Shape-based Cost Analysis of Skeletal Parallel Programs

Yasushi Hayashi

Doctor of Philosophy

Institute for Computing Systems Architecture
Division of Informatics
University of Edinburgh

2001

Abstract

This work presents an automatic cost-analysis system for an implicitly parallel skeletal programming language.

Although deducing interesting dynamic characteristics of parallel programs (and in particular, run time) is well known to be an intractable problem in the general case, it can be alleviated by placing restrictions upon the programs which can be expressed. By combining two research threads, the "skeletal" and "shapely" paradigms which take this route, we produce a completely automated, computation and communication sensitive cost analysis system. This builds on earlier work in the area by quantifying communication as well as computation costs, with the former being derived for the Bulk Synchronous Parallel (BSP) model.

We present details of our shapely skeletal language and its BSP implementation strategy together with an account of the analysis mechanism by which program behaviour information (such as shape and cost) is statically deduced. This information can be used at compile-time to optimise a BSP implementation and to analyse computation and communication costs. The analysis has been implemented in Haskell. We consider different algorithms expressed in our language for some example problems and illustrate each BSP implementation, contrasting the analysis of their efficiency by traditional, intuitive methods with that achieved by our cost calculator. The accuracy of cost predictions by our cost calculator against the run time of real parallel programs is tested experimentally.

Previous shape-based cost analysis required all elements of a vector (our nestable bulk data structure) to have the same shape. We partially relax this strict requirement on data structure regularity by introducing new shape expressions in our analysis framework. We demonstrate that this allows us to achieve the first automated analysis of a complete derivation, the well known *maximum segment sum* algorithm of Skillicorn and Cai.

Acknowledgements

First and foremost I would like to thank my supervisor Murray Cole for all his constant support, friendly encouragement, great patience, invaluable advice, and excellent guidance. He was always prepared to listen to my ideas, and gave me valuable suggestions and timely encouragement. I am particularly grateful for his careful reading of several drafts and suggestions he made.

I would also like to thank the FISh team at the University of Technology, Sydney and in particular Barry Jay and Paul Steckler both for their inspiring work in this area and for providing me with the source code for the original PRAM cost calculator from which my own system derives.

Many thanks to all friends I have made during my stay in Edinburgh.

Parallel machine facilities were provided by the Edinburgh Parallel Computing Centre. This work was partially funded by the Overseas Research Studentship, awarded by the British Council.

Finally, I would like to thank my parents for their encouragement and support over the years.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. Part of this work was published (or accepted for publication) in the following articles.

- Y. Hayashi and M. Cole. BSP-based Cost Analysis of Skeletal Programs. In G. Michaelson and P. Trinder, editors, *Trends in Functional Programming*, pages 20-28, Intellect, 2000.
- Y. Hayashi and M. Cole. Static Performance Prediction of Skeletal Programs. *Journal of Parallel Algorithms and Applications*, 2002, to appear.
- Y. Hayashi and M. Cole. Automated Cost Analysis of a Parallel Maximum Segment Sum Program Derivation. *Parallel Processing Letters*, 2002, to appear.

(*Yasushi Hayashi*)

To my parents

Table of Contents

Chapter 1

Introduction

1.1 Motivation

One of the main reasons for the failure of parallelism to enter mainstream computing is the lack of the portability and performance predictability enjoyed by sequential systems. In sequential programming, the measure of efficiency is based on instruction counts and order analysis, and is often called the von Neumann cost model or RAM model. A similar approach is widely used for theoretical analysis of parallel programs. However, it is less useful than its sequential counterpart for the following reasons.

- Real parallel computing usually involves communication and contention costs which can significantly depend on characteristics of parallel machines. Analysis ignores these and leads to unreliable predictability and poor performance portability.
- Counting the number of instructions is a complicated task in the parallel setting because it depends on the data allocation strategy as well as intermediate data size.
- Conventional asymptotic cost analysis, which is based on instruction counts and order analysis parameterised by input size and the number of processors, models behaviour when these parameters grow towards the infinite, but very often

the target architecture has a moderate and fixed number of processors and the application will involve only a particular range of problem sizes.

In practice, parallel programmers largely rely upon a combination of common sense, intuition and profiling to make the important algorithmic decisions which will affect performance. One approach to alleviating this problem is to place restrictions upon the programs which can be expressed. Two research threads which have taken this route involve the “skeletal” and “shapely” paradigms. In the extreme, these can produce a language for which static analysis becomes tractable.

The skeletal approach to the design of parallel programming systems [25, 29, 62, 67] proposes that the complexity of parallel programming be contained by restricting the mechanisms through which parallelism can be introduced to a small number of architecture independent control constructs, originally known as *algorithmic skeletons*. Each skeleton specification captures the logical behaviour of a commonly occurring pattern of parallel computation (such as “divide-and-conquer”, “farm” or “scan”), while pre-packaging and hiding the details of its implementation using the explicit parallelism of lower level primitives provided by the target system. Since a parallel program is constructed from predefined skeletons, the cost of the program can also be expressed in terms of its parallel control structure.

The shapely programming methodology [52, 53, 54] proposes that through careful language design the *shape* (loosely speaking, the size and structure) of data at any point during execution can be determined statically and automatically, even for programs in which shape is varied dynamically. This is fundamental for static cost analysis because it requires information about the sizes and shapes of the input data structure and, when programs are compositions of parallel operations it is extremely difficult to capture the behaviour of these intermediate results by hand. When we consider inter-processor communication cost, this is again fundamental since predicting the size of messages is also required.

Shape-based cost analysis for functional languages based on these restrictions was first proposed by Jay et al. [55]. The analysis has the characteristics of being automatable and giving absolute value prediction (rather than asymptotic). It used the tightly synchronised, uniform access cost, shared memory PRAM model as its target architecture. The PRAM is an abstract model which takes no account of the communication and contention costs incurred on realistic parallel machines (whether explicitly programmed or in support of a shared memory abstraction). The main motivation of this thesis is to develop a cost analysis for their language which can account for communication as well as computation, while keeping its characteristics of being automatable. To achieve this we choose the Bulk Synchronous Parallel (BSP) computation [48, 79, 83] as our implementation model (therefore, SPMD model) to introduce parallelism.

The BSP model proposes that decoupling communication and synchronisation is the key to a simple and accurate cost model that can be used to analyse and guide design of parallel algorithms. The purpose of the BSP cost model is to refine the standard simple asymptotic cost analysis by

- decoupling the asymptotic analysis of the problem size n, from the potentially modest number of processors p;
- costing communication as well as computation;
- introducing a small number of parameters that capture performance characteristics of a machine, so that the comparative performance of an algorithm can be analysed across machines.

It can be used both to choose an appropriate architecture among possible target computers and to adapt an algorithm which is more efficient on the target architecture.

This thesis investigates the use of skeletal, shapely and BSP approaches to produce a skeleton-oriented parallel programming language for which static performance analysis is completely automatable, communication sensitive, architecture characteristic sensitive and absolute value predictable. Our source language is functional, since this is the most convenient paradigm within which to express our constraints. Our analysis predicts the behaviour of these programs when compiled to a BSP target. Such information can be used to choose one algorithm over another, or one data structure over another when the program is constructed. The functional paradigm also has more advantages:

- ease of program construction;
- ease of function/module reuse;
- ease of program transformation;
- scope for optimisation.

In particular, one of our aims is to use our cost analysis to predict the effect of performance change in program transformation steps. The bulk of previous work has focused on the fundamental question of the semantic soundness of each step, with responsibility for choosing steps and for judging their effect on performance left to the programmer's intuition. Automatic cost modelling could provide the programmer with immediate feedback on performance implications.

In common with other skeletal languages, our approach provides a structured conceptual framework for message passing programming. Structured languages and methodologies promote an approach in which the key algorithmic decisions are taken early and at a high level, enhancing both portability and maintainability [4, 39]. Our language and analysis could be used either as a real programming framework in its own right, or as a testbed for algorithmic ideas which would subsequently be re-coded semi-automatically into a more conventional form, for example following the BSP implementation templates of the skeletons.

1.2 Overview of the Analysis

Our language VEC-BSP is a simple shapely functional language which operates upon nested vectors of data. It has a cost analysis system which produces predicted run-time costs based on the BSP model from a source code. Shapeliness means that the form and size of data structures can be deduced statically. Shape constraints (which are analogous to type constraints) are used to ensure that all elements of a vector have the same shape so that information about large structures can be captured and manipulated concisely. (This restriction is partially relaxed in an expanded version of VEC-BSP described in chapter 7). Parallelism is introduced by a small number of skeletons, that is higher order functions each of which has a predefined parallel implementation template. The program is written by means of the application of predefined operations (which includes conventional sequential operations and skeletons) and lambda expressions and conditionals. VEC-BSP excludes unbound iteration and recursion since our goal is full automation.

In essence, our approach to cost analysis is a form of abstract interpretation. A source program is translated to another program in the target language MSIZE which, when run, will compute some implementation information such as shape and run-time cost. MSIZE is essentially a variant of VEC-BSP in which types, terms and operators which represent and manipulate real data have been removed, and with the addition of new features which manipulate implementation information not present in VEC-BSP. The core of the method is a translation function *cost* which accepts VEC-BSP terms and returns MSIZE terms in the form of *cost tuples*, whose components capture some kinds of evaluation information. For example,

- data size - a measure of the quantity of data which would have to be communicated to describe the term (in order to compute communication cost from the transmitted data);
- data pattern - an indication of the data distribution strategy required by the term's implementation. (in order that communication of the data between evaluation phases can be optimised);
- cost - an evaluation cost function for the term, mapping from performance parameters to time (so that evaluation time for the term can be computed, given the performance characteristics of the specific target);
- application pattern - an indication of application structure, that is whether sequential or parallel (in order to compute communication cost between the component evaluation process and the application process, and to optimise the communication of data between evaluation phases).

This information is propagated during evaluation of the MSIZE program through the definition of an application operator of the MSIZE, which constitutes the heart of the analysis. Making the mechanism work efficiently requires careful design of the implementation model and choice of the components of the cost tuple. The inter-action of those components could generate useful static information on intermediate results which could be used by a compiler for various purposes to improve efficiency. In this thesis, we construct a cost analysis which has the property of automatic, communication sensitive, machine performance sensitive and absolute value cost derivation.

1.3 Contributions of the Thesis

The main contributions of this thesis can be summarised as follows.

- We demonstrate the first completely automated, communication sensitive shape-base cost analysis system for an implicitly parallel skeletal programming language of nested arrays. This builds on earlier work by Jay et al. [54] in the area by quantifying communication as well as computation costs, with the former being derived by changing target implementation model from PRAM to BSP;

- We add several built-in second-order functions, each of which has a parallel implementation template and predefined application cost which is parameterised by the argument shape, in order to enhance the skeletal approach of parallel programming and to broaden applicability of our analysis;

- We extend Jay's shape-based analysis framework with cost tuples which can contain useful static information as components, and illustrate how this information is used for costing the communication process, optimising interface communication and eventually computing BSP cost;

- We partially relax our strict requirements on data structure regularity (but without losing static predictability) by introducing new shape expressions in our analysis frame work;

- We present the first analysis of a complete derivation, the well known *maximum segment sum* algorithm of Skillicorn and Cai;

- We illustrate skeletal programming in VEC-BSP by implementing several example programs. The accuracy of predictions made by our cost calculator against the run time of real parallel programs is tested experimentally.

1.4 Structure of the Thesis

The following chapters can be divided into three parts. The first part, chapter 2 presents a survey of existing cost models giving the background of our cost model. In the second part, chapters 3, 4, 5 present our language, its implementation strategy, the definition and implementation of our analytic technique. The third part, chapters 6, 7 demonstrate applications and possible extensions of our analysis, with the concluding chapter 8. More details of contents of each chapter is as follows:

- **Chapter 2** begins with a survey of the main theoretical low level cost models for parallel programming. Next, we explain the concept of the “skeleton” methodology and investigate three works of cost analysis for BMF style parallel skeletal programming in detail. Then we give a short survey of more related works.
- **Chapter 3** is the central part of the thesis which gives the definition of VEC-BSP, its implementation strategy and cost analysis framework.
- **Chapter 4** gives the parallel implementation templates of the built-in second order functions with application costs, which completes the cost analysis presented in chapter 3.
- **Chapter 5** outlines the Haskell implementation of our cost analysis.
- **Chapter 6** demonstrates that our analysis allows us to compare the performance of alternative algorithms for the same problem against one another in a concrete way. The comparison between predicted costs against the run-time of equivalent hand-compiled BSP programs on a real machine is also given.
- **Chapter 7** augments our analysis framework to partially relax our strict requirements on data structure regularity. We demonstrate that the modified framework

allows the cost analysis system to cost complete derivation steps of an algorithm for the maximum segment sum problem.

- **Chapter 8** presents a summary and contributions of the thesis, with directions for future research.

Chapter 2

Cost Models for Parallel Computation

In sequential computing, the von Neumann model dominates. Parallel programs are inherently more complex than their sequential counterparts. This complexity seems tractable only within some abstracted and idealised model. However, no single model of parallel computation has yet come to dominate in the way the von Neumann model has dominated sequential computing. This chapter surveys some of the models for parallel computation. Section 2.1 reviews the most influential early theoretical model, the PRAM. Section 2.2 describes a dominant programming model, message passing programming, and two proposed cost models, BSP and LogP. Finally, section 2.3 surveys the skeleton-oriented languages and their cost models, especially BMF style programming.

2.1 The PRAM Model

The most influential early theoretical parallel computation model is the *parallel random access machine* (PRAM) introduced by Fortune and Wyllie [36], which has been used widely to assess the theoretical performance analysis of parallel algorithms. The PRAM consists of a shared memory and a number of processors each with local memory. The processors are controlled by a common clock and operate synchronously. In every cycle each processor may read a value from global memory, write a value to the global memory, or compute an operation. So, any location can be accessed by a processor in unit time (that is, in a single instruction time), independent of the access pattern. Normally the PRAM model is used with algorithms in which processors execute the same instruction together but operating on different data. There are four subclasses of the PRAM, provided to define how simultaneously reading and writing to the same memory location should be handled: EREW (Exclusive Read Exclusive Write), CREW (Concurrent Read Exclusive Write), ERCW (Exclusive Read Concurrent Write), CRCW (Concurrent Read Concurrent Write). The ERCW is often not considered because a machine powerful enough to support concurrent writes can also support concurrent reads. In those cases where concurrent write is permitted, an additional strategy is necessary to indicate how conflicts are resolved and what is actually stored in the location. Some possibilities often quoted are: *Common* when simultaneous writing is only allowed if the values to be written are the same; *Arbitrary* when the processor that succeeds in its write operation is selected arbitrarily from the writing processors; *Priority* when there exists a predefined priority order to select the processor that will succeed; *Combining* when the value written is a linear combination of all values being written by the individual processors.

The PRAM model is an idealised model that ignores practical considerations and focuses on concurrency. Its simplicity and generality have led to the widespread acceptance of the PRAM within the theoretical community and it offers worthwhile results in design and analysis of parallel algorithms. However, the idealisation hiding the issues of synchronisation, data locality, interprocessor or processor-to-memory communication and other machine-specific issues often leads to unreliable prediction of real execution costs. The complexity of a PRAM algorithm is given in terms of the number of time steps and maximum number of processors required in any one of those time steps. There is no straightforward way to add communication costs which could largely depend on communication performance characteristics of a real machine into the model and convert PRAM costs to real costs.

2.2 Message Passing Model

‘The message passing programming models provided by communications libraries such as PVM [37] or MPI [59] have been a dominant model for scientific and commercial parallel applications for the last decade. In this section, we first outline the basic concepts in message passing computing, and then we outline the two theoretical models, the BSP model and the LogP model which can model the cost of message passing processing with small numbers of architecture parameters.

Message Passing Multicomputer

The message passing multicomputer node consists of a processor and local memory that is not accessible by other processors. The memory is distributed among the computers and each computer has its own address space. A processor can only access a location in its own memory. The interconnection network is provided for processors to send messages to other processors. These messages can include data that other processors may require for their computations. The messages in a message passing multicomputer carry data from one processor to another as dictated by the program. The message passing paradigm can be implemented not only in a message passing multicomputer but also in a shared memory multiprocessor by using the shared memory to hold data to be sent from one process to another process.

Message Passing Programming

Programming a message passing multicomputer involves dividing the program into parts that are intended to be executed simultaneously to solve the problem. Programming could use a parallel or extended sequential language, but a common approach is to use message passing library routines that are linked to conventional sequential languages such as C for message passing. A problem is divided into a number of concurrent processes which may be executed on individual computers and will communicate by using message passing instructions to synchronise with and to access memory of other processes, that will be the only way to distribute data and results between processes. It is necessary to say explicitly what processes are to be executed, when to pass messages between concurrent processes, and what to pass in the messages. Send and receive message passing library calls often have the form

```
send(parameter_list)
recv(parameter_list)
```

where send() is placed in the source process, originating the message, and recv() is placed in destination process to collect the messages being sent. The actual parameters will depend upon the software and in some cases can be complex.

There are usually many other message passing and related routines that provide desirable features. A frequent requirement for the process originating the message is to send the same message to more than one destination process. The term *broadcast* is used to describe sending the same message to all the processes concerned with problem. The term *scatter* is used to describe sending each element of an array of data in the root to a separate process. The contents of the ith location of the array is sent to the ith process. The term *gather* is used to describe having one process collect individual values from a set of processes. Gather is normally used after some computation has been done by these processes. Most message passing systems provide for these operations and other related operations.

Message Passing Interface (MPI) [59] is an example of such communication routines. Processes communicate with one another by sending packets of information using the *point-to-point* communication routines such as MPI_Send() and MPI_Recv(). MPI includes a wide selection of routines to offer different sorts of synchronisation in the sending which can be employed to improve the efficiency of an implementation at the cost of increased program complexity. The type of the values can be simple basic types such as integer and real, or derived types can be created by the programmer. MPI also offers a range of collective communication routines which can be used to perform a common operation across all the processes in a specific communication context. Examples includes MPI_Bcast() to broadcast a value from one process to all the others, and MPI_Scatter() to distribute a one-dimensional array of values over all processes.

There is also limited scope for performing collective computation within a communication context, e.g. MPI_Reduce() to perform a tree of binary operations, such as addition, on one value from each process.

The message-passing paradigm usually requires the programmer to provide explicit message passing calls in code, which is very error prone and has been compared to low level assembly language programming. However, the message passing paradigm has the advantage of its direct applicability to the computers connected on a network. Using interconnected computers allows newer computers to be more easily incorporated into the system. These computers could be networked workstations.

2.2.1 The BSP Model

Message passing systems such as MPI and PVM have no simple analytic cost model for performance prediction. The bulk-synchronous parallel (BSP) model by Valiant [83] provides an alternative parallel model that has a cost model which is attractive by virtue of its conceptual simplicity and pragmatic accuracy.

The BSP Machine Model

In the BSP model, a parallel computer consists of three components, that is: a set of processors, each with a local memory; a global communication network that delivers message in a point-to-point manner among the processors; and a mechanism for globally synchronising all processors by means of a barrier. The model has no concept of processor locality or the topology of the underlying network.

The BSP Programming Model

It is normal for BSP programs to be written in a Single Program Multiple Data (SPMD) style in which a fixed number of processes, each of which executes the same program, is created at program start-up. The distinguishing feature of the model is that it decouples the two fundamental aspects of parallel computation: communication and synchronisation. This separation is the key to :

- a simple and accurate cost model that can be used to analyse and guide the design of parallel algorithms.
- achieving universal applicability across a wide range of parallel architectures, from shared-memory multiprocessors to tightly-coupled distributed-memory machines or networks of workstations.

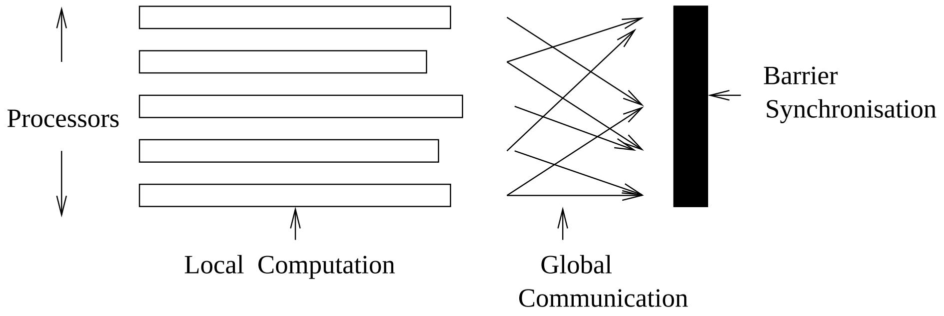


Figure 2.1: Superstep

BSP programs consist of a sequence of *supersteps* each of which is conceptually subdivided into three ordered phases (figure. 2.1) consisting of:

1. simultaneous local computation in each process, using only values stored in the memory of its processor;
2. communication actions amongst the processors, causing transfers of data between processors;
3. a barrier synchronisation, which waits for all of the communication actions to complete, and which then makes any data transferred visible in the local memories of the destination processors.

In addition to a model, BSP programs can be implemented. Although it was originally envisaged that BSP programs would be written using BSP languages, the model has actually been realised in terms programming libraries. Such libraries provide a very small number of library functions that implement the necessary communication and synchronisation primitives. For example, Oxford BSP [63] supports a simple set of communication primitives, but does not directly support communicating dynamically-allocated data. However, the library has been tested relatively widely. Versions are available for Cray, SGI, shared memory, as well as TCP. BSPlib [48] developed by the BSP Worldwide organisation, is the successor of Oxford BSP and an attempt at a standard. It offers a set of communication primitives which support both direct memory access (buffered and unbuffered puts and gets into remote memory for statically and dynamically allocated data) and a form of bulk synchronous message passing (sends to a remote buffer). The library is available in a number of native, hardware-specific implementations(Cray, SGI, etc.) as well as shared memory and generic (based on MPI) versions.

Green BSP library [41] offers a simple set of communication primitives based on bulk synchronous message passing rather than the put/get semantics of Oxford BSP. (These features are now available in BSPlib.) Shared memory, MPI, and TCP versions exist.

Because BSPlib has been long known as a standard and available for wide range of hardware including our available hardware, a Sun multiprocessor, we use the BSPlib to express BSP implementation template and to write real BSP programs to which example problems of VEC-BSP are hand compiled.

Recently, the Paderborn University BSP (PUB) Library [15] has been developed. The PUB library offers the same functionality as BSPlib, but in addition provides several other features. In particular, it has a mechanism to partition the machine into subsets which synchronise independently. In this way more complex programs made of patterns which synchronise independently can be built. It also provides other forms of synchronisation besides standard barrier synchronisation.

The BSP Cost Model

The standard way of analysing the cost of a parallel algorithm is to use instruction counts and order analysis. For example, a logarithmic combining technique can be used to calculate the sum of n values in time $O(\log n)$ on n processors. The BSP has a cost model which is attractive by virtue of its conceptual simplicity and pragmatic accuracy. Its cost calculus is straightforward because of the superstep structure of programs. As the barrier synchronisation involves all processes, then the cost of a sequence of supersteps is simply the sum of the costs of the separate supersteps. Although existing parallel computers have very different performance characteristics, these differences are captured by three parameters, that is: p: the number of processors; g: the ratio of communication throughput to processor throughput; and l: the time required to barrier synchronise all processors. All of the effects of contention and congestion on communication is captured in the parameter g. When the communication pattern requires at most h messages into or out of any processor, the communication time is determined as $h\,g$. The cost of a single superstep is determined by

$$\text{cost of a superstep} = \max_{0 \le i < p} w_i + \max_{0 \le i < p} h_i\, g + l$$

where w_i = local processing time on processor i, h_i = the number of words transmitted/received by processor i. Intuitively, the cost of a superstep is the execution time of the process that performs the largest local computation (denoted by $\max w_i$), plus the communication time of the process that performs the largest communication ($\max_{0 \le i < p} h_i \cdot g$), plus a constant cost l that arises from the barrier synchronisation and other one-time costs associated with the superstep, such as the overhead of initiating communication. If $h = \max h_i$, then such a communication pattern is called an h-relation. The costs given by this model are not theoretical costs, but closely match the observed execution times over a wide variety of applications and target architectures [72].

The g parameter of the cost model depends on the performance of the underlying architecture. For example, it depends on :

1. the bisection bandwidth of the communication network topology;

2. the protocols used to interface with and within the communication network;
3. buffer management by both the processors and the communication network;
4. the routing strategy used in the communication network.

The l parameter also depends on these properties of the architecture, as well as specialised barrier synchronisation hardware, if this exists. The BSP cost parameters for a variety of shared-memory and distributed-memory parallel machines are found in [79].

The BSP cost model can be used to analyse algorithms developed in any of the data parallel functional languages. Hill [46] discusses the potential for raising the level of abstraction of the programming language used to express BSP algorithms and concludes that the BSP cost calculus provides the right foundations upon which practical variants of parallel functional languages could be developed. For example, Caml-Flight [35] and BSML [6, 58] incorporate the one-sided communications of BSP within ML.

2.2.2 The LogP Model

Culler et al. [27] have developed the LogP model, an asynchronous model of a distributed memory multicomputer in which processors communicate by point-to-point messages. LogP specifies the performance characteristics of the interconnection network through a small number of machine parameters, but does not take into account the topology of the network.

The parameters of the LogP model are:
L - upper bound on latency incurred in sending a message from a source to a destination;
o - overhead, defined as the time the processor is engaged in sending or receiving a message, during which time it cannot do any thing else;
g - gap, defined as the minimum time between consecutive message transmissions or receptions;
P - number of processor/memory modules.

The parameters L, o and g are measured in processor cycles. Local operations take one cycle. It is also assumed that the interconnection network can only carry a finite number of messages at any instant, defined by $\lceil \frac{L}{g} \rceil$ messages from any processor to any other processor, where $\lceil\ \rceil$ is Gauss' symbol, that is, $\lceil x \rceil$ means the greatest integer that is less than or equal to x. If a processor attempts to send a message that would exceed this limit, it will stall. Communication is modeled by point-to-point messages of some fixed short size. As evidenced by experiential data collected on the CM-5 [82], this model can accurately predict communication performance when only fixed-sized short messages are used. Sending a small message between two processors takes $o+L+o$ cycles: o cycle on the sending processor, L cycles for the communication latency, and finally another o cycles on the receiving processor. Alexandrov et al. [3] incorporated long messages into the LogP model by introducing an additional parameter G, which is the time for each byte for long messages. Under the LogGP model, sending a message of k bytes first involves o cycles of sending overhead to get the first byte into the network. Subsequent bytes take G cycles each to go out. The last byte goes out at time $o+(k-1)G$. Each byte travels through the network for L cycles. Thus the last bytes exist the network at time $o+(k-1)G+L$. Finally, the receiving processor spends o cycles in overhead, so the entire message is available at the receiving processor at time $o+(k-1)G+L+o$. The sending and receiving processors are busy only during the o cycles of overhead, the rest of the time they can overlap computation with communication. Notice that the LogP model ignores $(k-1)G$ by assuming messages to be small.

2.3 Cost Models for Skeleton-Oriented Programming

2.3.1 Skeleton Approach

Parallelism introduces many more degrees of freedom into the space of programs, and into the space of architectures and machines. When we consider ways of executing a program on a machine, the number of possibilities is enormous. It is correspondingly difficult to find an optimal, or even an acceptable, solution within these spaces. It is also much more difficult to predict the detailed behaviour and performance of programs running on machines. One approach to alleviating this problem is to place restrictions upon the program which can be expressed. The skeleton approach introduced by Cole in [25] takes this route in terms of commonly occurring algorithmic patterns of parallel computation. This is led by the observation that many parallel applications developed up to now exploit parallelism according to a restricted set of regular patterns. In practice, parallel programmers more or less try to find a useful pattern or paradigm to solve their problem based on their programming experience. The skeleton-based language support this process, aiming to replace creating programs from scratch with the development of programs through the composition of a small number of architecture independent control constructs, known as algorithmic skeletons, thus improving programmability and ease of understanding of the derived program. Each skeleton specification captures a commonly occurring pattern of parallel computation, while prepacking and hiding the details of its implementation using the explicit parallelism of lower level primitives provided by the target system. Classical examples of skeletons include *farm*, which models master-slave parallelism, and *divide & conquer*, which solves a problem by recursive splitting.

For skeletons, owing to their regular structure, accurate performance models can be constructed. This enables estimations of the execution costs of skeletons which can be used for making algorithmic decisions at a high level.

One of the most commonly discussed skeleton is divide & conquer (`d&c`).

