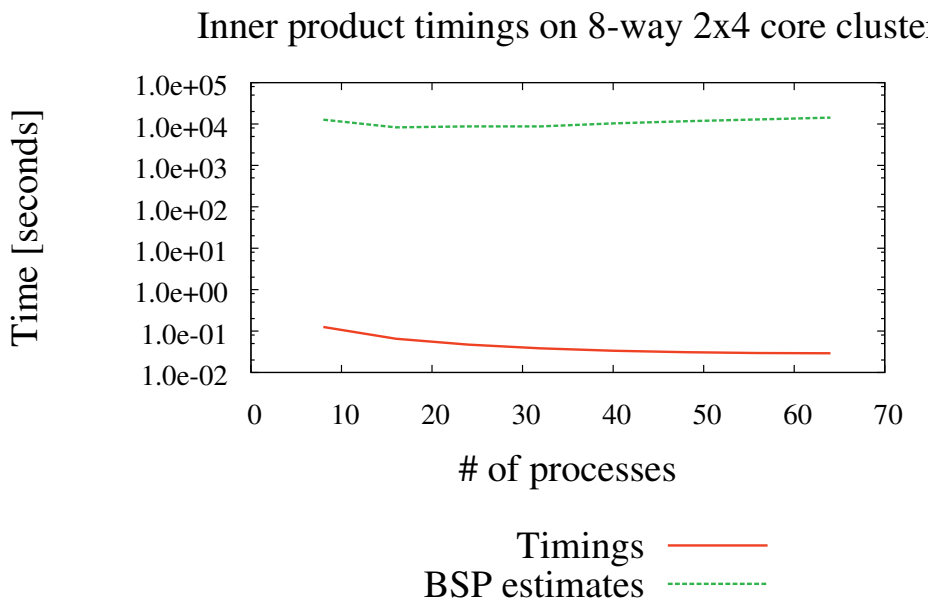

Figure 3.2: Inner product comparison on 8-way 2x4-core cluster
The figure compares values obtained using bspedupack *versions of the* bspbench *and* bspinprod *programs, with the* BSPonMPI *implementation. Predictions use Equations 3.1, 3.2 and 3.3 and values from Table 3.1, in conjunction with the program requirements in Equation 3.7. Note the logarithmic time scale.*

that the prediction contains a minimum, whereas the measured results are asymptotic, as Amdahl's law would predict for a strong scaling experiment. A number of reasons for these deviations are important to identify before attempting to improve accuracy. They are to some extent visible from the collected data already at this stage.

Table 3.1 matches a reasonable expectation that computation rate r is constant, and not subject to changes of platform scale. It reflects how individual processor performance is independent of their number, but conceals the assumptions that all computations are equivalent, and that performance varies linearly with a single rate. Nonlinearities in this parameter stem both from heterogeneous floating point capabilities among processors, and variations in performance due to interactions with the memory hierarchy of a modern processor. Our heterogeneous computational rate model must address both issues.

The latency parameter l spans orders of magnitude already at modest scales, reflecting the heterogeneity of the interconnect topology. The worst case latency of a multi-layer interconnect expressed in terms of the computation rate on a small problem creates an unrealistic, dominant term in the cost function. Furthermore, assuming that termination of a synchronous step costs the same as initiating communications is visibly incorrect by orders of magnitude. Adapting the latency measure to our revised processing model, it must be treated on a per-message basis. Doing so decouples it from the global network diameter, and attaches it to topological distance. This difference is significant, as seen from the contrast between numbers attained on a single node and on the entire machine.

The throughput parameter g suggests that the cost of message transmission is tied only to the data volume committed by the application. This assumption comes from the premise that all communication follows the same pattern at every communication step, which is true for networks that realize an h-relation regardless of its utilization. The communication pattern of the present experiment does not highlight it, but the objective of decoupling communication from synchronization cost requires bandwidth measurements to be associated with the locality of senders and receivers in a heterogeneous interconnect topology.

3.2 Changes to Architectural and Processing Models

The purposes of revising the architectural and processing model aspects of BSP are to refine their detail and accuracy, and to admit derivation of the overlap term in Equation 1.1. Figure 3.2 shows an overview of how architectural and program features combine with the processing model, to produce a performance model in a similar manner to Figure 3.1. These extensions cause a notable growth in the number of parameter values, requiring details of the associated performance model equations to be omitted from the figure. The level of abstraction conflicts with the level of detail, in a trade-off which warrants discussion.

A significant point is that BSP refrains from exposing detailed architectural structure, as it aims to provide a conceptual bridge between hardware and software design. As an example, the fully connected router abstraction subsumes any interconnect topology, and conceals idiosyncrasies in the performance parameters of the abstract mechanism. This is

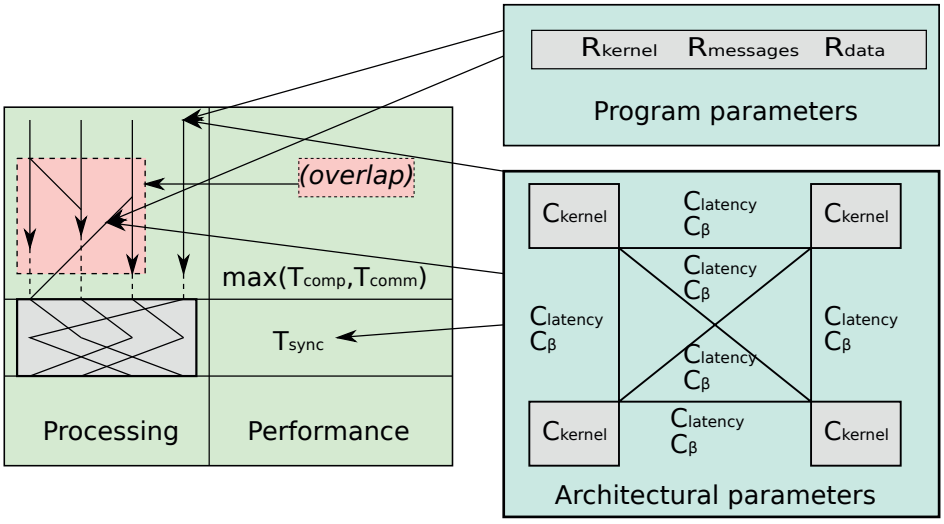

Figure 3.3: Relationship between revised models

The figure gives a schematic illustration of the correspondences between our revised models of architecture, program, processing and performance. The greatest difference is that the processing model fuses computation and communication phases, which makes it necessary to combine program requirements (R) and architectural costs (C) into a unified cost function. Architectural parameters are extended both with respect to computation and communication: computation rate is extended to be parametric in terms of the applied kernel*, while communications are modeled on a per-message basis using parameters inherent to individual edges in a fully connected graph. Finally, synchronization is considered as a specialized application of general communication, utilizing the same per-edge parameters as program messages.*

a desirable property, and Figure 3.2 retains the assumption of full connectivity, although increasing the number of parameters associated with the abstract network.

BSP semantics require that results communicated during one stage are not in use at their destination before synchronization. Both Hill and Skillicorn [44, 45] and Bisseling [19] compare overlapped communication and computation to the effect of postponed communication, and conclude that postponing is advantageous. Their arguments are that postponing communication reduces total latency because messages can be packed, and that it provides optimization possibilities for the implementation of the total exchange in communication phases. On the other hand, Goldman *et al.* [37] refer to this message exchange problem, and note that it is NP-complete for arbitrary combinations of message sizes on a uniform-cost fully connected network. With communication time as the product of message size and transfer capacity, the difficulty of minimizing the weighted sum of a message pattern also applies to uniform message sizes on a heterogeneous network. The relative merits of immediate and postponed transmission depend both on system characteristics and the effectiveness of a heuristic for the message exchange problem. Merging computation steps with simultaneous communication as in Figure 3.2 avoids the complexity of the optimization problem, at the expense of introducing multiple, but potentially maskable latencies in the cost function. Our reason for selecting the approach complementary to BSP convention, is the importance of exploring the spectrum of techniques available to reduce the growing impact of communication cost [11, 14, 43, 82].

Processing elements can be abstracted similarly to the router by acknowledging *virtual* processors, *i.e.*, decoupling the number of physical units from a program's notion of parallel work. This distinction is crucial to studies of *optimal simulation* of BSP algorithms [48, 93, 95, 96], which assume that committing some ideal amount of excess parallel work (captured in a *parallel slack* parameter) can be scheduled to mask communication latency, and raise the level of physical processor utilization. Applications of this principle are found in domains where exposed parallelism is abundant, and the cost of context switching is comparatively low. Examples include throughput optimizations for threaded server applications [62], hardware-scheduled threads available in general-purpose GPU programming [28, 67, 78], and the *task* construct of OpenMP [12].

We will consider one-to-one mappings of program parallelism to physical units only, for three reasons. First, it isolates the effect of explicit latency masking, for purposes of validation. Second, experiments are conducted on distributed-memory architectures, to utilize their scalability. This environment imposes severe technical obstacles and great cost on arbitrary context switches. Finally, designing an appropriate scheduling mechanism to work with a detailed cost model would require the model to be developed ahead of time. Investigating such an approach is interesting, but it is beyond the scope of this thesis.

To address the accuracy concerns raised in Section 3.1, the model extensions shown in Figure 3.2 must account for heterogeneous computation rates, communication latencies, communication bandwidth, and synchronization cost. In the following sections, the scalar parameters of the BSP performance model are replaced with matrices of parameters, containing the ranges of parameter values applicable to various subsystems. The overlap term is derived from this basic approach.

3.3 Heterogeneous Computation

The performance figures of heterogeneous devices introduce the concern that the details of individual subsystems may be most accurately captured in metrics which do not permit straightforward combination. One example can be seen in the shortcomings of equating time and floating point operations, as in Section 3.1. Here, the variability of communication features span different orders of magnitude from the computation rate, demanding a potentially unattainable accuracy of it. Other synthetic benchmarking practices also show this: the LINPACK benchmark [29] estimates operation throughput of floating point units, while SPECINT [83] estimates integer unit performance, and their inherent differences makes it meaningless to reduce them to a single metric of processor performance. These show heterogeneity not only due to the processor design, but also the different locality properties of programs which stress different units. Moreover, as they test peak performance using specialized workloads, inferring application performance strongly depends on the application resembling the benchmark. Combining both metrics might, however, give an accurate model of applications which work in stages, *e.g.*, a stage of dense matrix operations followed by a stage of data compression.

Starting from the basic assumption that operations are only comparable in their required execution time, the original T_{comp} term expresses the running time of a program as the sum of its operations, weighting them uniformly as one *flop*. Remaining with the example of vector products, the DAXPY numerical kernel from the BLAS package [64] is similar to the inner product benchmark, but with a widely used, more generic interface. Introducing different weights to each operation, the kernel for n elements

```
for ( i = 0 to n-1 )
  { y[i] = y[i] + a[i] * x[i] }
```

results in a cost function

$$\sum_{0}^{n-1} C(=) + \sum_{0}^{n-1} C(*) + \sum_{0}^{n-1} C(+) = \sum_{0}^{n-1} (C(=) + C(*) + C(+)) =$$

$$= n(C(=) + C(*) + C(+)) = t \tag{3.8}$$

with $C(op)$ in Equation 3.8 denoting the individual operation cost. The trivial conclusion is a cost of n multiplications, additions, and assignments.

The heterogeneity of modern architectures already challenges the treatment of basic operations as constant terms. The performance impact of hierarchical memory can, *e.g.*, make the cost of the first assignment much larger than that of the second. For the moment, we overlook this issue and write the weighted sum as the inner product of a requirement vector and a cost vector:

$$\vec{r} \cdot \vec{c} = \begin{bmatrix} n \\ n \\ n \end{bmatrix} \cdot \begin{bmatrix} C(=) \\ C(+) \\ C(*) \end{bmatrix} = t \tag{3.9}$$

This extends naturally to an SPMD [100] parallel program which executes the same logic on two processes, by encoding the requirements as a $2 \times |\vec{c}|$ matrix:

$$R \cdot \vec{c} = \begin{bmatrix} n & n & n \\ n & n & n \end{bmatrix} \cdot \begin{bmatrix} C(=) \\ C(+) \\ C(*) \end{bmatrix} =$$

$$= \begin{bmatrix} n(C(=) + C(+) + C(*)) \\ n(C(=) + C(+) + C(*)) \end{bmatrix} = \vec{t} \tag{3.10}$$

Assuming homogeneous conditions for both processes, this gives equal entries in $\vec{t}$. Analyzing the contents of a computational superstep, this system of 2 simple equations predicts that short of the synchronization overhead, $\vec{t}$ captures the individual costs of both processors, as well as a measure of the heterogeneity of computational cost within the superstep.

If the program computes the DAXPY kernel on one processor, and a difference such as

```
for ( i = 0 to n-1 )
   { y[i] = y[i] - x[i] }
```

on the other, the system works out to

$$R \cdot \vec{c} = \begin{bmatrix} n & n & 0 & n \\ n & 0 & n & 0 \end{bmatrix} \cdot \begin{bmatrix} C(=) \\ C(+) \\ C(-) \\ C(*) \end{bmatrix} =$$

$$= \begin{bmatrix} n(C(=) + C(+) + C(*)) \\ n(C(=) + C(-)) \end{bmatrix} = \vec{t} \tag{3.11}$$

and the inequality of the elements in $\vec{t}$ give a measure of computational load imbalance. With supersteps ending in synchronization, this exposes the magnitude of any mismatch in the computational requirements.

Heterogeneity can also stem from design differences between individual processors. Assuming constant operation costs gives no guarantee that constants are identical for all processors. Thus, the cost vector becomes a matrix of rows corresponding to processors, and columns corresponding to the set of operations. Consider a system of two DAXPY applications on two processors, one of which halves the cost of addition and multiplication due to a combined multiply-accumulate operation. Normalizing cost to the slowest processor, we get the matrices

$$R = \begin{bmatrix} n & n & n \\ n & n & n \end{bmatrix}, C = \begin{bmatrix} C(=) & C(+) & C(*) \\ C(=) & 0.5C(+) & 0.5C(*) \end{bmatrix} \tag{3.12}$$

The straightforward multiplication of requirement to cost implies that cost matrices combine by element-wise matrix product (denoted $\otimes$), to produce another 2×3 matrix, which is a complete map of aggregate operation costs per processor. The inner product with the vector of all ones $\vec{s} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ then produces superstep time requirements per processor.

The regular matrix product with a transposed cost matrix results in a 2×2 matrix with the time requirement appearing on the diagonal, but also featuring evaluations of the cost of mapping process 1's requirements onto the capabilities of processor 2, and vice versa. As our subsequent development emphasises process/processor affinity, the potential use to make scheduling decisions will not be examined, but it is interesting to note that the cost of various task mappings in a superstep can be analyzed by permutations of this matrix product.

The element-wise product estimates per-process superstep time as

$$R \otimes C \cdot \vec{s} = \begin{bmatrix} nC(=) & nC(+) & nC(*)) \\ nC(=) & 0.5nC(+) & 0.5nC(*) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \vec{t} \tag{3.13}$$

This is our general procedure to quantify computational heterogeneity in superstep terms. It has the advantage of giving a unified view of the variability caused by algorithmic and architectural concerns, while their causes are isolated in separate matrices.

To address the issue that operation costs are not constant, we must raise the abstraction level of basic operations. While single additions, assignments, etc. can vary nondeterministically, computation rate can be discussed more reliably in terms of execution rate of *kernels*. Asanovic *et al.* [11] identify a set of *dwarfs* which are representative kernels for a wide spectrum of applications, suggesting that this is a useful abstraction of computational demands. The access pattern of a kernel can often be profiled as independent of the input data, making it possible to execute them until they reach a steady processing rate which can be measured with great determinism. We will approach computational requirements with the assumption that applications apply them to large enough data sets that these steady states are reflected in execution.

A p processor system of k kernels then becomes a $p \times k$ requirement matrix R detailing the memory size the algorithm applies each kernel to, and a $p \times k$ cost matrix detailing the steady-state rate at which each processor computes each kernel, in terms of seconds per memory unit. These matrices are the model terms for computation, and the manner in which the $T_{compute}$ term in Equation 1.1 is decomposed to admit heterogeneous computational requirements and processing elements.

3.4 Heterogeneous Communication

Similarly to the development in Section 3.3, communication facilities are modeled in terms of linear systems. The main difference is that communication is already detached from application semantics. Message transmission cost is less dependent on the algorithm which requires it, so the description of rates lends itself to less application specific considerations.

We consider the communication capabilities of a p processor platform to be a fully connected graph, where any pair may exchange data during a superstep. Communication primitives are one-sided remote read/write operations, making the communication pattern a $p \times p$ incidence matrix, indexing source processors by row and destination processors by

columns. This already suffices to establish the cost of the synchronization which ends each superstep: it can be realized using a pattern of message counts only, because synchronization messages have negligible payload. The corresponding cost matrix is a $p \times p$ matrix encoding pairwise latencies. In the same manner, the cost associated with data payloads become a matrix of requirements containing the data volumes, and a cost matrix of inverse bandwidths between processor pairs.

This description is the *heterogeneous Hockney* model discussed by Lastovetsky *et al.* [61], and it is no more than a straightforward extension of the familiar Hockney model of communication cost [47]

$$T_{comm} = T_{latency} + w \cdot \beta \tag{3.14}$$

to a system of p^2 instances.

The terms of this model require refinements to validate with empirical measurements, but these are left for closer scrutiny in Chapter 5, which discusses their application.

3.5 Overlapped Computation and Communication

As discussed in Section 3.2, hiding communication overhead invites approaches of committing additional computation by exposing extra parallel work, replacing communication with duplicate computations, or by applications explicitly using delayed background communication. Scheduling and background approaches are complementary to each other, but have conflicting implications for system design. Although Bisseling [19] argues that perfect application overlap achieves only a speedup of 2, low scheduling overhead is tied to reducing the complexity of the processing cores, as context switches require storing their state. Sodan *et al.* [89] show that also on recent architectures, computational throughput can benefit significantly from complex processing cores.

We address the analysis of an algorithm's potential for overlap as displayed in the amount of computation inserted between transmission and reception. This is similar to the asynchronous *put* and *get* primitives of GASnet, which is employed as the communication layer of partitioned global address space (PGAS) languages [15]. It is also a common use for the non-blocking communication features of MPI [81], although the standard does not *require* that their implementations exploit overlap.

To derive the overlap term, we combine communication and computation as per Equation 3.15. It is validated using superstep time vectors, as in Equation 3.16.

$$\vec{t}_{compute} + \vec{t}_{communicate} =$$
$$(R_{comp} \otimes C_{kernel}) \cdot \vec{s} + (R_{messages} \otimes C_{latency} + R_{data} \otimes C_{\beta}) \cdot \vec{s} \tag{3.15}$$

$$\vec{t}_{overlap} = \vec{t}_{compute} + \vec{t}_{communicate} - \vec{t}_{total} \tag{3.16}$$

Notice that Equation 3.16 is subject to slightly different uses for validation and prediction purposes. The intention with respect to program analysis, is that identifying the operations applied between final communication and synchronization gives an estimate of the overlap, assuming complete transparency of the background communication. In an experimental setting, the exact magnitude of the overlap is more difficult to instrument than the total time. Thus, measurements of the right-hand side terms of Equation 3.16 yields an estimate of the actual workload successfully carried out in the background.

Chapter 4

Computational Rate

Determining the rate at which a modern processor can compute a given function is a complex issue. While the obvious relevance of the processor's clock speed is a parameter, multiplying an ideal rate in seconds per operation by a workload of operations falls short of characterizing the execution of a numerical kernel. Empirical performance figures vary with how the memory access pattern interacts with hardware memory hierarchy, the impact of virtual memory, the precision of the clock used, variability in operating system overhead, and properties of the input data in some cases.

To provide meaningful descriptions at the system level, this means that the heterogeneity of computation rate comes not only from a mixed processor configuration, but also from the balance of the computational load, and most parameters are affected by unpredictable interactions beyond user program control.

This chapter demonstrates some of the difficulties with obtaining a generalized processing speed, and suggests appropriate adaptations in order to characterize computational rate in a practical scenario, in a manner compatible with Equation 3.15. It focuses on program requirements in terms of kernel invocations, and the obtainable rate on processing elements. Figure 4.1 highlights the relevant components in the context of a complete model.

4.1 Impact of Variations in Time

To investigate the impact of memory layout, it is necessary to have a test framework which allows the various access patterns of computational kernels to be isolated. Arguably, our test platforms do not fully permit this, as sources of variability include not only a general operating system with several services on each node during run time, but also the fact that the systems service multiple users, offering no guarantee for repeatable experiments.

With the ambition of building a model which can be applied in such settings, we attempt to reduce the impact of these nondeterministic features, to produce a set of conditions

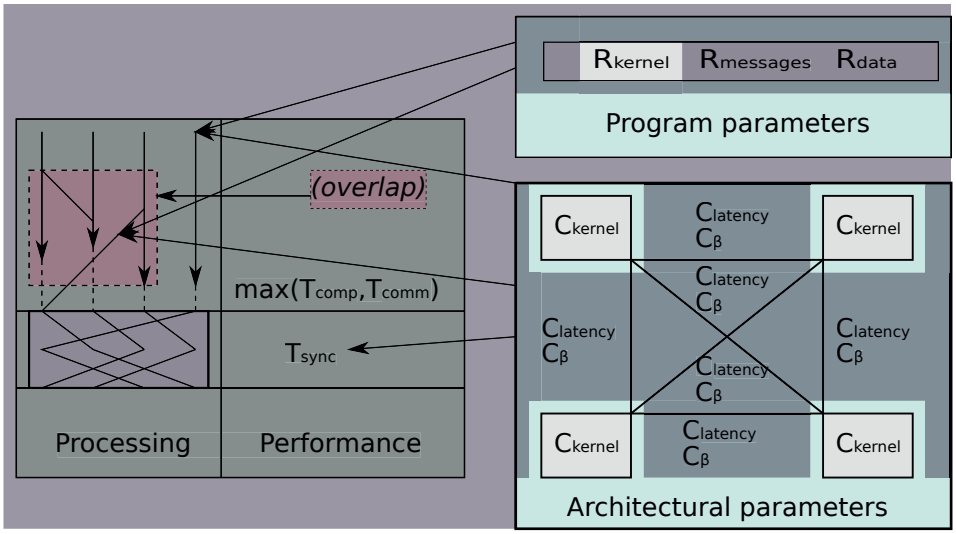

Figure 4.1: Architectural and program parameters of computational rate

where the model can be expected to validate. These errors, and the error margins obtained are as important to an applicable model as are the theoretical peak figures obtainable in a more controlled environment. We expect that modeling efforts on platforms with more deterministic behavior may produce more consistent results using a similar approach.

From the model point of view, we will ultimately attempt to characterize the computational power of a heterogeneous BSP machine; this makes it natural to begin with the *bspbench* program discussed in Section 3.1. Although it aims at establishing a uniform rate for an entire platform, its basic premise would allow a heterogeneous extension as suggested in Section 3.3 to introduce a vector $\vec{r} = [r_0, r_1, \cdots]$ to capture P different rates. The benchmark code [19] is compliant with *BSPlib*, so it can readily be run using any library implementation. We would like to emulate this portable mode of benchmarking because it provides direct comparisons between candidate implementations, but it regrettably lacks facilities for describing application and platform interactions.

As a starting point, Figure 4.2 shows the results of *bspbench* on one of our test platforms, including obtained rates for growing vector sizes in addition to the final output. It displays two issues: although the rate appears to stabilize around $1\frac{Gflop}{s}$ for this platform, the variations before this are not linear, meaning that the individual sample points are not descriptive of sustainable computation rate. More problematically, the $\frac{flop}{s}$ metric suggests that this result should also predict the attainable rate attainable for different kernels.

To adapt these measurements in a metric for sustainable rate in a heterogeneous environment, a few considerations must be taken into account. Most importantly, the averaging which provides the benchmark with its stability and repeatability must be made over several runs on a single processor. While averaging over the set of processors has the advantage of collecting a single statistic for the entire set, our aim of characterizing how

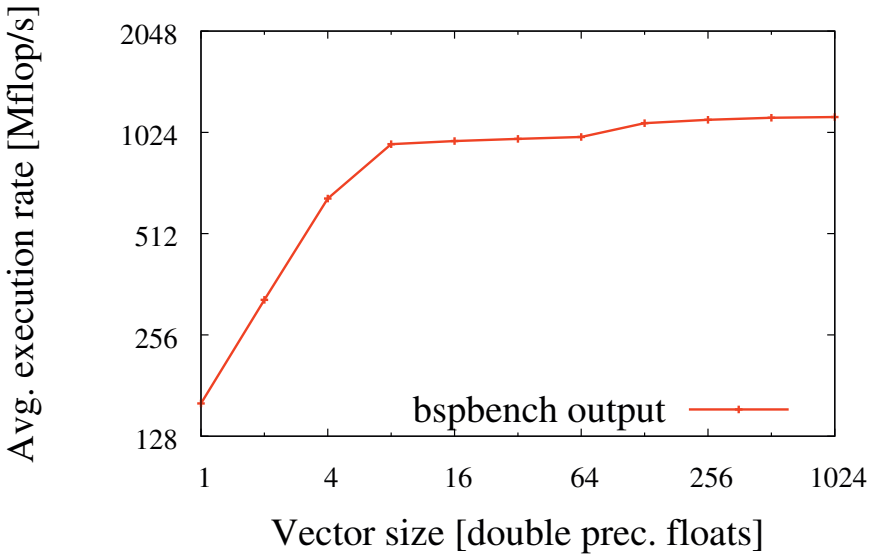

Figure 4.2: *bspbench* computation rates on 2x4 cluster node

problems map to various architectures means that we must capture any skew in its distribution, to obtain a more detailed picture than the central tendency provides.

Furthermore, since the objective of obtaining a rate per processor is to analyze steady state application behavior, it is problematic to describe it using only the largest sample distribution obtained by the benchmark. It is interesting to quantify how far this measure reflects the expected linear dependency between time and number of kernel applications, assuming that the problem size is large enough to reach a steady throughput.

A small benchmark program was developed for this purpose, isolating the timing and input parameters of a particular kernel from the measurement of its performance. The main assumption is that kernel behavior can be sufficiently isolated by controlling the virtual memory system, and the impact of OS-induced context switches. It overlooks the nondeterministic aspect of cache memory state, but since this is not usually under application control, we consider its impact to be embedded in sample variability.

The general framework program allocates a generic block of memory for kernel use, excluding virtual memory paging effects by pre-faulting all allocated pages, and pinning them in resident memory with the Linux `mlockall` system call. The kernel function call and relevant parameters are declared as external to the testing program, permitting different kernels to be isolated in object code which provides specific values for each of them. These are the amount of memory required for a test, the number of floating point operations in a single kernel application, pointers to functions which initialize data and apply the kernel, and the periodicity in terms of how many repeated runs can be performed before the data must be re-initialized.

Execution proceeds by growing iteration counts from 2 through 2^{12}, collecting 30 samples of per-iteration timings. The mean and standard deviations of these are recorded. Each distribution is tested for outliers, and outlier runs are collected again, until none remain. A linear relation between time and iteration count is computed as the least square error regression line through the distribution means. Finally, the execution rate predicted by the result is compared to a sequence of runs from 2 iterations up to 2^{24}, and relative error of the prediction is recorded.

The outlier filtering method warrants comment. Walpole *et al.* [98], give one definition of an outlier as a data point outside of an interval obtained from the other points in the sample. With a known distribution, the probability of extreme values can be evaluated, giving a quantitative confidence that the mean estimator is representative. As a mean of means is known to be normally distributed by the central limit theorem, this can restrict admitted samples to represent common system behavior. We require all sample distributions to have means within a 95% interval, repeating outlier runs until the criterion is satisfied. The outlier filter of the benchmarking program approximates normal distribution of the mean estimate using the Student-t distribution. Critical values of the interval are found by integrating its probability density using `tgamma` from the standard C library, using the trapezoid method to the nearest interval of $1 \cdot 10^{-4}$, and approximating the critical point by linear interpolation below this resolution.

Using the confidence interval as selection criterion instead of a hypothesis test creates

a bias in our observations. It renders the process vulnerable to non-representative initial samplings, as this makes the re-sampling continue until a consistent set of abnormal observations appear. Practical use amounts to calibrating experiments. Result sets that require a relaxed confidence will be difficult to reproduce, while restrictive bounds alert when a common effect is missed, since this makes the number of required re-runs larger than an expectable number of extreme observations. Discarding a 5% quantile from 30 values gives an expectation of 1.5 values to re-sample. Experiments consistently requiring 2 or more re-runs either suggest that initial sampling has recorded an uncommon result set, or that inherent variability in the experiment requires lowering our confidence in the accuracy of the mean value. A 95% interval gave stable results using only the named mechanisms, which are available to application programs in user-space. It is chosen as our balance between a controlled benchmark environment and realistic application conditions.

Figure 4.3 shows results from applying this general benchmark to two distinct numerical kernels, one being the DAXPY kernel for 1024-element vectors, and another being a 5-point stencil kernel applied to the interior of a $32^2 = 1024$ element area. Problem size is fixed at 1024 because it corresponds to the largest vector size in *bspbench*, and thus permits comparison with the predictions of the rate in Figure 4.2. These predictions are labelled "Mflops", and are computed by multiplying the number of kernel applications by the kernel's operation count, dividing by the maximal Mflops rate obtained by *bspbench*. The results in show that the benchmark predicts the behavior of the DAXPY kernel quite similarly to the *bspbench* approach, verifying that averaging across iterations produces comparable results to averaging across a homogeneous set of processors. More interestingly, the prediction for the 5-point stencil is more accurate than the corresponding prediction extrapolated from the *bspbench* rate. This verifies our expectation that the cost of a floating point operation must be considered in the context of its application.

