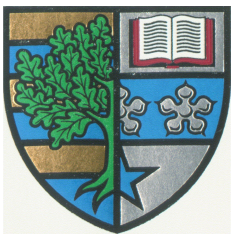
ARCHITECTURE AWARE PARALLEL PROGRAMMING IN GLASGOW PARALLEL HASKELL (GPH)

By

Mustafa KH. Aswad



Submitted for the Degree of

Doctor of Philosophy

at Heriot-Watt University

on Completion of Research in the

School of Mathematical and Computer Sciences

August, 2012

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

Mustafa Kh. Aswad (Candidate)

Phil Trinder, Hans-Wolfgang Loidl (Supervisors)

(Date)

Abstract

General purpose computing architectures are evolving quickly to become many-core and hierarchical: i.e. a core can communicate more quickly locally than globally. To be effective on such architectures, programming models must be aware of the communications hierarchy. This thesis investigates a programming model that aims to share the responsibility of task placement, load balance, thread creation, and synchronisation between the application developer and the runtime system.

The main contribution of this thesis is the development of four new architecture-aware constructs for Glasgow parallel Haskell that exploit information about task size and aim to reduce communication for small tasks, preserve data locality, or to distribute large units of work. We define a semantics for the constructs that specifies the sets of PEs that each construct identifies, and we check four properties of the semantics using QuickCheck.

We report a preliminary investigation of architecture aware programming models that abstract over the new constructs. In particular, we propose architecture aware evaluation strategies and skeletons. We investigate three common paradigms, such as data parallelism, divide-and-conquer and nested parallelism, on hierarchical architectures with up to 224 cores. The results show that the architecture-aware programming model consistently delivers better speedup and scalability than existing constructs, together with a dramatic reduction in the execution time variability.

We present a comparison of functional multicore technologies and it reports some of the first ever multicore results for the Feedback Directed Implicit Parallelism (FDIP) and the semi-explicit parallelism (GpH and Eden) languages. The comparison reflects the growing maturity of the field by systematically evaluating four parallel Haskell implementations on a common multicore architecture. The comparison contrasts the programming effort each language requires with the parallel performance delivered.

We investigate the minimum thread granularity required to achieve satisfactory performance for three implementations parallel functional language on a multicore platform. The results show that GHC-GUM requires a larger thread granularity than Eden and GHC-SMP. The thread granularity rises as the number of cores rises.

Acknowledgements

My praises to God for giving me the good health, the strength of determination and support to finish my work successfully.

I would like to thank my PhD supervisor, Professor Phil Trinder, for all his enthusiasm, guidance and support over the years. He was a constant source of inspiration and I have benefited greatly from his experience and wisdom. I would also like to thank the various second supervisors I have been lucky enough to work with over the course of this PhD, particularly Dr Hans Wolfgang Loidl who was invaluable in providing guidance from a different perspective.

Many thanks to the Libyan higher education sector for offering me this scholarship. And also I would like to thank them for providing the finance support throughout the years of study for me and my family.

Last, but by no means least, I wish to thank my family for their love and affection.

Contents

List of Tables

List of Figures

Glossary

closure represents a unit of computation and is evaluated by jumping to the code it points to. See also thunk.

FCFS First Come First Served (FCFS) is a scheduling algorithm for dynamic real-time computer system in which tasks arrive as random process.

FDIP Feedback Directed Implicit Parallelism (FDIP) is fully implicit parallel Haskell implementations..

GpH-GUM Graph-reduction on a Unified Machine-model (GUM) is a portable, parallel runtime environment for GpH [121], designed for both shared and distributed memory architectures..

GpH-SMP Glasgow Haskell Compiler (GHC) supports shared-memory implementation of Glasgow parallel Haskell (GpH).

NF Normal Form (NF), if an expression is in its normal form status, it means that no further reduction can be made on the expression.

parAtLeast is a parallel coordination construct that indicates that the expression may be executed in parallel and specifies the minimum distance in the communication hierarchy that the expression can be sent.

parBound is a parallel coordination construct that indicates that the expression may be executed in parallel and specifies the maximum distance in the communication hierarchy that the expression can be sent.

parDist is a parallel coordination primitive that indicates that the expression may be executed in parallel and specifies the execution boundaries in the communication hierarchythat the expression can be sent.

parExact is a parallel coordination construct that indicates that the expression may be executed in parallel and specifies a specific execution level in the communication hierarchy that the expression can be sent.

PE Processing Element (PE) is a core within a multicore machine. It may or may not associated with resources such as memory, disk, and screen.

process A process is a program that is running on a computer. A computer is likely to have more processes running than actual program. In parallel programming a program is divided into multiple process with the objective of running a program in less time.

QuickCheck is a tool which aids the Haskell programmer in formulating and

testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random or custom test data input.

rnf Reduce to Normal Form (rnf) means that reduce a given expression to a form contain no reducable expressions.

rwhnf Reduce to Weak Head Normal Form (rwnf) means that reduce a given expression to its top constructor only.

spark Spark is a pointer to an sub-graph indicates that the sub-graph can be evaluated in parallel.

sparkpool is simply a set of pointers to computations that have been sparked by a parallel coordination primitive.

Thread is a sequential computation whose purpose is to reduce a particular sub-graph to normal form. Threads are normally generated by a fork of a program in multiple parallel tasks.

thunk represents an unevaluated expression which will be updated with its result. See also closure.

WHNF Weak Head Normal Form is the evaluation of an expression to its top constructor.

Chapter 1

Introduction

For nearly half a century, processors have been in a constant development to meet the demand for computing capability. The development follows Moore's Law, where the number of transistors in an integrated circuit has doubled every two years [84]. Since 2002, however, the trend has reached a point where little further improvement can be achieved on a single processor, as clock frequency is constrained by power expenditure and heat generation. Hence, architectural design is driving to multicore architectures. However, a conventional single threaded program does not benefit from this new architecture. The performance may also suffer as a result of the lowered clock speed of each core. The shift towards new multicore architectures poses several challenges to software developers. These challenges aim to translate the potential processing power into an equal increase in computational performance.

Furthermore, future architectures will inevitably have a hierarchical, or tree-like, communications structure. The number of cores will steadily increase, as will the level of heterogeneity. Already the most common parallel architectures are clusters of multicore nodes, with three communication level in the hierarchy: on-core, sharing memory, on another node. The communication hierarchy is likely to become deeper as the number of cores increases. Heterogeneous parallel architectures have several different types of processing units, each of which is intended to execute a specific set of tasks. For example, graphics processing units (GPUs) are now present in commodity computer systems [102]. GPUs are specifically designed to take advantage of data parallelism and have significantly better floating point performance than an equivalent CPU. Field Programmable Gate Arrays (FPGAs) are another type of processing unit. FPGAs are non-conventional processors built primarily out of logic blocks connected by programmable wires[35]. FPGAs can be very useful to execute data parallelism.

This research is intended to exploiting only the heterogeneity of cores in clusters of multicores. To exploit such architectures, programming models must be aware of the communication hierarchy. Given the rate of architecture evolution it is important that the programming model preserves performance portability as far as possible.

The target of this work is to develop a high-level programming model for hierarchical architectures. The philosophy is to move the responsibility of task

placement, load balance, thread creation, and synchronisation from the application developer to the runtime system. However, the proposed approach keeps some control of parallel aspects, e.g. data locality and task allocation. The intention is that applications can exploit the potential performance of hierarchical architecture with minimal parallel coordination.

1.1 Thesis Statement

Given the difficulties involved in programming hierarchical architectures, this thesis asserts that an architecture-aware programming model with a high level of abstraction can effectively exploit hierarchical architectures. The assertion is demonstrated by providing an architecture-aware programming model that exploits the underlying architecture. GpH-GUM is extended to record the communication topology of the architecture for task placement. The performance of the architecture-aware programming model is evaluated by measuring a set of demonstration benchmarks on a hierarchical architecture.

The GpH-GUM implementation was ported from the early GHC-4.06 version to the considerably enhanced GHC-6.12 version. The work involved modifying many of the runtime system functions, as GHC-6.12 introduces many new features: new closures types, new data types, and new functions. Some modifications are also required to the compiler. This process took eight months to finish. While the current port successfully executes small programs, there are still

unknown stability issues, in particular the system fails for programs with large data structures.

1.2 Contributions

This thesis investigates parallel functional programming on multicore and distributed memory architectures. A primary contribution is to develop and evaluate new high level architecture-aware programming constructs for hierarchical parallel platforms.

- We propose four new architecture-aware constructs for GpH that exploit information about task size and aim to reduce communication for small tasks, preserve data locality, or to distribute large units of work. We define a semantics for the constructs that specifies the sets of PEs that each construct identifies and we check several properties of the semantics using Quickcheck. We investigate three common paradigms, data parallelism, divide-and-conquer and nested parallelism, on hierarchical architectures with up to 224 cores. The results show that the new constructs consistently deliver better speedup and scalability than existing primitives, together with a dramatic reduction in the execution time variability. At times, speedup is improved by an order of magnitude [9] (Section 5.6).

- We make a preliminary investigation into architecture-aware programming models that abstract over the new constructs. In particular, we propose

architecture-aware evaluation strategies, and architecture-aware skeletons. The abstractions aid performance portability by isolating architecture-specific aspects of the program. The new abstractions are used in the programs measured and the performance results are promising [9] (Section 5.6, Chapter 6).

- We demonstrate the first programming and performance comparison of four functional multicore technologies and report some of the first ever multicore results for two parallel Haskell languages, GpH and Eden. The comparison contrasts the programming effort each language requires with the parallel performance delivered. The study uses 15 typical programs to compare a "no pain", i.e. entirely implicit, parallel language with three "low pain", i.e. semi-explicit languages[1]. There are many encouraging signs for multicore functional languages. The GpH and Eden semi-explicit approaches deliver effective high level coordination, and hence require very small program changes, and modest effort to introduce and tune the parallelism, for a known program [8] (Chapter 3).

- We investigate the most profitable thread granularity required to achieve satisfactory performance for a distributed memory parallel functional language on a multicore platform. We address the question by undertaking a limit study and by studying more typical programs. The programs cover

[1] In contrast, much parallel programming is high pain for high gain. For example GPUs or classic HPC programming requires the programmer to expend intense programming effort to obtain the best possible parallel performance.

both divide-and-conquer and data parallel paradigms. The limit study identifies the most profitable thread granularity required to gain good performance from a message-passing semi-explicit functional language like Eden on a multicore architecture [3] (Chapter 4, Section 4.2).

- We have implemented and investigated the performance of a new formulation of evaluation strategies on a selection of parallel Haskell benchmarks [77] (Chapter 4).

1.3 Thesis Structure

Chapter 2 reviews relevant issues in the area of general purpose parallel computing architectures. It outlines the new trends towards hierarchical communications architectures. Parallel programming approaches are reviewed and discussed. At the end of the chapter, we outline the runtime aspects that are crucial to the implementation of implicit or semi-explicit parallel functional languages.

Chapter 3 presents the findings of the comparison of four different parallel Haskell implementations on multicore architecture. It outlines the key features, programming effort, and performance of each implementation.

Chapter 4 outlines the first uses of a new formulation of evaluation strategies for GpH by undertaking a systematic benchmarking of the new formulation. The chapter also investigates the thread granularity required to achieve acceptable

performance from a distributed-memory parallel functional language, i.e. Eden and GpH, on multicores.

Chapter 5 presents the design of new architecture-aware constructs for the GpH language which can exploit information about task size and aim to reduce communication for small tasks, preserve data locality, or distribute large units of work. The architecture-aware constructs provide multiple levels of parallelism to maximise the performance of new architectures. The behaviour of the architecture-aware constructs is also investigated.

Chapter 6 presents preliminary investigations of architecture-aware programming models that abstract over the new constructs. Specifically, we present some key abstractions and demonstrate some architecture-aware evaluation strategies and skeletons.

Chapter 7 concludes and suggests the opportunities for future work that may extend the work presented in this dissertation.

1.4 Authorship and Publications

Unless otherwise stated the work presented throughout this doctoral thesis was authored by myself and the work contained herein is my own. As a result of the research activities the following research papers were published:

[8] *Aswad, M.*, Trinder, P., Al Zain, A., Michaelson, G., and Berthold, J. Low

Pain vs No Pain Multi-core Haskells. TFP09, Sympo- sium on Trends in Functional Programming, Komarno, Slovakia 10 (June 2009), pp. 4963.

[3] Al Zain, A., Hammond, K., Berthold, J., Trinder, P., Michaelson, G., and *Aswad, M.* Low-pain, High-gain Multicore Programming in Haskell: Coordinating Irregular Symbolic Computations on Multicore Architectures. In Proceedings of the 4th workshop on Declarative aspects of multicore programming (2009), ACM, pp. 2536.

[77] Marlow, S., Maier, P., Loidl, H.-W., *Aswad, M.* K., and Trinder, P. Seq no more: Better Strategies for Parallel Haskell. In Proceedings of the 3rd ACM SIGPLAN symposium on Haskell (Baltimore, MD, United States, Sept. 2010), ACM Press, pp. 91102.

[9] *Aswad, M.*, Trinder, P. W., and Loidl, H. Architecture-Aware Parallel Programming in Glasgow Parallel Haskell (GPH). In Proceedings of the International Conference on Computational Science (ICCS) (Omaha, USA, June 2012), Procedia Computer Science, pp. 18071816.

The content of the papers is related to the chapters of the thesis as follows:

- In [8], we present a programming and performance comparison of functional multicore technologies material for four parallel Haskell implementations (Chapter 3).

- In [77], we present a performance evaluation of the new Eval monad strategies material (Section 4.1.3).

- In Section 6 of [3], we present the results of the thread granularity limited study on multicores (Section 4.2).

- In [9], we describe the design of the new architecture-aware constructs for the parallel Haskell extension GpH material (Chapter 5).

Chapter 2

Background

2.1 Parallel Hardware

Up until recently, improvements in performance of commodity machines relied on increased clock frequency and instruction-level parallelism. However, the amount of instruction-level parallelism that can be extracted from sequential programs is limited [110], and since 2002, CPU clock frequency increases have stalled, due to power and heat issues [101]. Therefore, the whole microprocessor industry has turned to manufacturing processors incorporating multiple cores onto a single die [12]. These architectures are known as multicores. These multicore machines are becoming the predominant architecture for general propose computing systems. In such environments, heterogeneity occurs in several forms: e.g. PEs may have different instruction sets, different operating systems, or there may be different network connections between PEs [111].

2.1.1 Classification by Memory Structure

Parallel architecture can be classified as shared memory or distributed memory. In a shared memory system, cores share a single address memory space. Typically, this is implemented through a shared bus, although this design is limited to a few dozen cores. In a distributed memory system, each core has exclusive access to its own local memory, whereas this memory space can be logically or physically distributed [106]. Separately from the physical structure of memory, it can be logically distributed or virtually shared. These forms of logical organisation can be combined with underlying physical structure by e.g. implementing a virtual shared memory abstraction. In addition to the memory structure, each core (Processor Element) has a private cache and possibly shares the cache with other local cores (e.g. `L2` cache). The `L1` cache is located close to the core and is used to store blocks of values required by the process executing on the core to exploit data locality. Cores may share the standard Random Access Memory (RAM) through a common bus.

Thus, modern architectures realise a deep memory hierarchy of remote memory, local memory, (several levels of) cache and register, with decreasing access time and decreasing size. Efficiently exploiting this memory hierarchy is crucial for high-performance computing.

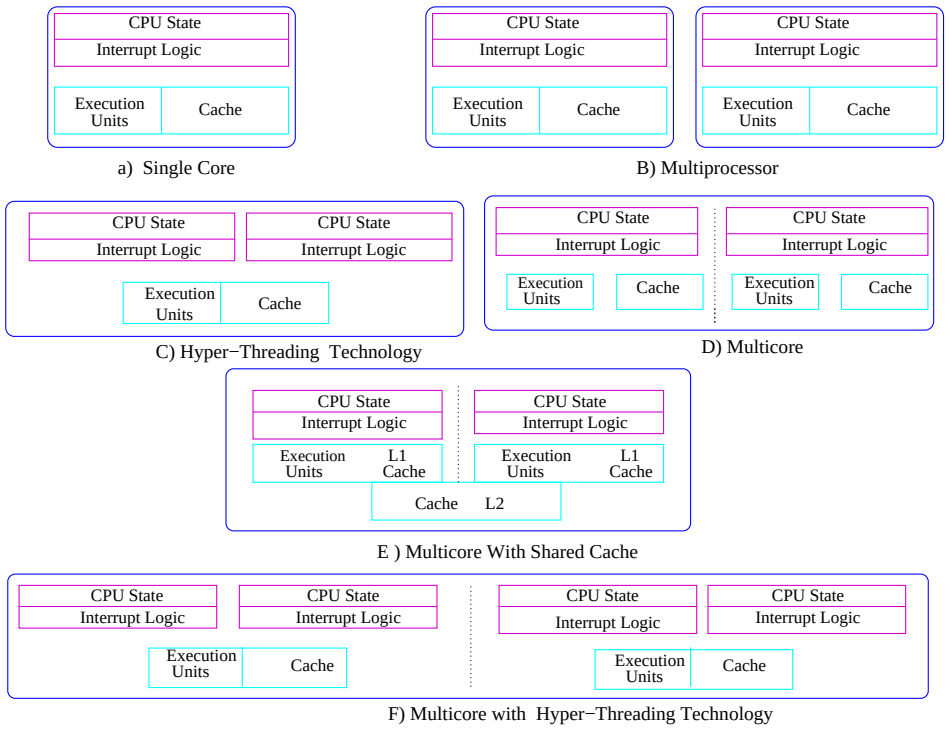


Figure 1: Comparison of Single Core and Different Multicore Architectures([2])

2.1.2 Multi-core Architecture

Multicore devices have been around for many years, in different forms [113]. Most of them are homogeneous devices consisting of several identical cores to form a multicore. However, future multicore systems will be heterogeneous multicore devices, including several cores with different capability [42, 60]. The homogeneous option is used in most of today's systems because it is easy to implement. An example a simple multicore system is a dual core system containing two identical cores within a single die, which aims to double the performance [62, 20]. Figure 1 illustrates typical multicore architecture, specifically some Intel multicore architectures and their memory topologies:

a) A single core consists of core and cache.

b) A multiprocessor consists of two identical chips where each has its core and cache.

c) Hyper-Threading on a single chip consists of a core with two or more CPUs and interrupt logic.

d) A multicore chip consists of two cores where each has its own cache.

e) A multicore consists of two cores sharing only L2 cache.

f) A multicore consists of two hyper-threaded cores sharing a cache.

In a dual core, the performance can be increased without any software modification as the operating system can dedicate one core for the main application and the other core for a specific task such as interrupt handling. However, in a multiple core device, it is desirable to use several cores to execute one user application and the application must be reshaped to make use of all the cores for optimal performance.

2.1.3 Homogeneous vs Heterogeneous Multicore

Today systems for massive parallelism often consist of several homogeneous clusters of different speeds and sizes interconnected through traditional networks, which means the architecture becomes heterogeneous [112]. A heterogeneous architecture has the advantage that it may improve performance and efficiency, both

through increased specialisation of core types and the large numbers of processors. The objective is to provide better balance between sequential and parallel workload. The different core capability of heterogeneous architectures are usually developed in a way that each core is dedicated to a different workload. The most powerful core is assigned to perform heavy tasks, e.g. operating system actions. In contrast, cores in homogeneous architectures are treated equally but not specialising any of the cores for a particular task. It is therefore very difficult to distribute an irregular parallel workload in a manner that balances the workload between cores [82]. The property of easy construction and mapping tasks in homogeneous parallel architectures, which are usually in a single machine or single cluster, will soon be unavailable because of the increasing number of cores in a single chip. This increase will allow the construction of multicore machines with cores of different capabilities.

2.1.4 Distributed Computing

A distributed memory system is a system in which the processing elements are connected by a network. Several computing paradigms can be considered as distributed computing: cluster computing, Grid computing and, more recently, Cloud computing. Sadashiv [104] has presented a detailed comparison between the three architectures. Buyya [23] defines these paradigms as *"a cluster is a category of parallel and distributed platform, which consists of a collection of*

interconnected standard computers working jointly as a single incorporated computing resource". A Grid is a parallel and distributed architecture that enables sharing of resources dynamically at runtime system level [1]. A Cloud is a parallel and distributed architecture consisting of a collection of interconnected and virtual parallel computers. It can be provided dynamically and viewed as a single or several computing resources depending on consumers needs.

This thesis primarily focuses on clusters, e.g. the proposed model is evaluated on a group of interconnecting clusters in Chapter 6.

Cluster Computing consists of multiple standalone computers connected by a local area network. Computers in one cluster should be identical to minimise the difficulty of achievement of the load balance. The most common type of such a cluster is the Beowulf cluster [99], which consist of multiple interconnected (identical) commercial off-the-shelf computers. Commonly, the ordinary network technology is used [109]. With multicores becoming standard machines, the individual nodes in the clusters are themselves dominant. This architectural change has complicated the implementation of load balancing and has introduced additional challenges to parallel programming developers. They have to take advantage of multicore features in their software development. One of the most important features that needs to be captured is the hierarchical memory structure, which can be achieved by exploiting data locality in a parallel program [5].

Grid Computing is a collection of computer resources where computing is executed over a network. This approach is viewed as one set of a virtual global community, where resources are heterogeneous. This virtual community is geographically distributed and belongs to different organisations. Grid are used to solve comprehensive computational problems in science, engineering, and commerce [11].

Hybrid Architecture is a high speed parallel computing architecture consisting of several nodes with private address space. A hybrid architecture can be viewed as a combination of Von Neumann features and Dataflow features [50]. It is an array of identical processors, connected through a suitable switching network to a global memory. Moreover, a hybrid architecture can take different forms: it can be a small cluster of single core machines or cluster of multicore machines. It can also be geographically distributed clusters connected by a network.

Hierarchical Architectures: These are geographically distributed hierarchical architectures [22]. Moreover, even common parallel clusters of multicore nodes can be considered as hierarchical, with three hierarchy levels. Threads on the same core can communicate most efficiently with a thread on the same core, more slowly with a thread on another core in the node, and even more slowly with threads on remote nodes. The communication hierarchy is likely to become deeper as the number of cores increases. For example, the number of cores sharing the same memory is likely to be restricted, and hence many core architectures

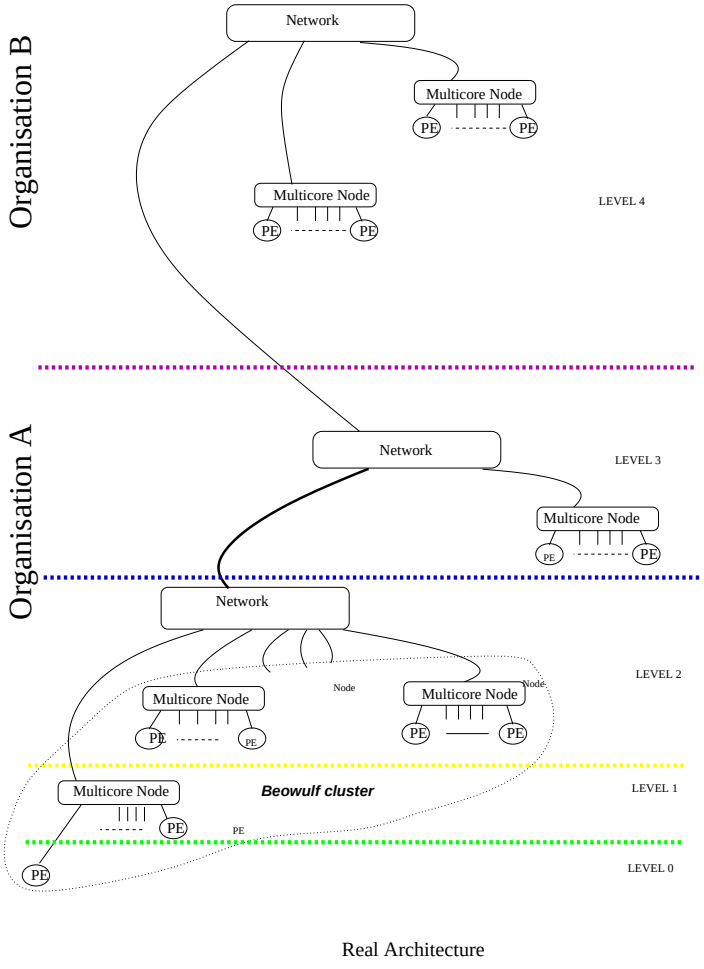


Figure 2: Hierarchical Architectures

may introduce another level within a node.

Figure 2 illustrates a hierarchical architecture of two organisations A & B. The architecture can be viewed as an unbalanced tree of nodes.

2.1.5 Summary

The increasing deployment of multicore architectures brings parallel computing into the mainstream. This paradigm shift creates a challenge of parallel programming and system understanding in general. Clusters are the dominant server technology and can be constructed from various computer types.

Memory	Single Core	Multi-Core L1+L2	Symmetric Multi-Core (SMP)	Distributed Memory
Own Cache	Yes	Yes		Yes
Own Ram	Yes			Yes
Shared Cache		Yes	Yes	
Shared Ram		Yes	Yes	
Communication		coordinated accesses	coordinated accesses	Message Passing

Table 1: Memory Classification

Table 1 summarises the types of computers from a memory perspective. The first column shows the comparison categories. The second column shows a single core architecture where nothing is shared. The third column presents a multicore architecture in which memory appears in the form of a hierarchy: each core has it own cache and two cores have a shared cache, and in addition to that all cores share the same RAM. The fourth column shows a symmetric multicore architecture; cores share both cache and RAM. The final column shows a distributed memory architecture, where each core has a local cache and RAM, and a bus or network interconnects all cores. Cores within multicore and SMP communicate with each other through memory access. In contrast, cores in distributed memory communicate through message passing.

Heterogeneous multicores and clusters consisting of these different kinds of processors are becoming the predominant architectures. This trend needs a matching programming model to exploit the potential performance.

2.2 Parallel Programming Classifications

Since hardware is parallel by default nowadays, software must become parallel, too. This requires modifications to the existing software to cope with new parallel architectures. For many application a key requirement of the parallel programming model is that it has to be easy. In particular, it should not be much more difficult than the traditional sequential programming model. Of course, there are other requirements like maintaining the existing sequential software, portability and, most of all, the performance. The main challenge in parallel language design is to find a suitable balance between performance and abstraction. To start with, the application must be designed in a manner that can be parallelised i.e. decomposed into a number of tasks that can be executed in parallel. This section discusses a number of alternative parallel programming languages.

Before the parallel software development process is studied in general, we have to explain the two possible types of parallelism in any given application: data parallelism or task parallelism.

Data parallelism describes a type of parallelism where a function is applied to multiple data items simultaneously. It can be flat data parallelism, where the function is applied to a one dimensional data set, or nested data parallelism, where the function is applied to a many dimensional data set. It is not necessarily the case that the same function is applied to both dimensions. It may happen that different functions are applied to different dimensions. Parallel array computation

is the most common approach used by many data parallel languages such as NESL [21], High Performance Fortran (HPF) [29], ZPL [26], and Hierarchically Tiled Arrays (HTAs) [17].

Task parallelism is a form of parallelism where computational units, e.g. functions, are distributed to be executed in parallel. The key notion is that an application has to be divided into several independent tasks. These tasks can be mapped automatically onto physical processors [6]. The scheduling of the tasks to the physical processors is usually performed by the implementation.

There are a number of important parallel paradigms and this section focuses on those related to the thesis. Some important paradigms not covered here include Single Program Multiple Data (SPMD) [51], Bulk Synchronous Processes (BSP) [123], and the increasingly important GPU programming models like CUDA [102] and OpenCL [85].

2.2.1 Parallel Software Development

Parallel software development has a number of distinctive issues in addition to issues inherited from sequential software development. Either the programmer or the compiler is responsible for addressing these issues [41, 96]. The following are the distinctive issues related to parallel software development:

- **Exploitation of the Potential Parallelism:** the algorithm should indicate the source of parallelism existing in the problem. This involves partitioning the program into sequential parts that can be performed in parallel. However, exploiting every single source of parallelism is not always the best choice because this generates extremely small pieces of computation, for which the system overhead dominates the computation.

- **Resource Utilisation:** the exploited parallelism has to be mapped to parallel hardware in a way to make efficient use of the available resources. For example a program should utilise a suitable number of cores, and make limited memory and communication demands. The efficiency of the parallelism depends on the programmer and on the parallel software model involved. High efficiency can be achieved by statically grouping tasks and scheduling their execution on cores equally or by achieving the load balance at runtime by assigning tasks to idle cores.

- **Synchronisation & Communication:** communication in parallel programming refers to the information exchange between parallel computations. The information exchange between parallel computations requires some form of synchronisation, either implicit or explicit. Synchronisation can be achieved by either shared variable communication primitives that control access to critical sections or message passing communication primitives, which guarantee exclusive access to shared objects. It is crucial to

avoid deadlock, starvation and non-termination.

- **Correctness** of parallel implementation needs to be preserved. Because parallelisation often requires structural changes to the code, as well as additional coordination code, assuring that the parallel program produces the same result as the sequential program is non-trivial. Not only must the result be correct but the performance must be reasonable. In parallel programs, we may get the correct result, but at the price of losing performance as a result of unequal distribution of tasks among processors.

- **Parallel Software Debugging** is the process of testing the parallel program and improving the performance. Tracking errors in parallel software (especially performance errors) is exceptionally difficult. There are tools to aid this process that give an overview of the cores' status at any given moment of the program execution. This is manageable for a small number of cores, but for a large number of cores, say thousands, this will be extremely difficult.

2.2.2 Parallel Programming Models and Languages

Parallel programming models can be classified in several ways to determine their suitability for challenges such as portability, efficiency and ease of programming. Unfortunately, these properties conflict with each other. For example, for ease of programing, the programmer can ignore the underlying architecture in their

application. However, high performance may require that the programmer gives some details about the underlying architecture. The more specified the architecture, the less the application will be portable and thus the higher the maintenance cost.

Parallel programming languages can be classified into imperative approaches and declarative approaches. An imperative parallel programming language [118] represents a computation as a group of interacting parallel tasks that communicate and synchronise using message passing, shared memory locks or via remote procedure call. The strength of the imperative is in explicitly representing tasks and operations from the problem specification and thus achieving detailed control of the parallel programming. Its disadvantage is the difficulty of expressing parallelism and the interaction patterns of its operations. In contrast, declarative parallel programming [73] represents the logic of computation without describing its control flow. Due to referential transparency, expressions can be evaluated in any order, in particular also in parallel. Thus, correctness of the parallel programming is often trivial. The coordination of the parallelism is typically managed by the compiler and by the runtime system. Consequently, programmers only have to deal with the decomposition of their problems into parallel tasks.

Skillicorn and Talia [106] categorised parallel models based on their degree of abstraction.

- **Nothing Explicit (Implicit)** models hide all parallel details from the

programmer. The programmer continues to program in a sequential manner. These are easy to use and highly abstract, but potentially inefficient.

- **Explicit Parallelism (semi-Explicit)** models require a programmer to expose the inherent parallelism in the program. However, the runtime system is responsible for determination of the actual parallelism, execution, mapping, communication and synchronisation.

- **Explicit Decomposition** models require a programmer to indicate the potential parallelism inherited from the program and divide this parallelism into tasks but to allow the placement of the tasks be decided by the runtime system.

- **Explicit Mapping models** require a programmer to indicate the potential parallelism inherited from the program: how the program is divided into pieces and where to place each task. However, the communication and synchronisation are left to the runtime system.

- **Everything Explicit models** require a programmer to identify all parallel details activities. It is extremely difficult to write an application using such models, because both correctness and performance can only be reached by knowing large number of complex details of the system and underlying architecture.

Loogen[71] classifies parallel programming models with respect to the level of

	Control Parallelism	Data Parallel
Implicit	automatic parallelisation	data parallel languages
Controlled semi-Explicit	annotation based languages evaluation strategies skeleton languages	high-level data parallelism
Controlled Explicit	process control languages message passing languages concurrent languages	

Table 2: Classification of Parallel Models

control and type of parallelism. The models are split into three categories. With implicit approaches, the system tries automatically to exploit the parallelism that is inherent in the semantics. With controlled approaches, the programmer is involved in inserting a notation either to exploit parallelism or to control the execution behaviour. With explicit approaches, a programmer is required to describe all parallel aspects in the program. Table 2 shows Loogen's classification.

2.2.3 Implicit Models

In an implicit model, the compiler automatically exploits the parallelism available in the computations [71]. There is no need for special directives, operators or functions to enable parallelism. Therefore, the job of the implementer of the language becomes much harder, since the compiler and/or runtime system must infer all parallel structures of the eventual program [59]. Ideally, automatic parallelisation would exploit all the potential parallelism of reasonable granularity. Whether generating parallelism is worthwhile heavily depends on the size of the

expression and the underlying architecture characteristics. Therefore, information about thread granularity is often involved in deciding the size of expression that should be evaluated in parallel. There are many implicit parallel models available. The rest of this section discusses some prominent systems supporting implicit parallelism.

Intel has developed an automatic partitioning packet processing applications compiler for pipelined architectures [33]. This approach automatically splits a sequential C code into coordinated pipeline parallel subtasks. These tasks are mapped to processing elements of a network or parallel architecture. The technique allows a minimisation of data transfer between subtasks and a balance of processing tasks in the pipeline. Using this compiler, the programmer can continue to write applications in a sequential manner even if they should be executed on heterogeneous multicore architecture. Nevertheless, the extraction of parallelisation depends on the amount of inherent data parallelism from the algorithm and its dependencies.

Feedback Directed Implicit Parallelism (FDIP) is an implicitly parallel implementation of the Haskell functional language [46]. FDIP extracts potential parallelism from sequential Haskell code in four processing stages. In the first stage, it executes and profiles the sequential code. In the second stage, it analyses the profile output to identify useful sources of parallelism. A Haskell program

usually contains a large number of potential computation *thunks*, which may represent useful sources of parallelism. However, the hard question is: which of these are suitable for potential parallelism. In the third stage, the program is automatically recompiled to introduce parallelism at the identified sites. In the final stage, the output of stage three is executed on sophisticated mechanisms implemented in the GHC runtime system. The implementation dynamically manages thread generation and load balance.

2.2.4 Semi-Explicit Parallel Models

A semi-explicit parallel model requires the programmer to indicate the source of potential parallelism in the algorithm, whereas the decision of realising the actual parallelism is left to the compiler or to the runtime system. Annotations to the compiler are used to identify sources of parallelism in the algorithm, and hence to control the parallel behaviour of the algorithm, whilst hiding the low level implementation details from the programmer. The annotations affect only the run-time behaviour of programs, but not the result.

OpenMP uses an imperative parallel programming style [28]. It is a portable programming interface for shared memory multithreaded programming using C/C++ and Fortran as host languages. OpenMP consists of a set of compiler directives, library routines, and environment variables that affect run-time behaviour. OpenMP uses a fork-join threading model; a master thread forks a

task into a number of worker threads that share the work and then wait until they finish to join before continuing. OpenMP is a scalable model that gives programmers a simple and flexible interface for developing parallel applications for a range of parallel architectures. The model is identified as easy to use and portable. The programmer does not need to put significant effort into parallelising the existing sequential program. However, this is not always the case, as the multicore resources are not fully utilised if the programmer is not expert in parallel programming.

High Performance Fortran (HPF) is a standardised imperative parallel language, which focuses mainly on the issues of distributing data across the memories of a distributed memory multicore [29]. It is an extension of Fortran90 that exploits data parallelism. Data parallelism exists when a single operation is carried out over a collection of data. HPF adds a set of directives such as `Processors`, `Distribute` and `alignments` directives. `Processors` allocates the number of abstract cores. It is usually equal to the number of actual physical cores. `Distribute` distributes the data in an array along cores in a combination of operational groups, and or block modes.

The Manticore project combines NESL style data parallelism with more general task parallelism as found in some other languages such as Concurrent ML (CML) [98]. In a sense, it represents a heterogeneous parallel language [40, 25]. Manticore combines features of CML supporting explicit concurrency as well as

of NESL data parallelism. It is based on three components: sequential functional programming features of SML; explicit concurrent programming primitives using threads and synchronous message passing inherited from CML; and implicit nested data parallelism from NESL and Nepal[24]. The underlying hardware topology can be hidden behind data and type abstraction using a CML abstraction mechanism called first-class synchronous operations. Using event values, programmers can easily specify very complicated communication and synchronisation protocols. Parallel arrays in Manticore can be of any type and they can be nested. The compiler maps parallel array operations onto appropriate parallel architecture.

NESL is a strict, strongly-typed, nested data-parallel language with implicit parallelism and implicit thread interaction [21]. It has been implemented on a variety of parallel architectures, including several vector computers. NESL fully supports nested sequences and nested parallelism. The language allows a programmer to perform a higher-order function on a list in parallel. NESL is loosely based on the ML functional language. In NESL the `Apply-to-each` is the central construct to exploit parallelism. This construct uses a set-like notation. NESL uses a method based on asynchronous core groups to reduce communication and a run-time load-balancing system to cope with dynamic data distributions. This is achieved by translating the user's algorithm into ANSI C with MPI calls, and linking this code with an MPI (Message Passing Interface) library.

Glasgow Parallel Haskell (GpH) is a modest extension of Haskell98 [121]. It uses a thread-based approach to parallelism. This approach allows the creation of parallel threads, but does not require mechanisms to control them. Synchronisation is implicit through shared variables. It exploits different types of parallelism such as data parallelism and divide-and-conquer parallelism by defining a higher order function which combines coordination primitives. Evaluation strategies [121] are the preferred mechanism of high-level coordination of parallelism. GpH extends Haskell98 with parallel `par` and sequential `pseq` composition primitives (see Figure 3). Denotationally, both primitives are projections onto the second argument. Operationally, `pseq` causes the first argument to be evaluated before the second and `par` indicates that the first argument can be evaluated in parallel with the second argument. The latter operation is called `sparking` of parallelism and it is implemented using a lazy task creation approach. A `spark` in GpH is not immediately converted to a thread. The runtime environment is responsible for determining which sparks are going to be converted to parallel threads based on load and other information.

```
par :: a -> b -> b   -- parallel composition
pseq :: a -> b -> b   -- sequential composition
```

Figure 3: Types of the basic coordination constructs in GpH

Several implementations of GpH are available for parallel architectures. The GpH-SMP is an optimised shared memory implementation integrated into GHC

from version 6.6 onwards [16]. The GpH-GUM implementation is a message-passing implementing a virtual shared heap [121]. The GpH-GUM has been ported from GHC version (GHC-4.06) to the recent GHC version (GHC-6.12) as part of this thesis.

Caliban is a declarative parallel programming language defined on top of the Haskell functional programming language [57, 114]. Caliban defines aspects of parallel runtime behaviours of the application program using annotations. The annotations do not effect the result produced by the program, although they can affect the termination properties of the program. As with GpH, the coordination level does not affect the result, although it can affect the termination properties of the program. The coordination-level entities in Caliban are processes that are connected by streams. Processes in the Caliban form a cyclic graph whose nodes are the physical processors and arcs are streams of data. The potential parallelism is gained from the data dependencies of the application. The `Node` construct is used to place the expression in a separate processor, where the `Arc` construct is used to connect nodes reflecting the data dependencies in the programme. The use of higher-order functions in the Caliban coordination language facilitates code re-use.

Clean is a pure concurrent lazy functional language[95]. It provides higher-order functions supporting concurrent processes and distributed execution, making use of an I/O library included in the language. Clean can dynamically create

processes, which may run interleaved or in parallel. The interconnection between processes such as communication and synchronisation is handled automatically by the runtime system. However, the process topology can be defined by programmer using higher-order functions. Clean has been extended with a set of primitives in D-CLean [48]. These new primitives allow programmers to distribute computation in several layers. These layers can be distributed easily over clusters. The details of the underlying architecture can be hidden from the programmer using skeletons. The primitives that D-CLean provide are: `DStart` starts the distributed computation. `DStop` receives and saves the result of the computation. `DApply` applies the same function expression in parallel `n` times.

Eden [15] extends Haskell with syntactic constructs to explicitly define and instantiate processes. In contrast to the other languages, such direct Eden programming exposes parallel tasks at the language level, and requires the programmer to manage them using the control mechanisms provided in the language. In practice however, Eden provides libraries of skeletons [13] for parallelising applications.

Eden supports a distributed memory parallel paradigm. That is, processes share no values, and communicate only by messages. It might be thought that such a paradigm would not be suitable for parallelism on shared-memory multicore architectures; however recent results have shown good performance [16].

In direct usage, Eden is explicit about process creation and about the communication topology, but implicit about other control issues, such as sending

and receiving messages and process placement. Granularity is under the programmer's control because he/she decides which expressions must be evaluated as parallel processes, and also some control of the load balancing is possible, at the program level.

Eden provides process abstractions and process instantiations for coordination, as shown in Figure 4.

```
newtype Process a b = ...

-- process abstraction (language construct)
process (Trans a, Trans b) => (a->b) -> Process a b

-- process instantiation
(#) :: (Trans a, Trans b) =>
        Process a b -> a -> b

-- non-deterministic merge process
merge :: Process [[a]] [a]
```

Figure 4: Basic Coordination Constructs in Eden

The expression `process (\ x -> e)` of a predefined polymorphic type `Process a b` defines a *process abstraction* that takes a function of type `(a->b)` and constructs an analogous `Process a b`. When instantiated, processes are executed in parallel and if the output or input expression is a tuple, a separate concurrent thread is created for the evaluation of each tuple element. We refer to each output tuple element as a channel. A *process instantiation* is achieved with the predefined infix operator `(#)`, and the `Trans` context supports the transmission of lists as streams and the creation of threads for tuple elements. Each time an

expression `e1 # e2` is evaluated, a new process is created to evaluate the application of `e1` to `e2`. We will refer to the latter as the *child* process, and to the owner of the instantiation expression as the *parent* process. The instantiation semantics specifies in which processes these expressions shall be evaluated: (1) Expression `e1` together with its whole environment is *copied* in the current evaluation state to a new processor, and the child process is created there to evaluate the application of `e1` to `e2`, where `e2` must be remotely received. (2) Expression `e2` is eagerly evaluated in the parent process. The resulting full normal form data is communicated to the child process as its input argument.

Once processes are created, only fully evaluated data objects are communicated. The only exceptions are lists: they are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Processes trying to access input not yet available are temporarily suspended. This is the only synchronising mechanism in Eden.

Algorithmic skeletons in Eden abstract common patterns of parallel evaluation into higher order functions [32]. They simplify the development of parallel programs by hiding coordination details from the programmer and may provide ready-made parallel cost models.

Para-Functional parallel programming is based on a functional programming model enhanced with features that allow programs to be mapped to specific multiprocessor topologies [49]. There are extended control clauses, which can be

used to express quite sophisticated placement and evaluation schemes. These control clauses effectively form a separation between the language and process control. Parallelism may be exploited by a number of annotations such as `$on, send, receive`. The compiler generates parallel PACLIB C with explicit task creation and synchronisation statements.

X10 is a parallel programming model under development at IBM for programming hierarchical parallelism [30]. X10 increases performance by integrating new constructs (notably, places, regions and distributions) to model hierarchical parallelism and nonuniform data access. X10 provides four storage classes: (1) `Activity-local` class to store private data structures of activities such as a stack for the local thread information and it is located in the activity `place` (thread-local). (2) `Place-local` is a local class for a `place`, but can be accessed coherently by all activities executing in the same place. (3) Partitioned-global (shared memory) is a global address space containing elements of `place` type. The elements are accessible by both local and remote activities. (4) `Values` (Virtual shared memory) is a class used to store values that can be moved between places depending on the implementation.

The `foreach` construct serves as a convenient mechanism for spawning local activities. The `ateach` construct serves as a convenient mechanism for spawning remote activities. X10 `place` and its `activities` can be mapped to underlying heterogeneous nodes, taking advantage of the hardware features of each node

while maintaining the portability of code.

2.2.5 Explicit Models

In explicit models, the programmer is required to describe details of the parallel coordination. In other words, it is the responsibility of the programmer to describe many implementation details. This detailed control allows the programmer to tune parallel performance and tailor it to one particular architecture, at the cost of programming simplicity and performance portability. In the research community, a consensus is emerging that such models are problematic in the direction of parallel programming, despite the good performance which can be obtained from it. The main problem with this model is that it requires detailed expert knowledge about parallel programming to achieve good parallel performance [56]. It also requires manual code changing if a program is executed on another architecture.

The rest of this section describes concrete systems for explicit parallelism.

Message Passing Interface (MPI) [125] standard defines an interface for sending and receiving messages. Specifically the interface includes point-to-point communication functions, `send` operations performing a data transfer between two concurrently executing tasks, and `receive` operations to accept data from another processor into program memory space. It also has other operations, such as broadcast barriers and reduction that explicitly involve a group of processors.

It is heavily used in high performance computing and, with considerable tuning, delivers an acceptable performance across a wide range of architectures. MPI is a MIMD style model. However a shared-memory style can be simulated using send and receive messages of MPI. MPI does not provide the dynamic creation or deletion of processes during a program runtime (the total number of processes is fixed) [107].

Parallel Virtual Machine (PVM) [88] is a parallel programming environment for developing and executing large scale applications consisting of several independent interacting tasks. It is developed to work on a collection of heterogeneous architectures interconnected by common networks. The PVM system provides a number of user interface primitives: for the invocation of processes, message manipulation, broadcasting, synchronisation via barriers, mutual exclusion, and shared memory. A parallel application views the PVM system as a general, flexible parallel computation resource that can be utilised through invoking its functions.