A general formulation is:

```
d&c::(a→bool)→(a→b)→(a→[a])→([b]→b)→a→ b
d&c trivial solve divide conquer P =
   if (trivial P) then (solve P)
   else conquer (map (d&c trivial solve divide conquer) (divide P))
```

This skeleton has four functional arguments: `trivial` tests if a problem is simple enough to be solved directly, `solve` solves the problem in this case, `divide` divides a problem into a list of subproblems, and `conquer` combines a list of sub-solutions into a new solution. The last argument `P` is the problem to be solved. The function `map` applies a given function to all elements. Given this skeleton, the implementation of an algorithm that has the structure of d&c requires only the implementation of the four sequential argument functions and a call of the skeleton. For instance, a *quicksort* procedure for lists can be implemented as follows:

```
quicksort list = d&c is_simple ident divide concat list
```

where `is_simple` checks if a list is empty or singleton, `ident` is the identity function and `divide` splits a list into three lists containing the elements that are smaller than given pivot element, the pivot element itself, and the elements greater than or equal to the pivot, respectively. Finally, `concat` concatenates three lists and `list` is the list to be sorted.

A variation is Rabhi's recursive partitioning skeleton [68] given by:

```
rp trivial solve divide conquer P =
   if (trivial P) then (solve P)
   else conquer P (map (rp trivial solve divide conquer)(divide P))
```

which differs from `d&c` in that conquer takes also the original problem `P` as parameter. [68] gives its implementation using distributed graph reduction and examples of its use.

Parallelism emerges naturally from the tree of computations produced by the combination of recursion and a `divide` function which generates more than one subproblem. Algorithms such as Strassen's matrix multiplication, polynomial evaluation, numerical integration, FFT, etc. [2] can be expressed similarly, only by using different customising argument functions. The tree of processes can be mapped down to physical processors in a number of ways. It would appear that all divide-and-conquer problems are not cost-optimal, since there is only one particular level in the tree which is active at any particular time. The main problem is in the difficulty of allocating tasks to processors, since tasks are generated dynamically. Some strategy for turning to sequential evaluation at some point to avoid the tiny tasks at the leaves of the tree may be required.

A common approach to achieving performance predictability is to derive symbolic mathematical formulae that describe the execution time of each skeleton. These formulae are typically parameterised by a set of parameters which capture the important factors that affect the execution time of the program. These parameters usually include the program size, number of processors used and other algorithm and hardware characteristics which can be given by a programmer, benchmarking, or a profiling tool.

For example, the performance model for the divide and conquer (DC) skeleton proposed by Darlington et al. [29] assumes the processors are organised into a balanced binary tree and all processors will eventually be used as leaves. The execution time can be predicted using the formula:

$$t_{sol_x} = \sum_{i=0}^{\log(p)-1} \left(t_{div_{\frac{x}{2^i}}} + t_{setup_{\frac{x}{2^i}}} + t_{comb_{\frac{x}{2^i}}} + t_{comm_{\frac{x}{2^i}}}\right) + t_{seq_{\frac{x}{2^{\log p}}}}$$

where t_{sol_x} is the time to solve a problem of size x, t_{div_x} is the time to divide a problem of size x, t_{comb_x} is the time to combine the two results, t_{setup_x} and t_{comm_x} are setup and transmission time for communication and t_{seq_x} is the time to solve a problem of size x sequentially.

A skeleton can be loosely defined as a pre-defined higher-order function with associated parallel implementations. Higher-order functions are commonly used in functional programs to express high-level operations on data structures, for example *map* and *fold* over lists. The style of programming based on higher-order functions over data structures has been influenced by the work of Backus [5] and the Bird-Meertens formalism [9]. Several researchers have developed skeleton programming systems where the only skeletons available are these data parallel higher-order functions such as *map* and *fold*. For example, the work of Skillicorn [77] uses the Bird-Meertens Formalism (BMF) as a calculus for deriving efficient programs from problem specifications using transformations. Examples include the derivation of data-parallel divide-and-conquer algorithms. We refer to such a programming style as BMF style parallel programming, which is similar to our language VEC-BSP. In this approach, performance models are required to investigate how the performance model of each higher-order function can be composed for estimation of the execution cost of a data-parallel functional program. Among cost models proposed for BMF style parallel programming, three of them provide the bases of our cost analysis, that is, Skillicorn's parallelised BMF, Rangaswami's HOPP model, and particularly Jay's shapely language VEC. The next three sections in this chapter investigate these models.

2.3.2 The Bird-Meertens Theory of Lists

The Bird-Meertens Formalism (BMF) [11, 10] developed by Bird and Meertens is a collection of second-order functions, algebraic identities and theorems relating these with concise notations which facilitate the transformation approach. Although a large amount of work has been done on other data types (arrays [10, 7], trees [38]), the theory of lists was the first studied and is the most well developed [9, 10]. We will focus on the theory of lists, as most of the work concerning the implementation and the cost calculus has been done on this theory. When given a data type along with a set of predefined collective operations, the programmer can express his/her algorithms only by means of the hierarchical composition of the operations provided in the language, much in the philosophy of combinatoric functional languages such as FP [5].

The following set of second-order functions are provided in the theory.

map (written $f*$), which applies f to all the elements of the list:

$$f * [a_1, a_2, \cdots, a_n] = [f a_1, f a_2, \cdots, f a_n]$$

reduce (written $\oplus/$) which reduces a list by an associative binary operator $\oplus$:

$$\oplus / [a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \ldots \oplus a_n$$

prefix (written $\oplus //$) which given a list returns the list of results of reduce applied to all the initial segments of the list:

$$\oplus // [a_1, a_2, \ldots, a_n] = [a_1, a_1 \oplus a_2, a_1 \oplus a_2 \oplus a_3, \ldots, a_1 \oplus a_2 \ldots \oplus a_n]$$

inits returns the list of initial segments of its argument list, shortest first:

$$\mathsf{inits}\, [a_1, a_2, \ldots, a_n] = [[\,], [a_1], [a_1, a_2], \ldots, [a_1, a_2, \ldots, a_n]]$$

tails, which returns the list of all final segments of its argument list, longest first:

$$\mathsf{tails}\,[a_1,a_2,\ldots,a_n] = [[a_1,a_2,\ldots,a_n],[a_2,\ldots,a_n],\ldots,[a_n],[\,]]$$

cross_products (written c_prod) which applies f to all pairs with one element from the first argument and the other from the second:

$$\begin{aligned}\mathsf{c\_prod}\ f\ [x_1,x_2,\cdots,x_m][y_1,y_2,\cdots,y_n] \quad = \quad & [[f\,x_1\,y_1,\ f\,x_2\,y_1,\cdots,f\,x_m\,y_1],\\ & [f\,x_1\,y_2,\ f\,x_2\,y_2,\cdots,\ f\,x_m\,y_2],\\ & \qquad\vdots\\ & [f\,x_1\,y_n,\ f\,x_2\,y_n,\cdots,\ f\,x_m\,y_n]]\end{aligned}$$

The theory has a set of algebraic identities as meaning-preserving laws which can be used to transform an algorithm with a poor performance to a more effective one. The following laws are some examples.

$$(f\circ g)* = (f*)\circ(g*) \qquad\qquad \text{(map distributivity)}$$

where $f\circ g$ means the function composition of f and g:

$$(f\circ g)\,x = f\ (g\,x)$$

It states that the map operator distributes over functional composition.

$$f*\circ +\!\!+/ = +\!\!+/\circ f** \qquad\qquad \text{(map promotion)}$$

where $+\!\!+$ denotes the concatenation function that takes two lists and returns a list which is the concatenation of the argument lists. It states that the result of concatenating a list of lists, and then apply f to each element, is the same as applying $f*$ to each component list and then concatenating the outcomes.

$$(\oplus/)*\circ +\!\!+/ = \oplus/\circ(\oplus/)\,* \qquad\qquad \text{(reduce promotion)}$$

which states that to reduce the components of a list of lists with an associative operator $\oplus$ we can either concatenate the component lists and reduce the results, or reduce each component list and then reduce the results.

A function h satisfying the following three equations is called a list homomorphism.

$$\begin{aligned} h\,[\,] &= \iota_{\oplus} \\ h\,[x] &= f\,x \\ h\,(xs \mathbin{+\!\!+} ys) &= h\,xs \oplus h\,ys \end{aligned}$$

where $\oplus$ is an associative binary operator with unit $\iota_{\oplus}$. We write $([f, \oplus])$ for the unique function h. Many important functions are defined as list homomorphisms.

The following result was first noted by Bird and Meertens and called the first homomorphism theorem [9].

The First Homomorphism Theorem Any homomorphism can be expressed in the form

$$h\,(xs \mathbin{+\!\!+} ys) = (\oplus/) \circ (f*)$$

2.3.3 Skillicorn's Cost Calculus for Parallelised BMF

BMF has good properties as a parallel programming model. Clearly $f*$ is the most basic form of parallelism. $\oplus/$ can be computed in parallel using the obvious tree-like structure. The first homomorphism theorem implies that any list homomorphism can be structured as a parallel algorithm consisting of two steps: a single parallel operation applied to each element followed by a tree-like reduction. Skillicorn established a methodology using BMF as a parallel programming model [77].

Programming Style

The programmer is provided with a set of aggregate data types (list, arrays, trees...), along with a set of predefined collective operations. The programmer can express his/her algorithms only by means of composition of the provided operations.

Implementation of the Model

The implementation of the model proposed reflects the structuring imposed by the Bird-Meertens theories. As the theory provides a small set of second order operators and all programs are compositions of these operators, the implementation of a program proceeds as a sequence of templates of the used operators.

Standard Topology

Because of the fixed set of operations and template-based implementation of the support, there is only a fixed set of communication and computation patterns that can occur. We need only to solve the mapping problem for the union of the patterns used in the templates, and then to use a combination of predefined templates to solve any problems. The union of the patterns needed is called the standard topology. Solving the mapping problem in this framework means embedding the standard topology in a target architecture. Observation of the templates implementing each operation reveals that the standard topology is given by a set of distributed memory MIMD processors whose interconnection satisfies the following requirements: the existence of a Hamiltonian cycle, the capability to do a tree-structured reduction in logarithmic time, and the ability to deliver an arbitrary permutation in logarithmic time. Skillicorn claims in [76] that such a standard topology can be mapped on any main class of massively parallel architectures with no more than a constant slowdown. That is, the cost of the emulation is asymptotically the same on those architectures. This result allows him to sketch an asymptotic performance prediction at the language level without considering communication characteristics of the target architectures.

Cost Calculus

Transformational derivations should be guided by some concept of the execution cost of the developing program. Skillicorn and Cai [78] present a strategy for building cost calculi which can be used for transformational program development. They take the following general approach. Whenever a composition $g \circ f$ has an implementation that is cheaper than the sum of the costs of g and f, define a new operation to represent the combined cheaper operation, thus:

$$newop = g \circ f$$

Both sides denote the same function. However, the left hand side denotes a single operation, while the right hand side means all processors must have completed f before any processors may begin computing g. The cost of the right hand side is the sum of the costs of composition. This view allows the cost of both sides to be computed and the equation to be labelled with its cost-reducing direction. For the theory of lists, a cost is computed as follows.

A list of length n is stored on a p-processor system with about n/p elements in each processor. Computing $f*$ applied to such a list means applying f sequentially to the n/p elements in each processor. The implementation equation for list map is

$$f*_n = (\overline{f*}_{n/p})*_p$$

where subscripts indicate the size of piece to which an operation applies and an overbar indicates the sequential version of an operation. Reductions are first done sequentially on the list segments stored in each processor, and the results are then reduced in parallel between processors. The implementation equation for list reduction is therefore

$$\oplus/_n = \oplus/_p \circ (\overline{\oplus/}_{n/p})*_p$$

For list map we have

$$t_p(f*_n) = \frac{n}{p}$$

and for list reduction we have

$$t_p(\oplus/_n) = \log p + \frac{n}{p}$$

The costs of other useful operations can be computed in a similar way and the cost information is used to direct equations of the theory. Cost results are parameterised by the number of processors used and the size of list elements, thus we can know the asymptotic behaviour of cost. Some translation laws which are cost-reducing directed based on the calculus are:

$$(f \circ g)* \quad \rightleftharpoons \quad f * \circ g * \tag{2.1}$$

$$f * \circ (+\!\!+ /) \quad \rightharpoonup \quad (+\!\!+ /) \circ (f*)* \tag{2.2}$$

$$(\oplus/) \circ (+\!\!+ /) \quad \rightharpoonup \quad (\oplus/) \circ (\oplus/)* \tag{2.3}$$

$$\oplus /\!\!/ \quad \leftharpoondown \quad (\oplus/) * \circ \, inits \tag{2.4}$$

$$inits \quad \leftharpoondown \quad (+\!\!+ /\!\!/) \circ ([\cdot])* \tag{2.5}$$

$$\otimes/_{id_\otimes}\oplus \quad \leftharpoondown \quad (\oplus/) \circ (\otimes/) * \circ \, tails \tag{2.6}$$

$$\otimes/\!\!/_{id_\otimes}\oplus \quad \leftharpoondown \quad (\otimes/_{id_\otimes}\oplus) * \circ \, inits \tag{2.7}$$

where *recur-reduce* (written $\otimes/_{b_0}\oplus$), given coefficients $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ computes the nth value generated by a linear recurrence function $x_{i+1} = x_i \otimes a_{i+1} \oplus b_{i+1}$ where $x_0 = b_0$, $\otimes$ and $\oplus$ are associative, and $\otimes$ distributes backwards over $\oplus$:

$$\begin{aligned} & [a_1, \ldots, a_i, \ldots, a_n] \otimes /_{b_0} \oplus [b_1, \ldots, b_i, \ldots, b_n] \\ = \quad & b_0 \otimes a_1 \otimes \ldots \otimes a_n \oplus \ldots \oplus b_{i-1} \otimes a_i \otimes a_{i+1} \otimes \ldots \otimes a_n \oplus \ldots \oplus b_{n-1} \otimes a_n \oplus b_n \end{aligned}$$

and *recur-prefix* (written $\otimes /\!\!/_{b_0} \oplus$) computes all values generated by the same linear recurrence:

$$\begin{aligned} & [a_1, \ldots, a_i, \ldots, a_n] \otimes /_{b_0} \oplus [b_1, \ldots, b_i \ldots, b_n] \\ = \quad & [b_0, b_0 \otimes a_1 \oplus b_1, \ldots, b_0 \otimes a_1 \otimes \ldots \otimes a_n \oplus \ldots \oplus b_{i-1} \otimes a_i \otimes a_{i+1} \otimes \ldots \otimes a_n, \ldots, \\ & b_0 \otimes a_1 \otimes \ldots \otimes a_n \oplus \ldots \oplus b_{i-1} \otimes a_i \otimes a_{i+1} \otimes \ldots \otimes a_n \oplus \ldots \oplus b_{n-1} \otimes a_n \oplus b_n] \end{aligned}$$

and when $[b_1, \ldots, b_n] = [id_\otimes, \ldots, id_\otimes]$ and $b_0 = id_\otimes$, where $id_\oplus$ is the identity element of operator $\oplus$, we write

$$\begin{aligned} & [a_1, \ldots, a_n] \otimes /_{b_0} \oplus [b_1, \ldots, b_n] \text{ as } \otimes /_{id_\otimes} \oplus [b_1, \ldots, b_n] \\ & [a_1, \ldots, a_n] \otimes /\!\!/_{b_0} \oplus [b_1, \ldots, b_n] \text{ as } \otimes /\!\!/_{id_\otimes} \oplus [b_1, \ldots, b_n] \end{aligned}$$

Transformational Development

A set of transformation laws allows the programmer to transform the programs from a first (possibly inefficient) formulation to a more efficient implementation. Transformational development of BMF is illustrated by an example problem *maximum segment sum*(mss). The problem is: given a list of integers, find the greatest sum of values from a contiguous sublist. It begins from an obviously correct solution: compute all of the subsegments, sum the elements of each, and select the largest of the sums. It can be expressed in BMF style by

$$mss = (\uparrow /) \circ (+/) * \circ\, segs$$

where $\uparrow$ denotes the function which takes the maximum of two arguments and *segs* is defined by using *tails* and *inits*:

$$segs = (+\!\!+ /) \circ tails * \circ\, inits$$

As an example, we have

$$mss[2, -4, 2, -1, 6, -3] = 7$$

In [78], Skillicorn and Cai derived an parallel algorithm from the specification:

$$\begin{aligned}
mss &= \{\text{definition}\}\\
&\quad (\uparrow /) \circ (+/) * \circ\, segs\\
&= \{\text{by definition}, segs = (+\!\!+ /) \circ tails * \circ\, inits\}\\
&\quad (\uparrow /) \circ (+/) * \circ +\!\!+/ \circ tails * \circ\, inits\\
&= \{\text{Eq. }(2.2), \text{cost} - \text{reducing}\}\\
&\quad (\uparrow /) \circ (+\!\!+ /) \circ ((+/)*) * \circ tails * \circ\, inits\\
&= \{\text{Eq. }(2.3), \text{cost} - \text{reducing}\}\\
&\quad (\uparrow /) \circ (\uparrow /) * \circ ((+/)*) * \circ tails * \circ\, inits\\
&= \{\text{map promotion, Eq. }(2.1), \text{cost} - \text{neutral}\}\\
&\quad (\uparrow /) \circ (\uparrow \circ (+/) * \circ\, tails) * \circ\, inits\\
&= \{\text{Eq. }(2.6), \text{cost} - \text{reducing}\}\\
&\quad (\uparrow /) \circ (+/_0 \uparrow) * \circ\, inits\\
&= \{\text{Eq. }(2.7), \text{cost} - \text{reducing}\}\\
&\quad \uparrow / \circ (+/\!\!/_0 \uparrow)
\end{aligned}$$

Note that the 0 subscript of $/$ and $/\!\!/$ is the identity element of $+$. The derived algorithm has complexity of $O(\log n)$ under the condition that n processors are available.

While BMF gives a formal foundation of transformational development, it has some aspects which might be seen as drawbacks. Firstly, the expression is restrictive because parallelism is introduced by only predefined second-order functions. Secondly, it is difficult to implement efficiently when the second-order functions are composed. Finally, successful cost-reduced transformations are often not easy to find out in the general case.

2.3.4 Compile-time Cost Analysis for HOPP

Rangaswami [71] has developed a compile-time cost analysis for a parallel programming model called Higher-Order Parallel Programming (HOPP). In the HOPP model, parallelism in programs is expressed implicitly using the fixed set of BMF functions. Its cost analysis aims to exploit a more concrete model than Skillicorn's cost calculus, considering lower level information such as architecture topologies and bandwidth of communication links. It also estimates the costs of different possible implementations of nested higher-order functions, in contrast to Skillicorn's one in which only parallelism at the level of the outermost higher-order function is handled, to choose the most cost-effective one. The HOPP model consists of three components: the program model, the machine model and the cost model.

The Program Model

The program model is similar to that of Skillicorn's parallelised BMF. It has predefined (parallel) recognised functions, which are second-order functions from BMF, some additional functions having parallel implementations, and user-defined sequential functions. A program is expressed only as a composition of those functions. Each component of the composition is referred to as a *phase* of the program. The only data structure is the list, on which all the functions operate. Lists can be arbitrarily nested and any type. Since the behaviour of each of the functions is predetermined, a *regular* program expressed in terms of these functions can be analysed at compile-time. A regular problem in this context is one whose behaviour does not depend on the actual input values. A further assumption which is made by the analyser is that sublists are of equal length. The cost analysis needs type information in order to compute communication costs. Consequently definition of sequential functions that allow polymorphism is not permitted.

The Machine Model

The programs are targeted at distributed-memory machines which consist of a set of interconnected processors. The machine model provides a range of target architectures, on which the cost model predicts execution cost of the program. It includes hypercube, 2-D torus, linear array and tree.

The Cost Model

In the HOPP model, the parallelism is exploited by the occurrence of recognised functions in each phase. The phases themselves are sequential so that phase i does not commence until phase $i-1$ is completed. The cost system examines cases in which the nesting level is less than three. Any recognised function more than four levels deep is considered as a sequential function. The cost analysis was implemented in the form of an analyser. The application program is input to the analyser which first constructs a program tree. Each branch in the tree corresponds to a phase of the program.

The information used in the analyser is in the form of the following tuple:

$$\text{program} = (P, M, D, I_s, F_t, S, C_f, F_s)$$

where
P is a program tree.
M is a 4-tuple which describes the characteristics of the parallel machine.

$$M = (\text{topology, number-of-processors, start-up cost, bandwidth})$$

D represents the level of nesting of the input lists.

$$D = \text{(variable of the input list, level of nesting of input list)}$$

I_s is a D-tuple which represents the list sizes at each level.
F_t is a function that computes the size of each element in level $(D-1)$ of the input list(s).
S is a set of relationships between sizes in different levels of the input list and the number of processors.
C_f is the cost of the sequential function.
F_s is the output type of the sequential function.

Although this information is supplied by a programmer, D, F_t, and F_s could be deduced from the type if the analyser incorporated a type-checker. C_f and I_s could be estimated if the analyser incorporated a profiler. The start-up cost and the bandwidth are specific to a given architecture and can be obtained from the machine manufacturer.

The cost of a program comprising n phases is given by:

$$Cost = \sum_{i=1}^{n} C_{pi} + \sum_{i=0}^{n-1} C_{i,i+1}$$

where C_{pi} is the cost of phase i and $C_{i,i+1}$ is the communication cost for rearranging the output of phase i to suit the implementation of phase $i+1$ when necessary.

The cost of implementing a recognised function, F, operating on an input list of sizes n, in parallel on p processors, is represented by:

$$C = F(n, p, C_f)$$

where, C_f is the cost of F's argument function. The analyser performs a cost analysis for each phase in the program, for a given topology. The cost of the sequential implementation is also computed in each case. The phase could contain up to three nested recognised functions. Seven possibilities arise, corresponding to the implementation of any one of the three functions in parallel, any two in parallel and all three in parallel.

The Cost of Parallel Functions

The recognised functions in HOPP have their definition of execution cost on each architecture topology considered. For example, the cost of map on any topology is

$$C_{map} = \frac{n}{p} C_f$$

where f is the sequential argument function and C_f is its cost. There are two versions for fold, that is s_fold and g_fold. The size of the intermediate results is constant in s_fold, but it grows in g_fold. The algorithms for the two versions are the same: each processor performs the sequential fold on its local elements, then the partial results are combined globally to obtain the final result. But the communication costs are different. For example, the cost of the both versions on a hypercube of dimension d is given by

- s_fold

$$C^h_{s\_fold} = C_f(\frac{n}{p} + d - 1) + T^1_{com} d$$

- g_fold

$$C^h_{g\_fold} = C_f(\frac{n}{p} + d - 1) + \sum_{i=0}^{d-1} T_{com}^{2^i \frac{n}{p} m}$$

where T^m_{com} represents the cost of communicating m elements of the lists to a neighbour.

$$T^m_{com} = K_0 + \frac{1}{K_1} ms$$

where s is the size of each element of the list, K_0 is the start-up cost, and K_1 is the bandwidth of the communication link.

Data Rearrangement Communication Cost

The rearrangement costs are computed using the information of the current data distribution which is obtained from the current node in the search tree and the required data distribution. Five models of data rearrangement communication were identified. As example, we give the each cost of them on the hypercube of d dimensions.

- Nearest Neighbour: The cost for communicating a data packet of size n bytes to a nearest neighbour is

$$K_0 + \frac{1}{K_1} n$$

- Broadcast: The cost for sending a data packet of size n bytes to all the other processors is

$$C^h_{broad} = d(K_0 + \frac{1}{K_1} n)$$

- Scatter: The cost for scattering data of size n bytes equally to the other processors is

$$C^h_{scatter} = dK_0 + \frac{1}{K_1} \frac{n}{p} (p-1)$$

- Gather: The cost for collecting distributed data of size n bytes across the processors is the same as that of scattering.

- Total Exchange: The cost for sending data of n bytes from every processor to every other processor is

$$C^h_{exchange} = 2dK_0 + 2\frac{1}{K_1} \frac{n}{p} (p-1)$$

The Search Tree

The costs predicted by the analyser are used to construct a search tree to realise cost-effective parallel implementation for a given architecture. The cost of all possible implementations for each of the phases are estimated by the analyser and a search tree is constructed. The weights on the nodes at a level represent the costs associated with the different implementations for the corresponding phase. The weights on the edges represent the costs of phase transition. The least-cost path in the search tree corresponds to the most efficient implementation for the whole program, for which code can then be generated and executed on the parallel machine.

2.3.5 VEC and Shape-based Cost Analysis

Static shape analysis to support compilation and cost prediction for parallel programs was originally suggested by Jay [52] and first applied in detail to the cost analysis of VEC, a small shapely functional language [55]. VEC supports a new account of arrays that combines the benefits of the list programming style with the efficiency of array programming, by means of shape analysis. [55] represents the first attempt to produce a formal cost calculus for a parallel programming language of nested arrays that automatically derive costs from program source. As the paper [55] by Jay et al. is the most fundamental previous work for this thesis, we review the large part of its contents here for the purpose of both giving the basis of our work and making clear the difference with our work presented in the following chapters.

Programming Model

VEC by Jay et al. is a simply-typed lambda-calculus with products, a unit type, and a vector type constructor for nested arrays that supports the BMF style of programming. Its types are

$$
\begin{aligned}
D &::= \mathsf{nat} \mid \mathsf{bool} \mid \dots \\
\tau &::= D \mid \mathsf{sz} \mid \mathsf{un} \mid \tau \times \tau \mid \mathsf{vec}\,\tau \\
\theta &::= \tau \mid \theta \times \theta \mid \theta \to \theta
\end{aligned}
$$

where D can include other simple *datum* types, and the type stratification precludes vectors of functions. The terminology of vectors (rather than lists) is used to emphasise the fact that the lengths of such objects will be statically determinable even though the language syntax itself will use the familiar nomenclature of lists. Type sz is introduced as the type of lengths of vectors and index vector entries, although the set of the values of sz is isomorphic to the set of natural numbers. Type un is a unit type.

Terms in VEC are given by

$$
t ::= d \mid c \mid x \mid \lambda x^{\theta}.t \mid t\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{ifs}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{rec} f^{\delta}.t
$$

where d ranges over simple constants (integers, booleans, arithmetic operations, etc) and c ranges over the combinators with non-trivial shapes (i.e. those whose behaviour impacts upon the shape of terms, such as vector constructors, a selection of conventional sequential functional operators such as length, fst, snd, hd, tl, entry and so on, and second-order vector operations like map, fold and zip). There are two forms of conditional: a data conditional if , whose condition is given by a datum; and a shape conditional ifs, whose condition is a size (with ˜0 interpreted as false, other sizes as true). The data conditional allows the condition to be data-dependent, but ensures shapeliness by requiring that the branches have the same shape, drawn from the shapes of terms in τ above. By contrast, the branch taken by the shape conditional ifs is known by shape analysis, so the branches may have arbitrary types and shapes.

The superscript on the recursion operator indicates restriction to functions with trivial shape (i.e. involving only datum types and their composition, but no vectors). This guarantees termination of shape analysis, but requires the programmer to estimate the number of unfoldings for recursive functions to perform cost analysis. Shape rules (analogous to type rules) ensure that all elements of a vector have the same shape. Collectively, the constraints ensure that the compiler is able to evaluate the shape of every program, and detect any shape errors. In turn, this facilitates efficient implementation since vectors can be implemented as arrays rather than lists. In VEC, shape $tycost_M$ of a type θ is defined inductively by

$$
\begin{aligned}
tycost_M(D) &= \mathsf{un} \\
tycost_M(\mathsf{un}) &= \mathsf{un} \\
tycost_M(\theta \times \theta') &= tycost_M(\theta) \times tycost_M(\theta') \\
tycost_M(\mathsf{vec}\,\theta) &= \mathsf{sz} \times (tycost_M(\theta)) \\
tycost_M(\mathsf{sz}) &= \mathsf{sz} \\
tycost_M(\theta \to \theta') &= tycost_M(\theta) \to (tycost_M(\theta') \times T)
\end{aligned}
$$

The shape of an object whose type is D or un is bang, which is defined as the canonical term of *discrete* type δ, which are constructed without using the vector construction or sz:

$$\delta ::= D \mid \mathsf{un} \mid \delta \times \delta \mid \delta \to \delta$$

The shape of a pair is a pair of the shape of each component and the shape of a vector is a pair comprising its length and the common shape of its elements. The shape of a function is a function from the shape of an argument to a pair comprising the shape of the result and the cost to apply it, whose type is T from a cost algebra, which is explained below.