While it is known that accurate measurement requires performance metrics to be parameteric in the workload, our experiment also gives a test of our procedure for determining processing rate in terms of kernel applications, and an estimate of how trustworthy the resulting predictions are. Figure 4.4 shows the error of predictions for the two kernels as a fraction of the total execution time of actual runs. While Figure 4.3 clearly displays the linearity with time, it important to note that the time scale spans several orders of magnitude. The deviations in Figure 4.4 must be seen in relation to this, as a 25% error in cases such as the largest 5-point stencil application means an absolute deviation of several seconds. This effect is unavoidable, as the inevitable variations over the benchmarking interval amount to an uncertainty which accumulates when predictions are made for longer runs. Note that although the results in Figures 4.3 and 4.4 remain within bounded relative error for intervals many times longer than the benchmark runs, their absolute accuracy inevitably depends on the admissible benchmark run length. For our further analysis, this implies that the procedure is best applied to scalability analysis in the weak mode, as fixed subproblem sizes per process translate directly into a workload unit.

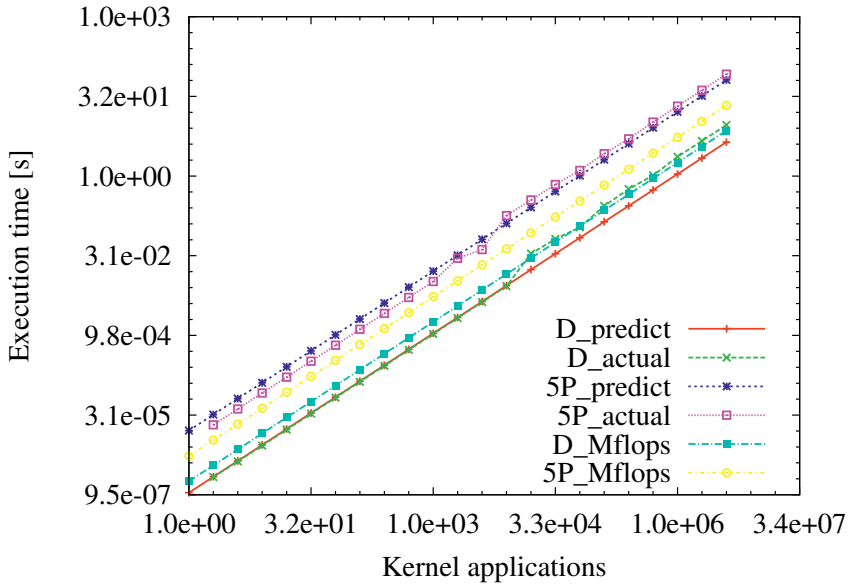

Figure 4.3: Rates and predictions of 2 kernels on 2x4 cluster node
D*: DAXPY kernel*
5P*: 5-point stencil kernel*
predict*: predictions obtained using benchmarks described in the text.*
actual*: empirically measured execution rate*
Mflops*: predictions obtained from* bspbench *computation rate*
Both kernels are applied to a data set of 1024 double precision numbers. Memory traffic is kept at a minimum by reusing a small in-cache area, making the graphs reflect the expected linear dependence between the number of operations and the time consumed. The results show how two different access patterns produce variations in rate even in a severely restricted setting. Predictions from benchmarks of individual kernels reflect performance in both cases. Predictions from the bspbench *computation rate remain close to the DAXPY kernel which is the basis of the benchmark, but deviate from actual performance when applied to the 5-point stencil. Log-log scales are used to even the distribution of data points, note that this conceals large absolute differences in high values.*

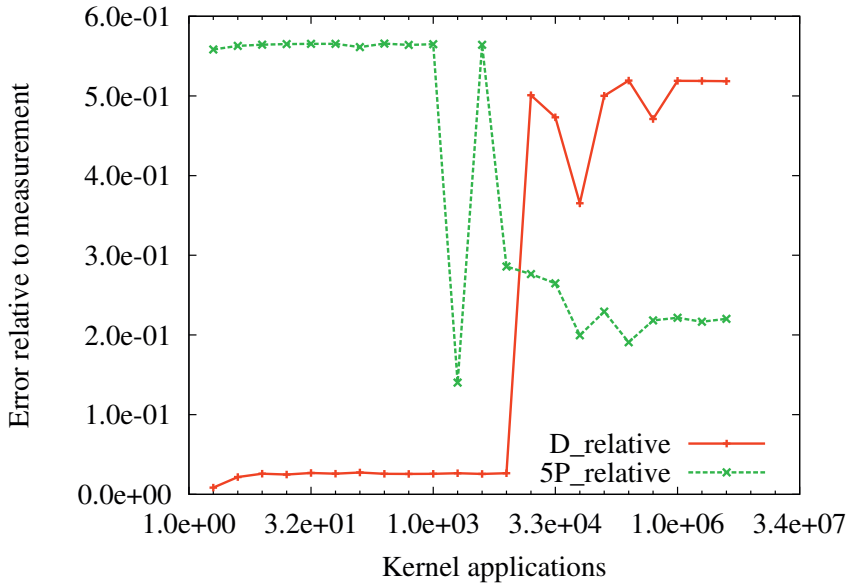

Figure 4.4: Relative misprediction of 2 kernels on 2x4 cluster node
D*: DAXPY kernel*
5P*: 5-point stencil kernel*
The ratio of kernel-specific prediction deviation to the magnitude of timings from Figure 4.3 shows that the accumulated inaccuracy of extrapolating benchmark predictions in time becomes large, but remains bounded. This highlights the importance of obtaining computation rate profiles on a time scale comparable to that of the desired prediction.

4.2 Impact of Variation in Memory Footprint

The benchmarking procedure in Section 4.1 provides a method for capturing the kernel-dependent processing rate for fixed problem sizes. Applying it repeatedly with various problem sizes releases control over the impact of locality in hierarchical memory, which yields an image of the impact it has on result variability.

Figures 4.5 and 4.6 show empirical results obtained by page-locking memory and collecting batches of 64 consecutive runs. Median time is reported as a function of memory use, for a selection of the level 1 BLAS routines from the automatically tuned ATLAS implementation [99]. An adaptively performance tuned implementation is used because its exploitation of the memory hierarchy is likely to be near optimal, making it probable that performance bottlenecks have architectural explanations. The reported tests were carried out using an ATLAS package adapted to an Athlon X2 processor, which features private caches per core, with 64K at level 1. The advantage of using this relatively old test system is that it has limited cache sizes which display data locality impact at small problem sizes, permitting tests to proceed quickly. Its private resources per core means that locking the executing program to one core effectively eliminates interference from other running software inside the O/S scheduling period, and exclusive access to the system could be guaranteed. The tests are not intended to illustrate ATLAS peak performance.

The selection of routines is restricted to the single precision level 1 (vector/vector) routines. This could easily be extended to include double precision, as well as matrix/vector and matrix/matrix operations at levels 2 and 3, but the performance effect we seek to illustrate is already visible. A natural problem size metric for testing BLAS kernels would be the n parameter supplied to each kernel, but Figures 4.5 and 4.6 express the problem size in bytes, through multiplying it by the size of the operand types, as well as a kernel-dependent factor of 1 or 2, depending on whether the given operation is a scalar/vector or a vector/vector operation. This clarifies the relationship between memory access pattern and observed performance, making the parameter values for *e.g.* the *scal* and *axpy* kernels comparable, even though the former causes access to half as many values as the latter.

As can be seen from Figure 4.5, time varies close to linearly with problem size for all kernels while problem sizes are restricted to the L1 cache, where the cost of memory access is close to uniform for regular access patterns such as the tested kernels. The value of this illustration lies in showing how the heterogeneity in computational cost which stems from choice of numerical kernel is significant enough to warrant that models must account for their magnitude. Even with the most homogeneous architectural performance components we can isolate (*i.e.* a private, fast memory of uniform access time), modeling the computational rate of a processor in terms of its rate on a *axpy* problem is inaccurate; in this case, it would mispredict the *dot* kernel performance by an approximate factor two.

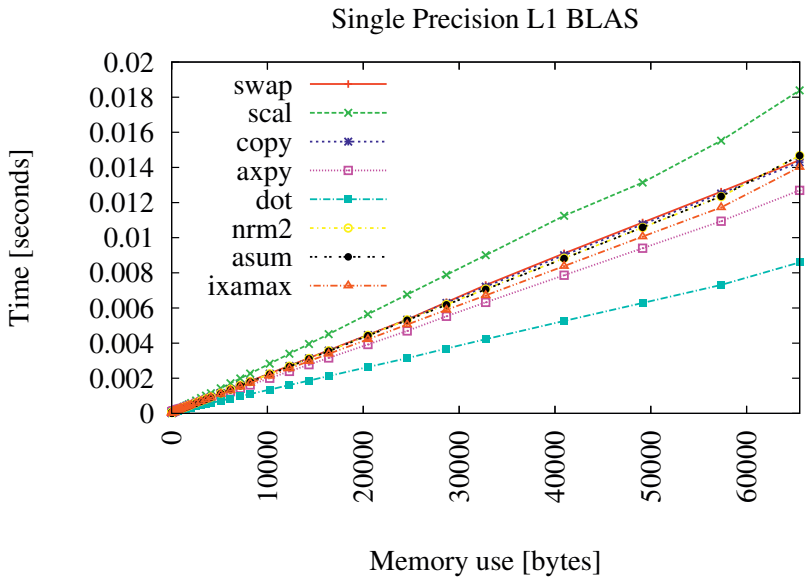

Figure 4.5: L1 BLAS performance, in-cache problem sizes on Athlon X2
Graph labels correspond to the naming scheme of L1 BLAS operations, which vector-vector operations common to numerical software. The restriction of input sizes to fit in private cpu cache provides practically uniform access time, resulting in a linear relationship between the number of operations in a kernel and the number of elements they are applied to.

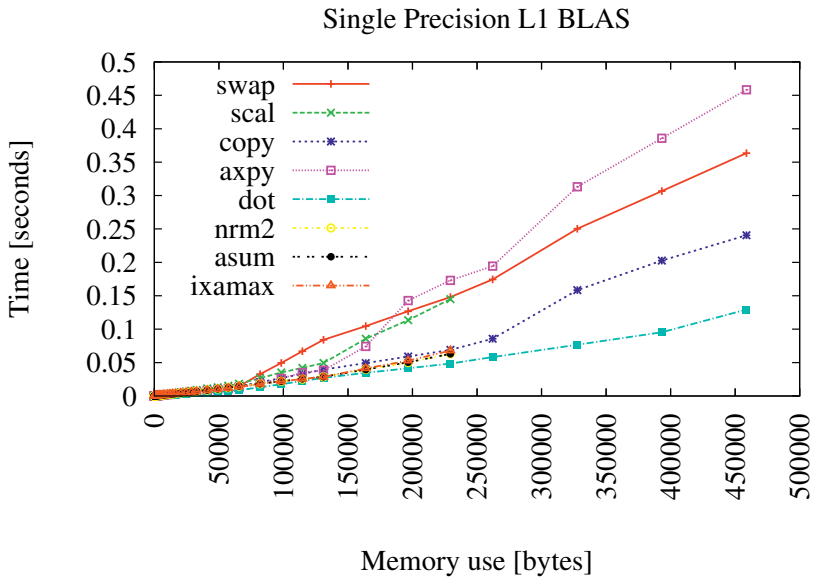

Figure 4.6: L1 BLAS performance, 64K-element problem sizes on Athlon X2
With an identical set of kernels as Figure 4.5, scaling the problem sizes out of cache memory shows that sustained computational rate develops nonlinearly, even when accounting for variations in both time and operation count.

4.3 Modeling Implications

In Sections 4.1 and 4.2, we have examined the consequences of increasing computational demand both by adding more iterations to a given memory footprint, and by growing the memory footprint for a fixed iteration count. The central idea of the proposed modeling framework is to approximate the complex interactions between aspects of execution by using linear systems. Figure 4.6 shows that the issue of varying the memory footprint of the problem creates a nonlinearity which must be addressed with respect to this.

The issues of nonuniform memory access are straightforward to observe, and have been pointed out countless times in the literature. Our context suggests two approaches to handle it: either the cost matrix of kernels can be expressed as an array of nonlinear functions of problem size, or it can be approximated by a set of piecewise linear functions.

The former approach is more general, but regrettably also less practical. The idea of modeling with linear systems of performance parameters which quickly outgrow what can be managed by pure reasoning, is that most of the work associated with evaluating large linear systems can be automated. The derivation of an appropriate set of functions to describe a set of sample observations is, however, rather involved. Introducing a matrix of nonlinear functions would thus undermine the purpose of the framework.

Approximation by piecewise linear functions is more feasible, but still requires human interaction and architectural understanding beyond what is reasonable to automate. The repeatability these experiments attain by excluding most accidental noise through memory locking, processor affinity, timer precision and disregarding outliers, indicates that the deviations from linearity in the resulting graphs are likely to be far smaller than deviations observed in a practical application scenario. Extracting linear regression lines from the entire spectrum of samples in Figure 4.6 would result in a measure of computation rate which would be inappropriate for the majority of cases. There are two obvious segments to the graph, with a steeper gradient breaking away around the L1 cache size limit, so these performance results could, in principle, be modeled by piecewise linear functions, through developing separate regression lines on the respective intervals. In model terms, this could be realized by adopting separate compute-rate matrix entries for a given kernel for distinct intervals, and similarly splitting the requirement into two parameters based on input size. Such a method certainly seems feasible for kernels which display the behavior we observe here, but rewriting Equation 3.15 to account for arbitrary extensions would serve more to confuse the notation than to clarify any modeling approach.

While a structured approach to a two-level memory hierarchy memory could be automatized, it would require benchmarks to expect the discontinuity, and thereby specialize them. As the tendency of later years indicates that the memory hierarchy will continue to deepen, each level adds $2P$ parameters per kernel to the computational rate terms of a performance model, and requires re-evaluation of benchmarking practices.

The implication which can be drawn from this is that the framework supports considerations of scalability with respect to problem size best in the weak mode of analysis. If subproblem size (in terms of memory footprint) is kept constant, the development in Sec-

tion 4.1 shows that a cost profile of linear characteristics can be extracted for a processing element, while accounting for the heterogeneity inherent to the algorithm. Upscaling the problem size then implies that the task be divided among more processors, effectively transferring the question of overall accuracy into the communication requirements.

This limitation is consistent with the subdivision of execution time into computational and communication parts, as traversal of the memory hierarchy can be considered communication, albeit at a level of granularity hidden by programming abstractions. The corresponding abstraction in model terms is that integrating communication cost into the cost of computation requires that its magnitude must be measured to a controlled constant.

Chapter 5

Communication Latency

This chapter introduces a model which accounts for heterogeneous latency, and develops a technique for automatic performance tuning of synchronization algorithms with respect to the underlying architecture. This latency-driven model is extended to account for bandwidth requirements, and the resulting model is shown to correspond with the empirical performance figures obtained from the barrier construct of a BSP implementation.

The following sections develop a performance model for barrier synchronization in terms of message counts and pairwise latencies. Figure 5.1 highlights these in complete model context. The resulting performance model component shows predictive power on multi-core cluster platforms, without depending on a particular communication topology.

Section 5.1 summarizes conclusions drawn from preliminary work, which guide the development of the model. Section 5.2 describes how benchmarks of topological distances are kept consistent throughout subsequent experimental work. Section 5.3 presents the model from both algorithmic and performance perspectives, and demonstrates how these relate to each other. Section 5.4 shows a brief analysis of three barrier algorithms in using big-O notation, in contrast to a more detailed method developed in Sections 5.5 and 5.6.

5.1 The Cost Impact of Locality

Due to the need for establishing mutual exclusion, developing efficient guarantees for consistent state is of immediate importance to any system which permits concurrent execution. In a shared-memory context, the requirements of ensuring exclusive access to any resource coincide with the requirements for ensuring that all processes have reached the same point in execution, by considering a data structure where processes register their arrival as the shared resource. A *spinlock* is a common construct provided at the operating system level, which guarantees mutual exclusion through atomic locks and busy-waiting. Because of the relative simplicity of this construct, studying its performance characteristics is a fruitful starting point for the development of more elaborate models.

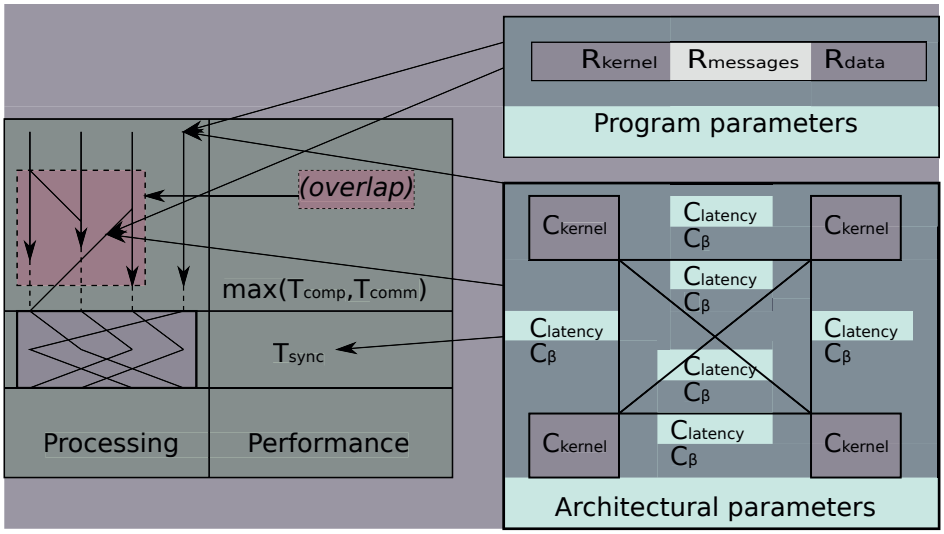

Figure 5.1: Architectural and program parameters of communication latency

The empirical study of Mellor-Crummey and Scott [69] provides an example of how extensive testing of classes of algorithms can yield architectural explanations for the obtained performance. This article successfully leveraged the conclusions drawn towards improving algorithmic scalability on the test platforms, and was able to make successful recommendations for future hardware design based on the outcome. With architectural development in supercomputing turning towards clusters of smaller shared-memory systems during the early phases of this work, the validity of Mellor-Crummey and Scott's conclusions approached limits in the scope of their empirical groundwork. Although several later works follow its standard of empirical testing and architectural justification on somewhat newer hardware [63, 65, 76], the growing number of memory hierarchy levels, and corresponding nonlinear memory access cost made it pertinent to revisit the topic.

Implementation and testing of Mellor-Crummey and Scott's selection of spinlock algorithms on more contemporary platforms confirms that process and lock locality is important to performance. Indeed, its significance has grown to a point where it overshadows the aggregate bandwidth limit emphasised in earlier studies. This leaves a simpler class of spinlock algorithms with limited utility, as hierarchical memory gives a vast advantage to a subset of processes in the face of lock contention. As the results which establish this effect have already been published [72], an outline of their impact on subsequent work is presented here in place of a detailed analysis.

Because of their definition, spinlock algorithms are tied to shared memory architectures. As the cost of hardware designs to present cache-coherent shared addressing physically distributed memory can become prohibitively expensive at large scales, the performance of spinlock based synchronization is relevant insofar as it can be taken to approximate synchronization costs for programs of limited parallelism, or for component subsystems in

a larger installation. With respect to the barrier implementations we are interested in here, the cost of acquiring a lock forms only a component of barrier cost, but it can be understood as a lower bound on a barrier cost function. A performance parameter which dominates lock acquisition time captures the best-case scenario for a barrier, in that it represents the overhead of a single process atomically signalling its arrival. It is worthwhile to note that this overhead remains measurably connected to the topological placement of processes at both intra-chip, inter-chip and network scales, even for tightly coupled systems where significant design efforts have been devoted to masking the locality effect.

In summary, the work done on spinlocks establishes two main guidelines which give direction to subsequent synchronization cost modeling efforts:

1. Process locality must be controlled to reliably measure synchronization cost.
2. As bandwidth no longer dominates cost under contention, a model of synchronization cost should focus on the relationship between topological distance and communication latency.

5.2 Processor Affinity

The guideline that subsequent developments rely on strict control of locality presents practical programming issues, as there is no portable or standardized general interface to control this aspect of program execution on multiple scales. In the following sections, all magnitudes which rely on locality are kept under control by applying the Linux process affinity control interface at the shared memory level, and ad-hoc adaptations to obtain identical node sets from system scheduling software on the distributed memory level.

This is done to bring the parameter under control for experimental purposes, but it is a poor solution in terms of developing robust software. The affinity interface does implement the ability to specify that a process should execute only on a given core index, but it cannot guarantee that there is a consistent relation between physical cores and core indices at the hardware level. Consistent grouping at the distributed memory level is done using specific IP host names of the target systems, which is not only dependent on the exact system used, but also represents a level of control which lies outside the scope of what is conventionally controllable from inside a given program.

Although unique identification of shared-memory systems in larger-scale interconnects is available, and many operating systems implement some interface for shared memory system affinity control, the lack of a standard interface for identifying locality on a system level means that programs which utilize these mechanisms suffer degraded portability.

A portable means of finding a unique processing core identifier might help with this issue, but might require a vendor-independent address authority, carry privacy concerns with respect to consumer products, require a consistent policy on what sort of circuitry should be identifiable by this mechanism, etc. etc. Addressing this sort of problem is more appropriate for a standards committee than for research work, but because of its impact on presented results, the method by which the issue is bypassed is described here, for the sake

of completeness and reproducibility. All empirical tests in the following work is layered on top of MPI and the POSIX threading library interface. For the purpose of maintaining locality between independent executions, a small library function is employed, which accepts the MPI rank of the calling process and the size of the world communicator as inputs. Internally, it gathers the RFC1035 host names of all nodes on the distributed memory level, and finds their core count using the POSIX defined `sysconf` system call. A sorted list of the ranks resident on a given SMP node permits each to declare affinity with the core index corresponding to its position in the list modulus the node's number of processors, which creates a consistent mapping between rank and core identifier. Note that while there is no guarantee of core indexing between nodes corresponding to identical mappings, the repeatability attained by creating the same mapping for every run on a given node is sufficient to associate whatever measurable properties it has with the corresponding MPI rank.

The applicability of obtained empirical results is thus restricted to programs which enforce the same affinity scheme, but for the purposes of this thesis, it will be considered sufficient to assume some equivalent placement scheme. Henceforth, this issue will only be briefly revisited, in order to highlight practical implications of presented results in Section 7.5.

5.3 A Practical Cost Model on Distributed Memory

Irrespective of the relative impacts of candidate performance parameters, the first step in establishing an effective cost model for a barrier primitive is to characterize its communication requirements. In order to clarify employed methods as they are presented, we will introduce three different barrier algorithms for use as running examples: the *linear* barrier, the *tree* barrier, and the *dissemination* barrier. In order to provide the reader with an intuition for their variable communication requirements, we begin with a brief, informal description of how each algorithm operates.

The linear barrier is the naive implementation of a simple arrival count. As each process arrives at the barrier, it is counted by a master process, and begins waiting for an acknowledgement that the barrier is complete. When the count of arrived processes equals the expected number of participants, the master process signals each of the waiting processes in turn, and execution proceeds.

The tree barrier, as its name suggests, creates a tree of process identifiers, with the process at the root taking the role of master process. The operation of the algorithm resembles the linear barrier in that each arriving process signals its master (the process one level above it in the tree), before waiting for a signal to proceed. Different from the linear barrier, however, the tree structure improves scalability by distribution the contention which would otherwise affect the single master process. Furthermore, both arrival and release signals can be propagated in parallel down the tree, relieving the process at the root from having to signal *every* participant.

The dissemination barrier differs from the other two, in that it does not explicitly nominate a master process, but relies instead on a cyclic communication pattern to guarantee that

no process is released prematurely. For P participants, the pattern proceeds in $0 \leq s < logP$ stages, with process identifier p signalling process $p + (2^s mod P)$ at each stage, and awaiting one signal. The resulting pattern is equivalent to a cyclic shift in each of the $logP$ dimensions of a hypercube: as no stage will be completed before the arrival of every process' neighbor along dimension s, each process can guarantee global arrival as soon as the cycle is complete in every dimension.

These particular algorithms are chosen because they span an interesting range of design tradeoffs. The linear barrier works in only 2 stages, but places the entire workload of tracking state on the single master process. The dissemination barrier makes the opposite choice, and distributes the responsibility for representing subsets of other processes evenly among all participants. This frees it from any single process becoming a bottleneck, at the expense of incurring a communication pattern which stresses the entire interconnect in most stages. Finally, the tree barrier offloads this bottleneck and retains the principle that one process is made responsible for the signalling related to those below it in the tree. This makes it suitable for adapting to interconnects which resemble its internal tree structure, but requires twice as many stages as the dissemination barrier.

The classification of the barrier algorithms according to the number of stages they proceed in, warrants further comment. Arguably, the collective cost of the operation is connected to the number and cost of the transmitted messages, as much as the number of stages of the overall algorithm. If we consider the linear barrier, its very name implies the expectation that its cost is in linear relationship with the number of participating processes, whereas the two stages remain constant for any level of parallelism. Similarly, although the tree barrier proceeds in a logarithmic number of stages, the number of processes which transmit a signal in each subsequent stage is halved (or doubled), depending on the phase of execution.

The justification for emphasising the stages of execution is that it corresponds to the general notion of splitting algorithms into their sequentially dependent parts, to expose the parallelism within each part. Thus, the constant number of stages in the linear barrier is not in direct proportion to its time complexity, but rather represents a count of the number of sequential dependencies it introduces. This approach maps easily onto the manner in which cost functions will be developed in subsequent sections. Practical application of the approach will also make it evident that this partitioning of algorithms has an effect on the accuracy of model predictions.

5.4 Asymptotic Barrier Analysis

A common textbook approach to analyzing barrier scalability is to derive a cost function for the asymptotic behavior, *i.e.* to explicitly attach costs to the messages in a given pattern, and derive a count of the number of messages for variable process count P. For a homogeneous communication topology with message cost c, this formula can be deduced

as a summation of the sequentially dependent signal paths, *i.e.*

$$T_{linear}(P) = 2 \cdot \sum_{p=0}^{P-1} c = 2cP = O(P)$$

$$T_{tree}(P) = 2c \cdot \sum_{s=0}^{\lceil log_2 P \rceil} c = 2c \cdot \lceil log_2(P) \rceil = O(log_2 P)$$

$$T_{diss}(P) = c \cdot \sum_{s=0}^{\lceil log_2 P \rceil} c = c \cdot \lceil log_2(P) \rceil = O(log_2 P)$$

As the assumption of uniform cost is unrealistic for most modern systems, these sums can be split into *e.g.* stages requiring local and remote communication, to determine the dominant term of the total. Doing so reveals that the even distribution of dissemination barrier signals causes remote communication to dominate most of its stages, which causes underutilization of local high-speed links. This in turn may explain why tree barriers are more commonly implemented on hierarchical interconnects.

Since this type of analysis is well known, and mainly represents an exercise in arithmetic, it will not be elaborated upon here, save to note that the results it produces are specific in terms of both algorithm and cost functions, and seldom accurately capture cases where the algorithm's interaction with the topology causes exceptions to the common case behavior.

5.5 Matrix Representation of Algorithms

In order to make a model which is independent from the specific acknowledgement of each message's impact on asymptotic time, it is fruitful to recognize that *any* barrier communication pattern can be encoded as a layered dependency graph, with each layer representing a stage. The possible signals transmitted in each stage form a directed graph with each process represented by a vertex, and the execution of the entire barrier amounts to each process transmitting its part of the signal pattern, and awaiting any inbound communication before proceeding to the next stage. Drawing this sort of graph by hand is a common way to understand a communication pattern before deriving an asymptotic cost function, but in order to capture this knowledge in the cost function itself, the dependency graph must be manipulated computationally.

A common encoding of an arbitrary directed graph with P vertices is by using a $P \times P$ boolean incidence matrix. As the communication pattern captured in one such matrix represents the execution of one barrier stage, the execution of a complete barrier can be described in a sequence of matrices $S_0, S_1, \ldots S_s$, where s is the total number of barrier stages, and each stage is a layer in the full dependency graph. The interpretation that $S_k(i, j) = 1$ means that "process i signals to process j in stage k" results in a notation which captures both the progress of a barrier in terms of sequential dependencies, and the set of signals which can be in transmission simultaneously.

$$S_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, S_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 5.2: 4-process linear barrier in matrix form

$$S_0 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}, S_1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 5.3: 4-process dissemination barrier in matrix form

$$S_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, S_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$S_2 = S_1^T, S_3 = S_0^T$$

Figure 5.4: 4-process binary tree barrier in matrix form

For illustration purposes, Figs. 5.2, 5.3, and 5.4 show the incidence matrices for our 3 examples in a 4-process case: 0 is chosen as the arbitrary master/root rank for the linear and tree barriers, and the degree of the tree barrier is chosen to be 2 (binary tree). Note that the stages in the acknowledgement stages of the tree barrier are the transposed arrival stages in reverse order. This is a reflection of how the algorithm successively extracts representative processes from shrinking subsets of the total. It is a property which will hold for any barrier of similar, hierarchical construction.

As an interesting side note, the presented view of barrier operation can be used to examine the correctness of a candidate algorithm, as the matrix representation naturally maps the barrier's flow of information onto linear algebra operations. Consider a $P \times P$ integer matrix K, where the element $K(i, j)$ represents the number of messages process i has received as acknowledgements of process j-s arrival. At the beginning of barrier execution, each process has knowledge only of its own arrival, which corresponds to the $P \times P$ identity matrix I under our interpretation. After the execution of stage 0, this is extended by the direct implication that any process which has received a message from another, can be confident that it has arrived. Thus, the aggregate knowledge after executing the pattern in S_0 grows to

$$K_0 = I + S_0.$$

In subsequent stages, the signals sent by a process i will signify not only its own presence, but also that it has received the signals from all processes which contacted it in previous stages. Thus, the pattern of the next stage multiplies with the knowledge already established, to yield an update for the global count:

$$K_1 = (I + S_0) + (I + S_0) \times S_1.$$

Inductively, this means that the equations

$$K_0 = I + S_0. \tag{5.1}$$

$$K_i = K_{i-1} + K_{i-1} \times S_i | 0 < i \leq s \tag{5.2}$$

establish the signal count K_s at stage s. The definition of a barrier is that no process may leave it before all processes have arrives. A test on whether this has been achieved in state s is equivalent to testing whether K_s contains all nonzero elements.