Erlang is a parallel programming language. It is a concurrent, real-time, distributed fault-tolerant programming language designed at the Ericsson Computer Science Laboratory in 1986 [61]. Erlang uses an explicit model, in which the programmer is required to explicitly specify which tasks are to be executed in parallel. It has primitives for multiprocessing: `spawn` starts a parallel computation (called a process); `send` sends a message to a process; and `receive` receives a message

from a process. These primitives are for controlling parallel tasks. The overhead associated with the creation of an Erlang process is really small. This is one of the main reasons why Erlang is of interest for programming parallel architectures.

2.2.6 Hybrid Programming Models

A hybrid programming model combines shared-memory and distributed-memory. This style of programming is used to program architectures similar to the architecture presented earlier in Figure 2. A common way to develop a hybrid model is a combination of OpenMP and MPI libraries. The idea is to use OpenMP threads to exploit the multiple cores per node while using MPI to communicate among the nodes [75]. The weakness of these models is that they are heavily dependent on the hardware, the OpenMP and MPI implementations, and the skill of the application programmer. Rabenseifner [97] categorised hybrid programming using MPI and OpenMP as follows:

1. Using only MPI, where each SMP node represents a MPI process. The MPI library communicates using a shared memory between MPI processes on the same SMP node, and by message passing between MPI processes on different nodes.

2. Hybrid MPI+OpenMP: each MPI process consists of several OpenMP threads. These MPI processes communicating with each using MPI outside of OpenMP parallel regions.

3. Using only OpenMP: this style of programming relies on virtual distributed shared memory systems (DSM). The programmer uses only shared memory directives to exploit parallelism. However, this can not be achieved with standard openMP. Some researchers have already used the NanosCompiler or the NthLib runtime library, which provides a full support for nested parallelism in OpenMP [36].

In fact, a hybrid programming model that extends MPI with OpenMP manages to program clusters consisting of a number of SMP nodes. However, there are remaining issues to address, such as thread granularity: the model requires large thread granularity to achieve better performance. It does not provide any advantage over a hybrid MPI-only approach. Moreover, this style of programming is not easy enough for a programmer to write efficient applications [47].

Global Arrays is a shared memory programming model which combines the advantages of a distributed memory model with the ease of use of shared memory. It exploits SMP locality by sharing data placed in shared address space rather than using message passing. This is achieved by functions that work on locally distributed data and functions that transfer data between a shared address space and local storage [86].

2.2.7 Summary

In order to build software that can take advantage of emerging parallel architectures, a parallel programming model needs to be provided which will coordinate

Model	Task Identification	Task Mapping	Data Distributed	Communication Mapping	Synchronisation
HPF	Implicit	Implicit	Implicit	Implicit	Implicit
FDIP	Implicit	Implicit	Implicit	Implicit	Implicit
OpenMP	Explicit	Implicit	Implicit	Implicit	Implicit
GpH	Explicit	Implicit	Implicit	Implicit	Implicit
Nesl	Explicit	Implicit	Implicit	Implicit	Implicit
Clean	Explicit	Explicit	Explicit	Implicit	Implicit
Eden	Explicit	Explicit	Explicit	Implicit	Implicit
X10	Explicit	Explicit	Explicit	Implicit	Implicit
MPI	Explicit	Explicit	Explicit	Explicit	Explicit
PVM	Explicit	Explicit	Explicit	Explicit	Explicit

Table 3: Comparison of some Popular Parallel Programming Languages.

parallelism and describe placement of data. In addition to these core parallel programming issues, a developer must often deal with scheduling tasks under a heterogeneous processor environment, and the management of nonuniform memory access. This chapter has reviewed a number of parallel programming models that aim to address these challenges. Table 3 ranks models according to five different aspects. The categories in the rows of the table start from the most implicit to most explicit models. The performance typically increases as the explicitness increases while at the same time, the productivity of parallel programming decreases, due to the complexity of the program. We draw the following conclusions.

Traditionally, implicit models are better approaches for parallel programming as a programmer does not have to control the parallel execution in detail. However, they are hard to implement sufficiently well to produce acceptable parallel performance across all parallel architectures.

The semi-explicit models provide some control of generated parallelism, which we believe is suitable for parallel programming. The level of explicit properties

varies in existing models. In the most abstract of these models, task identification is sufficient. The main advantage of this model is that it provides the necessary control for the model to be portable and yet easy to program.

The explicit models can deliver very high performance but at the price of programming productivity and performance portability. The loss of productivity means that developing the parallel programming is time consuming. The loss of performance portability means that the program cannot be moved from one architecture to another architecture without code changing, whilst achieving comparable parallel performance.

In Chapter 5 we propose semi-explicit mechanisms for controlling task placement, and hence data locality on hierarchical architectures.

2.3 Key Runtime Aspects in a Semi-Explicit Model

Given that the work in this thesis depends on the functional programming model, we discuss the runtime aspects of parallel functional programming. Many of these aspects are crucial to the implementation of implicit or semi-explicit parallel functional languages. Here we give a detailed introduction to the GUM runtime system implementing the GpH language [121]

2.3.1 Thread Creation and Synchronisation

Parallel models need to address the issue of when and how threads are created, how to prevent threads from evaluating expressions that are already under evaluation, and data transfer between threads. The decision as to when and how threads are created is achieved using a `sparking` mechanism to identify the potential parallelism. To prevent several threads from evaluating the same expression the `locking` mechanism is used. The `wait list` mechanism is used on the data structure for synchronisation.

Indicating Parallelism: The potential parallelism can be indicated in various ways: by using a parallel `let` annotation or by using fixed parallelism primitives for parallel programming that can be allocated to any expression. We focus on annotation based parallelism. This type of parallelism is deterministic, i.e. it does not change the result of the program. Some, annotated parallelism does not actually enforce parallel evaluation, but rather advises the runtime system

to generate parallelism. The runtime may make the decision, whether to exploit this potential parallelism or not, depending on the workload.

Sparking is the most popular mechanism to generate potential parallelism in graph reduction models [53]. A spark is a new pointer to a sub-graph which represents some computation. There are two approaches to parallelism using `sparks`. Mandatory parallelism ensures that all `sparks` are converted to threads. In contrast, advisory parallelism merely hints to the runtime system by using information provided by the programmer or by the compiler about potential parallelism. However, the runtime system is free to decide whether to convert the spark to a parallel thread. The weakness of mandatory parallelism is that converting every single spark to a thread results in a fine-grained parallelism which may possibly overload the system. In contrast, advisory parallelism yields a high flexibility in the number of threads created. Advisory sparking is similar to lazy task creation, where a task is generated only when it is demanded. It retains only the information required to generate the task later [124]. The laziness aims to reduce the overhead of exposing parallelism. These advantages come at the price of thread creation management. Advisory sparking has been implemented in many approaches such as GRIP [92], (ν, G)-machine [10], GUM [121], Feedback Directed Implicit Parallelism [46] and GpH-SMP [16].

Locking is the mechanism used to ensure the synchronisation between parallel threads. It is one of the major source of sequential overhead, and reducing locking is therefore potentially highly significant. Therefore, parallel model should minimise the locking as much as possible. There are systems minimise locking by allowing duplication of work. This is good only for short-running tasks. However for long-running tasks, it is necessary in a graph reduction mechanism to avoid the evaluation of sub-graphs by more than one thread at the same time. Locking prevents other threads evaluating a sub-graph currently under evaluation by another thread.

In a purely functional language locking can be omitted to reduce synchronisation overheads at the cost of duplication. However locking as implemented in most parallel graph reduction models such as GUM, (ν, G)-machine, etc. The implementation of locking in GUM is as follows: if a thread demands a result of a graph node currently under evaluation, the demanding thread will be blocked on this graph structure until the result becomes available [64]. The (ν, G)-machine implements the locking mechanism in a similar way: it locks the node under evaluation using an indicating flag. If the flag is set, all processes trying to access this node will be blocked until the node is updated [10].

The efficient implementation of locking is critical for performance, as updating nodes is an extremely common operation in graph reduction systems.

The Waiting List is a technique used to record threads while waiting for the result of a computation. It is usually attached to the locked node in a sub-graph: a thread starting to evaluate a sub-graph locks the root. When another thread tries to access the root closure, it finds the root closure locked and is added to the `waiting list` attached to the root. If the thread evaluating the graph updates the root closure all threads in the `waiting list` are re-awoken to continue with their evaluations, using the result of the completed computation [66]. However, there is an issue that should be taken into account when adopting the `waiting list`; the model must minimise the heap usage. GUM and (ν, G)-machine achieve this reduction by reusing parts of the node for the root of the `waiting list`. GUM uses the first two words of the closure. The (ν, G)-machine uses a back-link in the graph structure.

2.3.2 Storage Management

Large scale parallel processing involves the exploitation of hundreds or thousands of processing elements (PEs) connected by a high speed interconnect network. Normally every PE includes a physical processor, local memory and an interface device connecting the PE to the network. In this system, each processor has a favoured low latency, high bandwidth path to local memory bank, and a longer latency, lower bandwidth to remote memory. One method to access a global memory is by creating a virtual address heap interpreted to a physical address on the local memory of a processing element [90]. This method is discussed in more

detail here. Using a global address space is a technique to model a distributed memory in a graph reduction parallel system. The global space holds information about the remote processor and graph structures evaluated remotely. This storage mechanism is implemented in X10 [30] and GUM [121]. The locking mechanism described earlier is implemented: threads may block waiting upon arriving data from a remote processor. Therefore, it is necessary to hide the communication overhead by a computation or by avoiding communication between tasks (data locality).

2.3.3 Data Locality

Data locality refers to the use of data elements within relatively close storage locations, i.e. the arrangement of related tasks and their data close to each other as much as possible. This can improve parallel program execution time significantly by reducing the fetching time of remote data between threads. This can be achieved by executing tasks close to the data they need. A well-known fact to all parallel implementers is that the communication overhead can dramatically affect the performance of parallel computations [100]. This problem arises from small computations demanding remote data, causing a communication overhead much larger than the computation. Thread allocation and thread placement decisions are even harder on heterogeneous multicore architectures. Many parallel programming models use sophisticated adaptive algorithms that support dynamic,

lightweight threads. The heart of these models is a thread scheduler that balances the load among the processes. In particular, the processor can execute another thread while one thread is waiting for communication. However, not only a good load balance is essential for high performance, good data locality is an important factor too. One technique is to allow the programmer to specify the data access patterns in the program. This can be quite difficult for complex data access patterns. The most popular load balance algorithm is work stealing, which is based on random scheduling of threads. However, the randomisation in the work-stealing algorithm can work against data locality. The current models implementing work stealing still need to be improved to achieve good data locality.

X10 improves data locality by integrating new constructs (notably, places, regions and distributions) to model hierarchical parallelism and nonuniform data access [30]. X10 splits the memory space into parts known as partitioned global address space. Constructs enable programmers to assign a single `place` to each global address space. During execution of an `activity`, it will be located on the same partitioned global address of its `place`. With this design, X10 claims to minimise the communication between remote nodes.

Unified Parallel C (UPC) is a parallel extension of the C programming language developed for multiprocessors[37]. It is a distributed shared memory programming model. UPC facilities shared-memory programs, while exploiting

data locality. UPC provides constructs for placing data close to thread need it. These constructs provide a mechanism that a private object can only be accessed by its owner thread; any synchronisation with other threads can be made through a shared address space.

Scalable Locality-aware Adaptive Work-stealing Scheduler (SLAW) is a scheduler designed for parallel programming models [43]. The scheduler is currently implemented in the Habanero Java (HJ) language. The goal is to allow programmers or compilers to give locality hints to runtime. Like the X10 model, SLAW achieves locality by grouping workers into places. Each place has a mailbox to receive a remote task from a worker in another place. Each worker within a place has its own queue. Tasks in the mailbox have less priority than tasks in the worker queue. If a worker becomes idle, it looks for tasks in its local queue, then in other worker's queue in the same place, and finally from the mailbox.

2.4 Summary

A parallel model that can exploit heterogeneous multicore architecture must deal with a parallel heterogeneous programming environment, scheduling on heterogeneous processors, and with the management of nonuniform memory. Earlier work has focused on spawning threads dynamically using annotations, without taking the distance notion into account while spawning a thread. The distance notion means the communication overhead associated with a spawning thread on remote

processor element. It can be represented by the latency between processors or by the number of communication hubs traversed between PEs as we did in this thesis.

We have shown that existing mechanisms have limited support for improving data locality, which can have a high impact on parallel programming performance. In the next chapter, we will present the first programming and performance comparison of functional multicore technologies and report some of the multicore results for two languages. The results of the comparison is one of the motivations to introduce new annotations in GpH to improve data locality.

Chapter 3

Multicore Parallel Haskell Comparison

This chapter compares four different parallel Haskell implementations on a multicore architecture. It outlines the key features of each implementation. Finally, we present the findings of the comparison.

3.1 Introduction

There is a long-held assertion that functional languages are suited for parallel computation. The fundamental motivation of this claim is that, because of the lack of side effects, there will often be a chance to evaluate sub-expressions in parallel without risk of interference [52]. Moreover, parallelism can be achieved using high order functions, thus controlling the parallelism and at the same time

hiding the low-level coordination details. The low-level details are managed by the compiler or dynamically by the runtime system.

High level programming languages such as MPI [59], UPC, OpenMP [28] and Pthreads may be implemented on multicore architectures but this is usually not the best alternative for the migration of most mainstream applications [27]. MPI, UPC, and Pthreads require a high amount of code to be reorganised in order to achieve reasonable performance while OpenMP programs require less code reorganisation. However, OpenMP does not yet provide features for the expression of locality and modularity that may be needed for multicore applications.

While there are likely to be several successful approaches to multicore programming, we believe that functional language is relatively convenient for multicore. Parallel Haskell languages have been successfully deployed on shared memory systems (SMPs) and distributed memory architectures (DSMs).

This chapter presents a programming and performance comparison of functional multicore technologies and reports some of the first multicore results for the approaches. The comparison contrasts the programming effort required to specify coordination with the parallel performance delivered in each language. The comparison uses 15 programs carefully selected, i.e. without regard for their inherent parallelism, from representative parts of the Nofib benchmark suite [89]. In consequence, the results reflect the multicore performance that might be expected for a "typical" set of Haskell programs (Section 3.4).

This comparison was conducted between a "no pain" parallel implementation

Feedback Directed Implicit Parallelism (FDIP) [46], and three "low pain", i.e. semi-explicit languages (Section 2.2.4). The semi-explicit Haskells are Eden [74] and two implementations of Glasgow parallel Haskell (GpH) [121], GpH-SMP and GpH-GUM (Section 3.3).

Although the parallel Haskell implementations all share the same optimising Glasgow Haskell Compiler technology (GHC), each uses a different version, and hence the performance comparisons are based on speedups, which are normalised against different sequential performance. We establish a baseline for the speedup comparisons by reporting sequential and parallel runtimes and efficiencies for three of the languages (Section 3.5).

We report detailed parallel performance and programming effort studies, focusing on the number of programs improved, speedups delivered, and program changes required to coordinate parallel evaluation (Section 3.6). The study compares the scalability, the programming effort required, and the parallel performance achieved, in each language (Section 3.7). We conclude by summarising the key results and discussing their implications (Section 3.8).

For the GpH-GUM measurements presented in this chapter, we use an earlier GHC-4.06 version which was upgraded later as a part of work conducted for this thesis and is used to evaluate the architecture-aware constructs in chapter 6. Moreover, it uses the original strategies style [122] to parallelise the benchmark using GpH-GUM. The strategies have been improved in [77]. The next chapter discusses the new strategies in more detail.

3.1.1 BenchMark Suite

We compare the performance of the four parallel Haskell implementations using the 15 programs from the "real" and "spectral" sections of the Nofib benchmark suite [89]. The "real" and "spectral" sections of the Nofib suite are carefully designed to be representative of small Haskell programs, i.e. around 300 source lines of code. The programs are :

- `atom` is a floating point simulation.
- `Boyer` is a Gabriel suite Boyer benchmark. It rewrites a given input term according to a given set of lemmas in an attempt to produce the value True meaning that the original term was a valid theorem.
- `circsim` is a circuit simulator.
- `clausify` reduces propositions to clausal form.
- `compress` is a text compression algorithm.
- `fft2` performs Fourier transforms using floating point arithmetic.
- `hidden` is a line rendering tool.
- `lcss` is a hirschbergs LCSS algorithm.
- `multiplier` is a binary-multiplier simulator.
- `para` formats paragraphs in text.

- `primetest` is a primality testing program.
- `rewrite` is a equation rewriting system.
- `scs` is a circuit simulator. It largely relies on arrays, and arithmetic.
- `simple` is a hydrodynamics and heat-flow program.
- `sphere` is a ray tracing program.

The programs are a substantial subset of the 20 multicore benchmarks used in [46]. Of the five programs not measured, the `bsort-1` and `bsort-2` programs are not Nofib benchmarks, and three (`cacheprof`, `calendar` and `fibheaps`) are too small to benefit from parallel execution, i.e. where the input cannot be sized to give a runtime of 3s on current hardware. Crucially, other than to exclude short programs, the programs are not selected *a priori* for having obvious parallel structure. Hence, our results reflect the multicore performance that might be expected for a set of "typical" small Haskell programs.

This chapter is closely based on "Low Pain vs No Pain Multi-core Haskells" [8].

3.2 Parallel Haskell Language Comparison

This section outlines the parallel languages compared in the study. The FDIP implicit approach supports an unchanged Concurrent Haskell [93]. Indeed both Eden and GpH-SMP support multiple stateful (IO) threads (concurrency) and stateless parallel threads. More precisely, GpH-SMP is a superset of Concurrent

Haskell. However, concurrency is not used in our study and so our language comparisons focuses on parallelism only.

```
test :: Int -> Bool
test n = all test0 (take n (repeat (Var X)))
```

Figure 5: Sequential Top-level Boyer function

We illustrate the coordination extensions by using them to parallelise the Boyer Nofib program [89].

Figure 5 shows the key top-level function where the obvious extension can be made to the original program. The function takes n size as input parameter.

3.2.1 Indicating Parallelism in GpH

Many parallel functional languages including GpH, generate tasks of a finer grain than an implementation can exploit efficiently [83]. The `par` and `pseq` operations are used to write high-level functions that would assist in parallel programming. Experience of implementing nontrivial programs in GpH shows that the unstructured use of `par` and `pseq` operators can lead to rather obscure programs [65]. This problem can be overcome by using *evaluation strategies* [122]: lazy, polymorphic, higher-order functions controlling the evaluation degree and the parallelism of an expression. They provide a clean separation between coordination and computation. The driving philosophy behind evaluation strategies is that it should be possible to understand the computation specified by a function without considering its coordination.

```
type Strategy a = a -> ()  -- type of eval. strategies

using :: a -> Strategy a -> a -- strategy application

rwhnf :: Strategy a     -- reduce to weak hd norm form

class NFData a where    -- class of reducible types
  rnf :: Strategy a     -- reduce to normal form

parList :: Strategy a -> Strategy [a]
parList strat []   = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

Figure 6: Evaluation Strategies

Figure 6 shows the basic operations and composite evaluation strategies. The `using` construct applies a strategy to an expression. The basic strategy `rwhnf` reduces an expression to weak head normal form (WHNF). The overloaded basic strategy `rnf` reduces an expression to normal form (NF), and is instantiated for all major types. As functions, strategies can be combined using the power of the language, e.g. composed or passed as arguments. For example, `parList` applies strategy `strat` to every element of a list in parallel. Evaluation strategies will be revisited in Section 4.1.3, focusing on their functionality in more detail.

To demonstrate how to write parallel programs using GpH, Figure 7 shows the GpH parallelisation of the top-level Boyer `test` function, and works as follows. The input list is bound to a variable `xs`, and then split into `m` chunks and bound to `xs1`. Next the condition (`all test0`) is mapped over the chunks to give a list of intermediate results `res`. It is this mapping that is parallelised (`` `using` parList rnf``). The final stage is to combine the intermediate results `all (&&`

```
test :: Int -> Bool
test n m = all (&& True) res
   where xs  = take n (repeat (Var X))
         xs1 = splitAtN m xs
         res = map (all test0) xs1 `using` parList rnf

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
                where (ys,zs) = splitAt n xs
```

Figure 7: GpH Top-level Boyer function

`True) res`.

The parallelisation illustrates some interesting points. In this program, just one function of the 52 functions in the 300 line program changes. This is the case for many, but not all, programs. Exceptions include `Sphere` and `Hidden` where parallelism is introduced in more than one function. The parallel paradigm is chunked data parallelism. That is, the parallelism is determined by the underlying data structure, and to obtain suitable thread granularity, the program has been changed to aggregate the input. In other programs it is possible to introduce parallelism without changing the algorithmic or computational part of the program, e.g. [69].

3.2.2 Indicating Parallelism in Eden

Eden [74] extends Haskell with syntactic constructs to explicitly define and instantiate processes, as described earlier (see section 2.2.4). Eden uses the skeleton framework to overcome the difficulties of programming, and provide more control

over fine-grain tasks. Algorithmic skeletons abstract common patterns of parallel evaluation into higher-order functions [32]. They simplify the development of parallel programs by hiding coordination details from the programmer, and may provide ready-made parallel cost models. Eden supports a range of skeletons [13], and some of these skeletons have been used to parallelise the Nofib programs.

Farm skeleton is a scheme that generates a finite number `np` of processes and allocates every process more than one task. It uses the parameter functions `distribute` and `combine` for distributing tasks among the processes and combining the results. Figure 8 shows the implementation of the Eden farm skeleton.

```
farm :: (Transmissible a,Transmissible b) =>
        Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) ->
        Process [a] [b] -> [a] -> [b]
farm np distribute combine proc tasks
        =combine (parMap proc (distribute np tasks))
```

Figure 8: Eden Farm Skeleton

The skeleton takes three parameters: the number of processes, the distributed function and the combine function. The parMap is used to create the processes. The farm allocation is done under the assumption that tasks are regular, and the number of tasks is larger than the number of processors will keep every processor busy during the program runtime.

Master-Worker skeleton is a scheme that dynamically assigns tasks to free worker processes. The Master-Worker philosophy is that each processor accommodates one worker process and that the worker receives work from the workpool. If the worker finishes the task, the result is sent back. The received result is translated, as a new work request.

```
mw :: (Trans t, Trans r) => Int -> Int -> (t -> r) -> [t] -> [r]
mw n prefetch wf tasks = ress
where (reqs, ress) = (unzip . merge) (spawn workers inputs)
-- workers :: [Process [t] [(Int,r)]]
   workers  = [process (zip [i,i..] . map wf) | i <- [0..n-1]]
   inputs   = distribute n tasks (initReqs ++ reqs)
   initReqs = concat (replicate prefetch [0..n-1])
-- task distribution according to worker requests
   distribute :: Int -> [t] -> [Int] -> [[t]]
   distribute np tasks reqs = [taskList reqs tasks n | n<-[0..np-1]]
   where taskList (r:rs) (t:ts) pe | pe == r = t:(taskList rs ts pe)
                                   | otherwise = taskList rs ts pe
         taskList _ _ _ = []
```

Figure 9: Eden Master-Worker Skeleton (Static Task Pool)

Figure 9 shows the implementation of a relatively simple master worker skeleton. The details of the implementation are not salient here, but the essence is as follows. Tasks are distributed to `n` worker processes, the worker function `wf` is applied to each task and returns a pair consisting of the worker number and the result of the task evaluation to the master process, i.e. the process evaluating `mw`. The worker numbers are interpreted as requests for new tasks. The master uses a function `distribute` to send tasks to the workers according to the (`n*prefetch`) requests initially created and the ones received from the workers. The master

worker skeleton has been used to parallelise several programs of the Nofib suite, including Boyer.

```
test n m f = all (&& True) res
  where xs = take n (repeat (Var X))
        xs1 =  splitAtN  m  xs
        res = parallelMap (all test0) xs1
          where
            np = noPe
            parallelMap = mw np pf
            pf = min 100 maxpf
            maxf= max 2 (n `div` (m*np*f))