There are intrinsic limitations of expressiveness in the shapely programming model. In addition to the restriction about data conditional mentioned above, it excludes any use of functions whose result shape is data-dependent. The well known function *filter* is a typical example.

Cost Algebras

The cost analysis is founded on the concept of a *cost algebra*, which captures the characteristics which determine execution cost on some target architecture and mechanisms for the combination of such costs. A cost algebra has signature $(T,+,0,\oplus,\otimes,\mathsf{max})$ with binary operations

$$\begin{aligned} +,\oplus &: T \to T \to T \\ \otimes &: \mathsf{sz} \to T \to T \\ \mathsf{max} &: T \to T \to T \end{aligned}$$

where T represents execution costs of some kind. The operations $+$ and $\oplus$ are sequential addition and parallel addition of costs (in other words, capturing what happens when two computations are respectively run sequentially and concurrently). The operation $\otimes$ is parallel "multiplication" capturing the notion of running a number of copies of the same computation concurrently. 0 is the identity element of $+$. The operation max takes some kind of maximum between two costs.

Cost Calculus

Cost analysis of terms is structured compositionally, analogously to the shape analysis of earlier work. For example, where the shape of a term which reduces to a vector consists of a pair comprising the length of that vector and the (necessarily common) shape of its elements, the cost analysis computes both the shape and an element of T (from the cost algebra) corresponding to the cost of the reduction itself. That is, the analysis of a term whose type is θ produces an object of type

$$tycost_M(\theta) \times T$$

The analysis of a datum constant term defines

$$cost(d) = \langle \mathsf{bang}, 0 \rangle \qquad \text{where } d \text{ is a datum constant}$$

As the shape of a function term is a function from the shape of an argument to the pair of the shape of the result and the cost to apply it, the analysis of a function term (itself of type $\theta \to \theta'$) produces an object of type

$$(tycost_M(\theta) \to (tycost_M(\theta') \times T)) \times T$$

where, $tycost_M(\theta)$ and $tycost_M(\theta')$ reflect the shape behaviour of the function. The outermost T reflects the cost of reducing the term itself, while the inner T reflects the cost of applying the function. For example, the analysis of a binary datum operation term defines

$$cost(d) = \langle \lambda x. \langle \lambda x. \langle \mathsf{bang}, \mathsf{binOpConst} \rangle, 0 \rangle, 0 \rangle \qquad \text{where } d \text{ is a binary datum operation}$$

As examples of the analysis of purely sequential operation terms, the analysis of the terms hd (the usual "head of a list" function, but now as a vector operation) and length define

$$cost(\mathsf{hd}) = \langle \lambda x. \langle \mathsf{snd}\ x, \mathsf{hdConst} \rangle, 0 \rangle$$

and

$$cost(\mathsf{length}) = \langle \lambda x. \langle \mathsf{fst}\ x, \mathsf{lengthConst} \rangle, 0 \rangle$$

respectively, where fst and snd are the functions which take a pair and return the first and the second component of the pair, respectively. These indicate that the term itself costs nothing to evaluate and that the head (or length) function costs some machine dependent constant quantity of time to execute, producing a result whose shape is that of the elements for hd (or the length for length) of the vector to which it is applied. A more interesting example is the second-order function map.

$$cost(\mathsf{map}) = \langle \lambda f.\langle \lambda x.\langle \langle \mathsf{fst}\, x, \mathsf{fst}\,(f(\mathsf{snd}\, x))\rangle, (\mathsf{fst}\, x) \otimes\ (\mathsf{snd}\,(f\,(\mathsf{snd}\, x)))\rangle, 0\rangle, 0\rangle$$

This indicates that the term itself costs nothing to evaluate and that applying map to an argument function costs nothing but application of the resulting function to an argument vector costs some quantity of time that depends on the definition of $\otimes$ in the cost algebra, producing a result whose shape is a pair of the length of the vector and the shape of the result of the application of the function to the element of the vector.

The cost-accounting of parallelism in [55] reflects the implementation choices which were assumed. Firstly, the skeletal combinators (such as map) were assumed to introduce parallelism in the conventional way. Secondly, function application terms (i.e. terms of the form $\mathtt{t}_1\ \mathtt{t}_2$) were assumed to be implemented first by evaluating $\mathtt{t}_1$ and $\mathtt{t}_2$, possibly in parallel, then evaluating the application itself. The $\oplus$ of the cost algebra captures the first of these two stages (and its implicit compile-time optimisation). Thus, the analysis of application terms of the form $\mathtt{t}_1\ \mathtt{t}_2$ is performed by applying the corresponding shape-cost pair of $\mathtt{t}_1$, $\langle f, t\rangle$ to that of $\mathtt{t}_2$, $\langle x, t'\rangle$ by using a SIZE operator capp:

$$cost\ (t_1\ t_2) = \mathsf{capp}\ cost\ (t_1)\ cost\ (t_2)$$

where capp is defined by

$$\mathsf{capp} : (\theta \to (\theta' \times T)) \times T \to (\theta \times T) \to (\theta' \times T)$$
$$\mathsf{capp}\,\langle f, t\rangle\langle x, t'\rangle = \langle \mathsf{fst}\,(f\,x), (\mathsf{snd}\,(f\,x)) + (t \oplus t')\rangle$$

This implies that the cost of an application term is a combination of the application cost $(\mathsf{snd}(fx))$ and the cost of function term t and cost of argument term t'. $+$ and $\oplus$ can be changed to reflect the definition of a cost algebra.

Notice that two terms which reduce to the same value (and hence have the same shape) can have different costs, depending upon the method by which they are computed (e.g. which parallel operators are used, if any). Consider terms t_a and t_b which evaluate to the same vector of length n. Suppose t_a computes its result in parallel, while t_b is entirely sequential. The costs of the terms will take similar forms $\langle\langle n, \mathsf{bang}\rangle, t'_a\rangle$ and $\langle\langle n, \mathsf{bang}\rangle, t'_b\rangle$ indicating that both results have the same shape $\langle n, \mathsf{bang}\rangle$. Meanwhile, the cost functions t'_a and t'_b are distinct, distinguishing the implementations.

Implementation Model

The choice of implementation model is made by the definition of operations of a cost algebra without changing any other details of the cost analysis framework. For example, sequential executions are costed using the cost algebra $(T, +, \tilde{}0, +, *, \mathsf{max})$ in which T is sz and simply counts clock ticks, and the other components are the standard integer operations. For parallel execution, T is a set of functions from parallel machine descriptions to times. The chosen parallel model in [55] was the PRAM model. For the PRAM model, with its collection of processors computing synchronously in parallel and interacting through a unit access-time shared memory, we have $T = \mathsf{sz} \to \mathsf{sz}$, representing time functions from the number of processors to the number of time steps. Sequential cost addition is pointwise addition on time functions, and max is pointwise maximum. An addition for parallel execution $\oplus'$ is defined by using static cost information to determine an optimal division of processors between two parallel tasks (since this information would also, of course, be available to the compiler).

$$(f \oplus' g)\, p = \min_{0<q<p} \{\max\{f\, q,\ g\,(p-q)\}\}$$

Because sequential execution may be faster, parallel addition $\oplus$ in the cost algebra is defined by

$$(f \oplus g)\, p = \min\{(f+g)\, p, (f \oplus' g)\, p\}$$

Parallel multiplication $\otimes$ is defined by

$$\begin{array}{rcl} (n \otimes f) & = & \mathsf{if}\ (n \,\mathsf{mod}\, p == \tilde{}0) \\ & & \mathsf{then}\ (n \div p) * (f\ \tilde{}1) \\ & & \mathsf{else}\ (n \div p) * (f\ \tilde{}1) + (f\ (p \div (n \,\mathsf{mod}\, p))) \end{array}$$

Notice that static shape information is used to divide the "leftover" tasks among the processors to increase efficiency, just as a compiler in possession of this information would do. The skeleton combinators which are executed in parallel in VEC are map and a parallel fold, pfold.

In subsequent work [54], Jay raises the possibility of BSP costing of GoldFiSh (a related parallel shapely language), but no attempt is made to capture matters formally.

2.3.6 Related Work with Skeletons

There are currently a number of research groups working on the design and implementation of parallel languages with algorithmic skeletons.

Early work in the area of using algorithmic skeletons concentrated on describing each program using a single skeleton. Cole introduced in [25] the skeletal concept, defining four general skeletons: *divide & conquer, task queue, iterative combination*, and *cluster* (solving problem by decomposition on a grid network).

Subsequent work by various groups has been addressing the complications that arise by allowing the composition and nesting of algorithmic skeletons.

Darlington's group at the Imperial College has been one of the most prolific in this field. Their first approach was to embed a set of general skeletons, including *pipe, farm, d&c, and ramp* in a purely functional language [29]. This was followed by a refined approach, called $SPP(X)$, standing for Structured Parallel Programming parameterised by a base language X. $SPP(X)$ is a two-layer scheme, comprising a high-level, functional language, called the Structured Control Language (SCL), in which applications (containing skeleton calls) are written, and a Low-level Base Language for efficient sequential code called from within the skeletons [30]. The skeletons presented in this context are both general ones, like *farm* and *SPMD*, and data-parallel ones working on distributed arrays, like *map, fold*, and *rotate*. Although the language and prototype implementation supports programs consisting of many skeletons, the focus of the preliminary implementation is on transforming and optimising individual skeletons. To [81] addressed this issue by investigating the possibility of using cost functions, which are to be derived from skeleton performance models, in optimising the implementation of compositions of the components built using the approach.

The Pisa Parallel Programming Language (P3L) [67, 28], group led by Pelagatti and Danelutto has been similarly active for a number of years. P3L is an imperative-based (typically C) programming language that supports a set of predefined programming templates, or skeletons. These include *farm, pipe, map, tree reduction* and *loop*, each of which has an associated functional language definition. The language allows unrestricted composition and nesting of these skeletons in a user program. Each of the P3L implementation templates is associated with a performance model function parameterised by both machine and application specific parameters. The compilation philosophy employs templates for each construct, targeting the P3L abstract machine, a distributed-memory, message passing model with options for either full or mesh connectivity. Many decisions are guided by the use of profiling information gathered sequentially and plugged into the template performance models. Zavanella [86] describes the methodology to implement an adaptive support for a skeleton language (Skel-BSP) on top of the EdD-BSP (a simple extension of the BSP) computer as a method to provide both efficiency and performance portability.

The Heriot-Watt group have extracted and exploited skeletal parallelism within Standard ML programs. Busvine's PUFF compiler [24] generates sequential occam2 from SML and can identify useful parallelism in general linear recursion. Bratvold's SkelML compiler [18] recognises a set of predefined higher-order functions, or skeletons, including *map, filter*, and *fold* in standard ML programs and maps their implementations to abstract process network templates. The compiler also uses a set of preoptimised implementation templates for recognised sequential compositions of the supported skeletons. A performance model formula for each of the skeletons is derived by the compiler designer, and quantified by benchmarking. Using these formulae, the compiler compares the costs of different process to processor mappings in an attempt to minimise the total execution time of the program. The compiler relies on profiled sample data provided by the programmer to obtain information about the execution time of the user-supplied sequential code. Michaelson et al. [61, 62, 75] have developed a parallelising compiler for Standard ML using algorithmic skeletons including *map* and *fold*. While PUFF and SkelML are compilers from a subset of Standard ML to occam2 and are oriented specifically to the Meiko Computing architecture, based on T800 transputers, the new work generates predictably portable C with MPI from Standard ML. Arbitrary depth nesting of skeletons can be implemented in parallel using a static approach for generating parallel code. Hamdan's Ektran compiler [43] can also compile and execute arbitrarily nested skeletons. The static analysis of the source program generates a nesting structure which is used to combine the corresponding higher-order functions in a process termed "nesting deduction". The run-time scheduling relies on the compile-time analysis and uses message passing groups to run combined higher order functions in parallel.

Rabhi and Schwarz [69, 70], have developed a Paradigm Oriented Programming Environment (POPE) in which a purely functional realisation of the "static iterative transformation" skeleton is added to skeletons similar to those found in Cole's original approach.

Feldcamp et al. developed Parsec (Parallel System for Efficient Compilation) [33, 34], which is a skeleton-based parallel programming environment. The system provides virtual machines (called skeleton-template-module objects) which provide skeleton code for the supported template, which the user completes to implement an application. The supported skeletons are processor farms and divide and conquer. Each skeleton is parametrised on information such as number of processors, topology and granularity. The parameters include both static information such as its shape and size that can be specified by the user and dynamic performance tuning parameters that are determined by the analysis from information gathered from test runs and the performance model provided to each skeleton. The performance model was validated on a 74 node T800 Transputer based multicomputer system.

Deldarie et al. [31] developed special cases of skeletons related to image processing. The provided skeletons are *local window* (LW), where each pixel in the resulting output is derived from pixels in a window surrounding the corresponding pixel in the input image, and *split and merge* (SAM) where an image is partitioned into slices, an operation is applied to each slice then the results are merged. Performance models for the skeletons are derived in terms of the WPRAM computational model [65] and the execution time for a skeleton is presented as a generic higher order complexity function.

The time complexity of the particular application is derived when the skeleton with a specific set of parameters is instantiated. The approach is illustrated by some examples from image processing, and is extended to analyse the scalability of skeleton-based applications, using isoefficiency functions [42] . Measured performance on the WPRAM simulator shows a close match to theoretical predictions.

The recent work of Gorlatch et al. [39, 40] present a methodology for designing message passing programs with collective operations, such as reduction, scan, gather, etc. The design process is based on correctness-preserving transformation rules, provable in a formal functional framework. The impact of the design rules on the target performance is estimated analytically and tested in the machine experiments. The methodology is illustrated by a case study, the MPI implementation of the maximum segment sum problem, starting from an intuitive but inefficient algorithm specification.

The Skeletons Imperative Language (Skil) [16, 17] has been developed by Botorog and Kuchen. Skil is an imperative language aiming at integrating skeletal functional features with the efficiency of the C language.

NESL [14] developed by Blelloch is a data-parallel strict functional language, which has an ML-like syntax and supports polymorphism. The language based performance model gives a formal way to calculate the *work* (total number of operations) and *depth* (longest sequence of dependences, or critical path) of a program and defines rules for composing these costs across expressions. These measures can be related to running time on parallel machines.

Loidl [56] gives granularity analysis for a simple strict higher-order functional language. The purpose of this analysis is to statically derive information about computation costs that can be used by the parallel runtime-system to improve performance. Static analysis is based on the sized time system, which is a combination of the inference system developed by Reistad and Gifford [73] and sized types developed by Hughes et al. [51], to propagate information about sizes and costs.

Brinch-Hansen [19] presents a number of independent imperative skeletal case-studies.

2.3.7 Profiling Approach and Recursion Analysis

Apart from skeletal restrictions, our language VEC-BSP imposes additional restrictions to achieve static cost predictability and automatability of cost analysis. For example, we exclude non-shapely functions like *filter* whose result shape depends on input data, and recursive functions, whose termination and cost are not decidable in the general case. As these restrictions are strict, our static cost analysis in this thesis could be incorporated with more experimental approaches for a more practical analysis system. Those approaches such as profiling methods and recursion analysis themselves have been long known as important research areas. The remainder of this chapter summarises some related works of these issues.

Profiling Approach

One approach for extracting information about the performance of program is to execute it with some sample input and to generate profiling information. This information is then fed back into the program development or compilation process and can be used to generate more efficient code. Ideally, predicting the execution time solely by static analysis is preferable because a programmer or a compiler can make all decisions based on source code. In contrast, information of the profiling method depends on the choice of the initial input set. If the run-time behaviour of the program varies much between different inputs, to reach a good result without a large-compile time is difficult and, the choice of good sample input is not obvious in general. However, as the execution time of a program is not a decidable property and information of input data is desirable for accurate prediction in some case, for example at the branching points in the program, introducing a profiling approach would be indispensable in more practical use. Good examples of the profiling approach combined with a skeleton-based approach can be seen in the works of the Heriot-Watt group. In Busvine's PUFF compiler [24], the program is run on one or more sets of data, collecting statistics about computation costs and execution frequencies. This information is used to transform the program into a parallel version that has improved performance. Bratvold's SkelML [18] is a skeleton based parallelising compiler, which is based on sequential program instrumentation through Structural Operational Semantics (SOS) [64]. Skeleton performance models are instantiated using the SOS measures to determine useful parallelism. Michaelson et al. [61] present the design of an architecture-independent parallelising compiler for SML in which these costs are parameterised over machine specific parameters, so that instantiating these parameters and combining the profiling information can give accurate granularity information.

Recursion Analysis

Functions are said to be defined recursively when the body of the definition refers to the function itself. We usually demand that recursive definitions are terminating, i.e. given some particular input the function will call itself only a finite number of times before stopping with some output. In general, however, there is no guarantee that a function defined by recursion will always terminate. The usual approach is to provide the user with a pre-defined set of well-founded induction schemes. To use a scheme not specified in this set, the user must specify an ordering and prove that this orderings is well-founded. There are possible constraints on recursion to aid analysis of termination. The simplest way to ensure termination is to forbid recursion. This would give a restrictive language. Another alternative is to restrict the recursive function to be primitive recursive, as all primitive recursive programs terminate with easily characterisable time and space behaviour. There are functions that are not primitive recursive to which we cannot in any simple way give an upper bound for the number of reductions needed when applying it to an argument.

Burstall [22] contributed structured recursion, a generalised form of primitive recursion, to analytic syntax, with an associated principle of structural induction. Burstall [23] also showed that if the recursion is combined with a case expression which decomposes elements of the data type, the ordinary scoping rule for variables can be used to ensure termination, without imposing any special schema.

Abel [1] has introduced a language based upon lambda calculus with products, coproducts and strictly positive inductive types that allows the definition of recursive terms. Their termination checker foetus ensures that all such terms are structurally recursive, i.e. recursive calls appear only with arguments structurally smaller than the input parameters of terms considered.

Walter [84] has described reduction checking, which is sometimes referred to as Walter recursion. His estimation calculus examines whether functions are terminating and also whether the output of the function is smaller than the input. This information can be used to check termination of nested recursive functions.

More work on the estimation calculus has been done by Bundy and others. Recursion editor [20] is an editor for Prolog that only allows terminating definitions, which ensure the termination of severely restricted kinds of recursive procedures chosen from Peter's classification. More recently, in CYNTHIA [85], an editor for a subset of ML, which grew out of work on recursion editor, each ML function definition is represented as a proof of a specification of that function using the idea of proofs-as-programs [49]. The proof is written in Oyster [21], a proof-checker implementing a variant of Martin-Löf Type Theory. CYNTHIA restricts the user to the set of Walter recursive functions, which includes primitive recursive functions over an inductively-defined data types, multiple recursive functions, nested recursive functions and functions that reference previously defined functions in a recursive call. It analyse the termination of the program and gives useful feedback.

Telford and Turner [80] are investigating Elementary Strong Functional programming, i.e., functional programming where only terminating functions can be defined. They use abstract interpretations to ensure termination. They can handle a wider class of functions than Walters recursion since they keep track not of whether an argument is decreasing but how much it is decreasing or increasing, thus allowing temporary growth that is compensated by sufficient shrinkage later.

Related works on deriving cost information (statically or experimentally) and the treatment of recursion include the following.

Busvine's PUFF [24], which compiles SML to occam2, uses instrumentation to identify useful parallelism in linear recursion.

The ACE system of Le Métayer [60] transforms an FP program with call-by-name semantics into a program with call-by-value semantics. This performs a macro-analysis, that is, it measures the time in the number of applications of the dominant operation which is used in the program. He uses a set of rewrite rules to derive complexity functions, simplify them and finally eliminate recursion.

Huelsbergen et al. [50] were able to handle recursion successfully by using abstraction. They have defined an abstract interpretation of a higher order, strict language for determining computation cost, which uses dynamic estimates of the sized data structure. Their analysis uses the well-known trick of iteration in the abstract interpretation stops as soon as a certain bound for the computation costs of an expression is surpassed. This prevents non-termination in the analysis.

Rosendahl [74] presents a program transformation that yields a time bounded program for a given first-order Lisp program. His system deals with recursive functions by providing a set of translation rules that eliminate recursion.

Loidl and Hammond [57] present an inference system to determine the cost of evaluating expressions in a strict purely functional language. Upper bounds can be derived for both computation and size of data structures. The analysis is a synthesis of the sized system of Hughes et al. [51], and the time system of Dominic et al. [32], which was extended to static dependent costs by Reistad and Gifford [73]. Sized types can also be used to analyse the costs of user-defined recursive functions.

2.4 Chapter Conclusion

We surveyed some of the models for parallel computation. Table 2.1 summarises the cost modelling aspects of models and languages described in this chapter. Models which do not account for cost are omitted from the table. Many of the models use measurement analysis based on some benchmark data that is specific to a given architecture or profiling information obtained by running the input sets of data on a given architecture. Although typical cost modelling for skeletons includes parameters on some communication performance such as bandwidth and start-up cost, some models such as PRAM and VEC have no consideration of communication cost, to reduce the complexity of cost analysis. Only VEC and VEC-BSP have full-automatic static analysis. Some BMF style models and Darlington's skeletons discuss application of their cost models to program translation. Many models that support analysis tools have done accuracy tests but some theoretical models have not. A few models present a cost model which accounts for the costs of the implementation of programs in which some degree of nested skeletons is allowed.

Table 2.1 includes VEC-BSP, which will be presented in the next chapter. Its static cost analysis is developed building on Jay's cost calculus since it has a formal analytic framework and the characteristic of being automatable. The implementation model for VEC-BSP is BSP, whose level of abstraction is lower than the PRAM and Skillicorn's model but higher than Rangaswami's model. Communication performance characteristics of a target machine are considered in terms of the BSP parameters, but the architecture topology is abstracted. Accuracy tests have been performed for several examples. The analysis can predict the absolute value of execution cost based on the BSP benchmark. We will discuss application of our automated cost analysis to program derivation steps. The analysis deals with parallelism only at the level of the outermost higher-order function. Optimisations considering possible implementations of nested skeletal combinators remains a topic for future work.

Model Language	Meas. Asym.	Stat. Prof.	Comm. Sen.	Accur. Test	Prog. Trans.	Nest. Skel.
PRAM	Asy	Stat	No	No	No	No
BSP	Mea	Stat	Yes	Yes	No	No
LogP	Mea	Stat	Yes	Yes	No	No
Cole	Mea	Stat	No	No	No	No
Darlington	Mea	Stat	Yes	Yes	Yes	No
P3L	Mea	Prof	Yes	Yes	No	Yes
Skel-BSP	Mea	Prof	Yes	Yes	Yes	Yes
PUFF	Mea	Prof	Yes	Yes	No	No
SkelML	Mea	Prof	Yes	Yes	No	No
Parsec	Mea	Prof	Yes	Yes	No	No
Deldaie	Asy	Stat	Yes	Yes	No	No
Gorlatch	Asy	Stat	Yes	No	Yes	Yes
Skil	Asy	Stat	Yes	No	No	No
NESL	Mea	Stat	No	Yes	No	Yes
Loidl	Mea	Stat	No	No	No	No
HOPP	Mea	Stat	Yes	Yes	No	Yes
VEC	Mea	Stat*	No	No	No	Yes
Skillicorn	Asy	Stat	Yes	No	Yes	No
VEC-BSP	Mea	Stat*	Yes	Yes	Yes	No

Meas. Asym. column specifies if each cost model is measurement analysis (Mea) or asymptotic analysis (Asym).

Stat. Prof. column specifies if each cost model is static approach (Sta) or profiling-based approach (Prof). Sta* indicates its static analysis is automatic.

Comm. Sen. column specifies if each cost model is communication cost sensitive (Yes) or not (No).

Accur. Test column specifies if the accuracy of cost model was tested (Yes) or not (No).

Prog. Trans. column specifies if each cost model was applied to program translation process (Yes) or not (No).

Nest. Skel. column specifies if each model allows nested skeleton (Yes) or not (No).

Table 2.1: Summary of cost models

Chapter 3

VEC-BSP and its BSP Cost Analysis System

This chapter describes the detail of our language and its cost analysis system. In section 3.1 we overview our approach's structure and terminology. Section 3.2 presents the detail of the source language. Section 3.3 introduces the target language MSIZE. Section 3.4 gives the BSP implementation strategy for VEC-BSP. Section 3.5 describes our cost analysis technique and section 3.6 explains the details of translation functions.

3.1 Overview and Terminology

The proposed parallel model for VEC in [55] used the tightly synchronised, uniform access cost, shared memory PRAM model as its target architecture. The PRAM is an abstract model which takes no account of the communication and contention costs incurred on realistic parallel machines (whether explicitly programmed or in support of a shared memory abstraction). This chapter addresses this issue with BSP replacing the PRAM. There are several reasons for the choice of BSP. One is that it is able to model message passing, which is the dominant parallel programming style. Another is that it has a simple cost model which is suitable for predicting communication cost on wide range of actual machines. BSP and LogP models have been proposed in similar context and have attracted more attention than other alternative models. [8] compared these two models and summarised that BSP seems somewhat preferable due to greater simplicity and portability, and slightly greater power. In our context, the superstep structure of BSP, in which the communication phase and communication phase are separated by global synchronisation is particularly preferable because we need to only add a mechanism to the existing shape-based cost analysis so that it can cost the communication and synchronisation phases.

Changing the model requires a number of amendments to the assumed implementation mechanism (compiling VEC programs to BSP) and the analytic framework. New operators (and their implementation skeletons) are added to VEC, in order to broaden applicability and facilitate coding of our examples. We call the resulting language VEC-BSP to distinguish it from its predecessors.

As in the original analytic framework for VEC, our shape-based cost analysis aims to translate terms of source program to terms of another program which performs the analysis. Translated terms take the form of shape-cost pairs in the original work, but those in our analysis take the form of tuples whose components capture additional information from the BSP implementation. This information can be used at compile time for various purposes, in particular, costing communication (Hayashi and Cole [44]) and optimising communication (Hayashi and Cole [45]). The original analysis has a general framework based on the concept of a cost algebra whose operations may be customised to handle different cost regimes. Although the initial attempt for BSP costing in [44] was to customise the components of the cost algebra to compute BSP cost while keeping the original structure, it seems that some useful information to compute BSP cost is difficult to express within the operations of the cost algebra. Our solution is that such information is added to the shape-cost pair by extending it into the form of a tuple (called a cost tuple) rather than in the cost algebra. We now outline the structure of our analysis process mentioning differences with the original work for VEC.

Source Language

VEC-BSP is based on VEC, a shapely functional language which operates upon nested vectors of data. Shapeliness means that the form and size of data structures can be deduced statically. Shape constraints (which are analogous to type constraints) are used to ensure that all elements of a vector have the same shape (so that information about large structures can be captured and manipulated concisely.) This restriction is partially relaxed in an expanded version of VEC-BSP described in chapter 7. As VEC, VEC-BSP terms use standard functional terminology and have the expected semantics. The pair data type is extended as tuple data types and a small number of built-in second order functions are added to VEC-BSP in order to broaden applicability and facilitate coding our examples. The cost model for VEC needs assistance with general recursion; the programmer must indicate the anticipated recursion depth. We excludes recursion in this version since our goal is full automation. Our terms and types are discussed in more detail in 3.2.

Implementation Model

While the parallel implementation model proposed for VEC is the PRAM, that of VEC-BSP is the BSP model. Our basic BSP implementation structure is the nested structure of the implementation of the application term $t\,t'$ that consists of the four ordered parts, $E_{t'}$: evaluation of the argument, E_t: evaluation of the function, C: communication part and A: application part. Nesting arises because $E_{t'}$ and E_t can themselves be application terms. The communication part C is the communication of data rearrangement for the next application part. When function t is a built-in second order function which has a parallel implementation, (that is skeleton,) the application part follows its predefined BSP implementation template. More details of implementation strategies are discussed in 3.4.

Target Language

A source program is translated to another program in the target language MSIZE which, when run, will compute some implementation information such as shape and run-time cost. MSIZE is essentially a variant of VEC-BSP in which types, terms and operators which represent and manipulate real data have been removed, and with the addition of new features which manipulate implementation information not present in VEC-BSP. These are discussed in section 3.3.

Type Framework

The types of VEC-BSP terms are basically those of VEC, that is expected for an equivalent conventional functional program (primitive datum types, pairing and function types), with the addition of a type constructor vec for vectors (instead of lists), tupling (extention of pairing) types and types sz and un. sz is used to denote vector lengths, indices and other shape oriented quantities, while un denotes a unit type.

Types available in MSIZE are similar to those of VEC-BSP, with the exceptions that there are no primitive datum types (integer, boolean and so on) or structured types built from these.