This simple method falls short of providing a completely automated proof of correctness, because the computation must be reiterated for each desired value of P. Nevertheless, it is mentioned here because it illustrates the value of representing the algorithm in an encoding which can be examined programmatically, and because its modest computational demand makes it a useful debugging tool for automatically generated patterns. A straightforward empirical verification of correct synchronization is to execute the program P times on a P-way parallel system, and purposely delaying the arrival of each process in turn, to observe whether the expected delay is visible in the overall completion time. Testing by Equations 5.1 and 5.2 allows an incorrectly specified pattern to be simulated, identifying an exact trace of the failure within negligible time on a regular, personal computer.

Having introduced the relevant notation, it is appropriate at this point to remark on how the approach taken here considers static communication patterns with respect to process identifiers. This overlooks the possibility of a run-time optimization, taken in the *tournament* barrier algorithm. That algorithm is essentially identical to the tree algorithm, except that the selection of the process which is responsible for signalling a subset of others is made at run time. Such an optimization can easily be implemented on shared-memory systems, by maintaining a shared vector which permutes the process identifiers into equivalent ranks for the purposes of the barrier. As an example, the first arriving process from a subset could detect from the shared tree structure that it is the first to arrive, and thereby nominate itself for handling later communication, giving the possibility of masking associated setup costs while the rest of its group arrive. Altering the effective identifier of a process in this manner would also be possible in a distributed memory context, but the implementation issues of ensuring that the permutation of identifiers is consistent creates more complicated implementation issues. Furthermore, as the optimization aims to exploit the sequence and delay in arrivals, its effectiveness is connected to parameters outside the barrier algorithm design space. Since altering the relationship between process identifier and topological placement at run time also complicates fulfilling the ambition to control the impact of process locality, this type of optimization will not be subject to further discussion.

5.6 Matrix Representation of Performance Parameters

With a detailed representation of the sequencing of communication operations in place, it is appropriate to develop a model of architectural performance parameters in a compatible manner, so that the interaction between the two may be captured in a common cost function. The assumption which underlies the matrix representation of an algorithm, is that collective system behavior can be captured by superposition of the individual point-to-point interactions. Extending this assumption into the space of performance parameters, a cost function which expresses the behavior of a barrier algorithm must express the relative weights of the point-to-point interaction, so that the dominant term can be found from the total.

While this general consideration captures that the overall cost function will be parametric in matrix encodings of both the algorithm and some similar encoding of the weights of pairwise communication facilities, the exact form of the latter is obviously quite dependent on the implementation of the interconnection networks employed. Further discussion will therefore sacrifice some generality for the sake of showing how the approach applies to common ethernet connected clusters with multicore processors, with the assumption that similar efforts will result in a similar development for other architectures, although perhaps with a different balance between the identified parameters.

5.6.1 General Barrier Simulation

The benefit of encoding the point-to-point communication requirements of an algorithm in matrix format is that it permits a single, general program to simulate the algorithm, with

```
void barrier_execute ( barrier_t *barrier ) {
  for ( int s=0; s<barrier->stages; s++ ) {
    MPI_Startall (
      (barrier->srcs[s]+barrier->dsts[s]), barrier->reqs[s]
    );
    MPI_Waitall (
      (barrier->srcs[s]+barrier->dsts[s]), barrier->reqs[s],
       MPI_STATUSES_IGNORE
    );
  }
}
```

Figure 5.5: Barrier simulation function with C / MPI

its communication pattern as input. As we have divided barrier execution into synchronized stages, the central part of such a simulator can be expressed quite succinctly using MPI communication primitives, which permits it to be portably deployed on a variety of platforms.

Figure 5.5 shows a code fragment from a testing framework in the C language with MPI, which runs through a communication pattern specified in a `barrier_t` datastructure. The significant contents of this structure are the `srcs` (sources) and `dsts` (destinations) arrays of `MPI_Request` objects, which are allocated and initialized prior to calling this function. They contain simple lists of requests which set up the transmission and reception of a minimal signal from process i to j. The requests which correspond to the pattern from a given stage can be trivially initialized given the matrix which encodes it. The nonblocking start of all requests for a stage, paired with the blocking wait for its completion assume that the library implementation will expose the available cost benefits of asynchronous transmission within a stage. Regardless of the actual level of optimization, it will in any event show the potential for asynchronous transmission exposed to an application programmer using the given library and platform. In this manner, determining the per-process cost in a single barrier stage depends on finding the parameters which impact the cost of asynchronously starting a set of communication requests.

5.6.2 Performance Parameters for a Barrier Stage

A common approximation for point-to-point communication time lies in the Hockney model [47], which Lastovetsky *et al.* [61] write as

$$\alpha + \beta M \tag{5.3}$$

with α denoting a constant latency which is taken to stem from the software stack and network, M is the size of the transmitted message, and β is the inverse bandwidth of the interconnect. This model extends naturally to the heterogeneous Hockney model [61] by recording the relevant parameters for all pairs of processes, turning α and β into $P \times P$ matrices and requiring that they are independently determined.

Applying this model to the suggested testing framework is, however, an oversimplification. Analytically, it assumes that the network latency and software overhead are constant, which fails to capture that the simultaneous initiation of a vector of signals has a different cost (in terms of software overhead) than initiating them sequentially. In addition to this, we have observed that the bandwidth cost of minimal-length messages is practically unobservable on modern architectures [72], which reduces the heterogeneous Hockney model to applying a matrix of constant values for α_{ij} to model a barrier step. For all its conceptual simplicity, this model is insufficient to produce empirically verifiable results in application scenarios. This has prompted the development of a range of models which partition α in various ways. Providing a greater survey of the work in this area is inappropriate here, as our present focus on pure synchronization reduces most of these models to the special case of negligible message lengths. Therefore, it is noted that the empirical emphasis in Bosque and Pastor's proposal of the HLogGP model [22] served as the initial inspiration to draw elements from this interesting field, and the reader is referred to Lastovetsky *et al.* [61] for a coherent summary of several other approaches.

Our approach to find the significant performance parameters of a barrier cost model is strongly driven by the requirement of finding a method of measurement which provides results which are both precise under repeated experiments, and permit meaningful composition. This is driven by the desire to produce a model which is first and foremost consistent with what can be observed by an application program on a given platform. It should be noted that this approach towards specifying a benchmark runs the risk of overlooking components of a cost function which may appear negligible on one design, yet prove important on another. For this reason, we underline that the development presented here is not general in terms of every possible interconnection technology, as we only establish that the results attainable on our target systems are general to the point of spanning heterogeneous compositions with a uniform software layer. General consensus on an analytical model for heterogeneous communication cost would greatly simplify the work involved in adapting this approach to other platforms, but as that matter is an active research topic, specializing on the most commonly available technology will serve the purpose of examining what the more general procedure can obtain in practice.

As the objective is to decompose the cost of transmitting minimal-length messages, the LogP model [27] provides a valuable perspective, in that it proposes a distinction between network latency and software overhead which has proven applicable in practice [22]. Because of the form of the basic step in the general barrier algorithm is a function call which starts a variable number of messages with negligible payload, the expectation is that cost can be approximated as some combination of

- the overhead of invoking the function,
- a term related to the number of requests started, and
- a term related to the distance between source and destination.

5.6.3 Benchmark Statistics

Because the time taken to execute the two function calls in the simulation is the only directly measurable quantity presented to the application programmer without further instrumentation, the separation of the terms must be extracted from the cost of testing various controlled communication patterns. Even if each parameter could be precisely determined by instrumentation at a lower level, it would be unrealistic to expect a precise prediction of empirical behavior, given that the observed cost is subject to network load, state of processor caches, state of the operating system, and a great number of other temporal conditions independent of the communication call. Therefore, we approach the performance parameters as statistics, and attempt to determine a representative magnitude for the majority of executions.

The overhead of a pure function invocation is estimated using repeated calls of P empty requests, which should cause no communication. Each call is timed individually for a number of repetitions, and the median value is extracted. The obtained approximation of overhead at processor i will be denoted O_i.

The cost related to the number of requests started requires isolating other costs related to request transmission from the difference caused by one of them. This can be estimated by transmitting a variable number of minimal messages; expecting a linear dependency between the number of requests for simultaneous transmission and the time taken, permitting the added cost of one message to be approximated by the gradient of a linear regression line found from the measurements. This approximation of the overhead of adding a signal between processors i and j will be denoted O_{ij}, where $i \neq j$.

The cost of the network transmission is similarly isolated by transmitting a variable amount of data. As larger messages emphasise the impact of the topological distance over the near-constant setup costs, linear regression provides an approximation of the per-byte transmission cost. The distance-dependent component of a message transmission is approximated by following this line to its interception point, which we take as the wire latency of a zero-length message. This approximation to the latency between processes i and j will be denoted L_{ij}, where $i \neq j$.

5.6.4 Benchmark Validation

Preliminary experimentation with a benchmark program suggested that the desire to maintain reproducible variability in the measured figures stabilized at approximately an order of magnitude lower than the measured result, bearing witness to a strong central tendency. This was observable for sample sizes above 25, with the growth of message sizes in the latency benchmark going through powers of 2, from 0 through 20. While greater sample sizes could presumably be leveraged to establish stronger results, stable repetitions were used as the criterion for selecting a minimal benchmark size, because the cost of running tests for $O(P^2)$ pairs quickly grew inconvenient with increased sample sizes. One point made evident by testing the cost of transmitting a vector, is that the distance-dependent latency term L_{ij} drops significantly when the destination process is known to have estab-

lished its receiving buffer, and awaits the signal. While this phenomenon in hardly curious in itself, the magnitude of this effect proved significant enough that it requires consideration in the construction of an overall cost function which incorporates several stages.

Although the number of variables outside of experimental control makes it difficult to establish the appropriateness of the benchmarks through analytical means, it can be appreciated by considering the predictive power attained using the measurements. The role of the experimental work in this context is not to establish metrics which can be determined to be universal to any technology, but rather, to derive a cost function which is compatible with the description of the algorithms for the given architecture.

5.6.5 Barrier Cost Model

With a stable benchmarking scheme in place, a model for combining the stages of the barrier algorithm can be composed. Given that the observed time of a complete barrier is dependent on the critical path through its graph, the cost of the entire algorithm may be estimated by recursively tracing each path, adding the cost weights of each edge, and recording the maximal value attained at every visit to the final stage.

The delay of each process in a stage depends on its entire signal vector, because of how all requests are activated in a single function call. A tentative estimate of the cost process i adds to each path through its stage s is expressed in Equation 5.4:

$$cost(s, i) = 2 \cdot \sum L_{ij} \cdot S_{sij} + \max_{j}(O_{ij} \cdot S_{sij}) \tag{5.4}$$

Put differently, the cost of process i-s signal vector depends on the sum of initial transmission costs of all the messages, as well as the maximal overhead of contacting any recipient. Note the appearance of the factor 2 in the aggregate cost of contacting all the recipients in the vector. The method described thus far approaches every act of communication from the sender's side, attempting to model the cost of communication as seen by the individual processor, because the one-way transmission of signals is the basic step in the description of the algorithm. While this is convenient in order to construct a benchmark, it disregards the fact that at a lower level of abstraction, signal transmission requires an acknowledgement, which is subject to a similar cost incurred at the receiving end. The factor 2 is used here because our present development aims to establish a practical model for application on an architecture which provides symmetric communication capabilities. Extending this towards asymmetric links would be worthwhile, but would also require extensive validation efforts, which presently would be of modest relevance compared to the required labor investment.

In addition to this, two conditions apply:

1. The minimal cost is the invocation cost O_{ii}
2. If a process j is known to be awaiting a signal, its term O_{ij} in the maximization can be replaced with its invocation cost O_{jj}.

Incorporating these points into Equation 5.4 would serve to obscure rather than clarify the notation, but a function which captures all these conditions can easily be implemented in program logic. In particular, the minimal cost is a simple matter of initializing the variable which tracks the maximal overhead to O_{ii}, and since the description of the entire barrier is available to the predictor program, it is possible to detect if a signal $j \rightarrow i$ was the last action in process j before $i \rightarrow j$ is expected, and j has been idle for one or more stages between. The latter case makes it very probable that j has posted its receive prior to i-s transmission.

5.6.6 Test Cases: 8x2x4 and 12x2x6 cluster configurations

The described cost function and conditions permit the model to be tested with two different topologies, in order to quantify its predictive power and describe its behavior. For this purpose, two clusters of multi-socket, multi-core nodes are employed: one with 8 gigabit-ethernet connected nodes, each featuring dual 4-core Intel Xeon processors, and one with 12 gigabit-ethernet connected nodes featuring dual 6-core AMD Opteron processors.

Result material is collected from each of these configurations by simulating the *dissemination* (D), *tree* (T), and *linear* (L) barriers for each process count admissible on both architectures. Worst-case times were collected from 256 runs on each process count, and the arithmetic mean of these is reported. For comparison, the benchmarks outlined in Section 5.6.3 are independently run for all process counts, producing a set of files containing the two $P \times P$ matrices corresponding to each configuration. In order to produce predicted values, the combination of these and the matrix representation of the algorithms form the input of a (sequential) predictor program, which recursively traverses each path through the barrier and reports worst-case prediction according to Section 5.6.5.

Note that the architectural parameters of the cost function are obtained independently from the barrier timings. Because process locality plays such an important role in this method, consistency between the process mapping in a timing run and cost parameter determination was enforced by programming both the benchmark and test to request the same node set, and the process-to-core affinity was explicitly specified in both programs through the Linux CPU affinity interface, as described in Section 5.2.

Figs. 5.6 and 5.7 report the absolute time from execution and prediction on the 8-way 2x4-core configuration. The primary value of these figures is to permit a visual confirmation that predicted values are approximately correct in terms of the cost function's growth and features, as well as absolute magnitude. Aside from this, closer scrutiny of their shape reveals a number of observable effects which have not been explicitly considered in our modeling effort. The presence of these artifacts which are emergent from the application of the independently developed models builds confidence that the predictive power of the modeling effort thus far captures some measure of those interactions which motivate its construction.

The most obvious example of this is in the oscillating effect found in the D-barrier, for process counts from 9 through 16. These tests correspond to tests spanning 2 compute nodes,

Figure 5.6: Measured barrier timings on 8-way 2x4-core cluster
D*: Dissemination barrier*
T*: Binary tree barrier*
L*: Linear barrier*

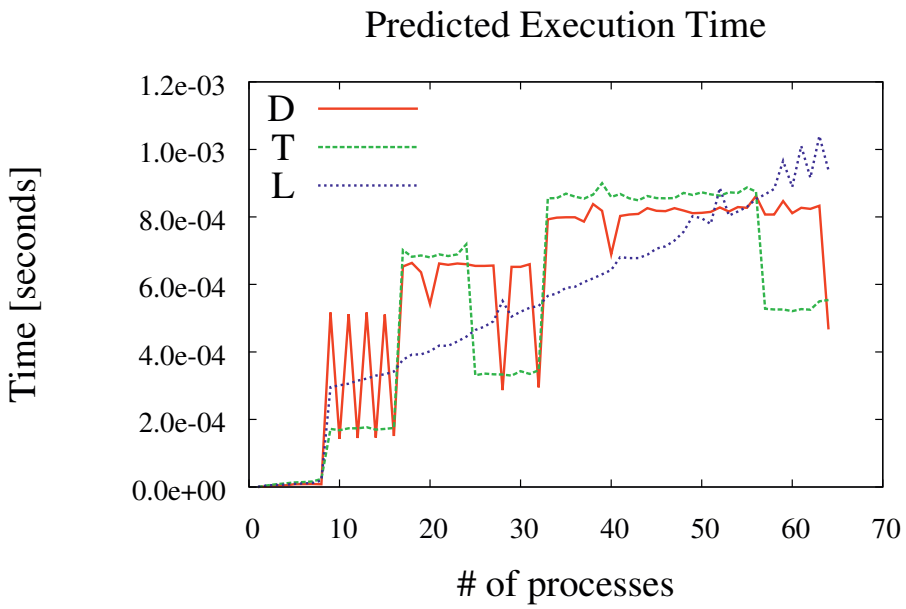

Figure 5.7: Predicted barrier timings on 8-way 2x4-core cluster
D*: Dissemination barrier*
T*: Binary tree barrier*
L*: Linear barrier*

and the observed effect can be explained from the interaction of the enforced process affinity with the algorithm and scheduling software which allocates processes to nodes on the cluster in question. The scheduling software of the test cluster allocates processes to nodes on a round-robin basis by default. Considering how the barrier in question distributes the aggregate communication requirement evenly on all participants, the case for 2 nodes thus maps out to processes with even and odd identifiers being located on the same node. As the communication pattern of the barrier modulates growing powers of 2 over P to find the communicating pairs in a given stage, stages where 2^s grows beyond P introduce a greater load on the more expensive inter-node interconnection in the odd cases than they do in the even. This emergent property is visible in the distance between odd and even case empirical timings, and the predictions visibly capture it. The oscillation itself can obviously be eliminated by reconfiguring the scheduling software, or even adapting the barrier pattern, but since our purpose presently is to construct a model which captures the interaction between synchronization cost and topological distance, the successful prediction of this effect is encouraging for the confidence in its validity. This interplay between mapping of identifiers is also visible in the sharp dips in cost at 28 and 32 processes, which is similarly captured by the model.

A second notable point is the generally lowered cost of the T-barrier for process counts from 25 through 32, and 57 through 64, *i.e.* node counts of 4 and 8. This is another interaction between the round-robin scheduler and the power of 2 layout of the algorithm: the successive halving of the communication pattern in the binary tree maps the advantage obtained by lower-cost local links better to the stages of the barrier when neighbors in the dependency graph are located near each other.

On the other hand, it is notable that the relationship between the predicted and actual performance of the L-barrier appears to deviate by some constant, and the precise relationship between the D and T barriers is reversed in the longer, flat regions of the graph.

Figs. 5.8 and 5.9 show the deviation between the predictions and measurements in absolute value, and relative to the magnitude of the measured value, respectively. The encouraging aspect of the absolute deviation is that its magnitude is rather small, remaining in the tenths of milliseconds even as overall barrier execution time grows. The worst case is the linear barrier, which appears to display linear growth in the inaccuracy, albeit at a lower rate than the overall execution time.

Similar performance measurements and predictions for a 12-node cluster configuration with 2x6-core nodes is displayed in Figs. 5.10 and 5.11, with corresponding absolute and relative error measurements in Figs. 5.12 and 5.13. Again, the deviation is strongest with the L-barrier, and the relative error measurements here suggests that the increased scale vs. attainable precision on this configuration may be straining the predictive power of the model in the larger cases.

On the other hand, the absolute measurements on this platform leave no ambiguity as to which of the D and T barriers has the superior performance in all multi-node configurations, and save for an aberration in the interception point between the L and T barriers, the growth rate of the cost function appears to be captured. The absolute error in Fig. 5.12 reveals that the deviation lies within tenths of milliseconds, making the case that predictions

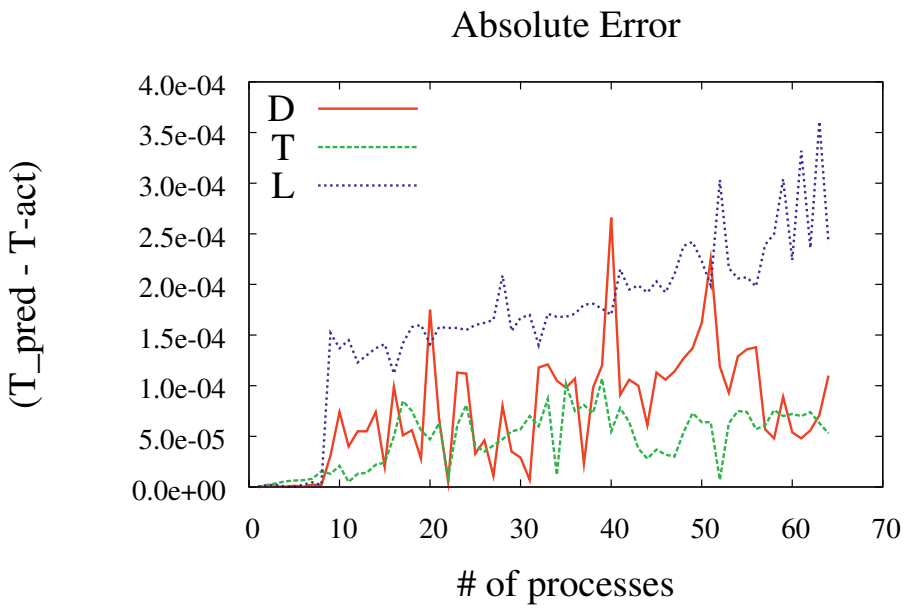

Figure 5.8: Absolute error of prediction/measurement on 8-way 2x4-core cluster
D*: Dissemination barrier*
T*: Binary tree barrier*
L*: Linear barrier*

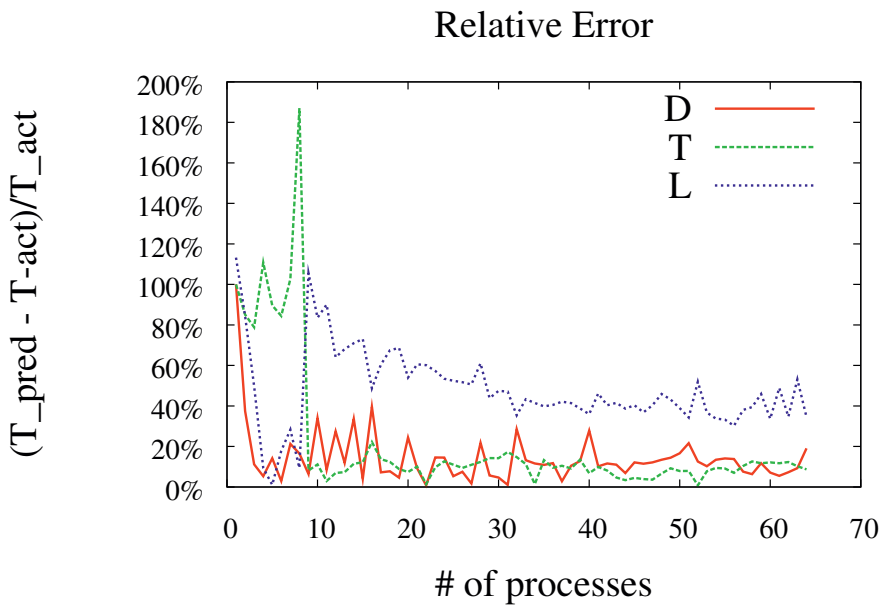

Figure 5.9: Relative error of prediction/measurement on 8-way 2x4-core cluster
D*: Dissemination barrier*
T*: Binary tree barrier*
L*: Linear barrier*

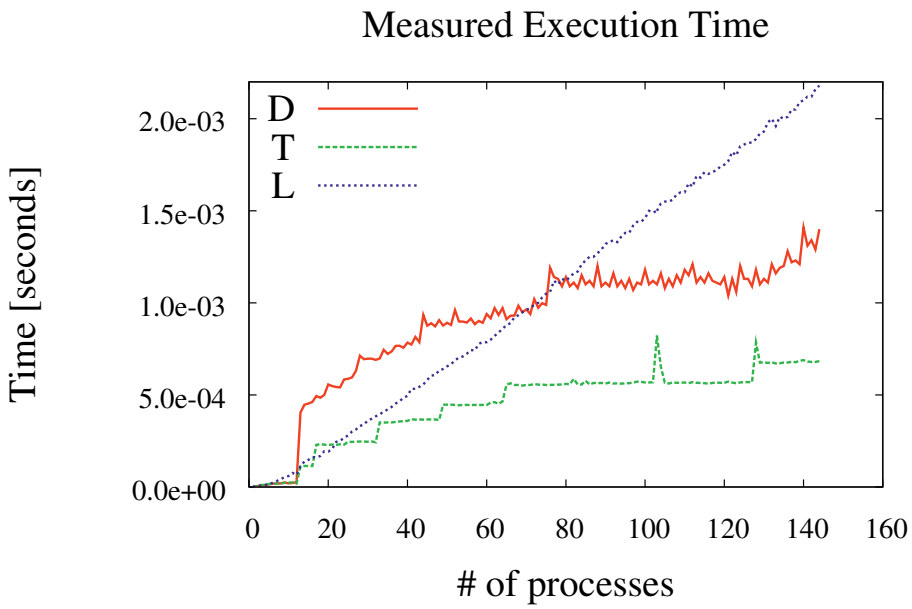

Figure 5.10: Measured barrier timings on 12-way 2x6-core cluster
D*: Dissemination barrier*
T*: Binary tree barrier*
L*: Linear barrier*

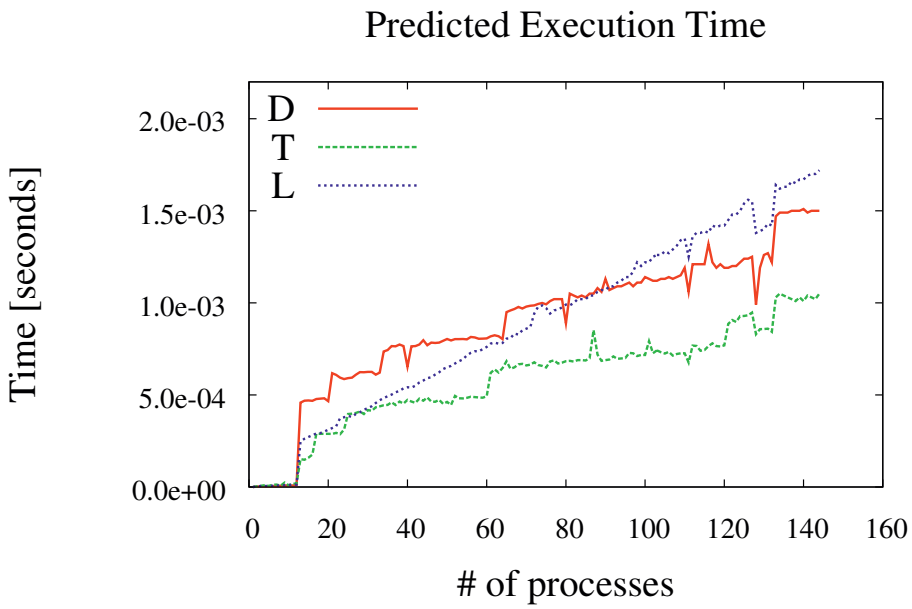

Figure 5.11: Predicted barrier timings on 12-way 2x6-core cluster
D*: Dissemination barrier*
T*: Binary tree barrier*
L*: Linear barrier*

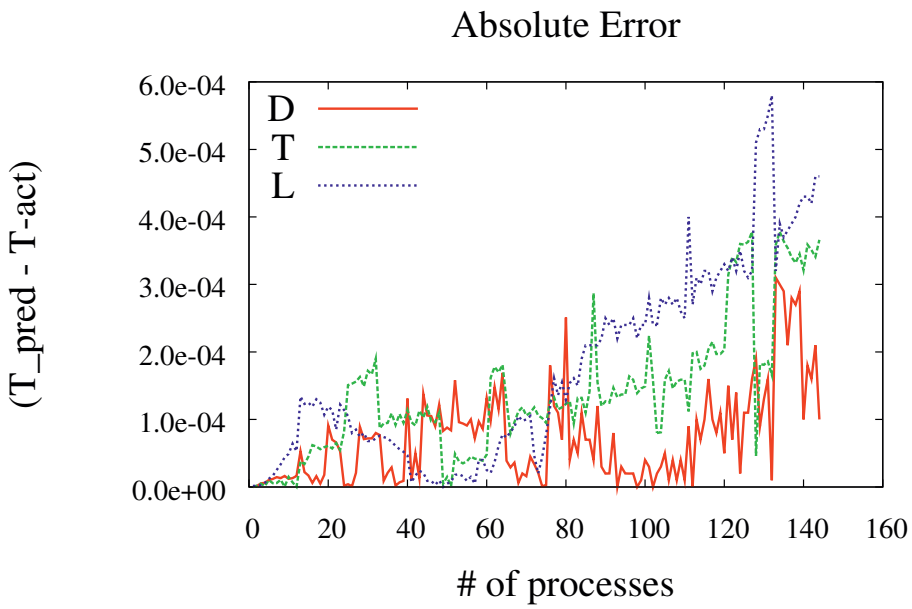

Figure 5.12: Absolute error of prediction/measurement on 12-way 2x6-core cluster
D*: Dissemination barrier*
T*: Binary tree barrier*
L*: Linear barrier*

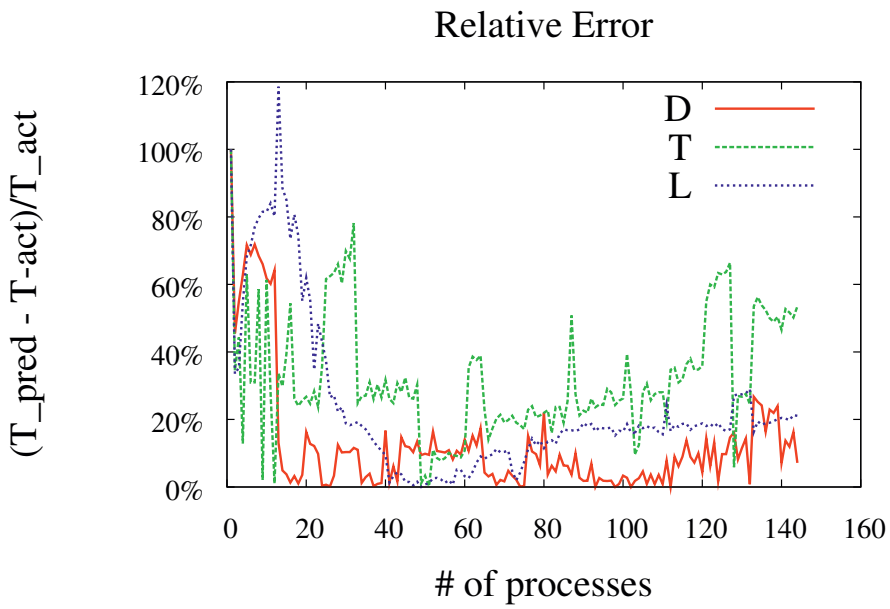

Figure 5.13: Relative error of prediction/measurement on 12-way 2x6-core cluster
D*: Dissemination barrier*
T*: Binary tree barrier*
L*: Linear barrier*

can be used to discriminate between barrier performances at least down to differences of this order. Note also that there appear to be no pronounced artifacts connected to the interplay between process counts which are powers of 2 on this architecture. This is quite expected, as the relationship between local and remote links does not favor powers of 2 when nodes have 2×6 core configurations. The value of making this remark is to support the claim that the model is neutral to the exact topologies of the test systems: the benchmark and test procedures applied on both platforms are identical, save for cosmetic details in small shell scripts to allocate nodes from the system scheduler at initialization.