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
            where (ys,zs) = splitAt n xs
```

Figure 10: Eden Top-level Boyer function

Figure 10 shows the Eden skeleton-based parallelisation of the top-level Boyer `test` function, and works as follows. The input list is chunked and bounded to the variable `xs1`, and the intermediate results combined by `all (&& True) res` in the final stage. For Eden, the mapping of `all test0` over the chunks is parallelised using a master worker skeleton parametrised by the number of cores and the number of tasks to prefetech: `mw np pf`. The f parameter specifies the number of tasks to be prefetched:$\frac{1}{f}$ of the average tasks per worker , but not more than 100 tasks. The average tasks per worker is list length (n) divided by chunk size (m) and no of workers (np),that is $\left\lfloor \frac{n}{m \cdot np \cdot f} \right\rfloor$. As in GpH the paradigm is chunked data parallelism, and just one out of 52 functions has been parallelised, although this time an algorithmic skeleton is used.

Description	FDIP	GpH	Eden
Classification	Implicit	Semi-explicit	Semi-explicit
Evaluation Order	Normal Order	Normal/ Mixed	Mixed
Methodology	FDIP Tools	Evaluation Strategies	Direct or Skeletons
Process Model & Creation	Speculative Threads	Optional Threads	Explicit Processes, Mandatory Creation
Thread Placement	Implicit	Implicit & Dynamic	Implicit & Static
Communication Channels	Implicit	Implicit	Implicit & Explicit

Table 4: Language-level Comparison of Parallel Haskells

3.2.3 Language Coordination Comparison

FDIP performs sophisticated static analysis and program synthesis in order to generate a sufficient amount of parallelism. Both GpH and Eden rely mainly on a sophisticated runtime-system with dynamic resource management.

Table 4 summarises the language level differences in coordination specification in the three parallel Haskells. Much of the table summarises aspects outlined above. However, a key distinction between the languages is that while FDIP preserves normal-order evaluation of pure expressions, GpH may not, and Eden does not. GpH preserves normal-order evaluation if every evaluation strategy added is no more strict than the embedding function. However, it is often useful to be more strict, e.g. speculatively evaluating expressions in the anticipation that they will be used. While Eden processes preserve some normal-order evaluation,

e.g. of expressions within the body of a process, they are more strict than the corresponding function, i.e. they eagerly evaluate their arguments.

As an entirely implicit language, FDIP has the highest level of coordination abstraction, GpH has an intermediate level and Eden has the lowest. That is, Eden is most explicit about coordination behaviour, but, as we shall see in Section 3.7, the use of appropriate skeletons can raise the level of abstraction.

3.3 Parallel Haskell Implementation Comparison

All of the parallel Haskells support high-level coordination, and rely on sophisticated implementations to effectively manage a vast array of low-level coordination issues, typically including task placement, communication, synchronisation and storage management. All four implementations perform parallel graph reduction [94]. No simple models have ever been constructed of such systems, and their performance is often extremely hard to analyse. Indeed, this is why profiling tools are an essential aid to understand parallel behaviour when tuning the parallel performance of programs written in this class of language.

3.3.1 Feedback Directed Implicit Parallelism (FDIP)

FDIP exploits information from a profiled execution of the program, and the number of processors available on the underlying platform, to take better parallelisation decisions. This is intended to optimise the program for best performance when performing the same work with the number of processors available. FDIP supports the full Concurrent Haskell language, compiled with traditional optimisations and including I/O operations, and synchronisation, as well as pure computation. The available parallelism in the program is realised by tracing and recording the dependencies between different pieces on `thunk` execution. Parallelism is introduced and controlled in FDIP in a four stage process [46] as follows.

Firstly an example execution of the program is profiled. Secondly the profile trace is analysed as a dependency graph of computations to identify useful sources of parallelism. Given the large number of potential computation `thunks` in almost any Haskell program, the challenge is to identify `thunks` that are simultaneously independent of other `thunks`, demanded by the program, and with large thread granularity. The third stage is to recompile the program to automatically introduce parallelism at the identified program sites. Finally, sophisticated mechanisms are introduced into the runtime system to manage the threads introduced at these sites. These include treating the parallel threads as speculative, and managing load with work stealing.

A simulated limit study shows the potential of FDIP to produce substantial

amounts of parallelism for many programs, e.g. utilising at least 8 cores for 40% and at least 2 cores for 80% of the 20 Nofib programs studied. However, the multicore performance is disappointing, with only 5 programs out of 20 delivering a speedup of more than 10% [46].

3.3.2 GpH-SMP

Since 2004, the Glasgow Haskell Compiler (GHC) has supported a shared-memory implementation of GpH. The shared memory implementation is evolving rapidly, and the precise version we describe here and measure in later sections is based on GHC 6.10.1. The GHC runtime system implements Concurrent Haskell threads using a system of lightweight threads multiplexed onto a small number of heavy-weight OS threads in order to achieve real parallelism on a multiprocessor, while still keeping the overheads of concurrency low. The parallel runtime system is built around the notion of `capability`. A capability represents the resources for running a Haskell computation. The number of capabilities equates to the number of Haskell threads that can be running simultaneously at any one time. GHC's capabilities correspond precisely to the Eden and GUM Processing Elements (PEs) described below. It is the responsibility of the scheduler to allocate Haskell threads to capabilities, and Haskell threads may migrate between capabilities at runtime depending on the scheduling policy and runtime parameters. Although the GHC 6.10.1 implementation measured here distributes work eagerly, later, unreleased versions gain improved performance by adopting a lazy

work stealing approach [16].

The capability holds all of the private state that a worker needs to execute Haskell code. The capability has its own allocation area so that allocation proceeds without expensive per-object synchronisation[45]. GHC 6.10 supports both parallel and sequential garbage collection, and the measurements in the following sections use the former. In this scheme, when memory is exhausted, all cores cease reduction and perform garbage collection in parallel.

GpH-SMP is under development and published performance results are sparse. Performance results comparing adapted versions of GHC 6.10.1 on an 8-core machine are reported in [16], together with the identification of a number of areas for improvement. The results in Sections 3.5, 3.6.2, and 3.7 are the first performance results for a released version of GpH-SMP.

3.3.3 GUM Implementation of GpH

Graph-reduction on a Unified Machine-model (GUM) is a portable, parallel runtime environment for GpH [121]. As the name suggests, GUM is designed for both shared and distributed memory architectures. It implements a Distributed Shared Memory (DSM) [87] model of parallel graph reduction on a distributed memory architectures as a virtually shared graph. Graph segments are communicated via messages, using standard communication libraries like PVM [88] or MPI [125] to provide an architecture-neutral and portable runtime environment. If the GUM implementation uses the PVM communication library environment.

It starts by creating a PVM manager task, which controls start up and termination. It spawns the required number of virtual PEs as PVM tasks. These PVM tasks are mapped to available processors. Once all PEs finish their initialisation, the main task starts executing the Haskell program on a main thread. The program terminates when either the main thread has finished or a FINISH message is received from the manager task, in case of an error.

Throughout the runtime of the program, each PE executes the main schedule loop until it receives a FINISH message. The main scheduler performs the following:

1. Perform garbage collection if required.
2. Process any incoming message.
3. Run thread from runable threads if there are any.
4. Look for work locally or remotely.

Thread Management: The unit of computation in GUM is a lightweight thread, and each logical PE is an operating system process that co-schedules multiple lightweight threads. Threads are automatically synchronised using the graph structure, and each PE maintains a pool of runnable threads. At each scheduling step, the runtime scheduler selects one thread from the thread pool for execution. This thread then runs until it finishes, blocks, or the system terminates as the result of an error. Parallelism is initiated by the `par` combinator:

when an expression `x 'par' e` is evaluated, the heap object referred to by the variable `x` is `sparked`, and then `e` is evaluated. By design, sparking a reducible expression (`thunk`) is a relatively cheap operation, and sparks may freely be discarded if they become too numerous. If a PE is idle, a spark may be converted to a thread and executed. Threads are more heavyweight than sparks as they must record the current execution state.

Load distribution: GUM uses dynamic, decentralised, and blind load management. The load distribution mechanism is designed for homogeneous architectures with uniform PE speed and communication latency, and works as follows. If (and only if) a PE has no runnable threads, it creates a thread to execute from a spark in its sparkpool, if there is one. If there are no local sparks, then the PE sends a FISH message to a PE chosen at random, seeking to steal work. Initially, only the main PE has work. The other PEs are all idle and they start fishing at the beginning of execution. A FISH has limited age before it is returned to the originator and discarded from the system.

If the PE that receives a FISH has a useful spark, it sends a SCHEDULE message to the PE that originated the FISH containing the sparked thunk packaged with a nearby graph. The originating PE unpacks the graph, and adds the newly-acquired thunk to its local sparkpool. Figure 11 shows the sequence of FISH and SCHEDULE messages. To maintain the shared virtual graph, a message is then sent to record a reference to the new location of the thunk.

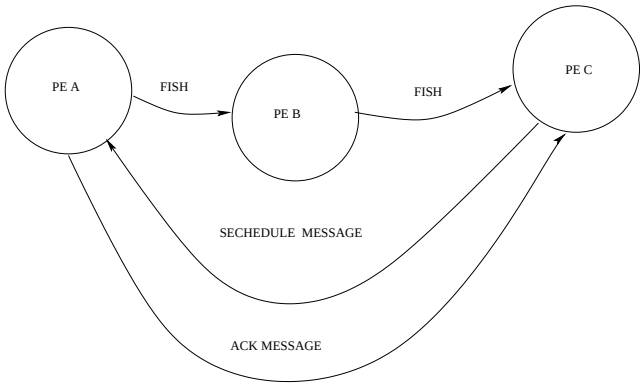


Figure 11: GUM FISH - SCHEDULE - ACK Sequence

Memory Management: GUM is a distributed shared memory model. It manages a virtual shared memory in the graph it inhabits. Each globally visible closure in the heap has a unique identifier via Global Address (GA). The FetchMe closure uses a GA to accesses the remote object. A GA consists of a PE identifier and a local identifier as a pair. In fact, it has three different tables: the GIT (Global Identifier Table) maps each allocated local identifier to its local address, GALA (Global Address to Local Address) maps remote GA to their LA, LAGA (Local Address to Global Address) maps local address to its corresponding global address [64].

GUM delivers good performance for a range of benchmark and real applications on a variety of parallel architectures, including conventional shared and distributed-memory architectures [69]. The results reported in Sections 3.5, 3.6.3, and 3.7 extend earlier shared memory results with *the first multicore* performance results for GUM. GUM's performance is also comparable with other mature parallel functional languages and with conventional parallel paradigms [67].

3.3.4 Eden Implementation

The Eden implementation extends GHC making a few changes to the front-end, but major modifications to the runtime environment [15]. When run in parallel, each PE runs a sequential copy of the GHC runtime system. Multiple PEs communicate by message-passing, and the communication layer has been designed to allow plug-in replacement of different message-passing libraries. Typically, it uses either PVM or MPI libraries.

Eden implements an explicit remote task creation mechanism and channel-based communication mechanisms between PEs. Both are exposed to the Haskell level via primitive operations. Eden language-level constructs are implemented as a Haskell module on top of these more basic primitives [15]. To synchronise communication between the PEs, placeholders in the heap are used, which will be replaced by arriving message data, i.e. computation subgraph structures, serialised into one or more packets for transmission.

An important difference between Eden and both GpH-SMP and FDIP is that the Eden process construct maintains *completely independent sub-heaps*. Unlike GUM which maintains a virtual shared graph by maintaining references between distributed heaps, an Eden process is a link to other sub-heaps via communication channels that have the key property that all communicated data is always fully evaluated.

Eden's distributed memory parallel performance is widely reported and shows

excellent runtimes, speedup and scale-up, e.g. [58, 91, 72]. Eden's distributed memory performance is also comparable with other mature parallel functional languages. The explicit process model and strict communication give performance advantages for applications with coarse thread granularity [67]. Impressive multicore performance results are now emerging, e.g. using Eden to coordinate large symbolic computations [3] and a comparison with optimised GHC-6.8 [16].

3.3.5 Implementation Comparison

Description	**FDIP**	**GpH-SMP**	**GpH-GUM**	**Eden**
GHC Version	GHC 6.6	GHC 6.10	GHC 4.06	GHC 6.8
Evaluation Model	Par. Graph Reduction	Par. Graph Reduction	Par. Graph Reduction	Par. Graph Reduction
Granularity Control	Dynamic	Dynamic	Dynamic	Static
Synchronisation. Unit	Thunk Locking	Thunk Locking	Thunk Locking	Channel Locking
Work Distribution	Work Stealing	Work Pushing	Work Stealing	Dynamic Process Placement
Work Duplication	Not Poss.	Possible	Not Poss.	Possible
Heap	Shared Heap	Shared Heap	Virtual Shared Heap	Distributed. Heap
Garbage	Depend. Sequential	Depend. & Parallel	Indep. & Parallel	Indep. & Parallel

Table 5: Implementation-level Comparison of Parallel Haskells

Table 5 summarises the implementation level differences between the four parallel Haskells. All four implementations perform parallel graph reduction. While an arbitrary number of Eden processes can be dynamically created, each process is mandatory. In contrast, the other implementations support dynamic

techniques, including thread subsumption, sparking, and the creation of optional or speculative threads. Eden also uses eager work distribution: newly created processes are pushed out to available PEs: while the other implementations are lazy and idle, PEs steal work, i.e. thunks. FDIP and GpH-GUM are both careful not to duplicate work by evaluating the same `thunk` more than once, but work may be duplicated in GpH-SMP or Eden.

A key distinction between the implementations is the heap model: while FDIP and GpH-SMP have shared heaps, GUM maintains a virtual shared heap, and Eden supports distributed independent heaps, both supported by message passing. Message passing is essential for distributed systems but initially seems enormously expensive compared with shared memory access. That is, a graph must be serialised into, and deserialised from, messages and computationally expensive message-passing libraries invoked.

However, the independent heaps maintained by GUM and Eden convey four significant advantages for shared-memory systems like multicores. Firstly, while the cores in shared heap implementations like FDIP and GpH-SMP must synchronise to garbage collect, GUM and Eden cores can collect independently and hence in parallel. Secondly, synchronisation is confined to limited shared memory areas, essentially the communication buffers. Thirdly, synchronisation granularity is often large, i.e. on large messages, rather than on individual thunks or memory locations. Finally, cache coherency issues are reduced, as tasks do not share caches [3]. We discuss the performance implications of the heap designs

further in Section 3.8.2.

Although both FDIP and GpH-SMP use dependent stop-the-world GC, such a design is not inherent. An implementation that maintains some form of thread-private heap, e.g. [34], would enable independent garbage collection and offer many of the advantages outlined above, without incurring the high communication costs of message passing. Indeed we argue that some form of independent heaps will be essential as multicores evolve towards many cores.

3.4 Experiment Design

3.4.1 Measurement Methodology

To parallelise the programs in Eden and GpH the programs were first time and space profiled to identify computationally expensive functions and these were parallelised. A variety of parallelisation options was investigated for each program and the best was selected. The same GpH program is evaluated under GpH-SMP and GpH-GUM, and the Eden program introduces an appropriate skeleton. Example GpH and Eden parallelisation of the Boyer benchmark are discussed in 3.2.

All programs are measured on the same input and with the same heap size. We follow the common practice of increasing input size in many cases, to match improvements in processor technology since the benchmarks were established in 1992. The best parallel performance is reported for each system.

The parallel implementations are all based on the GHC compiler, but use different versions of it. The FDIP approach uses GHC 6.6, GpH-SMP uses GHC 6.10.1, GpH-GUM uses GHC 4.06 and Eden uses GHC 6.8. As research platform, GHC evolves and typically the sequential execution time of programs is improved by later versions of the compiler. To address the issue of varying sequential performance, the primary comparative measure is *absolute speedup*, i.e. relative to the corresponding optimised sequential GHC compiler, e.g. GpH-SMP speedups are relative to GHC 6.10.1. This measure substantially normalises against sequential performance and is grounded by runtime measurements in Section 3.5.

The programs are all measured on common multicore architectures, namely eight core machines comprising two quad-cores. The GpH-SMP, GpH-GUM and Eden measurements are for Intel Xeon 5410 cores running at 2.33GHz, with a 1998 MHz front-side bus, 6144 KB and 8GB RAM running under Linux Fedora 7. The FDIP measurements are for Intel Xeon X5350 running at 2.66GHz with 4GB RAM running under Windows Server 2003 R2 x64 service pack 2.

All measurements reported throughout the thesis are given in terms of real (elapsed) time used by the program, and represent the median of several measurements (normally at least three times). As far as possible timing runs were made on machine with minimal other load. The benchmark programs have different runtimes and different speedups therefore the geometric mean is used to find a single runtime and speedup figure for them, as in other studies [105, 79, 80].

3.5 Runtime Comparison

As the parallel implementations use different versions of the GHC compiler (Section 3.4), this section provides a baseline for the speedup measurements in the following sections by comparing the runtimes and efficiencies of the GpH-SMP, GpH-GUM and Eden parallel implementations on 1, 2, 3, 4, 6, and 8 cores. FDIP is excluded, as an implementation is not available. On a single core, the runtimes of the parallel implementations are compared with the runtime of fully optimised sequential execution (GHC 6.10).

Program	**Sequential**			**1 Core**		
	GHC 6.10	**GHC 4.06**	**GHC 6.8**	**SMP**	**GUM**	**Eden**
Boyer	34.3	49.3	36.7	34.1	77.5	37.1
Clausify	31.1	51.2	29.1	30.4	78.7	29.3
Fft2	35.8	75.7	48.6	35.9	80.9	49.2
Rewrite	34.3	68.1	46.8	35.1	94.9	52.1
Geometric Mean	**33.8**	**60.1**	**39.5**	**33.8**	**82.7**	**40.9**

Table 6: Sequential Runtime Comparison (seconds).

Table 6 summarises the single core runtimes of 4 Nofib programs that deliver good speedup. To facilitate comparison, the inputs for the programs are sized to give sequential GHC 6.10 runtimes of approximately 35s. Columns 2–4 of the table report runtimes for the sequential compiler instances extended by the parallel Haskell implementations, and these form the basis for the absolute speedup calculations in the remainder of the chapter. The remaining columns of the table report the 1 core parallel runtimes for each implementation. We make

the following observations.

- The sequential runtimes vary by as much as a factor of 2.1: Fft2 under GHC 6.10 takes 35.8s, and under GHC 4.06 takes 75.7s, but typical variation is less.

- The geometric mean runtimes show that GHC 4.06 is the slowest on a single core, and 1.8 (60.1/33.8) times slower than GHC 6.10. This reflects recent GHC performance improvements. GHC 6.8 is 1.2 (39.5/33.8) slower than GHC 6.10. Longer runtimes for GHC 4.06 and GHC 6.8 give GpH-GUM, and to a lesser extent Eden, an advantage in the following speedup measurements, as the compute time is relatively large compared with communication time.

- It is generally anticipated that a parallel language implementation will introduce some sequential overhead compared with optimised sequential execution. The overhead is termed *sequential efficiency* and represents the additional costs of parallel execution, e.g. launching a single virtual PE and synchronising on closures. The overhead is a function of both the parallel program and the architecture, and is typically around 80% [121].

  Comparing Columns 3 and 6, and columns 4 and 7, of Table 6 shows that this expectation is met for GpH-GUM and Eden, with geometric mean sequential efficiencies of 72% (60.1/82.7) and 97% (39.5/40.9) respectively.

Surprisingly, however, GpH-SMP apparently has no overheads, i.e. an efficiency of 100%.

	SMP	GUM	Eden
Boyer	10.0	14.1	10.1
Clausify	4.7	11.5	5.1
Fft2	13.1	45.8	17.7
Rewrite	6.5	26.9	9.9
Geometric Mean	**8.0**	**21.1**	**9.7**

Table 7: 8 Core Parallel Runtime Comparison (seconds).

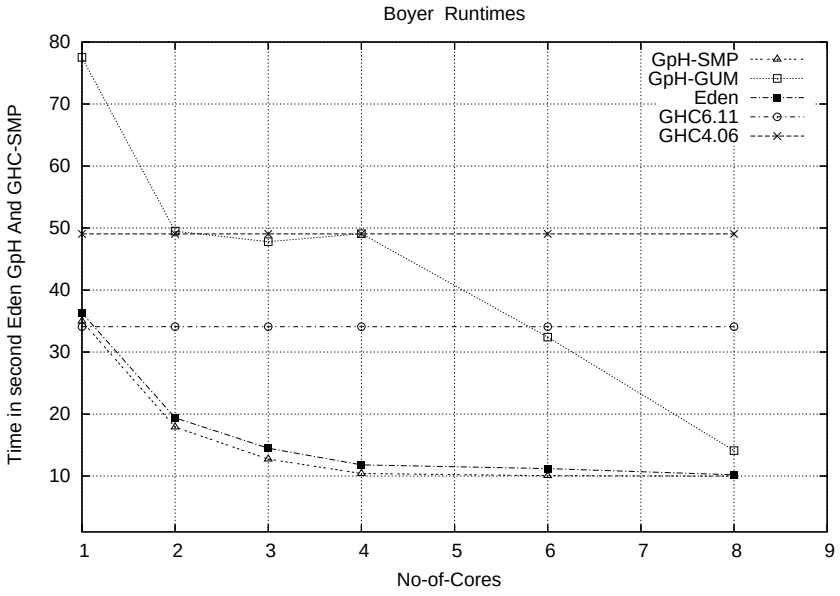
Table 7 summarises the runtimes of the same 4 programs on 8 cores. We make the following observations.

- On 8 cores the variation in runtimes is at most a factor of 4.1 (26.9/6.5), between GpH-SMP and GpH-GUM Rewrite, but is typically rather less.

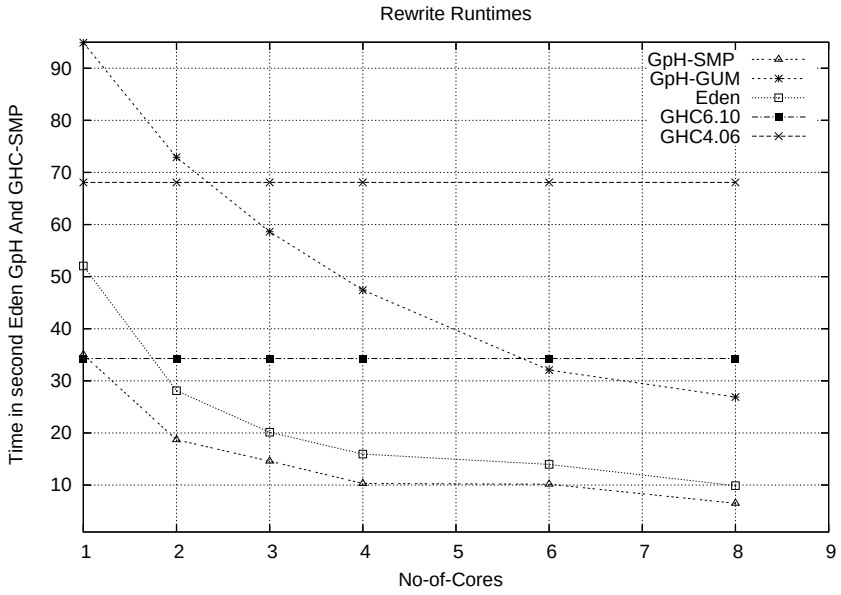
- The geometric mean 8-core runtimes show that, for this collection of programs, GpH-SMP remains fastest, Eden is just 21% (9.7/8.0) slower, and GpH-GUM slowest by a factor of 64% (21.1/8.0).

Figures 12 and 13 compare the runtimes and absolute speedups of two of the programs from Table 6, namely Boyer and Rewrite. The programs are measured on 1, 2, 3, 4, 6, and 8 cores. We make the following observations.

- For both programs the runtime curves are broadly similar for all implementations. For GpH-SMP and Eden the curves are very similar, and while the Eden is a little ($< 36\%$) slower on 1 core, the 8 core results are very

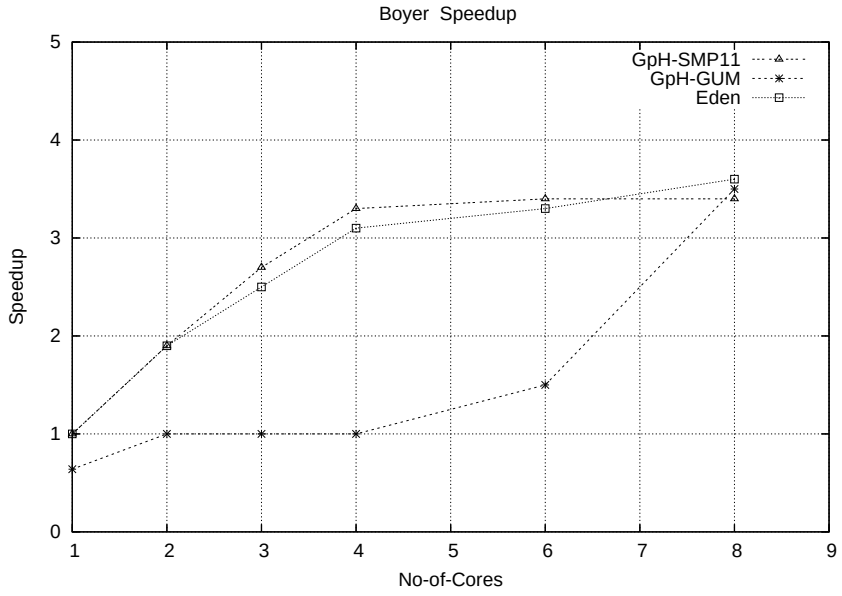


(a) Boyer Runtime

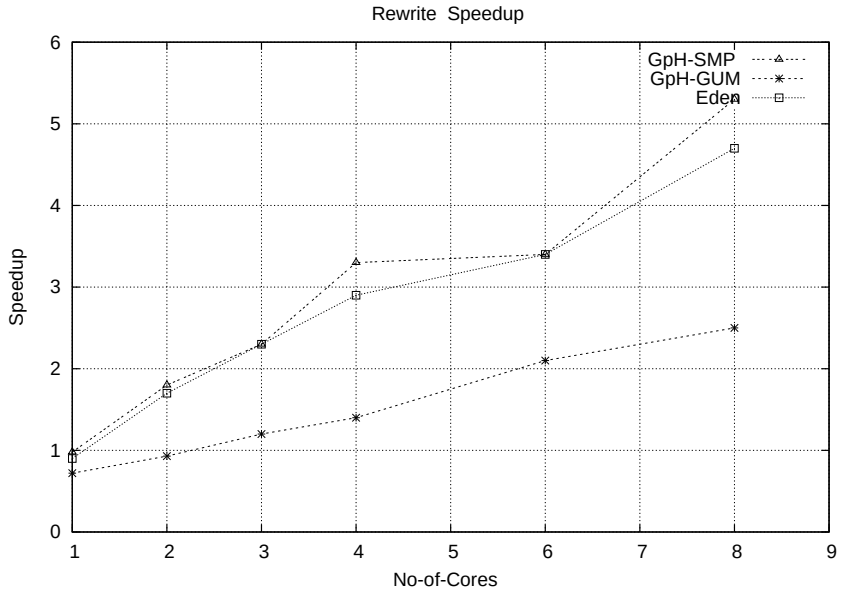


(b) Rewrite Runtimes

Figure 12: Runtime Comparison of Parallel Haskells (Boyer/Rewrite)



(a) Boyer Speedup



(b) Rewrite Speedup

Figure 13: Absolute Speedup Comparison of Parallel Haskells (Boyer/Rewrite)

similar. This is reflected in the speedup graphs, where Eden has better 8-core speedups.

- Both programs are scale for all three implementations, i.e. the runtimes fall as cores are added. The only exceptions are for Boyer between 2 and 4 cores under GpH-GUM and between 4 and 6 cores under GpH-SMP, and Eden. This is in marked contrast to FDIP, where the best performance may be achieved under 2,3 or 4 cores [46], and we shall return to this point in Section 3.7.2.

- The absolute speedups of the Boyer program on 8 cores is similar for all implementations, this is expected for both Eden and GpH-SMP implementations but for the GpH-GUM implementation we attribute this to lucky scheduling of potential parallelism.

- Reflecting the runtime curves, the speedup curves for both programs are broadly similar, and for GpH-SMP and Eden, very similar.

- The absolute speedup on a single core reflects the sequential efficiencies of the implementations.

3.6 Programming Effort and Performance Results

This section investigates the parallel performance of the four parallel Haskells in conjunction with the programming effort required to achieve that performance. Parallel performance is measured as *absolute* speedup over optimised sequential execution (GHC), i.e. not relative to the single core parallel execution. The programming effort is measured using logical source lines of code (SLOC), both as an absolute number and as a percentage of program length. The common definition of logical SLOC is a count of lines in the text of the program's source code excluding comment lines. SLOC is normally exploited to predict the amount of effort that will be required to develop an application, as well as to estimate programming productivity or maintainability [39]. The SLOC is used in our comparison for these reasons: all implementations use the same sequential code of benchmark. SLOC has the advantages of simplicity and relatively wide use. We also record the parallel paradigm applied in GpH and Eden.

The following subsections report the programming effort and performance of each language, and the absolute speedups achieved for all 15 programs in the four languages are depicted in Figure 15 and summarised in Table 11. Section 3.7 then makes a comparison of the approaches.

Program Name	**Speedup**	**Lines Of Code**
Hidden	1.82	316
Atom	1.27	57
Simple	1.27	1053
Geometric Mean	**1.4**	

Table 8: FDIP Programs Improved.

3.6.1 FDIP Multicore Performance

FDIP is entirely implicit, and so no programmer effort is expended other than in profiling and using a special compiler. Similarly the programmer does not need to identify and apply some parallel paradigm. The FDIP implementation is not publicly available, therefore, the FDIP performance results reported in this thesis are based on the ICFP'07 paper [46], augmented with some additional results from the authors. Where the other parallel Haskell implementations are measured on 8 cores, FDIP performs better on 4 cores than on 8 and hence Table 8 follows [46] in reporting the programs that are improved on 4 cores. It shows that FDIP speeds up 3 of the 15 programs, with a geometric mean speedup of 1.4, and maximum speedup of 1.8.

Automatically extracting good parallel performance is acknowledged to be a challenging problem. However, some of the reasons for the relatively poor performance of FDIP are because the implementation is immature compared with the other systems and has some known technical problems [46]. Specifically, the simulation phase of FDIP profiling ignores several crucial aspects of parallel coordination, namely contention within the GHC run-time system; the locking

overheads; and finally the overheads of sparking work and the cache effects of moving data from a sparking core to one running work speculatively.

3.6.2 GpH-SMP Multicore Performance

Program Name	**Speedup**	**Lines Code**	**Lines Changed**	% **Changed**	**Paradigm**
Clausify	6.6	101	6	6	Chunked Data Parallelism
Rewrite	5.3	408	14	3	Chunked Data Parallelism
Sphere	4.0	332	12	4	Nested Data Parallelism
Boyer	3.4	295	9	3	Chunked Data Parallelism
Fft2	2.7	705	13	2	Data Parallelism
Primetest	2.0	112	15	13	Chunked Data Parallelism
Hidden	1.8	316	6	2	Nested Data Parallelism
Para	1.2	274	3	1	Data Parallelism
Geometric Mean	**2.9**		**9**	**3.2**	

Table 9: GpH-SMP Programs Improved.

Table 9 reports the programming effort and parallel performance of programs improved by GpH-SMP on 8 cores. As a semi-explicit parallel language, GpH requires the programmer to identify a suitable parallel paradigm and introduce evaluation strategies to apply it. Introducing the parallelism requires changing an average of just 9 lines in each program, i.e. 3.2% of the code, and we discuss this further in Section 3.7.1.

The table shows that GpH-SMP improves more than half of the programs, 8 out of 15 programs. The geometric mean speedup is 2.9, with a best speedup of

6.6 for Clausify. It is impressive that 3 of the programs achieve speedups of 4 or more on 8 cores, i.e. a parallel efficiency of 50% or more.

3.6.3 GpH-GUM Multicore Performance

Program Name	**Speedup**	**Lines Code**	**Lines Changed**	% **Changed**	**Paradigm**
Clausify	4.5	101	6	6	Chunked Data Parallelism
Boyer	3.5	295	9	3	Chunked Data Parallelism
Rewrite	2.5	408	14	3	Chunked Data Parallelism
Sphere	1.8	332	12	4	Nested Data Parallelism
Fft2	1.7	705	13	2	Data Parallelism
Geometric Mean	**2.6**		**10**	**3.4**	

Table 10: GpH-GUM Programs Improved.

Table 10 reports the programming effort and parallel performance of programs improved by GpH-GUM on 8 cores. Only 12 of the 15 programs are attempted for GpH-GUM as Compress, Hidden and Primetest import modules not available in GHC 4.06.

As before, GpH requires the programmer to identify a suitable parallel paradigm and apply it. Introducing the parallelism requires changing an average of just 11 lines of each of these programs, i.e. 3.4% of the code, and we discuss this further in Section 3.7.1.

The table shows that GpH-GUM improves nearly half of the programs, 5 out of 12 programs. The geometric mean speedup is 2.6, with a best speedup of 4.5

for Clausify.

3.6.4 Eden Multicore Performance

Program Name	**Speedup**	**Lines Code**	**Lines Changed**	% **Changed**	**Paradigm**
Clausify	6.2	101	7	7	Data Parallelism
Rewrite	4.7	408	15	4	Chunked Data Parallelism
Boyer	3.7	295	14	5	Chunked Data Parallelism
Fft2	3.7	705	11	2	Data Parallelism
Compress	1.6	109	3	2	Data Parallelism
Sphere	1.5	332	7	2	Data Parallelism
Geometric Mean	**3.1**		**8**	**3.2**	

Table 11: Eden Programs Improved.

Table 11 reports the programming effort and parallel performance of programs improved by Eden on 8 cores. Eden requires that the programmer identify a suitable parallel paradigm and introduce an appropriately parameterised algorithmic skeleton to exploit it. This set of programs all use the master-worker skeleton discussed in Section 3.2.2, but some do so directly, while others like Boyer and Rewrite chunk the input to improve thread granularity. Introducing the parallel coordination requires changing an average of just 8 lines in each program, again just 3.2% of the program text.

The table shows that Eden improves a slightly smaller fraction of the programs than GpH-SMP and GpH-GUM, i.e. just 6 of the 15 programs. The maximum speedup of 6.2 is similar to GpH-SMP (6.6), and the geometric mean speedup is

slightly greater, 3.1. It is impressive that 4 of the programs achieve speedups of 3.7 or more on 8 cores, , i.e. a parallel efficiency of 46% or more.

3.7 Comparative Study

This section compares the parallel performance of the four Haskell languages and the programming effort required to achieve that performance. Table 12 summarises the key metrics from section 3.6.

Description	**FDIP***	**GpH-SMP**	**GpH-GUM**	**Eden**
No. Programs Measured	15	15	12	15
No. Programs Improved	3	8	5	6
% Programs Improved	20%	53%	42%	40%
No. Lines Changed	0	9	10	8
% Code Changed	0	3.2%	3.4 %	3.2 %
Geometric Mean Speedup	1.4*	2.9	2.6	3.1
* Performance on 4 Cores				

Table 12: Comparative Multicore Performance Summary

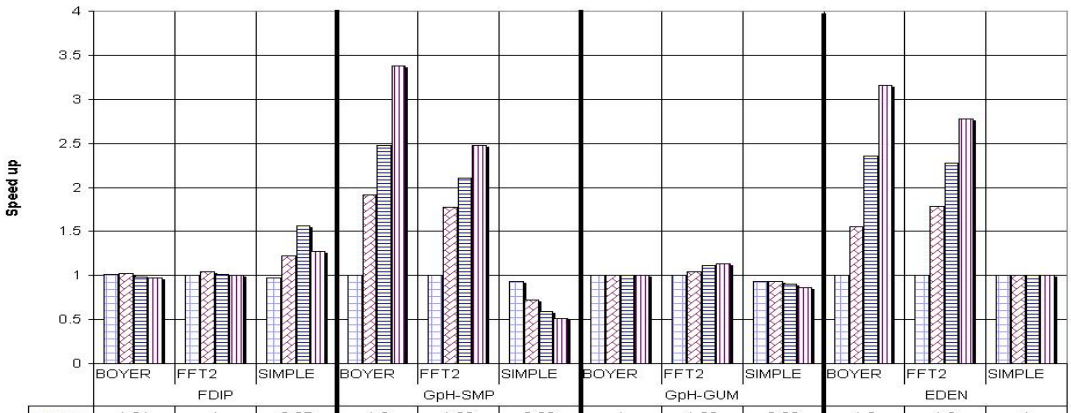
3.7.1 Programming Effort Comparison

As a purely implicit approach, FDIP requires minimal programmer effort, simply the execution of a profiling run. In contrast GpH and Eden both require programmer effort, to time profile the program, to insert evaluation strategies or skeletons and to tune the parallel performance. Tables 9, 10, and 11 show that the scale of the program changes is on average small, in both absolute and relative terms, e.g. representing just 10 lines or 3.4% of the program text in both languages. We

conclude that, for these relatively simple programs, using existing Eden skeletons represents a similar level of coordination abstraction to evaluation strategies used in GpH.

The results also illustrate that, in both GpH and Eden, some programs are easier to parallelise than others. That is, the scale of program changes induced by parallelisation may vary significantly in both absolute and relative terms. For example, Table 9 shows that in GpH, the number of lines changed may vary from 3 to 15, and the percentage of program text may vary from 1% to 13%. Similarly, Table 11 shows that in Eden, the number of lines changed may vary from 3 to 15, and the percentage of program text may vary from 2% to 7%.

Although the parallelisation changes are small, the number of source lines of code metric does not reflect the programmer effort expended on understanding the program, on sequential time/space profiling, and on investigating alternative parallelisations. While time/space profiling is a fast and routine activity, the key intellectual challenge is to understand the computational structure of a program written by another programmer. Some, like Clausify, are simple but others, like SCS, are far more complex. The complexity in all programs comes from the dependencies between the functions, and typically the longer the program the harder it is to complete. The time to comprehend programs was not measured systematically, but the mean time is roughly estimated at several days. Of course this effort would be minimised if programmers are parallelising a program they themselves wrote. Once the program is understood, only half a working day is



	FDIP			GpH-SMP			GpH-GUM			EDEN		
	BOYER	FFT2	SIMPLE	BOYER	FFT2	SIMPLE	BOYER	FFT2	SIMPLE	BOYER	FFT2	SIMPLE
1 PE	1.01	1	0.97	1.0	1.00	0.93	1	1.00	0.93	1.0	1.0	1
2 PE	1.02	1.04	1.22	1.9	1.77	0.72	1	1.04	0.93	1.6	1.8	1
3 PE	0.98	1.01	1.56	2.5	2.11	0.59	1	1.11	0.9	2.4	2.3	1
4 PE	0.97	1	1.27	3.4	2.48	0.51	1	1.13	0.86	3.2	2.8	1

Figure 14: Comparing the Performance Scalability of Parallel Haskells on 4 Cores.

required to introduce and tune the parallelism.

The parallel paradigms used in the improved programs are all forms of data parallelism, sometimes combined with *chunking* to increase thread granularity, or *nesting* to introduce additional parallelism. Section 3.2 outlines the chunking data parallelism in the Boyer program.

3.7.2 Scalability

A key property of a parallel implementation is *scalability*, i.e. whether performance increases as processing elements are added. We have already seen the scalability of the GpH-SMP, GpH-GUM and Eden implementations up to 8 cores in the discussion of Figure 12.

Figure 14 provides a more detailed analysis for three programs (Boyer, FFT2 and Simple) in each language on 1, 2, 3 or 4 cores. Each program gives good

performance on at least one implementation. The figure shows that, in GpH-SMP, GpH-GUM and Eden, the performance of programs that speed up, i.e. Boyer and FFT2, improves steadily as cores are added. In contrast FDIP delivers the best speedup for Simple on 3 cores. This is not an isolated result: the 5 programs delivering speedups under FDIP reported in [46] deliver maximum speedup twice on 3 cores, and three times on 4 cores.

Furthermore, FDIP ceases to scale beyond 4 cores [103] and this is illustrated by the 4 core performances of Boyer, Simple and FFT2 in Figure 14, which are uniformly better than the 8 core performances reported in Figure 15. The reasons for this have not been established but are likely to be either lock contention or low-level memory effects, e.g. disrupting caches when transferring threads between cores.

3.7.3 Performance Comparison

A complete comparison of the 8 core speedups achieved for all 15 programs in the four languages is depicted in Figure 15 and summarised in Table 13.

The performance price of FDIP's purely implicit approach is high, and it is the least effective of the languages surveyed here. It improves the fewest programs: 3 out of 15 on 4 cores (Table 8), and 2 out of 15 on 8 cores (Figure 15). Moreover, the geometric mean and maximum speedup are both relatively small, at 1.4 and 1.8 respectively, on 4 cores. However, a geometric mean speedup of 1.4 on 4 cores shows parallel efficiency approaching that of the semi-explicit implementations,

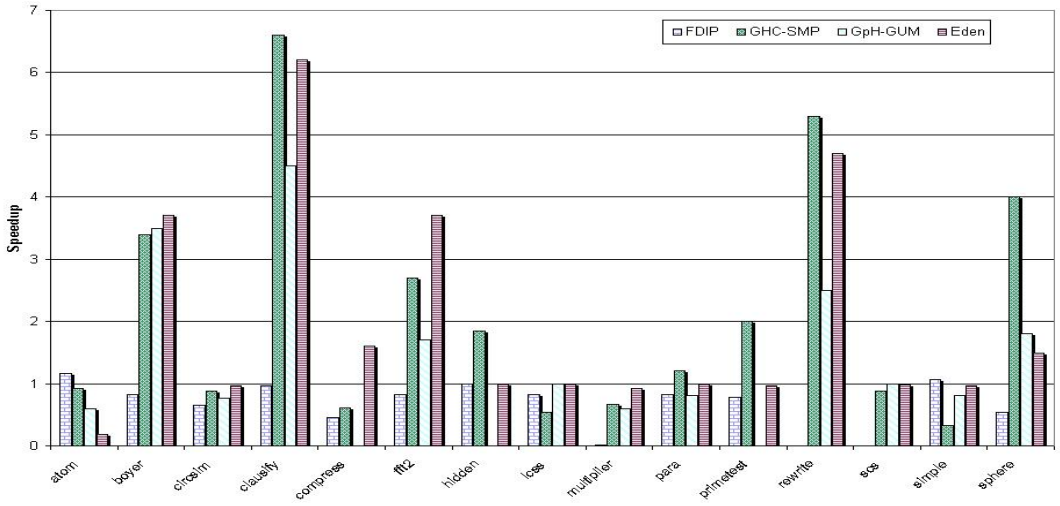


Figure 15: Performance Comparison of Parallel Haskells (8 cores)

i.e. speedups of approximately 3.5 on 8 cores. However, FDIP parallelism scales both irregularly, and only to a limited extent. That is, FDIP does not deliver significant performance gains beyond 4 cores, and it is hard to predict how many cores will deliver the maximum performance, as outlined in section 3.7.2.

The performance of GpH-SMP and Eden is broadly similar. The geometric mean of speedups are similar: 1.5 for GpH-SMP and 1.5 for Eden, as are the maximum speedups: 6.6 for GpH-SMP and 6.2 for Eden. However, Eden improves a smaller percentage of the programs: 40% (6/15) compared with 53% (8/15) for GpH-SMP.

The performance of GpH-GUM is marginally worse than GpH-SMP and Eden, with geometric mean speedup of 1.4 and maximum speedup of 4.5 GpH-GUM improves an intermediate percentage of programs, i.e. 42% (5/12).

We analyse the implications of these relative performances in section 3.8.2.

Program Name	**FDIP**	**GpH-SMP**	**GpH**	**Eden**
Atom	1.27	0.93	0.6	0.6
Boyer	0.8	3.40	3.50	3.70
Circsim	.65	0.88	0.77	0.96
Clausify	0.96	6.60	4.50	6.20
Compress	0.43	0.61	Not tested	1.60
FFT2	0.82	2.70	1.70	3.70
Fibheaps	0.83	Not tested	Not tested	Not tested
Hidden	1.80	1.84	Not tested	1.00
Lcss	0.82	0.54	0.93	0.99
Multiplier	0.00	0.67	0.93	0.93
Para	0.83	1.20	0.93	1.00
Primetest	0.77	2.00	Not tested	0.97
Rewrite	0.00	5.30	2.50	4.70
Scs	0.00	0.88	0.98	0.98
Simple	1.27	0.33	0.86	0.97
Sphere	0.53	4.00	1.80	1.50
Geometric Mean	**0.80**	**1.50**	**1.40**	**1.50**

Table 13: Comparative Speedup of Parallel Haskells on Multi-core Machine

3.8 Conclusion

This section summaries the comparison results and concludes the key lesson learnt. It outlines the advantages and disadvantages of using functional programming to exploit multicore architectures. Moreover, it suggests enhancing GpH with a new technique for exploiting multicore architectures.

3.8.1 Summary

The preceding sections have reported the first comparison of functional multicore technologies and are some of the first ever GpH-GUM and GpH-SMP multicore results. The study reflects the current state of the technology and compares the programming effort each variant requires with the parallel performance delivered.

We have contrasted a "no pain" parallel language with three "low pain" languages. The comparison uses 15 programs selected from the representative parts of the Nofib suite and hence reflects the multicore performance that might be expected for a typical set of Haskell programs (Section 3.4).

Although the parallel Haskell implementations all use GHC, they each use a different version, and hence the primary performance comparisons are based on absolute speedups, which normalise against sequential performance. To ground the speedup comparisons we have reported sequential and parallel runtimes and efficiencies for three of the languages. We have found that sequential runtimes vary by as much as a factor of 2.1, and 8-core runtimes by as much as a factor of 4.1. On a single core GpH-SMP is fastest and GpH-GUM slowest, and sequential efficiencies vary between 72% and 100%. Finally, runtime and speedup graphs show that GpH-SMP, GpH-GUM and Eden parallel performance scales, i.e. runtimes fall consistently as cores are added (Section 3.5).

We have reported detailed parallel performance and programming effort studies (Section 3.6), and made a comparative study with the following key results (Section 3.7).

- FDIP's purely implicit approach requires minimal programmer effort. In contrast, GpH and Eden both require programmer effort to understand the program's computational structure, to profile it, to insert parallel coordination, and to tune the parallel performance. As the languages provide

high levels of coordination abstraction the program changes are small, on average no more than 4.3% of the program text in both languages. We conclude that Eden skeletons represent a similar high level of coordination abstraction to evaluation strategies in GpH (Section 3.7.1).

- While GpH-SMP, GpH-GUM and Eden all scale consistently up to 8 cores, FDIP does not scale beyond 4 cores and may deliver best performance on 3 or 4 cores (Section 3.7.2).

- The performance price of FDIP's purely implicit approach is high: it improves the fewest programs (just 3 out of 15) and the geometric mean and maximum speedup are both relatively small at 1.4 and 1.8 respectively on 4 cores (Section 3.7.3).

- All three semi-explicit approaches improve approximately half of the programs, and the performance of GpH-SMP and Eden is broadly similar with geometric mean and maximum speedups of approximately 3.1 and 6.5. GpH-GUM performance is marginally worse with geometric mean speedup of 2.6 and maximum speedup of 4.5. (Section 3.7.3).

3.8.2 Discussion

As multicores become the dominant processor technology, it is beneficial that functional languages realise their theoretical potential to exploit them effectively.

Our study reflects some of the technologies emerging to do so, namely four multicore Haskell implementations, and the results have a number of implications for the field.

It is clear that purely implicit parallelism remains an elusive goal. The FDIP approach speeds up fewer programs, with smaller speedups, and does not scale well. While it is not clear that the scaling issues with FDIP are fundamental, the move towards many cores will make scalability a crucial property for languages and implementations.

It might be seen as discouraging that, even in the low pain languages, only half of the programs deliver absolute speedups (Table 13), and that the geometric mean parallel efficiencies are only around 45% (Tables 9, 10, and 11). However, recall that these programs were neither designed to be parallel, nor selected for their inherent parallelism. While some algorithms will remain inherently sequential, it is likely that, with thoughtful design, a far higher percentage of programs can be effectively parallelised. Moreover, the implementations are evolving fast and we can expect greater parallel efficiencies in the near future.

Interestingly, Eden, with an implementation designed for distributed memory architectures, performs fractionally better than GpH-SMP, which is designed for multicores. Similarly the geometric mean speedups of GpH-GUM, with an architecture designed for both distributed and shared memory systems, are within 11% of the GpH-SMP results. These implementations must have significant advantages to outweigh the massive communication and synchronisation overheads

incurred by serialising heap, calling expensive communication libraries, and deserialising heap.

We argue that *the key reason for the good performance of Eden and GUM is the maintenance of independent heaps* using a message-passing architecture. Independent heaps convey four significant advantages for shared-memory systems like multicores. They enable cores to garbage collect independently, they also confine synchronisation to both limited and large-grain memory areas, i.e. the message buffers, and simplify cache coherency issues (Section 3.3.5). We further predict that, as multicore scale to many cores, the advantages of independent heaps will be greatly magnified and that some form of thread-private heap, e.g. [34], will be essential on these architectures.

There are many encouraging signs for multicore functional languages. The GpH and Eden semi-explicit approaches deliver effective high level coordination, and hence require very small program changes, and perhaps only half a working day to introduce and tune the parallelism for a known program. The fact that there are 4 multicore Haskells to compare reflects the level of interest in addressing the challenges. The implementations have considerable room for improvement. For example where GpH-GUM and Eden currently use very naive distributed memory implementations, these could be significantly improved for shared-memory multicores, e.g. using shared-memory variants of the communication libraries. Another promising line of future work is to integrate distributed and shared-memory implementations to better exploit the increasingly ubiquitous

clusters of multicore architectures.

This thesis investigates the issue of improving GpH-GUM to achieve better exploitation of clusters of multicore architectures. In chapters 5 and 6, we investigate the possibility of extending parallel Haskells with architecture-aware constructs.

Chapter 4

Parallel Programming Practice

This chapter presents two areas of research into practical aspects of parallel functional programming. The first is one of the first uses of a new formulation of evaluation strategies for GpH and includes undertaking a systematic benchmarking of the new formulation against the evaluation section of [77] (Section 4.1).

The second is to investigate the thread granularity required to achieve good performance from a distributed-memory parallel functional language, e.g. Eden and GpH, on multicores. The initial Eden results reported in [3] are extended to be more systematic and to cover GpH-GUM and GpH-SMP (Section 4.2).

4.1 Using and benchmarking New Evaluation Strategies

The evaluation strategies are a key concept for specifying pure, deterministic parallelism in Haskell programs. With strategies, parallel specifications can be built up in a compositional way, and the parallel coordination can be separated from the algorithm. In addition, to what original strategies provide, the new strategies introduce an evaluation-order monad to provide clearer, more generic, and more efficient identification of parallel evaluation. The new formulation resolves a subtle space management issue with the original strategies, allowing parallelism (sparks) to be preserved while reclaiming heap associated with superfluous parallelism.

4.1.1 Original Strategies

We start by describing the original evaluation strategies. The basic operations provided for parallel Haskell programming are `par` and `pseq`:

```
par :: a -> b -> b
pseq :: a -> b -> b
```

The `par` construct annotates an expression (its first argument) as being potentially beneficial to evaluate in parallel, and evaluates to the value of its second argument. The `pseq` construct expresses sequential evaluation ordering: its first argument is evaluated, followed by its second. The `par` construct represents an

overlap between lazy evaluation and future execution. To benefit from lazy evaluation and futures, expressions must have a representation in which their value may be demanded later. Thus, those expressions may evaluate in parallel for future demand. This is what the `par` operator provides, annotating a lazy computation as being potentially profitable to evaluate in parallel, in effect turning a lazy computation into a future [78].

While a detailed control of evaluation order and degree is necessary in order to achieve significant parallelism. The unstructured use of `par` and `pseq` constructs in non-trivial GpH programs could lead to unnecessarily obscure programs. This problem can be overcome by using evaluation strategies.

The `par` and `pseq` constructs provide the raw material for expressing parallelism in Haskell. Evaluation strategies are high level abstraction functions built on top of them, allowing the expression of large scale parallel algorithms. A strategy specifies the dynamic behaviour required when computing a value of a given type. A strategy makes no contribution towards the value being computed by the algorithmic component of the function: it is evaluated purely for effect, and hence it returns just the unit tuple (). Therefore, the type of strategy is:

```
type Strategy a = a -> ()
```

The strategy may fully or partially evaluate its argument and may not perform any evaluation. Therefore, strategies specifying only the degree of evaluation are essential. The simplest strategy which introduces no parallelism is:

```
r0 :: Strategy a
r0 x = ()
```

`r0` is a strategy that evaluates none of its argument. This strategy is useful in a case that we need to evaluate only the first element of a data structure but not the second.

```
rnf :: Strategy a
rnf  x = x `pseq` ()
```

`rwhnf` evaluates its argument to weak-head normal from, the default evaluation degree in GpH. The `rwhnf` evaluation is not enough, many expressions can be reduced to normal form. The `rnf` is a strategy for reducing an expression to normal form.

```
rnf :: NFData a => Strategy a   -- reduce to  normal form
class NFData a  where
  rnf :: Strategy a
  rnf = rwhnf                   -- default definition
```

NFData is a class that specifies the evaluation degree of a evaluated value. For each data type, an instance of the `NFData` class must be declared that specifies how to reduce its value to normal form.

The `using` function applies a strategy to an expression, and to data structure `x` before returning it.

```
using :: a -> Strategy a -> a   -- applies a strategy to an expression
x `using` s = s x `pseq` x
```

The strategies described above do not contain any actual parallelism. A basic parallel strategy is `parList`, which applies a strategy to each element of a list in parallel:

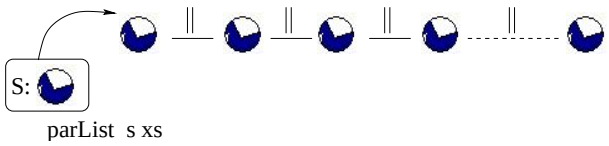


Figure 16: parList Strategy

```
parList :: Strategy a -> Strategy [a]
parList strat []     = ()
parList strat (x:xs) = strat x `par` parList strat xs
```

The function `parList` illustrates the compositional nature of the strategies abstraction achieved through the usage of the higher-order functions: it takes as an argument a strategy to apply to each list element and returns a strategy for the whole list. Circles shown in Figure 16 represent the list of computations and the parallel vertical bars indicate that the `parList` function evaluates its elements in parallel. The strategy argument `s` is typically used to specify the *evaluation degree*, that is, how much each list element should be evaluated. For instance, `parList rwhnf` causes each spark to evaluate its list element as far as the top-level constructor, whereas `parList rnf` evaluates the elements completely.

The `parList` function can also be used to illustrate the modular nature of strategies; for example:

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

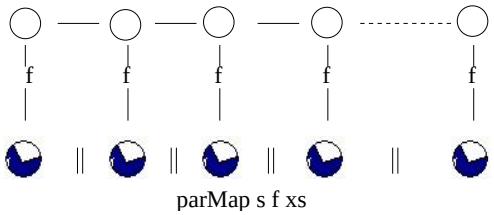


Figure 17: parMap Strategy

The `parMap` function takes a strategy `strat`, a function `f`, and a list `xs` as arguments and maps the function `f` over the list in parallel, applying `strat` to every element, as shown in Figure 17. The empty circles in the figure represent the list of elements before the function `f` is applied, while the filled circles represent the list of the function `f` applied on each element. Note how the construction of the result with `map`, on the left of `using`, is separate from the specification of the parallelism, on the right. This is a small-scale example, but the idea also scales to much more elaborate settings [70].

4.1.2 Space Leak Problem

This section describes a space problem with the original strategies. The discussion is from Marlow in *Runtime Support for Multicore Haskell* [80].

When the `par x y` expression is evaluated, the runtime system saves a pointer to the heap node representing x in a data structure called `sparkpool`. The `sparkpool` is a set of pointers to heap objects representing computations that have been sparked by `par`. This `sparkpool` is consumed by the runtime system: whenever there is an idle processor the runtime removes one spark from the pool.

There are two policies to treat the `sparkpool` data structure during garbage

collection:

- The `ROOT` policy: sparks in the `sparkpool` are treated as roots for the garbage collector. Thus the garbage collector retains all sparks and objects which they point to in the heap.
- The `WEAK` policy: the garbage collector retains only sparks that are reachable from the roots of the program.

In fact, both of these policies lead to problems with original strategies. First, let us consider `WEAK`, and examine how it works with the definition of `parList` in the previous section. The sparks created by `parList` are all expressions of the form strat x for some strategy `strat` applied to some list element `x`. Now, every such expression is uniquely allocated for the sole purpose of being passed to `par`; the sparkpool will contain references to many expressions of the form `strat x`, and in every case, *the reference from the spark pool is the only reference to that expression in the heap.* So, by definition, if we adopt the `WEAK` policy then every spark created by `parList` will be discarded by the garbage collector, and we lose all the parallelism.

Moreover, there is no definition of `parList` that can avoid this problem. The only value that the `parList` strategy can return is (), so the only way that `parList` can create a reachable spark is by sparking part of the structure it was originally given, such as the list elements. For example, we can define a non

parametric variant of `parListWHNF` as follows [1]:

```
parListWHNF :: Strategy [a]
parListWHNF []     = ()
parListWHNF (x:xs) = x `par` parListWHNF xs
```

But unfortunately we lose the compositional nature of strategies that was so appealing about the original formulation.

So what about the alternative garbage collection policy, `ROOT`, where we treat the spark pool as a source of roots? Considering the `parList` example again, the spark pool would still contain references to expressions of the form `strat x` in the heap, but this time, all the expressions will be retained by the garbage collector and no parallelism is lost. However, another problem arises: what happens when there are not enough parallel processors to evaluate all the sparks? The spark pool retains references to all the `strat x` expressions, perhaps long after each `x` is no longer required by the program and would otherwise be reclaimed by the garbage collector.

In an attempt to retain potential parallelism, the storage manager is retaining memory that should have been released: this is a space leak, and can and does have dramatic performance implications

[1] Non parametric compare with parametric parList :: Strategy a -> Strategy [a]

4.1.3 New Evaluation Strategies

In collaborative work with Marlow et al., we have developed and measured a new version of evaluation strategies. The new version preserves the key compositionality and modularity benefits of the original strategies, together with their low time and space overheads. The new Evaluation Strategies are developed on top of the `Eval` monad [77]. Before we go into any details of the new strategies, let us mention the reason leading to its development. The main reason behind rewriting the strategies is the difficulty of managing the space behaviour of sparks in the original strategies, particularly apparent on multicores. If a strategy is a function returning the unit type (), then there is no way for it to spark new expressions and to return them to the caller, thus ensuring that the sparked expressions remain reachable from the callers' heap. The key idea in the reformulation is that a strategy returns a new version of its argument, in which the sparked computations have been embedded. For example, when sparking a new parallel task of the form `strat x`, rather than discarding this expression, the strategy will now build a new version of the original data structure with `strat x` in place of `x`. The caller will consume the new data structure and discard the old, so that the parallel task `strat x` remains reachable as long as the consumer requires it. Furthermore, if the consumer evaluates `strat x` before it is evaluated by a parallel thread, then the spark fizzles; superfluous parallelism is discarded by the compiler garbage collector, which is exactly what we need. We build an elaborate

optimisation machinery to avoid unnecessary computation in trivial cases.

The new version reformulates the strategy type to return the value instead of unit.

```
type Strategy a = a -> Eval a
```

The strategies are identity functions, and do not contribute to the result other than to specify how the result will be computed. This is guaranteed for all functions defined in the strategies module but not for user defined strategies. The `Eval` monad is just a strict identity monad, and `runEval` is used to lift and unlift the result. A lifted type means that a strategy like `r0`, that performs no evaluation, can still return a (lifted) result.

```
data Eval a = Done a
runEval :: Eval a -> a
runEval (Done a) = a
```

The strategies library adds two useful strategies: `rpar` sparks the argument to be evaluated in parallel; and `rseq` evaluates the argument to weak head normal form.

```
rpar :: Strategy a
rpar x = x `par` Done x
rseq :: Strategy a
rseq x = x `pseq` Done x
```

In case of a composed data structure, if the user wants to apply different evaluation degree on a different element the `r0` strategy is redefined as follows:

```
r0:: Strategy a
r0 x = return x
```

The default reduction of lazy evaluation in GpH is Weak Head Normal Form (WHNF). WHNF means that the expression is evaluated as far as the top-level constructor. For example, if the expression is a list, then the applied strategy would perform enough evaluation to determine whether the list is empty ([]) or nonempty (_:_), but would not evaluate the head nor evaluate the tail of the list. In order to apply the strategy to all elements in the list, the `rdeepseq` strategy is used. The `rdeepseq` evaluates its argument to normal form.

```
rdeepseq :: NFData a =>  Strategy a
rdeepseq x = rseq (deepseq x)
```

where `deepseq` from the DeepSeq module traverses the entire data structure evaluating it completely. `NFData` is a class for specifying how to reduce a value of type `a` to normal form.

A new `using` function applies a strategy to value `a` and returns `a` as result.

```
using :: Strategy a -> a
x `using`  s = runEval (s x)
```

The `using` function is different from the `using` function in original strategies. The intentions is that all new strategies should be identity functions, but it is up to the programmer to guarantee this if they define a new strategy.

4.1.4 Using Strategies for Parallel Paradigms

This section demonstrates how strategies can be used for indicating parallelism in different parallel paradigms. We will first consider the divide-and-conquer (Section 4.1.4.1) and data-oriented (Section 4.1.4.2) paradigms.

```
let                                  ||   let
    x = fib (n-1)                    ||       x = fib (n-1)
    y = fib (n-2)                    ||       y = fib (n-2)
in                                   ||   in runEval $ do
    x `par` (y `pseq` x + y + 1)     ||       x <- rpar (fib (n-1))
                                     ||       y <- rseq (fib (n-2))
                                     ||       return (x + y + 1)
```

Figure 18: Original Strategies versus New Strategies

4.1.4.1 Task Parallelism

The strict identity monad gives a convenient and flexible notation for expressing evaluation order, i.e. the ordering between applications of `rseq` and `rpar`, which is exactly what a programmer needs for expressing basic parallel evaluation [77]. For example, the left hand side of Figure 18 presents a fragment of the parallel Fibonacci function (`fib`) written using the original `par` and `pseq` operators to indicate that `x` should be evaluated in parallel and guarantee that `y` is evaluated before the expression. Finally the result `(x+y+1)` should be returned. The right-hand side presents parallel code of the same function using `rpar` and `rseq`. The latter clearly expresses the ordering between `rpar` and `rseq`, using a notation that Haskell programmers will find familiar.

Another example to demonstrate how the parallel `Eval` monad basic operation

```
payne :: Int -> [(Int,Int)] -> Int
payne 0   coins     = 1
payne _   []        = 0
payne val ((c,q):coins)
  | c > val   = payne val coins
  | otherwise = (left + right)
                `using` strat
  where
    left  = payne (val - c) coins'
    right = payne val coins
    strat = rnf left `par`
            rnf right
    coins'
        | q == 1    = coins
        | otherwise =(c,q-1):coins
```

Figure 19: Coins Using Original Strategy

```
payne :: Int -> [(Int,Int)] -> Int
payne 0   coins     = 1
payne _   []        = 0
payne val ((c,q):coins)
  | c > val   = payne val coins
  | otherwise = res
  where
    left  = payne (val - c) coins'
    right = payne val coins
    res = runEval $ do
              l <- rpar left
              r <- rseq right
              return (l+r)

    coins'
        | q == 1    = coins
        | otherwise =(c,q-1):coins
```

Figure 20: Coins Using New Strategy

can be used is the simple `Coins` program. The `Coins` program takes a price and a list representing a set of coins, and determines how many different combinations of coins could be used to pay for an object at the given price.

We start with an original strategy parallel version for the main function in the program (shown in Figure 19). The function just returns the number of results, not the results themselves. If the value of coin (`c`) is greater than the value (`val`), then do not pay with this coin. Otherwise, the left branch detects the coin from the value and the right branch computes the number of coins.

Note that the expressions `left` and `right` can be evaluated in parallel. The red lines in Figures 19 and 20 describe how `left` and `right` branches can be evaluated in parallel without affecting the final result. The `left` branch is sparked

with `rpar` construct and bound to the variable `l`; while the `right` branch is evaluated sequentially using `rseq` construct and bound to the variable `r`.

The examples above explain how a programmer can express parallelism using `rpar` and `rseq`. However, a programmer can express parallelism in a smarter way, by using strategies. For example, a `fib` program can be rewritten in term of strategies, as follows.

```
fib n =  res `using` strat
     where
        x = fib (n-1)
        y = fib (n-2)
        res = (x+ y+1)
        strat res = do
            x' <- (rpar `dot` rdeepseq)  x
            y' <- (rseq `dot` rdeepseq)  y
            return (x'+ y'+1)
```

The `strat` describes how two subcomputations x and y should be evaluated.

A Divide-and-Conquer Pattern: the main power of strategies is the possibility of building abstractions over patterns of parallel computation. One way of such abstraction is to define high level functions that can be parameterised to exploit parallelism, commonly known as algorithmic skeletons [32]. A divide-and-conquer skeleton is shown in Figure 21. All coordination aspects of the function are encoded in the strategy `strat`, which describes how the two subcomputations `l` and `r` should be evaluated. The thresholding predicate `threshold`, provided by the caller, places a bound on the depth of parallelism, and this is used by `strat`

```
divConq :: (Ord a,Num a) => (a -> b) -> a -> (a -> Bool) ->
           (b -> b -> b) -> (a -> Bool) -> (a -> (a,a)) -> b
divConq f arg threshold conquer divisible  divide
  | not (divisible arg) = f arg
  | otherwise = conquer l r
    where
      (lt,rt)  = divide arg
      left = divConq f lt threshold conquer divisible divide
      right = divConq f rt threshold conquer divisible divide
      (l,r) = (left, right) `using` strat
      strat (l,r)
               | (threshold arg) = (evalTuple2 rseq rseq) $(l,r)
               |  otherwise =
                         (evalTuple2 (rpar `dot` rseq)
                         (rpar  `dot` rseq)) $ (l,r)
```

Figure 21: Divide-and-Conquer Skeleton

to decide whether to spark both `l` and `r` or to evaluate them sequentially. The `evalTuple2` is a strategy that evaluates a tuple according to a given strategy. The definition of `diverging` achieves a separation between the specifications of the algorithm and the parallelism, the latter being confined entirely to the definition of `strat`.

4.1.4.2 Data-oriented Parallelism

Data parallelism can be classified into: flat data parallelism and nested data parallelism. A flat data parallelism applies a function to a collection of elements. In this case, it is not appropriate to spawn a thread for each element. Because in most cases it leads to very tiny threads that do not offset their creation, scheduling and other overheads. The appropriate way to parallelise this class of problem is to chunk the collection into suitable blocks, depending on the available processors and the computational size of each element. Nested data parallelism applies a

function over a collection of elements but each of these elements may applies another function over another data structure. Thus, it is irregular parallelism where each task may take a different time.

In strategies, parallelism is expressed by applying a higher order function to collections of data. The `map` function that applies a function over a list of elements can be parallelised as follows:

```
parMap f xs = map f xs `using` parList rdeepseq
```

The benefits of the strategies are not only the separation of the algorithm from the parallelism, but also the reuse of the `map` function. The `parList` function is a strategy that applies `f` to each element in the list in parallel. It defined as following:

```
parList ::  Strategy a -> Strategy [a]
parList  strat [] = []
parList  strat (x:xs) = do
                  x' <- rpar ( x `using` strat)
                  xs' <- parList strat xs
                  return(x':xs')
```

The `rpar` creates a spark to evaluate the current element. The spark evaluates `( x `using` strat)` which applies a strategy `strat` to `x`.

A more advanced strategy like `parBuffer n s xs` yields a list in which evaluation of the ith element induces parallel evaluation of the $(i + n)$th element with the first n elements being evaluated in parallel immediately.