While the modeled evaluation costs in the PRAM analysis are functions from the number of processors to time, those in BSP analysis are functions from the standard BSP performance parameters to time. Thus, for a given program and data set, our analysis returns a function which can itself be evaluated with the characteristics of different real machines. We use T to denote the type of such time functions.

Translation Function

The core of our method is a translation function *cost* which accepts VEC-BSP terms and returns MSIZE terms. An MSIZE term takes the following form called a *cost tuple*,

$$\langle \text{shape, data size, data pattern, cost} \rangle.$$

The shape of a function term is a function from the shape of an argument to an *application tuple*,

$$\text{shape of argument} \rightarrow \langle \text{shape of result, application pattern, application cost} \rangle.$$

Each component captures some kind of evaluation information. We added three new components which are not addressed in the original work for VEC, that is, data size, application pattern and data patten. Information of data size during the computation is required to determine the cost of communication phases. Although we can get data size of a non-function term from its shape, we add the data size component to the shape-cost pair to define the data size for a function term as well. The motivation for the addition of the application pattern and the data pattern is that when parallel functions (that is skeletons) are successively applied to some argument, if we can statically know the information of implementation pattern of a function application and that of the previous function application which generates the argument, it is possible to optimise their interface communication statically. The former information is added in the shape of a function term as the application pattern and the latter is added to shape-cost pair as a new component, the data pattern. A brief description for each component is:

- data size - a measure of the quantity of data which would have to be communicated to describe the term (in order to compute communication cost from the transmitted data);
- data pattern - an indication of the data distribution strategy required by the term's implementation. (in order that communication of the data between evaluation phases can be optimised);
- cost - an evaluation cost function for the term, mapping from performance parameters to time (so that evaluation time for the term can be computed, given the performance characteristics of the specific target);
- application pattern - an indication of application structure, that is whether sequential or parallel (in order to compute communication cost between the component evaluation process and the application process, and to optimise the communication of data between evaluation phases).

At the heart of the translation lies the mechanism for costing application terms of the form $t\ t'$, given the costings of the function t and argument t'. The intricate details are captured in our MSIZE function bspapp presented in section 3.5.4. While the corresponding capp function in the original work is customised by the operators of the cost algebra, our bspapp function is defined specially for BSP cost modeling. Essentially, this combine the costs of computing t and t' with the cost of applying t to t', also deducing information on the shape, data content and distribution information of the result. The type information involved in the translation is presented in 3.5 and the translation itself is discussed in detail in section 3.6.

3.2 VEC-BSP: A Shapely Skeletal Language

The source language of VEC-BSP is based on that of VEC defined in [55]. The differences between them are: VEC-BSP has more parallel skeletons; the pair data type in VEC is extended as the tuple data type in VEC-BSP; recursion in VEC is excluded from VEC-BSP.

We summarise its features here. The types are

$$\begin{aligned}
D &::= \mathsf{nat} \mid \mathsf{bool} \mid \ldots \\
\tau &::= D \mid \mathsf{sz} \mid \mathsf{un} \mid \tau \times \cdots \times \tau \mid \mathsf{vec}\,\tau \\
\theta &::= \tau \mid \theta \times \cdots \times \theta \mid \theta \to \theta
\end{aligned}$$

where D can include other simple *datum* types, and the type hierarchy precludes vectors of functions. The terminology of vectors (rather than lists) is used to emphasise the fact that the lengths of such objects will be statically determinable as items of type sz. Although sz is isomorphic to the natural numbers we will initially use the notation $\tilde{n}$ to distinguish shape sizes from ordinary numbers. We introduce two tuple data types which have three and four component respectively as an extention of the pair type in VEC so that we can express our example problem in chapter 6 in which a tuple data structure is required to compute an *almost homomorphism* [26]. The tuple data structure is also used in the MSIZE language to express cost tuples. Terms in VEC-BSP are given by

$$t ::= d \mid c \mid x \mid \lambda x.t \mid t\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{ifs}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$$

where d ranges over simple constants (integers, arithmetic operations and so on) and c ranges over the combinators with non-trivial shapes (those whose behaviour impacts upon the shape of terms) including our skeletons and a selection of conventional sequential functional operators (length, fst, snd and so on). The general typing rules are given in figure 3.1

$$\textbf{id}\quad \frac{\Gamma(x) = \theta}{\Gamma \vdash x : \theta} \qquad\qquad \textbf{if}\quad \frac{\Gamma \vdash t : \mathsf{bool}\quad \Gamma \vdash t' : \delta\quad \Gamma \vdash t'' : \delta}{\Gamma \vdash \mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'' : \delta}$$

$$\textbf{abs}\quad \frac{\Gamma, x : \theta \vdash t : \theta'}{\Gamma \vdash \lambda x^{\theta}.\, t : \theta \to \theta'} \qquad\qquad \textbf{ifs}\quad \frac{\Gamma \vdash t : \mathsf{sz}\quad \Gamma \vdash t' : \theta\quad \Gamma \vdash t'' : \theta}{\Gamma \vdash \mathsf{ifs}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t''\ : \theta}$$

$$\textbf{app}\quad \frac{\Gamma \vdash t : \theta \to: \theta'\quad \Gamma \vdash t' : \theta}{\Gamma \vdash t\, t' : \theta'}$$

Figure 3.1: VEC-BSP type inference rules

A full list of primitive functions (except for arithmetic operations) and their type is given in figure 3.2. Note that these functions were chosen as a basic set of standard functions. Some functions are not used in the examples in the thesis. Other functions can be added as long as they are shapely.

VEC-BSP has five skeletal combinators. Their informal definitions are:

- map - applies some function f to each element of an argument vector.

$$\mathsf{map}\ f\ [x_1,\ x_2, \cdots,\ x_n] = [f\, x_1,\ f\, x_2, \cdots, f\, x_n]$$

- fold - combines the elements of a vector using an associative binary operator $\oplus$.

$$\mathsf{fold}\ \oplus\ [x_1,\ x_2, \cdots,\ x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

- pair_map - applies a function to elementwise pairs drawn from a pair of vectors of the same length.

$$\mathsf{pair\_map}\, f\, ([x_1, x_2, \cdots, x_n],\ [y_1, y_2, \cdots, y_n]) = [f\, x_1\, y_1,\ f\, x_2\, y_2, \cdots,\ f\, x_n\, y_n]$$

unit	: un
pair	: $\theta \to \theta' \to \theta \times \theta'$
fst	: $\theta \times \theta' \to \theta$
snd	: $\theta \times \theta' \to \theta'$
tuple3	: $\theta \to \theta' \to \theta'' \to \theta \times \theta' \times \theta''$
π_{13}	: $\theta \times \theta' \times \theta'' \to \theta$
π_{23}	: $\theta \times \theta' \times \theta'' \to \theta'$
π_{33}	: $\theta \times \theta' \times \theta'' \to \theta''$
tuple4	: $\theta \to \theta' \to \theta'' \to \theta''' \to \theta \times \theta' \times \theta'' \times \theta'''$
π_{14}	: $\theta \times \theta' \times \theta'' \times \theta''' \to \theta$
π_{24}	: $\theta \times \theta' \times \theta'' \times \theta''' \to \theta'$
π_{34}	: $\theta \times \theta' \times \theta'' \times \theta''' \to \theta''$
π_{44}	: $\theta \times \theta' \times \theta'' \times \theta''' \to \theta'''$
length	: $\mathsf{vec}\ \tau \to \mathsf{sz}$
hd	: $\mathsf{vec}\ \tau \to \tau$
tl	: $\mathsf{vec}\ \tau \to \mathsf{vec}\ \tau$
entry	: $\mathsf{vec}\ \tau \to \mathsf{sz} \to \tau$
map	: $(\tau \to \tau') \to \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau'$
fold	: $(\tau \to \tau \to \tau) \to \mathsf{vec}\ \tau \to \tau$
pair_map	: $(\tau \to \tau' \to \tau'') \to \mathsf{vec}\ \tau \times \mathsf{vec}\ \tau' \to \mathsf{vec}\ \tau''$
scan	: $(\tau \to \tau \to \tau) \to \mathsf{vec}\ \tau \to \mathsf{vec}\ \tau$
c_prod	: $(\tau \to \tau' \to \tau'') \to \mathsf{vec}\ \tau \times \mathsf{vec}\ \tau' \to \mathsf{vec}\ (\mathsf{vec}\ \tau'')$
iter	: $(\tau \to \tau) \to \tau \to \mathsf{sz} \to \tau$

Figure 3.2: VEC-BSP primitive functions

- scan - applies to a vector and the partial result of fold up to the ith element is returned as the ith element of the resulting vector.

$$\mathsf{scan} \oplus [x_1, x_2, \cdots, x_n] = [x_1, x_1 \oplus x_2, \cdots, x_1 \oplus x_2 \oplus \cdots \oplus x_n]$$

- c_prod - applies a function to the all elements of the cross product of two vectors.

$$\begin{aligned}\mathsf{c\_prod}\ f\ [x_1, x_2, \cdots, x_m][y_1, y_2, \cdots, y_n] \quad = \quad & [[f\,x_1\,y_1,\ f\,x_2\,y_1, \cdots, f\,x_m\,y_1], \\ & [f\,x_1\,y_2,\ f\,x_2\,y_2, \cdots,\ f\,x_m\,y_2], \\ & \qquad\vdots \\ & [f\,x_1\,y_n,\ f\,x_2\,y_n, \cdots,\ f\,x_m\,y_n]]\end{aligned}$$

As in the original work on VEC there are two forms of conditional: a data conditional if, whose condition is given by a datum; and a shape conditional ifs, whose condition is a size (with ˜0 interpreted as false, other sizes as true). The iter combinator allows bounded iteration, controlling repeated application of a function to data. The number of repeats must be statically determined. VEC has recursion, but the programmer needs to anticipate its depth. We exclude recursion, since our goal is full automation.

3.3 Msize: A Target Language

VEC-BSP is translated to a target language called MSIZE, which corresponds to the SIZE language for VEC, generating cost tuples which consist of information of shape, data size, data pattern and cost. The types of MSIZE are

$$\theta ::= \mathsf{un} \mid \mathsf{sz} \mid \theta \times \cdots \times \theta \mid \theta \overline{\times} \theta \mid \theta \to \theta$$

MSIZE has two kinds of pair type. One is $\theta \times \theta$, a special case of the tuple type that has two components and the other is $\theta \overline{\times} \theta$, a pair type for a shape expression that comprises the length and the element shape. The tuple type that has four components is used as the type of a cost tuple. The terms of MSIZE are given by

$$t ::= c \mid x \mid \lambda x.\, t \mid t\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$$

where c denotes VEC-BSP functions. Its type inference rules are the applicable VEC-BSP rules.

3.4 Implementation Strategy

Now we give an implementation strategy for VEC-BSP on its target implementation model BSP. Because BSP has no shared memory the main issue is to specify placement and movement of data in the style of message passing programming, while keeping things simple enough to predict cost automatically. Furthermore, our implementation model should follow the superstep structure of BSP. We use execution diagrams to explain the implementation model and data movements on it. In our execution diagrams, time proceeds from left to right with activities in a processor proceeding horizontally. Data flows from left to right being manipulated and transmitted following instructions from terms of the source program. We use one of the processors (at the top in our diagram) as a master processor in which the necessary data is stored at the beginning of computation and the result is eventually stored at the end of the computation.

A complete computation of a program $t\,t'$ has a nested structure consisting of four ordered parts, which are illustrated in the diagram of figure 3.3 in which shaded processors indicate existence of data in those processors.

- $E_{t'}$: an evaluation of the argument t'
- E_t: an evaluation of the function t
- C: a communication, in which the data of the results of $E_{t'}$ and E_t are redistributed to processors for the next process if necessary, followed by a barrier synchronisation
- A: an application, in which the result of E_t is applied to the result of $E_{t'}$.

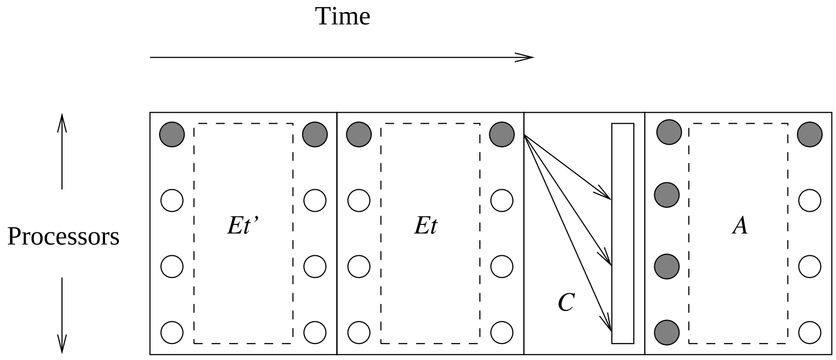


Figure 3.3: Parallel application

Nesting arises because $E_{t'}$ and E_t can themselves be application terms. In each part, the data is stored in the master processor at its end with the exception that automatic optimisation is used to remove some overhead incurred, as explained later. The application phase A may be either sequential or parallel. A sequential application illustrated in figure 3.4 is executed only in the master processor when t is a sequential function (its information can be known from the application tuple). There is no communication in

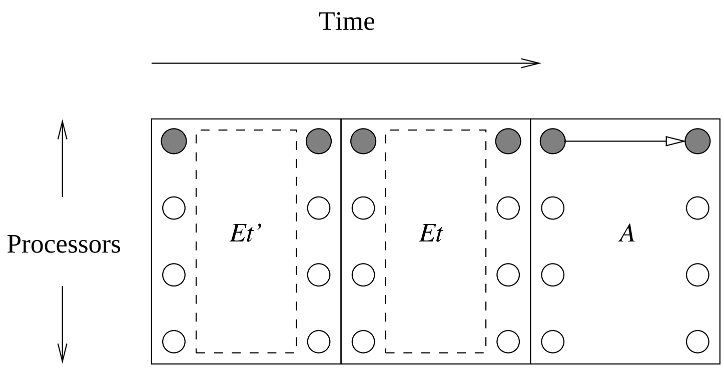


Figure 3.4: Sequential application

C because the necessary data already resides in the master processor. The parallel application illustrated in figure 3.3 is executed among the processors when the function t is a skeleton combinator (this information also can be known from the application tuple) whose parallel implementation template is predefined. We place the following restrictions on the implementation template.

- The template must follow the BSP model, that is computation and communication are separated by machine wide synchronisation.
- The data of the argument is distributed evenly at the beginning of the template.
- All processors perform the same operation.
- The result is eventually stored in the master processor.

Combinators map and fold are typical examples of second-order functions which have parallel templates. The implementation template of map applies the function sequentially on the vector segments in each processor then gathers the results to the master.

The fold implementation template folds sub-vectors sequentially on each processor. Results are transferred to the master processor which folds them together sequentially to compute the overall result. Figure 3.5 illustrates these applications. Solid lines indicate computation and dotted lines indicate gather. The narrow vertical box denotes machine wide synchronisation. The details of the full set of our current implementation

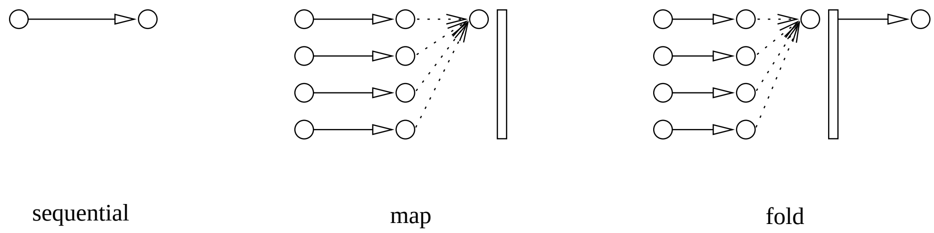


Figure 3.5: Application examples

skeletons are presented in chapter 4.

Using a skeleton requires communication for data rearrangement in C in which the data describing the result of $E_{t'}$ is scattered to all processors evenly. If there is any data component of the result of E_t, it is broadcast to all processors. For example, if t is map$(+t'')$, where t'' generates some value, then that value must be broadcast to all processors. Therefore, costing communication in C means costing the broadcast and scatter communication. The formula for this will be given in section in 3.5.3. Figure 3.6 illustrates the $scatter$ and $broadcast-scatter$ communication. In the $scatter$ diagram the master processor (at the top) scatters data between itself and three other processors. Dotted lines indicate scatter and dashed lines indicate broadcast.

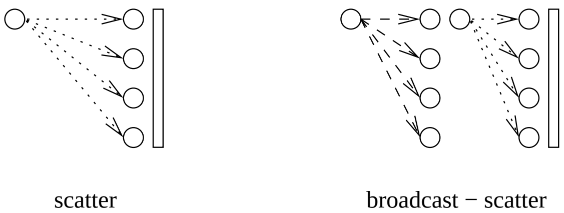


Figure 3.6: Communication patterns

Additional comments are required for the implementation of the application of a lambda

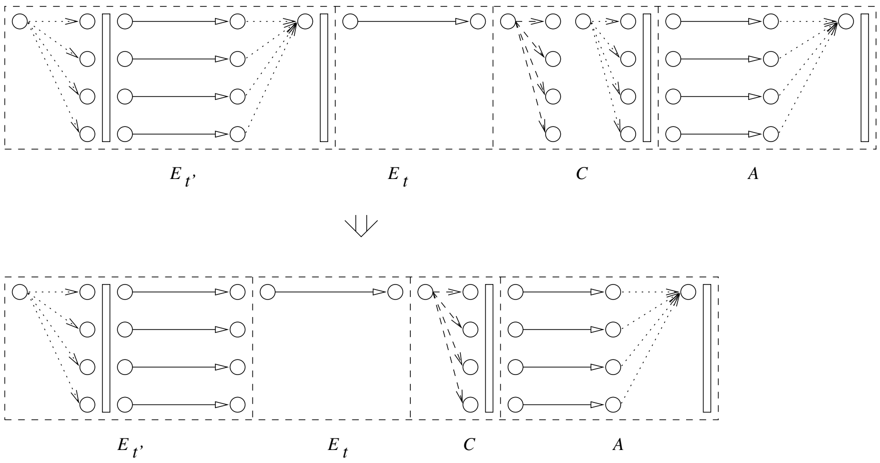


Figure 3.7: Removal of unnecessary communication

term. Implementation of the application of $(\lambda x.t(x))\ a$ is: a is evaluated first and x is substituted by the result of evaluation of a, and $t\,(a)$ is evaluated in the application part A following the strategy described above. We assume that it takes no time to evaluate the term $(\lambda x.t(x))$, that means we count the costs to evaluate a and to evaluate $t(a)$, which is performed in the application part and ignore the other costs involved.

Efficiency Problem

We required the parallel implementation templates to store data in the master processor at the end of A. Consequently, the data of the results of E_t' and E_t are also always stored in the master processor, since these are either themselves nested parallel applications abiding by the same rule, or are already sequential. This rule simplifies implementation and costing communication by providing a common interface for communication patterns across the nested term. However, it also causes an efficiency problem. For example, if a parallel application process finishes by gathering the local result in each processor to the master, only for these to be subsequently scattered as the inputs to an enclosing parallel function, then the gathering and scattering are superfluous. The upper half of figure 3.7 illustrates the structure of such a computation, for a term of the form $\mathsf{map}\ f\ (\mathsf{map}\ g\ v)$. The first phase implements the map of g (with g assumed to be primitive), the second phase computes f (sequentially in this example), the third phase broadcasts data describing f and scatters the result of the first phase and the final phase computes the outer map. The gathering and subsequent scattering of the result of the inner map is clearly redundant. When the data size of the result of `map` $g\ v$ is s,

$$2 \cdot s \cdot \frac{p-1}{p} \cdot g + l$$

where p, g, l are the BSP parameters, can be saved.

There are several possible solutions. One solution would be to define several versions of a skeleton, with implementations differing only in data distribution at the end of the application process and expect the programmer to chose one of them to optimise each C. It would be performed by hand and makes the programming more difficult. Another solution would be to predefine *combining skeletons* which combine skeletons so that interface communication of component skeletons can be easily optimised following To's work [81]. Instead we chose an automated route, demonstrating that our static analysis can be extended to analyse the interface communication pattern by adding an argument *data pattern* to cost tuples. How our tool detects and resolves such inefficient cases and excludes these unnecessary costs from the predicted BSP cost is described in the next section.

3.5 Cost Analysis

This section presents an analytic framework to compute both the computation costs and inter-processor communication costs of our BSP implementation. We also show how the analysis captures information on communication patterns by which some communication costs can be optimised.

3.5.1 Cost Tuples and Application Tuples

The translation is defined by a function *cost* which is now from VEC-BSP to MSIZE. The translated program, which when run, will compute the shape of the result with other information including the run-time cost of the corresponding compiled BSP program. The terms of the VEC-BSP programs are translated to cost tuples in MSIZE which take the form

$$\langle \text{shape, data size, data pattern, cost} \rangle$$

For a VEC-BSP term $t : \theta$, the corresponding MSIZE term has type

$$cost\,(t) : tycost_C\,(\theta) \times \mathsf{sz} \times \mathsf{sz} \times T$$

where $tycost_C(\theta)$ reflects the shape of t. The first sz reflects the size of the data which will be transmitted to the following process when t is involved in an application. The second sz reflects the *data pattern*, which we introduce to analyse interface communication patterns between individual skeletons. T reflects the BSP cost of term t. We now look at more details of each component.

Application Pattern

To simplify our explanation, we begin with the definition of the application pattern. To optimise communication between a component evaluation part and an application part, the data distribution of the result of $E_{t'}$ and the data distribution which is required for A should be well matched. In order to achieve this, we distinguish the parallel application pattern in which the result is obtained by just gathering the local results on the worker processors at the end of application process like map (referred to as *map pattern*) from the other parallel patterns like fold (referred to as *fold pattern*). The *sequential* pattern is referred to as *sequential pattern*. Although the language actually uses sz for application patterns, with˜1 for map pattern,˜2 for fold pattern, and˜0 for the other cases (sequential pattern or t itself is primitive), in the following explanation in the thesis (except for the Haskell implementation in chapter 5), we use MAP, FOLD, SEQ instead of ˜1,˜2,˜0 in order to improve readability.

Data Pattern

Data patterns indicate which application pattern was used to generate t, MAP for map pattern, FOLD for fold pattern, and SEQ for the other cases (sequential pattern or t

itself is primitive).

Shape

The type translation of shape for a non-function term $tycost_C$ is defined as follows:

$$\begin{aligned}
tycost_C\,(D) &= \mathsf{sz} \\
tycost_C\,(\mathsf{un}) &= \mathsf{sz} \\
tycost_C\,(\theta_1 \times \cdots \times \theta_n) &= tycost_C\,(\theta_1) \times \cdots \times tycost_C\,(\theta_n) \\
tycost_C\,(\mathsf{vec}\,\theta) &= \mathsf{sz}\ \bar{\times} tycost_C\,(\theta) \\
tycost_C(\mathsf{sz}) &= \mathsf{sz}
\end{aligned}$$

We assign ˜1 to the shape of D and un. The shape of a tuple is a tuple of the shape of each component. The shape of a vector is a pair comprising its length and the common shape of its elements. In MSIZE, the shapes of tuples and vectors are denoted by $\langle\ ,\cdots,\ \rangle$ and $(\ ,\ )$ respectively, corresponding to the types $tycost_C\,(\theta_1) \times \cdots \times tycost_C\,(\theta_n)$ and $\mathsf{sz}\ \bar{\times}\ tycost_C\,(\theta)$. The $\bar{\times}$ is used to indicate that its pair type is different from the pair type which is the special case of tuple type that has two components.

The shape of a function is a function from the shape of an argument to an *application tuple*, which is composed by attaching an application pattern and an application cost to a resulting shape, taking the form of

$$\text{shape of argument} \rightarrow \langle\text{shape of result, application pattern, application cost}\rangle,$$

with the corresponding type

$$tycost_C\,(\theta \rightarrow \theta') = tycost_C\,(\theta) \rightarrow (tycost_C\,(\theta') \times \mathsf{sz} \times T)$$

The shape types $tycost_C\,(\theta)$ and $tycost_C\,(\theta')$ reflect the change of shape. T is the type of a function from the BSP parameters to cost, reflecting the application cost of the function, sz reflects the application pattern. A term by term definition of *cost* is presented in the section 3.6 and chapter 4.

Data Size

Data sizes of non-function terms are computed from the shape by using an MSIZE operator size defined by

$$\begin{aligned}
&\mathsf{size}\,\langle x_1,\ldots,x_n\rangle = \mathsf{size}\,x_1 + \cdots + \mathsf{size}\,x_n \\
&\mathsf{size}\,(x,y) = \mathsf{size}\,x \cdot \mathsf{size}\,y \\
&\mathsf{size}\,\tilde{}\,n = \tilde{}\,n
\end{aligned}$$

Data sizes of primitive functions themselves are defined as 0. For function terms generated by partial applications, we define those data sizes as the sum of the data sizes of the function term and the argument term of the partial application.

Cost

Cost is a function from standard BSP performance parameters to time for evaluation of the term. We use T to denote the type of such time functions. Thus, for a given program and data set, our analysis returns a function which can itself be evaluated with the characteristics of different real machines.

3.5.2 Cost Translations Framework

According to the explanation of cost tuples in the previous section, the *cost* function for datum constant and primitive functions is given in figure 3.8. To further simplify presentation, we use the following notation to describe cost tuples in MSIZE whose elements are 0 (or SEQ for data patten) except for its first element:

$$\lambda' x.\, t = \langle \lambda x.\, t, 0, \mathsf{SEQ}, 0\rangle$$

If x is the shape of a vector, to take the length of the vector and the shape of the elements from x, we use the notations $\mathsf{t\_len}\,x$ and $\mathsf{t\_eshp}\,x$ for $\mathsf{fst}\,x$ and $\mathsf{snd}\,x$. From the

$$
\begin{array}{lll}
cost\,(d) & = & \langle 1, 1, \mathsf{SEQ}, 0\rangle \text{ where } d \text{ is a datum constant} \\
cost\,(d) & = & \lambda' x.\langle \lambda y.\langle 1, \mathsf{SEQ}, \mathsf{binOpConst}\rangle, \mathsf{SEQ}, 0\rangle \text{ where } d \text{ is a binary datum operation} \\
cost\,(\mathsf{length}) & = & \lambda' x.\langle \mathsf{t\_len}\, x, \mathsf{SEQ}, \mathsf{lengthConst}\rangle \\
cost\,(\mathsf{hd}) & = & \lambda' x.\langle \mathsf{t\_eshp}\, x, \mathsf{SEQ}, \mathsf{hdConst}\rangle \\
cost\,(\mathsf{tl}) & = & \lambda' x.\langle (\mathsf{fst}\, x - 1, \mathsf{t\_eshp}\, x), \mathsf{SEQ}, \mathsf{tlConst}\rangle \\
cost\,(\mathsf{entry}) & = & \lambda' x.\langle \lambda y.\langle \mathsf{t\_eshp}\, x, \mathsf{SEQ}, \mathsf{entryConst}\rangle, \mathsf{SEQ}, 0\rangle \\
cost\,(\mathsf{pair}) & = & \lambda' x.\langle \lambda y.\langle \langle x, y\rangle, \mathsf{SEQ}, \mathsf{pairConst}\rangle, \mathsf{SEQ}, 0\rangle \\
cost\,(\mathsf{fst}) & = & \lambda' x.\langle \mathsf{fst}\, x, \mathsf{SEQ}, \mathsf{fstConst}\rangle \\
cost\,(\mathsf{snd}) & = & \lambda' x.\langle \mathsf{snd}\, x, \mathsf{SEQ}, \mathsf{sndConst}\rangle \\
cost\,(\mathsf{tuple3}) & = & \lambda' x.\langle \lambda y.\langle \lambda z.\langle \langle x, y, z\rangle, \mathsf{SEQ}, \mathsf{tuple3Const}\rangle, \mathsf{SEQ}, 0\rangle, \mathsf{SEQ}, 0\rangle \\
cost\,(\pi_{13}) & = & \lambda' x.\langle \pi_{13}\, x, \mathsf{SEQ}, \mathsf{projecConst}\rangle \\
cost\,(\pi_{23}) & = & \lambda' x.\langle \pi_{23}\, x, \mathsf{SEQ}, \mathsf{projecConst}\rangle \\
cost\,(\pi_{33}) & = & \lambda' x.\langle \pi_{33}\, x, \mathsf{SEQ}, \mathsf{projecConst}\rangle \\
cost\,(\mathsf{iter}) & = & \lambda' f.\langle \lambda x.\langle \lambda y.\, \mathsf{iter}\,(\mathsf{preiter} f)\langle x, \mathsf{SEQ}, \mathsf{iterConst}\rangle\, y, \mathsf{SEQ}, 0\rangle, \mathsf{SEQ}, 0\rangle \\
cost\,(\mathsf{map}) & = & \lambda' f.\langle \lambda x.\langle (\mathsf{t\_len}\,(x), \mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\, x))), \mathsf{MAP}, \mathsf{apcost\_map}\, f\, x\rangle, \mathsf{SEQ}, 0\rangle \\
cost\,(\mathsf{fold}) & = & \lambda' \oplus .\langle \lambda x.\langle \mathsf{t\_shp}\,(\mathsf{iter}\,(\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_shp}\,(iter\_rlt)))))(\mathsf{t\_shp}\,(iter\_rlt))(p-1)), \\
 & & \mathsf{FOLD}, \mathsf{ap\_cost\_fold} \oplus x\rangle, \mathsf{SEQ}, 0\rangle \\
cost\,(\mathsf{scan}) & = & \lambda' f.\langle \lambda x.\langle x, \mathsf{MAP}, \mathsf{ap\_cost\_scan}\, f\, x\rangle, \mathsf{SEQ}, 0\rangle \\
cost\,(\mathsf{pair\_map}) & = & \lambda' f.\langle \lambda x.\langle (\mathsf{t\_len}\,(\mathsf{fst}\, x), \mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\, x))(\mathsf{t\_eshp}\,(\mathsf{snd}\, x)))), \mathsf{MAP} \\
 & & \mathsf{ap\_cost\_pair\_map}\, f\, x\rangle, \mathsf{SEQ}, 0\rangle \\
cost\,(\mathsf{c\_prod}) & = & \lambda' f.\langle \lambda x.\langle \lambda y.\langle (\mathsf{t\_len}\, y, (\mathsf{t\_len}\, x, \mathsf{t\_shp}\,(\mathsf{t\_shp}\,(f\, x)\, y))), \mathsf{MAP}, \mathsf{ap\_cost\_c\_prod}\, f\, x\, y\rangle, \\
 & & \mathsf{SEQ}, 0\rangle, \mathsf{SEQ}, 0\rangle
\end{array}
$$

Figure 3.8: Cost translations (1)

definition of the shape of a function, if f is the shape of a function and x is the shape of an argument, $f\,x$ takes the form

$$\langle \text{resulting shape of application, application pattern, application cost}\rangle$$

We use shorthands $\mathsf{t\_shp}\,(f\,x)$, $\mathsf{t\_pattern}\,(f\,x)$, $\mathsf{t\_apcost}\,(f\,x)$, $\mathsf{t\_size}\,(f\,x)$ (this is equivalent to $\mathsf{size}\,(\mathsf{t\_shp}\,(f\,x))$) to represent the operation to take the shape of the result, the application pattern, the application cost and the size of the result from $f\,x$.