Before proceeding with a practical application of this latency-centric model, it is pertinent to comment on the magnitude of errors in its predictions. Selecting the execution of a stage as our algorithmic step carries implications for the impact of the inaccuracy when the cost parameters are determined statistically, as nondeterministic variations in both initialization, transmission, and pairwise synchronization contribute to the uncertainty of the extracted statistic. In particular, we approach the cost of a step as a sum of medians from a set of samples, which in turn are sampled from an unknown distribution corresponding to each process pair. As each stage adds the terms for L_{ij} for all targets j, stages with many targets will also accumulate the error this value is subject to, contributing to overall mis-prediction. A similar accumulation results from adding the cost approximation by stages. Our subdivision of the L-barrier into 2 stages exhibits the former problem, as the master rank has $P - 1$ destinations in the last stage; this produces the expectation that the estimation error would grow in proportion to the number of processes. We could imagine extending the L-barrier into $2P$ stages with one signal per stage, a P-stage barrier in which a single signal is transmitted in a ring configuration, or indeed, a single-stage all-to-all barrier with the complete P-graph encoded in a single stage, which would cause the accumulated error to grow as P^2.

Since some growth in the number of stages and/or targets per stage is unavoidable when scaling up, some growth in the error of the prediction must be expected. Trivially, we are also expecting the overall cost of the barrier to display some polynomial growth with P. This makes it relevant to also consider the relationship between these, as it will determine whether predictions will remain approximate to the cost with growing scale. The relative error plot in Figure 5.9 shows that the growing inaccuracy of the predictions for the L-barrier is offset by the overall time consumed by the barrier, leading to improved predictions with upscaling, when considering the error as a fraction of the total cost.

The selection of the three barriers used here is influenced by the requirement of decreasing relative error. Both the fully connected 1 stage barrier, and the $2P$ stage linear barrier have been considered for inclusion here, because they represent extremities in the space of candidate algorithms, in terms of maximizing and minimizing the amount of concurrent communication, respectively. Preliminary experiments, show, however, that not only do these patterns scale poorly as one might expect, but the error of predictions also grows out of control.

The obtained results are omitted, because they display benchmark interference in the algorithm/architecture relationship, rather than the relationship itself. This limitation prompts the question of whether the approach can make justified claim to generality, or whether the

accuracy of the predictions shown here is merely a fortunate happenstance, where the parameters of the particular test platforms admit our model for select algorithms. Addressing this question would require demonstration that improved predictions from the benchmark result in admission of a greater domain of algorithms, but the variability of the benchmark's precision is in itself a composite matter. While some of the observed error may be the result of inaccurate estimation of parameters, the fact remains that the variability of observable parameter values also has an inherently nondeterministic component, which arises from the asynchronous nature of the modeled behavior. Establishing the boundary between inherent variability and that which may be introduced by erroneous measurement would require a similar study to be repeated on an architecture with known limits to nondeterministic variations. Aside from the difficulty of obtaining a platform with reliably documented limits on this parameter, the scope of such a study unfortunately also makes it infeasible to include in this thesis.

Thus, the validation of the methodology presented here is restricted to showing that predictions are accurate to a useful precision within the algorithm design space spanned by our three examples, with the analysis above justifying why this space is limited. Description of the customizations made for the sake of our test platforms is intended as a compelling argument that a similar development may be carried out also for radically different architectures, but pending further work, evaluating the soundness of this argument must presently be left to the discretion of the reader.

Chapter 6

Run-time System and Performance Model

The *BSPlib* programming interface [46] captures the semantics of superstep execution. This interface is a programming model to extend general-purpose languages with a direct mapping onto a theoretical machine which includes both algorithmic and architectural aspects. In the spirit of Valiant's ideal of a bridging model, the simplicity of this interface makes it feasible to implement using a variety of technologies. It is therefore an excellent vessel for exploratory testing, and a most attractive alternative to defining a new programming model for study purposes.

6.1 Overview of BSPlib

The names of the 20 programming primitives of BSPlib are listed in Table 6.1. Exact type signatures and precise semantics are provided by Hill *et al.* [46], but functions are restated here to permit meaningful discussion of their implementation.

The first 7 calls permit initialization, halting, timing, and related services. These are not particularly interesting, except to note that the library expects SPMD style programming, *i.e.* executing the same program file in all processes, and branching individual behavior based on unique process identifiers. The number of processes and the individual identifier are obtained by `bsp_nprocs` and `bsp_pid`, respectively.

The `bsp_sync` function is a barrier synchronization, which also enforces that all changes to remote memory have been carried out globally before it proceeds.

The `reg`, `put` and `get` functions facilitate fetching and retrieving values from remote memory, with the `hpput/get` varieties featuring relaxed requirements on buffering at both ends for performance reasons. The `push/pop_reg` functions embody the method by which this one-sided communication is made transparent to the distinction between

Table 6.1: BSPlib programming primitives

Function call	Operation
`bsp_init`	Initialize a parallel program
`bsp_begin`	Begin parallel execution
`bsp_end`	Halt parallel execution
`bsp_abort`	Abort execution with an error state
`bsp_nprocs`	Find number of processes
`bsp_pid`	Find index of this process
`bsp_time`	Report time
`bsp_sync`	Synchronize, finalize superstep communication
`bsp_push_reg`	Register memory area
`bsp_pop_reg`	Unregister memory area
`bsp_put`	Place data in buffered remote memory
`bsp_hpput`	Place data in remote memory
`bsp_get`	Fetch data from buffered remote memory
`bsp_hpget`	Fetch data from remote memory
`bsp_set_tagsize`	Set size of message tags
`bsp_send`	Send tagged message
`bsp_qsize`	Report number of received tagged messages
`bsp_get_tag`	Read the tag of first tagged message
`bsp_move`	Fetch buffered contents of tagged message
`bsp_hpmove`	Fetch reference to buffered tagged message

shared and distributed memory spaces, through the requirement that memory areas which are the targets of `put` and `get` operations are registered by each process. Thereby, programs can refer to a memory allocation by a consistent reference, and disregard differences in the exact placement of this reference between processes.

The remaining routines provide more conventional message passing, through allowing processes to place messages of arbitrary size in a *queue* maintained by each process, and act on the contents of these through examining a fixed-length *tag*. The only significant requirement placed on this messaging facility is that the tag size is determined collectively, per superstep.

While this brief discussion of the library interface is far from complete, it should serve to show the potential for addressing the overlap term in the extraction of a performance model. If we presume that the communication required by the `put`, `get` and `send` primitives can be overlapped with computation, the time required to realize them before the next superstep commences must be bounded by the computations performed in the meantime. Otherwise, the semantics of the barrier require that it delays further execution until communication is complete. Provided we can establish a strong, quantitative expectation of the cost of synchronization, deriving the extent to which communication has been masked by computation is a matter of determining the source of any significant deviations at synchronization time.

6.2 One-sided Communication

The implementation developed for the purposes of this study is composed from the abstractions provided by the MPI and POSIX threading interfaces, coupled with a small amount of system-dependent code which is required to control process locality.

The fundamental mechanism by which the library operates, is to separate the responsibility for executing user code from administrating communication, by spawning a separate thread to be responsible for the logistics of communication. This requires the run-time library to track the state of both the communication which is specified as part of the application, as well as a set of internal control messages required to retain consistency with BSP semantics. Specifically, it implements one-sided communications using the non-blocking varieties of MPI point-to-point communication calls, necessitating a header message to be transmitted for each *put* and *get* operation, containing a description of the type of operation, the initiating process identifier, the remote address, and message size. The BSPlib requirement that remotely accessible memory areas are registered at least one superstep prior to their use, permits remote addressing to be made relative to a reference to the registration rather than to an absolute memory location. In order not to burden the programmer with managing the indexing of these registrations, the *push_reg* and *pop_reg* operations are implemented using two queues of local pointer values and registration indices to track registrations throughout a superstep. Their contents are committed to a hash table at synchronization time, which is keyed on the value of the local pointer. As the C standard library lacks a sufficiently flexible hash table interface, *libghthash* [57] is employed.

The user program can thus interact with the run-time library by referring to the names of (registered) local pointers when referring to operations on remote memory, resulting in a software-implemented distributed shared memory programming style.

In detail, a header message is a tuple of 6 integers:

- Signal type, to identify the cause of communication
- Remote process id
- Reference to buffer registration, from which local target pointer can be identified
- Offset from the registration's base address
- Length of the following payload
- A sequence code to identify the corresponding payload message.

These control messages are the only data directly manipulated by the communication thread, as it only initiates the nonblocking requests which indirectly influence memory contents. Thereby, the completion of all these requests is not an issue until the computation goes into synchronization, at which point the `bsp_sync` function can await the completion any outstanding messages, and act appropriately on their contents in correspondence with the semantics of the functions. At synchronization time, the communication thread can be blocked, so that requests and message contents can be manipulated without race conditions. Aside from this, all necessary interactions with the MPI library are contained within the communication thread, effectively making it appear as an independent background communication system which the main program can activate through local signals. Its state is kept coherent with the computation thread using thread condition variables and locks.

Library semantics demand that the effects of committed communication calls during a superstep take effect in the next. With fully asynchronous transmissions, this requires that the communication thread is made aware of the number of transmissions in progress at synchronization time. While the reception of a header message can initialize the reception of the corresponding data transfer, any number of header messages may still be in transit when the user program calls for synchronization. This is addressed by each process maintaining a local table with *outgoing* message counts sorted by their destination, and gathering a complete map of the communication pattern at synchronization. The total exchange required to create this map is implemented by exchanging the message count map as a small data payload transmitted with the signals throughout the stages of the synchronizing barrier, leaving each process with the option to await any outstanding messages before completing synchronization.

The separation of responsibilities between the user code and communication thread is also implemented in a nonblocking manner: apart from initialization and finalization, the interaction between the functions which are exposed through the programming interface and functions which manipulate internal state is carried out using a set of message queues exclusively. Messages are atomically enqueued by the computational thread, and only dispatched by the communication thread when it is next scheduled by the operating sys-

tem. This is done in spite of the fact that the communication and computation thread share an address space, suggesting that a more direct signalling mechanism might carry a lower overhead. The reason for this choice is that MPI implementations are under no requirement to provide thread-safe functions. As the above discussion indicates, the implementation is required to interleave the resolution of point-to-point communications with collective operations, which makes it impossible to provide a robust implementation without guarantees of either atomic manipulation of MPI-internal data structures, or the presence of non-blocking collective operations.

6.3 Thread Scheduling Considerations

Implementing communication as a separate thread coupled with the process affinity required for predictable communication cost necessitates an acknowledgement of the impact of the scheduling policies in the underlying operating system. Locally queueing messages for dispatch by an independent thread holds the potential of delaying their actual transmission until the communication thread is next scheduled. As the affinity of a process to a processor effectively means that the two threads time-share the physical processor, this creates the possibility that the compute thread can starve out the communication thread, postponing message transmission until either the end of a time slice, or synchronization time. Both possibilities are detrimental to our purpose, as initiating communication as early as possible is the exact purpose of every other trade-off chosen in implementation.

This issue is resolved by relying on an assumption that the process consists of exactly these two threads, and that they are the primary sources of contention for the processor. These assumptions permit the POSIX *sched_yield* call to be used as a switching mechanism: its function is for the calling thread to relinquish the processor, permitting another thread to take control. The function which enqueues a remote access for transmission yields the processor directly thereafter, while the communication thread yields control after processing at most one outgoing and one incoming message. Aside from the consideration that such a method would admit an unrelated, computationally intensive thread to delay execution beyond the ordinary impact of system jitter, it also implies that outgoing communications will be effected immediately, while reception may be delayed. The communicator thread works by maintaining a non-blocking reception of a header message from any source at all times, and dispatching a non-blocking reception of the corresponding data transmission as soon as a header message is handled. The balance of how many messages to handle per thread context switch can be easily adjusted programmatically, but in the present implementation, this is limited to one in the interest of keeping the overhead of each invocation of the communication thread low, and from the consideration that an environment with buffered communications reduces the performance impact of delayed reception.

6.4 BSP Barrier Communications

The efforts to develop a cost function which relies on communication patterns presented in chapters 5 and 7 is not restricted to barrier synchronization, but could arguably be extended to general rootless collectives. The restricted requirements of the total exchange which our library needs to build its message map at synchronization time, means that it will suffice to construct performance models which express the consequences of adding a modest bandwidth requirement to the latency-centered model we have already developed.

Before exploring the addition of a performance parameter, it is important to revisit the purpose of examining barrier synchronization in the context of our objectives. The overall point of the performance model is not to create a general model of the communication cost of collective operations, but to determine the time required to synchronize a computational superstep with no outstanding background communication. Taking Bisseling's approach of applying all-to-all collectives to this end [19] would make a more general development appropriate, but as we already witnessed in Figure 5.13, the accuracy which the commodity component test systems afford is showing signs of tension between neutrality to collective patterns and predictive power already when approaching 120 processes.

Taking into account the desire to develop a non-blocking implementation of the BSPlib interface, it is clear that complete generality is not required to establish a baseline for barrier cost. A bandwidth parameter must nevertheless be introduced, because of the need to provide one-sided communication calls layered over MPI's basic point-to-point communications. The reason for this is quite simply that in order to determine when synchronization can be completed, a process which is the recipient of one-sided communication must be able to determine that it has received all outstanding communication, including any messages which have not yet been detected. This places a minimum extra requirement on a suitable barrier mechanism, which is that as a side-effect of synchronization, a global map of the number of messages between participants must be established. From an asymptotic complexity point of view, this might be taken as an argument that the difference in communication schemes is of little interest, because synchronization is ultimately bounded by all-to-all collective performance under either variety. A more detailed picture forms when considering the magnitude of the associated bandwidth requirement, by noting that the cost of the all-to-all collectives obtained by using it as the only means of communication is a function of the data volume committed for communication during the superstep. The nonblocking alternative, however, yields a bandwidth requirement which is strictly a function of the number of processes, making the synchronization cost an architectural feature which can be obtained separately from considerations of the deployed program.

Because we layer BSP communications above MPI, it is reasonable to question the necessity of the level of detail with which the point-to-point cost function has been treated. The end result is a functional equivalent of `MPI_Alltoall` with a payload of P integers, and using this would abstract the implementation details of the collective operation. The reason for not leveraging already implemented collectives is more technical than a scientific: it stems from the fact that in order to conform to specifications when using a communications layer without guarantees on the ordering of messages, our implementation must

interleave the handling of communications initiated by a user program with the handling of its internal control messages.

Introducing a message payload which modifies the cost of communication introduces the requirement that a process chosen to as representative for a subset in a hierarchical pattern not only has the responsibility for distributing acknowledgement signals, but also must accept the payload for each of the processes it represents, partition it, and communicate it to them. In a general context, this introduces the need to adapt the message pattern not only to the partitioning of processes, but also to the consequence this has for the bandwidth requirements of the associated algorithm. While such an extension is doubtlessly possible, it represents a non-trivial programming effort.

In order to restrict the scope of this analysis, the extension of the barrier algorithm into a fixed-size all-to-all communication will focus on the communication pattern of the dissemination barrier. While the studies of purely latency-bound cost functions has given evidence that this barrier is not an optimal choice in terms of its cost, the pattern it employes carries the significant advantage that it is a simple structure to facilitate all-to-all communication; indeed, its name derives from its usefulness as a method for disseminating global information, *i.e.* establishing a synchronized state. Because we are after a *predictable* cost function as much as an optimal one, this tradeoff between cost and complexity is made for the sake of enabling workable test programs with reasonable effort. The substitution of an equivalent, more efficient barrier pattern carries no further implications than an updated cost function.

6.5 Performance Model Extensions

The outlined implementation attaches a programming interface to the performance parameters considered so far. This enables us to adopt the revised processing model, and relate its application-independent parts to performance model components. While computation kernel details by necessity depend on the application, the architectural communication parameters of pairwise latencies and inverse bandwidths relate both to synchronization cost and general purpose communication, as highlighted in Figure 6.1.

To analyze the bandwidth requirement with respect to a dissemination barrier pattern, recall that the pattern corresponds to $\lceil logP \rceil$ successive transmissions along the axes of a hypercube. Each process will be required to receive a vector of integers to its neighbor along a first axis, and correspondingly send one of twice the length to its neighbor in the next dimension. The doubling of the payload follows from the fact that the neighbor on the second axis is not directly connected to the one on the first, so the received information must be transmitted along with the process' own contribution. Proceeding in this manner doubles the payload for each successive stage, until the last, which comes at a cost of $P - 2^{\lceil logP \rceil - 1}$ when P is not a power of two. After these $\lceil logP \rceil$ successive doublings, each process has received a full P^2 map of integers, corresponding to the one-sided messaging pattern. Receiving the vectors sequentially will permute them into an order which depends on the relation between the local process identifier and P, but the effect of the pat-

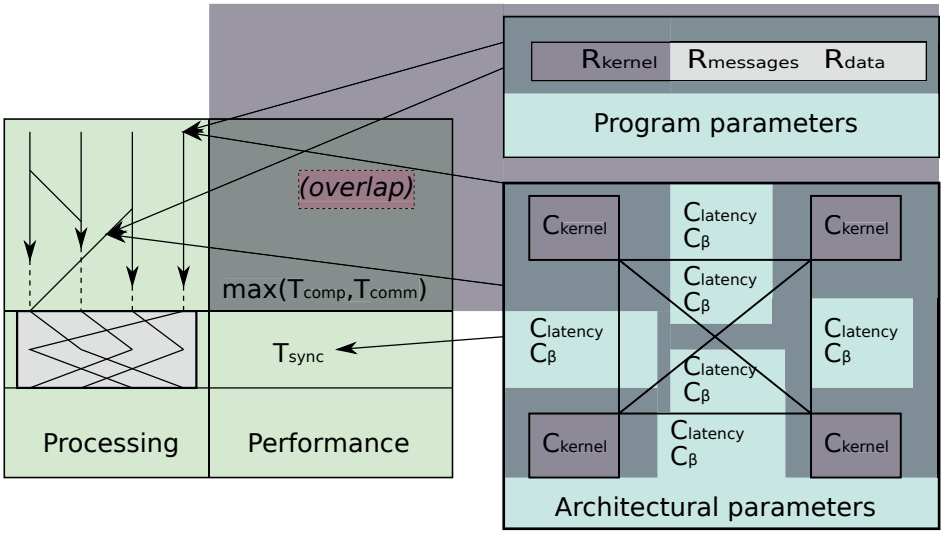

Figure 6.1: Relation of communication parameters to processing and performance models

tern is easily reversed given its regularity, resulting in a message map in natural order. The progress of the stages in the pattern can be traced internally by the communication thread, which is all that remains after computation has reached synchronization. Its completion implies that each process has reached synchronization, which means the communication thread can be suspended before the effects of communication are made final.

For the communication thread to correctly interleave barrier-related messages with any regular, outstanding traffic from the preceding superstep, barrier messages are implemented using the same header message structure as point-to-point communications. Both the size of the payload, and the location of the reception buffer are independent of user program requirements, and can be deduced from the internal state of the barrier. Therefore, the header messages of barrier related communication can be restricted to transmission of 4 integers.

The benchmark needs some modification in order to adapt a cost model for this operation. As described in Section 5.6.3, capturing the parameters for the latency-bound cost already includes a pairwise bandwidth test, performed in order to find its intercept for negligible messages. In the same test, the regression line which is produced gives an estimate of the pairwise inverse bandwidth, *i.e.* the cost of communication per byte. Storing this metric in a $P \times P$ matrix which can be weighted by the transmission cost per barrier stage is straightforward.

The assumption of symmetry which permitted a simplification in the treatment of the latency cost can no longer be used effectively, as the weight of the doubling of communication volumes between sending and reception skews the communication pattern for any topology. In order to compute predictions for the synchronization cost of the library barrier, a small program is utilized to weight the communication pattern by the bandwidth

```
double
barrier_predict ( barrier_t *b, int r, int stage )
{
    int logP = ceil ( log(size) / log(2) );
    int send_ints = (stage < (logP-1)) ?
            (1<<stage)*size : (size-(1<<(logP-1)))*size;
    int recv_ints = send_ints;
    double
        extreme = 0.0,
        overhead = ts_matrix[r*size+r],
        latency = 0.0, bandwidth = 0.0, headtraffic = 0.0;

    if ( stage != logP )
    {
        for ( int j=0; j<size; j++ )
        {
            if ( PATTERN(b,stage,r,j) > 0 )
            {
                // Send cost: 2 messages ( head & data )
                overhead = MAX(overhead,ts_matrix[j*size+r]);
                latency     += 2.0 * tl_matrix[r*size+j];
                headtraffic += 4 * sizeof(int) * tb_matrix[r*size+j];
                bandwidth   +=
                    4 * send_ints * sizeof(int) * tb_matrix[r*size+j];
            }
            if ( PATTERN(b,stage,j,r) > 0 )
            {
                // Receive cost: 2 messages ( head & data )
                latency     += 2.0 * tl_matrix[j*size+r];
                headtraffic += 4 * sizeof(int) * tb_matrix[j*size+r];
                bandwidth   +=
                    2 * recv_ints * sizeof(int) * tb_matrix[j*size+r];
            }
        }
        extreme = 0.0;  // Estimate worst case
        for ( int j=0; j<size; j++ )
        {
            if ( PATTERN(b,stage,r,j) > 0 || r == j )
            {
                double test = barrier_predict ( b, j, stage+1 );
                extreme = MAX(extreme,test);
            }
        }
    }
    return extreme +                              // Rest of barrier
           overhead + latency +                   // Startup terms
           ((bw) ? headtraffic+bandwidth : 0.0); // BW term (optional)
}
```

Figure 6.2: Recursive Critical Path Search

requirements, and recursively traverse it in search of the critical path.

The program fragment which implements this recursive traversal is given in Figure 6.2. Some externally declared entities are omitted for the sake of brevity, specifically, the `barrier_t` structure, a boolean flag `bw`, and the `ts`, `tl` and `tb` matrices. The latter three are $P \times P$ matrices of startup overhead, wire latency and inverse bandwidth terms, read from a file emitted by the benchmark program used in Section 5. The `barrier_t` structure is used for containing the communication pattern incidence matrices, permitting the `PATTERN` macro to index them by barrier structure, process identifier, and barrier execution stage. The `bw` flag is a simple command line parameter which indicates whether or not the accumulated sum should include the communication payload terms, or only account for overhead and latency.

A few points in Figure 6.2 warrant further comment. The primary extension of the previous cost function is the separate treatment of sending and receiving costs, where only the send cost includes the overhead of initiating transmission. This distinction is made because the initialization of the internal structures of the barrier state already register non-blocking receptions necessary for all stages of a barrier execution, which means that this cost is not incurred at the time of synchronization. A second point is that the per-stage cost of the send transmission is twice that of the reception. This corresponds to the successive doubling of the communication volume, discussed in the previous section. Finally, the per-integer bandwidth cost is doubled once more as a simplification of the cost of internal structure manipulation. This stems from the fact that the dissemination of the message map through the pattern of the barrier requires each individual process to reorganize the map with respect to its position at completion. Modeling this activity in detail would require separate benchmarking of memory movement, which would complicate cost estimation. Instead of elaborating upon this point, we make the observation that the cost is analytically $O(P^2)$, and can thus be expected to be proportional to the aggregate transfer cost. As we already have a cost term bound to this magnitude, the expectation that inter-process communication will be at least as expensive as the intra-process variety implies that doubling this cost will provide a relaxed upper bound on the additional cost. This simplification is made with the awareness that the relative magnitude of the constants involved is likely to vary, and present a significant source of error at extreme scales. For the experiments at hand, however, this simplified model is sufficient to obtain a satisfactory approximation to empirically observable synchronization cost.

In order to complete the method for predicting the observable performance of the library synchronization primitive, the existence of one final term needs to be described. While the use of the dissemination barrier pattern completely distributes the message map, and thereby implicitly guarantees a global state, the operation of the communication thread must be suspended when the map is completed, in order to prevent potential race conditions against the reception of any outstanding messages from the completed superstep. In order to retain consistency with superstep semantics, any such delayed communications must be effective before the transmissions of the *next* superstep are initiated, lest we leave the possibility that a process with no outstanding communication prematurely sends data to another process which is stuck waiting for another message. This requires a second, internal barrier to complete the `bsp_sync` function, before the internal structures of the

communication thread are reinitialized for the next superstep. As that requirement can be fulfilled without any data transmission, and the communication thread of any process which reaches this stage is dormant, the regular `MPI_Barrier` function can be used for this purpose. In order to cost this extra work, the cost function for an additional tree barrier is added to the prediction obtained by the function in Figure 6.2. As was seen in Chapter 5, this cost function bears a strong similarity to the observable cost of the MPI barrier primitive.

The decision to employ the MPI barrier for this purpose instead of adding one of the profiled patterns is made in order to restrict program complexity. While the implicit reliance on the MPI barrier being implemented with a tree pattern is detrimental to portability, it should be noted that the matter *can* instead be addressed by using a more accurately profiled barrier algorithm. The approximation made here is motivated purely by practicality, as the term in question neither dominates the synchronization call, nor renders the resulting estimate unusable.

6.6 Empirical Validation

Having charted the communication requirements for a dissemination barrier as used for a nonblocking BSPlib implementation, its accuracy can be investigated using a full, working library implementation. Although the favorable generality of a simulated scenario might add further enlightenment to the discussion, it would be beyond the scope of this work, as its implementation and evaluation would require extensive implementation efforts which are not necessary to cost synchronization overhead as experienced by the programs we will develop using the dissemination barrier implementation.

Because the requirements we have analyzed only describe the expected cost due to communication, it remains to account for the additional overhead of buffering the partial message maps and sorting the result in natural order. As all this additional overhead will apply to every transmitted element, we expect that this cost may be approximated by a simple scaling of the cost function per datum, *i.e.* that the shape of the predicted performance graph will differ from observations by only some small constant factor for the number of integers transmitted.

Figures 6.3 and 6.4 show obtained performance figures and standard deviations, in comparison with predicted values.

In summary, we have established a statistic which can be used to provide a bound on what delay can be expected from the pure synchronization overhead in our nonblocking BSPlib implementation. We also have an experimentally established estimate for the magnitude of the common degree of variability. The synchronization overhead obviously plays an important role in the establishment of performance expectations for given applications. The degree of variability may to some extent be governed by inaccuracies in the benchmarking procedure, but the sensitivity of synchronization to external influences suggests that there will be an inherent variability to implementations on platforms which do not give exclusive and complete control to user programs. Like the observations of the impact of variability

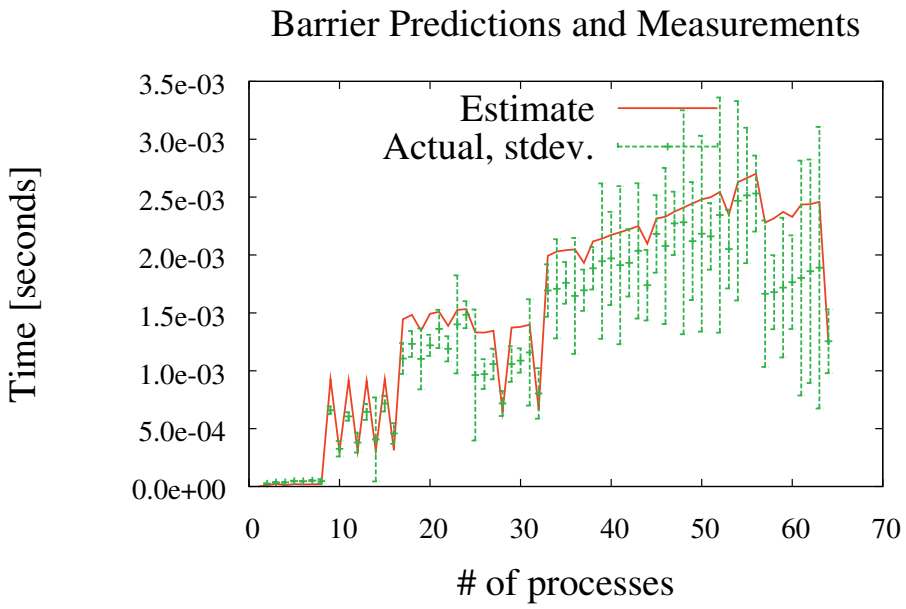

Figure 6.3: Measured barrier timings and estimate on 8x2x4 cluster
Estimate*: Prediction computed by the program in Figure 6.2, with platform benchmark results as input*
Actual, stdev*: Empirical barrier timings of the augmented dissemination barrier of the BSPlib implementation, with sample std. deviation.*

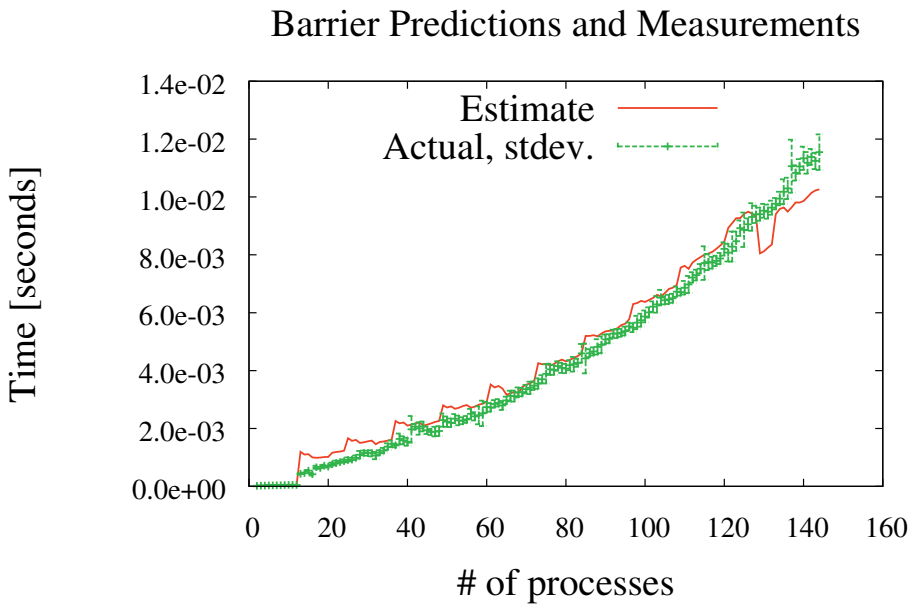

Figure 6.4: Measured barrier timings and estimate on 12x2x6 cluster
Estimate: Prediction computed by the program in Figure 6.2, with platform benchmark results as input
Actual, stdev: Empirical barrier timings of the augmented dissemination barrier of the BSPlib implementation, with sample std. deviation.

in computation rate over longer stretches of time, this suggests that estimates of execution time will deviate in proportion to the error estimates over longer stretches of time, leaving the accuracy of predictions limited to attaining a stable or limited *relative* error, subject to the extent of the sets of observations taken to be representative of application behavior.