```
parBuffer :: Int -> Strategy a -> Strategy [a]
parBuffer n s = evalBuffer n (rpar dot s)
```

The result list must therefore be lazy, at least beyond the first n elements.

4.1.5 Evaluation of the New Strategies

This section discusses our measurements in detail, but first we summarise the key results:

- For all programs, the speedup and runtime results with original and new strategies are very similar, giving us confidence that they specify the same parallel coordination for a range of programs and parallel paradigms (Figure 23).
- The speedups achieved with the new strategies are slightly better compared to those with the original strategies: a mean of 3.85 versus 3.72 across all applications (Columns 3 & 2 of Table 16).
- The new strategies fix the space leak outlined in Section 4.1.2, and better support speculative parallelism.
- The sequential run-time overhead of the new strategies is very low: a mean of 1.91% (Table 15). Memory overheads are low for most programs (Columns 8 – 11 of Table 16).

4.1.5.1 Apparatus

Our measurements were made on an eight-core, 8GB RAM, HP XW6600 Workstation comprising two Intel Xeon 5410 quad-core processors, each running at 2.33GHz. The benchmarks are run under Linux Fedora 7 using a recent development GHC snapshot (6.13 as of 20.5.2010), and parallel packages 1.1.0.1 and

2.3.0.0, for original and new strategies respectively. The data points reported are the median of 3 executions and we measure up to 7 cores, as measurements on the 8th core are known to introduce some variability.

Our benchmarks are 10 parallel applications from a range of application areas; some have previously been studied [70, 68] and others are taken from the GHC Nofib suite and parallelised [8]. The programs are the computational kernels of realistic applications, cover a variety of parallel paradigms and employing several important parallel programming techniques, such as thresholding to limit the amount of parallelism generated and clustering to obtain coarser thread granularity. Table 14 summaries the program characteristics.

Program	Application Area	Paradigm	Regularity
`LinSolv`	Symbolic Algebra	Nested Data Parallelism	Limit irregular
`Sphere`	Graphic	Nested Data Parallelism	High irregular
`Hidden`	Graphic	Nested Data Parallelism	Irregular
`Coins`	Search Application	Divide-and-Conquer	Irregular
`MiniMax`	Game Application	Divide-and-Conquer	Irregular
`Queens`	Game Application	Divide-and-Conquer	Regular
`Genetic`	Scientific Application	Divide-and-Conquer	Irregular
`MatMult`	Numeric	Divide-and-Conquer	Irregular
`Maze`	Scientific Application	Data Parallelism	Irregular
`TransClos`	Scientific Application	Data Parallelism	Irregular

Table 14: Programs Characteristics

`Genetic` is a program aligns RNA sequences from related organisms. The parallel paradigm used in the program is divide-and-conquer parallelism and nested data parallelism. The sequential and parallel versions of the program are well optimised in [7]. The parallelism is in a divide-and-conquer style, using **par** and **seq**

on the top level function. The second source of parallelism is in a data parallel style, using the `parMap` function.

`MiniMax` performs an alpha-beta search in a tree representing positions in a two-player game. The program is in a divide-and-conquer style and laziness is exploited to prune unnecessary subtrees. The program is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game.

The `Queens` program places chess pieces on a board. The `Queens` is intended to solve the n-queens problem. The program is implemented using divide-and-conquer parallelism with an explicit threshold.

`LinSolv` finds an exact solution to a set of linear equations, employing the data parallel multiple homomorphic images approach often used in symbolic computation.

`Hidden` performs hidden-line removal in 3D rendering and uses data parallelism via the `parList` strategy.

`Maze` searches for a path in a 2D maze and uses speculative data parallelism.

`Sphere` is a ray-tracer from the Haskell Nofib suite, using nested data parallelism, implemented as `parMap`.

`TransClos` finds all elements that are reachable via a given relation from a given set of seed values, i.e. that are in the range of the transitive closure of the given relation. The algorithm uses a queue-based `parBuffer` over an infinite list.

`Coins` computes the number of ways of paying a given value from a given set of coins, using a divide-and-conquer paradigm.

Program	Sequential Runtime (seconds)	Δ Time (%)	
		Original Strategies	New Strategies
`LinSolv`	23.40	+0.90	+1.97
`Sphere`	21.11	+4.78	+3.32
`Hidden`	41.49	+8.41	+2.70
`Coins`	42.49	+1.11	+2.12
`MiniMax`	36.98	+0.87	+3.22
`Queens`	25.51	+1.37	+6.12
`Genetic`	33.46	+2.96	+3.97
`MatMult`	35.48	-1.35	-2.06
`Maze`	40.93	-2.22	-3.59
`TransClos`	83.13	+0.75	+1.68
Geom. Mean		+1.72	+1.91

Table 15: Sequential Runtime Overheads

`MatMult` performs matrix multiplication using data parallelism with explicit clustering.

4.1.5.2 Sequential Overhead

Table 15 shows the sequential runtime as baseline, and the difference of the single processor runtime with both original and new strategies. For the new strategies, we encounter a runtime overhead of, at most, +6.12% for the divide-and-conquer style `Queens` implementation. Two programs show a negative overhead and we do not yet understand why this happen. The geometric mean of the runtime overhead with the new strategies (+1.91%) is only slightly higher than with the original strategies (+1.72%). Notably, the data parallel programs have a fairly low overhead, despite the additional traversal of a data structure to expose parallelism. Comparing the runtime overheads imposed by both old and new versions

of the strategies, we do not encounter a consistently higher overhead for the new strategies. This justifies the new strategy approach of high-level generic abstractions.

4.1.5.3 Parallel Performance of Strategies

Runtimes: Figure 22 compares the runtime curves for applications with the original and new strategies. We chose one program from each paradigm that has a similar sequential runtime. They are measured on 1, 2, 3, 4, 5, 6, and 7 cores. The runtime curves are broadly similar for all applications. There is an exception; the New Strategies is slower than Original Strategies for the `Genetic` program on one core. This reflects the claim was made in section 4.1.5.2. The second observation is that both strategies are scaling, i.e. the runtime falls as cores are added.

Speedups: Figure 23 compares the absolute speedup curves (i.e. speedup relative to sequential runtime) for the applications with the original and new strategies. Both the runtime curves (not reported here) and speedup curves for the original and new strategies are very similar. This pattern is seen to be repeated in more detailed analysis, e.g. in Columns 2 and 3 of Table 16. We conclude that the original and new strategies specify the same parallel coordination for a variety of programs representing a range of parallel paradigms, and several tuning techniques.

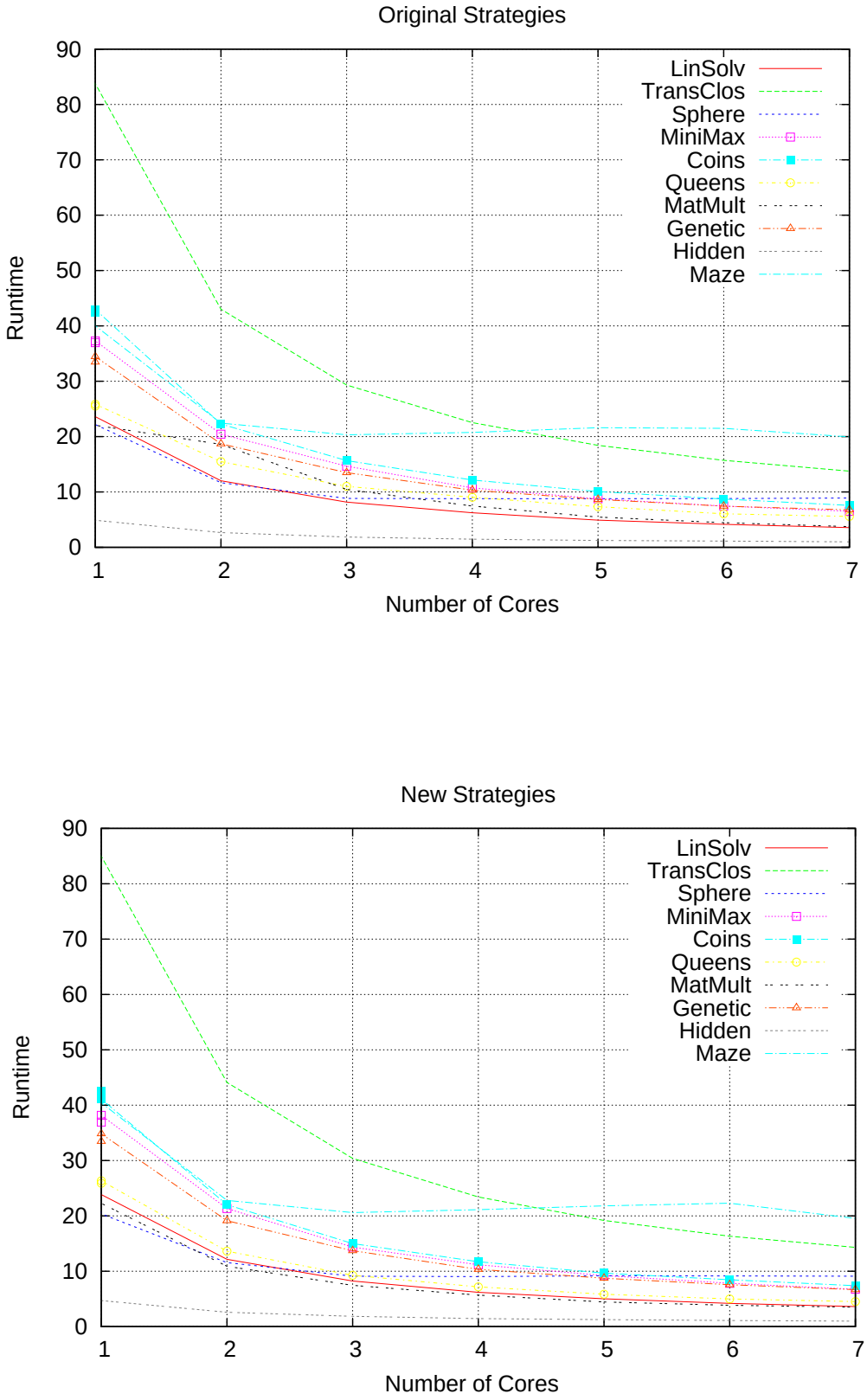


Figure 22: Runtime Comparison of the Original and the New Strategies

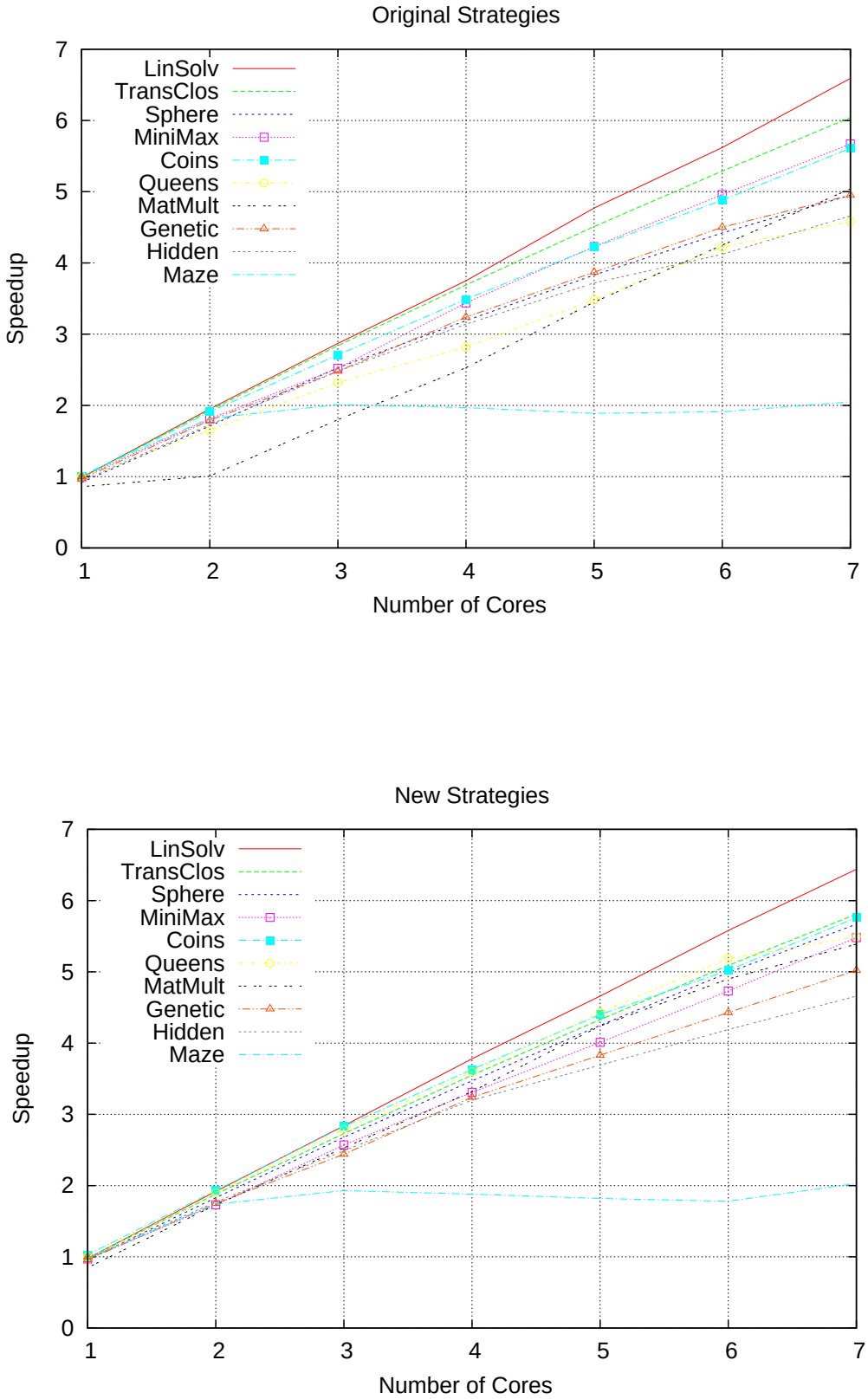


Figure 23: Speedups Comparison of the Original and the New Strategies

	Speedup		Generated Sparks		Converted Sparks		Allocated Heap		Maximum Residency	
	Orig.	New	Orig.	New	Orig.	New	Orig. (MB)	New Δ%	Orig. (KB)	New Δ%
`LinSolv`	**6.59**	6.44	7562	7562	7562	7562	6050.10	+0.15	7104.70	+3.87
`TransClos`	**6.04**	5.81	1041	1041	1041	1040	80174.60	+0.07	108.60	+1.47
`Sphere`	4.95	**5.67**	160	160	160	160	8636.40	-1.14	120943.30	-14.53
`MiniMax`	**5.67**	5.48	1464	1464	1464	163	30476.85	-0.01	98.05	-7.17
`Coins`	**5.61**	5.53	145925	146853	2702	1060	79833.20	+1.59	302.10	+20.36
`Queens`	4.58	**5.49**	1589	1563	1589	636	14903.30	-17.52	19134.50	-24.11
`MatMult`	5.04	**5.39**	100	100	100	100	109.00	-6.97	12272.80	+102.04
`Genetic`	4.95	**5.02**	659	674	659	166	12180.20	-6.75	493.90	+7.88
`Hidden`	4.66	**4.66**	324	324	324	324	4805.50	-0.01	2349.80	-0.44
`Maze`	**2.05**	2.01	2723	2835	2525	481	194122.00	+7.74	71.20	-33.15
Geom. Mean	4.83	**4.96**						-2.51		+1.03

Table 16: Speedups, Number of Sparks and Heap Consumption on 7 Cores.

The top six programs in Table 16 have been carefully tuned for parallelism, and hence are most relevant when assessing the performance of the new strategies. The mean speedups of these programs are 4.83 for the original and 4.96 for the new strategies. The remaining applications have potential for additional performance tuning, and yet none has a significantly lower speedup with the new strategies.

4.2 Granularity Control

4.2.1 The Importance of Thread Granularity

Thread granularity in parallel computing means the balance between the size of parallel computation and its associated communication. In other words, it is the ratio of computation to the amount of communication. The tasks should be neither too fine nor too coarse. The runtime will not be able to effectively load-balance to keep all CPUs constantly busy if there are too few large tasks,

i.e. coarse tasks: the costs of creating and scheduling the tiny tasks outweigh the benefits of executing them in parallel [54]. Many real parallel programs produce small grained computations. For this reason, fine grained computations need to be arranged into groups, to produce large computations, depending on the communication overhead. The key question is: how large do the computations need to be for a parallel model to be effective on a multicore architecture? The answer to this question entails balancing two dynamic properties of a parallel program, namely the thread granularity and amount of communication. We address the question by undertaking a limited study using two different parallel programs. The programs are all measured on common multicore architectures, namely eight core machines, comprising two quad-cores, Intel Xeon 5410 cores running at 2.33GHz, with a 1998 MHz front-side bus 6144 KB and 8GB RAM running under Linux CentOS 5.5.

```
                ....
                ....
                xs = take l  (repeat  n)

foo   [] = []
foo  xs =(map nfib  xs) `using` parList  rdeepseq

nfib ::Int-> Int
nfib 0 = 1
nfib 1 = 1
nfib n = nfib (n-2) + nfib (n-1) + 1
```

Figure 24: `nfibList` Program

4.2.2 Eden Multicore Thread Granularity

A key question arises is how coarse-grained each thread needs to be in order for us to achieve reasonable parallel performance. The answer to this question entails balancing two dynamic properties of a parallel program, namely the thread granularity and the amount of communication.

In Section 6 of [3], we perform a limit study to measure thread granularity on multicore. The results show that the most profitable thread granularity required to achieve reasonable parallel performance for the Fibonacci program is of the order of 0.25ms. This time is converted to its equivalent number of cycles as follows:

$$cycles = CPU\ speed\ (cycle/second)\ *\ TIME\ (second)$$

From the above equation the number of cycles for a thread granularity of 0.25ms in a 2.33GHz processor is $((2.33 * 10^9) * (0.25 * 10^{(-3)}))/10^6 = 0.58$Mcycle is approximately (1 Mcycle). Similarly a thread granularity of 30ms is $((2.33 * 10^9) * (30 * 10^{(-3)}))/10^6 = 69.9$Mcycle (70 Mcycle) is required for programs communicating more data per thread, i.e. `Clausify` and `Rewrite` from the Nofib benchmark suite [89]. The reason for converting the obtained time to Mcycles is to provide an architecture neutral measure.

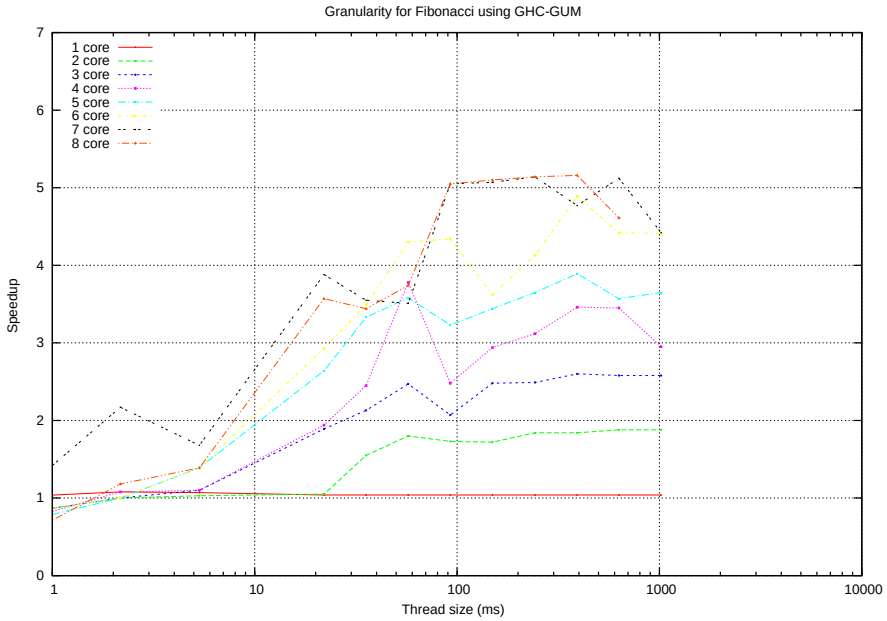
The next section will investigate the thread granularity in more details using three different parallel Haskells implementations GpH-GUM, Eden, and GpH-SMP.

4.2.3 Thread Granularity of Parallel Haskells

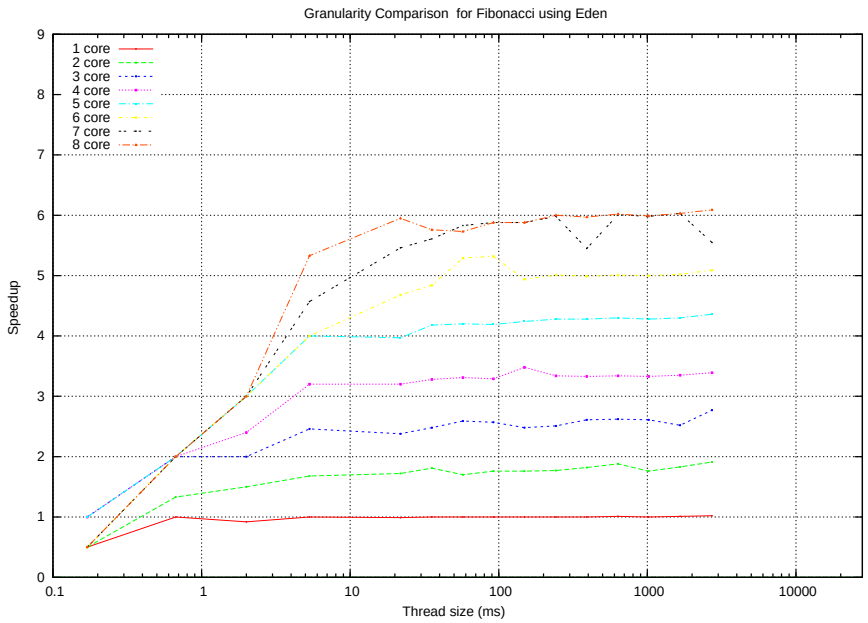
We construct two programs to determine the most profitable thread granularity on a multicore architecture. The `nfibList` program performs a naïve Fibonacci function over a list of integers. Figure 24 presents `nfibList` code. The program uses a data parallel paradigm which requires communicating just a single integer between threads. For the purposes of the experiment, we start with small numbers that generate fine-grain threads and then we increase the input number gradually to generate bigger thread sizes, attempting to determine the optimal granularity for multicore architecture.

Figures 25(a), 25(b), and 26 compare the absolute speedup curves (i.e. speedup relative to the sequential runtime) for the `nfibList` program with GpH-GUM, Eden, and GpH-SMP. The figures show how speedup varies against thread granularity, as determined by the input parameter. The speedup curves for the implementation show an increase in speedup as the thread granularity increases. However, each system has a different maximum value, where no further speedup can be achieved.

Figure 27 compares the thread granularity threshold of the three parallel approaches to `nfibList`. The threshold values achieved by each system reflect the overheads associated with it. For instance, for GpH-GUM on 8 cores the threshold is 92.67ms but just 35.5ms on 2 cores, for Eden on 8 cores 21.83ms but just 5.3ms on 2 cores, and GHC-SMP on 8 cores the threshold is 12.17ms but



(a) `nfibList` using GUM



(b) `nfibList` using Eden

Figure 25: Thread Granularity vs Speedup Comparison of `nfibList`

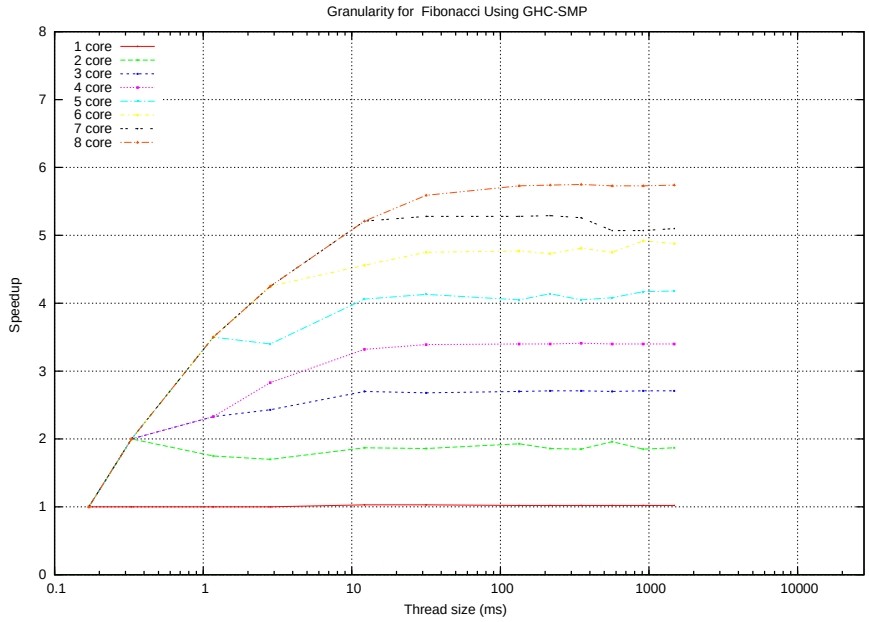


Figure 26: Thread Granularity vs Speedup Comparison of `nfibList`

just 1.17ms on 2 cores (Table 17).

Table 17 analyses the performance of `nfibList` given an input list of length 60 and varying the input parameter of a recursive Fibonacci function (`fib`). The first column shows the number of cores that are involved in the computation; the second and third columns report the threshold in milliseconds and Mcycles for GpH-GUM, Eden and GpH-SMP. The threshold granularity, given in Mcycles, reflects the required thread size independent from the target architecture. The sixth and seventh columns report the execution time for each thread in milliseconds and Mcycles for GHC-SMP. Analysis of the results reported in

Table 17 and Figure 27 show that the minimum thread granularity varies between `35.5ms` on two cores and `92.6ms` on eight cores for GHC-GUM, between

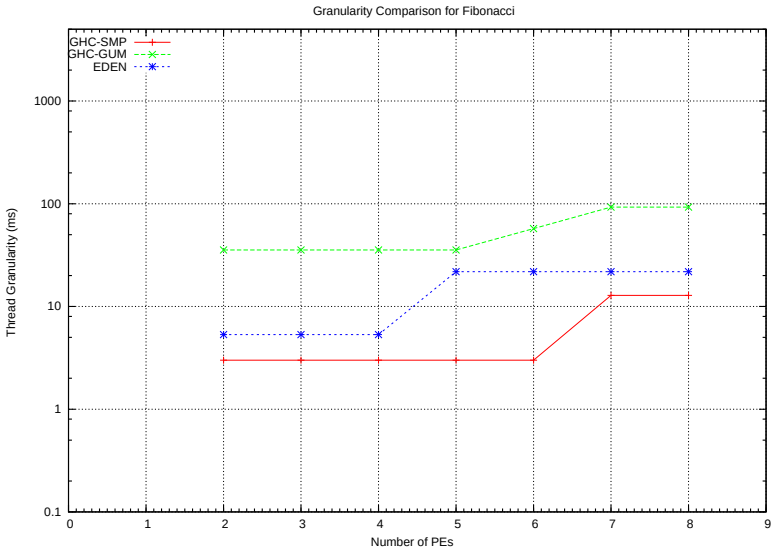


Figure 27: The Most Profitable Thread Granularity Comparison of `nfibList` Program

`5.3ms` on two cores and `21.83ms` on eight cores for Eden, and `1.17ms` on two cores and `12.17ms` on eight cores for GHC-SMP.

We conclude that the most profitable thread granularity required to gain satisfactory performance on similar architecture to the experimental multicore is of the order of `92.6ms (215.92 Mcycles)`, `21.83ms (50.86 Mcycles)`, and `12.17ms (28.36 Mcycles)`, for each implementations respectively. Interestingly, the threshold of 50.86 Mcycle obtained using Eden implementation is higher than 1Mcycle obtained in [3] for the same Eden implementation. We attribute this to two reasons: first, the difference between Eden versions GHC-6.8 and GHC-6.12. Second, in [3], we believe that bigger thread sizes evolve in the computation rather than the 1Mcycle thread.

	GHC-GUM		Eden		GHC-SMP	
NO-of-PEs	Thread Gran. (ms)	Thread Gran. (Mcycles)	Thread Gran. (ms)	Thread Gran. (Mcycles)	Thread Gran. (ms)	Thread Gran. (Mcycles)
2	35.50	82.72	5.33	12.42	1.17	2.73
3	35.50	82.72	5.33	12.42	2.83	6.59
4	35.50	82.72	5.33	12.42	2.83	6.59
5	35.50	82.72	5.33	12.42	12.17	28.36
6	57.33	133.58	21.83	50.86	12.17	28.36
7	92.67	215.92	21.83	50.86	12.17	28.36
8	92.67	215.92	21.83	50.86	12.17	28.36

Table 17: The Most Profitable Thread Granularities for the `nfibList` Program

Programs communicating more data per thread may require coarser granularities to offset the communication time. We therefore construct `sumEulerList` program involving more communication per thread than the `nfibList` program.

```
                ....
                ....
                xs = take l  (repeat  n)
                xss = map gList  xs

gList :: Int -> [Int]
gList  x = [1..x]

foo   [] = []
foo  xs =(map sumTotient  xs) `using` parList  rdeepseq

sumTotient :: [Int] -> Int
sumTotient  xs = sum (map euler xs)
```

Figure 28: `sumEulerList` Program

Figure 28 shows the main functions of the `sumEulerList` program, and it works as follows. The program takes two input arguments: the first parameter represents the number `n` we need to calculate its sum Euler value. The second parameter represents the length of the list `l`. The `sumEulerList` program generates a list of length `l` from the `n` parameter. The generated list is bound to a variable

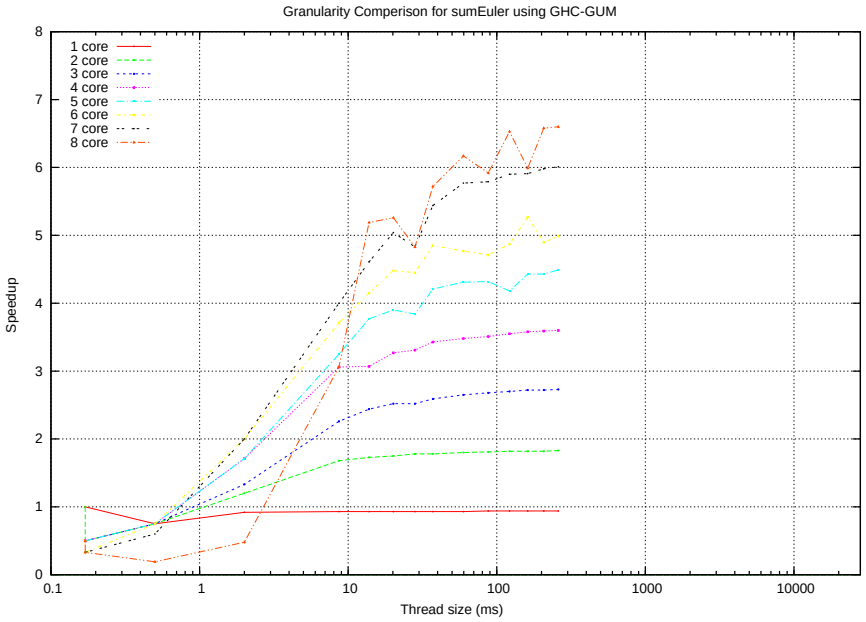
	GpH-GUM		Eden		GpH-SMP	
NO-of-PEs	Thread Gran. (ms)	Thread Gran. (Mcycles)	Thread Gran. (ms)	Thread Gran. (Mcycles)	Thread Gran. (ms)	Thread Gran. (Mcycles)
2	8.67	20.20	15	34.95	2.17	5.06
3	13.83	32.22	15	34.95	2.17	5.06
4	20.17	47.00	15	34.95	9.33	21.74
5	37.17	86.61	22	51.26	9.33	21.74
6	37.17	86.61	22	51.26	9.33	21.74
7	37.17	86.61	30.67	71.46	9.33	21.74
8	87.83	204.64	30.67	71.46	15.17	35.35

Table 18: The Most Profitable Thread Granularity of the `sumEulerList` Program

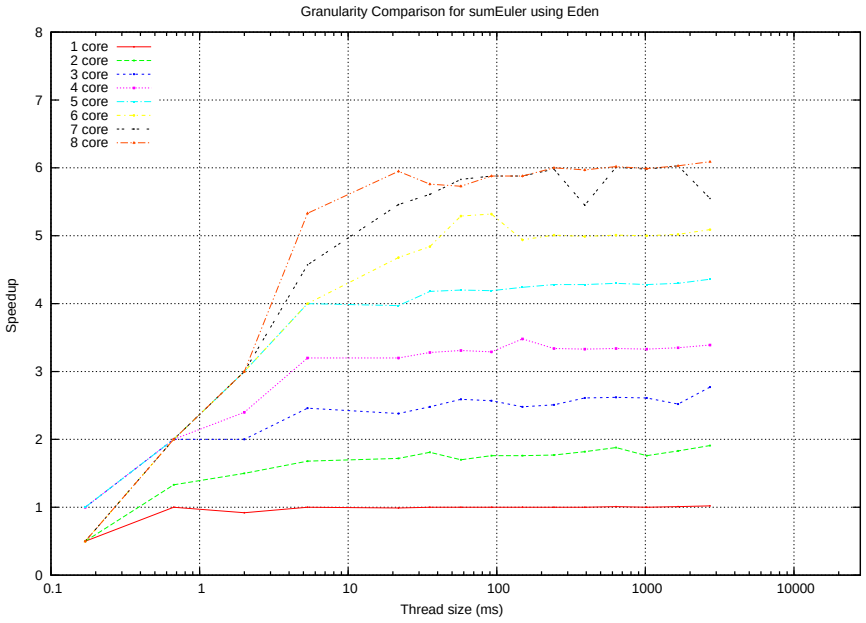
`xs`. Next the `gList` is mapped over the `xs` list and bound to a variable `xss`. In the final stage the `foo` function shown in Figure 28 maps `sumTotient` over `xss` list in parallel, using `parList` function. The definition of the `euler` function is given in Appendix A. We use a parameter `n` to control the granularity of threads that are created, by varying the size of the list that is evaluated in each single data parallel task.

Figures 29(a), 29(b), and 30 shows speedup graphs for the `sumEulerList` program, plotted against various thread granularity sizes. We can see that speedups for `sumEulerList` are similar to speedups for the `nfibList` program. The speedup increases as the thread granularity increases. As expected, the `sumEulerList` program requires bigger thread granularity to achieve maximum speedup.

Figure 31 compares the thread granularity thresholding of the three parallel approaches of `sumEulerList`. The threshold values achieved by each system reflects the overheads associated with it. For GpH-GUM on 8 cores the threshold is 87.83ms but just 8.67ms on 2 cores, Eden on 8 cores 30.67ms but just 15.0ms



(a) sumEulerList using GUM



(b) sumEulerList using Eden

Figure 29: Thread Granularity vs Speedup Comparison of GpH-GUM and Eden Implementations of `sumEulerList` Program

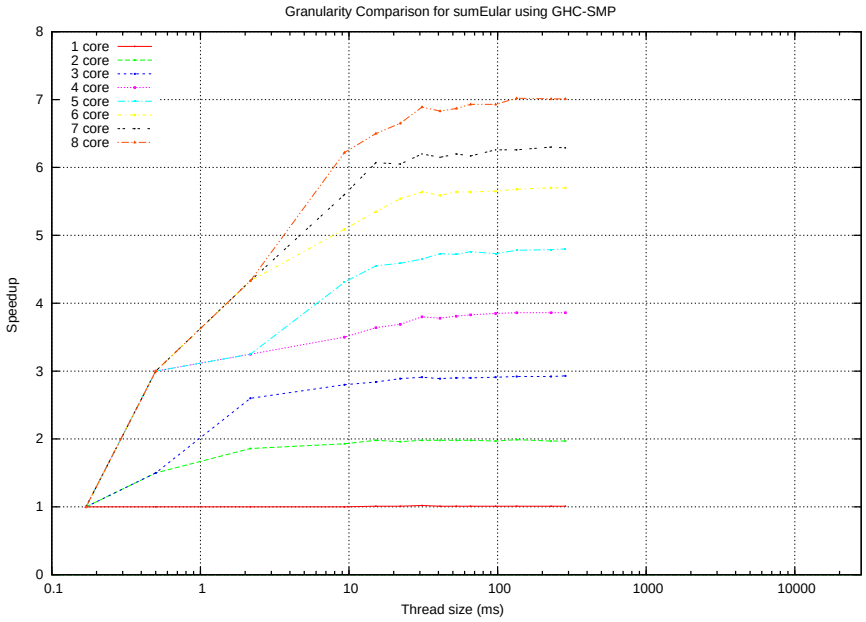


Figure 30: Thread Granularity vs Speedup Comparison of GpH-SMP Implementation of `sumEulerList` program

on 2 cores, and GHC-SMP on 8 cores the threshold is 15.17ms but just 2.17ms on 2 cores (Table 18).

Translating the results to Mcycles to achieve good parallel performance, GpH-GUM requires 204.64 Mcycles, Eden requires 71.46 Mcycle, and GpH-SMP requires 35.35 Mcycles. The results show that threshold granularity of 71.46 Mcycles is in agreement with the [3] result using the Eden implementation.

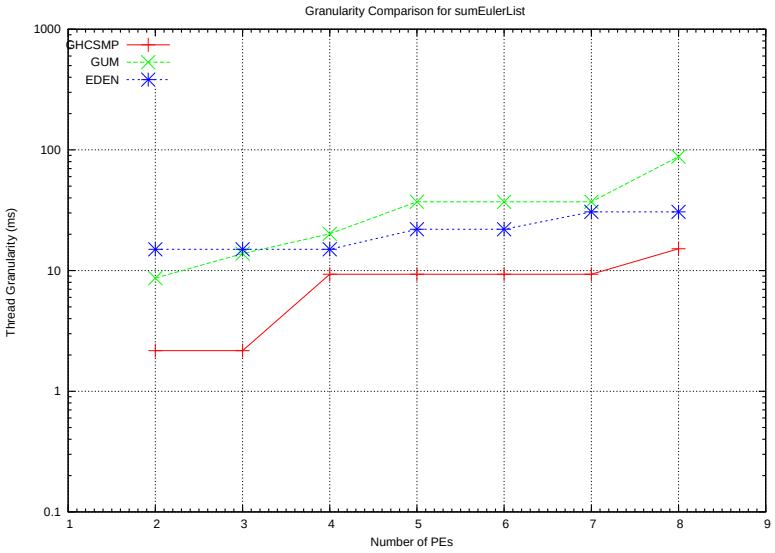


Figure 31: The Most Protable Thread Granularity of `sumEulerList` program

4.2.4 Discussion

Thread granularity information is important to gain better performance for a parallel application. We have performed a study on a multicore architecture using two different data parallel programs. We have compared three different GHC parallel implementations, GHC-GUM, Eden and GHC-SMP. The difference is in the technique that each implementation uses to communicate between participating PEs. GHC-SMP is a purely shared memory module, which does not perform any message passing other than to exchange pointers. GHC-GUM uses message passing to communicate between PEs as well as to maintain a shared heap between them. Eden uses message passing to communicate between PEs through an explicit channel. We make the following observations.

- The most profitable thread granularity rises as the number of cores rises. As the number of cores increases, more overhead occurs and more work is needed. One of the reasons for this is the increase number of messages needed, (i.e. GHC-GUM requires sending 213 messages on two cores compared with 953 messages on eight cores, for the same input). The number of messages rises by a factor of 4.5 (953/213) between 1 and 8 cores. Another reason may be from the costs of maintaining the shared heap.

- GpH-GUM requires larger thread granularity, since it needs to maintain a virtual shared heap along with communicating data. Figures 27 and 31 show that GHC-GUM requires up to 92.67ms for the `nfibList` program and 87.83ms for the `sumEulerList` program on eight cores. We believe this is for two reasons, first GpH-GUM maintains a shared heap between PEs in addition to messages passing overhead; second GHC-GUM needs extra communications to indicate which processor can be used, i.e. the time spent by an idle PE looking for work.

- GpH-SMP requires the smallest thread granularity because it does not need to communicate data between PEs. Instead, GpH-SMP exchanges pointers between PEs.

4.3 Summary

In this chapter we have explained the philosophy behind the evaluation strategies model. We have discussed the redesign of the evaluation strategies module and assessed the new strategies with a very carefully selected number of parallel benchmarks, covering a data parallel paradigm and divide-and-conquer parallel paradigm. However it has some minor drawbacks: being relatively complex, providing relatively weak type safety, and requiring care to express control parallelism, the advantages are many and substantial: It provides clear, generic, and efficient specification of parallelism with low runtime overheads. It resolves a subtle space management issue associated with parallelism, better supports speculation, and is able to directly express parallelism embedded within lazy data structures.

We have investigated the most profitable thread granularity for parallel functional languages on multicore architectures. From the limit study, we believe that for a program similar to `nfibList` benchmark and executed on similar architecture, the most profitable thread granularity for the three implementations GHC-GUM, Eden, and GHC-SMP are in the range of 200, 50, and 30 megacycles respectively. Most parallel programs communicate far more data than such ideal programs and greater thread granularity is required to offset the communication time. Measurement of one such program, `sumEulerList`, on the three implementations suggests a thread granularity of 200, 70, and 35 megacycles respectively.

Chapter 5

Architecture-Aware Constructs

Increasingly, physical limitations are forcing hardware designers toward developing multicore architectures, which means parallel hardware is coming to commodity hardware. Therefore, general purpose parallel programming that maximises application performance is essential. Many existing parallel programming languages target scientific applications; however only a few languages are targeting general-purpose parallel programming, which must be the mainstream [40].

This chapter describes the challenges involved in designing architecture-aware constructs for the GpH language in particular, that will *exploit information about task size* and aims to preserve data locality, or to distribute large units of work, thereby reducing communication for small tasks. The architecture-aware constructs need to provide mechanisms for multiple levels of parallelism because the new architectures provide parallelism at multiple levels to maximise performance [55].

We have chosen the non-strict functional language GpH to implement architecture-aware constructs, because we believe that architecture-aware constructs are easier to implement in a functional language. The non-strict semantics (laziness) of GpH means that an expression is evaluated only when its result is needed. GpH takes advantage of this flexibility by annotating expressions that can be evaluated in parallel. Moreover, a functional language requires minor modifications to the sequential program to exploit parallelism. However, the constructs could be implemented in any language that has process spawning.

Section 5.1 outlines the trend towards hierarchical architectures and illustrates the view of this architecture as a virtual architecture, using the new constructs. Section 5.2 describes how the constructs preserve the data locality and compares this approach with other approaches. Section 5.3, we discusses the design issues of the new constructs and their features. Section 5.4 defines the semantics of the new constructs. Section 5.5 demonstrates the implementation of the constructs as an extension of the GUM runtime features. Section 5.6 reports the performance results of the architecture-aware constructs compared with the `par` construct. Section 5.7 describes how the constructs could be implemented in other languages such as Erlang.

5.1 The Trend Towards Hierarchical Architectures

Physical limitations and manufacturing technologies are driving general purpose computing architectures inexorably towards many cores, with the number of cores following Moore's Law. It is widely anticipated that future architectures will both evolve quickly, and have hierarchical communications structures [81]. The number of cores will steadily increase as will the level of heterogeneity. Already the most common parallel architectures are clusters of multicore nodes, with 3 levels in the hierarchy: cores, chips and nodes. Threads on the same core can communicate most quickly with a thread on the same core, more slowly with a thread on another core in the node, and slower still with threads on remote nodes, as discussed in Section 2.1.4. The communication hierarchy is likely to become deeper as the number of cores increases. For example, the number of cores sharing the same memory is likely to be restricted, and hence many core architectures may introduce another level within a node. Figure 32 illustrates both a virtual hierarchical architecture, and the real hierarchical architecture. Only Heriot Watt machines from Figure 32 are used for measurements in this research. The virtual architecture comprises a tree, possibly unbalanced, and where the degree of the nodes may vary. We expect communication at lower levels in the tree to be faster than at higher levels. Of course, not only the physical distance is a factor in clustering processor elements (PEs). Communication latency and CPU speed are

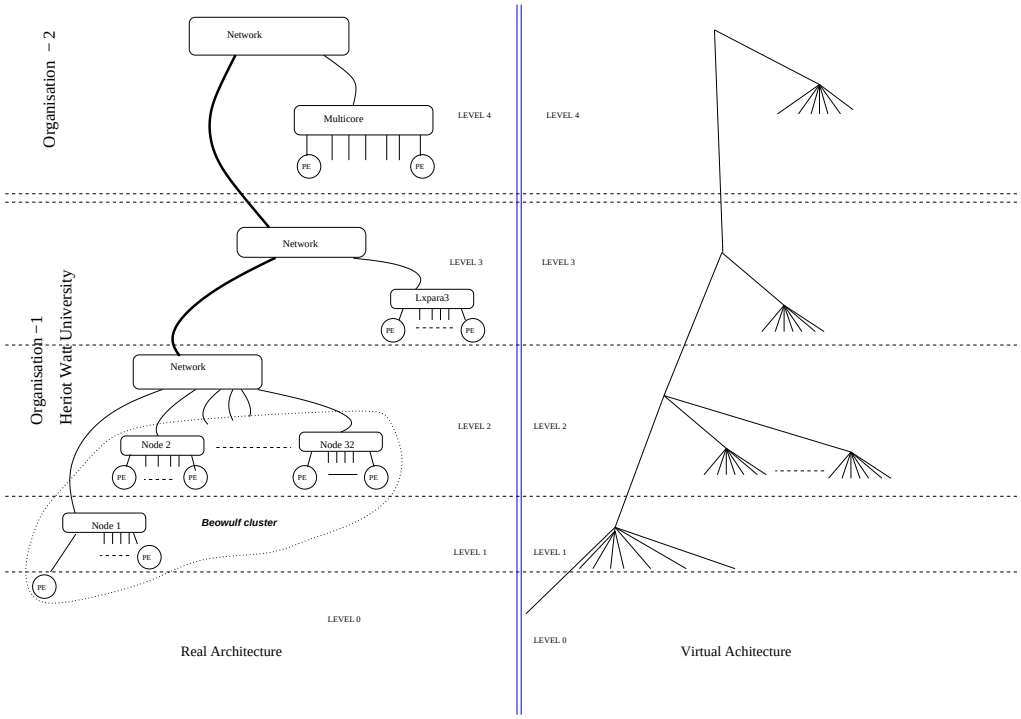


Figure 32: Real and Virtual Hierarchical Architectures

other factors that can be considered in clustering processors. For example, if we have two processors physically located far from each other, but connected with exceptionally fast network facilities, they may be considered to be as in the same level.

5.2 Other Architecture-Aware Languages

Shared memory architectures are sufficiently simple that parallel Haskell implementations do not need architecture awareness to preserve data locality, and the current multicore support in GHC does provide it [80].

The Eden distributed memory parallel Haskell has no architecture awareness. Processes are randomly placed by the runtime system [14]. Other distributed

memory parallel Haskells, like Cloud Haskell, explicitly place a process at named location (node) [38]. Architecture-aware GpH occupies a middle ground between explicit placement and implicit placement by specifying placement abstractly.

Outside the functional language community, some recent parallel languages are architecture-aware. For example X10 [30] improves data locality by integrating new constructs: (activity, places, regions and distributions), into hierarchical parallel model and nonuniform data access. X10 splits the memory space into parts known as a partitioned global address space. Constructs enable programmers to assign a single `place` to each global address space. During execution of an `activity`, it will be located at the same partitioned global address of its `place`. The X10 language constructs are more explicit than our new notion of abstract communication levels.

Other parallel languages use similar architecture-aware approaches to ours is the Scalable Locality-aware Adaptive Work-stealing Scheduler (SLAW)[43]. Workers in SLAW execute spawned tasks eagerly and leave the continuation to be stolen. The placement of spawned tasks is done by the runtime system. It has the notion that the worker can only steal work from workers which appear in the same place.

5.3 New Architecture-Aware Constructs

This section describes the key features for architecture-aware constructs. We describe the virtual architecture and how it is related to parallelism, and discuss the challenges required for scheduling and placing parallel tasks in a parallel platform. Finally, we describe the basic operational concepts of constructs.

5.3.1 Virtual Architectures

Many applications can exhibit parallelism at multiple levels with different granularities, which reflects the modern parallel clusters of multicore architecture [40]. The challenges are to develop a parallel programming model that allows the programmer to represent parallelism in a multi level style. We believe that a fully implicit approach will not deliver an acceptable performance on hierarchical architectures because for some problems, e.g. regular problems like many matrix manipulations, optimal performance can only be obtained on a specific architecture by explicitly placing threads within the architecture. However, many problems do not exhibit this regularity. Moreover, explicit placement prevents *performance portability*: the program must be rewritten for a new architecture, a crucial deficiency in the presence of fast-evolving architectures. To avoid these deficiencies we, like others, propose language constructs that expose a virtual

architecture rather than the actual architecture. Clearly the virtual architecture must be readily mapped to physical architectures. In addition, our constructs minimise prescription: they identify sets of locations where the thread may be placed. Moreover, we support *performance portability* by isolating the architecture-specific parts of the program in just a few functions that can be re-factored for a new architecture.

5.3.2 Placing Task on Hierarchical Architecture

A mapping means the process of assigning and scheduling the abstract representation of tasks onto physical resources, determining where and when each task executes. The existing assigning and scheduling mechanism of GUM that we use for implementing architecture-aware constructs has been described earlier in Section 2.3. The thread management of the model is responsible for deciding when to generate a new thread and how to schedule the threads. However, this management requires some enhancement in order to perform well on a hierarchical architecture.

Broadly speaking there are two challenges to be solved simultaneously, when controlling parallelism on a hierarchical architecture:

- **Limit the communication costs for small computations**. This entails limiting how far small computations are communicated, and requires information about thread granularity (i.e. execution time). Without this

information, programs often have poor resource utilisation, as the system is saturated with small threads [44]. Thread granularity information may be obtained from a number of sources, for example, from some resource analysis, or by profiling, or by the programmer. There are many models which use the resource analysis techniques to achieve better performance on a parallel platform. Most models focus on the level of abstraction over the hardware that is provided and they are classified by the level of machine abstraction. For example, the Parallel Memory Hierarchy (PMH) model developed by Alpern et al., uses a single mechanism to model the costs of both interprocessor communication and the memory hierarchy [4]. The physical platform is modeled as a tree of memory modules with processors at the leaves. The PMH program can be viewed as a collection of modules. Each module `m` has four parameters: the `blocksize`, which tells how many bytes there are per block; `blockcount`, which tells how many blocks fit in `m`; the `childcount`, which tells how many children there are in `m`; and the transfer `time`, which tells how many cycles it takes to transfer a block between `m` and its parent.

Another multi-level parallel programming model example, is the bulk synchronous parallel model (BSP) proposed by Valiant in 1989 [19]. The BSP model consists of a collection of processors, each with private memory, and a communication network that allows processors to access memories of other

processors. If a processor reads or writes from its private memory the operation is relatively fast. If it reads or writes from a remote memory, a message must be sent through the communication network, and this operation is slower. The BSP computation proceeds in a series of `supersteps` comprising three stages. The independent concurrent computation step on each processor using only local values. The communication step in which each processor exchanges data with every other processor. The barrier synchronisation step where all processes wait until all other processes have finished their communication actions.

Bischof et al. in [18] have presented a multi-level parallel programming model. This model is a cost-optimal implementation of the divide-and-conquer skeleton. The model consists of a number of conventional lists, called segments. These segments have different costs. The cost depend on the underlying architecture.

We do not address the problem of obtaining this information here. Although the examples throughout this thesis use granularity information from program parameters, we have adapted the runtime system to store granularity information with each spark, if required.

- **Keep all cores busy**: for example, at system start up we must quickly distribute work to all cores. This entails sending large grain computations long distances over the communications hierarchy.

5.3.3 New Constructs