Note that these *cost* terms are *predefined* according to the meanings of cost tuple components under the assumption of the cost modeling and implementation strategy. The

$$
\begin{aligned}
&cost\,(x) && = \langle x, \mathsf{size}\,x, \mathsf{SEQ}, 0\rangle \\
&cost\,(\lambda x.t) && = \lambda' x.\ \langle \pi_1\,(cost\,(t)), \mathsf{SEQ}, \pi_4\,(cost\,(t))\rangle \\
&cost(t\,t') && = \mathsf{bspapp}\ cost(t)\ cost(t') \\
&cost\,(\ \mathsf{if}\ \ t\ \ \mathsf{then}\ \ t'\ \ \mathsf{else}\ \ t'') && \\
& && = \mathsf{add}_1\,(\mathsf{tuplemax}\ cost\,(t')\,cost\,(t''))\,(\pi_4\,cost\,(t)) \\
&cost\,(\ \mathsf{ifs}\ \ t\ \ \mathsf{then}\ \ t'\ \ \mathsf{else}\ \ t'') && \\
& && = \mathsf{add}_1\,(\ \mathsf{ifs}\ \ \pi_1\,cost\,(t)\ \mathsf{then}\ \ cost\,(t')\ \ \mathsf{else}\ \ cost\,(t''))\,(\pi_4\,cost\,(t))
\end{aligned}
$$

Figure 3.9: Cost translations (2)

shapes of the primitive functions are defined so that they capture the change of shape, application pattern, and application cost. The change of shape is captured by the result shape expressed as a function of input shape. The application pattern is defined according to the predefined implementation template. The application costs that do not depend on the argument shape are expressed like binOpConst for a binary operation and lengthConst for length. We assign times to the various costs later, based on some benchmarks of the target machine. The application costs that do depend on the argument shape are expressed as a function of input shape. Application cost function for our parallel function, that is skeleton, are too complex to present in-line and so we give them name here (apcost_map and so on), presenting full definition in Chapter 4. The data size, data patten and cost of a primitive function term itself are defined as 0, SEQ, and 0 respectively. The *cost* functions for other expressions are given in figure 3.9. How this information is predefined (including the definitions of functions in the list, which have not yet been defined) for each individual construct is explained later in more detail in section 3.6 and chapter 4.

The program in VEC-BSP is transformed to a program in MSIZE by *cost* using these definitions as translation rules. A simple (sequential) example of the translation is:

$$
\begin{aligned}
& && cost(+\ 3\ 5) \\
&= && \{\text{definition of } cost(t\,t')\} \\
& && \mathsf{bspapp}\ (cost(+\ 3))\ cost(5) \\
&= && \{\text{definition of } cost(t\,t')\}
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{bspapp}\ (\mathsf{bspapp}\ cost(+)\ cost(3))\ cost(5) \\
= \ & \{\text{definition of } cost(d)\} \\
& \mathsf{bspapp}\ (\mathsf{bspapp}\ \lambda' x.\langle \lambda y.\langle 1, \mathsf{SEQ}, \mathsf{binOpConst}\rangle, \mathsf{SEQ}, 0\rangle\ \langle 1, 1, \mathsf{SEQ}, 0\rangle)\ \langle 1, 1, \mathsf{SEQ}, 0\rangle
\end{aligned}
$$

When the translated MSIZE program is evaluated, the fourth component of the resulting tuple is the predicted cost of the source VEC-BSP program. The cost modeling, that is explained in the next section relies on the application mechanism, that is how function is applied to the argument in MSIZE using information from cost tuples.

3.5.3 Cost Modeling

The BSP cost modeling of our assumed implementation model is carried out by adding the mechanism to compute communication and synchronisation phase costs into the definition of capp for the PRAM cost modeling, resulting in the definition of bsppapp. bsppapp also includes the mechanism to optimise communication. We explain first how to cost the communication phase, next how to detect the inefficient case, and finally the definition of bsppapp.

Costing Communication

The communication which is involved in our computation model occurs in two situations, firstly in C when the parallel pattern is used, and secondly within A when a parallel template which includes a communication process is used. Here we discuss how to compute communication cost in the former situation, giving the reasons for the rules imposed in the implementation strategy. The communication cost in the latter situation is counted as the part of application cost, which is defined for each skeletal combinator in chapter 4.

In the BSP cost model, the cost of a communication phase is determined by the formula $h \cdot g$, where g is one of the BSP parameters and h is the largest message size h sent or received by any one processor during the phase. Since g is determined as a parameter to capture the communication performance of the target architecture, which is obtained experimentally by running a benchmark program on the architecture, what we need is machinery to determine the value of h at every communication phase in our analytic framework.

In the general case, determining h in the communication part C requires the following information:

- the distribution pattern of components evaluation, that is how the data of the results of $E_{t'}$ and E_t are distributed among the processors.
- the size of the data of the results of $E_{t'}$ and E_t placed in each processor.
- the distribution pattern which is required by the application.
- the communication pattern to realise the data rearrangement from the data distribution at the end of component evaluation to the data distribution which is required by the application.

To get all this information in the framework of shape analysis would require a complex mechanism and consequently may impose expensive analysis costs. To simplify the issue, we used one processor as the master processor and imposed the rule that the data of the result of each part is eventually stored in the master processor. This made information on distribution pattern and size of the result of components evaluation quite simple. We also restricted the data distribution patterns at the beginning of application parts. If the function is sequential, we use the master processor for the application part so that there is no communication in C since the necessary data all resides in the master processor. For a parallel function, we placed a restriction on the parallel application templates so that the data of the argument are always distributed evenly among the processors and all processors perform the same operation. Thus, the communication pattern in C in the case of parallel application is determined uniquely that is, the data of the result of $E_{t'}$ is scattered to the processors evenly and that of E_t is broadcast to the processors. Whether a function is sequential or parallel can be known from information on the application pattern in the cost tuple. Consequently, the information components, message size in cost tuples and application pattern in application tuples are sufficient to determine h.

The communication cost in C is now determined by computing the number of words transmitted by processor 0, that is

$$s \cdot (p-1) \cdot g$$

for broadcasting s words of the results of E_t to the worker processors, and

$$s' \cdot \frac{p-1}{p} \cdot g$$

for scattering s' words of the results of $E_{t'}$ to the worker processors.

Optimising Communication

The strategy described above allows us to compute the communication cost in C but it caused an efficiency problem as explained in 3.4. It requires further refinement of our analytic framework to solve this problem.

In our translation framework, information on the application pattern is used for two purposes. One is that it tells whether the function is parallel or sequential, which determines the data distribution pattern required in A, and is used to compute the communication cost. The other is that it tells whether the application template finishes by gathering the local results or not. This information is not used for costing this application process itself but is kept in the cost tuple of the result as a component, the data pattern, giving information on how the data is generated, which is used when the result becomes an argument of another function. Thus, the cost tuple of any argument always has information on the data pattern. Note that the data pattern of a primitive datum is predefined as SEQ. To indicate information for the second purpose, one of the three notations SEQ, MAP, FOLD which indicate sequential, parallel with a gather at the end, and other parallel, respectively is used and it also gives information for the first purpose (SEQ is sequential, other values are parallel). The inefficient case is detected by checking the combination of the application pattern in the application tuple of the function and the data pattern in the cost tuple of the argument. If the application pattern indicates that the function is parallel (MAP or FOLD), and the data pattern indicates that the data of the argument was generated by gathering the local results (that is, MAP), the inefficient case is detected and the cost for the gather and the scatter is not counted in the resulting cost. Note that this decision would be available to the compiler to make the corresponding optimisation in a real implementation.

3.5.4 bspapp Operation

In the original work for VEC, the analysis process of a program $t_1 t_2$ is captured by the capp operator in VEC, which applies to the corresponding translated terms of t_1 and t_2, that is $\langle f, t_1' \rangle$ and $\langle x, t_2' \rangle$ respectively.

$$cost\,(t_1\,t_2) = \mathsf{capp}\; cost\,(t_1)\; cost\,(t_2)$$

where the definition of capp is

$$\mathsf{capp}\,\langle f, t_1' \rangle \langle x, t_2' \rangle = \langle \mathsf{fst}\,(f x), (\mathsf{snd}\,(f x)) + (t_1' \oplus t_2') \rangle$$

This implies that the cost of the application term is some kind of the combination of application cost $\mathsf{snd}\,(f\,x)$, the cost of the function term t_1' and the cost of the argument term t_2'. $+$, $\oplus$ and T can be changed reflecting the underlying cost model. The proposed cost model was the PRAM model. For the PRAM model, we have $T = \mathsf{sz} \rightarrow \mathsf{sz}$, representing time functions from the number of processors to the number of time steps. Sequential cost addition $+$ is pointwise addition on time functions, and addition $\oplus$ for parallel execution is:

$$(f \oplus g)\,p = \min\{(f+g)\,p,\; (f \oplus' g)\,p\}$$

where

$$(f \oplus' g)\,p = \min_{0<q<p} \{\max\{f\,q,\; g\,(p-q)\}\}$$

Our BSP cost analysis process described in 3.5.3 is captured by a MSIZE operation bspapp using information from cost tuples and application tuples described in 3.5.1 rather than operations of a cost algebra. For the BSP model, the cost is a function from BSP parameters to time for evaluation of the term. Reflecting our implementation model described in 3.4, the BSP cost of the application term is the sum of the cost of the argument term, the cost of the function term, cost of the communication phase, the cost of the synchronisation and the cost of application phase. The first two costs are already known as the fourth component of the argument and the function. The communication cost can be computed by the formula presented in the previous section using information of data size, that is third components of the argument and the function. The synchronisation cost is l. The application cost can be obtained from the application tuple in the first component of the function. In addition, the inefficient case of communication interface can be detected by checking the data pattern information that is the third component of the data argument and the application pattern information, that is the third component of the function. All these costing mechanisms are formulated as the definition of bspapp:

$$cost\,(t_1\,t_2) = \mathsf{bspapp}\; cost\,(t_1)\; cost\,(t_2)$$

The definition of bspapp is

$$\begin{aligned} & \mathsf{bspapp}\,\langle f, s, d, t\rangle \langle x, s', d', t'\rangle \\ = \; & \langle \mathsf{t\_shp}\,(f\,x), \mathsf{data\_sz}\;(\mathsf{t\_apcost}\,(f\,x)), \mathsf{t\_pattern}\,(f\,x), \\ & (t+t') + \lambda\langle p,g,l\rangle.((\mathsf{comm\_cost}\;(\mathsf{t\_pattern}\,(f\,x))\; d'\; s\; s') + l) + \mathsf{t\_apcost}\,(f\,x)\rangle \end{aligned}$$

where

$$\begin{aligned} \mathsf{data\_sz}\;\; ap\_c & = s + s', && \text{if } ap\_c = \tilde{}0 \\ & = \mathsf{t\_size}(f\,x), && \text{otherwise} \end{aligned}$$

$$\begin{aligned} & \mathsf{comm\_cost}\;\; ap\_pat\;\; dat\_pat\;\; f\_sz\;\; x\_sz \\ & \qquad = \tilde{}\,0, && \text{if } ap\_pat = \mathsf{SEQ} \\ & \qquad = (f\_sz \cdot (p-1) - x\_sz \cdot ((p-1)/p)) \cdot g - l, && \text{if } dat\_pat = \mathsf{MAP} \\ & \qquad = (f\_sz \cdot (p-1) + x\_sz \cdot ((p-1)/p)) \cdot g, && \text{otherwise} \end{aligned}$$

$\mathsf{t\_shp}\,(f\,x)$ represents the result shape. If the result of an application is a function, then there is no application cost and the message size of $t_1\,t_2$ is just the sum of s and s'. When the result is not a function, the message size of $t_1\,t_2$ is $\mathsf{t\_size}\,(f\,x)$. The cost of term $t_1\,t_2$ combines the four costs, that is the costs of the component evaluations $t+t'$, the communication cost, synchronisation cost l and the application cost $\mathsf{t\_apcost}\,(f\,x)$. The communication cost depends on the application pattern $\mathsf{t\_pattern}\,(f\,x)$ and the argument data pattern d'. If the application pattern is not SEQ and the data pattern is MAP, the communication costs for gathering the local results and the synchronisation at the end of the evaluation of the argument are removed and the communication cost for the next scattering of the data is not counted. The data pattern is equal to the application pattern. Note that the communication which occurs in A depends on the assumed implementation template of each skeleton and its cost is counted in $\mathsf{t\_apcost}\,(f\,x)$ which is defined in chapter 4.

Using the bspapp operation, MSIZE is evaluated reflecting our cost model and generating the predicted cost. Here is the evaluation of the MSIZE example program given in section 3.5.2, which was translated from $+\ 3\ 5$.

$$
\begin{aligned}
& \mathsf{bspapp}\ (\mathsf{bspapp}\ \lambda' x.\langle \lambda y.\langle 1, \mathsf{SEQ}, \mathsf{binOpConst}\rangle, \mathsf{SEQ}, 0\rangle\ \langle 1, 1, \mathsf{SEQ}, 0\rangle)\ \langle 1, 1, \mathsf{SEQ}, 0\rangle \\
= \ & \mathsf{bspapp}\ \langle \lambda y.\langle 1, \mathsf{SEQ}, \mathsf{binOpConst}\rangle, \mathsf{SEQ}, 0\rangle\ \langle 1, 1, \mathsf{SEQ}, 0\rangle \\
= \ & \langle 1, 1, \mathsf{SEQ}, \mathsf{binOpConst}\rangle
\end{aligned}
$$

A complete example of costing for a VEC-BSP program which includes a parallel function is given in chapter 5, after chapter 4 has explained the application cost of skeletons.

3.6 Details of Cost Translation Rules

We now present the cost translation rules from VEC-BSP to MSIZE for basic term expressions and functions except for those for our parallel combinators, which are given in chapter 4. We now omit the notation ˜ for size numerals to reduce clutter. Semantically, the terms of the language have the obvious strict functional operational interpretation with the exception of parallel skeletons like map and fold which are operationally parallel, as indicated by the presence of the parallel patterns in their cost expressions.

Datum Constant and Binary Data Operator

The cost function of an atomic datum constant d is

$$cost\,(d) = \langle 1, 1, \mathsf{SEQ}, 0 \rangle$$

The shape of datum constant is 1 and so its size is 1. That means the constant is initially stored in the master processor and is transmitted to the processors when it is used for parallel evaluation. It takes no time to evaluate the term d itself and the data pattern of the term itself is SEQ.

The cost function of a binary datum operation d is

$$cost\,(d) = \langle \lambda x. \langle \lambda y. \langle 1, \mathsf{SEQ}, \mathsf{binOpConst} \rangle, \mathsf{SEQ}, 0 \rangle, 0, \mathsf{SEQ}, 0 \rangle$$

Working from the right hand end of the expression in, the first 0 indicates that it takes no time to evaluate the term d itself and SEQ indicates data pattern of the term itself is SEQ. The next 0 indicates that it carries no data (in other words it can be compiled directly onto the processors which use it). The next 0 indicates that it takes no time to apply d to a first argument and SEQ indicates the application pattern involved is SEQ. binOpConst is the time to apply the resulting function to a second argument, that is the time to execute a binary operation. As in the original work for VEC, it and other constants that appear later such as lengthConst are determined "somewhat arbitrarily" and assigned values according to some benchmark of the target machine. In our examples in the thesis, binOpConst is set at 1 and converted to seconds by the instruction rate of a processor gained by running the BSPlib benchmark program on our target architecture. The other constants are set as 0. The next SEQ indicates that the application pattern of the application d to the first argument is SEQ. Finally, 1 is the shape of the result.

Conventional Sequential Functions

The cost function of length is

$$\mathsf{cost}\,(\mathsf{length}) \quad = \langle \lambda x. \langle 1, \mathsf{SEQ}, \mathsf{lengthConst} \rangle, 0, \mathsf{SEQ}, 0 \rangle$$

The first 0 from the right hand end indicates that it takes no time to evaluate the term length itself. The SEQ indicates that the data pattern of the term itself is SEQ. The next 0 indicates that it carries no data. lengthConst is the time to apply length to an argument and SEQ indicates its application pattern is SEQ. Finally, 1 is the shape of the result.

Cost functions of other sequential functions are defined similarly. Note that each cost function captures the impact of shape change. The cost function of hd (usual head function) is

$$cost\,(\mathsf{hd}) = \langle \lambda x. \langle\, \mathsf{t\_eshp}\, x, \mathsf{SEQ}, \mathsf{hdConst} \rangle, 0, \mathsf{SEQ}, 0 \rangle$$

Note that the shape of the result is the shape of the elements of the argument vector, that is $\mathsf{t\_eshp}\, x$ since all the elements of the argument vector have the same shape. The cost function of tl (usual tail function) is

$$cost\,(\mathsf{tl}) = \langle \lambda x. \langle (\mathsf{fst}\, x - 1, \mathsf{t\_eshp}\, x), \mathsf{SEQ}, \mathsf{tlConst} \rangle, 0, \mathsf{SEQ}, 0 \rangle$$

Note that application of tl decreases the length of the argument vector by 1 while it does not change the shape of its elements. The cost function of pair is

$$cost\,(\mathsf{pair}) = \langle \lambda x. \langle \lambda y. \langle \langle x, y \rangle, \mathsf{SEQ}, \mathsf{pairConst} \rangle, \mathsf{SEQ}, 0 \rangle, 0, \mathsf{SEQ}, 0 \rangle$$

The shape of a pair of two arguments is the pair of the shapes of each argument. The cost function of fst is

$$cost\,(\mathsf{fst}) = \langle \lambda x. \langle\, \mathsf{fst}\, x, \mathsf{SEQ}, \mathsf{fstConst} \rangle, 0, \mathsf{SEQ}, 0 \rangle$$

Since fst takes the first component of the pair, the shape of the result is the first component of the shape of the pair. The cost function of snd is

$$cost\,(\mathsf{snd}) = \langle \lambda x. \langle\, \mathsf{snd}\, x, \mathsf{SEQ}, \mathsf{sndConst} \rangle, 0, \mathsf{SEQ}, 0 \rangle$$

Since snd takes the second component of the pair, the shape of the result is the second component of the shape of the pair.

We extend the pair data structure to the tuple data structures, which have more than two components. tuple3 is an extention of pair which constructs a tuple with three components. Correspondingly, fst and snd are also extended to projection operators π_{i3} $(i = 1,2,3)$.

$$cost\,(\mathsf{tuple3}) = \langle \lambda x.\langle \lambda y.\langle \lambda z.\langle \langle x,y,z\rangle, 0, \mathsf{tuple3Const}\rangle, \mathsf{SEQ}, 0\rangle, \mathsf{SEQ}, 0\rangle, 0, \mathsf{SEQ}, 0\rangle$$

$$cost\,(\pi_{13}) = \langle \lambda x.\langle \pi_{13}\,x, \mathsf{SEQ}, \mathsf{projecConst}\rangle, 0, \mathsf{SEQ}, 0\rangle$$

$$cost\,(\pi_{23}) = \langle \lambda x.\langle \pi_{23}\,x, \mathsf{SEQ}, \mathsf{projecConst}\rangle, 0, \mathsf{SEQ}, 0\rangle$$

$$cost\,(\pi_{33}) = \langle \lambda x.\langle \pi_{33}\,x, \mathsf{SEQ}, \mathsf{projecConst}\rangle, 0, \mathsf{SEQ}, 0\rangle$$

The cost functions of tuple constructors which construct a tuple with j (more than three) components and π_{ij} $(i = 1,2,\ldots,j)$ are defined similarly. We often omit j from the notation π_{ij} in the following descriptions to reduce clutter.

Lambda Expression

$$\begin{aligned} cost\,(x) &= \langle x, \mathsf{size}\,x, \mathsf{SEQ}, 0\rangle \\ cost\,(\lambda x.t) &= \langle \lambda x.\langle \pi_1\,(cost\,(t)), 0, \pi_4\,(cost\,(t))\rangle, 0, \mathsf{SEQ}, 0\rangle \end{aligned}$$

Remember that the implementation strategy of the term $(\lambda x.t\,(x))\;a$ is: after x is substituted by the result of evaluation of a, $t\,(a)$ is evaluated in the application part A (figure 3.10).

Since x is substituted by the result of evaluation a, the cost function of x is $\langle x, \mathsf{size}\,x, \mathsf{SEQ}, 0\rangle$. The resulting shape of $\lambda x.t$ is the shape of the result of the application part, that is $\pi_1(cost\,(t))$. The application pattern is SEQ even if t itself involves a parallel function because no communication between the component evaluations part, (E_t and $E_{t'}$), and application part A occurs. The application cost of $\lambda x.t$ is the cost of the application part, that is $\pi_4(cost\,(t))$. The data size, data pattern and cost of $(\lambda x.t\,(x))$ itself are 0, SEQ, 0 respectively. The data size of $(\lambda x.t\,(x))$ is determined as 0 since the evaluation of t is performed in the application process after the value of x is determined. In other words, the lambda expression has no data which is transmitted to the processors. For example, in the term $\mathsf{map}\;\lambda x.(x+(1+2))\,v$ where v is a some vector, 1 and 2 are statically allocated in each processor and $x+(1+2)$ is performed in the application process A in each processor after x is determined.

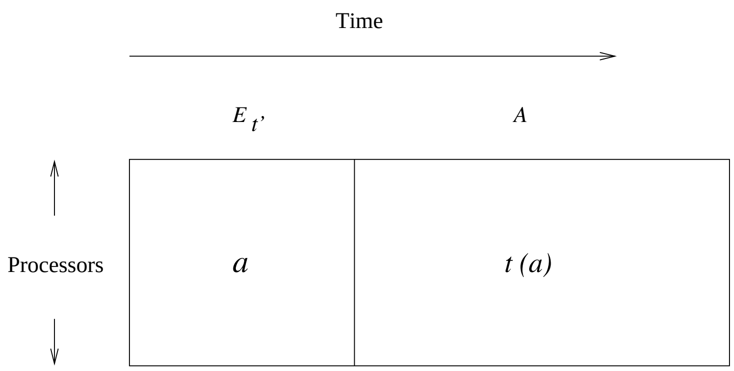


Figure 3.10: Implementation of $(\lambda x.t\,(x))\;a$

If Statements

There are two kinds of conditional. In a data conditional if, we require that both branches have the same shape. We need to define a kind of max operation tuplemax to determine the cost tuple of the branch, which gives upper bound information of each component.

$$\mathsf{tuplemax}\,\langle x,s,d,t\rangle\,\langle x',s',d',t'\rangle = \langle \mathsf{shpmax}\,(x,x'),\mathsf{max}\,(s,s'),\mathsf{max}\,(d,d'),\mathsf{fmax}\,(t,t')\rangle$$

where shpmax gives the maximum of the shape components, taking the pointwise maximum for functions and fmax takes the pointwise maximum of the time functions. The cost function is defined taking account of the cost of the conditional.

$$\begin{aligned} cost\,(\mathsf{if}\ t\ \mathsf{then}\ \ t' \ \ &\mathsf{else}\ t'') \\ = \ &\mathsf{add}_1\,(\mathsf{tuplemax}\,cost\,(t')\,cost\,(t''))\,(\pi_4\,cost\,(t)) \\ &\text{where the definition of } \mathsf{add}_1 \text{ is} \\ &\mathsf{add}_1\,\langle x,s,d,t\rangle\,t' = \langle x,s,d,t+t'\rangle \end{aligned}$$

Even if the branches have the same shape, their costs can be very different in general programs and in that case, the upper bound could lead to a much larger cost than real run time. This is a fundamental problem of static analysis since it alone cannot determine the choice of a branch. It would require the help of a dynamic approach to improve this. This is beyond the scope of our work.

By contrast, since the result of a shape conditional is known statically, the result of the cost function is obtained from the result of the cost function of the taken branch and the cost for evaluation of the conditional.

$$\begin{aligned} \mathit{cost}\,(\mathsf{ifs}\ t\ \mathsf{then}\ \ t'\ \ \mathsf{else}\ t'') \\ = \mathsf{add}_1\,(\mathsf{ifs}\ \pi_1\,\mathit{cost}\,(t)\ \mathsf{then}\ \mathit{cost}\,(t')\ \mathsf{else}\ \mathit{cost}\,(t''))\,(\pi_4\,\mathit{cost}\,(t)) \end{aligned}$$

Iteration

The cost function of iteration iter is

$$\mathit{cost}\,(\mathsf{iter}) = \langle \lambda f.\langle \lambda x.\langle \lambda y.\ \mathsf{iter}\,(\mathsf{preiter}\,f)\langle x, 0, \mathsf{iterConst}\rangle\, y, \mathsf{SEQ}, 0\rangle, \mathsf{SEQ}, 0\rangle, 0, \mathsf{SEQ}, 0\rangle$$

where the definitions of preiter and add_2 are

$$\begin{aligned} \mathsf{preiter}\,f\,\langle x, d, t\rangle &= \mathsf{add}_2\,(f\,x)\,t \\ \mathsf{add}_2\,\langle x, d, t\rangle\,t' &= \langle x, d, t+t'\rangle \end{aligned}$$

Notice that items f, x and y in iter $f\,x\,y$ correspond to the function to be iterated, the initial data and the number of iterations respectively, while preiter adds the structure required to gather costs as iteration proceeds. In the iteration steps, the shape is replaced for the shape of the application result at every step and the cost is accumulated through iteration steps.

3.7 Chapter Conclusion

Our language VEC-BSP is based on the shapely language VEC. The VEC-BSP term is translated into an MSIZE term, which takes the form of a cost tuple that includes the additional information components of data size, application pattern and data patten as well as shape and cost. We predefine the *cost* function for the VEC-BSP primitive constructs and expressions according to our assumptions of cost modeling and our underlying BSP implementation model. The translation from the VEC-BSP program to MSIZE program is performed using these predefined *cost* functions as translation rules.