Chapter 7

Case Study I: Adaptive Barrier Implementation

Our stated purpose is to examine the capabilities model components provide for automating performance analysis and tuning. As Chapter 5 shows, the subset of pairwise latency parameters and message counts suffice to capture the cost of various signal patterns for synchronization. The separation of concerns between application communication and synchronization cost thus suggests that automatically refining synchronization algorithms makes a useful application. Figure 7.1 highlights the relevant components in their context.

The latency-centered cost function derived in Chapter 5 is of limited utility in that it only accounts for the cost of transmitting messages of negligible length. Its ability to discriminate between the cost of equivalent patterns, as well as subject the patterns themselves to algebraic manipulation still suggests that it may be employed to tailor barrier synchronization for given topologies. This chapter explores that possibility, obtaining results which indicate that the level of precision is sufficient to apply our framework to practical scenarios. It provides an example of how sufficiently precise performance models can extend beyond the ability to estimate programmatic and architectural complexity, and also be leveraged as a design tool.

The method presented in this chapter was published as part of the HCW workshop at the IPDPS'11 conference [75], featuring early empirical results. Aside from a more detailed discussion, the result material has been slightly extended.

7.1 Barrier Combination

The incidence matrix representation of barrier algorithms provides a finite design space of algorithms to explore. By evaluating the cost of any given algorithm for a fixed P on a benchmarked architecture, we can find a tentative upper bound on execution cost. This

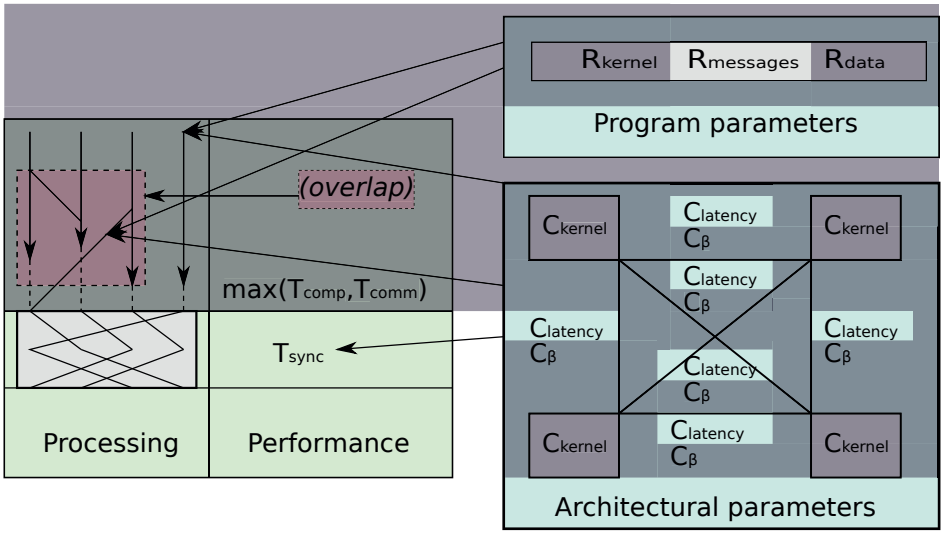

Figure 7.1: Model components of synchronization cost

excludes algorithms of arbitrary stage counts, because any stage in our cost model includes a small overhead term even when no communication is required. With a finite bound on the stage count, indicating the arrival of one process to another more than once introduces a cost which accomplishes nothing in subsequent stages, so candidate graphs should include at most one edge in each direction between a given pair. In principle, this bounds the space of admissible algorithms to a bit pattern which is $O(P^2)$. Thus, a brute-force method may in principle examine all candidate bit patterns, and apply Equations 5.1 and 5.2 to filter the patterns which do not encode synchronization.

Although this would require exponential time in the length of the bit pattern, the small cost of running one such test might permit it to scale to modest problems. Our test platforms make a different concern more prominent, because even though the barrier encoding may have general validity, the limited domain of the cost function indicates that even optimizing the search to generate only valid algorithms in sequence, the criterion for selecting the better out of a pair of candidates may not match empirical results. Therefore, we will restrict the search to candidate algorithms which can be generated by combining the three algorithms we have evidence that the cost function is accurate for.

A method for creating combinations of the D, T and L barriers can be found by revisiting the intentions of their construction: the L and T barriers distribute signal and acknowledgement hierarchically, while the D barrier trades the cost of the acknowledgement signal for the full participation of every process. Either algorithm can naturally be employed to guarantee the synchronization of a subset of processes, so by taking the hierarchical idea from the T barrier, the responsibilities of collecting an arrival signal from a subset and broadcasting the acknowledgement can be fulfilled by a smaller barrier with only the members of that subset. The number of participants which makes it informative to inspect such a

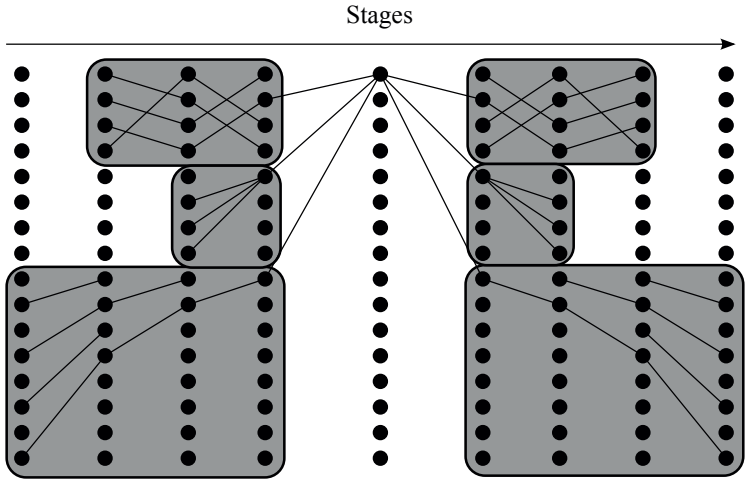

Figure 7.2: Hierarchical hybrid barrier with marked subsets

derived algorithm also makes it inconvenient to read and verify in the form of incidence matrices. Instead, Figure 7.2 shows a dependency graph layered in stages, where a set of 16 processes are divided in subsets of 4, 4 and 8 processes, using the D, L and T barriers respectively. The subsets are synchronized by representatives which use the L-barrier at a higher level of the hierarchy, and the acknowledgement is propagated back along the same paths. Creating instances from this family of patterns, it is feasible to test an extensive number of hybrid varieties in negligible time. Although this approach ignores a great number of candidate algorithms which are *not* combinations from our selection, the variety in the tested group displays enough variation to demonstrate the principle.

7.2 Determining Subset Sizes

While restricting the range of algorithms to test per subset restricts the number of candidate algorithms to examine, selecting an appropriate partitioning of the participating processes is another point where the number of combinations grows exponentially. We have already developed the groundwork for a useful heuristic to restrict this, modeling cost as a function of the topological distance between process pairs. Assuming that the exploitation of locality gives a natural way to partition the interconnect, finding suitable subsets becomes a matter of discovering the subsets with similar signalling costs in the platform benchmarks. Obtaining this information using minimal *a priori* platform knowledge creates a clustering problem of modest extent.

To identify a suitable method, observe that the weight matrices imposed on the dependency graphs in the cost function form a weighted, fully connected graph, where weights capture the proximities of processes to one another. The general assumption that we have captured costs which are somehow proportional to physical locality makes it reasonable to expect

that these form a *metric space*. In terms of the model, we assume three conditions:

1. The cost of a signal $i \to j$ is 0 if and only if $i = j$.
2. The combined cost of signals $i \to j$ and $j \to k$ is at least equal to the cost of $i \to k$.
3. The cost of $i \to j$ is equal to that of $j \to i$.

Condition 1 is trivial for $i \neq j$, insofar as the regression lines from varying message size and signal count both have positive intercepts. While an unfortunate benchmark sample can cause this assumption to fail, it would be inappropriate to rectify that by altering the clustering algorithm, as it would be a symptom that the cost function is inadequate.

The case of $i = j$ requires further examination, as the cost model implies a small expense per stage even for a process which does not communicate anything, as nodes in the dependency graph technically reflect process *states* rather than processes. The distances given by the weights thus depend on time as well as space, *i.e.*, an accurate representation separates the moment a process sends a signal to itself from the moment when it is received as two graph nodes. In our case, we carefully disregard this issue by noting that to a clustering algorithm, the "before" and "after" states of a single process create node pairs far more tightly coupled than any other pairs in the system. By paying attention to the threshold for grouping points, a process is thus considered to have a signal cost distance to itself which is indiscernible from 0, and thereby fulfill the requirement of condition 1.

Condition 2 captures the assumption that we cannot expect to lower the cost of a chain of signals by introducing more steps, which holds if the captured cost function is indeed proportional to physical distances.

Condition 3 requires some extra attention, as it is not true for asymmetric interconnects. Considering the cost matrices of our test platforms as metric spaces presents no problem, but applying the approach we describe to an interconnect such as a unidirectional ring topology would require reassessment of how to partition the topological layout.

The problem of discovering platform structure from the cost function brings out two important properties of the topology graph:

- The number of clusters is not an input parameter, as it reflects platform structure.
- The space has no inherent notion of center points or origin, as all distances are relative.

Unfortunately, these points make the popular *k-means* algorithm [68] and its derivatives unsuitable for clustering graphs such as ours. This is because it relies on the choice of a number of k cluster centroids in order to obtain its partitioning. Using such a solution would not only require the somewhat cumbersome extra work of consistently projecting the graph into Cartesian coordinates, but would also limit adaptivity through requiring the number of clusters to be predetermined. In our context, this either requires assuming the number of local areas the processes should be partitioned into, or examining the outcome of clustering for a range of k and selecting the best fit.

Sparse Spatial Selection [25] makes a better choice, as it works by limiting the distance

between cluster members, rather than the number of clusters. The procedure is to maintain a set of pivot nodes, and introduce new nodes by comparing their distance from existing pivots to the diameter[1], scaled by a constant *sparseness* parameter. New pivots are created when the introduced point is too far from all existing pivots to include in an existing cluster. This iteration converges to a fixed point, and applies recursively to the obtained clusters, dynamically obtaining a map according to how scattered points are relative to each other. Its most significant assumption is that cluster pivots will be distributed sparsely compared to the diameter of the clusters themselves, which fits our purpose of discriminating between localities when measurements indicate that they are remote from each other.

Choosing this algorithm is the reason for accepting the symmetry requirement of a metric space. Requiring a constant ratio between the diameter of a cluster and the diameter of the system creates a potential problem because it may ambiguate the classification of a point with respect to pivots. The fact that a candidate barrier algorithm implies a *directed* graph can be leveraged to resolve this problem, by adapting the clustering method according to the direction of communication of the algorithm being evaluated. Because it does not directly influence work on our test platforms, this will not be considered further here.

Determining the sparseness parameter requires deciding on the desired influence of relative distances. Since the distinction between local and remote communication featured on our test platforms spans orders of magnitude, it can be expected to appear within a liberal range of values. Brisaboa *et al.* [25] suggest that values in the range $[0.35, 0.4]$ are appropriate for similarity based search applications. Since these applications apply to spaces which are large enough that it is desirable to *estimate* diameters rather than obtain them from comparing all pairs, the reported efficiency benefit is of no consequence to our work, but testing with parameters in this range certainly does not introduce any overhead.

Experimental verification of the clustering with the overhead matrix O as a metric, and using a sparseness parameter of 0.35, confirms that the approach correctly recognizes the topology of distributed and shared memory. The distinction between on-chip and off-chip communication at the node level is observable in the range $[0.11, 0.12]$, but the inaccuracies of the benchmark procedure made it difficult to produce a consistent partitioning at this level. This does not present any problem with respect to applying the result, so much as it verifies that the distinction between nodes is rather more pronounced than node-local variations on our test architectures, and as such makes a suitable target for optimization.

The process identifier clustering from the experiment with sparseness 0.35 is tabulated for 60 and 115 process cases in Tables 7.2 and 7.1. The information contained in these tables is hardly surprising, but their value is that they are generated from statistics on network performance, rather than knowledge of platform structure. The metric clearly captures the shared/distributed memory distinction robustly, and the platform schedulers' different mapping of processes by node or round-robin allocation can be seen by their orderings of process identifiers.

The outcome of this clustering can be reproduced by specifying the topological layout manually, or by programmatically mapping host names and core numbering. The purpose

[1] *i.e.* the longest distance separating any two points

Table 7.1: Output of 60-process SSS clustering on 8x2x4 node configuration

Cl.#								
1	000	008	016	024	032	040	048	056
2	001	009	017	025	033	041	049	057
3	002	010	018	026	034	042	050	058
4	003	011	019	027	035	043	051	059
5	004	012	020	028	036	044	052	
6	005	013	021	029	037	045	053	
7	006	014	022	030	038	046	054	
8	007	015	023	031	039	047	055	

Table 7.2: Output of 115-process SSS clustering on 10x2x6 node configuration

Cl.#												
1	000	001	002	003	004	005	006	007	008	009	010	011
2	012	013	014	015	016	017	018	019	020	021	022	023
3	024	025	026	027	028	029	030	031	032	033	034	035
4	036	037	038	039	040	041	042	043	044	045	046	047
5	048	049	050	051	052	053	054	055	056	057	058	059
6	060	061	062	063	064	065	066	067	068	069	070	071
7	072	073	074	075	076	077	078	079	080	081	082	083
8	084	085	086	087	088	089	090	091	092	093	094	095
9	096	097	098	099	100	101	102	103	104	105	106	107
10	108	109	110	111	112	113	114	115				

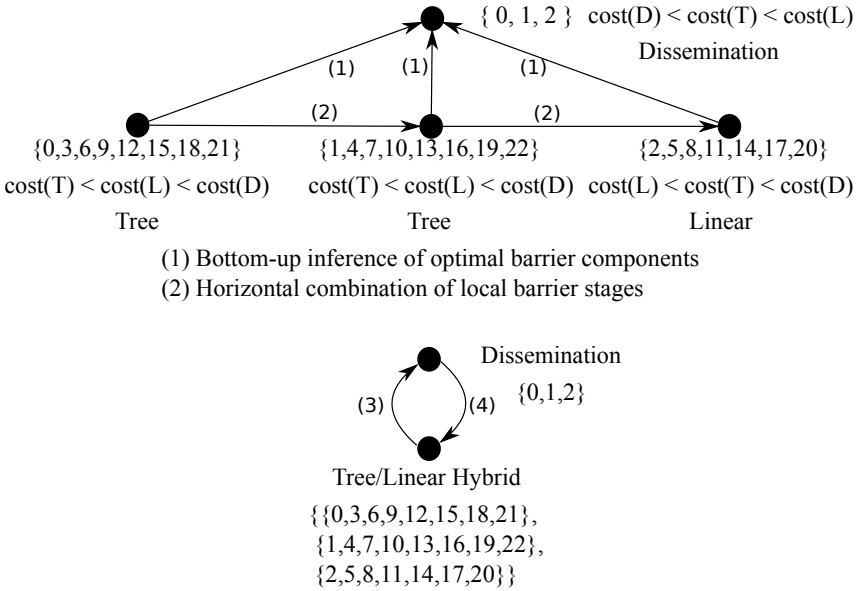

Figure 7.3: Greedy construction of a hierarchically clustered, customized barrier

of this elaborate topology partitioning method is twofold. Firstly, it automates experiments so that human error or platform idiosyncrasies only affect results through reduced benchmark accuracy. Secondarily, it demonstrates that benchmarks contain implicit information which permits programmatical inference of structural information about the execution environment. The significance of this latter point will be developed in the following section.

7.3 Greedy, Adaptive Barrier Construction

Restricting the candidate algorithms and defining the partitioning of the processes suggests a small enough space of combinations to explore in greater detail, using the assumption that combining locally optimized methods will result in improved global behavior.

This results in a greedy algorithm in two phases. An example of its operation with size-8 clusters is illustrated in Figure 7.3. It is broken into four steps:

1. Inference of optimal barrier stages
2. Combination of local barriers
3. Synthesis of the arrival stages
4. Transposition of the arrival stages, to produce the acknowledgement pattern.

Step 1 amounts to traversing the tree structure which is implied by the SSS clustering, using the pivot from each cluster as the representative in higher levels of the hierarchy. The cost of the three candidate algorithms can then be determined per cluster, and the cluster is marked with the method of lowest predicted cost.

Step 2 requires a horizontal traversal to create a hybrid barrier from the chosen candidates at each level in the hierarchy. Finding the optimal barrier per cluster can be executed concurrently, but they may differ in the number of stages, both because of variable cluster size and variable stage count of selected algorithms. It is therefore necessary to project these onto a single set of $P \times P$ matrices, with the longest partial barrier determining the count. The output of this step is a set of matrices which encode the parallel execution of local barriers. Shorter barriers must be padded with a number of empty stages, and the test implementation concatenates these at the end. This may impact the cost of the algorithm when run in the general simulator from Figure 5.5, but the significance of this is not explored, because the padded encoding is an intermediate representation which will be pruned.

Step 3 is the vertical combination, which takes the hybrid patterns developed for each level in the hierarchy, and combines them sequentially into a global arrival pattern. This can be achieved by concatenating them so that the arrival phases from lower levels precede the arrival patterns which connect the pivot elements at higher levels, effectively creating a pattern which will collect arrival signals from the entire process set at the top level.

Step 4 uses that the produced arrival pattern is a hierarchical construction, so that a corresponding distribution acknowledgement signals can be obtained by reversing and transposing the pattern. This calls attention to the D-barrier, as it does not designate a master process. At lower levels of the hierarchy, the cluster pivot is selected as an arbitrary master for the sake of the hierarchical composition. At the top level, no such requirement applies: at the completion of the arrival stages of this barrier, all acknowledgements have also been implicitly distributed. Therefore, we make an exception when the D-barrier has been selected at the top level, and do not require that its transpose pattern is included as part of broadcasting of the acknowledgements.

After the optimized matrices have been generated, the padding stages and other empty steps can be revisited. When producing an adaptively optimized barrier algorithm, the role of Algorithm 5.5 is to provide a common evaluation of the relative costs of alternatives. When a communication pattern has been derived, this role is largely fulfilled, permitting design tradeoffs to be reconsidered. Specifically, since the representation has already been specified in terms of P and pattern, simulating all barriers in a common framework is no longer necessary, so the choice of communication calls can be modified.

Since the dependency graph has been derived programmatically, its representation can be used as input to a small program generator which emits a barrier function with a hard-coded signal pattern. The output declares a set of communication requests which are initialized to the desired signal pattern, and traverse the dependency graph, emitting non-blocking, synchronized mode communication calls to the requests matching given edges. The output program initially branches to differentiate between process identifiers, and contains a customized sequence of transmissions and wait calls. Altering the communication

Chapter 8

Case Study II: Laplacian Stencil

Reviewing the framework outline in Figure 1.3, the work presented up to this point has developed methods for the independent observations of level 1, and indicated how matrices of weights can be derived to capture platform capabilities for level 2. In order to retain generality, these developments have addressed the performance of generic programming primitives, but in order to obtain the desired system-level model of execution cost at level 3, it is necessary to specify the requirements of a particular program. Having a structured method for extracting these requirements is the most significant reason for selecting the BSP model as our object of study. The strength of partitioning execution progress with respect to the flexible unit of a superstep, is the implication that a model of single steps permit an overall model to be derived by simple accumulation.

The error in any estimate of superstep execution time accumulates accordingly, a flexible way to examine how well system behavior can be modeled is to test the framework with an application which features a variable number of similar supersteps, such as data-parallel loop iterations. Furthermore, algorithms which feature computational and communication requirements which can be analyzed at compile time make suitable candidates for model validation, as the matrices representing their requirements can be derived without accounting for variable dynamic conditions.

For these reasons, this chapter demonstrates the analysis of a reasonably simple practical application, *i.e.* an approximation of the 2D Laplacian operator by a 5-point finite difference stencil and domain decomposition. This problem has several favorable properties: its neighborhood communication pattern provides beneficial scalability characteristics, its requirements can be relatively easily derived from static program analysis, and application to image data acts as an edge detection filter, providing immediate visual feedback that the computation has found a correct result. A further property which makes this an interesting test case is that there is a certain amount of flexibility in the granularity of the computational superstep, permitting testing to vary the balance of computation and communication, to examine how balanced execution is with respect to the executing platform.

The following experiments examine three aspects of the performance model used in con-

method in this manner obviously affects the accuracy of the predictions from the cost model, but assuming that the empirical values are dominated by the characteristics of the interconnect, the relative cost measures which resulted in the selection of a given pattern will still hold, unless the small cost of the eliminated empty steps alter the pattern's critical path. For the sake of argument, one might imagine that this would result from the horizontal combination of a linear barrier with 2 participants and a tree barrier of 2^{10} participants, if the waiting cost of the former is greater than $1/10$ of the maximal signalling cost per stage in the latter. This issue could be ameliorated by refining the general simulation in Figure 5.5 to examine the signal pattern for skipping past empty stages, and introducing corresponding conditions on zero-cost stages in the cost function.

For the tests at hand, the cost matrices from our two test platforms reveal that the cost of empty steps O_{ii} are in the the 10^{-7}-second order of magnitude, while signals to neighboring processes on the same shared memory already cost on the order of 10^{-6} seconds. The even clustering of subsystems on these platforms leave little concern that eliminating waiting stages will affect the tradeoffs made in pattern construction. It should still be acknowledged that this consideration is likely to affect *e.g.* barrier mechanisms for programming models with a unified notion of process across *e.g.* multi-core processors and graphics devices, such as OpenCL kernels [41]. We proceed with the cost function derived from our target systems, as refinement in this direction would lead to increases in program complexity which are unlikely to produce measurably different results.

7.4 Empirical Validation of Hybrid Barriers

Generating barrier code from the combination of the algorithm and architecture description lets us not only evaluate the optimizations provided by adaptivity, but also to argue the validity of our developed framework outside comparative studies of simulation. In particular, the ability to emit and compile hard-coded algorithms permits meaningful comparison to implementations in production use. This section empirically compares generated barrier performance to the barrier implementation in the systems' default MPI library.

Both test systems provide the OpenMPI implementation of MPI, as distributed with Rocks Linux for computational clusters. The source code of this implementation is open to public inspection. The *OpenMPI 1.4* source code shows that it employs a binary tree barrier with blocking synchronized-mode sends. Our testbed applies nonblocking sends, but this still establishes a strong *a priori* prediction that the performance of this barrier should resemble the one obtained our compiled T-barriers, to within a constant factor. We expect the difference of communication modes to reflect in a constant factor because the model is expressed as a weighted sum of signal costs. A constant deviation in the overhead of one signal will thus scale the cost of the entire pattern, but maintain the topology-dependent shape of the graph induced by the communication pattern.

The performance measurements presented in Figures 7.4 and 7.5 verify our expectation, suggesting that factors 0.5 and 1 are close to the mark, respectively. Thus, absolute performance of optimized barriers should offer improvements on the larger system. Effectiveness

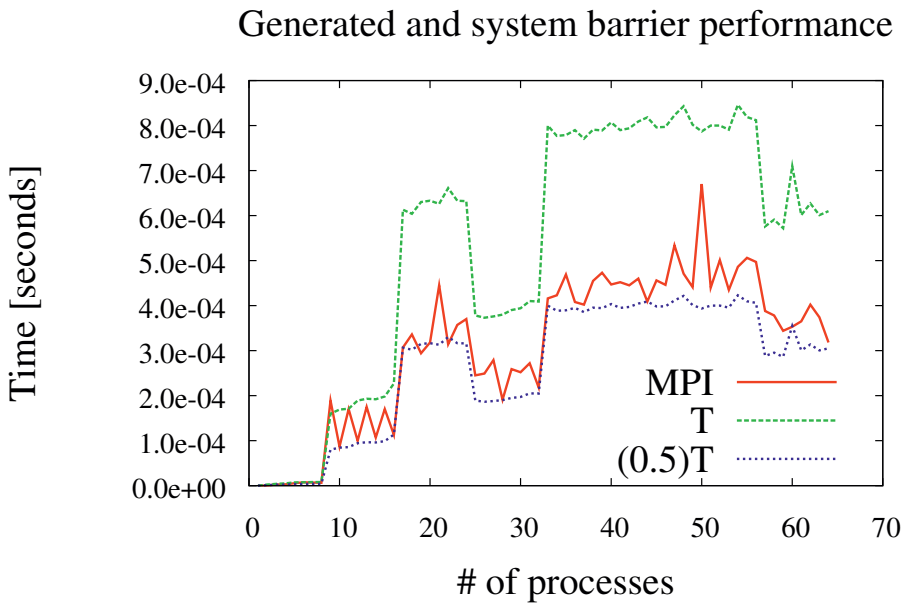

Figure 7.4: Barrier performance on 8-way 2x4-core cluster
MPI*: Timings of system library* MPI_Barrier
T*: Timings of general barrier simulation in Figure 5.5, binary tree barrier pattern as input*
0.5(T)*: The values from T, scaled by* 0.5
The scaled graph is included to show that the generic barrier simulator captures the correct shape of tree barrier characteristics. Absolute deviation can be attributed to constant differences in operation cost, as execution of the simulator uses different point-to-point communications from an explicitly programmed tree barrier.

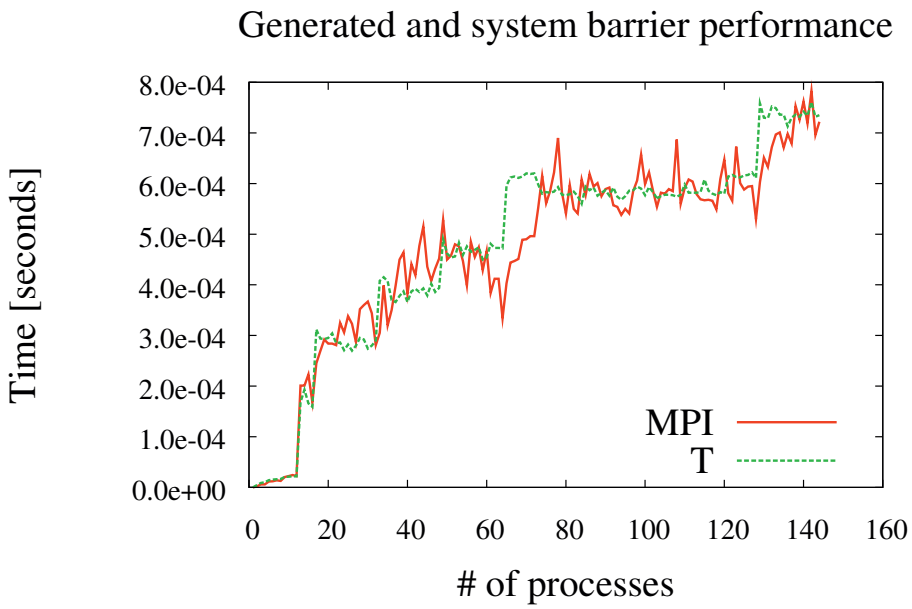

Figure 7.5: Barrier performance on 12-way 2x6-core cluster
MPI*: Timings of system library* MPI_Barrier
T*: Timings of general barrier simulation in Figure 5.5, binary tree barrier pattern as input*

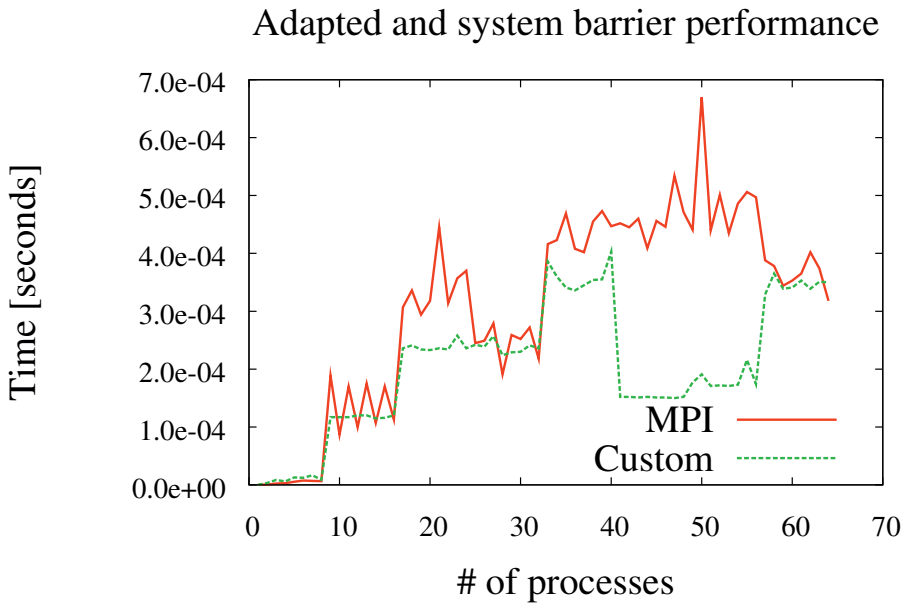

Figure 7.6: Adapted barrier performance on 8-way 2x4-core cluster
MPI*: Timings of system library* MPI_Barrier
Custom*: Timings of explicit barrier function compiled from hybrid pattern matrices*
While custom function timings are obtained empirically on actual hardware, the function code is generated off-line, using platform benchmark matrices as input and negligible computational resources for generation.

on the smaller depends on whether customization provides speedup greater than 2.