```
parDist    :: Int -> Int -> a -> b -> b
parBound   :: Int -> a -> b -> b
parAtLeast :: Int -> a -> b -> b
parExact   :: Int -> a -> b -> b
```

Figure 33: New Architecture Aware Constructs

The new architecture-aware constructs are summarised in Figure 33. The constructs identify sets of locations where a computation may be performed, and the runtime system is free to place the computation within this set. These sets often include multiple levels in the communication hierarchy. The `parDist` primitive is implemented in the GUM runtime system. The `parDist` primitive introduces a potential for parallel evaluation of its third argument. The first two arguments specify a minimal and maximal distance for the placement of the parallel evaluation. The value of the expression is its fourth argument. The distance is the shortest path between two nodes. The exact semantics of `parDist`, in terms of the set of possible locations is defined in Section 5.4. All remaining constructs are implemented using the `parDist` primitive. To limit the communication costs for small computations, or to preserve data locality, we propose `parBound` that behaves like `par`, except that it takes an additional integer parameter specifying the *maximum* distance in the communication hierarchy that the computation may be located. The distance represents a level in the hierarchy illustrated in Figure 34.

`parBound` illustrates a key characteristic of our constructs, namely that while

placing restrictions on how work is communicated, they aim for minimal prescription. That is, the constructs identify sets of locations where a computation may be performed, and the runtime system is free to place the computation within this set. These sets often include multiple levels in the communication hierarchy. For example a computation sparked by a `parBound 1` may be placed on any core in the shared memory node, and a computation sparked by a `parBound 2` may be placed on any core in the set identified in Figure 34.

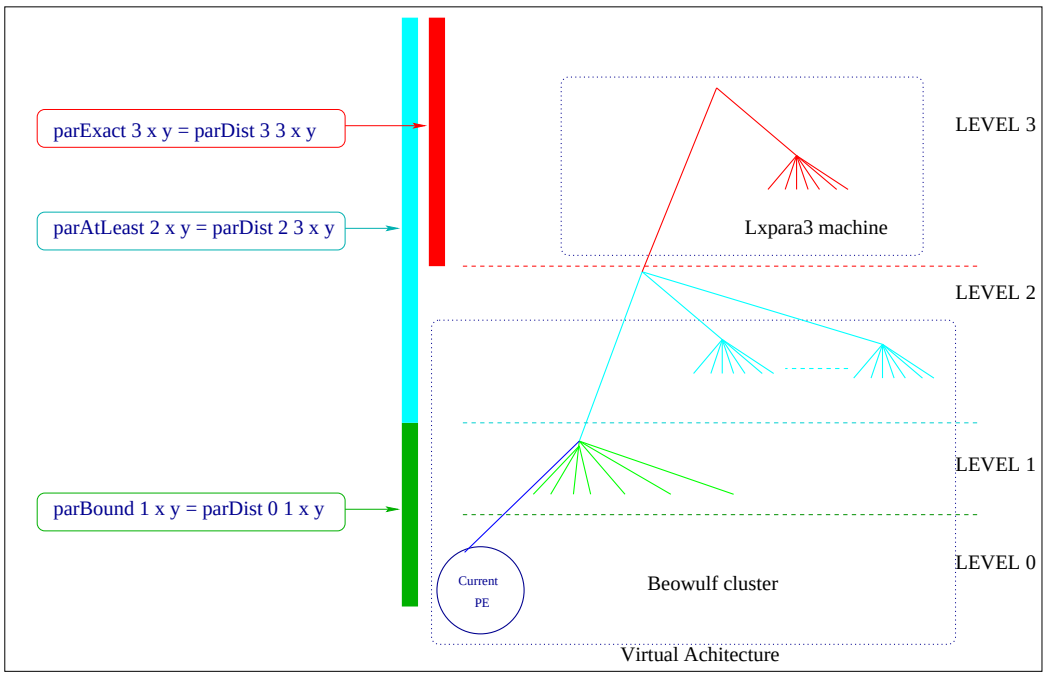


Figure 34: Using New Architecture Aware Constructs

`parAtLeast` is a dual to `parBound`, as it takes an additional integer parameter specifying the *minimum* distance in the communication hierarchy that the computation may be communicated. The idea is to communicate large-grain computations over large distances, for example to evenly distribute work across the machine at the beginning of the execution. On the example architecture

```
parBound:: Int -> a -> b -> b
parBound n = parDist  0 n x y

parAtLeast:: Int -> a -> b -> b
parAtLeast n = parDist  n  maxLevel  x y

parExact :: Int -> a -> b -> b
parExact n = parDist  n n  x y
```

Figure 35: Architecture Aware Construct Definitions

shown in Figure 34, `parAtLeast 3` means that the computation must be sent the greatest distance in the communications hierarchy, i.e. to the `lxpara3` machine. `parAtLeast 2` means that the computation must be communicated at least to another node in the Beowulf cluster, `parAtLeast 1` means that the computation must be communicated at least to another core within a shared memory node, and `parAtLeast 0` means that it may be communicated freely to any core in the machine.

`parDist` can also be used to define other constructs. By way of illustration, `parExact` in Figure 33 specifies an exact level, although not a specific processor. A `parDist` can also be parameterised to capture other notions, for example `parDist (n-1) n` specifies that a spark may be communicated to a core residing at either level `(n-1)` or level `n`, and `parDist (n-2) n` is similar. This provides more flexibility as there is a bigger set of locations that can fish the spark away.

Table 19 summarises the new constructs in comparison with `par`. Thread creation remains optional for all constructs; likewise, no specific location is identified by any of the constructs. The constructs may, or may not, restrict the placement

Construct	`par` Existing	`parBound` New	`parAtLeast` New	`parDist` New	`parExact` New
`Thread Creation`	Optional	Optional	Optional	Optional	Optional
`Placement`	Not Restricted	Not Restricted	Not Restricted	Not Not Restricted	Not Restricted
`Hierarchy Placement`	Not Restricted	Restricted	Restricted	Restricted	Specific
`Work Distribution`	Passive	Passive	Passive	Passive	Passive

Table 19: GpH `par` Construct Comparison (Increasingly Specific)

within the communication hierarchy, and `parExact` identifies a specific hierarchy level. Work distribution for all constructs is dynamic and passive, that is, idle cores seek work, and select only sparks for the appropriate communication level.

5.4 The Semantics of Constructs

The new constructs are not prescriptive: rather than specifying a single PE for a task, they identify sets of PEs within the communication hierarchy of the architecture. We present a simple Haskell specification of the sets of PEs that each construct identifies when executed on any PE of participating PEs. The complete Haskell program specifying the constructs, including all auxiliary functions, can be found in Appendix B. We first need some mechanism specifying paths and distances in the tree hierarchy.

5.4.1 Distance Function

In order to illustrate how the `distance` function is working, we define a binary tree `(Tree t)` structure representing an underlying parallel platform (e.g. the one shown in Figure 36). The definition of the tree data structure is as follows:

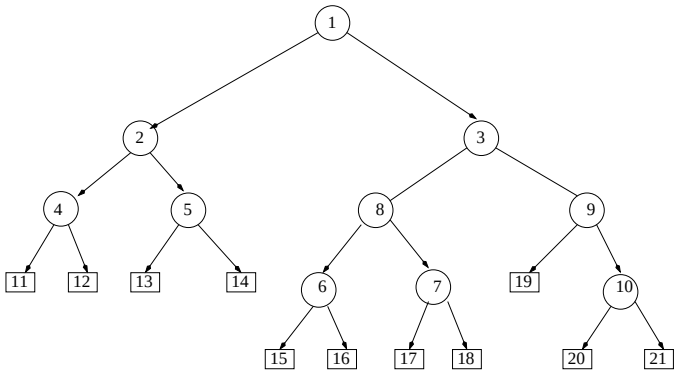


Figure 36: An Example of Hierarchical Architecture.

```
data Tree a =  Node a  (Tree a)  (Tree a)

               | Leaf {pId ::Int} deriving (Eq, Show)
```

A tree is a leaf with a PE Id as value, or a node represents network possibly parametrised with information such as latency, leaves represents PEs. A node has a value and two branches, each of which is a subtree.

The function `distance t p1 p2` calculates the distance, defined as the number of steps to the nearest common node in the hierarchy between two leaves in the architecture hierarchy. The function takes a tree `t` representing the architecture and two leaves `p1` and `p2` as input and returns the distance between the two leaves as an integer.

The definitation of the distance function uses an additional auxiliary functions, `path` shown in Figure 37. The call `path (Node v t u ) p` calculates the path to leaf `p` from the root of the tree represented as a list. It takes a tree `(Node v t u )` and leaf `p` and returns list of nodes that lead to the leaf `p`. The complete Haskell program defining all auxiliary functions, e.g. `prefixOf` function can be

```
distance :: Tree Int -> Int -> Int -> Int
distance  t  p1 p2  =  d1+d2
         where
          pathTop1 = path t p1
          pathTop2 = path t p2
          comNodes =    length (prefixOf pathTop1 pathTop2)
          d1 = length pathTop1 - comNodes
          d2 = length pathTop2 - comNodes


path ::  Tree Int -> Int -> [Int]
path (Leaf p ) s
      | p==s = [s]
      | otherwise = []
path  (Node v t u) s
      | v == s =  [v]
      | left== [] && right  == [] = []
      | left /= [] =   [v]  ++ left
      | right  /= [] = [v] ++ right
      where
        left = path t s
        right = path u s
```

Figure 37: Distance Function

found in Appendix B. For simplicity, we use a tree of integers shown in Figure 36 to demonstrate the constructs semantics. Squares in the tree represent PEs in the hierarchy (*Leaf*). Circles in the tree represent the networks connecting Pes or sub-networks (*Node*).

Aa an example, if we need to compute a distance between Leaf 15 and Leaf 20, we proceed by the following steps.

1. Path to Leaf 15 $\Longrightarrow$ `pathTop1` = [1,3,8,6,15]

2. Path to Leaf 20 $\Longrightarrow$ `pathTop2` =[1,3,9,10,20]

3. Length of the longest common prefix of `pathTop1` and `pathTop2` $\Longrightarrow$ comNodes = length [1,3] =2

   Node 3 is the nearest common node between Leaf 15 and Node 20

4. Length of path to Leaf 15 from nearest common Node $\Longrightarrow$ d1 = `length (pathTop1) - length (comNodes)` = 3

5. Length of path to Leaf 20 from nearest common Node $\Longrightarrow$ d2 = `length(pathTop2) - length(comNodes)` = 3

6. The distance between Leaf 15 and Leaf 20 = d1 + d2 = 6, is the sum of steps moving from one leaf to the nearest common parent and down to other leaf.

5.4.2 setparDist Function

The most basic primitive we propose is the `parDist` primitive. We therefore start by defining its semantics in terms of the possible locations defiened by it. A `setparDist t m u p` specifies the set of PEs on which a `parDist m u` task may be executed from PE_p in an architecture `t`. It takes an architecture tree `t`, a minimum bound `m`, maximum bound `u`, and a leaf `p`, the current location, as input and returns a list of all possible PEs (Figure 38). For example, if we need to generate sparks intended to be executed between levels 1 and 3 from Leaf 20 of the tree shown in Figure 36, we perform the following:

1. Calculate path to Leaf 20 $\Longrightarrow$ `pathTop` = [1,3,9,10,20].

2. Calculate the common node distant by `u` levels from Leaf 20 $\Longrightarrow$ `commonu` = last (take (5 - 3) [1,3,9,10,20]) $\Longrightarrow$ 3.

3. Calculate the common node that is `m` levels from Leaf 20 $\Longrightarrow$ `commonm`= last (take (5 - 1) [1,3,9,10,20]) $\Longrightarrow$ 10.

4. Calculate the subtree of the `commonu` leaf 3 $\Longrightarrow$ `subtreeu` = (Node 3 (Node 8 (Node 6 (Leaf pId = 15) (Leaf pId = 16)) (Node 7 (Leaf pId = 17) (Leaf pId = 18))) (Node 9 (Leaf pId = 19) (Node 10 (Leaf pId = 20) (Leaf pId = 21))))

5. Calculate the complementary tree of the `common` node 10. The complementary tree is the original tree excluding the subtree of a given node.

```
setparDist :: Tree Int -> Int -> Int -> Int -> [Int]
setparDist t m u p
       | ((m<0) || (u<0)) = []
       | ((m==0) && (u==0))= [p]
       | (u > (length (pp)-1)&& (m==0))  = (rLeaf ( t))
       | ((m==u) || (u > (length (pp)-1))) = exact ( subexact) p
       | m==0 = [p]++setPes
       | otherwise = setPes
      where
       pp = path  t p
       commonnu = last (take (length (pp) - u) pp)
       commonnm =last ( take (length (pp) - m) pp)
       subu=subTree t commonnu
       subexact= subTree t commonnm
       complementtree =   (complementTree subu commonnm)
       setPes= filter (/= commonnm) (rLeaf (complementtree))


subTree ::Tree Int -> Int -> ( Tree Int)
subTree (Leaf  p) s =EmptyTree
subTree (Node v t u) s
    | v == s  = (Node v t u)
    | left== EmptyTree && right == EmptyTree = EmptyTree
    | left /= EmptyTree = left
    | right  /=EmptyTree =right
    where
     left =    subTree t s
     right =   subTree u s

complementTree ::  Tree Int -> Int -> Tree Int
complementTree (Leaf p1) s = (Leaf p1)
complementTree  (Node v l EmptyTree ) s
       | v==s = EmptyTree
       | otherwise  = (Node v (complementTree l s) EmptyTree)
complementTree (Node v t u) s
       | v==s = EmptyTree
       | otherwise  = (Node v
                      (complementTree t s)
                      (complementTree u s))
```

Figure 38: `setparDist` Locations Function

```
setparBound ::Tree a ->Int ->  a -> [a]
setparBound t n p = setparDist t 0 n p
```

Figure 39: `setparBound` Locations Function

In our case, we calculate the complement for `subtreeu` of node 10. $\Longrightarrow$ `complementtree` = Node 3 (Node 8 (Node 6 (Leaf pId = 15) (Leaf pId = 16)) (Node 7 (Leaf pId = 17) (Leaf pId = 18))) (Node 9 (Leaf pId = 19))

6. Finally calculate the leaves of the `complementtree` subtree $\Longrightarrow$ setPes = [15,16,17,18,19].

5.4.3 setparBound Function

As mentioned in the previous section, `parDist` is the most basic primitive. We can use it to define the other constructs. A `setparBound t n p` (Figure 39) specifies the set of PEs that tasks generated by a `parBound n` may be executed on, from PE_p in architecture `t`. For example if we need to generate sparks bounded by two levels from leaf 11 of the tree shown in Figure 36, we just call `setparDist` with the following parameters `t 0 2 11`, where `t` is the tree. The result is [11,12,13,14], a list of leaves with a distance of at most 2 in the architecture tree `(t)`.

5.4.4 setparAtLeast Function

A `parAtLeast` is similar to `parBound`, as it takes an additional integer parameter specifying the *minimum* distance in the communication hierarchy that the

```
setparAtLeast::(Ord a, Show a) =>Tree a ->Int ->  a -> [a]
setparAtLeast  t n p  = setdFun t n maxLevel p
           where
               maxLevel = 3
```

Figure 40: `setparAtLeast` Locations Function

computation may be communicated. Therefore, it can be defined in a similar way, as shown in Figure 40. So, if we need to generate sparks intended to be executed at least two levels from leaf 11 of the tree shown in Figure 36, we just call `setparDist t 2 maxLevel 11`, where `t` is the tree and `maxLevel` is the maximum distance that sparks can be sent within the architecture hierarchy. In this example, the result is [15,16,17,18,19,20,21].

5.4.5 Construct Properties Test

This section presents implementation-relevant properties that the architecture-aware semantics should satisfy. These properties are expressed as boolean functions in Haskell and validated using QuickCheck [31], that is the properties are written as Haskell functions and can be automatically checked on either random input or with custom test data generators. Two types of properties are tested: the basic properties and specialised properties. The complete Haskell program specifying the properties can be found in Appendix B.

5.4.5.1 Basic Properties

Let $\mathcal{P}$ be the set of PEs which are the leaves of the tree representing a given hierarchical architecture. The domain $\mathcal{H}$ is the domain of all possible tree hierarchies. The domain $\mathcal{P}$ is the domain of all possible processor elements.

1. **Basic property one.** For any processing element $p \in \mathcal{P}$, the only possible placement of a bounded spark with upper and lower bounds of 0 and 0 is the p itself. Formally, this is written as:

   $\forall h \in \mathcal{H}, \forall p \in \mathcal{P}. \qquad setparDist\ h\ 0\ 0\ p = \{p\}$

2. **Basic property two.** Let `path p` be a function that returns the longest path to the PE from the root of the tree hierarchy. Let `rLeaf h` be a function that returns all processor elements (PEs) in the tree hierarchy. The `path` and `rLeaf` functions are described in Section 5.4.1. For $p \in \mathcal{P}$ and $h \in \mathcal{H}$, the set of PEs returned by calling the `setparDist h 0 (length (path h p)) p` function is equal to the set of all PEs in the hierarchy, as returned by `rLeaf h`.

   $\forall h \in \mathcal{H}, \forall p \in \mathcal{P}. \qquad setparDist\ h\ 0\ (\ length\ (path\ h\ p))\ p = rLeaf\ h$

3. **Basic property three.** If the upper bound `u` is less than 0 or lower bound `m` is greater than the longest path to the PE from root of the tree hierarchy then the set of PEs returned by calling `setparDist h m u p` function is an

empty set of PEs.

$\forall h \in \mathcal{H}, \forall p \in \mathcal{P},\ m \in \mathbb{Z},\ u \in \mathbb{Z}.$

$((u < 0) \,||\, m > (\ length\ (path\ h\ p))) \Rightarrow setparDist\ h\ m\ u\ p = \{\ \}$

The above three basic properties can be considered sanity checks of the semantics. All basic properties have passed `Quickcheck` testing using one hundred randomly generated inputs, each with a randomly generated tree hierarchy.

5.4.5.2 Specialised Properties

The proposed architecture-aware model exposes the tree hierarchy to the programmer through the `parDist` primitive. The `parDist` primitive provides a mechanism to spark tasks that can be executed in certain levels of the architecture hierarchy. We believe, for the implementation, it is important that these sparks do not leave their neighbourhood, where *neighbourhood* is the set of PEs specified by `parDist` primitive. Otherwise the bounded spark may *diffuse* to arbitrary locations after several steps of fishing (workstealing, as outlined in Section 3.3.3). We define the following property to formally specify and check this property.

1. **Specialised proposed property one.** This property reflects the initial intention of the bounded `parDist`. For $p \in \mathcal{P}$ and $h \in \mathcal{H}$, the set of PEs returned by `setparDist h 0 u p` is equal to the set of PEs returned by `setparDist h 0 u p'`, where `p'` is a possible location after one step of

fishing. The aim is to guarantee that if the spark is fished again from `p'` it will be executed in the same neighbourhood specified by the original `p`.

$\forall h \in \mathcal{H}, \forall p, p' \in \mathcal{P}, u \in \mathbb{Z}\ ,\ p' \in (\ setparDist\ h\ 0\ u\ p) \Rightarrow$

$setparDist\ h\ 0\ u\ p' = setparDist\ h\ 0\ u\ p$

On closer examination, this property fails under the `quickcheck` test. In the case of an unbalanced tree hierarchy, the result of `setparDist h 0 u p` may return a subset of the set returned by `setparDist h 0 u p`.

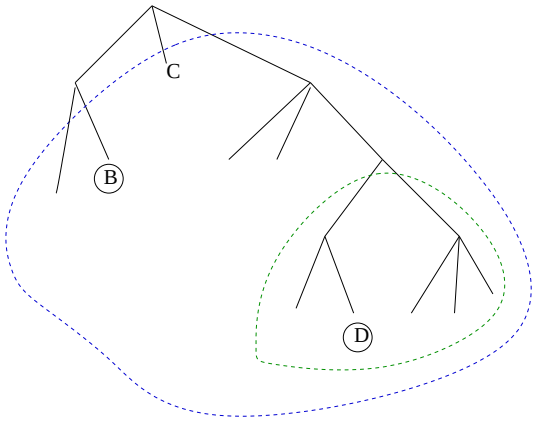


Figure 41: Tree Example of Specialised Proposed Property One

For example, in the tree hierarchy shown in Figure 41, if PE (`B`) launches a spark with boundaries 0 and 2, then any PE in the outer circle can fish the spark. In particular, it can be fished by PE (`D`). In second step the spark can be fished only from PEs in the inner circle. That is why the property fails. However, this is not always true, as illustrated in the next proposed property.

2. **Specialised proposed property two.** For $p \in \mathcal{P}$ and $h \in \mathcal{H}$, the set

of PEs returned by `setparDist h 0 u p'` is a subset of the set of PEs returned by `setparDist h 0 u p`.

$\forall h \in \mathcal{H}, \forall p, p' \in \mathcal{P},\ u \in \mathbb{Z}.\ p' \in (\ setparDist\ h\ 0\ u\ p) \Rightarrow$

$setparDist\ h\ 0\ u\ p' \subseteq setparDist\ h\ 0\ u\ p$

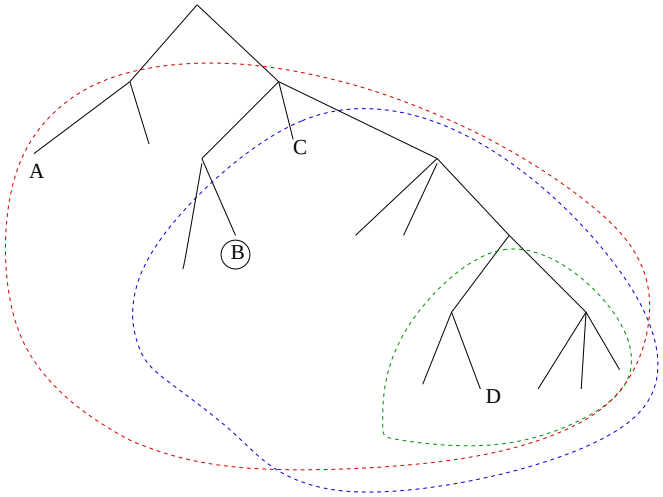


Figure 42: Tree Example of Specialised Proposed Property Two.

The property also fails the `quickCheck` test, because of unbalanced tree hierarchies. In the second step, the spark may be fished by a PE which is not one of the elements of the original PE neighbourhood. For example, in the tree hierarchy shown in Figure 42, if PE (`B`) launches a spark with boundaries 0 and 2, then any PE in the blue circle can fish the spark. In particular, it can be fished by PE (`C`). In a second step, the spark can be fished from PE (`C`) by any PE in the red circle. In particular, it can be fished by PE (`A`), which is outside the original neighbourhood. We call this behaviour of the fishing mechanism *diffusion of sparks*. In the

implementation, we must prevent this scenario from happening. We achieve this by resetting the boundaries of the spark to be 0 and 0, after the first fishing stage. This forces evaluation of the spark on the initial target PE, and thus within the neighbourhood specified by the spark.

3. **Specialised proposed property two.** For $p \in \mathcal{P}$ and $h \in \mathcal{H}$, after fishing a spark from `p` to `p'`, which is within the bound `u` and after resetting the bound `u` for the spark to 1, this spark can only be fished by a p inside the original neighbourhood of `p`. The set of PEs returned by `setparDist h 0 1 p'` is a subset of the set of PEs returned by `setparDist h 0 u p`.

   $\forall h \in \mathcal{H}, \forall p, p' \in \mathcal{P},\ u \in \mathbb{Z},\ p' \in (\ setparDist\ h\ 0\ u\ p) \Rightarrow$

   $setparDist\ h\ 0\ 1\ p' \subseteq setparDist\ h\ 0\ u\ p$

   This property passes the `quickCheck` and guarantees that there is no *diffusion of sparks*, i.e. sparks always remain in the neighbourhood specified by the original `parDist`.

5.4.6 Summary

We have presented the semantics for the architecture aware constructs, specifying the set of possible locations when providing boundaries to the sparks. We have specified the expected behaviour of the constructs by Haskell functions, achieving an executable specification. We have formulated several properties as Haskell predicates and used `Quickcheck` to check them on random input. The three

basic properties represent a sanity checks of the semantics. Two proposed implementation relevant properties did not hold, and counterexamples extracted from `Quickcheck` identified *diffusion of sparks* to be the problem. In the implementation, we avoided this problem by resetting the boundaries after one fishing stage. The final property, checked with `Quickcheck`, shows that with this modification, the desired property holds.

5.5 Implementation of Architecture-Aware Constructs

The implementation of the new constructs requires modifying the GpH language in various places in the runtime system and the GHC compiler.

5.5.1 Runtime Systems Modification

The implementation of architecture-aware constructs requires adapting the GUM runtime system outlined in Section 2.3 to store granularity information with each spark. The modifications involve two kinds of information:

- Static information, including the number of the PE and the distances from the PE to other PEs.
- Dynamic information, including the minimum and maximum distance for each spark.

Each PE maintains static information about other PEs that participate in the computation in a local table that is collected at the beginning of the program execution.

PEId	Speed	Distance	IP address
1	1596	0	137.195.143.104
2	1596	2	137.195.143.101
3	1998	3	137.195.27.241
4	1998	3	137.195.27.241
5	1596	1	137.195.143.104

Table 20: A Static Information Table for Five Cores from Figure 34

Table 20 shows the static information table. A PEId is a unique number generated by the communication library, to distinguish between PEs in the case of sending and receiving messages. CPU speed and IP address are collected using the `pvm_config` function. After collecting the IP addresses, the distance between the current PE and other PEs is calculated.

5.5.2 Work Placement Mechanism

The work placement mechanism in GUM identifies randomly a destination PE to donate work. Original GUM responds with any spark FCFS. This mechanism has been improved by replying to the work request with only work that is worth executing on the requester location.

Figure 44 shows the improved work placement mechanism in GUM. Idle PEs which lack local sparks may seek work from other randomly chosen PEs by sending them FISH messages. If a fished PE does have sparks available in the sparkpool,

```
IF idle(localPE) THEN
 IF runnable thread  THEN
  evaluate new thread
 ELSE
  IF spark in spark pool  THEN
    activate new spark
  FI
FI
IF no spark  THEN
  send FISH to random PE
FI
IF received FISH THEN
 IF spark in spark pool  THEN
     send SCHEDULE to origin-PE
   ELSE
    send FISH to random PE
   FI
 FI
FI
```

Figure 43: The original GUM Work Placement Mechanism

```
 IF idle(localPE) THEN
 IF runnable thread  THEN
  evaluate new thread
 ELSE
  IF spark in spark pool  THEN
    activate new spark
  FI
FI
IF no spark  THEN
  send FISH to random PE
FI
IF received FISH THEN
 IF spark in spark pool  THEN
   IF (Distance of origin-PE
       between minimum and
       maximum attached to spark)
       THEN
       send SCHEDULE to origin-PE
   ELSE
    send FISH to random PE
   FI
 FI
FI
```

Figure 44: Extended GUM Work Placement Mechanism

GUM uses static information when it is replying to the incoming FISH. It compares the distance of the original PE stored in the static table with distances attached to the spark. If the PE distance is less than the maximum and greater than the minimum spark distance, then a SCHEDULE message will be sent to the original PE. If it is not, the FISH message will be forwarded to another randomly chosen PE, unless its maximum life time is reached and it is re-forwarded to the original PE.

5.5.3 `parDist` Primitive Implementation

The implementation of the basic `parDist` primitive does not need modification only in the GUM runtime system. It also requires some modification to the GHC compiler in order to the `parDist` primitive be recognised from the Haskell program. We followed exactly the implementation of the `par` primitive. The only difference is that `parDist` primitive takes two additional integers representing the minimum and the maximum boundaries and a pointer to an expression in the graph (`Closure`).

5.6 Architecture-Aware Constructs Evaluation

We first investigate whether the new architecture-aware constructs can deliver improved performance on hierarchical architectures. We do so by considering common paradigms data parallelism, divide-and-conquer, and nested parallelism.

All measurements that have been performed in this chapter and the next chapter of this thesis use the machines located at Heriot-Watt University (MACS). **lxpara3** is an eight-core 8GB RAM, HP XW6600 workstation, comprising two Intel 5410 quad-core processors each running at 2.33GHz. The 32 Beowulf cluster nodes each comprise eight Intel 5506 cores running at 2.13GHz and 6GB RAM. All machines run CentOS 5.5 Linux. The Beowulf nodes are connected via a Baystack 5510-48T switch with 48 10/100/1000 ports. Both Beowulf and **lxpara3** are connected to the network with Extreme Networks Summit 400-48t, 48

```
parFibDist ::  Int  -> Int -> Int
parFibDist 0  t  = 1
parFibDist 1  t  = 1
parFibDist  n  t
          |  n <= t  = nFib n
          |  otherwise = parDist min max x (y `pseq` (x + y + 1))
          where
                x = parFibDist (n-1) t
                y = parFibDist (n-2) t
                (min, max) = findLevel n

nFib n = nFib (n-1) + nFib (n-2) + 1

findLevel ::(Ord a,Num a)=> a -> (a,a)
findLevel  x
        | (x <= 46) =(0,0)
        | (x == 47) =(1,1)
        | (x == 48) =(2,2)
        | otherwise =(3,3)
```

Figure 45: `parFibDist` Program

10/100/1000BASE-T, 4 mini-GBIC, Extremeware.

The behaviour of the new architecture-aware constructs is investigated with common parallel programming paradigms. A divide-and-conquer paradigm is a parallelism that generates more parallelism from sub-workers. A data parallel paradigm is a parallelism which generates by a master process with varying computation size. A combined paradigm parallelism is a mixed source of parallelism, where the inner source of parallel should be kept together.

5.6.1 Divide-and-Conquer Parallelism

To investigate the new architecture-aware constructs for divide-and-conquer parallelism, we use the `parFibDist` version of the parallel nfib function shown in Figure 45. Here, `n` is the Fibonacci number, and `t` is the threshold value below

Block Size	`par`	`parBound`	`parAtLeast`	`parDist (n-1) n`	`parDist (n-2) n`	`parExact`
Very Small	0 – maxLevel	0 – 0	0 – maxLevel	0 – 0	0 – 0	0 – 0
Small	0 – maxLevel	0 – 1	1 – maxLevel	0 – 1	0 – 1	1 – 1
Medium	0 – maxLevel	0 – 2	2 – maxLevel	1 – 2	0 – 2	2 – 2
Large	0 – maxLevel	0 – 3	3 – maxLevel	2 – 3	1 – 3	3 – 3

Table 21: `findLevel` Configuration

which we use sequential computation (`nFib`). Above the threshold, `x` is sparked with a `parDist` parametrised by levels computed by `findLevel`.

`findLevel` is a programming abstraction that is used, with the parameters specified in Table 21, for several experiments in this section. Table 21 shows possible configurations of the `findLevel` function, ordered from left to right with increasingly specific spark placement.

To obtain reasonable performance, a key issue is to determine the threshold thread granularity values in the `findLevel` function. One approach, and that used for these benchmarks, is to determine the threshold values by experimentation. An automatic runtime system level solution is possible if the thread granularity of each spark can be identified, for example by program or resource analysis. Then suitable thread granularity thresholds can be determined for each level in a given architecture, for example using a benchmarking suite.

Figure 46 compares the performance of `par` and the architecture-aware constructs on the architecture specified in Section 5.1 configured with 1, 2, 4, 8 or 16 cores. The cores are distributed equally among machines. Throughout the results section, each data point, unless otherwise stated, is the median of 3 executions. The results show that the architecture-aware constructs perform better

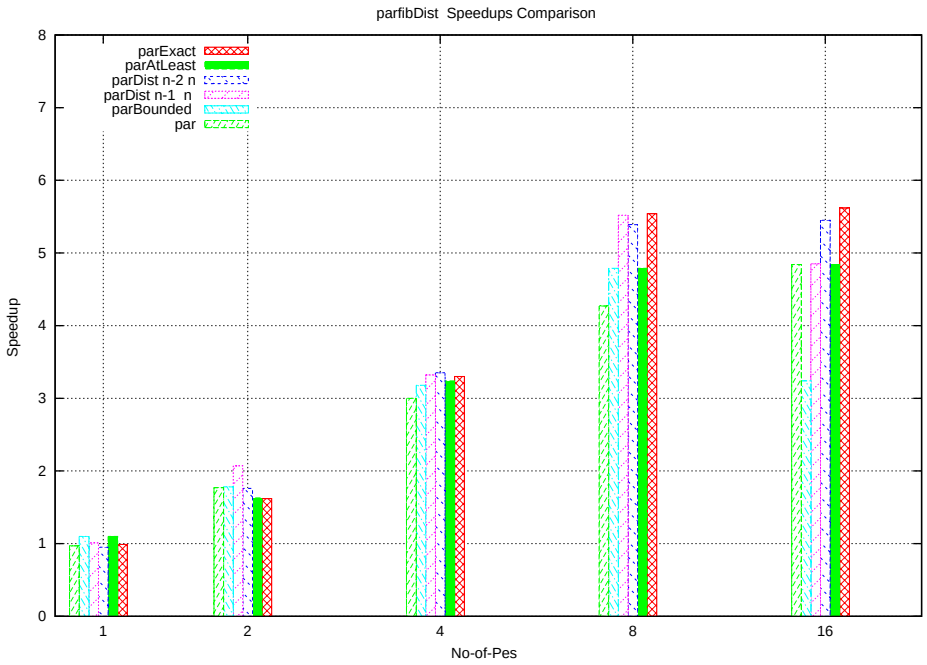


Figure 46: `parFibDist` Speedup

than `par` in almost all cases. A rare exception occurs on 16 cores where the `parBound` speedup is 3.24, and `par` is 4.8, and occurs as all constructs are subject to scheduling accidents. However, this degree of influence depends on the specificity of the spark: the more specific the spark is, the less influence scheduling accidents have. Interestingly, all constructs scale as the number of cores increases, except `parBound` which performs slightly worse on 16 cores. The best speedup is achieved by `parExact`: a speedup of 5.62 on 16 cores.

Moreover, the constructs give almost identical performance on a single core and very similar performance on small architectures up to 4-cores. This pattern is repeated for all programs in the following sections.

Program Name	Smallest Spark Runtime (s)	Largest Spark Runtime (s)	Number of Sparks
`parMapList`	0.14	40.47	277
`parMapIntervals`	0.19	55.00	276
`Allparam`	0.43	2.90	3003

Table 22: Task Size and Irregularity

5.6.2 Data Parallelism

To investigate the new architecture-aware constructs for data parallelism, we use two programs. The intention for both programs is to generate data parallel tasks of random thread granularity. Both programs compute some function on every element of a list. The first program, `parMapList` in Figure 47, splits the list into sublists of random sizes, and the second program, `parMapIntervals` in Figure 48, splits the interval into subintervals of random sizes, and the variation in task sizes for the programs is shown in Table 22. The `Allparam` program will be presented in Section 5.6.3. The difference between the programs is that `parMapList` communicates the list, where `parMapIntervals` communicates only the start and end points of the interval. Both programs compute the Euler `sumTotient` on each list interval or sublist. The complete code for the `parMapList` program can be found in Appendix A.1.

For both programs the architecture-aware constructs distribute the work depending on the size of the sublist or interval: small intervals are executed locally and large intervals are sent to be executed on a remote core. We use the `parMapLevel` skeleton in both programs to achieve parallelism. A `parMapLevel`

```
-- Top level functions of  parMapList program

dataListtop :: ([Int] -> a) -> Int -> Int -> Int -> a
dataListtop f lower upper  t  =
         let
          randomList  = mkRandom t
          list  = splitWithSize randomList [lower..upper]
          in
            sum (parMapLevel f findLevel  list)

sumTotient :: [Int] -> Int
sumTotient  xs  = sum (map euler  xs )

parMapLevel :: (Ord a ,Num a)=>(a -> b)->(a -> (Int,Int))-> [a] -> [b]
parMapLevel f fl [] =  []
parMapLevel f fl (x:xs)= parDist min max fx (fxs `pseq`( fx : fxs))
        where
            fx  = f x
            fxs = parMapLevel f xs
            (min,max) = fl x

splitWithSize::[Int] -> [Int]-> [[Int]]
splitWithSize _  [] = []
splitWithSize  (b:bs) xs = xss
             where
               xss = (take b xs):splitWithSize bs (drop b xs)
```

Figure 47: `parMapList` Program