BSP costing is carried out by the evaluation of the translated MSIZE program, in which bspapp plays the central rule of including the communication cost and synchronisation cost as well as computation cost by using the information from the cost tuples.

Chapter 4

Implementation Templates for Skeletons

4.1 Introduction

In parallel programming, some kinds of pattern of parallel control structure often appear in different application programs. The idea of the concept called algorithmic skeletons [25, 29] or parallel program paradigms [19] is to separate these common parallel structures from details of applications and predefine them as program components. In skeletal programming, a program is composed of pre-defined parallel components that implement parallel control structures and sequential components for a specific application.

In our context, each predefined parallel combinator such as map and fold represents a skeleton and has a parallel control structure implementation template which follows the BSP model. Programs are composed of those skeletons and other sequential components.

In this chapter, we define our BSP implementation template for the skeletons of VEC-BSP and their shapes to complete the definition of *cost*. The application cost of each combinator is defined as a function of shape so that it can be embedded in our shape-based cost analysis framework. We express our implementations in an SPMD pseudo-code, indicating calls to the standard BSP operations bsp_put (copy to remote memory), bsp_get (copy from remote memory), bsp_sync (barrier synchronisation), and bsp_pid (find my process identifier) [48].

4.2 Implementation and Costing of the Parallel Combinators

4.2.1 map

map applies some function t_f to each element of an argument vector $t_x = [x_1, x_2, \cdots, x_n]$.

$$\mathsf{map}\; t_f\; [x_1, x_2, \cdots, x_n] = [t_f(x_1), t_f(x_2), \cdots, t_f(x_n)]$$

It has a simple parallel implementation in which the same operation is applied to each element in the segment distributed to each processor. This corresponds to Darlington's FARM skeleton [29] and plays a central role in other paradigms.

Our BSP implementation strategy for map is:

1. the data in t_f is broadcast and the data in t_x is scattered to all p processors;
2. synchronisation;
3. each processor applies t_f to the local element of t_x;
4. the local result in each processor is gathered to the master processor;
5. synchronisation.

The corresponding SPMD pseudo-code is:

```
bsp_get(data describing t_f from P0);
bsp_get(local share of t_x from P0);
```

```
bsp_sync();
for each local item
   apply t_f to this local element of t_x;
bsp_put(result to P0);
bsp_sync();
```

The application cost of map in terms of the BSP cost model takes the form of a function of shapes of arguments. We express the shapes of the t_f and t_x as f and x respectively. The application cost is: the local computation cost $\mathsf{t\_apcost}\,(f\,(\mathsf{t\_eshp}\,(x)))\cdot(\mathsf{t\_len}\,(x)/p)$ for step 3, the communication cost

$$\mathsf{t\_size}\,(f\,(\mathsf{t\_eshp}\,(x)))\cdot(\mathsf{t\_len}\,(x)/p)\cdot(p-1)\cdot g$$

for step 4, and the synchronisation cost l for step 5. Thus, the overall application cost is

$$\begin{aligned} & \mathsf{apcost\_map}\,f\,x\,\langle p,g,l\rangle \\ = \ & \mathsf{t\_apcost}\,(f\,(\mathsf{t\_eshp}\,(x)))\cdot(\mathsf{t\_len}\,(x)/p) \\ & +\mathsf{t\_size}\,(f\,(\mathsf{t\_eshp}\,(x)))\cdot(\mathsf{t\_len}\,(x)/p)\cdot(p-1)\cdot g+l \end{aligned}$$

Note that the application cost does not include the costs for step 1 and step 2, which are computed by the bspapp operation as a communication cost of C. The shape of map is:

$$\lambda f.\langle\lambda x.\langle(\mathsf{t\_len}\,(x),\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,x))),\mathsf{MAP},\mathsf{apcost\_map}\,f\,x\rangle,\mathsf{SEQ},0\rangle$$

Notice that this is the first component of $cost\,(\mathsf{map})$. This packs in the following information. Working from the right hand end, it takes no time to apply map to a given function t_f and its application pattern is SEQ. The application cost to apply $\mathsf{map}\,t_f$ to a given vector is apcost_map, which was given above. The application pattern used to apply $\mathsf{map}\,t_f$ to the given vector is MAP since the implementation skeleton ends by gathering the local result. The resulting shape of the application $\mathsf{map}\,t_f$ to the given vector is $(\mathsf{t\_len}\,(x),\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(x))))$.

The cost function of map is

$$cost\,(\mathsf{map}) \;=\; \langle \lambda f.\langle \lambda x.\langle (\,\mathsf{t\_len}\,(x), \mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,x))), \mathsf{MAP}, \mathsf{apcost\_map}\,f\,x\rangle, \mathsf{SEQ}, 0\rangle, 0, \mathsf{SEQ}, 0\rangle$$

Working from the right hand end of the expression in, the first 0 indicates that it takes no time to evaluate the term map itself and data pattern of the term map itself is SEQ. The next 0 indicates that it carries no data (in other words that it can be compiled directly onto the processors which use it). The next component

$$\lambda f.\langle \lambda x.\langle (\,\mathsf{t\_len}\,(x), \mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,x))), \mathsf{MAP}, \mathsf{apcost\_map}\,f\,x\rangle, \mathsf{SEQ}, 0\rangle$$

is the shape of term map, which was explained above.

4.2.2 fold

fold combines the elements of a vector $t_x = [x_1, x_2, \cdots, x_n]$ using an associative binary operator $t_\oplus$.

$$\mathsf{fold}\; t_\oplus\; [x_1, x_2, \cdots, x_n] = x_1\, t_\oplus\, x_2\, t_\oplus \cdots t_\oplus\, x_n$$

The combination of map and fold forms the important "map and reduce" paradigm in BMF [77]. fold has a few possible BSP implementations. One is the well-known tree-like structure implementation. For example, the result of fold $(+)$ x (that generates sum of n elements of an argument vector x), can be calculated in two stages where each of the processors sequentially sums the values in their possession in time $O(\frac{n}{p})$, and then parallel sum of the resulting p values can be obtained using the logarithmic technique in time $O(\log p)$. This asymptotic cost analysis of the logarithmic summation can be refined into the BSP cost calculus by considering the communication and synchronisation costs of a single stage of the logarithmic algorithm. Combining the cost of locally summing each processor's $\frac{n}{p}$ values with the cost of the summation of p values gives a total cost of summing n values on p processes as $\frac{n}{p} + \log p(1+g+l)$. From this cost formula, and from the values of l and g for typical parallel machines [47] (e.g. $p = 16$, $l = 751$ [flops] and $g = 1.6$ [flops/word] for a 16 processor Cray T3E), it can be seen that the logarithmic number of barrier synchronisations used in this algorithm will form the dominant cost, unless $n > p \log p(1+g+l)$. Therefore, although the logarithmic fold minimises the computational cost of summing p values, it places a great burden on the communication performance. Another implementation is that after the cost of locally summing, the resulting p values are gathered into one processor where sequential sum is calculated. Its BSP cost is $\frac{n}{p}+(p-1)+(p-1)g+l$. This reduces the synchronisation cost significantly and the total cost is usually smaller than the former when p is a moderate number. For example, $\log p(1+g+l) = 3014.4$ and $(p-1)+(p-1)g+l = 790$ on a 16 processor Cray T3E. This example shows the importance of considering the communication and synchronisation costs as well as the computation cost. We take the latter implementation strategy for implementation of fold to avoid the logarithmic number of barrier synchronisations and to reduce the complexity of the analysis cost.

1. the data in $t_{\oplus}$ is broadcast and the data in t_x is scattered from the master processor to all p processors;
2. synchronisation;
3. each processor folds the vector segments with $t_{\oplus}$;
4. the local result in each processor is gathered to the master processor;
5. synchronisation;
6. the gathered local results are folded with $t_{\oplus}$ in the master processor.

The corresponding SPMD pseudo-code is:

```
bsp_get(data describing t⊕ from P0);
bsp_get(local share of tx from P0);
bsp_sync();
for each local item
   combine this item into emerging local result;
if (p > 1) {
  bsp_put(local result to P0);
  bsp_sync();
```

```
  if (bsp_pid() == 0)
    sequentially fold together collected sub results;
}
```

The cost of each part of the application of fold is as follows. The computation cost is $\mathsf{t\_apcost}\,(iter\_rlt)$ for step 3 and

$$\mathsf{t\_apcost}\,(\mathsf{iter}\,(\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_shp}\,(iter\_rlt)))))\langle \mathsf{t\_shp}\,(iter\_rlt),\mathsf{SEQ},0\rangle(p-1))$$

for step 6, where

$$iter\_rlt = \mathsf{iter}\,(\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))))\langle \mathsf{t\_eshp}\,(x),\mathsf{SEQ},0\rangle((\mathsf{t\_len}\,(x)/p)-1)$$

and the definitions of preiter and add_2 are

$$\begin{aligned}\mathsf{preiter}\,f\,\langle x,d,t\rangle &= \mathsf{add}_2\,(f\,x)\,t\\ \mathsf{add}_2\,\langle x,d,t\rangle\,t' &= \langle x,d,t+t'\rangle\end{aligned}$$

Note that the $iter\_rlt$ denotes the result of the initial local folding phase. The arguments of iter, that is $\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x))))$, $\langle \mathsf{t\_eshp}\,(x),\mathsf{SEQ},0\rangle$, and $(\mathsf{t\_len}\,(x)/p)-1$ correspond to the function to be iterated, the initial data and the number of iterations respectively, while preiter adds the structure required to gather costs as iteration proceeds. The iter combinator is required to model the repeated application of the $t_{\oplus}$ to allow for situations in which the resulting shapes of the intermediate results are not same as the shape of the original elements. This means we do not need different fold operators (so it is an improvement on Skillicorn's and Rangaswami's schemes, for example). The communication cost is $\mathsf{t\_size}\,(iter\_rlt)\cdot(p-1)\cdot g$ for step 4. The synchronisation cost is l for step 5. Thus the overall application cost of fold is expressed as

$$\begin{aligned}&\mathsf{ap\_cost\_fold}\,\oplus\,x\,\langle p,g,l\rangle\\ =\;&\mathsf{t\_apcost}\,(iter\_rlt)+\mathsf{t\_size}\,(iter\_rlt)\cdot(p-1)\cdot g+l+\\ &\mathsf{t\_apcost}\,(\mathsf{iter}\,(\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_shp}\,(iter\_rlt)))))\langle \mathsf{t\_shp}\,(iter\_rlt),\mathsf{SEQ},0\rangle(p-1))\end{aligned}$$

The shape of fold is:

$$\lambda\oplus.\langle\lambda x.\langle \mathsf{t\_shp}\,(\mathsf{iter}\,(\mathsf{preiter}\,(\oplus\,(\mathsf{t\_shp}\,(iter\_rlt))))\langle \mathsf{t\_shp}\,(iter\_rlt),\mathsf{SEQ},0\rangle(p-1)),$$
$$\mathsf{FOLD},\mathsf{ap\_cost\_fold}\oplus x\rangle,\mathsf{SEQ},0\rangle$$

This packs in the following information. Working from the right hand end, it takes no time to apply fold to a given function $t_\oplus$ and its application pattern is SEQ. The application cost to apply fold $t_\oplus$ to a given vector is apcost_fold, which is given above. The application pattern involved to apply fold $t_\oplus$ to the given vector is FOLD since the implementation skeleton ends with the sequential folding. The resulting shape of the application fold $t_\oplus$ to the given vector is

$$\mathsf{t\_shp}\,(\mathsf{iter}\,(\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_shp}\,(iter\_rlt)))))\langle \mathsf{t\_shp}\,(iter\_rlt),\mathsf{SEQ},0\rangle(p-1))$$

Notice that the preiter adds the structure required to gather costs as iteration proceeds in the sequential folding phase.

The cost function of fold is

$$\begin{aligned} cost\,(\mathsf{fold}) \quad = \quad & \langle\lambda\oplus.\langle\lambda x.\langle \mathsf{t\_shp}\,(\mathsf{iter}\,(\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_shp}\,(iter\_rlt))))) \\ & (\mathsf{t\_shp}\,(iter\_rlt))(p-1)),\mathsf{FOLD},\mathsf{ap\_cost\_fold}\oplus x\rangle,\mathsf{SEQ},0\rangle,0,\mathsf{SEQ},0\rangle \end{aligned}$$

where

$$iter\_rlt = \mathsf{iter}\,(\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))))\langle \mathsf{t\_eshp}\,(x),\mathsf{SEQ},0\rangle((\mathsf{t\_len}\,(x)/p)-1)$$

The first 0 from the right hand end indicates that it takes no time to evaluate the term fold itself and data pattern of the term fold itself is SEQ. The next 0 indicates that it carries no data. The next component

$$\lambda\oplus.\langle\lambda x.\langle \mathsf{t\_shp}\,(\mathsf{iter}\,(\mathsf{preiter}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_shp}\,(iter\_rlt)))))(\mathsf{t\_shp}\,(iter\_rlt))(p-1)),$$
$$\mathsf{FOLD},\mathsf{ap\_cost\_fold}\oplus x\rangle,\mathsf{SEQ},0\rangle$$

is the shape of the term fold, which was explained above.

4.2.3 scan

scan applies to a vector $t_x = [x_1, x_2, \cdots, x_n]$ and the partial result of fold $t_\oplus$ up to the ith element is returned as the ith element of the resulting vector.

$$\mathsf{scan}\, t_\oplus\, [x_1, x_2, \cdots, x_n] = [x_1,\, x_1\, t_\oplus\, x_2, \cdots, x_1\, t_\oplus\, x_2\, t_\oplus \cdots t_\oplus\, x_n]$$

It is known that the scan operation is useful for describing various data-parallel algorithms, and leads to efficient run time codes. For example, [12] describes five algorithms that illustrate how the scan can be used in algorithm design: a radix-sort, a quick sort, a minimum-spanning-tree algorithm, a line-drawing algorithm and a merging algorithm. In some parallel computation models such as the Scan Vector Model [13], simple operations are implemented through scan.

Our BSP implementation strategy for scan is:

1. the data in $t_\oplus$ is broadcast and the data in t_x is scattered to all p processors;
2. synchronisation;
3. each processor scans the segment distributed from the master processor with $t_\oplus$;
4. the final element of the local scan in each processor is scanned across processors with $t_\oplus$ using the obvious tree algorithm, which involves $\log p$ iterations of a (data transmission + synchronisation + execution of $t_\oplus$) process;
5. the result of the global scan in the processor $i\,(< p)$ is sent to processor $i+1$;
6. synchronisation;
7. each processor applies $t_\oplus$ to the pair of the value sent to the processor in 5 and each element of the results in 3;
8. the local result in each processor is gathered to the master processor;
9. synchronisation.

The corresponding SPMD pseudo-code is:

```
bsp_get(data describing t⊕ from P0);
bsp_get(local share of tx from P0);
bsp_sync();
for each local item
   scan this item into emerging local result;
if (p > 1){
   for(i=1; log(p); i++) {
     send the item to the next processor;
     bsp_sync();
     apply t⊕ to the item and the sent item;
  }
send the item of the result of the global scan to the next processor;
apply t⊕ to the sent item and each item;
bsp_put(local result to P0);
bsp_sync();
}
```

The cost of each part of the application of scan is as follows. The computation cost is

$$\mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(x)))\cdot(\mathsf{t\_len}\,(x)/p-1)$$

for step 3,

$$\mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(x)))\cdot\log(p)$$

for the computation part in step 4, and

$$\mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(x)))\cdot(\mathsf{t\_len}\,(x)/p)$$

for step 7. As this cost is necessary only when $p > 1$ but this formula does not result in 0 when $p = 1$ (therefore, $g = l = 0$), we use the formula

$$(\mathsf{eqone}\ p)\cdot\mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(x)))\cdot(\mathsf{t\_len}\,(x)/p)$$

where eqone is defined as $\mathsf{eqone}\ y = \mathsf{if}\ \ (y=1)\ \ \mathsf{then}\ \ 0\ \ \mathsf{else}\ \ 1.$

The communication cost is $\mathsf{size}\,(\mathsf{t\_eshp}\,(x)) \cdot g \cdot \log(p)$ for the communication part in step 4, $(\mathsf{size}\,(\mathsf{t\_eshp}\,(x)) \cdot g$ for step 5, and $(\mathsf{size}\,(x)/p) \cdot (p-1) \cdot g$ for step 8. The synchronisation cost is $l \cdot \log(p)$ for the synchronisation part in step 4, l for step 6 and l for step 9. Therefore, the overall application cost is expressed as

$$
\begin{aligned}
& \mathsf{ap\_cost\_scan} \oplus x \langle p, g, l \rangle \\
= \; & \mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(x))) \cdot ((\mathsf{t\_len}\,(x)/p + \mathsf{eqone}\; p) + \log(p) - 1) \\
& + (\mathsf{size}\,(\mathsf{t\_eshp}\,(x)) \cdot (\log(p) + 1) + (\mathsf{size}\,(x)/p) \cdot (p-1)) \cdot g \\
& + (\log(p) + 2) \cdot l
\end{aligned}
$$

The shape of scan is:

$$\lambda \oplus . \langle \lambda x. \langle x, \mathsf{MAP}, \mathsf{ap\_cost\_scan} \oplus x \rangle, \mathsf{SEQ}, 0 \rangle$$

This packs in the following information. Working from the right hand end, it takes no time to apply scan to a given function $t_{\oplus}$ and its application pattern is SEQ. The application cost to apply $\mathsf{scan}\, t_{\oplus}$ to a given vector is apcost_scan, which was given above. The application pattern involved to apply $\mathsf{scan}\, t_{\oplus}$ to the given vector is MAP since the implementation skeleton ends by gathering the local results. The resulting shape of the application $\mathsf{scan}\, t_{\oplus}$ to the given vector is the same as the shape of the vector t_x since we assumed that application of $t_{\oplus}$ does not change the shape.

The cost function of scan is

$$cost\,(\mathsf{scan}) = \langle \lambda f. \langle \lambda x. \langle x, \mathsf{MAP}, \mathsf{ap\_cost\_scan}\, f\, x \rangle, \mathsf{SEQ}, 0 \rangle, 0, \mathsf{SEQ}, 0 \rangle$$

The first 0 from the right hand end indicates that it takes no time to evaluate the term scan itself and data pattern of the term scan itself is SEQ. The next 0 indicates that it carries no data. The next component

$$\lambda f. \langle \lambda x. \langle x, \mathsf{MAP}, \mathsf{ap\_cost\_scan}\, f\, x \rangle, \mathsf{SEQ}, 0 \rangle$$

is the shape of the term scan, which was explained above.

4.2.4 pair_map

pair_map applies a function t_f to pairs of elements drawn from a pair of vectors of the same length.

$$\mathsf{pair\_map}\, t_f\, ([x_1, x_2, \cdots, x_n], [x'_1, x'_2, \cdots, x'_n]) = [(t_f\, x_1\, x'_1), (t_f\, x_2\, x'_2), \cdots, (t_f\, x_n\, x'_n)]$$

Our BSP implementation strategy for pair_map is:

1. the data in t_f is broadcast and the data in fst t_x and snd t_x (the two vectors) are scattered to all p processors;
2. synchronisation;
3. each processor applies t_f to the elementwise pairs;
4. the local result in each processor is gathered to the master processor;
5. synchronisation.

In SPMD pseudo-code this is:

```
bsp_get(data describing t_f from P0);
bsp_get(local share of fst t_x from P0);
bsp_get(local share of snd t_x from P0);
for each local items from t_x
   apply f_x to local (fst t_x) and corresponding local (snd t_x);
bsp_put(results to P0);
bsp_sync();
```

The application cost of pair_map in terms of the BSP cost model is the computation cost $\mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\,x)))(\mathsf{t\_eshp}\,(\mathsf{snd}\,x))) \cdot (\mathsf{t\_len}\,(\mathsf{fst}\,x)/p)$ for step 3, the communication cost $\mathsf{t\_size}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\,x)))(\mathsf{t\_eshp}\,(\mathsf{snd}\,x))) \cdot (p-1) \cdot g$ for step 4 and the synchronisation cost l for step 5. Thus, the overall cost is expressed as

$$\begin{aligned} & \mathsf{ap\_cost\_pair\_map}\, f\, x \langle p, g, l \rangle \\ = \; & \mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\,x)))(\mathsf{t\_eshp}\,(\mathsf{snd}\,x))) \cdot (\mathsf{t\_len}\,(\mathsf{fst}\,x)/p) \end{aligned}$$

$$+\,\mathsf{t\_size}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\,x)))(\mathsf{t\_eshp}\,(\mathsf{snd}\,x)))\cdot(p-1)\cdot g+l$$

The shape of pair_map is:

$$\lambda f.\langle\lambda x.\langle(\mathsf{t\_len}\,(\mathsf{fst}\,x),\mathsf{t\_shp}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\,x)))(\mathsf{t\_eshp}\,(\mathsf{snd}\,x)))),\mathsf{MAP},$$
$$\mathsf{ap\_cost\_pair\_map}\,f\,x\rangle,\mathsf{SEQ},0\rangle$$

This packs in the following information. Working from the right hand end, it takes no time to apply pair_map to a given function t_f and its application pattern is SEQ. The application cost to apply pair_mapt_f to a given vector is apcost_pair_map, which is given above. The application pattern involved to apply pair_mapt_f to the given vector is MAP since the implementation skeleton ends by gathering the local result. The resulting shape of the application pair_mapt_f to the given vector is

$$(\mathsf{t\_len}\,(\mathsf{fst}\,x),\mathsf{t\_shp}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\,x)))\,(\mathsf{t\_eshp}\,(\mathsf{snd}\,x))))$$

The cost function of pair_map is

$$\begin{aligned}cost\,(\mathsf{pair\_map}) \;=\; & \langle\lambda f.\langle\lambda x.\langle(\mathsf{t\_len}\,(\mathsf{fst}\,x),\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\,x))(\mathsf{t\_eshp}\,(\mathsf{snd}\,x)))),\\ & \mathsf{MAP},\mathsf{ap\_cost\_pair\_map}\,f\,x\rangle,\mathsf{SEQ},0\rangle,0,\mathsf{SEQ},0\rangle\end{aligned}$$

The first 0 from the right hand end indicates that it takes no time to evaluate the term pair_map itself and SEQ indicates data pattern of the term pair_map itself is SEQ. The next 0 indicates that it carries no data. The next component

$$\lambda f.\langle\lambda x.\langle(\mathsf{t\_len}\,(\mathsf{fst}\,x),\mathsf{t\_shp}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(\mathsf{fst}\,x)))(\mathsf{t\_eshp}\,(\mathsf{snd}\,x)))),\mathsf{MAP},$$
$$\mathsf{ap\_cost\_pair\_map}\,f\,x\rangle,\mathsf{SEQ},0\rangle$$

is the shape of the term pair_map, which was explained above.

4.2.5 c_prod

c_prod applies a function to each member of the cross-product of two vectors t_x and t_y.

$$\begin{aligned}\mathsf{c\_prod}\,t_f\,[x_1,x_2,\cdots,x_m][y_1,y_2,\cdots,y_n] \;=\; & [[t_f\,x_1\,y_1,\ t_f\,x_2\,y_1,\ \cdots,t_f\,x_m\,y_1],\\ & [t_f\,x_1\,y_2,\ t_f\,x_2\,y_2,\cdots,t_f\,x_m\,y_2],\\ & \quad\vdots\\ & [t_f\,x_1\,y_n,\ t_f\,x_2\,y_n,\cdots,t_f\,x_m\,y_n]]\end{aligned}$$

It is used for a class of algorithms in which each object interacts with every other and corresponds to the All-Pairs Paradigm in [19] and the RaMP(Reduce-and-Map-over-Pairs) skeleton in Darlington's skeletons [29].

The BSP implementation strategy for c_prod is:

1. the data in t_f and in t_x is broadcast and the data in t_y are scattered to all p processors;
2. synchronisation;
3. each processors applies t_f to the all members of the cross product of t_x and the local segments of t_y;
4. the local result in each processor is gathered to the master processor;
5. synchronisation

In SPMD pseudo-code this is:

```
bsp_get(data describing f from P0);
bsp_get(copy of t_x from P0);
bsp_get(local share of t_y from P0);
bsp_sync();
for each local item t_y' from t_y
  for each item t_x' from copy of t_x
     apply t_f to t_x' and t_y';
bsp_put(results to P0);
bsp_sync();
```

The application cost for c_prod is: the communication cost

$$\mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(y)))\cdot(\mathsf{t\_len}\,(y)/p)\cdot(\mathsf{t\_len}\,(x))$$

for step 3, the communication cost

$$(\mathsf{t\_size}\,(\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(y)))\cdot(\mathsf{t\_len}\,(y)/p)\cdot(\mathsf{t\_len}\,(x))\cdot(p-1))\cdot g$$

for step 4 and the synchronisation cost l for step 5. Thus, the overall cost is expressed as

$$\begin{aligned}
& \mathsf{ap_cost_c_prod}\, f\, x\, y\, \langle p, g, l\rangle \\
= \;& \mathsf{t_apcost}\,(\mathsf{t_shp}\,(f\,(\mathsf{t_eshp}\,(x)))(\mathsf{t_eshp}\,(y)))\cdot(\mathsf{t_len}\,(y)/p)\cdot(\mathsf{t_len}\,(x)) \\
& +(\mathsf{t_size}\,(\mathsf{t_shp}\,(f\,(\mathsf{t_eshp}\,(x)))(\mathsf{t_eshp}\,(y)))\cdot(\mathsf{t_len}\,(y)/p)\cdot(\mathsf{t_len}\,(x))\cdot(p-1)) \\
& \cdot g + l
\end{aligned}$$

The shape of c_prod is:

$$\begin{aligned}
& \lambda f.\langle\lambda x.\langle\lambda y.\langle(\mathsf{t_len}\,(y),(\mathsf{t_len}\,(x),\mathsf{t_shp}\,(\mathsf{t_shp}(f\,x)\,y))),\mathsf{MAP},\mathsf{ap_cost_c_prod}\,f\,x\,y\rangle, \\
& \mathsf{SEQ},0\rangle,\mathsf{SEQ},0\rangle
\end{aligned}$$

This packs in the following information. Working from the right hand end, it takes no time to apply c_prod to a given function t_f and its application pattern is SEQ. The application cost to apply c_prod t_f to a given vector t_x is 0 and its application pattern is SEQ. The application cost to apply c_prod $t_f\ t_x$ to a given vector t_y is apcost_c_prod, which is given above. The application pattern involved to apply c_prod $t_f\ t_x$ to the given vector t_y is MAP since the implementation skeleton ends by gathering the local result. The resulting shape of the application c_prod $t_f\ t_x$ to the given vector t_y is $(\mathsf{t\_len}\,y,(\mathsf{t\_len}\,x,\mathsf{t\_shp}\,(\mathsf{t\_shp}\,(f\,x)\,y)))$.

The cost function of c_prod is

$$\begin{aligned}
cost\,(\mathsf{c_prod}) \;=\; & \langle\lambda f.\langle\lambda x.\langle\lambda y.\langle(\mathsf{t_len}\,y,(\mathsf{t_len}\,x,\mathsf{t_shp}\,(\mathsf{t_shp}\,(f\,x)\,y))), \\
& \mathsf{MAP},\mathsf{ap_cost_c_prod}\,f\,x\,y\rangle,\mathsf{SEQ},0\rangle,\mathsf{SEQ},0\rangle,0,\mathsf{SEQ},0\rangle
\end{aligned}$$

The first 0 from the right hand end indicates that it takes no time to evaluate the term c_prod itself and SEQ indicates data pattern of the term c_prod itself is SEQ. The next 0 indicates that it carries no data. The next component

$$\begin{aligned}
& \lambda f.\langle\lambda x.\langle\lambda y.\langle(\mathsf{t_len}\,y,(\mathsf{t_len}\,x,\mathsf{t_shp}\,(\mathsf{t_shp}\,(f\,x)\,y))),\mathsf{MAP},\mathsf{ap_cost_c_prod}\,f\,x\,y\rangle, \\
& \mathsf{SEQ},0\rangle,\mathsf{SEQ},0\rangle
\end{aligned}$$

is the shape of the term c_prod, which was explained above.

4.3 Chapter Conclusion

A common approach to cost consideration for the skeleton approach is to formulate the cost of each skeleton based on its low level implementation using some parameters. In our context, skeletons are higher-order functions each of which has a predetermined template based on the BSP computation model. We presented details of the algorithm and its SPMD pseudo-code for the implementation of each skeleton and defined a cost formula in the form of a function of shape and the BSP parameters.