Figures 7.6 and 7.7 show that adapting the barrier algorithm to the topology is quite effective. It obtains comparable performance to the system library on the $8 \times 2 \times 4$ configuration, picking up a significant advantage from 40 to 56 processes. On a $12 \times 2 \times 6$ configuration, our expectation of equal or better performance is met, and the system library is outperformed by an approximate factor two from 60 processes.

7.5 Impediments to Production Deployment

The method for constructing topologically customized barriers presented in the preceding sections could form a drop-in replacement for the system libraries' barrier algorithm. Being a proof-of-concept implementation, however, a nontrivial amount of engineering work

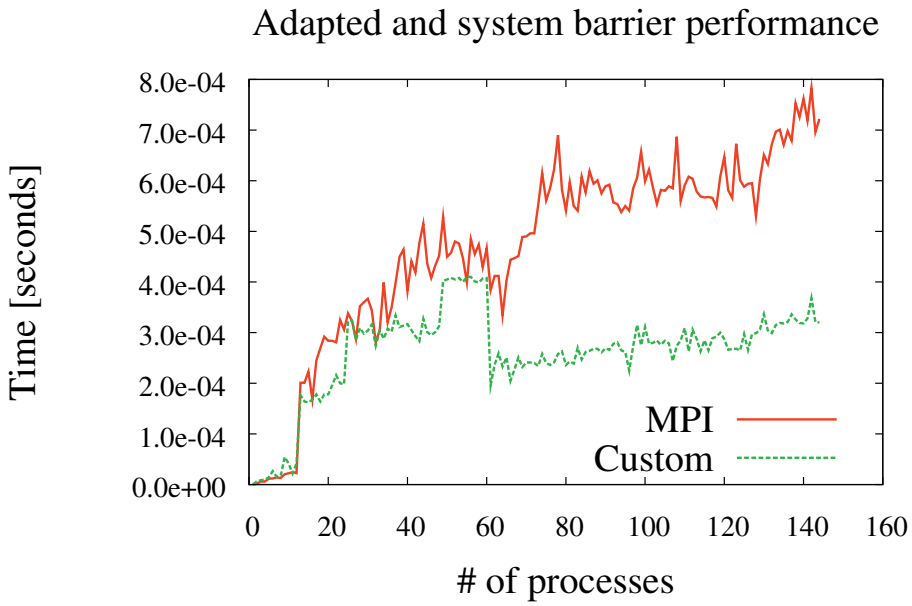

Figure 7.7: Adapted barrier performance on 12-way 2x6-core cluster
MPI*: Timings of system library* MPI_Barrier
Custom*: Timings of explicit barrier function compiled from hybrid pattern matrices*
While custom function timings are obtained empirically on actual hardware, the function code is generated off-line, using platform benchmark matrices as input and negligible computational resources for generation.

remains before it can replace the `MPI_Barrier` library function. Its most immediate shortcoming is that our approach has no attachment to *communicators*, meaning that it can presently only replace calls which synchronize the world communicator. Moreover, the process of adapting the barrier algorithm relies on benchmark data collected from the platform, and precompilaton of specialized barrier varieties for each desired value of P. Since the expensive step is the platform benchmark, and both benchmarking and precompilation need only be done once for a given system, there is no principal problem with generating a library of custom-case barriers for all values 1 through P the system affords. Dynamic linking would permit these to be linked at runtime for a given case, thereby providing a black-box extension to the system library.

The greatest weakness of our approach in a production scenario is its reliance on processor affinity and location. The manner in which the benchmarking procedure associates a process identifier with a location in the interconnect topology is surmountable in an experimental setting, as the same mapping can be enforced between benchmark and test runs by requesting the same resources from the system scheduler. Few production programs are at liberty to afford this, as it can not only lead to excessive waiting on a busy system, but also strips it of any liberties to schedule with respect to load balance or throughput. In the context of a large, expensive, shared resource, it stands to reason that maximizing utilization must take precedence.

Because our purpose is to validate the model using physical hardware, these concerns are immaterial to the conclusion that we attain sufficient detail to approximate locality-dependent latency. Having attained this, no further attention will be devoted to the problem of optimizing barriers, save to state that both the problem of mapping to less deterministically allocated resources, and the prohibitive cost of benchmarking on the fly could be resolved. Extracting a unique identifier from a processing core, and using pairs to look up interconnect statistics from a pregenerated database would make it feasible to construct the parameter matrices per communicator at run time: the computation of specific predictions is of negligible cost.

Chapter 8

Case Study II: Laplacian Stencil

Reviewing the framework outline in Figure 1.3, the work presented up to this point has developed methods for the independent observations of level 1, and indicated how matrices of weights can be derived to capture platform capabilities for level 2. In order to retain generality, these developments have addressed the performance of generic programming primitives, but in order to obtain the desired system-level model of execution cost at level 3, it is necessary to specify the requirements of a particular program. Having a structured method for extracting these requirements is the most significant reason for selecting the BSP model as our object of study. The strength of partitioning execution progress with respect to the flexible unit of a superstep, is the implication that a model of single steps permit an overall model to be derived by simple accumulation.

The error in any estimate of superstep execution time accumulates accordingly, a flexible way to examine how well system behavior can be modeled is to test the framework with an application which features a variable number of similar supersteps, such as data-parallel loop iterations. Furthermore, algorithms which feature computational and communication requirements which can be analyzed at compile time make suitable candidates for model validation, as the matrices representing their requirements can be derived without accounting for variable dynamic conditions.

For these reasons, this chapter demonstrates the analysis of a reasonably simple practical application, *i.e.* an approximation of the 2D Laplacian operator by a 5-point finite difference stencil and domain decomposition. This problem has several favorable properties: its neighborhood communication pattern provides beneficial scalability characteristics, its requirements can be relatively easily derived from static program analysis, and application to image data acts as an edge detection filter, providing immediate visual feedback that the computation has found a correct result. A further property which makes this an interesting test case is that there is a certain amount of flexibility in the granularity of the computational superstep, permitting testing to vary the balance of computation and communication, to examine how balanced execution is with respect to the executing platform.

The following experiments examine three aspects of the performance model used in con-

junction with the developed run-time library:

1. The magnitude of the overhead from providing a software implementation of distributed shared memory
2. The accuracy of performance predictions based on the performance model
3. The utility of the performance model with respect to automatic optimization

The chapter begins by considering the issues of comparing implementations using different programming models, in Section 8.1. Section 8.2 provides a brief summary of the test application, and Section 8.3 describes implementations of it using BSP, MPI and hybrid MPI/OpenMP programming models. Section 8.4 compares the obtained performance of the implementations. Section 8.5 describes experiments comparing performance predictions of the BSP implementation to empirically measured values. Finally, Section 8.6 describes an experiment where the application's opportunity for trading communication for computation is exploited, using model predictions to find an optimal point.

8.1 Experimental Design Trade-offs

Because the ultimate purpose of the presented framework is to provide a structured approach towards investigating practical performance, the validity of the approach depends not only on its consistency with predictions made on its own terms. In order to relate its utility to real application performance, an ideal test would be to examine the behavior of a full-scale application program. This is problematic, because BSP program examples are relatively rare, and mostly developed for purposes of research or education. Assessing the performance of the run-time library developed here features both the aspect of how it compares to alternative *BSPlib* implementations, as well as how it compares to implementations in more commonly applied programming models. The former suggests experimentation with a particular source program compiled with variable underlying implementations, while the latter requires a single application to be implemented using a range of models which may or may not provide programs with a similar range of operations. While the former type of test is simpler to execute, it also presents a significant risk of biased evaluation, because it will fail to capture model-specific limitations.

In order to address this issue, subsequent sections feature a comparison of three different implementations of the same numerical algorithm, using a requirement of identical output as the criterion for comparability. One of these is realized using *BSPlib*, which admits testing with two implementations of the interface. The other two represent more common choices for the target execution platforms, one being a pure MPI-based implementation, while the other programmatically recognizes the underlying platform topology by using OpenMP for node-level parallelism, and employing MPI for inter-node communication requirements. This selection is by no means represents an exhaustive exploration of available alternatives: it is made to highlight particular details.

Firstly, the comparison of the two *BSPlib* implementations is made in order to contrast design with eager background communication to a design which postpones communication

until synchronization, thereby dividing execution into alternating phases of computation and communication.

Secondarily, results obtained with a pure MPI implementation are presented because both *BSPlib* implementations we examine are layered over MPI, leveraging its portability and efficiency on commodity hardware. As the run-time library assumes responsibility for a nontrivial amount of work which would otherwise be the burden of the programmer, some overhead is expected: the purpose of showing results obtained without the use of automatic facilities is to show the magnitude of the corresponding performance loss.

Thirdly, the results from a hybrid MPI/OpenMP programming model are presented because the heavyweight processes used for MPI arguably can represent a poor choice for efficient execution on tightly coupled parallel hardware, utilizing the network stack of the operating system in order to transmit data between otherwise closely coupled processing cores.

As the choice of programming model is a variable factor, it is important to highlight that each model implies its own set of program design decisions which are relevant to the obtained performance figures. One major shortcoming of this is that it may result in a bias with respect to the programmer's familiarity and experience with a given model. Because comparison is based on the criterion that all programs perform the same computation, as validated by identical output, all three source programs were developed to this end. To dampen the impact of the bias due to the fact that all implementations are developed in conjunction by the same author, neither has been subjected to extensive tuning in order to obtain optimal performance. While an interesting goal in itself, aiming for such an objective would increase the emphasis on the author's variable programming ability with various models, without otherwise adding substantially to the discussion. On the other hand, development in order to produce programs with the most similar run-time behavior obtainable would ignore the difference in approach which comes naturally from the models' differences in abstractions.

As an attempt at striking a middle ground between these conflicting aspects, the three implementations are written with the qualitative goal of providing straightforward solutions without extensive programming effort. In particular, the BSP implementation commits communication as early as possible, the MPI implementation utilizes a cartesian communicator topology to simplify the communication pattern, and the hybrid implementation exploits shared memory for the convenience of automatic loop parallelization.

8.2 Laplacian Stencil and Domain Decomposition

As a greyscale image can be treated as a function of two variables with even spacing in both dimensions, a frequently encountered form is a five point stencil applied to 3×3 neighborhoods of the image, as given in Equation 8.1 [39].

$$z_{i,j}^{n+1} \approx 4 \cdot z_{i,j}^{n} - (z_{i-1,j}^{n} + z_{i+1,j}^{n} + z_{i,j-1}^{n} + z_{i,j+1}^{n}) \tag{8.1}$$

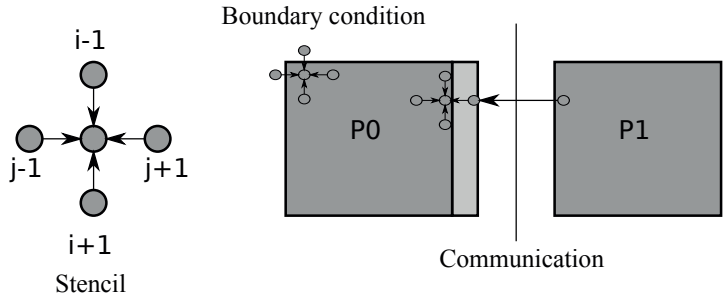

Figure 8.1: Boundaries and Ghost Area in 5-point Stencil Computation

Applying this stencil iteratively gives a sequence of arrays $z^1 \ldots z^n$ of successively closer approximations to the Laplacian $\bigtriangledown f(x,y) = \delta^2 f/\delta x^2 + \delta^2 f/\delta y^2$.

Equation 8.1 results from differentiating Taylor polynomials in both directions and assuming a uniform step size of 1; more elaborate equations feature wider neighborhoods from increasing the order of the polynomials or using f of higher dimension, or otherwise increase the number of terms in the kernel by using different step sizes per dimension. The point here is that to finite precision, the stencil will apply to finite neighborhoods, and thus produces a similar communication pattern when the domain is split across parallel processes. The situation stems from finding values for the missing stencil points along the boundaries of the locally stored subdomain. When the boundary is also a global one, applicable conditions can be given from the problem specification, but internal boundaries are artificial, in the sense that they arise from the partitioning. Figure 8.1 illustrates the shape of the stencil from Equation 8.1, and the boundaries when it is applied to the 2-way decomposition of vertically splitting a rectangular image in halves. The image point which must be passed from $P1$ to $P0$ (and implicitly, the symmetric case from $P0$ to $P1$) can be buffered in a *ghost area*, the contents of which are exchanged once for every iteration. When the decomposition is also in 2D, this creates a communication pattern of pairwise exchanges per iteration which resemble the stencil itself, and gives a natural decomposition of the computation into a superstep per iteration.

For the sake of simplicity, consider decompositions of $m \times n$ images into balanced, rectangular sections of $a \times b$, such that $P = ab$ processes each receive a $(m/a) \times (n/b)$ section, and a, b divide m, n without remainder. In this case, the communication load per iteration for a subdomain in the interior becomes proportional to $2(m/a) + 2(n/b)$ points, corresponding to the two horizontal and vertical boundaries of the subdomain. With periodic boundary conditions this is global, but otherwise there is a slight imbalance at the edges and corners; since this will be a reduction in the local requirement, we will consider communication to be bounded by the processes in the interior. With a greater ghost area, the communication of corner points also becomes an issue, but for the sake of this analysis we will disregard them, as they represent a small expense in relation to other boundaries. The computational requirement grows with ab, as the stencil applies to each interior point.

8.3 Implementation Details

The BSPlib implementation described in chapter 6 suggests a major weakness in the overhead involved with software emulation of one-sided operations. Indeed, the fact that it manages communication using MPI primitives suggests that obtained performance will *at best* be on par with a pure MPI implementation, and more likely, slower. On the other hand, the benefits which the run-time library is intended to automate are eager use of nonblocking communications, and deadlock freedom, which in the terminology of McCool [67] leads to a *simpler* and *safer* programming model. In order to estimate the cost and benefit in this trade-off, it is interesting to compare the sample application performance as obtained by equivalent implementations using different programming models.

For this purpose, the overall execution time of implementations using BSPlib, MPI and a hybrid MPI/OpenMP code are examined here. Given that the implementations are written using different programming models, their respective design decisions differ slightly, which affects the degree to which they are truly comparable. Furthermore, because the intention is to examine the utility of our performance model, a comparison of extensively optimized implementations would not serve to illustrate the support which it provides. Therefore, we will accept that a program which utilizes nonblocking communication explicitly most likely would outperform all of the considered implementations, and turn to examine the consequences of exploiting it using a run time library which affords it through the semantics of the programming model.

Among the considered implementations, the notion of bulk synchronous execution is unique to the BSP variety, which means that the computational superstep cannot be used as a common unit for comparison without further comment. Instead, we will use the fact that the application's computation proceeds by sequentially dependent iterations which require some measure of interprocess communication, and base comparisons on this. This does present an obvious mapping between supersteps and iterations which would provide grounds for comparison, but as there is reasonable argument that all the implementations might benefit from relaxing the manner in which this synchronization is effected, the impact of this is also examined.

8.3.1 BSP implementation

The first priority of the BSP implementation of the application is to expose all available overlap time to the communication library. Initialization therefore amounts to identifying 17 rectangular areas of interest, as illustrated in Figure 8.2. The eight outermost of these are the ghost areas which represents the borders of the local subdomain, as mirrored on the neighboring processes. Inside the local domain, there are 8 similar regions which represent the values which synchronization must replicate on the neighboring processes before the beginning of the subsequent iteration. These 16 areas must be consistent before the computation which updates the local domain is carried out. As the regions are not located in contiguous memory, initialization sets up communication buffers of similar extent, and registers them as targets of remote communication; after synchronization,

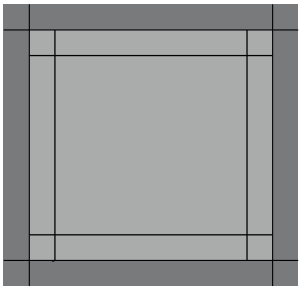

Figure 8.2: 17 Regions in BSP implementation

these buffers are copied into their correct positions using loops which manage the strided access. It should be noted that because of the shape of the 5-point star stencil, the corner areas which mirror data diagonally on the process grid is not necessary for borders of thickness 1; their communication is nevertheless included in all implementations because some of the experiments to follow will utilize thicker border areas. It also retains some measure of generality by which the results obtained here could more easily be extended to stencils of different shape and extent.

The requirement that the border areas are consistent before the local update can proceed means that the maximal time afforded for overlapping communication with computation is the update of the final, interior area of the local subdomain. Therefore, the update computation is divided into subroutines which initially receive/solve for the border areas, initiate communication, and computes the interior in the interim before synchronization. Because the border areas vary linearly with problem size while the interior varies with the square, we may expect various problem sizes to present different degrees of potential overlap to exploit.

Between the completion of the update and the reception of new ghost values, the iteration step is carried out with a block memory copy of the entire local subdomain from a buffer of updated values into a buffer representing the current step. This update could also be facilitated at a lower cost by switching pointers to the respective buffers, but the simplest management of the region layout is through arrays of pointers which relate the ghost and border buffers to the local domain. With this design decision in mind, handling double buffering using pointer switching is certainly feasible, but as it is not trivial, it is not implemented here in order to retain a conservative estimate of the performance attained by an un-optimized implementation of the application.

8.3.2 MPI implementation

The MPI implementation of the application lends itself naturally to the use of MPIs cartesian topology feature. Through the use of a cartesian communicator and an initial call to the `MPI_Cart_shift` function, the border exchange is implemented in a straightforward manner by 4 consecutive calls to `MPI_Sendrecv`, which is an operation provided

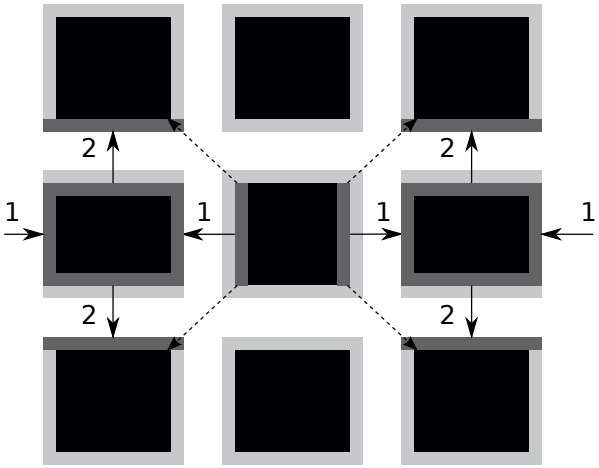

Figure 8.3: 2-Stage Border Exchange in MPI Implementation

precisely for shifting data around global matrices in the manner the application requires. Furthermore, vector types are used for strided (column) access, as they provide an opportunity for libraries to lay out an internal data structure for the sake of efficiency through repeated use. The independence of these access patterns from an absolute origin makes it simple to achieve the double buffering required by alternating iterations in terms of swapping the pointers which indicate which buffer contains the present and which is the previous iteration, which saves the time of an $O(N^2)$ memory copy operation with little programmer effort. Finally, the border exchange phase is completed in 2 stages, initially exchanging column vectors, and row vectors only in a second stage, as illustrated in Figure 8.3. As the column vectors in the first stage complete the ghost area of the rows communicated in the second, this is an optimization which saves the overhead and latency of performing 4 separate communications for corners/diagonal directions. The omitted transmissions are shown with dotted lines in Figure 8.3. This method is used because it amounts to a trivial extension of the vector types committed for communication, and is quite effective on high-latency interconnects.

8.3.3 Hybrid implementation

The hybrid implementation tested is a reduced version of the MPI implementation, which allocates one process per computational node and spawns OpenMP threads with a *parallel for* directive preceding the outer loop of the doubly nested loops traversing the local domain. The communication requirement is reduced for corners as in the MPI solution, as well as the pointer handling of double buffers. There is no explicit communication internal to the nodes, owing to shared memory capabilities. This has the side effect of creating unfortunate splits for prime numbers of nodes, as they are bound to divide the domain in $p \times 1$ striped layouts, but no particular measure is taken to avoid this, in the interest of

keeping implementations simple within the affordance of the programming model.

8.4 Comparisons of Strong Scalability

Throughout this section, strong scaling characteristics are measured on both systems using one larger problem size of 4096^2 points, and one smaller of 1024^2 points, as well as one case of 2048^2 points. These problem sizes are selected to show application behavior in the case where the size of the interior of a local subdomain is sufficiently large to give benefits by increasing parallelism, and the case where the computational intensity is too low, thus making the application communication bound, providing diminishing returns for increasing parallelism. Amdahl's law predicts that a threshold where increasing parallelism reaches diminishing returns is inevitable for fixed problem sizes, so studying sustainable application scalability would mandate using the weak mode (fixed parallel time). As our purpose here is to investigate the properties of a model, seeking out this threshold is useful to examine the features of the transition.

Results from a set of 11 comparative experiments in 3 categories will be presented. These are tabulated in Table 8.1, where configurations are categorized according to their purpose. The A set consists of comparisons of absolute performance figures obtained from the set of different implementations, intended to establish the performance loss due to the technical realization of one-sided messaging in our *libbsp* implementation. These experiments are discussed in Section 8.4.1. The B set consists of comparisons between performance figures obtained from the *libbsp* implementation and predicted values obtained off-line from a small simulator program which relies on captured platform parameters as input. These experiments are discussed in Section 8.5. Finally, C1 is a single experiment which builds on the previous sets, to utilize the manner in which the application affords adjustments to the balance of computation and communication. The purpose of this experiment is to demonstrate the utility of the performance model by displaying its ability to predict suitable modifications to application behavior, in order to exploit the performance potential of the underlying platform. This is discussed in Section 8.6.

8.4.1 Comparison of All Implementations

Results from preliminary tests of all implementations with the large problem size on both platforms are presented in Figures 8.4, 8.5 and 8.6. Figure 8.7 shows results with the small problem size on the larger platform.

The purpose of these tests is twofold. Primarily, they give a brief comparison of absolute performance, to establish the magnitude of the expected performance advantage of the MPI and Hybrid implementations due to their more explicit communication specifications. Secondarily, these tests double as a verification that all implementations are working correctly, and give identical results. To the latter end, the input sets are chosen as a pregenerated image of a section of the Mandelbrot set, chosen because it can generate images of

Table 8.1: Experimental Configurations

Configuration	Platform	Problem size	Metric
A1	8x2x4	4096^2	T(15)
A2	12x2x6	4096^2	T(15)
A3	12x2x6	4096^2	T(500)
A4	12x2x6	1024^2	T(500)
B1	8x2x4	4096^2	T(15)
B2	8x2x4	1024^2	T(15)
B3	12x2x6	4096^2	T(15)
B4	12x2x6	1024^2	T(15)
B5	12x2x6	4096^2	T(500)
B6	12x2x6	4096^2	T(500)
C1	12x2x6	2048^2	T(1)

arbitrary sizes simply by fixing the resolution of the generator. Applied to a greyscale image, the Laplacian operator works as an edge detecting image filter, which gives an instant, visual feedback if there are issues in the more complicated parts of the computation, such as the border exchange routines. Therefore, the modest count of 15 iterations was found to give stable and repeatable measurements on the test platforms, while still producing a distinct edge map of the input image, for examination and debugging purposes.

The results in Figure 8.4 show the expected performance difference between the BSP and other implementations, with an initial performance advantage close to a factor 3.5, shrinking to a factor 2 when all nodes of the cluster are employed. It should be noted that for the sake of comparison, another implementation of the BSPlib interface was tested, by recompiling the same source program as used with our *libbsp* implementation. The BSPonMPI implementation [91] is also a library which implements BSP using MPI as a communication layer, but instead of eagerly using nonblocking communication, it opts for delaying communication until synchronization time, ultimately realizing it using an *Alltoall* operation, thus separating computation and communication into distinct, alternating phases. Both Bisseling [19] and Hill and Skillicorn [45] argue the virtue of such an approach, referring its reduction in overall latency by reducing the number of messages, as well as how it exposes any attainable message combining/scheduling advantages to the system software. Sound as this argument may be, the first effect observed at modest scale is that the version of our laplacian solver compiled with *libbsponmpi* shows a sudden degradation at 64 cores, in Figure 8.4. Bearing in mind that the barrier tests on our 8x2x4 platform showed exceptional behavior for power-of-2 cases, another set of tests with the same problem size and iteration count were performed on the 12x2x6 platform, as shown in Figure 8.5.

The unmistakable tendency visible in Figure 8.5 is that between four and six nodes (48–72 cores), this implementation strategy begins to cause an overhead which quickly outgrows the order of magnitude of the remaining implementations, and further testing using this implementation is abandoned. Note, however, that prior to this point, the performance of the *Alltoall* approach is superior to that of our nonblocking implementation, suggesting

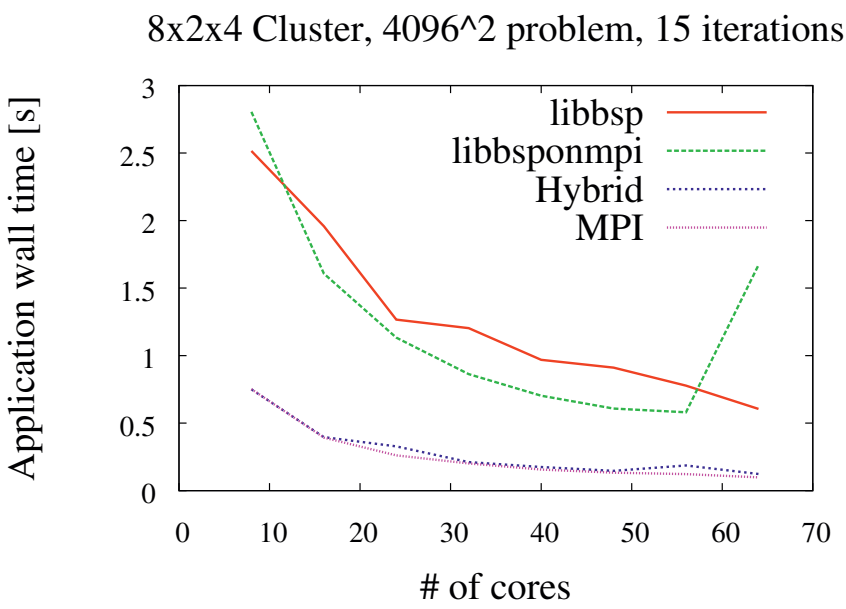

Figure 8.4: A1: All Implementations
libbsp*: BSP implementation compiled with our BSPlib implementation*
libbsponmpi*: BSP implementation compiled with BSPonMPI BSPlib implementation*
Hybrid*: MPI and OpenMP hybrid implementation with implicit synchronization*
MPI*: MPI implementation with implicit synchronization*
Comparison using large input set and fixed iteration count. The fixed iteration count enables the superior MPI and Hybrid codes to synchronize implicitly, as no global convergence criterion is needed to halt the computation.

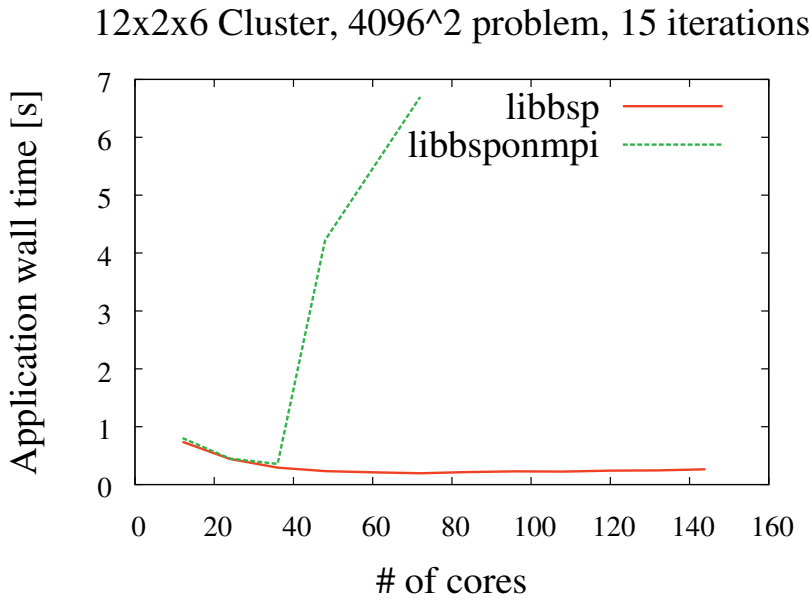

Figure 8.5: A2: BSP Implementations Only
libbsp*: BSP implementation compiled with our BSPlib implementation*
libbsponmpi*: BSP implementation compiled with BSPonMPI BSPlib implementation*
Comparison of the two BSPlib alternatives only, showing a scalability problem with BSPonMPI*. Timings for runs greater than 60 nodes were unobtainable due to program failure.*

that the most flexible strategy might be to provide a library capable of both methods, as well as switching between them as conditions dictate, either by programmatic hint or automatic detection.

Disregarding the BSPonMPI version and focusing on the more comparable set of implementations, Figure 8.6 presents the absolute execution times of 6 variations executed on the 12x2x6 platform, using the larger input data set. The three different implementations are examined in two varieties each, to examine the impact of the previously mentioned possibility for relaxing synchronization. The basis of this is the observation that the communication pattern of the border exchange already implies an implicit synchronization, as each pair of neighbors exchange blocking messages before proceeding to the next iteration. While this may practically mean that some processes at one end of the topology leave the exchange phase before processes at the other have entered, it still represents a guarantee that no process will proceed to an iteration step for which it has not received the values it requires, thus implicitly enforcing that the computation proceeds in lock step. If the computation is to run for a fixed number of iterations, this creates a greater robustness to variability in arrival to the synchronizing function call, as well as reduces the number of messages required to establish synchronous iterations. On the other hand, if the termination of the computation is bound to some threshold of convergence or similar, explicit synchronization becomes a necessity in order to disseminate the information on whether to proceed with the next iteration at all, typically in the form of a reduction and a broadcast operation. In order to make a conservative estimate, and not impose any application-specific assumptions on the communication pattern and data payload required in such a scenario, the Hybrid+R and MPI+R figures emulate this behavior by a single, explicit *MPI_Barrier* operation at the end of the border exchange phase. The *libbsp+R* version already features explicit synchronization, so in this case, the similar emulation is to collect a single value from each process, transmitted just after the iteration is complete.