```
-- Top level function of  parMapintervals program

dataIntervaltop f lower upper seed =
         let
          randomList = mkRandom seed
          intervalList = splitIntervals (lower,upper) randomlist
          in
           sum (parMapLevel f findLevel  intervalList)

splitIntervals ::(Ord a,Num a )=> (a,a) -> [a] -> [(a,a)]
splitIntervals (lower,upper) (b:bs)
        | ((upper-b-1) <= lower) =[(lower,upper)]
        | otherwise = ((upper-b),upper):splitIntervals
                      (lower,(upper-b-1)) bs

mkRandom  mx  =
         let
          g = mkStdGen 1601
          cs :: [Int]
          cs = randoms  g
          randomList = map (`mod` mx) $ cs
          in
           randomList
```

Figure 48: `parMapIntervals` Program

sparks the mapped function using `parDist` constructs for each list element. Figure 47 includes the definition of the `parMapLevel` skeleton. We will discuss the skeleton in more detail later in section 6.3.

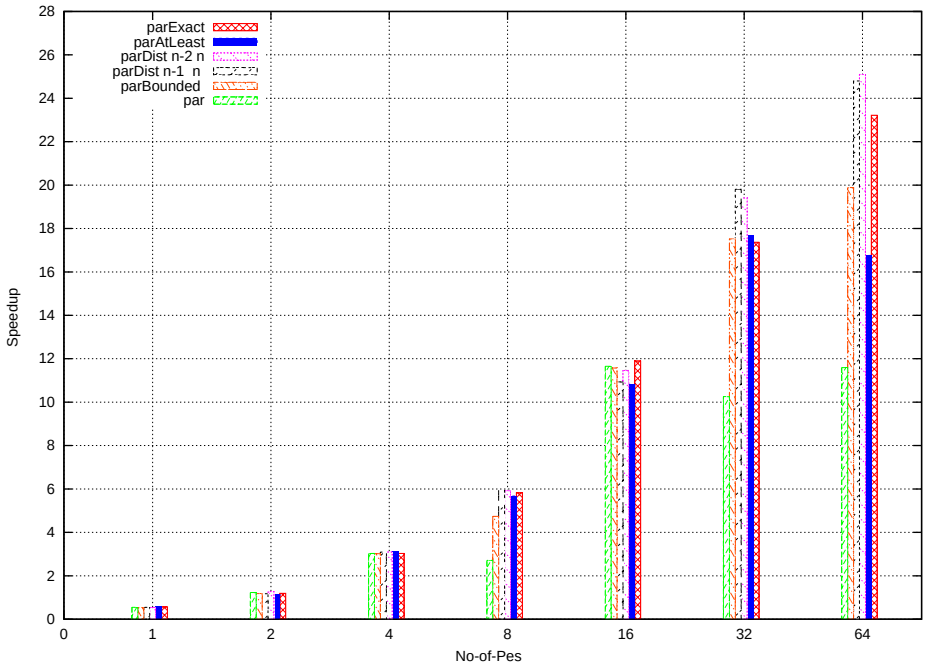


Figure 49: `parMapList` Speedups (64 Cores)

Both programs have similar performance on 64 cores. Figure 49 compares the absolute speedup of `par` and architecture-aware constructs for both programs, on up to 64 cores. We make the following observations:

- All architecture-aware constructs perform better than `par`, sometimes by a factor of 2, e.g. on 64 cores.

- All of the architecture-aware constructs scale: as the number of processors increase, the speedup increases. `par`, however does not scale beyond 16 cores. The reason is that `par` does not provide any restriction on thread placement. Thus, a small thread can be executed remotely which does not cover the communication cost.

- The maximum speedup of 25.1 is obtained from `parDist (n-2) n`.

- While `parExact` gives good performance, it is vulnerable to adverse scheduling because it identifies a specific level. Less prescriptive constructs like `parDist (n-2) n` can give better performance on large architectures, e.g. `parMapIntervals` on 32 and 64 cores.

Figures 50 and 51 contain the corresponding runtime curves for both programs. The efficiency of the constructs and `par` are compared by plotting its runtime for increasing numbers of PEs. The x-axis enumerates the number of participating PEs, and the y-axis gives the resulting runtimes in seconds. Note that the x-axis starts from 8 PEs, the number of cores in each multicore machine. These results show that `par` is slower than the architecture-aware constructs by a factor of 2.27 (674.14/296.53) for `parMapIntervals` and by a factor of 1.56 (577.54/368.67) for `parMapList` on 8 cores.

Figures 52 and 53 show the absolute speedups of `parMapList` and `parMapIntervals`

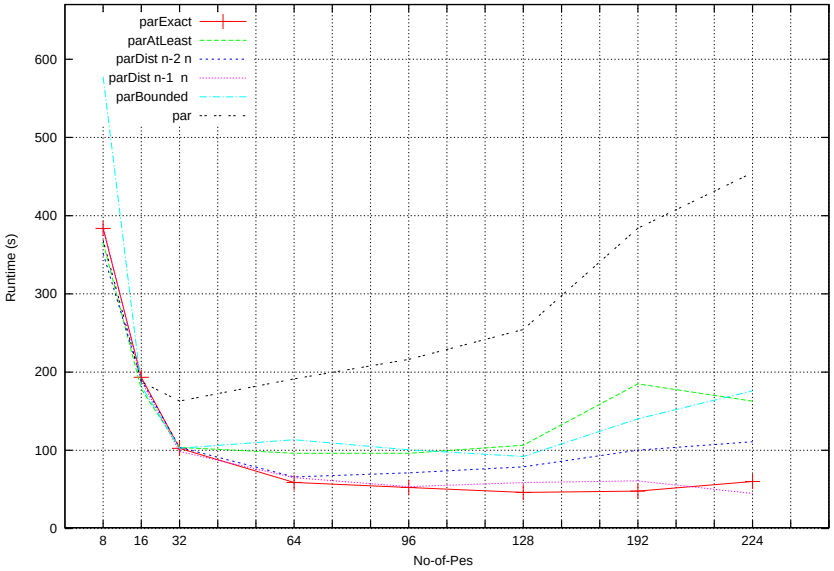


Figure 50: `parMapList` Runtimes

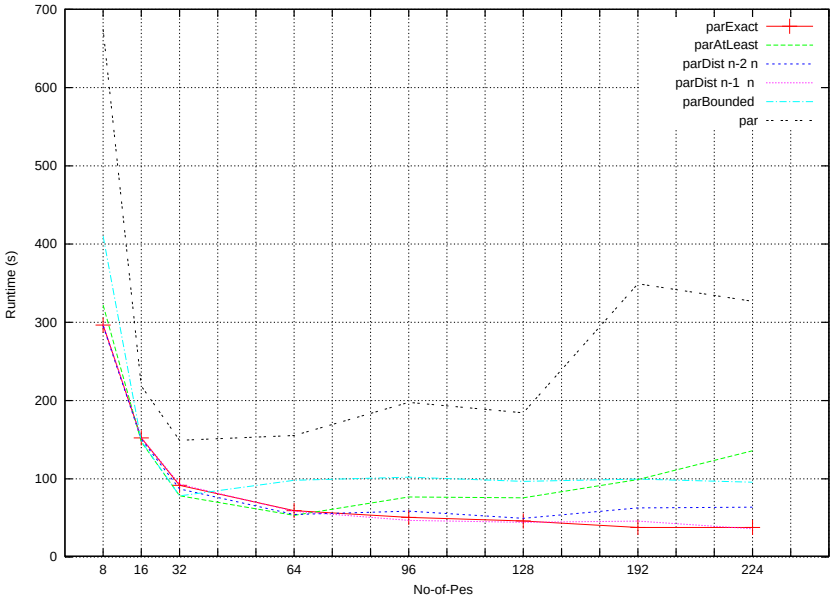


Figure 51: `parMapIntervals` Runtimes

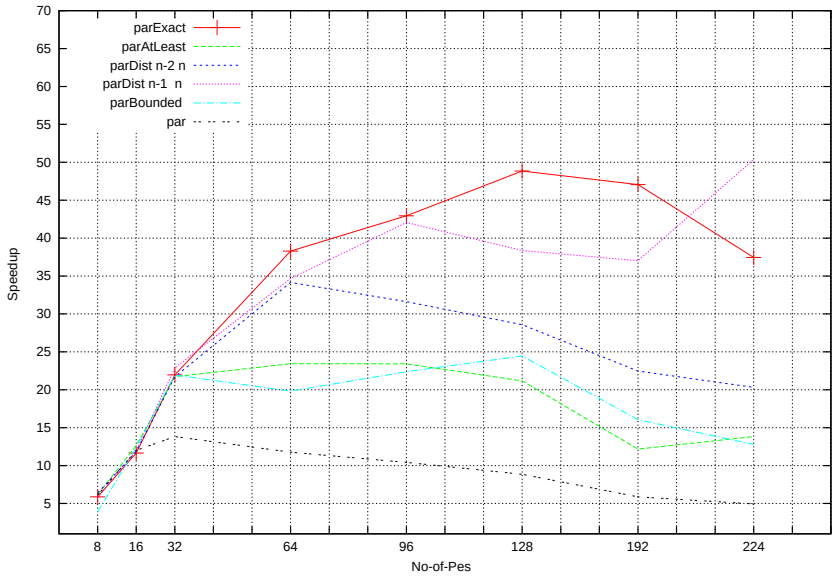


Figure 52: `parMapList` Speedups (224 Cores)

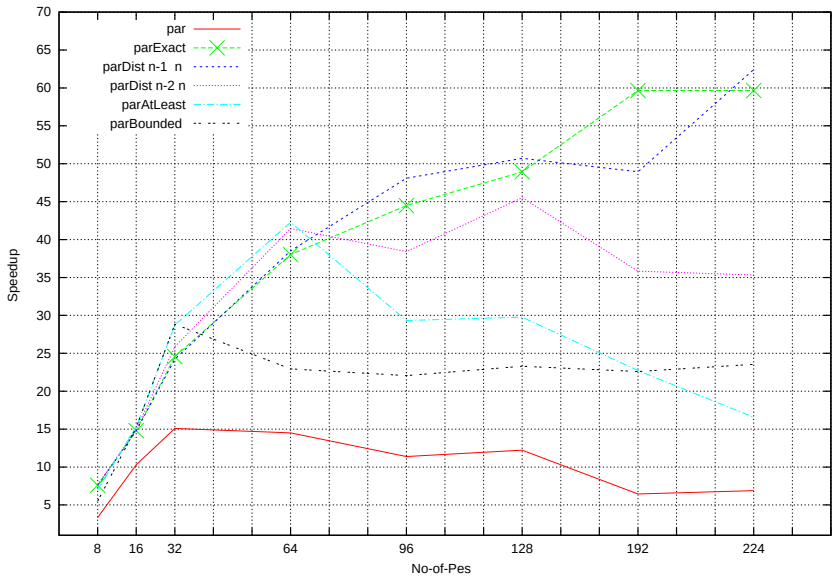


Figure 53: `parMapIntervals` Speedups (224 Cores)

programs measured on 8, 16, 32, 64,..., 224 cores. The absolute speedups are calculated with respect to the optimised sequential runtime (2252.60s) for the larger problem size of 140000. We make the following observations:

- As before, the architecture-aware constructs consistently perform better than `par`: by a factor of 2.5 (454.4/175.9) in the worst case for 224 cores, and 10.2 (454.4/44.7) in the best case.

- The `parExact` architecture-aware construct scales: as the number of processors increases, the speedup increases. In general, the architecture-aware constructs consistently perform better than `par`: by a factor of between 2.5 and 10.2 on 224 cores.

- The `par` has poorer performance because the `par` program generates significantly higher number of messages than the other construct programs. E.g. on 224 cores the `par` exchanges 13360 messages, where the `parExact` program exchanges just 2292 messages.

- While `parExact` gives good performance, it is vulnerable to adverse scheduling because it identifies a specific level. Less prescriptive constructs like `parDist (n-2) n` can give better performance on large architectures, e.g. `parMapIntervals` on 32 and 64 cores.

- All of the architecture-aware constructs scale better than `par`. `par`, however, does not scale beyond 32 cores. The reason is that `par` does not provide

any restriction on thread placement. Thus small thread can be executed remotely, which does not cover the communication cost.

- In both graphs `parExact` scales more smoothly, but `parDist (n-1) n` ultimately delivers the best performance.

- In absolute terms, `parMapIntervals` has 20% better performance than `parMapList`, as it communicates only a pair of numbers, rather than a list segment.

```
mapparfib  n t xs = parMapLevel (parFibOneLevel t) findLevel randomList
        where
          randomList = take  n ( filter (>35) xs )

parFibOneLevel ::Int-> Int -> Int
parFibOneLevel t n
           | n < t = nfib n
           | otherwise =  parBound 1  x  (y `pseq` (x+y+1))
                 where
                     x = parFibOneLevel t (n-1)
                     y = parFibOneLevel t (n-2)
```

Figure 54: `Allparam` Program

5.6.3 Nested Parallelism

To investigate the new architecture-aware constructs for nested parallelism we use the `Allparam` program, in Figure 54, with top level data parallelism and nested divide-and-conquer parallelism. More specifically, it maps the parallel divide-and-conquer Fibonacci function `parFibOneLevel` over a list of random integers, in parallel, using `parMapLevel`.

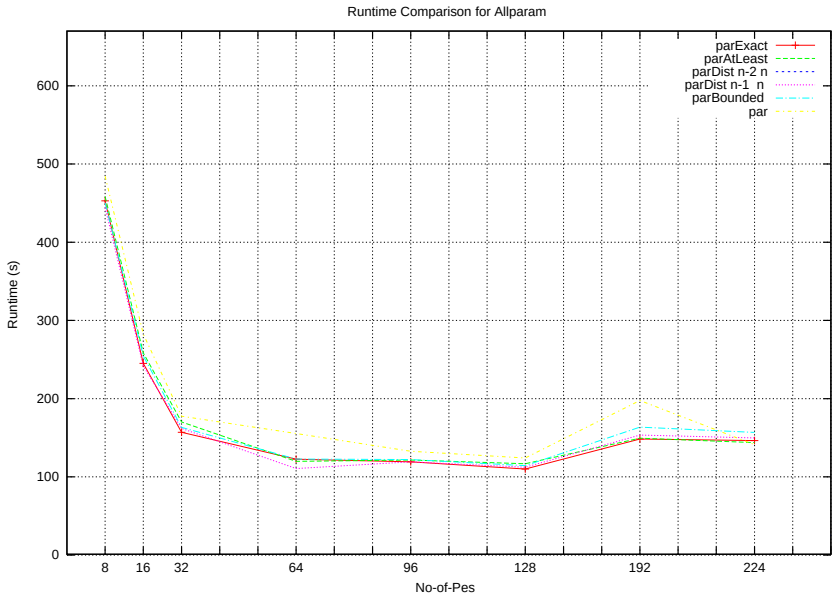


Figure 55: `Allparam` Runtimes

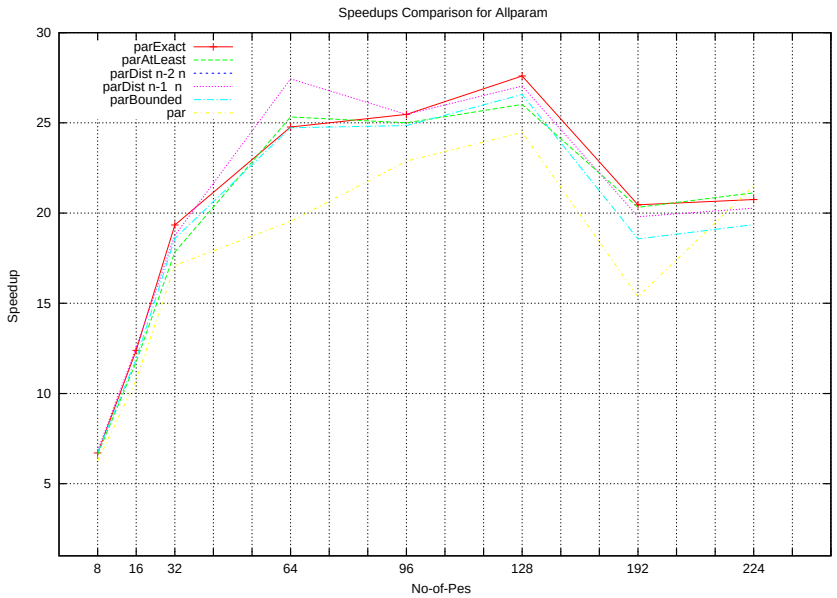


Figure 56: `Allparam` Speedups(224 Cores)

Figure 55 illustrates the runtime curves obtained with the constructs. The efficiency of the constructs and `par` are compared by plotting their runtimes for increasingly large numbers of PEs. These results show that `par` is slightly slower than architecture constructs, by a factor of 1.1 (484.46/452.88) on 8 cores. All runtime curves decrease as the number of PEs increases up to 128 cores.

Figure 56 shows the absolute speedups obtained with the constructs. The results are similar to those observed with the divide and conquer and data parallel programs: the architecture-aware constructs consistently outperform `par`, with an exception at 224 cores. `parExact` and `parDist (n-1) n` deliver the best performance. None of the constructs scales beyond 128 cores, and we attribute this to the task sizes being too small for large architectures, as illustrated by the maximum task size in Table 22.

5.6.4 Performance Variability

Finally, we investigate the performance variability induced by each construct. The variability reflects how susceptible the work distribution policy of each construct is to scheduling accidents. Table 23 shows the wide variation in runtimes for 11 executions of the three programs with the different constructs. We make the following observation, taking `parMapIntervals` as an example for discussion:

- The architecture-aware constructs have far less variation than `par`, with a range of 121s and a standard deviation (sd.) of 33s.

		par	parBound	parAtLeast	parDist (n-1) n	parDist (n-2) n	parExact
parMapIntervals	Median	98.2	32.7	42.4	**23.4**	26.9	29.9
	Mean	89.6	35.7	45.7	25.4	26.9	30.3
	Max	149.7	58.4	79.1	35.3	33.3	38.6
	Min	29.1	24.2	24.9	21.6	23.1	24.4
	Range	120.6	34.2	54.2	13.7	**10.2**	14.2
	Stdev	33.0	10.2	15.6	**4.3**	3.2	4.8
parMapList	Median	57.0	23.1	30.2	22.9	23.6	**22.4**
	Mean	54.3	24.5	31.0	22.9	24.1	22.7
	Max	62.6	37.1	38.5	25.3	31.8	29.6
	Min	41.4	21.1	22.4	19.5	19.8	19.0
	Range	21.2	16.0	16.0	**5.8**	12.1	10.6
	Stdev	6.3	4.3	4.6	1.4	3.5	**2.5**
Allparam	Median	27.7	23.9	**21.5**	23.4	23.5	24.2
	Mean	27.8	23.6	22.2	22.9	23.5	23.8
	Max	32.1	27.0	26.2	25.7	25.2	26.6
	Min	22.9	20.0	19.2	20.0	20.8	21.0
	Range	9.2	7.1	7.0	5.7	**4.4**	5.6
	Stdev	2.8	2.3	2.3	1.9	**1.3**	1.7

Table 23: Variability of benchmark runtimes (11 executions) on 64 cores.

- `parDist (n-1) n` and `parDist (n-2) n` have the least variation (range 13.7s and 10.2s, sd.s 4.3s and 3.2s). It seems that having multiple levels available enables the runtime system to ameliorate scheduling accidents.

- Of the architecture-aware constructs, `parAtLeast` has the worst performance.

5.6.5 Discussion

The results obtained for the prospective architecture-aware constructs are consistent for all three classes of programs investigated. Every architecture-aware construct consistently delivers better speedup and scalability than `par`, with dramatically reduced variability. The `parExact` and `parDist (n-1) n` constructs

deliver the best performance, and for these programs with simple coordination, `parExact` scales more smoothly. `parDist` is clearly the most powerful construct, but requires the programmer to specify the most information, i.e. both an upper and a lower bound. It is therefore best to be used internally or only by expert parallel programmers.

In summary, we recommend using both the `parDist` and the `parExact` constructs: `parDist` for expressive power and `parExact` for simplicity and performance.

5.7 Applying constructs in other Languages

The suggested architecture-aware constructs could be implemented in any language with task creation. For example, the Erlang distribution model permits explicit process placement [61]. A process in Erlang is spawned on a named node. Such a static, directive mechanism is hard for programmers to manage for anything other than small scale or very regular process networks. We believe there is a possibility to build an abstraction layer that maintains a tree of node groups, loosely modeling the underlying architecture. This tree contains a number of levels: Level 0 for this Core, Level 1 for Cores in SMP, Level 2 for Cores in cluster, Level 3 for Cores in other clusters, ..., etc. An explicit spawn function can be defined to place a process in a group of PEs. This group of PEs may appear in a multi-cluster or group of multi-clusters.

```
RemoteProcess = spawnAt(GroupId,...)
```

The process placement can be directly controlled with a new mechanism, and more explicit portable distribution can be obtained by semi-automatically controlling of two locality aspects:

- Affinity specifies how close processes must be located, e.g. two rapidly communicating processes may need to be located in the node group on the same SMP. The language primitive is a bounded spawn that specifies the maximum number of levels in the node group hierarchy in which a spawn may occur in. For example:

```
SpawnBounded(0,...)      Process must be placed on
                         the same core
SpawnBounded(1,...)      Process must be placed on
                         the same SMP node
SpawnBounded(2,...)      Process must be placed on
                         the same cluster
```

- Distribution specifies how far the process must be placed from the spawning process. For example, two large computations, e.g. simulation components or test sets, may need to be placed on separate clusters. The language primitive is a bounded spawn that specifies the maximum number of levels in the node group hierarchy that a spawn may occur in. For example:

```
SpawnAtLeast(1,...)      Process must be placed on
                         some other core
```

`SpawnAtLeast(2,...)`	*Process must be placed on some other SMP*
`SpawnAtLeast(3,...)`	*Process must be placed on some other cluster*

Of course these primitives can be generalised as a primitive that controls maximum and minimum simultaneously, e.g. `SpawnDist(0,3,...)`.

A key point of the mechanisms is that the mapping between the node group hierarchy is abstract, e.g. two clusters with a fast interconnect can be treated as a single cluster. The distance measures are abstract, and could be computed. While the primitives restrict the set of nodes where a process may be placed, they do not identify them specifically. The advantages of this technique are preserving performance portability by exposing a virtual, rather than physical architecture and minimising prescription.

5.8 Summary

In this chapter, we highlighted the trend towards hierarchical architecture as the dominating low-level architecture for high performance computing, which has become the usual form for building clusters of computers. We present an example of a Beowulf cluster built from multicore nodes and how this cluster may connect to other clusters within the same organisation or other organisations (Section 5.1).

We have presented a selection of relevant proposed approaches. We believe

that the concept of architecture awareness is essential for hierarchical architectures. We also outlined how the approach proposed in this thesis can be implemented in other languages that include task creation (Sections 5.2 and 5.7).

In response to the architecture trend towards hierarchical communication topologies, and due to rapid evolution of these topologies, we have proposed an architecture-aware programming model for parallel Haskell. We started by defining four new architecture-aware constructs for GpH. The constructs aim to control data locality and work distribution, guided by information about task size. They do so by constraining communication abstractly and with as little specific prescription as possible; that is, the constructs identify layers of the communication hierarchy, and allow the implementation to dynamically control placement within the layer (Section 5.3).

We have presented a simple Haskell definition of the sets of PEs that each construct identifies when executed on a PE of participating PEs. We have defined a set of functions to demonstrate the constructs' semantics. A `distance` function is used to calculate the distance between two given PEs in the architecture hierarchy. A `setparDist` function specifies a set of PEs on which task may be executed in the architecture hierarchy, for given boundary. The `setparDist` function is used to define the `setparBound` and `setparAtLeast` functions (Section 5.4).

In Section 5.5, we have presented the implementation of the suggested constructs. We have described the improvement that has been made on the GUM runtime system. The amendments include the storage of granularity information

with each spark, the storage of distance information between PEs and the extension of the fishing mechanism. We also describe the implementation of the basic `parDist` primitive on the compiler.

We have evaluated the constructs using simple data parallel, divide-and-conquer, and mixed-source parallel programs, each with irregular thread granularity. The evaluation shows that every architecture-aware construct consistently delivers better speedups, better scalability on up to 224 cores and far less variation in execution time than the existing `par` primitive. Because `par` does not provide any restriction on thread placement, small threads can be executed remotely, thus leading to poorer performance. At times, speedup is improved by an order of magnitude. Of the proposed constructs, `parExact` and `parDist (n-1) n` deliver the best performance with good task size information, `parExact` scales most smoothly, and `parDist` is the most expressive construct. We recommend `parDist` for expressive power and `parExact` for simplicity and performance (Section 5.6).

Chapter 6

Towards Architecture Aware Programming Models

This chapter presents some high-level abstractions over the new architecture-aware constructs. We develop some architecture-aware evaluation strategies (Section 6.2) and skeletons (Section 6.3).

A key issue in the programming model is deciding on what level in a hierarchical architecture a computation of a given size should be executed? We discuss this issue and seek abstractions over the architecture topology (Section 6.1).

We illustrate the performance of the architecture-aware strategies by examining the behaviour of two programs representative of two common parallel paradigms: a data parallel `sumEulerDist` program, and a divide-and-conquer queen placement program (Section 6.4). We report the performance results of

architecture-aware skeletons (Section 6.5). We demonstrate the effect of architecture-aware programming on memory consumption and generated parallelism (Section 6.6). The most significant improvements are due to preventing small tasks from being executed far from their originator and the dramatic reduction of communication overheads.

6.1 Architecture-Aware Decisions

To achieve reasonable performance with modest programming effort, four issues must be addressed in the proposed architecture-aware programming model. The first two are identification of the architecture hierarchy levels and the thread granularity of each task. The third is providing an effective mechanism for mapping threads onto levels. Finally abstractions must be provided that enable programmers to exploit the architecture while preserving performance as far as possible.

The parallel computational resources are increasingly hierarchical. They may be imbalanced in communication costs, processor speed, and memory access, as previously discussed in Section 2.1.4 and Section 5.1. The abstraction must capture the essential topological properties of the hierarchy as a tree, possibly unbalanced. The leaves of the tree represent PEs which are grouped into levels, depending on the expected latency of communications between them. For the benchmarks in this thesis, we have adapted the runtime system to automatically generate a virtual architecture tree, reflecting the underlying architecture and

store the tree in each participating PE, to be used for threads mapping. To generate the virtual tree, the runtime enviroment generates its own unique number for each PE to distinguish between PEs, and collects the CPU speed and network IP address. Then the PEs are grouped in a number of levels according to the distance to each PE according to their IP address. The levels constructed are: Level 0 represents the current PEs, Level 1 represents all PEs in the same node, Level 2 represents all PEs in other nodes of the same cluster, Level 3 represents all PEs in other clusters.

Thread granularity [108] is one of the most relevant parameters for parallel programs, since it determines the amount of real work in the parallel task. Too fine a granularity leads to high communication overheads, while too coarse granularity leads to imbalanced loads. The aim is to determine the right granularity for parallel tasks to achieve the best performance.

In a hierarchical architecture a key issue is to determine at what level a task of a given granularity should execute. For some programs, like our benchmarks, this may be readily apparent. However, an automatic runtime system level solution is possible if the thread granularity of each spark can be identified, for example by program or resource analysis. Teresco et. al. present a variety of architecture-aware approaches to parallel computation some of which automatically use resource information [117]. The task granularity for the benchmarks used in this thesis is determined by experimentation, based on a threshold value.

We support architecture-aware programming by providing high level abstraction functions. Figure 57 shows the definition of a `findLevel` function based on input value which is used in the programs measured in Section 5.6.1. Figure 58 shows an alternative method based on the depth of recursion. The depth of recursion is the number of recursive calls within a function. The second alternative is a relative choice because it is independent of problem description. We use the depth of recursion alternative in all programs measured in Section 6.5. Here `x`

```
findLevel::(Ord a,Num a)=>a->(a,a)
findLevel  x
        | (x <= 46) =(0,0)
        | (x == 47) =(1,1)
        | (x == 48) =(2,2)
        | otherwise =(3,3)
```

Figure 57: findLevel Based on Input Argument

```
findLevel::(Ord a,Num a)=>a->(a,a)
findLevel x
      | (x < 8) = (3,3)
      | (x >= 8 && x < 10) = (2,2)
      | (x >= 10 && x < 12) = (1,1
      | otherwise = (0,0)
```

Figure 58: findLevel Based on Depth of Recursive Call

represents the depth of recursion, i.e. the number of calls to the worker function. The constant numbers in Figures 57, 58 are obtained by sequential profiling of the programs and then manually coding them into the programs. This is useful, for example, for divide-and-conquer algorithms, which start with large problems and breaks them into smaller problems with each recursive call. We can therefore allocate tasks generated from different recursive calls among different architecture hierarchy levels. The measurements in Section 5.6.1 show similar performance for both alternatives.

6.2 Architecture Aware Evaluation Strategies

This section illustrates how architecture awareness can be expressed in the evaluation strategies. The basic architecture awareness strategy is `rparDist`. It is a replacement of the `rpar` strategy.

```
rparDist :: Int -> Int -> Strategy a
rparDist min max x = parDist min max x (return x)
```

The `rparDist` strategy is used for creating parallelism; it indicates that its argument could be evaluated in parallel within a lower bound and an upper bound of hierarchy. The `rparDist` strategy is used for the development of all strategies and skeletons presented in this chapter.

We have discussed different versions of `parList` in Chapter 4. All `parList` versions described previously traverse the list, sparking list elements. The `parList` function in Figure 59 returns a new list containing the strategy applied to each element of the list.

We define a `parDistList` strategy to include architecture awareness in Figure 60, which takes a boundary directly from the programmer and sparks tasks accordingly. The `parDistList` is used in the `Queen` program measured in Section 6.4.2. Figure 62 shows only the key top-level function where the parallel coordination is inserted.

Another architecture-aware strategy similar to the `parList` strategy is the

```
parList :: Strategy a   ->
           Strategy [a]
parList start [] = []
parList start (x:xs) = do
    x' <- rpar (x `using` strat)
    xs' <- parList strat xs
    return(x':xs')
```

Figure 59: Original `parList`

```
parDistList :: Int -> Int ->
               Strategy a ->
               Strategy [a]
parDistList min max strat (x:xs) = do
  let
  x'<-rparDist min max (x `using` strat)
  xs'<-parDistList min max strat xs
  return (x':xs')
```

Figure 60: Architecture-aware `parDistList` Strategy

`parListLevel` shown in Figure 61. The `parListLevel` strategy takes an additional list, containing the level of each element of the input list. This could be genrated by mapping the `findLevel` function over the input list. The `parListLevel` strategy is used in the `sumEulerDist` program measured in Section 6.4.1. The complete code for the program is listed in Appendix C.1 .

```
parListLevel :: [Int] -> Strategy a -> Strategy [a]
parListLevel ns strat [] =return []
parListLevel (n:ns) strat (x:xs)  = do
          let
          x' <-  rparDist n n  (x `using` strat )
          xs'<- parListLevel ns  strat xs
          return (x':xs')
```

Figure 61: Architecture-aware `parListLevel` Strategy

6.2.1 Using the `parDistList` Function

We have used the `parDistList` function in the `Queen` program, which places queens on a NxN chess board so that they do not attack each other. Figure 62 shows the key top-level function where the parallel coordination is inserted using

the rparDist and parDistList strategies.

```
pargen :: Int -> [Int] -> [[Int]]
pargen n b
       | n >= threshold = iterate gen [b] !! (nq -n)
       | otherwise      = concat bs
       where
         bs  = do
          let
           level = findLevel (nq-n)
          xs <- rparDist level level ((map (pargen (n+1))
                (gen [b]))`using` parDistList 0 1 rdeepseq)
          return(xs)

findLevel n
          | (n < 8) = 0
          | ((n >=8) && (n < 10 )) = 1
          | ((n >=10) && (n < 13 )) = 2
          |otherwise =3
```

Figure 62: Queen Top Level Function

The rparDist strategy is to generate sparks for a certain level. The parDistList strategy is to localise the generated parallelism to be executed on the same multi-core machine. The performance of Queen using parDistList strategy is presented Section 6.4.2.

6.2.2 Using the parListLevel Function

To investigate the performance of parListLevel we use the sumEulerDist program. Figure 63 shows the key function of the sumEulerDist program, using the parList and parListLevel strategies. The program computes the sum of the Euler's totient function applied to each integer up to a given bound. The

`sumEulerDist` program splits the input list into sublists of fixed size and then computes the sumTotient on each sublist, in parallel. The complete code for the `sumEulerDist` can be found in Appendix C.1. The performance results for program are presented in Section 6.4.1.

```
-- parList strategy version

fun lower upper block = sum((map sumTotient subList)
                          `using`  parList  rdeepseq)
     where
      subList =  splitAtN block [lower ..upper]

-- parListLevel strategy  version

fun lower upper block = sum((map sumTotient subList)
                          `using`  parListLevel cosList  rdeepseq)
     where
      subList =  splitAtN block [lower ..upper]
      cosList= map findLevel subList
```

Figure 63: Architecture-aware `sumEulerDist`

6.3 Architecture Aware Skeletons

Algorithmic skeletons define general patterns of computation which are useful for exposing the computational structure of a program [32]. Skeletons are high level abstractions that support widely used parallel programming patterns, where parallelism details are encapsulated into these abstractions [13]. We have adapted some skeletons to make use of the architecture-aware constructs. The `parMapLevel` skeleton is used in all programs measured in Section 5.6.2. Further

performance results can be found in Section 6.4.1.2. The remaining skeletons are used for measurement in Section 6.5.

6.3.1 An Architecture Aware Parallel Map Skeleton

Figure 64 presents both the sequential map and a simple parallel map skeleton.

```
-- Sequential  version
map :: (a -> b) -> [a] -> [b]
map f (x:xs) =  f x : map f xs

-- Parallel version
parMap :: (a -> b) -> [a] -> [b]
parMap f (x:xs) = fx `par` fxs
                  `pseq` res
            where
              fx = f x
              fxs = parMap f xs
              res = fx:fxs
```

Figure 64: Sequential Map and Parallel parMap Skeletons

```
parMapLevel :: (Ord a ,Num a)=>
               (a -> b)->
               (a -> (Int,Int))->
               [a] -> [b]
parMapLevel f fl (x:xs)= res
    where
      fx  = f x
      fxs = parMapLevel f fl xs
      (min,max) = fl x
      res = parDist min max fx (fxs
            `pseq`(fx:fxs))
```

Figure 65: parMapLevel Skeleton

The parMap function could be modified to capture the architecture topology by replacing the `par` with the architecture-aware construct `parDist`. We call the new architecture abstraction function `parMapLevel`.

The function `parMapLevel` takes three arguments: `f` is the function that will be applied to each element in the list, `fl` is the `findLevel` architecture abstraction function, and a list. The performance of a `parMapLevel` is illustrated in Section 5.6.2. Figures 66 and 67 present both the original and architecture-aware `parMap` skeletons.

```
parMap f (x:xs) = do
          b  <- rpar (f x)
          bs <- parMap f xs
          return(b:bs)
```

Figure 66: Monadic parMap Skeleton

```
parMapLevel fl  f (x:xs) = do
         let
            (mn,mx) = fl x
          b <- rparDist mn mx  (f x)
          bs <- parMapLevel fl  f xs
          return(b:bs)
```

Figure 67: Monadic parMapLevel Skeleton

6.3.2 A Divide-and-Conquer(DC) Skeleton

The general divide-and-conquer skeleton can be modified to capture the architecture topology by just one additional parameter. Figure 68 shows a simple parallel divide-and-conquer skeleton using the original evaluation strategies model presented in chapter 4.

```
divCon :: (a -> Bool)->
          (a -> b)  ->
          (a -> [a]) ->
          ([b] -> b) ->
          a -> b
divCon  t s d c x
    | t x = s x
    | otherwise = runEval $(do
       let
         (l,r) = (d x)
       x' <- (rpar `dot` rdeepseq)
             (divCon t s d c l)
       y' <- (rpar `dot` rdeepseq)
             (divCon t s d c l)
       return(c x' y')
```

Figure 68: General Parallel Divide-and-Conquer Skeleton

```
divCon :: (a -> Bool)->
          (a -> Int)  ->
          (a -> b)  ->
          (a -> [a]) ->
          ([b] -> b) ->
          a ->
          a -> b
divCon  t f s d c depth  x
  | t x = s x
  | otherwise = runEval $(do
   let
     (l,r) = (d x )
      mx =  f r
    x'<-((rparDist mx mx) `dot` rdeepseq)
        (divCon t f s d c (depth-1) r)
    y'<-((rparDist mx mx) `dot` rdeepseq)
        (divCon t f s d c (depth-1) l)
    return(c x' y')
```

Figure 69: Architecture Aware Divide and Conquer Skeleton

The parallel skeleton can be adapted to make use of architecture-aware constructs, as in Figure 69. The skeleton is parameterised with the typical control functions for divide-and-conquer which decide if a subproblem is trivial (t), how to solve a trivial subproblem (s), how to divide a non-trivial problem (d) and how to combine results of subproblems (c). Additionally, we provide a `findLevel` function along with depth. `findLevel` is a new parameter to exploit the communication architecture. The depth is to limit the amount of parallelism, by evaluating children below the depth sequentially. The performance of the skeletons is discussed in Section 6.5 and Section 6.6.

6.4 Evaluation of Architecture Aware Strategies

This section reports a performance evaluation of the architecture-aware strategies by reporting distributed memory results for the `sumEulerDist` program (Section 6.4.1) and shared and distributed memory results for the `Queen` program (Section 6.4.2). We investigate the performance of the abstractions on the target architecture described in Section 5.6. The runtime reported is the median of three executions to ameliorate variability of parallel runtimes caused by factors like the operating system (OS) and other users.

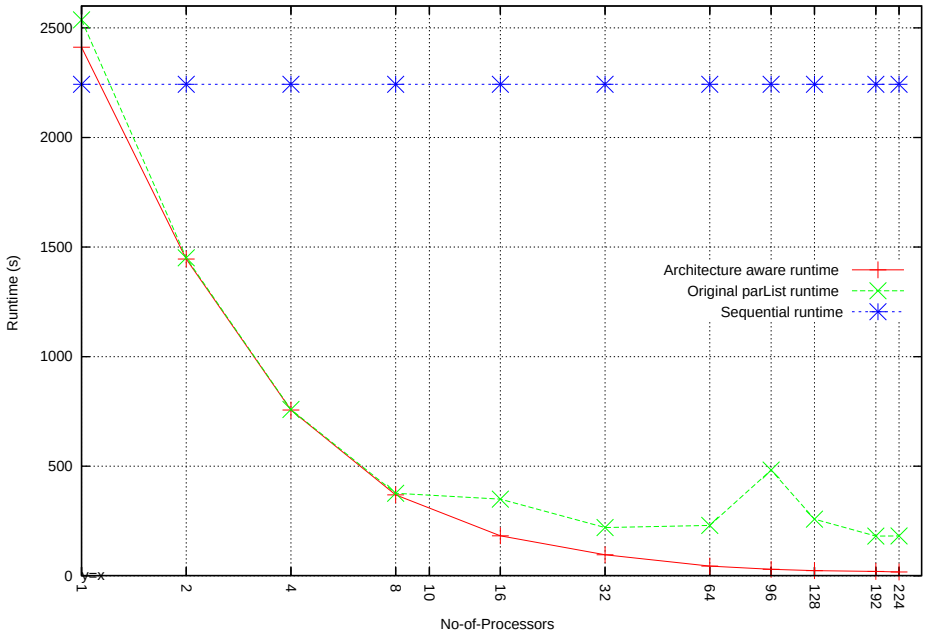


Figure 70: Arch. Aware vs Orig. Strategies Runtime Comparison (`sumEulerDist`)

6.4.1 SumEulerDist

The `sumEulerDist` program was measured on 1, 2, 4, 8, 16, 32, 64, ..., 224 cores of the architecture described in Section 5.1. The absolute speedup is based on the optimised sequential runtime (2242.59s) for problem size 140000. We evaluated two architecture-aware versions the `parListLevel` strategy and `Deep` strategy.

6.4.1.1 `parListLevel` Strategy Results

Figure 70 shows the runtime curves of the `parList` and the `parListLevel` strategies. Both strategies produce the same number of sparks. Therefore, we do not

expect any sequential overhead difference on a single core. Both original and architecture-aware strategies produce similar sequential overhead between 13.12% and 7.54% (Table 24). We believe the small difference in sequential overhead is a result of the system load during the program execution. If the underlying machines are heavily loaded by other processes, the system overhead will increase, as a result of the increase in OS processes scheduling time.

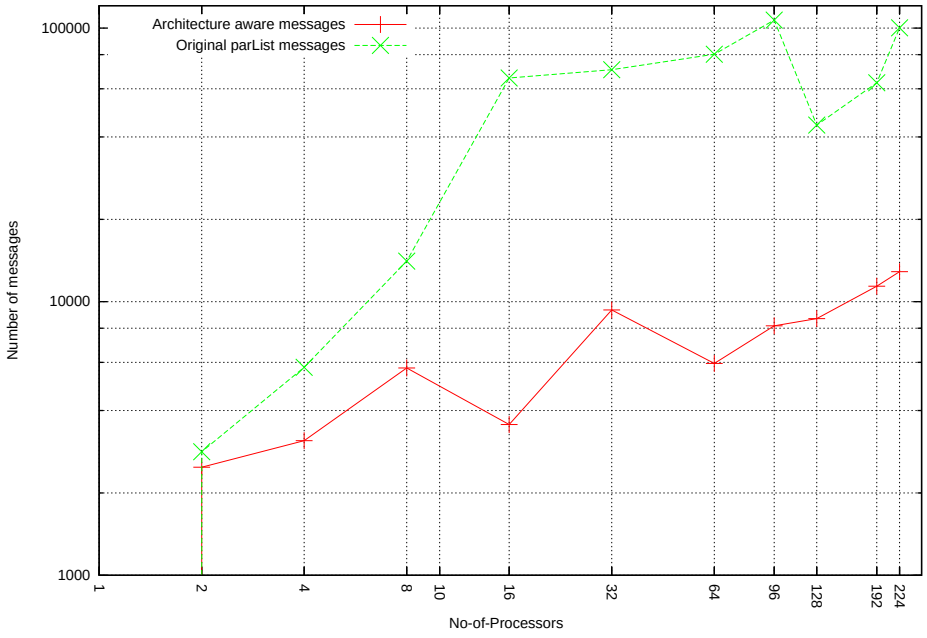


Figure 71: Arch. Aware vs Orig. Strategies: messages Comparison (`sumEulerDist`)

The runtime curves for all strategies are very similar on a small number of cores, and scale as cores are added. However, the `parListLevel` strategy scales better than the `parList` beyond 8 cores. This is due to the difference in the

amount of messages. Figure 71 shows that the `parList` program exchanges more messages than the `parListLevel` program. This pattern is repeated for all numbers of cores from 2 through 224. The interesting observation is that the difference in the number of messages increases as cores are added.

No PEs	sumEulerDist (Sequential Runtime 2242.72s)							
	Orig. parList			Arch. Aware parListLevel			Δ Time (%)	Δ Msgs (%)
	Time (s)	Spd	Msgs	Time (s)	Spd	Msgs		
1	2536.93	0.88	-	2411.82	0.93	-	-4.93	-
2	1450.99	1.55	2829	1445.58	1.55	2483	-0.37	-12.23
4	758.6	2.96	5756	756.56	2.96	3105	-0.27	-46.06
8	376.58	5.96	14075	369.28	6.07	5725	-1.94	-59.33
16	376.58	6.41	65728	182.16	12.31	3558	-51.63	-94.59
32	220.26	10.18	70495	96.12	23.33	9325	-56.36	-86.77
64	229.8	9.76	80257	44.16	50.78	5950	-80.78	-92.59
96	482.17	4.65	107000	29.7	75.51	8161	-93.84	-92.37
128	258.33	8.68	44249	23.61	94.98	8669	-90.86	-80.41
192	181.25	12.37	63134	19.72	113.72	11398	-89.12	-81.95
224	181.71	12.34	100230	17.55	127.78	12876	-90.34	-87.15

Table 24: Comparison of `parList` and `parListLevel` (`sumEulerDist`)

Table 24 analyses in detail the runtimes, speedups and number of messages of each strategy, running on 224 cores. Columns 2 and 5 of the table report runtimes of strategies. Columns 3 and 6 of the table report speedups. Columns 4 and 7 of the table report total messages. On 192 cores the wall-clock time for the `parListLevel` program is decreased by 89.12% compared with the `parList` program and the number of messages is decreased by 81.95% on the same number of cores. This indicates that the architecture-aware approach is successfully decreasing the messages by sending only large tasks to remote cores. To investigate this in more detail, we analyse the type of messages between cores.

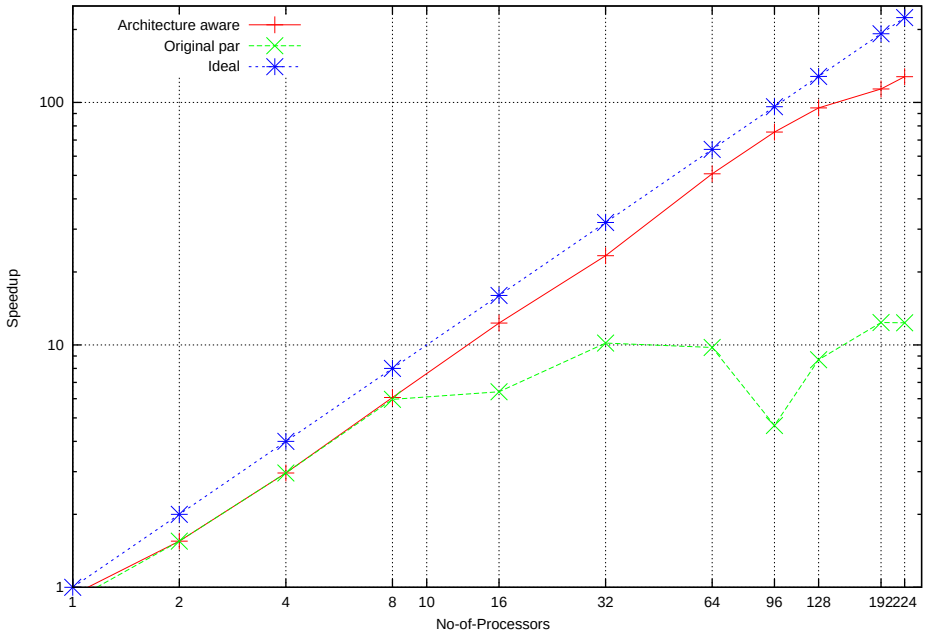


Figure 72: Arch. Aware vs Orig. Strategies Speedup Comparison (`sumEulerDist`)

Section 3.3 describes GUM messages between cores: `Fish`, `Schedule`, `Ack`, `Resume` and `Fetchme`. We focus only on the most important. `Fish`, `Schedule` and `Resume`. We study the number of fired messages when we execute both `parList` and `parListLevel` programs on 192 cores. The `parList` sends and receives 63134 `Fish` message in total compared with 11398 using `parListLevel`. In contrast to data messages, they send and receive 655 compared with 704, respectively. This means cores in the `parList` programs spend more time seeking for work than when other strategies are used. It also has another consequence: small tasks generated at the beginning of the program execution can be chosen by remote

No PEs	sumEulerDist					
	Orig. parList			Arch. Aware Deep		
	Time (s)	Spd	Msgs	Time (s)	Spd	Msgs
1	2536.93	0.88	-	2444.21	0.92	-
2	1450.99	1.55	2829	1477.08	1.52	1170
4	758.60	2.96	5756	682.29	3.29	2585
8	376.58	5.96	14075	342.99	6.54	3777
16	376.58	6.41	65728	167.58	13.38	3345
32	220.26	10.18	70495	80.27	27.94	3975
64	229.80	9.76	80257	42.34	52.97	6280
96	482.17	4.65	107000	30.11	74.48	9158
128	258.33	8.68	44249	22.40	100.12	6979
192	181.25	12.37	63134	19.10	117.41	12690
224	181.71	12.34	100230	25.60	87.60	21881

Table 25: Comparison of `parListLevel` vs Deep

cores in the `parList` programs but not in the `parListLevel` programs.

Figure 72 depicts the absolute speedup curves relative to sequential runtime for the `sumEulerDist` program, with the original `parList` and the `parListLevel` strategy. Both strategies scale with the number of cores added, and the architecture aware strategy performs better than `parList` beyond 8 cores.

6.4.1.2 Deep Strategy Results

This subsection presents another version of `sumEulerDist` using architecture-aware abstraction functions. The `fun` function of the program presented in Section 6.2.2 can be rewritten to capture the architecture topology, as follows:

```
fun lower upper block = sum(deep$runEval$( parMapLevel
                        findLevel  sumTotient subList))

deep :: NFData a => a -> a
deep  a = deepseq  a a
```

The program uses the `parMapLevel` skeleton defined in Section 6.3.1. Comparing Column 6 of Table 24 and Column 6 of Table 25, we conclude that both `Deep` and `parMapLevel` strategies perform better than the original `parList` strategy and also have similar performance.

6.4.1.3 Summary:

- Both original and architecture-aware strategies have similar sequential overhead.
- The architecture aware approach successfully reduces communication overhead by sending only large tasks to remote cores. This claim can be drawn from the fact that fine grain tasks may lead to communication overhead. Both the original and architecture-aware programs produce the same number of tasks but the architecture-aware programs restricts the execution locations of small tasks. This is supported by the experiment results. The number of messages are reduced by 81.95% on 192 cores (Table 24 and Figure 71).
- Architecture aware strategy scales as the number of cores increases and performs better than the original strategy. It delivers a maximum speedup of 127.78 on 224 cores compared with 12.34 for the original strategy (Table 24).

	Queen									
	Original					Arch. Aware parDistList			Δ Time (%)	Δ Msgs (%)
	GpH-SMP		GpH-GUM							
NoPEs	T(s)	Spd	T(s)	Spd	Msgs	T(s)	Spd	Msgs		
1	45.04	1.02	52.95	0.87	0	58.42	0.79	0	+10.33	-
2	24.26	1.90	35.17	1.31	462	34.21	1.34	696	-2.73	+50.64
3	15.95	2.89	24.83	1.85	678	23.31	1.97	382	-6.12	-43.65
4	12.57	3.67	20.62	2.23	748	20.03	2.29	1854	-2.86	+147.86
5	10.01	4.61	17.33	2.65	1685	15.98	2.87	1240	-7.79	-26.40
6	8.50	5.43	16.05	2.86	1817	15.26	3.01	1265	-4.92	-30.38
7	7.29	6.33	15.09	3.04	2851	14.06	3.27	2353	-6.83	-17.47
8	6.51	7.09	13.54	3.39	1673	12.47	3.68	1043	-7.90	-37.66

Table 26: Runtime Comparison on Shared Memory (`Queen`)

6.4.2 Queen

Having ported GUM implementation to GHC version 6.12, it is worth investigating its efficiency on shared-memory architecture, before investigating the architecture-aware strategies. We compare the `Queen` program on a single node of the Beowulf cluster with two parallel implementations GpH-GUM and GpH-SMP. They both use the same version of GHC compiler (6.12).

6.4.2.1 Shared Memory Results

Table 26 summarises runtimes and speedups of the `Queen` program running on a single node of the Beowulf cluster. The program was measured on 1, 2, 3, 4, ..., 8 cores. The absolute speedup is based on the optimised sequential runtime (45.93s) for problem size 14x14 queens. Columns 2, 4 and 7 of the table report the 1-8 cores parallel runtimes for the parallel implementations. Columns 3, 5 and 8 of the table report the speedups. Columns 6 and 9 of the table report the

messages exchanged between cores using GpH-GUM. Columns 10 and 11 of the table report the difference in time and messages for architecture-aware strategy compared with original strategy.

The sequential efficiency can be determined by comparing the sequential runtime of a program with the runtime of the same program after inserting the parallel coordination on a single core. The sequential efficiency is defined as the additional overhead introduced by parallel functions, on one PE runtime over the sequential runtime, e.g. launching the process elements (PEs). Comparing the runtimes on a single core shows sequential efficiencies of 102% (46.16s/45.04s) for GpH-SMP and 87% (45.93s/52.95s) for GpH-GUM. The runtimes show that, for this program, GpH-SMP remains faster than GpH-GUM for all numbers of cores, by a factor of 2 (13.54s/6.51s) on 8 cores.

Figure 73 and Figure 74 show the runtime and speedup curves for GpH-SMP and GpH-GUM `Queen` implementations. They show that the program under GpH-SMP performs better, yielding a maximum speedup of 7.09 on 8 cores. Comparing columns 3 and 5 of Table 26, GpH-GUM is slower than GpH-SMP, for all numbers of cores. This can be attributed to communication overhead introduced by message passing in GpH-GUM.

6.4.2.2 Distributed Memory Results

As expected, comparing the runtime curves of Figures 73 and Table 27 shows that the `Queen` program is slower in GpH-GUM, as it communicates by using

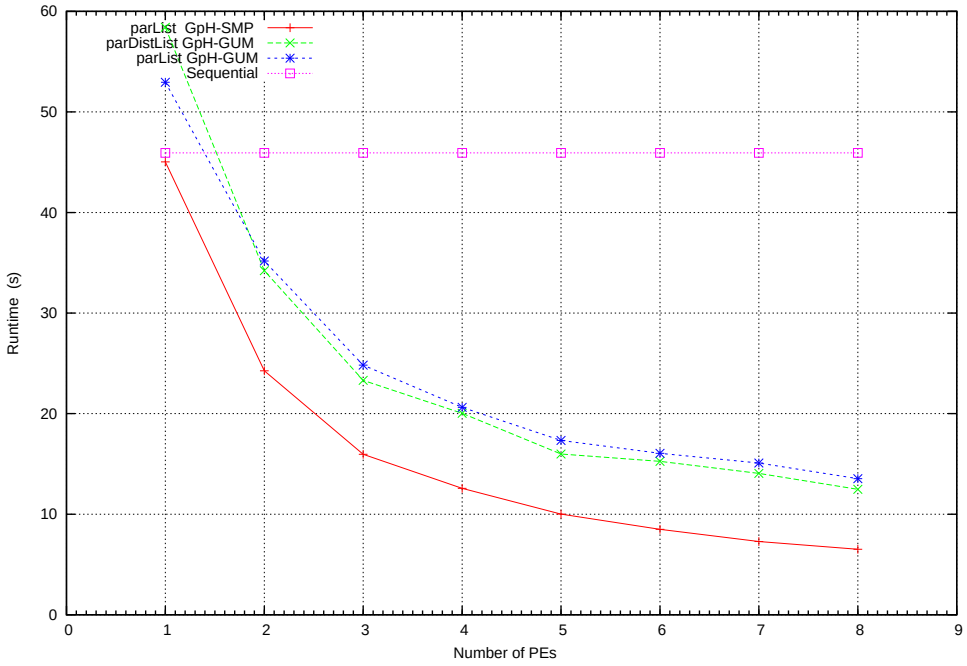


Figure 73: Runtime Comparison on Shared Memory (Queen)

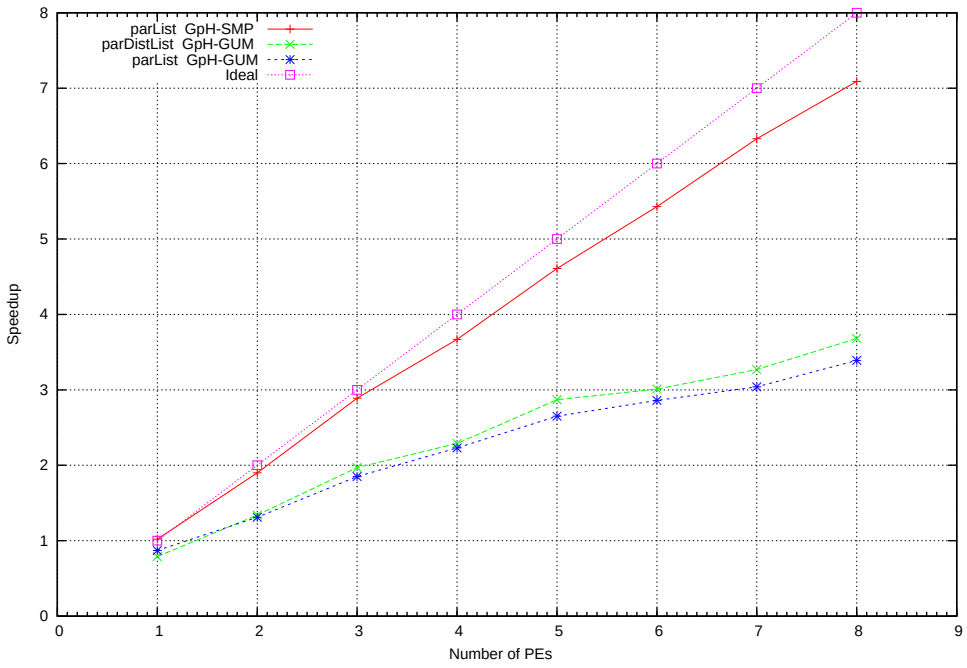


Figure 74: Speedup Comparison on Shared Memory (Queen)

NoPEs	Original parList			Arc. Aware parDistList			Δ Time (%)	Δ Msgs (%)
	T(s)	Spd	Msgs	T(s)	Spd	Msgs		
1	51.52	0.89	0	49.37	0.93	0	-4.17	
2	34.80	1.32	609	40.45	1.14	1383	16.24	127.09
3	27.16	1.69	749	24.78	1.85	234	-8.76	-68.76
4	20.97	2.19	1311	21.42	2.14	1184	2.15	-9.69
5	22.07	2.08	1887	17.21	2.67	808	-22.02	-57.18
6	17.32	2.65	1332	16.00	2.87	1247	-7.62	-6.38
7	17.63	2.61	2018	16.01	2.87	1371	-9.19	-32.06
8	16.84	2.73	2079	16.35	2.81	1796	-2.91	-13.61

Table 27: Runtime Comparison on Distributed Memory (`Queen`)

the expensive message passing library PVM. For example, on eight cores the `parDistList` strategy is slower by a factor of 1.3 (16.35s/12.47s) and by a factor of 1.2 (16.84s/13.54s) for the `parList` strategy on a distributed memory architecture (Table 26 Columns 4 and 7, Table 27 Column 2 and 5). Table 27 also shows that `parDistList` performs better on all numbers of cores. There is an exception, however: on two and four cores, `parList` performs better. We will discuss the circumstances in the following section.

6.4.2.3 Architecture Awareness Comparison

Figures 75 and 76 compare the number of messages of the `Queen` program for the `parDistList` and the `parList` strategy. We observe that `parDistList` requires fewer messages than the `parList` strategy. The only exception is under two and four cores on shared memory architecture, where `parDistList` sends more messages. There are two possible reasons: first, the generated processes do not enough to cover all generated sparks for each level. Second, the lucky

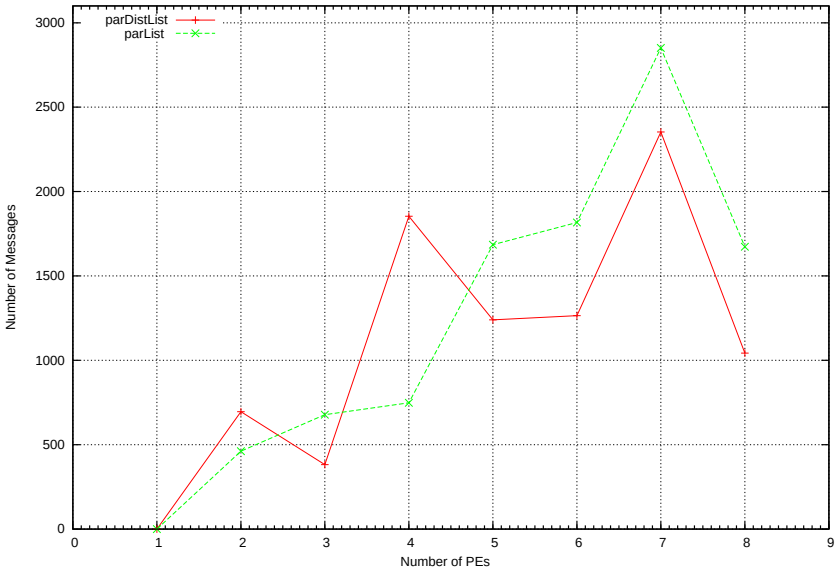


Figure 75: Messages Comparison on Shared Memory (`Queen`)

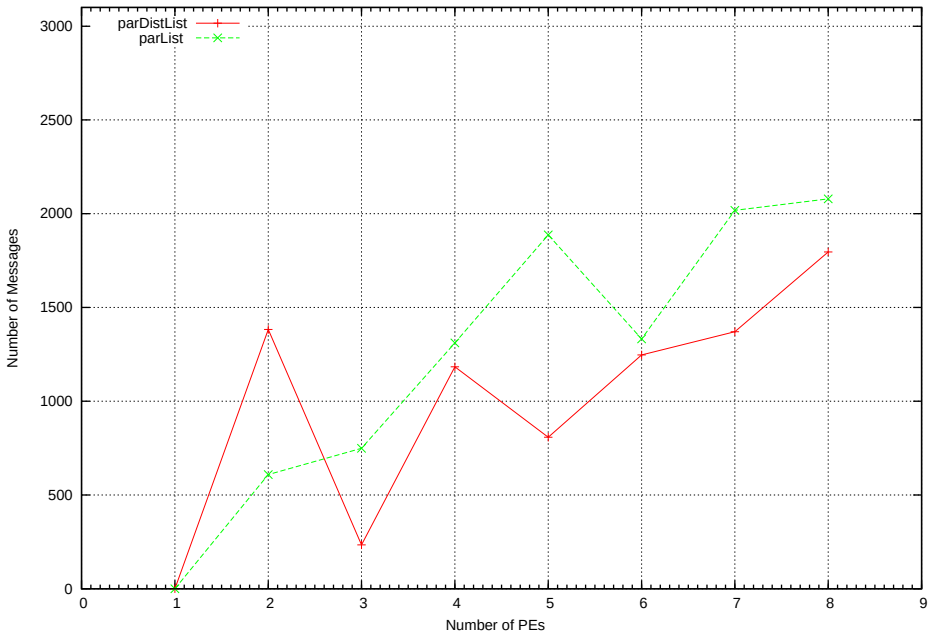


Figure 76: Messages Comparison on Distributed Memory (`Queen`)

scheduling of `parList` sparks, as discussed earlier in Section 5.6.4. Excluding the two cases where `parDistList` sends more messages than `parList`, the difference in messages varies in a range between -17.47% and -43.65% on shared memory architecture (Table 26), and between -6.38 and -68.78 on distributed architecture (Table 27).

Table 27 reports runtimes, speedups and the number of exchanged messages for the `Queen` program executing on eight nodes of the Beowulf cluster, specified in Section 5.6. The results show that while the architecture-aware approach improves performance on most configurations, it degrades it on two; the 2-cores and 4-cores configurations. The performance we have for the `Queen` program for both strategies is modest. Figure 77 plots the speedup curves for the `Queen` program, executed on 8 cores. In the graph, the top line is linear speedup. The second line is the `parDistList` speedup. The third line is the `parList` speedup. The architecture-aware strategy consistently gives a very small performance improvement; ($\sim$ 2%) (2.81s/2.73s) when the program is run on a distributed memory architecture and ($\sim$ 8%) (3.68s/3.39s) when the program is run on a shared memory architecture. The GHC-SMP is faster by factor of 1.8 (6.33s/3.39s) than GpH-GUM, on a shared memory architecture. This is expected, as GHC-SMP does not send any messages for thread communication. We believe that the poor scalability of Queen program is due to the high communication costs of large list, which dominates the total program execution time for larger number of PEs.

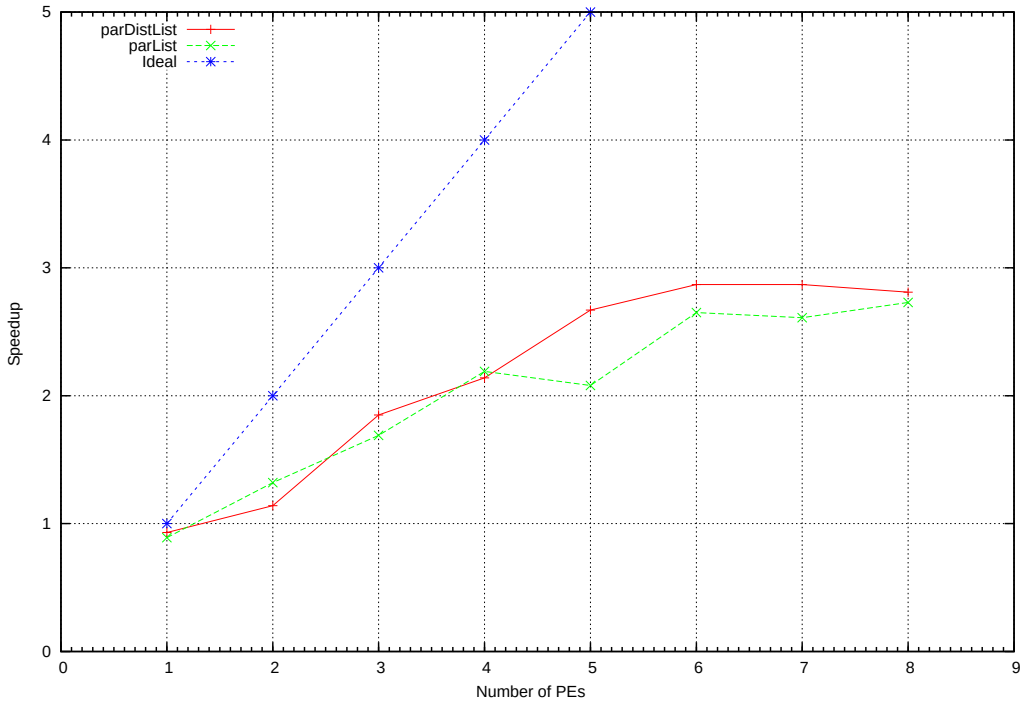


Figure 77: Speedup Comparison on Distributed Memory (Queen)

6.4.2.4 Summary:

- GpH-SMP has a higher sequential efficiency (102%) compared with 87% (45.93s/52.95s) for GpH-GUM. This can be attributed to the differences in launching parallel coordination between the two implementations on single cores.

- GpH-SMP has better speedup, a maximum speedup of 7.09 compared with

3.39 under GpH-GUM implementation on an 8 cores shared memory architecture. This can be attributed to the communication overhead introduced by message passing in GpH-GUM (Table 26).

- Adding architecture awareness to `Queen` consistently gives a small performance improvement on both shared-memory and distributed-memory architectures, i.e. ∼8% and ∼2%, respectively.

6.5 Evaluation of the Architecture Aware Skeletons

This section evaluates the performance of the divide and conquer skeleton presented in section 6.3. It reports performance results of two programs: the `sumEulerSkel` program, which is an artificial skeleton program that computes the sum of the value of Euler's totient function, and a highly parallel `Coins` program that computes the number of ways of paying the given value from a given set of coins. The complete code for the programs can be found in Appendices C.2 and C.3.

6.5.1 sumEulerSkel Results

In this section we investigate a divide-and-conquer skeleton using the `sumEulerSkel` program measured on 1, 2, 4, 8, 16, ..., and 224 cores. The `sumEulerSkel` program

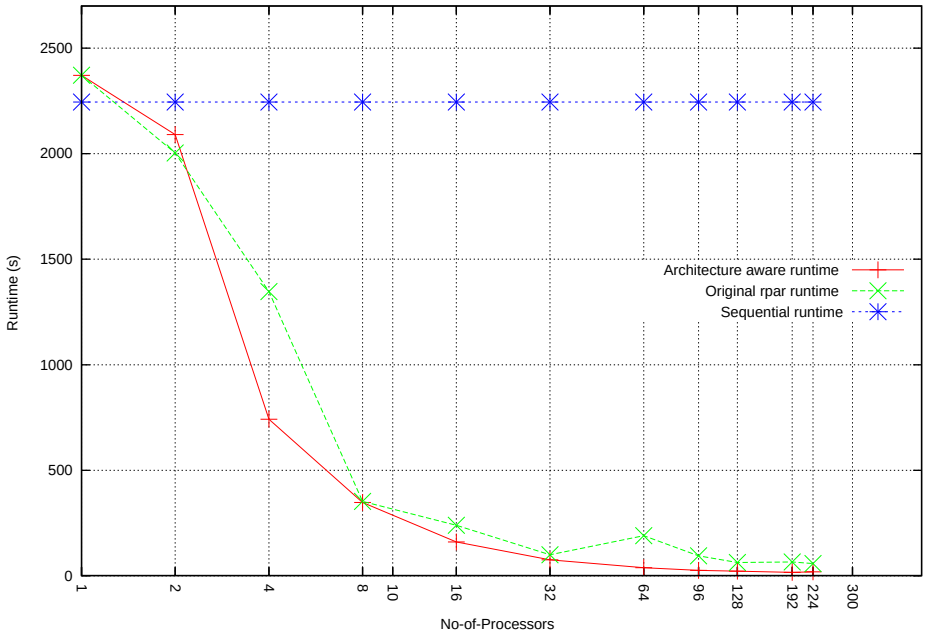


Figure 78: Arch. Aware vs Orig. Skeleton Runtime Comparison (sumEulerSkel)

is listed in Appendix C.2, and uses the divide-and-conquer skeleton presented in section 6.3. The program computes the sum of the value of Euler's totient function. Figure 78 shows the runtime curves for the skeletons from Section 6.3, using the original `rpar` and `rparDist` strategy. Both skeletons produce the same number of sparks (1392) on a single core, so we do not expect any difference in sequential overhead. The runtime curves show very similar performance, for small configurations of up to 8 cores of both constructs. Beyond 8 cores the `rparDist` strategy performs better than the `rpar` strategy, as `rpar` sends more messages as the number of cores increases, shown in Figure 79. This reflects that, with `rpar`,

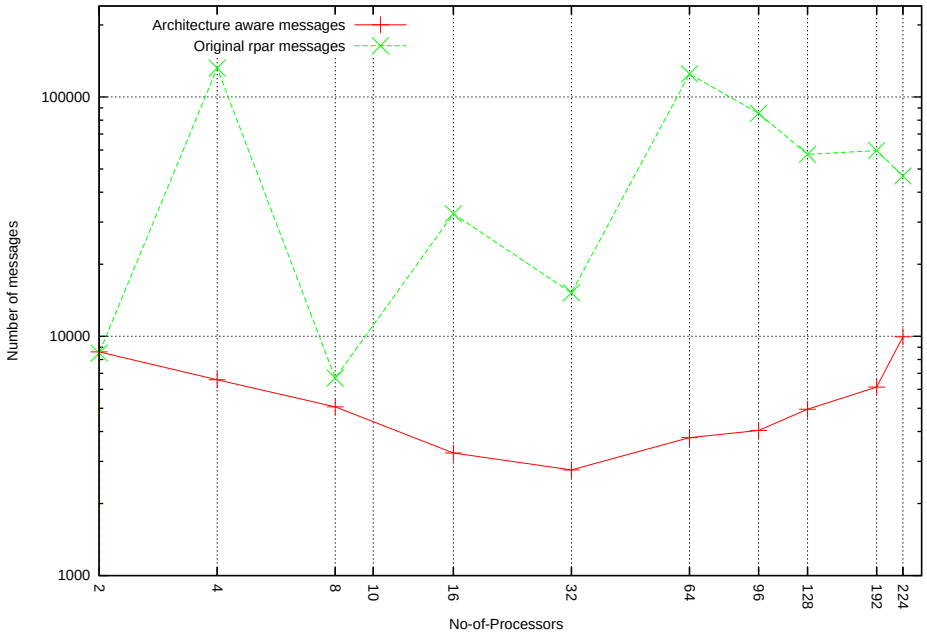


Figure 79: Arch. Aware vs Orig. Skeleton Messages Comparison (sumEulerSkel)

small tasks are executed by remote processors, which increases the possibility for processors to become idle and thus release more messages searching for work.

Table 28 analyses the runtimes, speedups and messages of the skeletons. The results are very similar to those obtained from using the `parListLevel` strategy in section 6.4.1. As the number of cores increases, the difference in the amount of exchanged messages becomes bigger. This gives further confidence that architecture awareness is a better direction for exploiting hierarchical architectures. For example, on 192 cores, the number of released messages falls by 89.73% for the architecture-aware skeleton, with respect to the `rpar` skeleton. The only

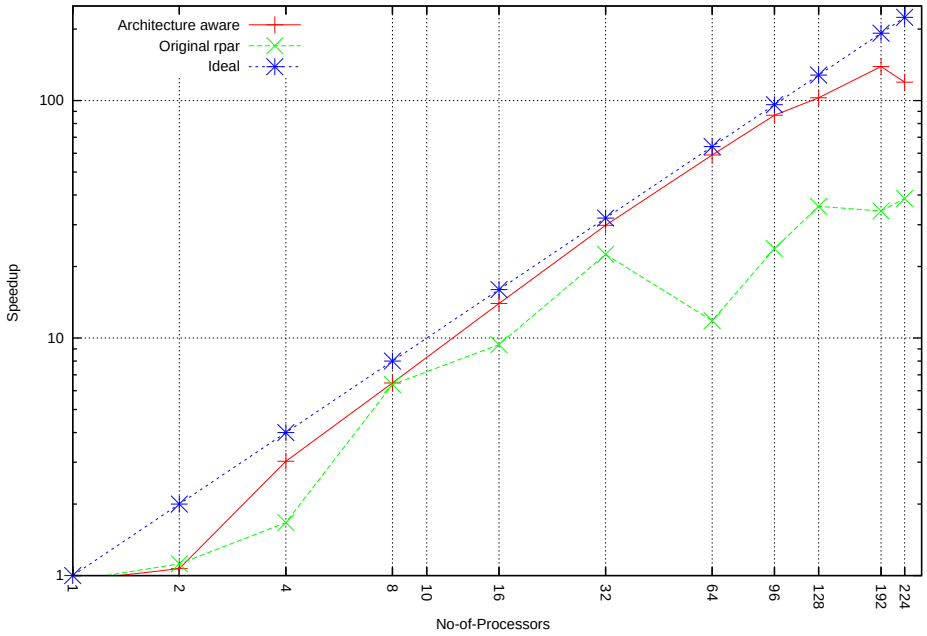


Figure 80: Arch. Aware vs Orig. Skeleton Speedup Comparison (sumEulerSkel)

exception is on two cores, where the number of messages is slightly higher for the architecture-aware program. We again believe that there are not enough PVM processes available for each level to pick up the generated sparks. For the two cores configuration, the system selects one core from two different multicore nodes, which means that only the two level sparks will be picked by the remote core.

For instance, on 192 cores the `rpar` skeleton sends and receives 59716 `Fish` messages, compared with 6134 `Fish` messages, sent by the `rparDist` skeleton. The number of `Fish` messages for the `rpar` is higher by a factor of 9.74 (59709/6130)

	`sumEulerSkel`							
	Original `rpar` Skeleton			Arch. Aware `rparDist` Skeleton			Δ Time	Δ Msgs
NoPEs	Time(s)	Spd	Msgs	Time (s)	Spd	Msgs	(%)	(%)
1	2371.66	0.95	0	2371.03	0.95	0	-0.03	
2	2003.17	1.12	8507	2090.41	1.07	8605	4.36	1.15
4	1346.81	1.67	132131	742.10	3.03	6584	-44.90	-95.02
8	352.42	6.37	6692	347.22	6.47	5069	-1.48	-24.25
16	239.43	9.38	32475	160.50	13.99	3252	-32.97	-89.99
32	99.76	22.50	15187	75.29	29.82	2763	-24.53	-81.81
64	189.76	11.83	124837	38.05	59.00	3769	-79.95	-96.98
96	94.42	23.78	85460	25.90	86.68	4040	-72.57	-95.27
128	62.64	35.84	57600	21.86	102.70	4957	-65.10	-91.39
192	65.57	34.24	59716	16.15	139.01	6134	-75.37	-89.73
224	58.04	38.68	46750	18.79	119.48	9954	-67.63	-78.71

Table 28: Comparison of Divide-and-Conquer Skeleton (`sumEulerSkel`)

than `rparDist Fish` messages, indicating that cores spend more time seeking for work.

The speedup curves reflect the runtime curves. Figure 80 shows speedups curves for both `rparDist` and `rpar`. Both strategies scale as the number of cores increases. The `rparDist` performs better than `rpar` beyond 4 cores.

Summary:

- Adding architecture awareness to a general divide-and-conquer skeleton used in the `sumEulerSkel` program consistently delivers better performance, i.e. the maximum speedup is 139.01 for the architecture-aware program compared with 34.24 for the original program on 192 cores (Table 28).

- The number of messages for the `rpar` is higher by a factor of 9.7 (59709/6130) than `rparDist Fish` messages. This is reflected in the runtime and speedup

	Coins							
	`rpar` Skeleton			`rparDist` Skeleton			Δ Time	Δ Msgs
NoPEs	Time(s)	Spd	Msgs	Time (s)	Spd	Msgs	(%)	(%)
1	633.23	0.65	0	637.22	0.65	0	0.63	–
2	355.70	1.16	84	325.52	1.27	76	-8.48	-9.52
4	231.66	1.78	193	206.97	1.99	223	-10.66	15.54
8	138.15	2.99	331	123.25	3.35	371	-10.79	12.08
16	171.60	2.40	686	101.93	4.05	535	-40.60	-22.01
32	85.87	4.80	796	89.97	4.58	815	4.77	2.39
64	103.06	4.00	1313	76.79	5.37	872	-25.49	-33.59
96	125.56	3.28	1800	64.40	6.40	1392	-48.71	-22.67
128	85.94	4.80	1725	62.78	6.57	1484	-26.95	-13.97
192	85.95	4.80	2261	64.72	6.37	1917	-24.70	-15.21
224	88.17	4.68	2037	64.19	6.43	2002	-27.20	-1.72

Table 29: Comparison of Divide-and-Conquer Skeleton (Coins)

curves, where the `rparDist` strategy is faster than `rpar` (Table 28, Figure 79).

6.5.2 Coins Results

This section investigates the performance of the architecture-aware skeleton defined in Section 6.3.2. We reported performance results for an artificial program in Section 6.5.1, now we report results for the highly parallel `Coins` program. The program is the computational kernel of a realistic application that uses the thresholding technique to limit the amount of parallelism generated and increases thread granularity.

Table 29 shows that the architecture-aware skeleton scores modest improvement against the `rpar` skeleton of the `Coins` program. For instance, the improvement in execution time is between 8.48% and 40.60%. This is again because

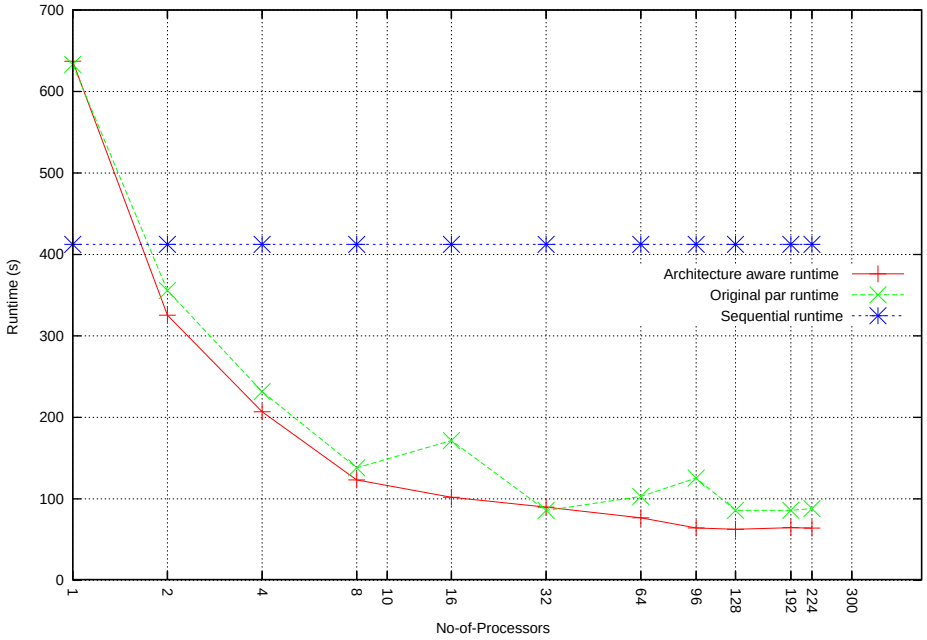


Figure 81: Runtime Architecture Aware Skeleton Comparison (Coins)

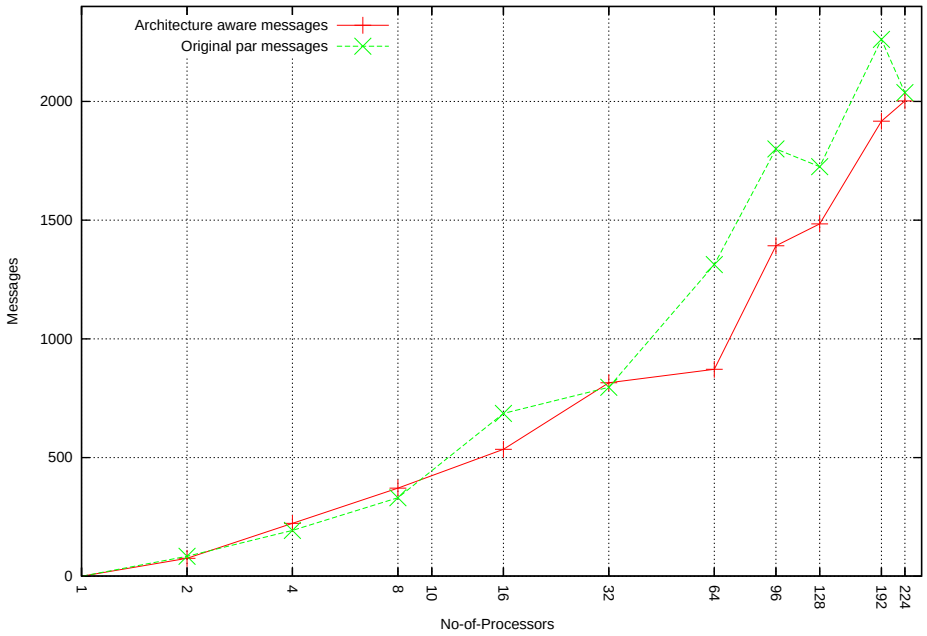


Figure 82: Messages Architecture Aware Skeleton Comparison (Coins)

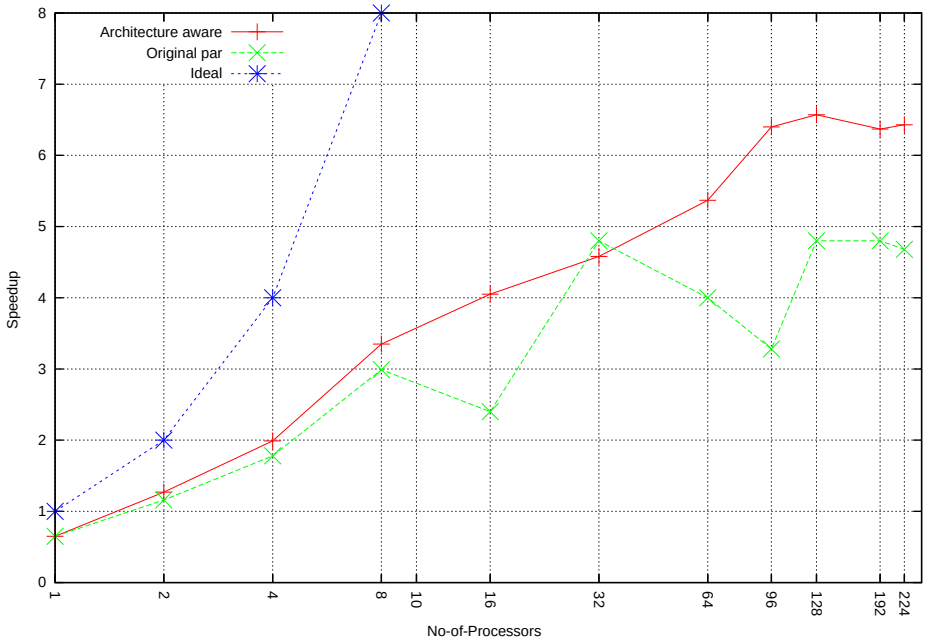


Figure 83: Speedup Architecture Aware Skeleton Comparison (Coins)

the architecture-aware skeleton successfully localises the small tasks and reduces the number of exchanged messages. The reduction in the amount of messages is between 1.72% on 224 cores and 33.59% on 64 cores. Of course, there is an exception: on 4 and 8 cores the number of exchanged messages is increased for architecture-aware skeleton by 15.54% and 12.04% respectively. We believe this is due to the random scheduling mechanism implemented GUM. In these two cases, the lucky scheduling of `rpar` sparks is the cause of `rpar` sending fewer messages than `rparDist`.

Summary: Adding architecture awareness to a general divide-and-conquer skeleton used in the `Coins` program consistently delivers better performance. Figures 81, 82, and 83 show the runtime, speedup and the number of exchanged messages for `rparDist` and `rpar` skeleton `Coins` program. The speedup curves show that `rparDist` delivers better speedup. It delivers a maximum speedup of 6.57 on 128 cores, compared with 4.80 delivered by the `rpar` on the same number of cores(Table 28). The `rpar` strategy has a higher number of messages 1725 compared with 1484 for the `rparDist` strategy. This is reflected in the runtime and speedup curves, where the `rparDist` strategy is faster than the `rpar` strategy (Table 81, Figure 83). Of course, such speedup is poor on 128 cores. We attribute this to instability of the GUM implementation. We believe better performance can be obtained when the system becomes stable as discussed in Section 1.1. In general, the architecture-aware approach exchanges fewer messages as shown in Figures 82.

6.6 Memory and Potential Parallelism Performance

This section is a preliminary investigation into the effect of architecture-aware programming on memory consumption and on the amount of parallelism generated. The most significant improvements are due to preventing small tasks from

being executed far from their originator and the dramatic reduction in communication overheads. The investigation uses four programs.

Name	Speedup		Generated Sparks		Converted Sparks		Allocated Heap		Maximum Residency		Messages	
	Orig.	Arch.	Orig.	Arch.	Orig	Arch. Δ%	Orig. (MB)	Arch. Δ%	Orig. (KB)	Arch. Δ%	Orig.	Arch. Δ%
SumEulerDist	9.38	13.99	1392	1392	688	1.16	5296	148.49	28144	0.09	32475	-89.99
Coins	2.40	4.05	1367	1367	19	247.37	37661	6.10	44032	-0.04	686	-22.01
Queen	3.01	3.26	16883	16632	11185	147.81	8677	-66.05	491	-1.12	2663	-29.89
Fib	6.70	8.74	49853	8019	91	68.13	11,188	-81.99	72,272	-28.57	269	24.91
Geom. Mean	4.62	6.34				205.14		-59.82		-10.95		-11.94

Table 30: Speedups, Number of Sparks and Memory Consumption on 16 cores

Table 30 summarises the speedups, number of sparks and memory consumption of four programs, running on 16 cores, with the original strategies and new architectures aware strategies. The results show that the architecture-aware programs convert more sparks than the original construct, e.g. the geometric mean is +205.14% more. This reflects that, with `par`, large sparks may be picked by any PE, including the main PE, during the early stage of the program execution. This means that only small sparks remain for remote PEs. However, in the architecture-aware case, large sparks are kept in the sparkpool until they are picked by remote PEs.

The results show that the mean of heap residency of architecture-aware programs is smaller by 10.95% compared with the original strategies. This reflects the fact that the architecture-aware programs have fewer active threads. The comparison of exchanged messages demonstrates that the new approach substantially reduces the communication overhead, leading to performance improvement

by a mean factor of 1.38 (6.34/4.62).

6.7 Summary

We have outlined the development of high-level architecture-aware programming constructs that abstract over the new parDist and parExact constructs. In particular, we have developed architecture-aware evaluation strategies (Section 6.2) and architecture aware skeletons (Section 6.3).

We have discussed the key issue of determining threaded granularity and how it can be used to distributed tasks on the architecture's levels (Section 6.1). In this work we assume that the programmer supplies the task granularity information to the program. We seek to find an abstraction like the `findLevel` function to address this issue. The function is used for developing architecture-aware abstraction functions employed by the programs measured in Chapters 5 and 6. For example, the performance of the `parMapLevel` abstraction skeleton developed (Section 6.3.1) is exploited in the programs measured in Section 5.6. The remaining abstraction functions are exploited in Sections 6.4 and 6.5. The results show that architecture-aware strategies can improve the performance of programs dramatically. For example, the `sumEulerDist` program improves by a factor of 10.4 (127.78s/12.34s) on 224 cores.

Measurement of four programs shows that the new abstractions improve heap residency by 10.95% over the original strategies. The new abstractions deliver

performance benefits, for example, the mean improvement in speedup is 1.4 and in the number of exchanged messages, it is -11.94% for the programs measured in Table 30.

Chapter 7

Conclusion

This chapter provides a summary of the work from this thesis, pointing out the main issues investigated and discusses the main achievements and contributions (Section 7.1). It also discusses some limitations of the work conducted in the thesis and suggests avenues of future work (Section 7.2).

7.1 Achievements and Contributions

The objective of this thesis was to develop a parallel programming model exploiting general purpose hierarchical computing architectures using a functional programming language. We proposed a new parallel programming model to exploit such architectures. The model is aware of the abstract communication hierarchy of the architecture and preserves performance portability, as far as possible. The main contributions of this thesis are summarised as follows:

7.1.1 Architecture-aware Constructs

We propose four new architecture-aware constructs for GpH that exploit information about task size and aim to reduce communication for small tasks, to preserve data locality, or to distribute large units of work. They do so by constraining communication abstractly and with as little specific prescription as possible, that is the constructs identify layers of the communication hierarchy, and allow the implementation to dynamically control placement within the layer. The constructs are implemented using a single `parDist` primitive in the GUM runtime system. The constructs aid performance portability by exposing a virtual, rather than physical, architecture and minimising prescription.

The first step to implement the proposed `parDist` primitive was to assist in porting `GUM` to a recent GHC compiler (GHC-6.12). In addition to the porting, the GUM runtime system was adapted to store granularity information with each spark, if required. The modifications comprise two kinds of information. The first is static information including the number of the PE and the distances from the PE to other PEs. The second is dynamic information including the minimum and maximum distance for each spark (Section 5.5).

We also outlined how the approach proposed in this thesis can be implemented in other languages that include task creation. A process in Erlang is spawned on a named node. We believe there is a possibility to build an abstraction layer over this process directive. The abstraction layer maintains a tree of node groups,

loosely modelling the underlying architecture (Sections 5.2 and 5.7).

We have investigated the constructs using three common paradigms, data parallelism, divide-and-conquer parallelism, and nested parallelism, on hierarchical architectures with up to 224 cores. The results obtained for the architecture-aware constructs are consistent for all three programming paradigms investigated. This also shows that the new constructs consistently deliver better speedup and scalability than existing primitives together with dramatic reduction in the execution time variability (Section 5.6). A preliminary version of these results has been published in [9]. We have demonstrated high-level architecture-aware programming abstractions that abstract over the new constructs. In particular, we have demonstrated architecture-aware evaluation strategies and architecture aware skeletons (Chapter 6). We have discussed the key issues that must be addressed in the proposed architecture-aware programming model: the identification of the architecture hierarchy levels, the identification of the thread granularity of each task, the mapping of threads onto levels, and the architecture-aware abstractions that allow the programmer to exploit the hierarchical architectures. The experimental results show that the new abstractions deliver performance benefits over the original constructs (Table 24 page 195). The architecture-aware approach is successfully decreasing the number of messages by sending only large tasks to remote cores (Section 6.4 and Section 6.5).

7.1.2 Programming and Performance Comparison:

We have demonstrated the first programming and performance comparison of four functional multicore technologies and report some of the first ever multicore results for three functional languages of the field FDIP, Eden and GpH. The comparison has been made between a purely implicit parallel language and three semi-explicit languages. The comparison reflects the current state of the technology and compares the programming effort each variant requires with the parallel performance delivered. The experimental results show that semi-explicit languages are a better choice for exploiting hierarchical architectures. A purely implicit parallelism remains an elusive goal as the FDIP's purely implicit approach improves the fewest number of programs. However, the semi-explicit approaches GpH-SMP, GpH-GUM, and Eden improve approximately half of the programs (Tables 9, 10, and 11 pages 82, 83, and 84).

We have presented a semantics for the architecture-aware constructs, specifying the set of possible locations when providing boundaries to the sparks. We have shown the expected behaviour of constructs, using Haskell functions. We have formulated several properties as Haskell predicates and used `Quickcheck` to check them on random input. The three basic properties represent a sanity check of the semantics. Two proposed implementation relevent properties did not hold, and counter examples extracted from `Quickcheck` identified *diffusion of sparks* to be the problem. This observation justifies our implementation, where

we avoided this problem by resetting the boundaries after one fishing stage. The final property checked with `Quickcheck` shows that with this modification, the desired property holds (Section 5.4.5).

Finally, we have investigated thread granularity on multicore architecture using three different GHC parallel Haskell implementations: GHC-GUM, Eden and GHC-SMP. GHC-SMP is a purely shared memory module that does not perform any communication between PEs other than exchange pointers. GHC-GUM uses message passing to communicate between PEs, as well as maintaining the shared heap between them. Eden uses message passing to communicate between PEs through an explicit channel. We have suggested the most profitable thread granularity for the three implementations, GHC-GUM, Eden, and GHC-SMP in case of programs similar to our benchmark are used.

7.2 Limitations and Future Work

There are a number of limitations to this thesis' investigation of an architecture-aware programming model. This section discusses some of these limitations and possible future work.

System stability. The current GUM implementation used for the measurements is not yet stable, i.e. the current GUM implementation fails for programs with large data structures. Therefore, the performance evaluation of the model is illustrated by a limited number of programs that expose different parallel

paradigms. Further work on stabilising the GUM implementation is essential to evaluate the architecture aware approach with a range of benchmarks and better evaluate its effectiveness.

Investigated architectures. The benchmarks were measured only on one hierarchical architecture as the current architecture-awareness was implemented using GUM, which works on both shared memory and distributed memory architectures. We have made some limited measurements on a small shared memory architecture (Section 6.4.2) and the results are encouraging. It would be valuable to investigate the model on a range of other hierarchical architectures including larger scale architectures. In particular, GPUs (Graphics Processing Unit) are becoming increasingly available for general purpose programming. GPUs provide another level to multicore machines and open a new field for investigating the architecture-awareness of the model.

An automatic thread granularity system. We have discussed how a key issue to achieve better performance is the identification of the thread size and the determination of in what architecture level, where it should be performed. The current architecture-aware approach depends on the explicit insertion of the required information into the program. It would be valuable to extend the current system to automatically determine the thread granularity of tasks, the execution level of a computation of given size. This can be done by some resource analysis and program profiling [120, 115, 116], maybe following the FDIP implementation

which uses the profiling technique for identifying the dependency, and automatically inserting parallelism into the program. There are also emerging analyses for such hierarchical architectures, like Multi-BSP [123] or SGL [63].

High-level abstraction. We presented some architecture-aware abstractions like strategies and skeletons (Sections 6.2 and 6.3). The avenue remaining open for future work is to develop a fuller set of architecture-aware high-level abstractions. In particular, for developing a complete evaluation strategies model and a complete set of skeletons.

Architecture-awareness in other languages. As described in Section 5.7, the architecture-aware approach could be implemented in any language with task creation. The Erlang distribution model permits explicit process placement [61]. A process in Erlang is spawned on a named node. Such a static, directive mechanism is hard for programmers to manage for anything other than small scale or very regular process networks. RELEASE is an EU FP7 STREP (287510) project which has already started looking at using the idea of this thesis to build general-purpose software (Scalable Distributed (SD) Erlang) [119]. This idea is also implemented in Haskell distributed parallel Haskell (HdpH) as part of SymGrid-Par2 in the HPC-GAP project [76].

Appendix A

Benchmark Code

This appendix presents the complete code for the benchmark programs measured in Section 5.6.2. The `parMapList` program is presented in Section A.1. Section A.2 presents only the `parMapIntervals` functions that are different from the `parMapList` program.

A.1 `parMapList` Program

The `parMapList` program splits the list into sublists of random sizes and calculates each sublist in parallel. The `findLevel` function is presented in five different definitions, reflecting all possible configurations, as described in table 21.