In skeleton-based models in which a parallel algorithm is expressed using more than one skeleton, cost would be expressed as some kind of combination of the cost formula of each single skeleton. However, a simple summation of each formula does not work well because the input size (or shape) parameterised in each formula will take different values in the general case. We need to take account of the impact of size (or shape) changes between skeletons. The distinguishing feature of shape-based cost analysis is that the composition of these formula can be automated by the incorporation of automatic shape analysis. Our analysis adds to this feature the ability to compute the communication and synchronisation cost considering impact of architecture characteristics through BSP parameters.

Efficiency of the BSP implementation of each skeleton could be improved by investigating the costs of possible alternative implementations and the implications of parameter sizes. This remains as future work.

Chapter 5

Implementation of Cost Analysis

This chapter outlines the Haskell implementation of our cost analysis, which was described in chapters 3 and 4. It illustrates some details of the system structure and definitions of functions by using examples rather than full source code. The system was developed by modifying the Haskell implementation of PRAM cost analysis developed at the University of Technology Sydney, reflecting the amendments to achieve our BSP cost analysis. The basic structure of the system is based on that of the original PRAM cost analysis implementation.

5.1 Automating Cost Analysis

The natural use of our system would be as an aid during program development, allowing the programmer to experiment with the behaviour of various equivalent program structures on various data sets. Since the cornerstone of shapely programming is that behavioural structure is independent of data content, it would be both unnecessary and time-consuming to require the provision of real data sets during development (e.g. constructing an array of 1000 by 1000 values only for the cost calculator to immediately throw them away). Thus, for development purposes we add a new constructor dummyvec, which allows the programmer to directly specify the input shape as its argument, and use dummyvec *ishp* instead of the real input data vector. This would be replaced by calls to IO operations in the real program. The cost function for dummyvec *ishp* is simple, as the programmer provides the input shape directly.

$$cost\ (\mathsf{dummyvec}\ ishp) = \langle ishp,\ \mathsf{sz}\ ishp,\ 0,\ \mathsf{SEQ},\ 0\rangle$$

Note that this implies that we are not costing the I/O for the real program.

5.2 Example of Cost Analysis by Hand

The analysis process can be illustrated by an example of a complete cost derivation by hand. We derive the cost of

$$\mathsf{map}\ (+9)\ [1,2,3,4,5,6,7,8]$$

The input data $[1,2,3,4,5,6,7,8]$ is replaced by dummyvec $(8,1)$ by the programmer before *cost* is applied.

$$\begin{aligned}
& cost\ (\mathsf{map}\ (+9)\ \mathsf{dummyvec}\ (8,1)) \\
= \ & \{\text{def. of } cost\ t\ t'\} \\
& \mathsf{bspapp}\ cost\ (\mathsf{map}\ (+9))\ cost\ (\mathsf{dummyvec}\ (8,1)) \\
= \ & \{\text{def. of } cost\ t\ t' \text{ and def. of } cost\ (\mathsf{dummyvec}\ (8,1))\} \\
& \mathsf{bspapp}\ (\mathsf{bspapp}\ cost\ (\mathsf{map})\ cost\ (+9))\ \langle(8,1),8,\mathsf{SEQ},0\rangle \\
= \ & \{\text{def. of } cost\ t\ t'\} \\
& \mathsf{bspapp}\ (\mathsf{bspapp}\ cost\ (\mathsf{map})\ (\mathsf{bspapp}\ cost\ (+)\ cost\ 9))\ \langle(8,1),8,\mathsf{SEQ},0\rangle
\end{aligned}$$

Within the above,

$$\begin{aligned}
& cost\ (\mathsf{map}) \\
= \ & \{\text{def. of } cost\ \mathsf{map}\} \\
& \langle \lambda f.\langle \lambda x.\langle (\mathsf{t_len}\,(x), \mathsf{t_shp}\,(f\,(\mathsf{t_eshp}\,x))), \mathsf{MAP}, \mathsf{apcost_map}\,f\,x\rangle, \\
& \mathsf{SEQ},0\rangle,0,\mathsf{SEQ},0\rangle
\end{aligned}$$

and

$$
\begin{aligned}
& \textsf{bspapp}\ \mathit{cost}\ (+)\ \mathit{cost}\ 9 \\
= \ & \{\text{def. of } \mathit{cost}\ + \text{ and } \mathit{cost}\ 9\} \\
& \textsf{bspapp}\ \langle \lambda x.\langle \lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle, \textsf{SEQ}, 0\rangle, 0, \textsf{SEQ}, 0\rangle \langle 1, 1, \textsf{SEQ}, 0\rangle
\end{aligned}
$$

Therefore, the translated MSIZE program is

$$
\begin{aligned}
& \textsf{bspapp}\ (\textsf{bspapp}\ \langle \lambda f.\langle \lambda x.\langle (\textsf{t\_len}\,(x), \textsf{t\_shp}\,(f\,(\textsf{t\_eshp}\,x))), \textsf{MAP}, \textsf{apcost\_map}\,f\,x\rangle, \\
& \textsf{SEQ}, 0\rangle, 0, \textsf{SEQ}, 0\rangle (\textsf{bspapp}\ \langle \lambda x.\langle \lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle, \textsf{SEQ}, 0\rangle, 0, \textsf{SEQ}, 0\rangle \\
& \langle 1, 1, \textsf{SEQ}, 0\rangle))\ \langle (8,1), 8, \textsf{SEQ}, 0\rangle
\end{aligned}
$$

Within the above,

$$
\begin{aligned}
& \textsf{bspapp}\ \langle \lambda x.\langle \lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle, \textsf{SEQ}, 0\rangle, 0, \textsf{SEQ}, 0\rangle \langle 1, 1, \textsf{SEQ}, 0\rangle \\
= \ & \{\text{def. of } \textsf{bspapp}\} \\
& \langle \lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle, 1, \textsf{SEQ}, 0\rangle
\end{aligned}
$$

And so,

$$
\begin{aligned}
& \textsf{bspapp}\ \langle \lambda f.\langle \lambda x.\langle (\textsf{t\_len}\,(x), \textsf{t\_shp}\,(f\,(\textsf{t\_eshp}\,x))), \textsf{MAP}, \textsf{apcost\_map}\,f\,x\rangle, \\
& \textsf{SEQ}, 0\rangle, 0, \textsf{SEQ}, 0\rangle\ \langle \lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle, \textsf{SEQ}, 0\rangle, 1, \textsf{SEQ}, 0\rangle \\
= \ & \{\text{def. of } \textsf{bspapp}\} \\
& \langle \lambda x.\langle (\textsf{t\_len}\,(x), \textsf{t\_shp}\,(\lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle\ (\textsf{t\_eshp}\,x))), \\
& \textsf{MAP}, \textsf{apcost\_map}\ \lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle\ x\rangle, 1, \textsf{MAP}, 0\rangle \\
& \text{where} \\
& \quad \textsf{apcost\_map}\ \lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle\, x \\
& = \{\text{def. of } \textsf{apcost\_map}\} \\
& \quad \textsf{t\_apcost}\,(\lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle(\textsf{t\_eshp}\,(x))) \cdot (\textsf{t\_len}\,(x)/p) \\
& \quad + \textsf{t\_size}\,(\lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle\ (\textsf{t\_eshp}\,(x))) \cdot (\textsf{t\_len}\,(x)/p) \cdot (p-1) \cdot g + l
\end{aligned}
$$

Thus,

$$
\begin{aligned}
& \textsf{bspapp}\ \langle \lambda x.\langle (\textsf{t\_len}\,(x), \textsf{t\_shp}\,(\lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle\ (\textsf{t\_eshp}\,x))), \\
& \textsf{MAP}, \textsf{apcost\_map}\ \lambda y.\langle 1, \textsf{SEQ}, \textsf{binOpConst}\rangle\, x\rangle, 1, \textsf{MAP}, 0\rangle\ \langle (8,1), 8, \textsf{SEQ}, 0\rangle \\
= \ & \{\text{def. of } \textsf{bspapp}\}
\end{aligned}
$$

$$
\begin{aligned}
& \langle (8,1), \mathsf{t\_size}\ (8,1), \mathsf{MAP}, (\mathsf{comm\_cost}\ 1\ 0\ 0) + l + \\
& \mathsf{apcost\_map}\, \lambda y. \langle 1, \mathsf{SEQ}, \mathsf{binOpConst} \rangle\ (8,1) \rangle \\
= \;& \{\text{def. of } \mathsf{apcost\_map} \text{ etc.}\} \\
& \langle (8,1), 8, \mathsf{MAP}, (1 \cdot (p-1) + 8 \cdot ((p-1)/p)) \cdot g + l + \\
& \mathsf{binOpConst} \cdot (8/p) + 1 \cdot (8/p) \cdot (p-1) \cdot g + l \rangle \\
= \;& \langle (8,1), 8, \mathsf{MAP}, (8 \cdot \mathsf{binOpConst}/p) + (p + 15 - 16/p) \cdot g + 2 \cdot l \rangle
\end{aligned}
$$

The result shows that the cost of

$$\mathsf{map}\ (+9)\ [1,2,3,4,5,6,7,8]$$

is

$$(8 \cdot \mathsf{binOpConst}/p) + (p + 15 - 16/p) \cdot g + 2 \cdot l$$

This example shows that the hand calculation of the analysis is hard task even for a simple example and, therefore, automation of the analysis is important for practical use. The analysis might looks expensive even if it can be automated, notice, however, that the analysis cost of this example does not change even if input vector is replaced by a large sized vector. For example, the analysis of

$$\mathsf{map}\ (+9)\ [1,2,3,\ldots,80000]$$

changes the value 8 to 80000 in the calculation without changing the complexity of the analysis cost, while the run time cost of the program will be roughly 10000 times larger.

5.3 System Structure

The whole cost analysis system is divided into seven modules.

CostDefsBsp.hs: definitions of cost tuples;
CostTransBsp.hs: definitions for translation from VEC-BSP terms to MSIZE terms;
CostConstBsp.hs: definitions of constants used in the analysis;

CostTestBsp.hs: VEC-BSP codes for test programs;

CostParaBsp.hs: BSP parameters;

VecBspSugar.hs: syntax sugar for VEC-BSP programming;

TimingsBsp.hs: target file in which generated Haskell codes are stored.

The structure of VEC-BSP terms is given in the file CostTransBsp.hs. The user writes VEC-BSP programs in the module CostTestBsp.hs. CostTestBsp.hs also includes a do expression that contains a sequence of the operations including: transforming VEC-BSP programs into MSIZE programs using the definitions in other modules; Outputting MSIZE programs as Haskell programs into the output file TimingsBsp.hs; and applying the resulting cost functions to the BSP parameters, outputting the BSP cost. CostTransBsp.hs which is imported to CostTestBsp.hs also has the structure of MSIZE terms, the definitions for the cost function, which references cost tuple definitions in CostDefsBsp.hs, and a pretty-printer to convert MSIZE programs into Haskell programs. The definitions of cost tuples in CostDefsBsp.hs use the values of constants defined in CostConstBsp.hs and BSP parameters in CostParaBsp.hs. CostConstBsp.hs and CostParaBsp.hs are imported to CostDefsBsp.hs. VecBspSugar.hs includes definitions of syntax sugar for convenient VEC-BSP programming and is imported to CostTestBsp.hs. In the following sections in this chapter, we look at the definitions in CostTestBsp.hs (section 5.4), definitions of cost tuples in CostDefsBsp.hs (section 5.5), definitions of cost translation in CostTransBsp.hs (section 5.6), and definitions in other modules (section 5.7).

5.4 CostTestBsp.hs: Definitions of Cost Tests

The test VEC-BSP programs for which a user wants to calculate costs are given in the list `theTests` in CostTestBsp.hs.

```
{- list of Vec-BSP terms that are tested -}
theTests =  [
             TestProgram1,
             TestProgram2,
```

```
             TestProgram3
            ]
```

The structure of VEC-BSP terms is given in the file CostTransBsp.hs as the data type `VecBspTerm` as explained later in this chapter. CostTestBsp.hs also has definitions to manage the test procedure.

```
{- give the name of the file for generated Haskell code -}
outputFile = "timingsBsp.hs"

{- testing procedure -}
genTests = do { putStr "\nGenerating code ... ";
                appendFile outputFile "import CostDefsBsp\n\n";
                appendFile outputFile ("run (_,_,_,f) = timeFunApp f"
                                       ++ (show paraBSP) ++ "\n\n");
                appendFile outputFile (codeGen theTests);
                putStr ("done!\n\n");
                putStr ("First, load " ++ outputFile ++
                                         "into Hugs\n\n");
                putStr ("Then, to get the timing for the i'th entry
                          in the list of VecBsp terms,\n");
                putStr ("enter \"run termi\" at the Hugs prompt\n")
              }
```

After CostTestBsp.hs is loaded to Hugs (the Haskell system), the user enters `genTests` at Hugs prompt which performs: displaying `Generating code ...` on the screen; writing `import CostDefsBsp` in timingsBsp.hs; writing `run(_,_,_,f)=timeFunApp f paraBSP` in TimingsBsp.hs; performing `codeGen theTests` and then writing the result, that is translated MSIZE terms, into TimingsBsp.hs, where `codeGen` is a code generator which is defined in CostTransBsp.hs as explained later in this chapter; displaying `done!` on the screen; displaying `First, load TimingsBsp.hs into Hugs` on the screen; displaying `Then, to get the timing for the i'th entry in the list of VecBsp terms` on the screen; and displaying `enter 'run termi' at` `the Hugs prompt` on the screen. Following the given instructions, the user loads TimingsBsp.hs and then enters `run termi` to evaluate `run (_,_,_,f) = timeFunApp f paraBSP`, that is, to evaluate the MSIZE term and then apply the resulting time function to the BSP parameters, getting the calculated cost of `TestProgrami`.

5.5 CostDefsBsp.hs: Definitions of Cost Tuples

CostDefsBsp.hs has definitions of cost tuples for constructors. First, some algebraic types of the data which are used in the cost tuple definitions are defined.

The algebraic type for time functions from the BSP parameters to time, that is `Timefun` type is defined by

```
data TimeFun =    VarTimeFun ((Int, Float, Float) -> Float)
                | ConstTimeFun Float
```

in which the CostTimeFun case is introduced to improve analysis speed for the case in which the time function is a constant function. The algebraic type for shapes, that is `Shape` type is defined by

```
data Shape =     Size Int
               | Tuple2 (Shape, Shape)
               | Tuple3 (Shape, Shape, Shape)
               | Tuple4 (Shape, Shape, Shape, Shape)
               | Pair2 (Size Int, Shape)
               | Fun (Shape -> (Shape, Size Int, TimeFun))
```

in which we define the tuple data types which have up to four components.

Next, some auxiliary functions which are used in the cost tuple definitions are defined.

`timeFunApp` takes a time function and BSP parameters and returns BSP cost. Its definition is

```
timeFunApp :: TimeFun -> (Integer,Float,Float) -> Float
```

```
timeFunApp (ConstTimeFun n) _ = n
timeFunApp (VarTimeFun f) (p,g,l) = f (p,g,l)
```

`funApp` takes the shape of a function and the shape of an argument, and returns an application tuple, whose components are the result shape, application pattern and application cost. Its definition is

```
funApp :: Shape -> Shape -> (Shape,Integer,TimeFun)
funApp (Fun f) x = f x
```

`size`, which takes the shape of a data item and returns its size is defined by

```
size :: Shape -> Integer
size (Tuple2 (x1,x2)) = size x1 + size x2
size (Tuple3 (x1,x2,x3)) = size x1 + size x2 + size x3
size (Tuple4 (x1,x2,x3,x4)) = size x1 + size x2 + size x3 + size x4
size (Pair2 (x,y)) = x * size y
size (Size n) = n
```

`bspapp` takes the cost tuple of a function term and the cost tuple of an argument term and returns the cost tuple of the result term. The definition is referenced in the definition of the cost function for application term `App t t'` in CostTransBsp.hs as shown in the next section. It is defined by

```
bspapp::(Shape, Integer, Integer, TimeFun)
         ->(Shape, Integer, Integer, TimeFun)
         ->(Shape, Integer, Integer, TimeFun)


bspapp (Fun f, s, d, h) (t, s', d', h') =
 let (v, ap, g') = f t
 in (v , (if (constFunEq g' (ConstTimeFun 0)) then (s + s')
          else size v), ap,
           VarTimeFun (\(p,g,l) ->
            ((timeFunApp (timePlus (timePlus h h') g') (p,g,l) +
              (if (p == 1) then 0
```

```
               else (if ap == 0 then 0
                     else (if d' == 1 then
                            ((fromIntegral s * fromIntegral(p-1)
                            - (fromIntegral s'
                            * (fromIntegral(p-1)/ fromIntegral p)))
                            * g - l)
                           else ((fromIntegral s * fromIntegral(p-1)
                           + (fromIntegral s')
                           * (fromIntegral(p-1)/fromIntegral p))
                           * g + l)))) )))
```

where, `constFunEq`, which checks the equality of two constant time functions is defined by

```
constFunEq :: TimeFun -> TimeFun -> Bool
constFunEq (ConstTimeFun a) (ConstTimeFun b) =
                           if (a == b) then True
                           else False
constFunEq (VarTimeFun a) _ = False
```

and `timePlus`, which adds two time functions is defined by

```
timePlus :: TimeFun -> TimeFun -> TimeFun
timePlus (ConstTimeFun n) (ConstTimeFun m) = ConstTimeFun (n + m)
timePlus (ConstTimeFun n) (VarTimeFun h) =
        VarTimeFun (\(p,g,l) -> n + (h (p,g,l)))
timePlus (VarTimeFun f) (ConstTimeFun m) =
        VarTimeFun (\(p,g,l) -> (f (p,g,l)) + m)
timePlus (VarTimeFun f) (VarTimeFun h) =
        VarTimeFun (\(p,g,l) -> (f (p,g,l)) + (h (p,g,l)))
```

A cost tuple is defined for each VEC-BSP constructor, which is referenced by the `cost` function for the constructor in CostTransBsp.hs as shown in the next section. As examples, we here give the definitions of cost tuples for an atomic datum constant, a binary datum operation, length, and map.

```
constCost :: (Shape, Integer, Integer, TimeFun)
constCost = (Size 1, Size 1, Size 0, ConstTimeFun 0)

primBinOpCost :: (Shape, Integer, Integer, TimeFun)
primBinOpCost = (Fun (\x ->
                   (Fun (\y ->
                      (Size 1, Size 0, ConstTimeFun primBinOpConst)),
                       Size 0, ConstTimeFun 0)),
                   Size 0, Size 0, ConstTimeFun 0)

lengthCost :: (Shape, Integer, Integer, TimeFun)
lengthCost = (Fun (\x -> (Size 1, Size 0, ConstTimeFun lengthConst)),
              Size 0, Size 0, ConstTimeFun 0)

mapCost :: (Shape, Integer, Integer, TimeFun)
mapCost = (Fun (\f -> (Fun (shapeMap f), Size 0, ConstTimeFun 0)),
           Size 0, Size 0, ConstTimeFun 0)

shapeMap :: Shape -> (Shape -> (Shape, Integer, TimeFun))
shapeMap f (Pair2 (len, eshp)) =
 let shps = Pair2 (len, pi1From3(funApp f eshp))
 in (shps, Size 1,
   VarTimeFun (\(p,g,l) ->
    (timeFunApp(tensorMult len (pi3From3(funApp f eshp)))(p,g,l) +
    (if (p==1) then 0
     else (fromIntegral(size(sndPair2(shps))) *
          fromIntegral(ceiling(fromIntegral(len)/fromIntegral p)) *
          fromIntegral(p - 1) * g + l))))
```

where the functions such as `pi1Tuple3`, which take the ith components of a three-components-tuple are defined by

```
pi1Tuple3 (Tuple3' (t,_,_)) = t
pi2Tuple3 (Tuple3' (_,t,_)) = t
pi3Tuple3 (Tuple3' (_,_,t)) = t
```

and the functions such as `sndPair2`, which take a component from a size-cost pair are defined by

```
fstPair2 (Pair2 (t,_)) = t
sndPair2 (Pair2 (_,t)) = t
```

and `tensorMult`, which takes the number of tasks multiplied and the time function of the single task, and returns the time function of the multiple tasks using p processors is

```
tensorMult :: Integer -> TimeFun -> TimeFun
tensorMult 0 _ = ConstTimeFun 0
tensorMult n (VarTimeFun f) =
 VarTimeFun (\(p,g,l) ->
  ((fromIntegral (ceiling ((fromIntegral n) / (fromIntegral p)))) *
  (f (1,g,l))))
tensorMult _ (ConstTimeFun 0) = ConstTimeFun 0
tensorMult n (ConstTimeFun m) =
 VarTimeFun (\(p,g,l) ->
 (fromIntegral (ceiling ((fromIntegral n) / (fromIntegral p)))) * m)
```

5.6 CostTransBsp.hs: Definitions for Cost Translation

CostTransBsp.hs has definitions to translate VEC-BSP terms to MSIZE terms. First, the algebraic type for constructors, and then the algebraic type for VEC-BSP terms and Msize terms are defined.

The algebraic type for term constructors, that is `TCons` type is defined by

```
data TCons = Hd | Tl | Pi1From2 | Pi2From2 | Pi1From3 | Pi2From3 |
```

```
            Pi3From3 | Pi1From4 |Pi2From4 | Pi3From4 |
            Pi4From4 | Length | Pair | Tuple2 | Tuple3 | Tuple4 |
            Scan | Map | Pairmap | Equal | Geq | And | Entry |
            Plus | Mult | Minus | Div | Mod | Fold | Iter |
            Cproduct | Min | Max
```

The algebraic type for VEC-BSP terms, that is `VecBspTerm` type is defined by

```
data VecBspTerm = Combr TCons | Const Integer | BoolConst Bool |
                  Var String | Abs String VecTerm |
                  App VecTerm VecBspTerm |
                  If VecTerm VecBspTerm VecBspTerm |
                  Ifs VecBspTerm VecBspTerm VecBspTerm
```

The algebraic type for MSIZE terms, that is `MsizeTerm` type is defined by

```
data MsizeTerm = MszCombr TCons | MszConst Integer |
                 MszBoolConst Bool | MszVar String |
                 MszAbs String MsizeTerm | MszConstFun MsizeTerm |
                 Msz Integer | MszApp MsizeTerm MsizeTerm |
                 MszIfs MsizeTerm MsizeTerm MsizeTerm
```

The MSIZE terms are converted to Haskell expressions by a pretty-printer and output in the file TimingBsp.hs. First, we make MsizeTerm an instance of the class `Show`. The instance declaration is

```
instance Show MsizeTerm where
    showsPrec p = showMsizeTerm
showMsizeTerm t = shows (getMsizeTermString t)
```

The names of the constructors are converted to strings by getCombrString. For example,

```
getCombrString Hd = "head"
getCombrString Tl = "tail"
```

```
getCombrString Fst = "fst"
getCombrString Snd = "snd"
getCombrString Entry = "entry"
getCombrString Map = "map"
getCombrString And = "and"
getCombrString Equal = "=="
getCombrString Geq = ">="
getCombrString Plus = "+"
getCombrString Mult = "*"
getCombrString Minus = "-"
getCombrString Div = "div"
getCombrString Mod = "mod"
```

MSIZE terms are converted to strings by getMsizeTermString.

```
getMsizeTermString (MszApp (MszApp (MszCombr Pair) t) t') =
        "(" ++  (getMsizeTermString t) ++ "," ++
                (getMsizeTermString t') ++ ")"
getMsizeTermString (MszApp (MszApp (MszCombr Tuple2) t) t') =
        "(" ++ (getMsizeTermString t) ++ "," ++
               (getMsizeTermString t') ++ ")"
getMsizeTermString (MszApp (MszApp (MszApp (MszCombr Tuple3)
        t) t') t'') = "(" ++ (getMsizeTermString t) ++ "," ++
                      (getMsizeTermString t') ++ "," ++
                      (getMsizeTermString t'') ++ ")"
getMsizeTermString (MszApp (MszApp (MszApp (MszApp
  (MszCombr Tuple4) t) t') t'') t''') =
                                  "(" ++
                                  (getMsizeTermString t) ++
                                  "," ++
                                  (getMsizeTermString t') ++
                                   "," ++
                                  (getMsizeTermString t'') ++
```

```
                                   "," ++
                                   (getMsizeTermString t''') ++
                                   ")"
getMsizeTermString (MszApp (MszApp (MszCombr Equal) t) t') =
                   showAsInfix t t' "=="
getMsizeTermString (MszApp (MszApp (MszCombr Plus) t) t') =
                   showAsInfix t t' "+"
getMsizeTermString (MszApp (MszApp (MszCombr Minus) t) t') =
                   showAsInfix t t' "-"
getMsizeTermString (MszApp (MszApp (MszCombr Mult) t) t') =
                   showAsInfix t t' "*"
getMsizeTermString (MszApp (MszApp (MszCombr Div) t) t') =
                   showAsInfix t t' ""
getMsizeTermString (MszApp (MszApp (MszCombr Mod) t) t') =
                   showAsInfix t t' "`mod`"
getMsizeTermString (MszApp (MszApp (MszCombr And) t) t') =
                   "and [" ++
                   (getMsizeTermString t) ++
                   "," ++
                   (getMsizeTermString t') ++
                   "]"
getMsizeTermString (MszBoolConst b) = show b
getMsizeTermString (MszCombr c) = getCombrString c
getMsizeTermString (MszVar x) = x
getMsizeTermString (MszConst n) = show n
getMsizeTermString (MszVectorConst n) = show n
getMsizeTermString (MszAbs x t) = "(Fun (\\" ++ x ++ " -> "
                    ++ (getMsizeTermString t) ++ "))"
getMsizeTermString (MszConstFun t) = "(ConstTimeFun " ++
                    (getMsizeTermString t) ++ ")"
getMsizeTermString (MszApp t t') = "(" ++
```

```
                                    (getMsizeTermString t) ++
                                    " " ++
                                    (getMsizeTermString t') ++ ")"
getMsizeTermString (MszIfs t t' t'') = "(if " ++
                                        (getMsizeTermString t)
                                        ++ " then " ++
                                        (getMsizeTermString t') ++
                                        " else " ++
                                        (getMsizeTermString t'') ++
                                        ")"
getMsizeTermString (Msz n) = (show n)
showAsInfix t t' s = "(" ++ (getMsizeTermString t) ++ " " ++ s ++
                     " " ++ (getMsizeTermString t') ++ ")"
```

The function `cost` takes a VEC-BSP term and returns an MSIZE term. Many of the definitions of `cost` for constants and combinators in VEC-BSP reference the corresponding cost tuple definitions in CostDefsBsp.hs. Some examples are

```
cost :: VecBspTerm -> MsizeTerm
cost (Const _) = (MszVar "constCost")
cost (VectorConst v) = app(MszVar "vectorCost") (MszVectorConst v)
cost (Combr Plus) = (MszVar "primBinOpCost")
cost (Combr Minus) = (MszVar "primBinOpCost")
cost (Combr Mult) = (MszVar "primBinOpCost")
cost (Combr Div) = (MszVar "primBinOpCost")
cost (Combr Max) = (MszVar "primBinOpCost")
cost (Combr Hd) = (MszVar "headCost")
cost (Combr Map) = (MszVar "mapCost")
cost (Combr Fold) = (MszVar "foldCost")
```

`cost` for an application term references the definition of bspapp operation.

```
cost (App t t') = MszApp(MszApp(MszVar "bsppapp")(cost t))(cost t')
```

The Haskell code corresponding to an MSIZE program is generated by the following code generator, `codeGen`.

```
noDblSlash [] = []
noDblSlash (h:'\\':'\\':tl) = h:'\\':(noDblSlash tl)
noDblSlash (h:tl) = h:(noDblSlash tl)
vec2HaskellCost t =
          let lst = (show (cost t))
            in noDblSlash (tail (take ((length lst) - 1) lst))
codeGen lst = codeGenLoop 1 lst
codeGenLoop _ [] = ""
codeGenLoop n (h:t) = "term"  ++ " = " ++
         (vec2HaskellCost h) ++ "\n\n" ++ (codeGenLoop (n + 1) t)
```

5.7 Other Modules

5.7.1 CostParaBsp.hs: Definitions for BSP parameters

CostParaBsp.hs includes definitions of BSP parameters, which are obtained from running a benchmark program on the target architecture. For example,

```
paraBSP :: (Integer, Float, Float)
paraBSP = (8,1.6,67150)

p :: Integer
p = pi1From3(paraBSP)
g :: Float
g = pi2From3(paraBSP)
l :: Float
l = pi2From3(paraBSP)
```

5.7.2 CostConstBsp.hs: Definitions for Constants

The constants used in the definitions of cost tuples in CostDefsBsp.hs are defined in CostConstBsp.hs. In current version, primBinOpConst is defined as 1 and the others as 0.

```
primBinOpConst = 1 :: Float
```

5.7.3 VecBspSugar.hs: Syntax Sugar

VecBspSugar.hs includes syntax sugar for convenience of VEC-BSP programming. For example,

```
ct n = Const n
v x = Var x
vabs x t = Abs x t
vapp f t = App f t
vapp2 f t t' = vapp (vapp f t) t'
vapp3 f t t' t'' = vapp (vapp2 f t t') t''
vapp4 f t t' t'' t''' = vapp (vapp3 f t t' t'') t'''
vmap f v = vapp2 (Combr Map) f v
fold f v = vapp2 (Combr Fold) f v
```

For example, the straightforward VEC-BSP expression for map $(\times 2)\ v$ is

```
App (App (Combr Map) (App (Combr Mult) (Const 2))) v
```

but it can be expressed with sugaring as

```
vmap (vapp (Combr Mult) (ct 2)) v
```

5.8 Chapter Conclusion

This chapter outlined the Haskell implementation of our cost analysis, which is based on the existing PRAM calculator. The main differences with the PRAM calculator reflect those of the analysis itself, that is: the new components of size, application pattern and data pattern; the cost function is a function of BSP parameters; the `bspapp` which captures the communication and synchronisation cost as well as the computation cost; a set of skeletons each of which has the application cost that is based on the BSP cost model. One more technical change is that we avoided the analysis of the input vector itself, which is done in the PRAM calculator using *singleton* and *cons* operators (and which would be possible for our analysis as well), by giving the input shape directly. This saves analysis cost and brings the desirable property that the analysis cost is independent of input vector size.