Although difficult to divine from Figure 8.6, there is an observable, small overhead involved in this operation; Table 8.2 displays the two varieties of the MPI implementation, along with their absolute difference. A 500 iteration run will not amplify the additional cost per iteration to a point where significant differences arise, but it is noticeable already at this scale. Note that for extended runs, the overall impact grows in proportion to its influence on each single iteration, which will be of importance when we turn to the per-iteration cost as a metric of execution time.

Another interesting feature is to note that as long as the problem continues scaling for the BSP implementation, the additional communication requirement of the reduction appears to be well masked, while as the problem turns communication bound, an observable difference emerges as this additional message adds further delay to a computation which is already held up at the synchronization point.

The most important feature Figure 8.6 is, however, that it identifies an area of the parameter space where the performances of the MPI/Hybrid and BSP implementations part ways, as the BSP implementation becomes communication bound at 72 cores.

Figure 8.7 shows that test runs with a significantly reduced problem size on the larger of the two platforms produces communication bound behavior in all implementations, *i.e.*

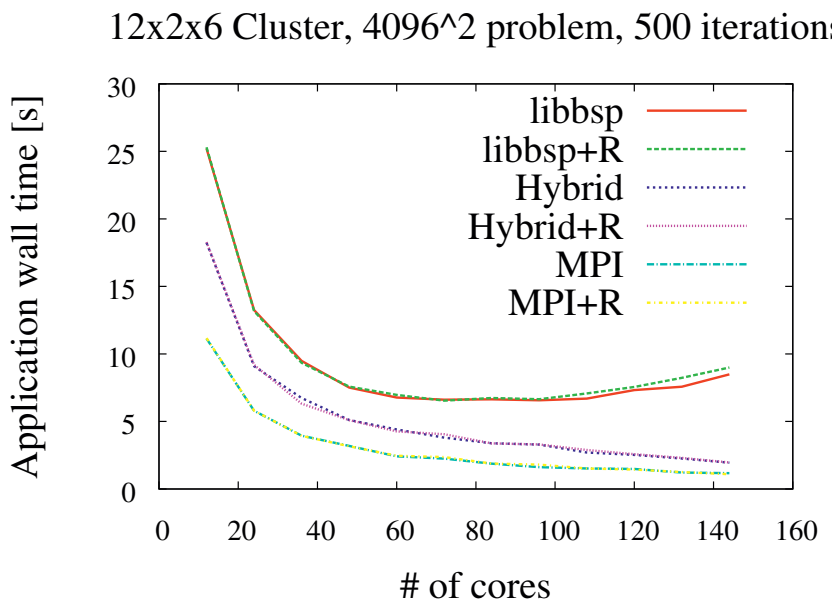

Figure 8.6: A3: Selected Implementations

libbsp*: BSP implementation compiled with our BSPlib implementation*
libbsp+R*:* libbsp *modified with global reduction of convergence criterion every synchronization* Hybrid*: MPI and OpenMP hybrid implementation with implicit synchronization*
Hybrid+R*: MPI and OpenMP hybrid implementation with global reduction of convergence criterion every synchronization*
MPI*: MPI implementation with implicit synchronization*
MPI+R*: MPI implementation with global reduction of convergence criterion every synchronization*
Comparison shows superior scalability for Hybrid and MPI varieties for large input set and longer, fixed iteration count. Note added overhead for the work of reducing the convergence criterion. BSP performance is largely unaffected by the addition of this reduction, as the addition to inherent synchronization overhead is small. Iteration count is fixed, enabling the use of implicitly synchronous implementations.

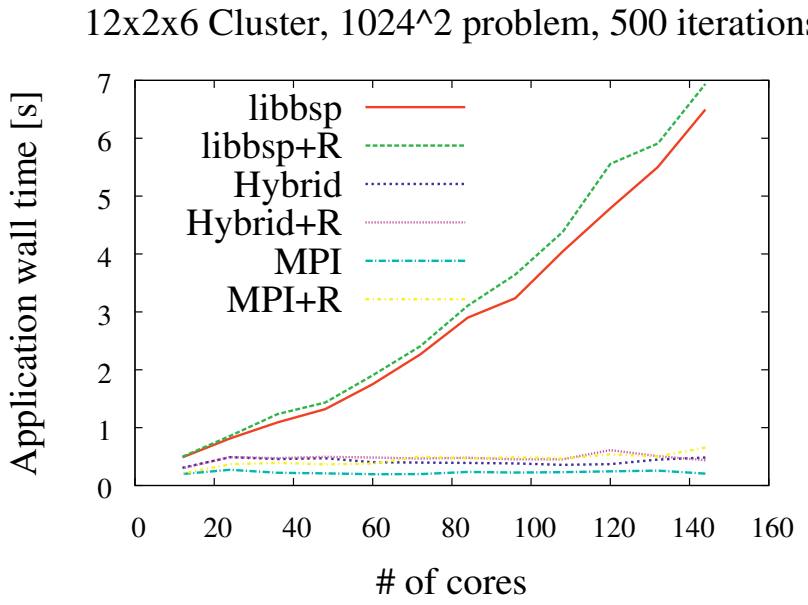

Figure 8.7: A4: Selected Implementations
libbsp*: BSP implementation compiled with our BSPlib implementation*
libbsp+R*:* libbsp *modified with global reduction of convergence criterion every synchronization* Hybrid*: MPI and OpenMP hybrid implementation with implicit synchronization*
Hybrid+R*: MPI and OpenMP hybrid implementation with global reduction of convergence criterion every synchronization*
MPI*: MPI implementation with implicit synchronization*
MPI+R*: MPI implementation with global reduction of convergence criterion every synchronization*
Comparison of small input set and long, fixed iteration count shows impact of reaching diminishing returns in strong scaling mode. Hybrid and MPI implementations do not benefit from additional resources, while BSP is dominated by a growing synchronization overhead.

Table 8.2: MPI And MPI+R Wall Times

P	MPI	MPI+R	Absolute Difference
012	1.113838e+01	1.117476e+01	0.03637
024	5.794761e+00	5.807795e+00	0.01303
036	3.945278e+00	3.939208e+00	0.00617
048	3.189016e+00	3.160863e+00	0.02815
060	2.421752e+00	2.465682e+00	0.04393
072	2.247901e+00	2.360757e+00	0.11286
084	1.875643e+00	1.881669e+00	0.00602
096	1.614748e+00	1.804870e+00	0.19012
108	1.517356e+00	1.522756e+00	0.00540
120	1.482026e+00	1.441480e+00	0.04054
132	1.220200e+00	1.257055e+00	0.03686
144	1.172386e+00	1.097181e+00	0.07521

increases in the number of nodes produces performance degradation, most notably, a particularly distinct one for the *libbsp* implementation. This is, in and of itself, merely a confirmation of Amdahl's law, in that the numerical intensity of the smaller problem size is insufficient to justify the addition of further computational resources, and the communication requirement added by scaling the core count unmistakably becomes overhead only. The limited impact this has on the Hybrid and MPI implementations is testament to their efficient communication, but there is no evidence of any benefit from increasing the computational power. Although the results in Figure 8.7 do not establish any novel conclusion in themselves, using them in conjunction with the results of Figure 8.6 provides bounds on problem size. Specifically, experiments A3 and A4 show that the crossover point where the application gives diminishing returns will occur for some problem size between 1024^2 and 4096^2 for the fastest implementations.

8.5 Application Performance Predictions

As the precision of the individual benchmarks has been determined in isolation, it is necessary to investigate whether the interaction between an application program and a platform is accurately captured by their composition. Part of the reason for studying a communication-oblivious algorithm like our present application, is that this enables us to easily isolate the requirements of computation and communication from a static analysis of the source program. Examining predicted and observed run times of this program thus produces an empirical basis for evaluating whether the performance parameters of the model represent a realistic choice in an applied context.

```
// Setup of pxp matrices encoding communication requirements
memset ( msgs, 0, p*p*sizeof(double) );
for ( int r=0; r<p; r++ )
{
    int i = r/dims[1], j = r%dims[1];
    // Determine neighbor process identifiers in 8 directions:
    int
        n = PMAP(i-1,j), s = PMAP(i+1,j),          // North, south
        w = PMAP(i,j-1), e = PMAP(i,j+1),          // East, west
        nw = PMAP(i-1,j-1), ne = PMAP(i-1,j+1),
        sw = PMAP(i+1,j-1), se = PMAP(i+1,j+1);

    // Communication pattern: 8 neighbors
    MSGS(r,n) = MSGS(r,s) = MSGS(r,w) = MSGS(r,e) =
    MSGS(r,nw) = MSGS(r,ne) = MSGS(r,sw) = MSGS(r,se) = 1.0;

    // Problem size specific message sizes
    MSGSZ(r,n) = MSGSZ(r,s) = lpsize[1] * sizeof(double);
    MSGSZ(r,w) = MSGSZ(r,e) = lpsize[0] * sizeof(double);
    MSGSZ(r,nw) = MSGSZ(r,ne) = MSGSZ(r,sw) = MSGSZ(r,se) = sizeof(double);
}
```

Figure 8.8: Application-specific Matrix Setup

```
double kernels = lpsize[0]*lpsize[1]; // Local problem size
double predcomp = kernels / rate;     // Computation time
for ( int r=0; r<p; r++ )
{
    double l, b, o;
    b = l = o = 0.0;

    // Latency & bandwidth
    for ( int dst=0; dst<p; dst++ )
    {
        maxoverhead = MAX(maxoverhead, MSGS(r,dst) * O(r,dst) );
        timev[r] = MAX(timev[r], 2.0*L(r,dst) +
            4*sizeof(int) * B(r,dst)/2.0 +
            MSGSZ(r,dst) * B(r,dst)/2.0
        );
    }
    timev[r] += maxoverhead;
    // Max comm time dominates superstep
    maxcomm = MAX(maxcomm,timev[r]);
}

printf ( "%d %e\n", p,
    steps * (MAX(predcomp,maxcomm)+barriercost)
);
```

Figure 8.9: Predictor Program

8.5.1 Experimental Methodology

In order to obtain performance predictions, a short program was written to estimate overall run time from benchmark matrices. Figures 8.8 and 8.9 show the relevant fragments of the program which generated all reported application performance predictions. Initialization and I/O code for loading the measurement matrices is omitted for the sake of brevity. Because of how the C language manages dynamic memory allocations, the manipulation of matrices is written using the indexing macros `PMAP`, `MSGS` and `MSGSZ`, which hide a straightforward translation from 2D indices to linear arrays. All of these index zero-initialized matrices of integers and doubles, with `PMAP` being a cartesian map of process ranks, while `MSGS` is a $P \times P$ map of message counts between ranks in a superstep, and `MSGSZ` a corresponding map of the message sizes. The matrices `L`, `O` and `B` index the $P \times P$ matrices of platform benchmark data loaded from disk, signifying latency, overhead and inverse bandwidth, respectively. The value of the variable `barriercost` is obtained from the simulations described in Chapter 6.

Note that as the program logic in Figure 8.8 only generates the communication pattern of this particular application, the entire program could be generalized by loading it as input data instead of hardwiring it in the code. This is a very surmountable technical task, but is not done here because the presented program already illustrates that the prediction computed in Figure 8.9 is parametric with respect to the application communication pattern.

On a similar note, the variable `rate` is an estimate of the participating processors' sustainable number of 5-point stencil updates per second, obtained from a benchmark like those employed in Chapter 4. On the larger test platform, this figure was measured to 24931455 for local subproblem sizes on the order of hundreds of kilobytes, and 237758547 for tens of kilobytes; the smaller platform also produced a similar variation. To obtain the reported predictions, these figures were provided as input to the predictor in large and small test cases as appropriate, with the observation that they might as easily have been encoded in a $P \times 2$ matrix and selected programmatically. This is unnecessary in our present case, because the local subproblem size is uniform per test, and it is the primary source of variable computational rate in the examined system. As a side note, we may observe that the processing cores of the larger system feature private 64 kilobyte level-1 data caches, and speculate that more systematic benchmarking would be likely to reveal it as a threshold for the order-of-magnitude performance leap. For our present purpose, however, the number of test configurations makes it unnecessary to account for the full range of performance properties of the employed stencil kernel, as there is a limited number of test cases which require measured values.

8.5.2 Results And Discussion

Figures 8.10 and 8.11 plot predicted values from the predictor program alongside measured walltime figures on the small cluster, for runs of 15 iterations. We may note again that the small problem size shows characteristics of immediately becoming communication bound. The increasing tendency of walltime measurements in Figure 8.11 shows

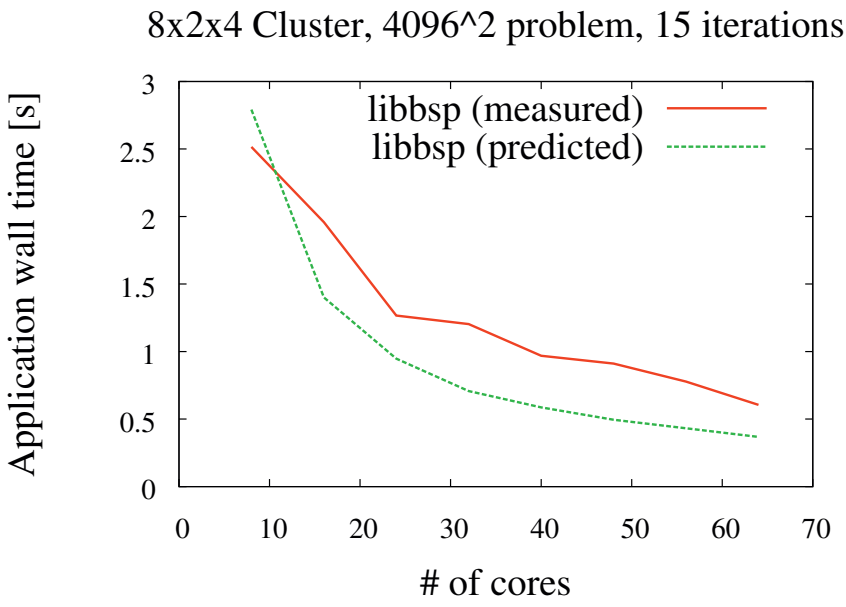

Figure 8.10: B1: Prediction vs. Measurement, Large Problem
libbsp (measured)*: Timings of BSP implementation compiled with our BSPlib implementation*
libbsp (predicted)*: Predicted performance using predictor programs in Figures 8.8 and 8.9 with benchmark data*
Comparison examines performance for the large problem set, and small, fixed iteration count on the smaller platform.

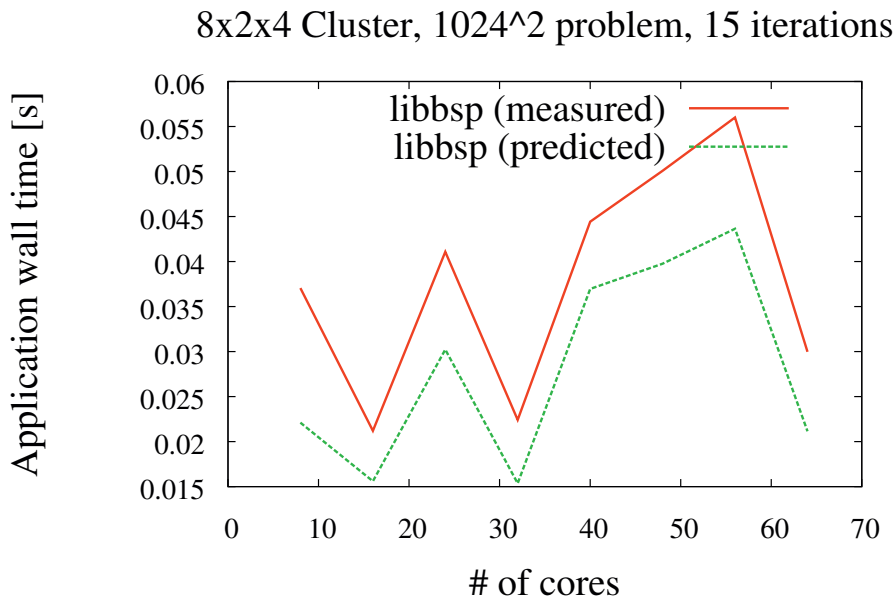

Figure 8.11: B2: Prediction vs. Measurement, Small Problem
libbsp (measured)*: Timings of BSP implementation compiled with our BSPlib implementation*
libbsp (predicted)*: Predicted performance using predictor programs in Figures 8.8 and 8.9 with benchmark data*
Comparison examines performance for the small problem set, and small, fixed iteration count on the smaller platform.

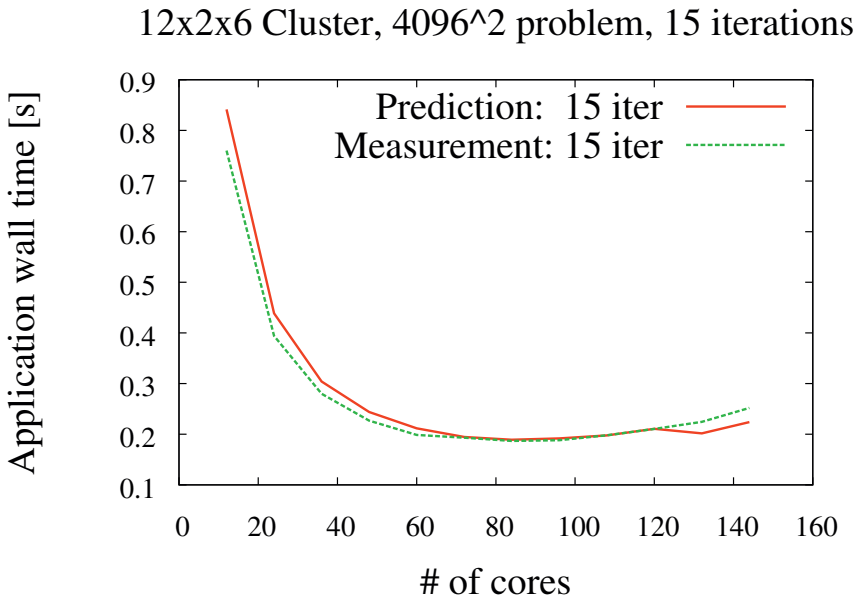

Figure 8.12: B3: Prediction vs. Measurement, Large Problem
Measurement*: Timings of BSP implementation compiled with our BSPlib implementation*
Prediction*: Predicted performance using predictor programs in Figures 8.8 and 8.9 with benchmark data*
Comparison examines performance for the large problem set, and small, fixed iteration count on the larger platform.

significant favorable exceptions at those core counts which are powers of 2. This is predictable with the knowledge that the communication pattern of the underlying synchronization primitive favors these sizes, but it is nevertheless noteworthy to find that the automated prediction method describes the effect fairly accurately without any explicit acknowledgement of its origin.

Figures 8.12 and 8.13 show the results obtained with the same problem parameters, using the larger cluster. These results show similar predictive power, with general tendencies clearly mirrored between predicted and measured results, and relative errors ranging from the negligible to $10 - 20\%$ at worst.

At this point, we note that the results reported so far have examined runs of relatively small iteration counts, and that predicted values are compared to a simple walltime measurement of a single run. This is done for several reasons, most importanly, in order to stay true to the objective of assessing the usability of the developed model in a practical scenario. It would

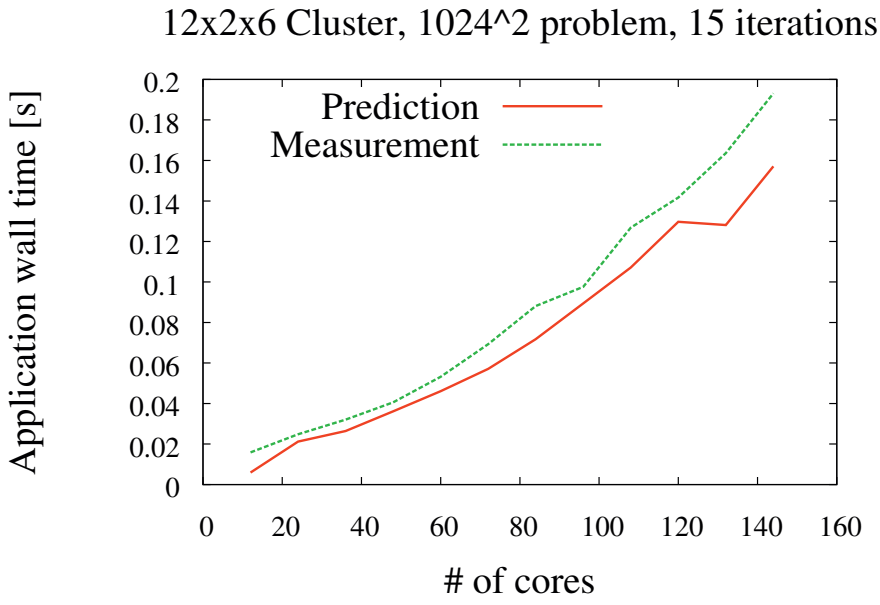

Figure 8.13: B4: Prediction vs. Measurement, Small Problem
Measurement*: Timings of BSP implementation compiled with our BSPlib implementation*
Prediction*: Predicted performance using predictor programs in Figures 8.8 and 8.9 with benchmark data*
Comparison examines performance for the small problem set, and small, fixed iteration count on the larger platform.

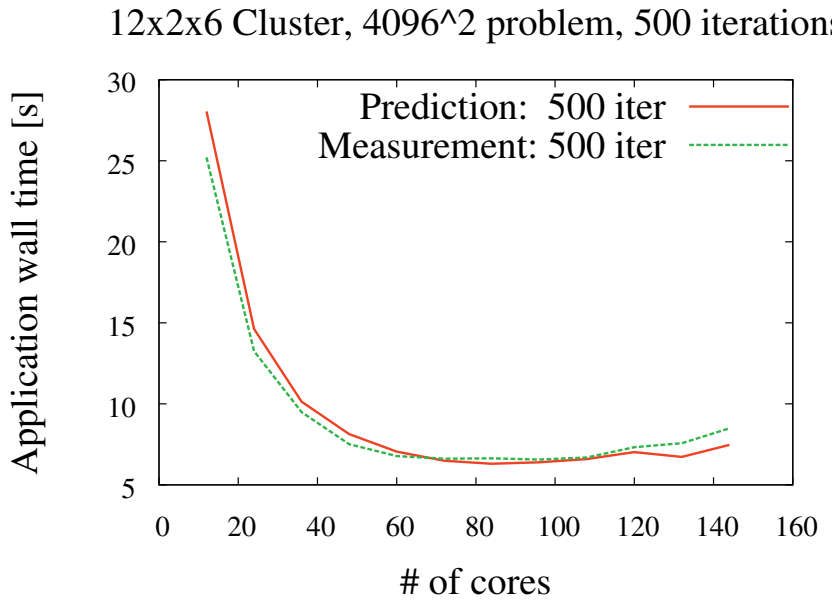

Figure 8.14: B5: Prediction vs. Measurement, Large Problem
Measurement*: Timings of BSP implementation compiled with our BSPlib implementation*
Prediction*: Predicted performance using predictor programs in Figures 8.8 and 8.9 with benchmark data*
Comparison examines performance for the large problem set, and large, fixed iteration count on the larger platform.

be quite feasible to improve model accuracy both by tracking the source of deviations and specializing the benchmarks to devote more time to yield more accurate estimates precisely where the application behavior deviates, as well as developing some statistic for multiple runs on the measurement side. Neither of these approaches are taken, because they impose post-fact considerations of the application/platform interaction, and therefore would undermine the integrity of the *predictive* aspect of the modeling effort. Practical use of the model would involve considering the consequences of its predictions, rather than extensive work to retrofit a test set to its premises.

While measuring in a single sample of 15 iterations avoids the potential bias of selecting a measure of central tendency, it also introduces the potential fallacy of obtaining a particularly favorable run by chance. In order to address this concern, the results reported in Figures 8.14 and 8.15 report a similar experiment run for 500 iterations. These tests are carried out on the larger test system, firstly because it avoids any potential bias intro-

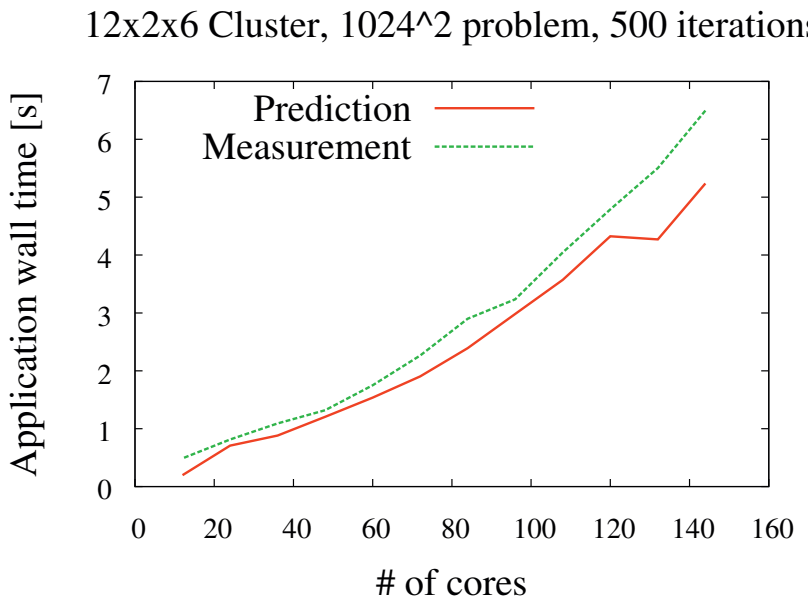

Figure 8.15: B6: Prediction vs. Measurement, Small Problem
Measurement*: Timings of BSP implementation compiled with our BSPlib implementation*
Prediction*: Predicted performance using predictor programs in Figures 8.8 and 8.9 with benchmark data*
Comparison examines performance for the small problem set, and large, fixed iteration count on the larger platform.

duced by the fact that the smaller system was used as the development platform for the benchmark programs, and secondarily, because the larger system admits tests at greater core counts. While 500 iterations is still a small number, these experiments represent a significantly longer interval than the previous experiments, and indicates that a similar relationship between predictions and observations holds for run times which extend orders of magnitude beyond the previous walltime observations.

One important observation here is that although the *relative* error of prediction retains the same characteristics, the *absolute* error has naturally grown into the order of seconds, along with the overall run time. The reason is the obvious fact that our model inaccurately predicts the execution time of a single iteration, and accordingly, the absolute error scales linearly with the number of supersteps executed, like the wallclock time. A simple corollary to this observation is that the method will be poorly fit to provide an off-line bound on resource requirements such as a tight bound on the wall time of a long-running application. For our present concerns, however, it suffices to retain the result that the developed model captures behavior which is sustainable, to justify an experiment which examines its applicability towards balancing the application's execution on given hardware.

8.6 Model-driven Optimization

As shown in the preceding sections, we have instantiated from the framework a model of which is parametric both in the performance parameters of the target platforms, and the characteristics of the application program. The benefit of this is that it enables experimentation with all associated performance parameters without actual execution of the program; in principle, this permits answering "what-if" questions regarding *e.g.* the impact of improving a subset of the processors, halving the network latency, or similar proposals.

For practical validation purposes, the executing platforms available for this study are fixed, which restricts our investigation to establish the impact of adapting the programmatic side of these equations to fit a given platform. Furthermore, the application program contains data dependencies which restrict our flexibility in the tradeoff between communication and computation. In order to achieve optimal parallel efficiency, the communication and computation of the program would have to be tuned to exactly balance the highly variable facilities of the underlying platform, for a globally uniform perfect overlap. Although the platform profile we have captured suggests that such an application might be synthesized, its utility would be restricted to demonstrating that all components can be employed to their capacity, within some error bound.

Examining the dependencies of the laplacian stencil code instead, we find that it does present an adjustable trade-off between computation and communication, albeit one which is constrained by application requirements. Specifically, the periodic exchange of neighboring border points admits the possibility of exchanging borders which are b cells wide, reducing the frequency of exchanges to once every b iterations, but requiring that the computations which update the extra cells are duplicated on neighbor processes. This permits our BSP implementation to proceed in supersteps which encompass b iterations, to adjust

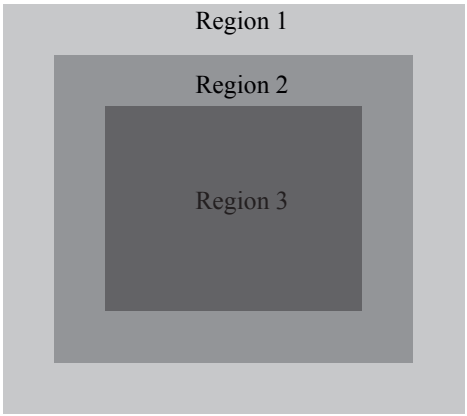

Figure 8.16: Shadow Cell Regions In A Local Subproblem

the balance of computation and communication within them. We can therefore attempt to improve the exploitation of the potential overlap, and thereby the parallel efficiency, without compromising the integrity of the computed result.

For the discussion of the potential and restrictions when exploiting this potential, Figure 8.16 illustrates and labels 3 regions of a local subproblem. Region 1 represents the external border, which must be periodically copied from neighboring processes. Region 2 represents the local border, where an updated result must be updated to the value of iteration b prior to sending its contents off for replication on a neighboring process. Finally, region 3 represents the values which are of concern only to the local process.

Our first observation will be that in order for the cells in region 1 to propagate correct contributions inwards in the local domain, the stencil computation must also be applied in this region for the $b - 1$ iterations when borders are not updated by communication. Relative to the cost of computation per *iteration* this is only a linear increase, but as we are analyzing the total amount of computation carried out in a superstep, each of these intermediate iterations also adds the cost of computing the entire interior. A slight reduction in the added computation can be obtained from the observation that each iteration propagates the contribution of a cell by only one cell spacing, so the impact of the cell values at the extremities on regions 2 and 3 is delayed the number of iterations given by their distance from the interior. Since region 1 is updated by communication every b iterations, this means that the computed area in region 1 can diminish by a border of 1 for each iteration carried out.

The second observation is that updated values from region 2 must be transmitted in the final iteration of a superstep. In order to maximize the potential overlap, this means that the order of computation in regions 2 and 3 must be adapted so that region 2 is completed and sent before computation in region 3 begins.