```
module Main where

import System(getArgs)
import Random
import Control.Parallel
import Control.Parallel.Strategies
```

```
main = do args <- getArgs
   let
     lower = read (args!!0) :: Int    -- Get  lower value
     upper = read (args!!1) :: Int    -- Get  upper value
     seed = read (args!!2) :: Int     --  Get seed
     res = dataListtop  lower upper seed
   putStrLn ("sumEuler [1.." ++(show upper)++ "] = "++(show res ))

dataListtop lower upper seed  =
         let
           cs0 = mkRandom seed -- Generated random blocks list
           dataList = sumTotient lower upper cs0
          in
           dataList

sumTotient :: Int -> Int -> [Int]-> Int
sumTotient lower upper bs = sum (parMapLevel (faa euler)
                                 findLevel (xs1))
     where
       xs1 = splitWithsize bs [lower..upper]
       faa f ys =sum(map  f ys)


parMapLevel  f findLevel [] =  []
parMapLevel f findLevel (x:xs) = parDist min max fx
                                 (fxs `pseq` ( fx : fxs))
            where
             (min , max)= findLevel x
             fx = f x
             fxs =parMapLevel f findLevel  xs

--  splitWithsize splits alist into sublists
splitWithsize::  [Int] -> [Int]-> [[Int]]
splitWithsize   _  [] = []
splitWithsize    (b:bs) xs = xss
         where
          xss = (take b xs):splitWithsize bs (drop b xs)

mkRandom  n  =
         let
          g = mkStdGen 1601
          cs :: [Int]
          cs = randoms  g
          cs0 = map  (`mod` n) \$ cs
```

```
          in
           cs0

euler :: Int -> Int
euler n = let
            relPrimes = let
                          numbers = [1..(n-1)]
                        in
                          (filter (relprime n) numbers)
          in
            (length relPrimes)

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

#if defined(PARBOUND)
#warning "parBound compilation "

findLevel :: (Ord a,Num a )=> [a]-> (a,a)
findLevel  x
           | b < 300 = (0,0)
           | (b >= 300 && b<=600) = (0,1)
           | (b >= 600 && b<=800)= (0,2)
           | otherwise = (0,3)
           where
            b = length x
#elif defined(PARN_1)
#warning " parDist  N-1 compilation"
findLevel :: (Ord a,Num a )=> [a]-> (a,a)
findLevel  x
           | b < 300 = (0,0)
           | (b >= 300 && b<=600) = (0,1)
           | (b >= 600 && b<=800)= (1,2)
           | otherwise = (2,3)
           where
              b = length x
#elif defined(PARN_2)
#warning "parDist N-2 compliation "
findLevel :: (Ord a,Num a )=> [a]-> (a,a)
findLevel  x
           | b < 300 = (0,0)
           | (b >= 300 && b<=600) = (0,1)
```

```
            | (b >= 600 && b<=800)= (0,2)
            | otherwise = (1,3)
            where
             b =length x

#elif defined(PARATLEAST)
#warning " parAtLeast compilation "
findLevel :: (Ord a,Num a )=> [a]-> (a,a)
findLevel  x
            | b < 300 = (0,3)
            | (b >= 300 && b<=600) = (1,3)
            | (b >= 600 && b<=800)= (2,3)
            | otherwise = (3,3)
            where
             b = length x
#elif defined(PAREXACT)
#warning "parExact compilation "

findLevel :: (Ord a,Num a )=> [a] -> (a,a)
findLevel  x
            | b < 300 = (0,0)
            | (b >= 300 && b<=600) = (1,1)
            | (b >= 600 && b<=800)= (2,2)
            | otherwise = (3,3)
            where
             b = length x
#endif
```

A.2 parMapIntervals Program

The parMapIntervals program splits the interval into subintervals of random sizes and calculates subintervals in parallel. The difference between the two programs is described in Section 5.6.2.

```
splitIntervals ::(Ord a,Num a )=> (a,a) -> [a] -> [(a,a)]
splitIntervals (lower,upper) (b:bs)
        | ((upper-b-1) <= lower) =[(lower,upper)]
        | otherwise = ((upper-b),upper):splitIntervals
                      (lower,(upper-b-1)) bs

sumTotient :: (Int , Int) -> Int
sumTotient (lower,upper) = sum (map euler [lower,lower+1..upper])
```

Appendix B

Location Semantics of Architecture-Aware Constructs

This Appendix presents a simple Haskell specification of the sets of PEs that each construct identifies when executed on a PE in some hierarchy of PEs. The program defines a tree data structure representing the architecture. Leaves in the tree represent PEs and nodes represent network connections (Figure 36). Section 5.4 describes how the program works and gives example output.

```
module Main(main) where

import System
import Debug.Trace
import List
import QuickCheck
import System.Random
import Control.Monad

data Tree a = EmptyTree
            | Leaf a
            | Node a (Tree a) (Tree a) deriving (Eq ,Show)
```

```
insertT :: (Ord a,Num a) => a -> Tree a -> Tree a
insertT x   EmptyTree  =  (Node x EmptyTree  EmptyTree )
insertT x  (Node a EmptyTree  EmptyTree )
    | x < a = (Node x (Node x EmptyTree  EmptyTree)
                EmptyTree)
    | otherwise = (Node x   EmptyTree  (Node x EmptyTree
                   EmptyTree))

insertT x (Node a  left  right)
    | x < a = Node a (insertT x left)  right
    | x > a = Node a left  (insertT x right)

    | otherwise = Node x left  right


mkRandom  n  =
         let
          g = mkStdGen 1601
          cs :: [Int]
          cs = randoms  g
          cs0 = map  (`mod` n) $ cs

          in
           cs0

buildTree :: Int -> Tree Int -> Tree Int
buildTree n EmptyTree =
         buildTree (n-1) (Node n  EmptyTree  EmptyTree)
buildTree n (Node a  EmptyTree  EmptyTree) =
             buildTree (n-1)(Node a (Node n EmptyTree
             EmptyTree) EmptyTree)
buildTree n (Node a l  EmptyTree) =
             buildTree (n-1) (Node a l (Node n EmptyTree
             EmptyTree))
buildTree n (Node x l r) = setTree n (Node x l r)
   where

   setTree n (Node x l r)
        | (n<=0) =  (Node x l r)
        |((n>0) && (even n))= setTree (n-1)(Node x left r)
        |((n>0) &&(odd  n))= setTree (n-1)(Node x l right)
        | otherwise = (Node x l r)
```

```
      where
        left = insertT n l
        right = insertT n r



fillTree :: Int -> Tree Int
fillTree   n   =Node n left right
          where
             t = EmptyTree
             depth = mkRandom n
             left = buildTree (head depth)  t
             right = buildTree (head(drop 1 depth)) t

--  The rLeaf returns all leaves in the tree
rLeaf ::  (Tree Int) -> [Int]
rLeaf ( (Leaf p)) =  [p]
rLeaf (Node v EmptyTree  EmptyTree)= []
rLeaf (  (Node v EmptyTree u ))  =  rLeaf ( u)
rLeaf (  (Node v  l EmptyTree  ))  =  rLeaf ( l)
rLeaf ( (Node v t u)) = rLeaf (t) ++ rLeaf ( u)

--  The insertLeaf function  fill random leaves to thtree
insertLeaf (x:xs)  EmptyTree = Leaf x
insertLeaf (x:y:xs) (Node v EmptyTree EmptyTree)=
                              (Node v (Leaf x) (Leaf y))
insertLeaf (x:xs) (Node v EmptyTree u) =
                              (Node v (Leaf x)(insertLeaf xs u))
insertLeaf (x:xs) (Node v l EmptyTree) =
                              (Node v (insertLeaf xs l)(Leaf x))
insertLeaf xs (Node v l u) = (Node v (insertLeaf lx l )
                              (insertLeaf rx u))
      where
        lx = take (length(xs) `div` 2 ) xs
        rx = drop (length(xs) `div` 2 ) xs

--  The setparDist  Haskell  implemntation of parDist
setparDist :: Tree Int -> Int -> Int -> Int -> [Int]
setparDist t m u p
            | ((m<0) || (u<0)) = []
            | ( m >  (length (pp))) = []
            | ((m==0) && (u==0))= [p]
            | (u > (length (pp)-1)&& (m==0))  = (rLeaf ( t))
            | ((m==u) || (u > (length (pp)-1))) = exact(subexact) p
            | m==0 =filter (/= commonnm) ([p]++setPes)
            | otherwise = setPes
```

```
          where
          pp = path  t p
          commonnu = last (take (length (pp) - u) pp)
          commonnm =last ( take (length (pp) - m) pp)
          subu=subTree t commonnu
          subexact= subTree t commonnm
          complementtree =   (complementTree subu commonnm)
          setPes= filter (/= commonnm) (rLeaf (complementtree))

distance :: Tree Int -> Int -> Int -> Int
distance  t  p1 p2  =  d1+d2
          where
           pathTop1 = path t p1
           pathTop2 = path t p2
           comNodes =    length (prefixOf pathTop1 pathTop2)
           d1 = length pathTop1 - comNodes
           d2 = length pathTop2 - comNodes


prefixOf [] _          =  []
prefixOf _  []         =  []
prefixOf (x:xs) (y:ys)
         |   x /= y  = []
         | otherwise  =x: prefixOf xs ys


path ::  Tree Int -> Int -> [Int]
path (Leaf p ) s
       | p==s = [s]
       | otherwise = []
path  (Node v t u) s
     | v == s =  [v]
     | left== [] && right  == [] = []
     | left /= [] =   [v]  ++ left
     | right  /= [] = [v] ++ right
     where
      left = path t s
      right = path u s

subTree ::Tree Int -> Int -> ( Tree Int)
subTree (Leaf  p) s =EmptyTree
subTree (Node v t u) s
     | v == s  = (Node v t u)
     | left== EmptyTree && right  == EmptyTree = EmptyTree
     | left /= EmptyTree =  left
     | right  /=EmptyTree =right
```

```
    where
     left =    subTree t s
     right =   subTree u s


complementTree ::  Tree Int -> Int -> Tree Int
complementTree (Node v (Leaf p1) (Leaf p2) ) s
       | v == s  = EmptyTree
       | otherwise = (Node v (Leaf p1)  (Leaf p2) )
complementTree (Node  v (Leaf p1) u) s
       | v == s  =   EmptyTree
       | otherwise = (Node  v (Leaf p1)(complementTree u s))
complementTree (Leaf p1) s = (Leaf p1)
complementTree (Node v EmptyTree EmptyTree) s
       | v==s = EmptyTree
       | otherwise  =(Node v EmptyTree EmptyTree)
complementTree  (Node v EmptyTree u) s
       | v==s = EmptyTree
       | otherwise  = (Node v EmptyTree (complementTree u s))
complementTree  (Node v l EmptyTree ) s
       | v==s = EmptyTree
       | otherwise  = (Node v (complementTree l s) EmptyTree)
complementTree (Node v t u) s
       | v==s = EmptyTree
       | otherwise  = (Node v (complementTree t s)
                       (complementTree u s))



exact (Node v t u )  p
        | (not (elem p zz )) = zz
        | otherwise = yy
        where
          yy = rLeaf( t)
          zz = rLeaf( u)



subset [] ys =trace( show("amrr"))$  False
subset xs []   = True
subset xs (y:ys)
         | (not (y `elem` xs)) = False
         | otherwise=  subset xs ys

--The test1 function is the implementation of  properties (1,2)
test1  h  m u  p p'
```

```
      | (setparDist  h  m u  p == [])= True
      | otherwise = (not(p' `elem` l ))  || (subset  l l')
      where
       l = setparDist  h m u  p
       l'= setparDist  h m u  p'

--The test2 function is the implementation specialised  property (3)
test2  h m u  p p'
      | (setparDist h m u p == [])= True
      | otherwise = (not(p' `elem` l ))  || (subset  l l')
      where
       l = setparDist  h m u  p
       l'= setparDist  h 0 1 p'


randomPE = head( filter (>100) (mkRandom 300))
randomTree n = (insertLeaf [100..300] (fillTree n))

main = do args <- getArgs
        let
          n = read (args!!0) :: Int -- lower limit of the interval
          newtree =   (insertLeaf [100..n+100] (fillTree n))
          --t1 = xlll [1,2,3] [1,4,3,2,5,6,8]
        putStrLn (show("The test of  basic property  (1) "))
        res<-quickCheck( (\s->(setparDist
                  (randomTree n) 0 0 s == [s]))
                  ::Int -> Bool)


        putStrLn (show("The test of  basic property  (2) "))
        res<-quickCheck( (\s->(setparDist (randomTree n)  0
                  ((length (path (randomTree n)  randomPE))+1)
                  randomPE  == (rLeaf ((randomTree n)))))
                       ::Int -> Bool)

        putStrLn (show("The test of  basic property  (3) "))
        res<-quickCheck( (\s->let
                               h =(randomTree n)
                               m = 100
                               u = 200
                              in
                               (setparDist h m u  s  == []))
                               ::Int -> Bool)
        putStrLn (show("The test of  relevent  property  (1,2) "))
        res<- quickCheck((\s->let
                               h =(randomTree n)
```

```
                              p'=head (setparDist h 0 s randomPE)
                             in
                              test1  h  0 s randomPE p')
                                ::Int -> Bool)

        putStrLn (show("The test of  relevent  property  (3) "))
        res<- quickCheck((\s->let
                              h =(randomTree n)
                              p'=head(setparDist h 0 s randomPE)
                             in
                              test2  h  0 s  randomPE p' )
                                ::Int -> Bool)

        putStrLn (show(" Quick check  test is finish"))
```

Appendix C

Architecture-Aware Programs

This Appendix contains the architecture-aware code for the programs measured in Chapter 6. Section 6.4.1.2 presents more results for `sumEulerDist` program from Section 6.4.1.

C.1 sumEulerDist Code

This section presents complete code for `sumEulerDist` program using `parListLevel` strategy measured in Section 6.4.1.

```
module Main where

import System(getArgs)
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq

main = do args <- getArgs
          let
            lower = read (args!!0) :: Int
            upper = read (args!!1) :: Int
```

```
            block = read (args!!2) :: Int --  block size
            result= fun  lower upper block
           putStrLn ("sumEuler [" ++(show lower)++".." ++
                     (show upper) ++"] = " ++ (show result ))

fun lower upper block = sum((map sumTotient subList)
                        `using`  parListLevel cosList rdeepseq)
     where
      subList = splitAtN block [lower ..upper]
      cosList= map findLevel subList

sumTotient :: [Int]-> Int
sumTotient lower = sum (map euler lower )

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
           where (ys,zs) = splitAt n xs

findLevel :: [Int] -> Int
findLevel  (x:xs)
         | (x < 10000)   = 0
         | ((x >= 10000) && (x < 20000))  =1
         | ((x >= 20000) && (x <= 80000))  =2
         |  otherwise =3

euler :: Int -> Int
euler n = let
           relPrimes = let
                        numbers = [1..(n-1)]
                       in
                        (filter (relprime n) numbers)
         in
           (length relPrimes)

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1
```

C.2 sumEulerSkel Program

This Section presents complete code for `sumEulerSkel` program using the architecture-aware divide-and-conquer skeleton measured in Section 6.5.1.

```
module Main where

import System(getArgs)
import Random
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq

main = do args <- getArgs
          let

            lower = read (args!!0) :: Int
            upper = read (args!!1) :: Int
            block = read (args!!2) :: Int
            res = foo lower upper block
          putStrLn ("sumEuler [1.." ++ (show upper)
                   ++ "] = " ++ (show res ))


sumTotient:: [Int] ->  Int
sumTotient xs =sum (map euler xs)


findLevel :: Int -> Int
findLevel  n
         | (n < 10000)    = 0
         | ((n >= 10000) && (n <= 20000))  =1
         | ((n >= 20000) && (n <= 30000))  = 2
         |  otherwise = 3


euler :: Int -> Int
euler n = let
            relPrimes = let
                          numbers = [1..(n-1)]
                        in
                          (filter (relprime n) numbers)
          in
```

```
            (length relPrimes)

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1


divCon ::(Enum b, NFData a) =>
  (b -> b -> a -> Bool)
  -> (b -> Int)
  -> ([b] -> a)
  -> (b -> b -> a -> ((b, b), (b, b)))
  -> (a -> a -> a)
  -> a
  -> (b, b)
  -> a

divCon divisible
       findLevel
       solve
       divide
       combine  block  (lower,upper)
    |(divisible lower upper block ) = solve [lower.. upper]
    | otherwise = runEval $(do
                let
                   (c1,(r1,r2)) = (divide lower upper  block)
                   mx =  findLevel r1
                x' <- ((rparDist mx mx) `dot` rdeepseq)
                      (solve [r1.. r2])
                y' <- ((rparDist mx mx) `dot` rdeepseq)
                      ((divCon divisible findLevel solve
                      divide combine  block ) (c1))
                return( ( combine x' y')))

foo :: Int -> Int -> Int -> Int
foo  lower upper block =divCon  divisible findLevel
                                sumTotient  divide
                                combine  block (lower,upper)

divisible :: (Num a, Ord a) => a -> a -> a -> Bool
divisible lower upper block
      | ((upper-block) <= lower) = True
      |otherwise = False
```

```
divide :: (Num a) => t -> a -> a -> ((t, a), (a, a))
divide lower upper block =  (l,r)
     where
        l = (lower,(upper-block-1))
        r = ((upper-block) , upper)

combine:: Int -> Int -> Int
combine x  y  = x+y
```

C.3 Coins Program

This section presents the complete code for `Coins` program using the architecture-aware divide-and-conquer skeleton measured in Section 6.5.2.

```
module Main where

import System(getArgs)
import Random
import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq

main = do
         let vals = [250, 100, 25, 10, 5, 1]
         let quants = [55, 88, 88, 99, 122, 177]  -- large setup
         let coins  = concat (zipWith replicate quants vals)
             coins1 = zip vals quants
         [n,arg] <- fmap (fmap read) getArgs
         print $  foo n  arg  coins1

findLevel :: Int -> Int
findLevel  x
         | (x < 7) =  2
         | (x== 7) =  1
         | otherwise =0

foo :: Int  -> Int ->[(Int,Int)] -> Int
foo  depth val coins = divCon  divisible findLevel
                       payN  divide combine depth (val,coins)
```

```
divisible:: Int -> Int ->[(Int,Int)] -> Bool
divisible  _ _  [] = True
divisible  depth  val (coins)
    | ( depth <= 1 ) = True  -- False
    | ( val <=0 ) =  True
    | otherwise =  False


divide ::( Int,[(Int, Int)]) ->
         ((Int,[(Int, Int)]), (Int,[(Int, Int)]))
divide (val, ((c1,q1):[])) = (((val-c1),[]),(val,[]))
divide (val, (c1,q1):coins)
        |q1==1  = (((val-c1),coins ) ,(val,coins) )
        |otherwise = (((val-c1),(c1,(q1-1)):coins),(val,coins))

combine:: Int -> Int -> Int
combine x  y  = x+y


divCon divisible findLevel  payN  divide combine  depth  (val,coins)
    |(divisible depth  val coins ) =  payN  val coins -- d
coins
    | otherwise = runEval $(do
              let
                 (xs', xs'') = (divide  (val,coins))
                 (mx) = findLevel depth
              x' <- ((rparDist mx mx) `dot` rdeepseq)
                    (divCon divisible findLevel  payN
                    divide combine   (depth-1)  xs')
              y' <- ((rparDist mx mx) `dot` rdeepseq)
                    (divCon divisible findLevel  payN
                    divide combine  (depth-1) xs'')
              return( combine x' y'))

payN :: Int -> [(Int,Int)] -> Int
payN 0   coins     = 1
payN _   []        = 0
payN val ((c,q):coins)
  | c > val   = payN val coins
  | otherwise = left + right
  where
    left  = payN (val - c) coins'
    right = payN val coins

    coins' | q == 1    = coins
           | otherwise = (c,q-1) : coins
```

Bibliography

[1] Adabala, S., Chadha, V., Chawla, P., Figueiredo, R., Fortes, J., Krsul, I., Matsunaga, A., Tsugawa, M., Zhang, J., Zhao, M., et al. From Virtualized Resources to Virtual Computing Grids: The In-VIGO System. *Future Generation Computer Systems 21*, 6 (2005), pp. 896–909.

[2] Akhter, S., and Roberts, J. *Multi-Core Programming*. No. ISBN:0-9764832-4-6. Intel Press, April 2006.

[3] Al Zain, A., Hammond, K., Berthold, J., Trinder, P., Michaelson, G., and Aswad, M. Low-pain, High-gain Multicore Programming in Haskell: Coordinating Irregular Symbolic Computations on Multicore Architectures. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming* (2009), ACM, pp. 25–36.

[4] Alpern, A., Cater, L., and Ferrante, J. Modeling Parallel Computers as Memory Hierarchies. *IEEE* (1993), pp. 116–123.

[5] Anand, C., and Kahl, W. Synthesising and Verifying Multi-Core Parallelism in Categories of Nested Code Graphs. Tech. Rep. SQRL No. 50, Department of Computing and Software, McMaster University, January 2008.

[6] Andrade, D., Fraguela, B., Brodman, J., and Padua, D. Task-parallel Versus Data-Parallel Library-Based Programming in Multicore Systems. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on* (2009), IEEE, pp. 101–110.

[7] Aswad, M. Multi-Architecture Parallel Programming Using GpH, a Functional Language. Master's thesis, School of Mathematics and Computer Science,Heriot-Watt University., 2002.

[8] Aswad, M., Trinder, P., Al Zain, A., Michaelson, G., and Berthold, J. Low Pain vs No Pain Multi-core Haskells. *Symposium on Trends in Functional Programming (TFP09) 10* (June 2009), pp. 49–63.

[9] Aswad, M., Trinder, P. W., and Loidl, H. Architecture-Aware Parallel Programming in Glasgow Parallel Haskell (GPH). In *Proceedings of the International Conference on Computational Science (ICCS)* (Omaha, USA, June 2012), Procedia Computer Science, pp. 1807–1816.

[10] Augustsson, L., and Johnsson, T. Parallel Graph Reduction with the (ν, G)-Machine. In *Proceedings of the Fourth International Conference*

on Functional Programming Languages and Computer Architecture (1989), ACM, pp. 202–213.

[11] Baker, M., Buyya, R., and Laforenza, D. Grids and Grid Technologies for Wide-area Distributed Computing. *Software: Practice and Experience 32*, 15 (2002), pp. 1437–1466.

[12] Barroso, L., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *ACM SIGARCH Computer Architecture News* (2000), vol. 28, ACM, pp. 282–293.

[13] Berthold, J., Dieterle, M., Loogen, R., and Priebe, S. Hierarchical Master-worker Skeletons. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages* (2008), Springer-Verlag, pp. 248–264.

[14] Berthold, J., Klusik, U., Loogen, R., Priebe, S., and Weskamp, N. High-level Process Control in Eden. *Euro-Par 2003 Parallel Processing* (2004), pp. 732–741.

[15] Berthold, J., and Loogen, R. Parallel Coordination Made Explicit in a Functional Setting. *Implementation and Application of Functional Languages* (2007), pp. 73–90.

[16] Berthold, J., Marlow, S., Al Zain, A., , and Hammond, K. Comparing and Optimising Parallel Haskell Implementations on Multicores. In *IFL'08 International Symposia on Implementation and Application of Functional Language , Inproceeding* (Hatfield, UK, 2008), pp. 386–393.

[17] Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B., Garzarán, M., Padua, D., and Von Praun, C. Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (2006), ACM, pp. pp. 48–57.

[18] Bischof, H., Gorlatch, S., and Kitzelmann, E. Cost Optimality and Predictability of Parallel Programming with Skeletons. *Parallel Processing Letters 13*, 4 (2003), pp. 575–587.

[19] Bisseling, R. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. No. ISBN:0190829392. Oxford University Press, USA, 2004.

[20] Blake, G., Dreslinski, R., and Mudge, T. A Survey of Multicore Processors. *IEEE, Signal Processing Magazine 26*, 6 (2009), pp. 26–37.

[21] Blelloch, G. Programming Parallel Algorithms. *Communications of the ACM 39*, 3 (1996), pp. 85–97.

[22] Buyya, R., Abramson, D., and Giddy, J. An Economy Driven Resource Management Architecture for Global Computational Power Grids. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), Las Vegas, USA* (2000), vol. 2, pp. 26–29.

[23] Buyya, R., Yeo, C., Venugopal, S., Broberg, J., and Brandic, I. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as The 5th Utility. *Future Generation Computer Systems 25*, 6 (2009), pp. 599–616.

[24] Chakravarty, M., Keller, G., Lechtchinsky, R., and Pfannenstiel, W. Nepal-Nested Data Parallelism in Haskell. *Euro-Par 2001 Parallel Processing* (2001), pp. 524–534.

[25] Chakravarty, M., Leshchinskiy, R., Jones, S., Keller, G., and Marlow, S. Data Parallel Haskell: a Status Report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming* (2007), ACM, pp. 10–18.

[26] Chamberlain, B., Choi, S., Lewis, E., Snyder, L., Weathersby, W., and Lin, C. The Case for High-level Parallel Programming in ZPL. *Computational Science & Engineering, IEEE 5*, 3 (1998), pp.76–86.

[27] Chapman, B., and Huang, L. Enhancing OpenMP and Its Implementation for Programming Multicore Systems. *PARCO. Advances in Parallel Computing 15* (2008), pp. 3–18.

[28] Chapman, B., Jost, G., and Ruud, V. *Using OpenMP. Portable Shared Memory Parallel Programming.* No. ISBN-13: 978-0-262-53302-7. The MIT Press Cambridge, Massachusetts, London, England, 2008.

[29] Chapman, B., Mehrotra, P., and Zima, H. High Performance Fortran without Templates: an Alternative Model for Distribution and Alignment. In *PPOPP '93: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1993), ACM, pp. 92–101.

[30] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., and Sarkar, V. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *ACM SIGPLAN Notices* (2005), vol. 40, ACM, pp. 519–538.

[31] Claessen, K., and Hughes, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Acm sigplan notices* (2000), vol. 35, ACM, pp. 268–279.

[32] Cole, M. *Algorithmic Skeletons: Structured Management of Parallel Computation.* PhD thesis, University of Edinburgh, 1988. Also published in book form by Pittman/MIT, 1989.

[33] Dai, J., Huang, B., Li, L., and Harrison, L. Automatically Partitioning Packet Processing Applications for Pipelined Architectures. *ACM SIGPLAN Notices 40*, 6 (2005), pp. 237–248.

[34] Domani, T., Goldshtein, G., Kolodner, E. K., Lewis, E., Petrank, E., and Sheinwald, D. Thread-Local Heaps for Java. In *SIGPLAN Not* (2002), ACM Press, pp. 76–87.

[35] Draper, B., Beveridge, J., Bohm, A., Ross, C., and Chawathe, M. Accelerated Image Processing on FPGAs. *Image Processing, IEEE Transactions on 12*, 12 (2003), pp. 1543–1551.

[36] Duran, A., Gonzàlez, M., and Corbalán, J. Automatic Thread Distribution for Nested Parallelism in OpenMP. In *Proceedings of the 19th Annual International Conference on Supercomputing* (2005), ACM, pp. 121–130.

[37] El-Ghazawi, T., and Cantonnet, F. UPC Performance and Potential: A NPB Experimental Study. In *Proceedings of the 2002 Conference on Supercomputing* (2002), IEEE Computer Society Press, pp. 1–26.

[38] Epstein, J. Functional Programming for The Data Centre. Master's thesis, University of Cambridge Computer Laboratory Fitzwilliam College, June 2011.

[39] Fenton, N., and Pfleeger, S. *Software Metrics: a Rigorous and Practical Approach*. PWS Publishing Co. Boston, MA, USA, 1998.

[40] Fluet, M., Rainey, M., Reppy, J., Shaw, A., and Xiao, Y. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore programming* (2007), ACM, pp. 37–44.

[41] Foster, I. *Designing and Building Parallel Programs*, vol. 95. Addison-Wesley Reading, MA, 1995.

[42] Gepner, P., and Kowalik, M. Multi-core Processors: New Way to Achieve High System Performance. In *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on* (2006), IEEE, pp. 9–13.

[43] Guo, Y., Zhao, J., Cave, V., and Sarkar, V. SLAW: a Scalable Locality-Aware Adaptive Work-Stealing Scheduler. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 1–12.

[44] Hammond, K., and Peyton-Jones, S. L. Some Early Experiments on the GRIP Parallel Reducer. In *Intl. Workshop on the Parallel Implementation of Functional Languages* (Nijmegen, The Netherlands, June 1990), pp. 51–72.

[45] Harris, T., Marlow, S., and Jones, S. Haskell on a Shared-memory Multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell* (2005), ACM, pp. 49–61.

[46] Harris, T., and Singh, S. Feedback Directed Implicit Parallelism. In *The 12 th ACM SIGPLAN International Conference on Functional Programming(ICFP 2007); Freiburg* (2007), ACM, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, pp. pp. 251–264.

[47] Henty, D. Performance of Hybrid Message-passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)* (2000), IEEE Computer Society, p. 10.

[48] Horváth, Z., Hernyák, Z., and Zsók, V. Coordination Language for Distributed Clean. *Acta Cybernetica 17*, 2 (2005), pp. 247–271.

[49] Hudak, P. Para-Functional Programming. *IEEE, Computer Society Press 19*, 8 (1986), pp. 60–70.

[50] Iannucci, R. Toward a Dataflow/Von Neumann Hybrid Architecture. *ACM SIGARCH Computer Architecture News 16*, 2 (1988), pp. 131–140.

[51] James-Roxby, P., Schumacher, P., and Ross, C. A Single Program Multiple Data Parallel Processing Platform for FPGAs. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on* (2004), IEEE, pp. 302–303.

[52] Jones, M., and Hudak, P. Implicit and Explicit Parallel Programming in Haskell. Tech. Rep. YALEU/DCS/RR-682, University of Yale, August 1993.

[53] Jones, S., Clack, C., and Salkild, J. High-Performance Parallel Graph Reduction. *Lecture Notes in Computer Science 365* (1989), pp. 193–206. Berlin/Heidelberg.

[54] Jones Jr, D., Marlow, S., and Singh, S. Parallel Performance Tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (2009), ACM, pp. 81–92.

[55] Karonis, N., De Supinski, B., Foster, I., Gropp, W., Lusk, E., and Bresnahan, J. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *Parallel and Distributed Processing Symposium, 2000. Proceedings.* (2000), IEEE, pp. 377–384.

[56] Kasim, H., March, V., Zhang, R., and See, S. Survey on Parallel Programming Model. *Network and Parallel Computing* (2008), pp. 266–275.

[57] Kelly, P. *Functional Programming for Loosely-Coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing, Cambridge, MIT Press, 1989. ISBN 0262610574.

[58] Klusik, U., Loogen, R., Priebe, S., and Rubio, F. Implementation Skeletons in Eden - Low-Effort Parallel Programming. In *IFL'00 - Intl. Workshop on the Implementation of Functional Languages* (Aachen, Germany, Sept. 2000), vol. 2011 of *LNCS 2011*, Springer, pp. 71–88.

[59] Krste, A., Ras, K., Christopher, B., James, G., Parry, H., Kurt, K., Patterson, A., Plishker, W., Shalf, J., Williams, S., and Yelick, K. The Landscape of Parallel Computing Research: A View from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[60] Kumar, R., Farkas, K., Jouppi, N., Ranganathan, P., and Tullsen, D. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (2003), pp. 81–92.

[61] Larson, J. Erlang for Concurrent Programming. *ACM Queue 6*, 5 (2008), pp. 18–23.

[62] Levy, M., and Conte, T. Embedded Multicore Processors and Systems. *IEEE, Micro 29*, 3 (2009), pp. 7–9. NY NY 10017-2394 USA.

[63] Li, C., and Hains, G. A Simple Bridging Model for High-Performance Computing. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on* (2011), IEEE, pp. 249–256.

[64] Loidl, H. The Virtual Shared Memory Performance of a Parallel Graph Reducer. In *Cluster Computing and the 2nd International Symposium on Grid* (2002), IEEE Computer Society, pp. 311–311.

[65] Loidl, H., Trinder, P., K., H., A., A. Z., and C., B.-F. Semi-Explicit Parallel Programming in a Purely Functional Style: GpH. In *Process Algebra for Parallel and Distributed Processing*, M. Alexander and W. Gardner, Eds. Chapman & Hall/CRC, 2008, pp. pp. 47–76.

[66] Loidl, H.-W. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, Mar 1998.

[67] Loidl, H.-W., Rubio Diez, F., Scaife, N., Hammond, K., Klusik, U., Loogen, R., Michaelson, G., Horiguchi, S., Pena Mari, R., Priebe, S., Rebon Portillo, A., and Trinder, P. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation 16*, 3 (2003), pp. 203–251.

[68] Loidl, H.-W., Rubio Diez, F., Scaife, N., Hammond, K., Klusik, U., Loogen, R., Michaelson, G., Horiguchi, S., Pena Mari, R., Priebe, S., Rebon Portillo, A., and Trinder, P. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation 16*, 3 (2003).

[69] Loidl, H.-W., Trinder, P., Hammond, K., Junaidu, S., Morgan, R., and Peyton Jones, S. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience 11*, 12 (October 1999), pp. 701–752.

[70] Loidl, H.-W., Trinder, P., Hammond, K., Junaidu, S., Morgan, R., and Peyton Jones, S. Engineering Parallel Symbolic Programs in GPH. *CPE 11* (1999), pp. 701–752.

[71] Loogen, R. Programming Language Constructs. In *Research Directions in Parallel Functional Programming*, K. Hammond and G. Michaelson, Eds. Springer-Verlag, 1999, pp. 63–91.

[72] Loogen, R., Ortega-Mallén, Y., Peña, R., Priebe, S., and Rubio, F. Parallelism Abstractions in Eden. In *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003, ch. 4, pp. 95–124.

[73] Loogen, R., Ortega-Mallén, Y., and Peña-Marí, R. Parallel Functional Programming in Eden. *Journal of Functional Programming 15*, 03

(2005), pp. 431–475.

[74] Loogen, R., Ortega-Mallén, Y., and Peña-Marí, R. Parallel Functional Programming in Eden. *Journal of Functional Programming 15*, 3 (2005), pp. 431–475.

[75] Lusk, E., and Chan, A. Early Experiments With The OpenMP/MPI Hybrid Programming Model. In *Proceedings of the 4th International Conference on OpenMP in a New era of Parallelism* (2008), Springer-Verlag, pp. 36–47.

[76] Maier, P., and Trinder, P. Implementing a High-level Distributed-memory Parallel Haskell in Haskell. Tech. Rep. HW-MACS-TR-0091, School of Mathematical and Computer Sciences, Heriot-Watt University, 2011. www.macs.hw.ac.uk/∼pm175/papers/Maier_Trinder_IFL2011_XT.pdf.

[77] Marlow, S., Maier, P., Loidl, H.-W., Aswad, M., and Trinder, P. Seq no More: Better Strategies for Parallel Haskell. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell* (Baltimore, MD, United States, Sept. 2010), ACM Press, pp. 91–102.

[78] Marlow, S., Newton, R., and Peyton Jones, S. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell* (2011), ACM, pp. 71–82.

[79] Marlow, S., and Peyton Jones, S. Multicore Garbage Collection With Local Heaps. In *Proceedings of the International Symposium on Memory Management* (2011), ACM, pp. 21–32.

[80] Marlow, S., Peyton Jones, S., and Singh, S. Runtime Support for Multicore Haskell. In *ACM SIGPLAN Notices* (2009), vol. 44, ACM, pp. 65–78.

[81] McGowen, R., Poirier, C., Bostak, C., Ignowski, J., Millican, M., Parks, W., and Naffziger, S. Power and Temperature Control on a 90-nm Itanium Family Processor. *Solid-State Circuits, IEEE Journal of 41*, 1 (2006), pp. 229–237.

[82] McIlroy, R. *Using Program Behaviour to Exploit Heterogeneous Multicore Processors*. PhD thesis, School of Computing Science, University of Glasgow, April 2010.

[83] Mohr, E., Kranz, D., and Halstead Jr, R. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems* (1991), pp. 264–280.

[84] Moore, G. Cramming more Components onto Integrated Circuits. *IEEE SolidState Circuits Newsletter 20*, 3 (2006), pp. 33–35.

[85] Munshi, A., Gaster, B., Mattson, T., Fung, J., and Ginsburg, D. *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.

[86] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., and Aprà, E. Advances, Applications and Performance of The Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications 20*, 2 (2006), 203.

[87] Nitzberg, B., and Lo, V. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE, Computer 24*, 8 (1991), pp. 52–60.

[88] Oak Ridge National Laboratory. *Parallel Virtual Machine Reference Manual*. University of Tennessee, Aug 1993.

[89] Partain, W. The Nofib Benchmark Suite of Haskell Programs. In *Glasgow Workshop on Functional Programming* (Ayr, Scotland, 1992), Workshops in Computing, Springer-Verlag, pp. 195–202.

[90] Pase, D., and Wagner, D. Method of Managing Distributed Memory within a Massively Parallel Processing System, Oct 1996. US Patent 5,566,321.

[91] Peña, R., Rubio, F., and Segura, C. Deriving Non-Hierarchical Process Topologies. In *3rd Scottish Functional Programming Workshop (SFP01), selected papers* (2001), K. Hammond and S. Curtis, Eds., vol. 3 of *Trends in Functional Programming*, Intellect, pp. 51–62.

[92] Peyton J., S., Clack, C., Salkild, J., and Hardie, M. GRIP a High-Performance Architecture for Parallel Graph Reduction. *Lecture Notes in Computer Science 274* (1987), pp. 98–112.

[93] Peyton Jones, S., Gordon, A., and Finne, S. Concurrent Haskell. In *POPL'96 - Symposium on Principles of Programming Languages* (St Petersburg, Florida, Jan. 1996), ACM, pp. 295–308.

[94] Peyton Jones, S., and Lester, D. *Implementation of Functional Programming Languages*. No. ISBN:013453333X. Prentice-Hall International Series in Computer Science, Upper Saddle River, NJ, USA, 1987.

[95] Plasmeijer, M. J. CLEAN: A Programming Environment Based on Term Graph Rewriting. In *Proc. of Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA95* (1995), Elsevier, pp. 233–240.

[96] Quinn, M. *Designing efficient algorithms for parallel computers*. McGraw-Hill, Inc., 1986.

[97] Rabenseifner, R. Hybrid Parallel Programming on HPC Platforms. In *proceedings of the Fifth European Workshop on OpenMP, EWOMP* (2003), vol. 3, pp. 185–194.

[98] REPPY, J., RUSSO, C., and YINGQI, X. Parallel Concurrent ML. *ACM SIGPLAN notices 44*, 9 (2009), pp. 257–268.

[99] Ridge, D., Becker, D., Merkey, P., and Sterling, T. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Aerospace Conference, 1997. Proceedings., IEEE* (1997), vol. 2, IEEE, pp. 79–91.

[100] Rinard, M. Locality Optimizations for Parallel Computing using Data Access Information. *International Journal of High Speed Computing 9*, 2 (1997), pp. 161–161.

[101] Ross, P. Why CPU Frequency Stalled. *IEEE, Spectrum. 45*, 4 (2008). NY NY 10017-2394 USA.

[102] Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., and Hwu, W. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), ACM, pp. 73–82.

[103] S., S. Private Communication Regrading FDIP Performance. 11 2008.

[104] Sadashiv, N., and Kumar, S. Cluster, Grid and Cloud Computing: A Detailed Comparison. In *Computer Science & Education (ICCSE), 2011 6th International Conference on* (2011), IEEE, pp. 477–482.

[105] Santos, A. M. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, July 1995.

[106] Skillicorn, D., and Talia, D. Models and Languages for Parallel Computation. *ACM Computing Surveys (CSUR) 30*, 2 (1998), pp. 123–169.

[107] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarr, J. *MPI: The Complete Reference* , vol. 1. The MIT Press, 1998. MIT, Cambridge.

[108] Sohn, A., Sato, M., Yoo, N., and Gaudiot, J. Data and Workload Distribution in a Multithreaded Architecture. *Journal of Parallel and Distributed Computing 40*, 2 (1997), pp. 256–264.

[109] Sterling, T., Gropp, W., and Lusk, E. *Beowulf Cluster Computing with Linux, Second Edition.* No. ISBN:0262692929. The MIT Press, Massachusetts Institute of Technology, London, England, 2003.

[110] Steven, G., Christianson, B., Collins, R., Potter, R., and Steven, F. A Superscalar Architecture to Exploit Instruction Level Parallelism. *Microprocessors and Microsystems 20*, 7 (1997), pp. 391–400.

[111] Sunderam, V., and Geist, G. Heterogeneous Parallel and Distributed Computing. *Parallel Computing 25*, 13-14 (1999), pp. 1699–1721.

[112] Suter, F., Desprez, F., and Casanova, H. From Heterogeneous task Scheduling to Heterogeneous Mixed Parallel Scheduling. In *Euro-Par 2004 Parallel Processing* (2004), Springer, pp. 230–237.

[113] Svennebring, J., Logen, J., Engblom, J., and Stőmblad, P. Embedded Multicore: An Introduction. Tech. Rep. EMBMCRM Rev 0, Freescale Semiconductor, Inc., 07 2009.

[114] Taylor, F. *Parallel Functional Programming by Partitioning.* PhD thesis, Imperial collage of Science, Technology and Medicine, Depart. of Computing, London, UK, 9 1996.

[115] Taylor, X., and Stevens, R. Design and Implementation of Prophesy Automatic Instrumentation and Data Entry System. In *Proc. 13th IASTED Intl Conf. Parallel and Distributed Computing and Systems (PDCS01)* (2001).

[116] Teresco, J., Fair, J., and Flaherty, J. Resource-aware Scientific Computation on a Heterogeneous Cluster. *IEEE, Computing in science & engineering 7*, 2 (2005), pp. 40–50.

[117] Teresco, J., Flaherty, J., Baden, S., Faik, J., Lacour, S., Parashar, M., Taylor, V., et al. Approaches to Architecture-aware Parallel Scientific Computation. Tech. Rep. CS-04-09, Williams College Department of Computer Science, 2005.

[118] Thornley, J. Integrating Functional and Imperative Parallel Programming: CC++ Solutions to The Salishan Problems. In *Parallel Processing*

Symposium, 1994. Proceedings., Eighth International (1994), IEEE, pp. 61–67.

[119] Trinder, P. RELEASE A High-Level Paradigm for Reliable Large-Scale Server Software. Web Page, 02 2012. http://www.release-project.eu/publications.php.

[120] Trinder, P., Cole, M., Hammond, K., Loidl, H., and Michaelson, G. Resource Analyses for Parallel and Distributed Coordination. *Concurrency and Computation: Practice and Experience* (2011). Wiley Online Library (to appear).

[121] Trinder, P. W., Hammond, J., Mattson, J., Partridge, A., and Peyton Jones, S. L. GUM: a Portable Parallel Implementation of Haskell. In *PLDI 1996: Proc ACM SIGPLAN 1996 Conf. on Programming Language Design and Implementation* (New York, NY, USA, 1996), ACM Press, pp. 79–88.

[122] Trinder, P. W., Hammond, K., Loidl, H. W., and Jones, P. Algorithm + Strategy = Parallelism. *Journal of Functional Programming 8* (1998), pp. 23–60.

[123] Valiant, L. A Bridging Model for Multi-core Computing. *Journal of Computer and System Sciences 77*, 1 (2011), pp. 154–166.

[124] Wagner, D., and Calder, B. Leapfrogging: A Portable Technique for Implementing Efficient Futures. In *ACM SIGPLAN Notices* (1993), vol. 28, ACM, pp. 208–217.

[125] Walker, D., and Dongarra, J. MPI: a Standard Message Passing Interface. *Supercomputer 12* (1996), pp. 56–68. ASFRA BV.