Third and finally, we note that although the amount of computation per superstep grows with the square of the increase in communication volume, the amount which may be ef-

```
double extracomp = 0.0,
       mask = pow(lpsize[0]-steps,2.0) / pow(lpsize[0],2.0);
for ( int i=0; i<steps-1; i++ )
    extracomp += (lpsize[0]+i)*(lpsize[1]+i);
extracomp /= rate;
printf ( "%d %e\n", p,
    ((extracomp + barriercost)/steps) + // Amortized
    (1.0-mask)*predcomp +               // Final step
    MAX(mask*predcomp,maxcomm)
);
```

Figure 8.17: Adapted Superstep Prediction

fectively overlapped with this communication (*i.e.* region 3) shrinks as a consequence of data dependencies.

Figure 8.17 gives a modified version of how the predictor program in Figure 8.9 emits its final value, where these concerns are taken into account. In order to permit comparisons with runs of variable length, the metric of time is normalized to the cost per iteration, with the variable `steps` representing the number of iterations per superstep. The extended number of kernel applications is found in the variable `extracomp`, while the variable `mask` represents the ratio of kernel invocations which can be overlapped to the number of kernel invocations in the overall local domain; this is used to compute the amount of overlapped computation from the same estimate of local domain computation cost as was used in previous experiments.

In order to find a test case for applying the proposed optimization, we may start by investigating the case where the balance of communication to computation is at its least favorable, *i.e.* the properties of the 1024^2 problem on 144 cores. Extracting the barrier cost, predicted computation and worst-case communication times from the predictor program reveals that the model estimates these to be

$$T_{sync} \approx 1.02 \cdot 10^{-2} s$$

$$T_{comp} \approx 3.03 \cdot 10^{-5} s$$

$$T_{comm} \approx 2.08 \cdot 10^{-4} s$$

which gives us Equation 8.2 as an approximation of the imbalance between computation and communication.

$$T_{comm} - T_{comp} = 2.08 \cdot 10^{-4} s - 3.03 \cdot 10^{-5} s \approx 1.78 \cdot 10^{-4} s \tag{8.2}$$

At an observed execution rate of $237758547 kernels/s$, this corresponds to

$$1.78 \cdot 10^{-4} s \cdot 237758547 \frac{kernels}{s} \approx 42321 kernels$$

which in turn suggests local subproblem sizes of $\sqrt{42321} \approx 205$ elements square. On a 12×12 process grid, this suggests that a global problem size of 2460^2 will be closer to

Chapter 9

Conclusions and Future Work

In this thesis, we have illustrated how bulk synchronous programming and processing models can be leveraged to capture heterogeneous performance parameters of both programs and execution platforms, and partially or fully automate the analysis of their interaction.

We have used the notion of synchronized supersteps to create a system-wide bound on the amount of outstanding computation and communication at a point of execution, and shown how this permits the work of all processing elements to be captured in P-dimensional vectors of communication and computation time for the present step. In order to obtain these vectors, the computational and communication characteristics of the executing platform can be captured in matrices which contain linear approximations of empirically measured subsystem behavior. The construction of these matrices result in similar shapes, which permits the program and platform characteristics to be combined by element-wise product, and superstep execution time to be obtained by the maximum of horizontal sums. The resulting method couples the computation and communication requirements of a program to an independent profile of the executing platform.

We have seen that synchronization cost can be modeled as the critical path through a weighted dependency graph, producing accurate predictions parametric in system topology and scale. For bulk-synchronous programs, the impact of synchronization is an unavoidable overhead, which suggests that program scalability requires that computational and communication intensities are of sufficient magnitude to make this overhead tolerable. Two case studies illuminated the cost of synchronization in different ways. One showed that the performance parameter space of software synchronization methods admits efficient synchronization methods to be automatically generated. Another showed that cost of synchronization in a stencil application can become a limiting performance factor at modest scales, but that measures to improve the balance of computation and communication per synchronization can be leveraged towards reducing this effect. This demonstrates a technique to determine the impact of synchronization on program behavior, and validates it by practical utilization of its predictions.

balanced execution than the previously examined problem sizes. The assumption that the problem size would be entirely flexible to fit the executing platform is not representative of practical cases, nor is it necessary to examine the consequences of trading computation for communication. As we are simply interested in a hitherto unseen test case to admit performance tuning, we therefore proceed with the 2048^2 problem size in experiment C1.

Figure 8.18 displays the predictions for per-iteration time obtained by employing the adapted predictor program in Figure 8.17 with the variable `steps` adjusted from 5 through 50, thus varying the number of iterations assigned to each superstep. It also displays empirical measurements with variable iteration counts, averaged over the iterations executed. The per-iteration figures previously obtained from runs with the MPI implementation at 144 cores are shown for comparison, with and without explicit synchronization. The data displayed in Figure 8.18 reflect too many differing assumptions and implementation choices to be read as an unbiased performance comparison, but the key point is that the predictions correctly identify the minimal iteration time obtainable by the BSP implementation at a border width of 31, using figures which either are, or could be automatically computed in a straightforward manner.

The collected results were extracted from runs of 500 *supersteps*, which obviously implies a differing total number of iterations across the parameter range. The reason for this is that terminating after a fixed number of *iterations* would halt computation in the middle of a superstep in cases where the border width is not a factor of the iteration count. One alternative to this is establishing a common iteration count with all test cases as factors, but this would require excessive run times without adding significantly to the result material, as the objective is simply to observe system behavior in a steady, predictable state.

To explain the difference between the predicted and observed results in Figure 8.18, note that the model representation of the test program refrains from including the overhead associated with copying data between successive iteration buffers, instead of swapping them in the manner of the other implementations. Adding detail to close this gap will not be examined further, as the approximation which disregards it has already captured a sufficient detail to predict the optimal width. The benefit obtained by the additional effort would be the ability to pinpoint the intersection between the compared performances, but as peak attainable performance has been disregarded from all previous experiments, such a result would not be meaningful in our context.

The purpose of displaying the two MPI variations Figure 8.18 is to relate the benefit which is obtained by guiding optimization by our model to the cost it carries. Observing that adjusting the application with respect to maximizing overlap comes close to a speedup factor of 2, would ignore the fact that the cost introduced by the relatively elaborate run-time library means that an un-optimized implementation of lighter overhead provides comparable absolute performance. As before, we should also recall that because of the run-time library's reliance on MPI, an implementation which explicitly encoded a similar exploitation of the potential overlap would provide superior execution times, at the expense of increasing program complexity.

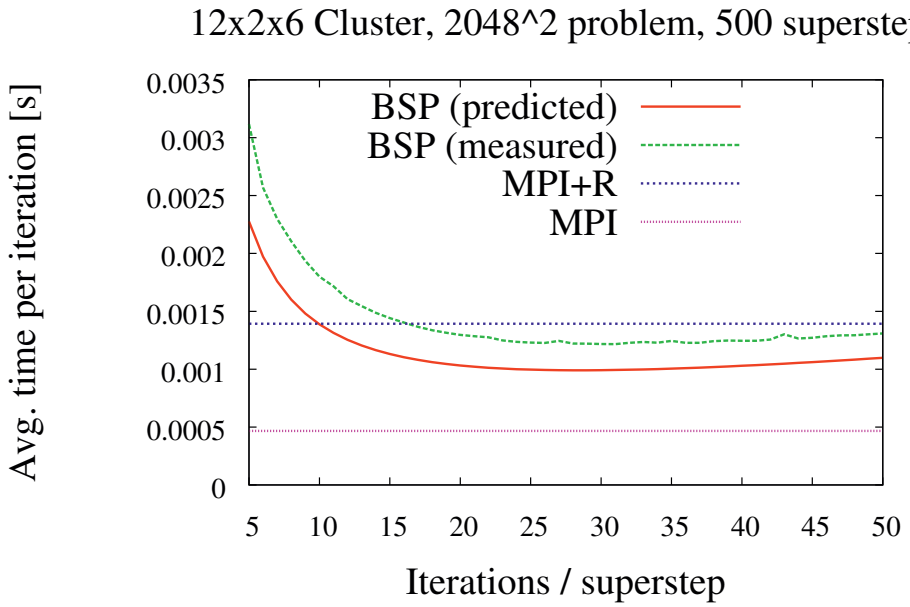

Figure 8.18: C1: Predicted vs. Measured Iteration Time

BSP (predicted)*: Predicted per-iteration performance using predictor programs modified by Figure 8.17 with benchmark data*

BSP (measured)*: Timings of variable-border BSP implementation*

MPI+R*: MPI implementation with global reduction of convergence criterion every synchronization*

MPI*: MPI implementation with implicit synchronization*

Comparison uses input size predicted to improve communication/computation balance for our largest configuration, and a large iteration count to smooth variations due to the nonuniform periodicity of synchronization. The model is sufficient to predict the optimal balance of maskable communication for the application and platform, obtaining superior performance to the reducing MPI implementation, although remaining slower than MPI with implicit synchronization.

Chapter 9

Conclusions and Future Work

In this thesis, we have illustrated how bulk synchronous programming and processing models can be leveraged to capture heterogeneous performance parameters of both programs and execution platforms, and partially or fully automate the analysis of their interaction.

We have used the notion of synchronized supersteps to create a system-wide bound on the amount of outstanding computation and communication at a point of execution, and shown how this permits the work of all processing elements to be captured in P-dimensional vectors of communication and computation time for the present step. In order to obtain these vectors, the computational and communication characteristics of the executing platform can be captured in matrices which contain linear approximations of empirically measured subsystem behavior. The construction of these matrices result in similar shapes, which permits the program and platform characteristics to be combined by element-wise product, and superstep execution time to be obtained by the maximum of horizontal sums. The resulting method couples the computation and communication requirements of a program to an independent profile of the executing platform.

We have seen that synchronization cost can be modeled as the critical path through a weighted dependency graph, producing accurate predictions parametric in system topology and scale. For bulk-synchronous programs, the impact of synchronization is an unavoidable overhead, which suggests that program scalability requires that computational and communication intensities are of sufficient magnitude to make this overhead tolerable. Two case studies illuminated the cost of synchronization in different ways. One showed that the performance parameter space of software synchronization methods admits efficient synchronization methods to be automatically generated. Another showed that cost of synchronization in a stencil application can become a limiting performance factor at modest scales, but that measures to improve the balance of computation and communication per synchronization can be leveraged towards reducing this effect. This demonstrates a technique to determine the impact of synchronization on program behavior, and validates it by practical utilization of its predictions.

The accuracy of obtained performance predictions is limited by two main constraints: the uncertainty inherent to approaching platform parameters as statistics, and corresponding limitations on the accuracy of extrapolated tendencies. The latter is to some extent an artifact of the former, but the effect can be dampened by improving the precision of the parameter benchmarks. Estimates of limited relative error for extended runs can be obtained, but in order to approximate an absolute time of completion, initial approximations must likely be updated with information gathered throughout run time.

A model derived from the proposed framework has proven effective in predicting the optimal point of overlap for a stencil application which provided some flexibility in the trade-off between communication and computation. Although the analysis was partly performed manually, this was restricted to transferring parameter values from one benchmark program to another, suggesting that the entire process would be automatable using only slightly more elaborate software. The framework is thus well suited for the purposes of automatic application performance tuning.

In conclusion, these partial results combine to answer our research question:

How can automation support the analysis of interactions between a parallel algorithm and the executing platform when both show heterogeneous performance characteristics?

We have seen that the modeling of algorithmic requirements and architectural facilities in terms of matrices and vector cost functions yields a system model of sufficient accuracy to approximate and expose optimized parameter choices. This benefit arises from composing both algorithmic and architectural models from piecewise linear functions which can be determined in isolation. The effects of their interactions can then be examined by simulations which are computationally inexpensive compared to the parameter space they capture. The accuracy of the resulting model is obtained by admitting a large number of performance parameters, which can be effectively used in analysis because their manipulation can be effectively automated.

9.1 Process and Publications

The initial motivation of this work was to investigate the impact of heterogeneous performance parameters in COTS computational clusters on the scalability of numerical computations, with a view towards developments in automatic performance tuning methods. That intention permeates the work detailed in this thesis, but published parts of it have altered the perspective slightly.

A shift in the relative costs of moving and manipulating data has been apparent for a number of years, gradually making application performance depend strongly on data movement [11]. This communication requirement rapidly eclipses the cost of computation, whether in the form of memory traffic or network transmissions. For this reason, this thesis is primarily concerned with the impacts of locality and communication.

With the view that memory traffic and remote communication manifest the same principles at different scales, it is natural to begin an exploratory study from the bottom up. Initial studies addressed pure signaling costs on distributed shared memory architectures. This resulted in a conference paper [73], subsequently extended into a journal publication [72], which tested a range of spin-lock synchronization algorithms. A key point from that work is that measurements of synchronization time can expose locality characteristics which are otherwise hidden from software control.

Attention was then devoted to the BSP computational model, as its programming interface permits a direct mapping between software constructs and model terms. A performance comparison of BSP, MPI and hybrid implementations of a stencil application showed that BSP permits the program to expose the potential for sustained performance while up-scaling, but the implementation failed to realize it, creating an artificial communication bottleneck [70]. As this is not necessarily inherent to the model, work was begun to create an implementation and adapted performance model to incorporate locality and overlap. The results of that effort is presented in this thesis.

The aspect of automatic performance tuning relies on accurate performance models to search parameters for optima. Both analytic [71] and empirically driven [32] approaches to dynamic optimization have been investigated in application specific contexts. While such approaches are application dependent by nature, their use is instrumental to exploiting the potential of increasingly complex computing platforms, and most of the work in this thesis is therefore written with a view towards automated model management. An abbreviated description of the method presented in Chapter 7 was published as a conference paper [75], representing a demonstration of the power of these techniques, applied to a general problem which occurred in the course of the work. A summary was also published as a Ph.D. forum short-paper [74], giving an overview of the framework presented here.

9.2 Future Work

While the work presented in this thesis shows that the proposed framework can produce a practically applicable heterogeneous performance model of a simple application on COTS SMP clusters, a number of interesting variables have been treated as fixed, to restrict the scope of the experimental work. This section describes several directions to extend the scope of the developed model.

9.2.1 Profiling Extensions

The developed run-time library provides a great level of detail in the mapping between the execution model, and the elements of the performance model. Specifically, the size and destination of each act of communication are known at the time of its execution, and the cost of computation can be evaluated by the time between initializing transmissions and/or synchronization. Capturing such timings at a local level is feasible through a negligible

overhead, permitting an fairly accurate view of execution to be gathered and analyzed off-line. Providing simple library extensions to programmatically turn profiling on and off reduces the work of instrumenting an application program to a matter of selecting which supersteps should be recorded.

9.2.2 On-Line Adaptivity

An issue with the performance data captured by the benchmarking procedures given in this thesis, is that they are subject to variation at run time, leaving the model to approximate them statistically. While we have seen that a balance between accuracy and variation permits the model to predict the behavior of programs far more complex than the benchmark, the inaccuracy will invariably grow proportionally with the difference of scale between benchmark and application.

Two ways to address the former problem would be to increase the level of detail in the platform model, and to provide hardware which allows tighter bounds on observable performance. While these are both worthwhile endeavors which would benefit the strength of predictions, any nondeterministic behavior will ultimately bound their validity even in the absence of external factors such as O/S or background load. An alternative to such improvements is to exploit the low cost of obtaining performance predictions using our method, noting that the application program itself could construct them using observed values while running. For cases which admit static analysis, this would relieve the programmer of determining the requirements of the program, and it would also apply to programs which alter their requirements based on dynamic results.

A further challenge in this area is to determine an appropriate level of abstraction to provide a simple programming interface for informing the running application about the performance parameters of its platform. Presently, programs are required to read and interpret the structure of the raw benchmark data explicitly. This is impractical, and creates a strong dependency between the program logic and the exact profiling method. An approach to this would be to provide library functions for loading and storing profiles to isolate the internal representation from user code, along with routines for obtaining simulated costs of function calls at run time.

9.2.3 Range Of Applications

The validation of results in this thesis use programs developed for the purpose of our experiments, meaning that a proper real-world scenario has not been tested. Although full application programs written with *BSPlib* are rare, the implementation used in this work has been tested with the programs in *bspedupack* [19], which includes a sparse matrix-vector multiplication function. That implementation unfortunately favors *get* communication, which carries no potential for communication overlap, and the input matrices are formatted by a particular preprocessing tool, which prevents simple testing with arbitrary matrices without additional work. A feasible test would still be to adapt the SpMV im-

plementation to use *put* communication, and measure its performance when applied in a solver for any of the larger systems published *e.g.* at the Matrix Market [79].

9.2.4 Range Of Interconnects

The decision to target commodity clusters with MPI over TCP/IP on gigabit ethernet links poses the question of how our benchmarks would be modified in order to provide predictability using other communication mechanisms. For our purposes, focus on these widely available technologies enabled testing on two independent systems of variable scale and topology. It is important to note that overlapping communication and computation is presently an issue of great concern throughout much research in high performance computing, and there is ongoing development in a wide range of related approaches. The family of Partitioned Global Address Space (PGAS) languages is particularly relevant, using one-sided remote memory access similar to our library, with compilers which admit a less intrusive notation than function calls. The GASNet library provides a one-sided communication functionality very similar to that implemented here, supplying back-ends for a range of interconnects with hardware supported remote memory access.

As the facilities of the GASNet interface allow an almost direct mapping of *BSPlib* operations, an initial version was developed to the point of featuring memory registration, put and synchronize calls, but remained incomplete because the back-ends on our test platforms proved not to overlap asynchronous operations. Completing this implementation to compare performance figures attainable using LAPI and Myrinet back-ends would be an interesting extension of this work.

Bibliography

[1] A. Adinetz. A higher-level and portable approach for gpgpu programming. In *Proceedings of SYRCoSE 2007*, volume 2, pages 49–52, 2007.

[2] A. V. Aho, Lam M. S., R. Sethi, and Ullman J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2006.

[3] S. Akhter and J. Roberts. *Multi-Core Programming: Increasing Performance through Software Multi-threading*. Intel Press, 2006.

[4] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. *Journal of parallel and distributed computing*, 44:71–79, 1997.

[5] Gabrielle Allen, Ian Foster, Nicholas T. Karonis, Matei Ripeanu, Edward Seidel, and Brian Toonen. Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus. In *SC'01: 2001 ACM/IEEE Conference on Supercomputing*, page 52. IEEE Computer Society, 2001.

[6] George Almasi, Siddhartha Chatterjee, Alan Gara, John Gunnels, Manish Gupta, Amy Henning, Jose E. Moreira, and Bob Walkup. Unlocking the performance of the BlueGene/L supercomputer. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 57. IEEE Computer Society, 2004.

[7] AMD. AMD "close to metal" technology unleashes the power of stream computing, 2006. AMD press release [Online; `http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~114147,00.html`, accessed 09-May-2009].

[8] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings National Computer Conference*, pages 483–485, 1967.

[9] E. Anderson, J. Brooks, C. Grassl, and S. Scott. Performance of the CRAY T3E multiprocessor. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 1–17, 1997.

[10] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[12] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, March 2009.

[13] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang, Scott Pakin, and José Carlos Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 1, 2008.

[14] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho. Using performance modeling to design large-scale systems. *Computer*, 42(11):42–49, nov. 2009.

[15] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, apr. 2006.

[16] G. Bilardi, C. Fantozzi, and A. Pietracaprina. On the effectiveness of D-BSP as a bridging model of parallel computation. In *Proc. of the Int. Conference on Computational Science*, volume 2074 of *Lecture Notes in Computer Science*, pages 579–588. Springer-Verlag, 2001.

[17] G. Bilardi, A. Pietracaprina, and G. Pucci. *Handbook of Parallel Computing: Models, Algorithms and Applications*, chapter 1, Decomposable BSP: A Bandwidth-Latency Model for Parallel and Hierarchical Computation. CRC Press, 2007.

[18] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. BSP vs LogP. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, pages 25–32. ACM, 1996.

[19] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Oxford, UK, Mar 2004.

[20] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference*, page 27, 2000.

[21] Taisuke Boku, Mitsuhisa Sato, Akira Ukawa, Daisuke Takahashi, Shinji Sumimoto, Kouichi Kumon, Takashi Moriyama, and Masaaki Shimizu. PACS-CS: A large-scale bandwidth-aware PC cluster for scientific computations. In *CCGRID*, pages 233–240, 2006.

[22] José Luis Bosque and Luis Pastor. A parallel computational model for heterogeneous clusters. *IEEE Transactions on Parallel and Distributed Systems*, 17(12):1390–1400, 2006.

[23] D. P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall Europe, 1994.

[24] Tony M. Brewer. Instruction set innovations for the Convey HC-1 computer. *IEEE Micro*, 30(2):70–79.

[25] Nieves Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *Proceedings of the 34th Conference on Current Trends in Theory and Practice of Computer Science*, number 4910 in Lecture Notes in Computer Science, pages 186–197. Springer, 2008.

[26] F. Capello and D. Etiemble. MPI versus MPI + OpenMP on IBM SP for the NAS benchmarks. In *SC '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 12. IEEE Computer Society, 2000.

[27] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28:1–12, 1993.

[28] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364. ACM, 2010.

[29] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15, 2003.

[30] N. Doulamis, E. Varvarigos, and T. Varvarigou. Fair scheduling algorithms in grids. *IEEE Transactions on Parallel and Distributed Systems*, 18:1630–1684, 2007.

[31] N. Drosinos and N. Koziris. Load balancing hybrid programming models for SMP clusters and fully permutable loops. In *ICPP 2005 Workshops. International Conference Workshops on Parallel Processing*, pages 113–120, 2005.

[32] Anne C. Elster and Jan Christian Meyer. A super-efficient adaptable bit-reversal algorithm for multithreaded architectures. In *23rd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2009.

[33] Z. Fan, F. Qiu, and S. Kaufman, A.and Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[34] Great Lakes Consortium for Petascale Computation. Blue waters project: Sustained petascale computing, 2009. [Online; `http://www.ncsa.uiuc.edu/BlueWaters/bw-booklet.pdf`, accessed 18-July-2010].

[35] Ian Foster and Carl Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., 1999.

[36] Geoffrey C. Fox. An application perspective on high-performance computing and communications. Technical report, Syracuse University, 1996.

[37] A. Goldman, J. G. Peters, and D. Trystram. Exchanging messages of different sizes. *Journal of Parallel and Distributed Computing*, 66(1), 2006.

[38] J. A. Gonzalez, C. Leon, F. Piccoli, M. Printista, J. L. Roda, C. Rodriguez, and F. de Sande. Performance prediction of oblivious bsp programs. In *7th International Euro-Par Conference (Euro-Par 2001)*, number 2150 in Lecture Notes in Computer Science, pages 96–105. Springer, 2001.

[39] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Addison Wesley, 1992.

[40] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1994.

[41] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010. [Online; `http://www.khronos.org/opencl/`, accessed 23-July-2010].

[42] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In *Conference on Computing Frontiers*, pages 1–8, 2006.

[43] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 2007.

[44] J. M. D. Hill and D. B. Skillicorn. Practical barrier synchronisation. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, pages 438–444. IEEE Computer Society Press, 1996.

[45] J. M. D. Hill and D. B. Skillicorn. Lessons learned from implementing BSP. *Journal of Future Generation Computer Systems*, 13(4–5):327–335, 1998.

[46] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[47] Roger W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[48] Qiming Hou, Kun Zhou, and Baining Guo. Bsgp: Bulk-synchronous gpu programming. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–12, New York, NY, USA, 2008. ACM.

[49] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. WCB/McGraw-Hill, 1998.

[50] Intel. Intel unveils new product plans for high-performance computing, 2010. Intel news release [Online; `http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm`, accessed 24-Oct-2010].

[51] Ben H.H. Juurlink and Harry A.G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 339–347. Springer Berlin Heidelberg, 1996.

[52] U. J. Kapasi, W. J. Dally, S. Rixner, and J. D. Owens. The imagine stream processor.

In *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, page 282, 2002.

[53] Alan C. Kay. The early history of smalltalk. In *ACM SIGPLAN Notices*, volume 28, pages 69–95, 1993.

[54] Christoph Kessler. A practical access to the theory of parallel algorithms. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 397–401. ACM, 2004.

[55] Jin-Soo Kim, Soonhoi Ha, and Chu Shik Jhon. Efficient barrier synchronization mechanism for the BSP model on message-passing architectures. In *12th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 255–259, 1998.

[56] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25:21–29, 2005.

[57] S. Kågström. libghthash, 2010. [Online; `http://www.ipd.bth.se/ska/sim_home/libghthash.html`, accessed 10-Dec-2010].

[58] K. Krewell. Cell moves into the limelight. *Microprocessor Report*, feb. 2005.

[59] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, nov. 2005.

[60] Pande lab Stanford University. Folding@home: Statistics reporting, team setup, server statistics, 2010. [Online; `http://folding.stanford.edu/English/Stats`, accessed 18-July-2010].

[61] A. Lastovetsky, V. Rychkov, and M. O'Flynn. Accurate heterogeneous communication models and a software tool for their efficient estimation. *International Journal of High Performance Computing Applications*, 24:34–48, 2010.

[62] James Laudon, Robert Golla, and Greg Grohoski. Throughput-oriented multicore processors. In Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee, editors, *Multicore Processors and Systems*, Integrated Circuits and Systems, pages 205–230. Springer US, 2009.

[63] James Laudon and Daniel Lenoski. The sgi origin: A ccNUMA highly scalable server. In *ISCA*, pages 241–251, 1997.

[64] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, Sep 1979.

[65] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, mar 1992.

[66] E. Lusk and A. Chan. Early experiments with the OpenMP/MPI hybrid programming model, OpenMP in a new era of parallelism. In *Proceedings of the 4th Inter-*

national Workshop on OpenMP (IWOMP 2008), volume 5004 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2008.

[67] M.D. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008.

[68] J. McQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of 5th Berkeley Symp. on Math. Statistics and Probability*, pages 281–298, 1967.

[69] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[70] Jan Christian Meyer and Anne C. Elster. BSP as a performance predictor on heterogeneous interconnect platforms. In *Intl. Workshop on State of the Art in Parallel and Scientific Computing (PARA08)*.

[71] Jan Christian Meyer and Anne C. Elster. A load balancing strategy for computations on large read-only data sets. In *Intl. Workshop on State of the Art in Parallel and Scientific Computing (PARA06)*, number 4699 in Lecture Notes in Computer Science. Springer, 2006.

[72] Jan Christian Meyer and Anne C. Elster. Latency impact on spin-lock algorithms for modern shared memory multiprocessors. *Scalable Computing: Practice and Experience*, 9(3):197–206, 2008.

[73] Jan Christian Meyer and Anne C. Elster. Latency impact on spin-lock algorithms for modern shared memory multiprocessors. In *Proceedings of CISIS2008*, pages 786–791. IEEE, 2008.

[74] Jan Christian Meyer and Anne C. Elster. Performance modeling of heterogeneous systems. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–4, 2010.

[75] Jan Christian Meyer and Anne C. Elster. Optimized barriers for heterogeneous systems using MPI. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011.

[76] M. M. Michael and M. L. Scott. Scalability of atomic primitives on distributed shared memory multiprocessors. Technical Report TR528, University of Rochester Computer Science Department, 1994.

[77] N. Nehra and R. B. Patel. Towards dynamic load balancing in heterogeneous cluster using mobile agent. In *Proceedings of the International Conference on Computational Intelligence and Multimedia Applications*, volume 1, pages 15–21, 2007.

[78] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[79] NIST. Matrix market. [Online; `http://math.nist.gov/MatrixMarket/`, accessed 02-Jan-2011].

[80] NORDUGRID. Extended resource specification language: Reference manual, 2010. [Online; `http://www.nordugrid.org/documents/xrsl.pdf`, accessed 18-July-2010].

[81] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., 1997.

[82] David A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, 2004.

[83] Aashish Phansalkar, Ajay Joshi, Lieven Eeckhout, and Lizy K. John. Measuring program similarity. In *Proceedings of the Intl. Symposium on Performance of Systems and Software*, 2005.

[84] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In *SC 2008: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2008.

[85] R. Rabenseifner. Hybrid parallel programming on HPC platforms. In *Proceedings of Fifth European Workshop on OpenMP (EWOMP'03)*, 2003.

[86] A. D. Reid and Y. Lin. SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 95–104. ACM, 2008.

[87] T. Scogland, P. Balaji, W. Feng, and G. Naryanaswamy. Asymmetric interactions in symmetric multi-core systems: Analysis, enhancements and evaluation. In *SC 2008: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2008.

[88] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[89] A.C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh. Parallelism via multithreaded and multicore cpus. *IEEE Computer*, 43(3):24–32, mar. 2010.

[90] Alan Stewart, Maurice Clint, Joquim GabarrÃş, and Maria J. Serna. Towards formally refining BSP barriers into explicit two-sided communications. In *7th International Euro-Par Conference (Euro-Par 2001)*, number 2150 in Lecture Notes in Computer Science, pages 549–560. Springer, 2001.

[91] Wijnand J. Suijlen. Bsponmpi, 2006. [Online; `http://bsponmpi.sourceforge.net/`, accessed 24-Oct-2010].

[92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, 2001.

[93] Alexandre Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196(1-2):109–130, 1998.

[94] TOP500. Top 500 supercomputing sites, 2012. [Online; `http://www.top500.org`, accessed 25-October-2012].

[95] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[96] Leslie G. Valiant. A bridging model for multi-core computing. In *ESA*, volume 5193 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2008.

[97] T. Valich. AMD ditches close-to-metal, focuses on DX11 and OpenCL, 2008. AMD press release [Online; `http://www.tomshardware.com/news/AMD-stream-processor-GPGPU,6072.html`, accessed 09-May-2009].

[98] R. E. Walpole, Myers R. H., Myers S. L., and K. Ye. *Probability & Statistics for Engineers & Scientists*. Pearson Prentice Hall, 2007.

[99] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27, 2000.

[100] B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Pearson Prentice Hall, 2nd edition, 2005.

[101] Dong Hyuk Woo and H.-H.S. Lee. Extending Amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, dec. 2008.

[102] A. N. Yzelman and Rob H. Bisseling. An object-oriented BSP library for multicore programming. Preprint, 2011.

[103] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Elsevier Academic Press, 2nd edition, 2000.

[104] Xiangliang Zhang, Michele Sebag, and Cecile Germain Renaud. Multi-scale Real-time Grid Monitoring with Job Stream Mining. In *CCGrid*, 2009.