Chapter 6

Experiments: Comparing Different Algorithms

In this chapter we describe our experimental framework for automatic cost prediction. We consider different algorithms for simple example problems, namely, matrix-vector multiplication and maximum segment sum, and show that our method allows detailed consideration of constant factors across a range of problem sizes which would be difficult in a pencil-and-paper analysis. We then report on the results of experiments which compare our predictions with the performance of real programs.

6.1 Matrix Multiplication

The first example problem is a matrix vector multiplication Mv, where M is an $m \times n$ matrix and v is an n element vector. We consider different two algorithms, contrasting the analysis of their efficiency by traditional, intuitive methods with that achieved by our cost calculator. The communication optimisation described in chapter 3 is applicable in the second algorithm. The first algorithm is expressed in VEC-BSP as:

$$\mathsf{map}\ (\lambda y.\,(\lambda x.(\mathsf{fold} + (\mathsf{pair\_map}(\cdot)(\mathsf{pair}\ y\ x))))\ v)\ M \qquad (1)$$

where, $v = \mathsf{dummyvec}\,(n,1)$ and $M = \mathsf{dummyvec}\,(m,(n,1))$ for analysis purposes. Its BSP implementation based on our strategy is:

1. the elements of v in the master processor are broadcast to the p processors and M's contents, consisting of mn integers in the master processor, are scattered to the p processors in vector-block-wise manner;

2. synchronisation;

3. each processor computes the elementwise multiplication of v and each distributed vector;

4. the results of 3 are folded with addition on each processor;

5. the local result on each processor is gathered to the master processor;

6. synchronisation.

Notice that the function $\lambda x.(\mathsf{fold} + (\mathsf{pair\_map}\,(\cdot)\,(\mathsf{pair}\,v\,x)))$ takes a vector and returns its inner product with v. The parallel structure of the algorithm is illustrated in a diagram in figure 6.1. An intuitive BSP cost analysis is made by counting the number

Figure 6.1: Parallel structure of algorithm (1)

of operations and message size by hand. The resulting computation cost is mn/p for integer multiplications and $m(n-1)/p$ for integer additions. Communication cost is $n(p-1)g$ to broadcast n integer elements of v, $mn((p-1)/p)g$ to scatter mn integer elements of M and $(m/p)(p-1)g$ to gather local results, so the overall communication cost is $((mn+m+np)(p-1)/p)g$. There are two synchronisations at a cost of $2l$.

The second algorithm is expressed in VEC-BSP as:

$$\mathsf{fold}\,(\lambda x\,y.(\mathsf{pair\_map} + (\mathsf{pair}\,x\,y)))\,(\mathsf{pair\_map}\,(\lambda x\,y.\mathsf{map}\,(\lambda z.(y \cdot z))\,x)\,(\mathsf{pair}\,L\,v)) \quad (2)$$

where, $L = M^t = \mathsf{dummyvec}\,(n,(m,1))$ and $v = \mathsf{dummyvec}\,(n,1)$ as before. The implementation of this skeletal program has two parallel phases.

- pair_map phase:

  1. the contents of L and v in the master processor are scattered to the p processors;

  2. synchronisation;

  3. each processor computes the element-wise application of $\lambda x\,y.\mathsf{map}\,(\lambda z.(y \cdot z))\,x$ to each distributed integers from v and each corresponding distributed vectors from L. (The effect of this is that each element of the ith vector of L is multiplied by the ith element of v.).

- fold phase:

  4. each processor computes the element-wise addition of all local vectors;

  5. the local result on each processor is gathered to the master processor;

  6. synchronisation;

  7. the master processor computes element-wise addition of the gathered vectors.

Notice that the communications implied by the gather step at the end of the pair_map phase and the broadcast-scatter step in the beginning of the fold phase can be optimised away by our analysis, leading directly to the computation step of the fold phase. Thus, our cost analysis does not count these communication costs. The parallel structure of this algorithm (2) is illustrated in a diagram in figure 6.2. An intuitive BSP

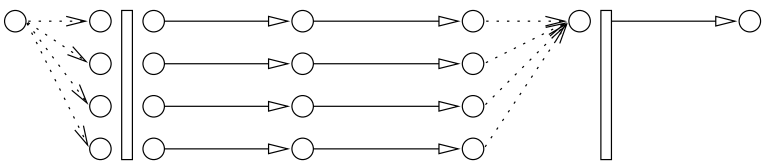
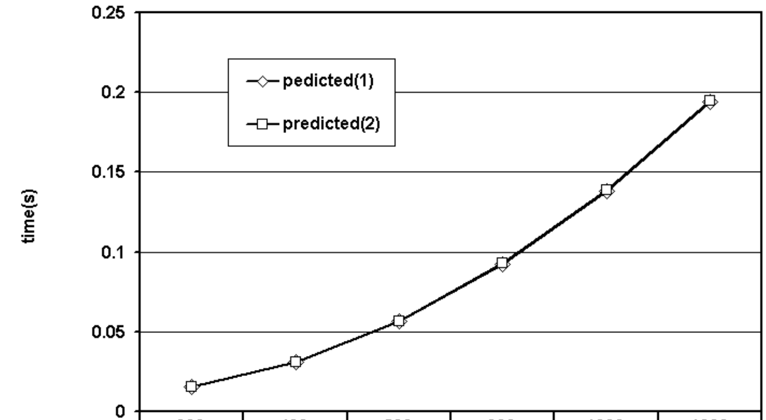


Figure 6.2: Parallel structure of algorithm (2)

cost analysis of (2) is made as follows. The computation cost is mn/p integer multiplications and $m(n/p-1)+m(p-1)$ integer additions. The communication cost is $(m+1)n((p-1)/p)g$ for scattering the elements of L and the elements of v, and $m(p-1)g$ for gathering local results in the fold application, so the overall communication cost is $((mn+n)((p-1)/p)+m(p-1))g$. The synchronisation cost is $2l$.

We now apply our cost calculator to the two algorithms. Our target system is an 8-processor Sun HPC 3500 UltraSPARC II machine hosted by the Edinburgh Parallel Computer Centre. BSP parameters obtained by running a benchmark program provided by Oxford BSPlib are $p=8$, $g=1.6$, $l=67150$. The binary operator constant is set at 1 and the total calculated cost in operations is converted into seconds by dividing by 13 million as directed by s, the benchmark returned factor which normalises l and g to the single processor computational speed. In this section we investigate the performance predicted by our cost calculator, and compare with a pencil-and-paper asymptotic analysis. Comparison of predicted and real execution costs is presented in section 6.3.



	200	400	600	800	1000	1200
pedicted(1)	0.0156	0.031022	0.056599	0.092329	0.138214	0.194252
predicted(2)	0.015694	0.031211	0.056882	0.092706	0.138685	0.194812

Figure 6.3: Prediction when m=n, p=8

First, as a concrete example we investigate the case in which $p = 8$ and M is square (i.e. $m = n$), with m varying. Our cost system automatically translates the source program to a Haskell program which computes BSP cost. Figure 6.3 shows the predicted result of varying m in increments of 200 up to 1200. We can see that the predicted BSP costs of the two programs are almost the same and that their time complexity seems to be $O(m^2)$. This concurs with the intuitive BSP analysis above. From the intuitive analysis, we can easily see that both algorithms have BSP cost complexity $O(m^2)$ when p is fixed. In computation cost, (2) needs $m(p+1/p-2)$ more additions than (1). These come from the use of parallel fold that has a phase in which only one processor is working, while (1) uses sequential fold in parallel map. Since the difference of the communication costs, (2)−(1) is $((m-n)(p-1)^2/p)g$, the communication costs are the same when $m = n$. Therefore, while the BSP cost complexity of both programs are $O(m^2)$, the actual difference of BSP cost, $m(p+1/p-2)$, has complexity of $O(m)$. This means that the difference is not significant when $m(=n)$ is large.

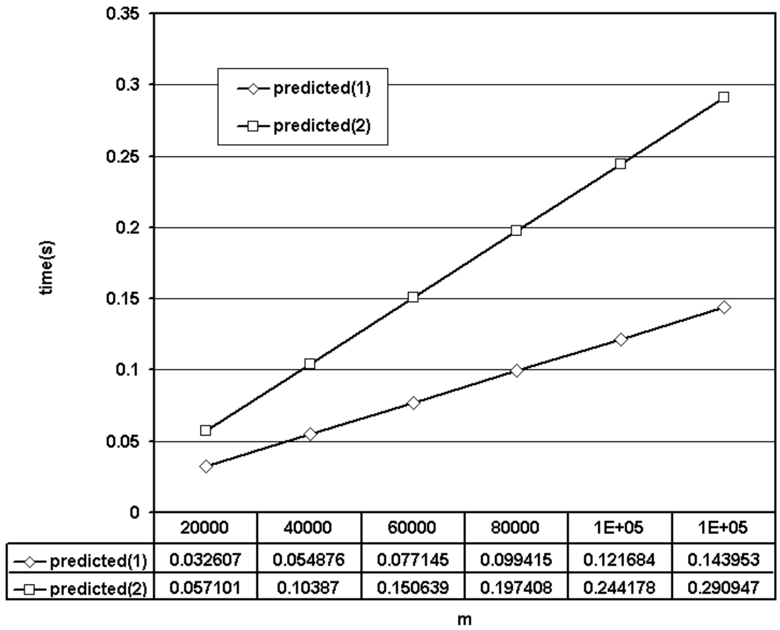
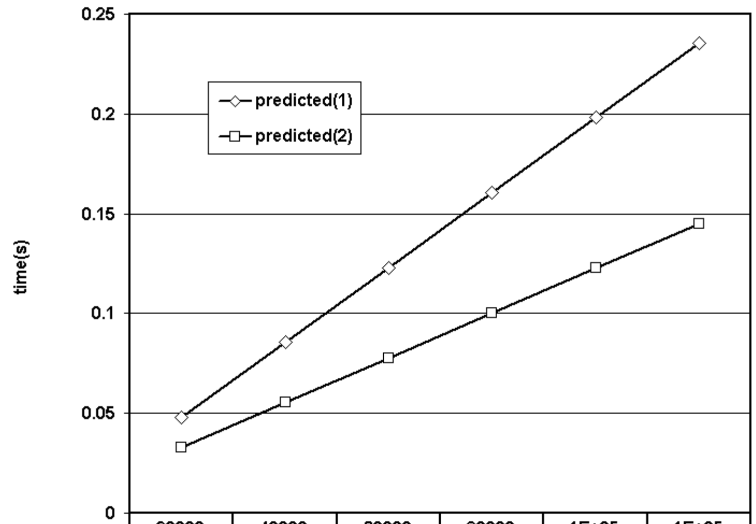


Figure 6.4: Prediction when n=8, p=8

Next we investigate the case in which n is fixed and m varies. Is there any significant difference in efficiency between (1) and (2)? Figure 6.4 shows the cost predicted by our calculator when n is fixed at 8 and m varies in increments of 20000 up to 120000. We can see that (1) is more efficient than (2). According to the intuitive analysis both algorithms have BSP cost complexity $O(m)$. Since the difference of computation costs (2)−(1), $m(p+1/p-2)$ and the difference of communication costs (2)−(1), $((m-n)(p-1)^2/p)g$ have complexity of $O(m)$, the overall difference of costs also has complexity of $O(m)$. This could be significant, and the results from figure 6.4 predict that this is indeed the case.

Finally we investigate the case when m is fixed and n is varied. Figure 6.5 shows the predicted results when m is fixed at 8 and n varies in increments of 20000 up to 120000. Now (2) is more efficient than (1). According to the intuitive analysis, both algorithms



	20000	40000	60000	80000	1E+05	1E+05
predicted(1)	0.04787	0.085408	0.122947	0.160485	0.198029	0.235562
predicted(2)	0.032803	0.055264	0.077726	0.100188	0.122649	0.145111

n

Figure 6.5: Prediction when m=8, p=8

have BSP cost complexity $O(n)$. The computation cost of (1) is less than that of (2) but the difference $m(p+1/p-2)$, is only constant. In contrast, the communication cost of (1) is more than that of (2) and the difference, $((n-m)(p-1)^2/p)g$, has complexity of $O(n)$. As before, this could be significant and the prediction of figure 6.5 again shows this to be the case.

6.2 Maximum Segment Sum Problem

As the second example problem, we take the *maximum segment sum* problem, which is often used as an example of BMF style algorithm derivation. The problem is to find the sum of the contiguous segment of a list whose members have the largest sum among all such segments. As an example, we have

$$mss[2,-4,2,-1,6,-3] = 7$$

We predict the costs of three different algorithms taken from papers of the field, that is Bird's algorithm [11], Skillicorn and Cai's algorithm [78] and Cole's algorithm [26].

6.2.1 Three Different Algorithms

Bird's Algorithm

Bird derived a $O(n)$ sequential time algorithm from a $O(n^3)$ sequential time specification [11] by BMF style program calculation. It uses a sequential second-order function *left accumulate* defined by

$$\oplus \not\!\!\!\twoheadrightarrow_e [a_1, a_2, \ldots, a_n] = [e, e \oplus a_1, \ldots, ((e \oplus a_1) \oplus a_2) \oplus \cdots \oplus a_n].$$

The algorithm is expressed concisely as

$$mss = \uparrow / \circ \odot \not\!\!\!\twoheadrightarrow_0$$

where $a \odot b = (a+b) \uparrow 0$.

Skillicorn and Cai's Algorithm

In [78], Skillicorn and Cai derived a parallel algorithm from the same specification. It has complexity of $O(\log n)$ under the condition that n processors are available. It uses the *recur-prefix* operation defined in section 2.3.3. The algorithm is expressed concisely as

$$mss = \uparrow / \circ + /\!/_0 \uparrow$$

Cole's Algorithm

The mss problem is also used by Cole [26] to explain the idea of constructing a homomorphism from a *near homomorphism* by the use of a tuple with extra functions. It has $O(\log n)$ parallel time complexity under the condition that $O(n/\log n)$ processors are available. The algorithm is expressed as

$$mss = tieup \circ (\oplus/) \circ (f*)$$

where

$$\begin{aligned}
&f x = (x \uparrow 0, x \uparrow 0, x \uparrow 0, x),\\
&(x_1, x_2, x_3, x_4) \oplus (y_1, y_2, y_3, y_4)\\
&= (x_1 \uparrow y_1 \uparrow (x_3 + y_2), x_2 \uparrow (x_4 + y_2), (x_3 + y_4) \uparrow y_3 + y_4) \uparrow y_3, x_4 + y_4),\\
&tieup(a, -, -, -) = a
\end{aligned}$$

Note that both Cole's and Skillicorn and Cai's cost analysis assumes that the number of processors can increase infinitely according to the increase of the input data size. They also assumes that the data is already distributed across the processors. Our analysis will fix the number of available processors of the target machine as one of BSP parameters and will charge for the initial distribution cost explicitly (so their complexities will be $O(n)$ at best).

6.2.2 VEC-BSP Programs of the mss Problem

Our strategy to write a VEC-BSP program from a BMF expression is:

- If a BMF function can be directly expressed by some VEC-BSP predefined functions, it is replaced by the VEC-BSP terms.
- If a sequential BMF function cannot be expressed by any predefined functions a new operator is added and its cost function is defined.
- All compositions of BMF functions are expressed as corresponding application terms in VEC-BSP.

Bird's Algorithm

Expressing Bird's algorithm as a VEC-BSP program is straightforward. We need only to add the left-accumulate operation laccum and to define its cost function.

$$\begin{aligned} cost(\mathsf{laccum}) \quad = \quad & \langle \lambda \oplus . \langle \lambda x. \langle (\mathsf{t\_len}\, x + 1, \mathsf{t\_eshp}\, x), \mathsf{SEQ}, \\ & \mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\, x))(\mathsf{t\_eshp}\, x)) \cdot \mathsf{t\_len}\, x \rangle, \mathsf{SEQ}, 0 \rangle, 0, \mathsf{SEQ}, 0 \rangle \end{aligned}$$

Using laccum, Bird's algorithm is expressed in VEC-BSP as:

$$mss\, v = \mathsf{fold}\,(\uparrow)\,(\mathsf{laccum}\,(\lambda xy.((x+y) \uparrow 0))\, v)$$

Its implementation is that the master processor computes the left accumulation of v with binary operator $\lambda xy.((x+y) \uparrow 0)$, and then the result is folded in the master processor with the maximum operator. The cost of the left accumulation is n additions and n maximum operations. The cost of the fold is $n-1$ maximum operations. The total cost is n additions and $2n-1$ maximum operations, so its overall time complexity is $O(n)$.

Skillicorn and Cai's Algorithm

Since the recur-prefix can be expressed as a prefix following [77]

$$\otimes /\!/_{id_\otimes} \oplus [a_1, \ldots, a_n] = [id_\otimes] +\!\!+ (\oplus) * (\oslash /\!/ ([a_1, \ldots a_n] \curlyvee [id_\otimes, \ldots, id_\otimes]))$$

where $\curlyvee$ is *zip* function and

$$(a,b) \oslash (c,d) = (a \otimes c, b \otimes c \oplus d)$$

Skillicorn and Cai's algorithm is expressed by using scan in VEC-BSP as

$$mss = \mathsf{fold}\,(\uparrow)\,(\mathsf{shiftright}(0)\,(\mathsf{map}\,(\uparrow)\,(\mathsf{scan}\,(\oslash')\,(\mathsf{map}\,(\mathsf{pair}\, 0)\, v))))$$

where

$$\oslash' = \lambda x\, y.\, \mathsf{pair}\,(\mathsf{fst}\, x + \mathsf{fst}\, y)((\mathsf{snd}\, x + \mathsf{fst}\, y) \uparrow 0)$$

and $\mathsf{shiftright}\,(0)$ rotates the entire list right one place, moving a single element from each processor $i\,(p \leq p-1)$ to the processor $i+1$ and inserting 0 at the left end in the master processor. Its cost function is

$$\begin{aligned} cost\,(\mathsf{shiftright}) &= \langle \lambda e.\langle \lambda x.\langle (\,\mathsf{t\_len}\,(x)+1, \mathsf{t\_eshp}\,x), \mathsf{SEQ}, \mathsf{size}\,(\mathsf{t\_eshp}\,x)\cdot g\rangle, \\ &\qquad \mathsf{SEQ}, 0\rangle, 0, \mathsf{SEQ}, 0\rangle \end{aligned}$$

The BSP implementation of the algorithm based on our implementation strategy is:

- $\mathsf{map}\,(\mathsf{pair}\,0)$ phase:
  - **1.** the elements of v in the master processor are scattered to the p processors;
  - **2.** synchronisation;
  - **3.** each processor applies $(\mathsf{pair}\,0)$ to each distributed element.
- $\mathsf{scan}\,(\oslash')$ phase:
  - **4.** each processor computes the local scan with $\oslash'$ for the local results of 3;
  - **5.** the final value of the each local scan is scanned in parallel across the processors in the tree-structured way;
  - **6.** the result of the global scan in processor $i\,(p \leq p-1)$ is sent to processor $i+1$;
  - **7.** synchronisation;
  - **8.** each processor applies $\oslash'$ to the pairs of the pair received in 6 and each pair of the results of 4.
- $\mathsf{map}\,(\uparrow)$ phase:
  - **9.** each processor takes the maximum element of each pair of the result of 8.
- $\mathsf{shiftright}\,(0)$ phase:
  - **10.** $\mathsf{shiftright}\,(0)$ rotates the entire list right one place, moving a single element from each processor to the next and inserting 0 at the left end.
- $\mathsf{fold}\,(\uparrow)$ phase:

11. each processor folds the local results of 10 with the maximum operation;

12. the local result of 11 in each processor is gathered to the master processor;

13. synchronisation;

14. the master processor folds the gathered results with the maximum operation.

The BSP cost of each phase is as follows. $\mathsf{map}\,(\mathsf{pair}\,0)$ phase: the computation cost $\frac{n}{p}$ pair operations, communication cost $n(p-1)g$ and synchronisation cost l. $\mathsf{scan}\,(\oslash')$ phase: the computation cost $\frac{2n}{p}-1$ $\oslash'$ operations each of which has 2 additions and 1 maximum operation, the communication cost $(\log p+1)g$ and the synchronisation cost $(\log p+2)\,l$. $\mathsf{shiftright}\,(0)$ phase: the communication cost g and the synchronisation cost l. $\mathsf{map}\,(\uparrow)$ phase: $\frac{n}{p}$ maximum operations. $\mathsf{fold}\,(\uparrow)$ phase: $\frac{n}{p}+p-2$ maximum operations. We can see that the overall complexity of Skillicorn's algorithm is $O(n)$ in our cost models when p is fixed.

Cole's Algorithm

In order to deal with algorithms to compute a homomorphism with a tuple, we introduced a tuple data structure as an extension of pair. Cole's algorithm can be expressed in VEC-BSP as:

$$mss\,x = \pi_1\,(\mathsf{fold}\oplus(\mathsf{map}\,f\,v))$$

where

$$\begin{aligned}
f &= \lambda x.\langle x\uparrow 0, x\uparrow 0, x\uparrow 0, x\rangle \\
\oplus &= \lambda x.y.\langle \pi_1 x\uparrow \pi_1 y\uparrow(\pi_3 x+\pi_2 y), (\pi_2 x\uparrow \pi_4 x)+\pi_2 y, \\
&\quad (\pi_3 x+\pi_4 y)\uparrow \pi_3 y, \pi_4 x+\pi_4 y\rangle
\end{aligned}$$

Its BSP implementation based on our strategy is

- map phase:

1. the elements of v in the master processor are scattered to the p processors;

2. synchronisation;

3. each processor applies f to each scattered element.

- fold$\oplus$ phase:

4. each processor computes the sequential fold of the result of 3 with $\oplus$;

5. the local results in each processor are gathered to the master processors;

6. synchronisation;

7. the master processor folds the gathered results with $\oplus$.

- π_1 phase:

8. π_1 is applied to the result of 7 in the master processor.

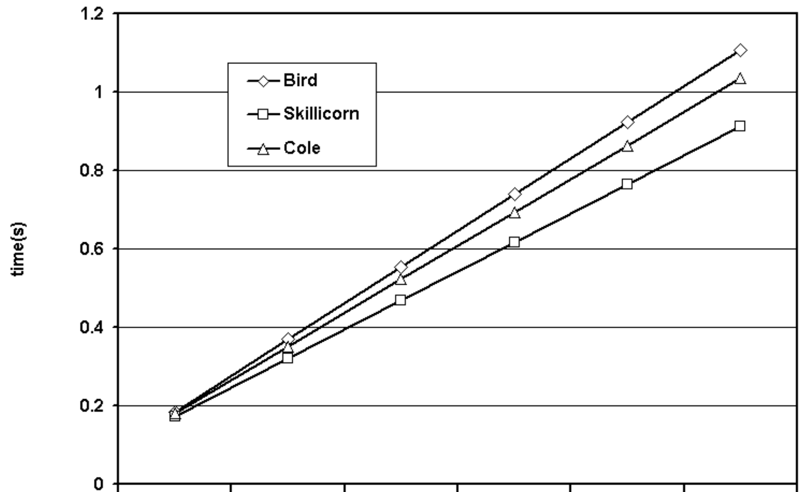
The BSP costs of the two phases are as follows. map phase: the computation cost $\frac{n}{p}$ f operations, each of which has three maximum operations, the communication cost $(n-1)g$ and the synchronisation cost l. fold phase: the computation cost $(\frac{n}{p}+p-2)$ $\oplus$ operations, each of which has four maximum operations and three additions, the communication cost $4(p-1)g$ and the synchronisation cost l. We can see that Cole's algorithm has cost complexity of $O(n)$ when p is fixed.

6.2.3 Predicted Results

Figure 6.6 shows the cost of each algorithm predicted by our calculator when the input list size n varies in increments of 800000 up to 4800000. We can see that all the predicted costs seem to have complexity $O(n)$ as we predicted by intuitive cost analysis. The predicted results also show that the efficiency of the three algorithms are almost the same in our cost model under the condition that only 8 processors are available.

6.2.4 Complexity of Cost Analysis

From the definition of bspapp in chapter 3, the analysis cost involved in the calculation of bspapp $\langle f, s, d, t\rangle\langle x, s', d', t'\rangle$ is: one calculation of $f\ x$; five projections (two t_apcost, two t_pattern and one t_shape); four additions; one data_sz; one comm_cost.



	8E+05	2E+06	2E+06	3E+06	4E+06	5E+06
Bird	0.184615	0.369231	0.553846	0.738462	0.923077	1.107692
Skillicorn	0.17354	0.321248	0.468957	0.616667	0.764374	0.912085
Cole	0.181115	0.351894	0.522672	0.693449	0.864231	1.035008

n

Figure 6.6: Predicted costs

The cost for data_sz is one conditional and one addition (or one projection). The cost for comm_cost is two conditionals, three multiplications, four subtractions and one division (in the most expensive case). Therefore, at most the total charged cost for bspcost except for calculation of $f\ x$ is: five projections, thirteen arithmetic operations and three conditionals. The calculation of $f\ x$ largely depends on f. When f is the shape of a conventional sequential function, its costs are a few arithmetic operations or projections to calculate the resulting shape. When f is a shape of a parallel function such as map t_f, its costs can depend on the argument function and involve the calculation of application costs according to the definitions in chapter 4 as well as the calculation of the resulting shape. For example, in the case of map t_f, the cost for calculation of the resulting shape is the cost for f (t_eshp (x)) and three projections. The cost for calculation of the application cost is twice of the cost for f (t_eshp (x)) and four projections, eight arithmetic operations and one size operation. Note that these calculations for shapes and application costs are calculated using the components of the argument shape, that is the length of the argument and the element shape, but these costs do not depend on the size of an input vector, as changing input size changes the first components of shapes. In conclusion, the analysis of shape and cost charges some projections and arithmetic operations proportional to the number of applications, but its cost does not depends on the input data size. Therfore, when the input vector becomes bigger, the analysis cost becomes relatively less significant.

6.3 Accuracy Tests

To test the accuracy of our static cost prediction against time on a real machine we hand compiled VEC-BSP programs into Oxford BSPlib following the compilation strategy proposed in 3.4 and implementation templates for each skeleton given in 4.2, trying to write natural straightforward C code for the computation part without any technical optimisation, and then ran them on an 8-processor Sun HPC 3500.

6.3.1 Matrix Multiplication

Following the same sequence of experiments as for the predictions, figure 6.7 and figure 6.8 plot the predicted BSP cost, the predicted computation cost and the real run time of each program when $m = n$ varying m in increments of 200 up to 1200. "precomp" represents the predicted computation cost obtained by setting BSP parameters g and l to 0 to show the impact of counting communication and synchronisation costs. Similarly, figure 6.9 and figure 6.10 plot times when n is fixed and m varies in increments of 20000 up to 120000, and figure 6.11 and figure 6.12 plot times when m is fixed and n varies in increments of 20000 up to 120000. In five out of six cases, real and predicted curves are very close. They also show that counting only computation costs for our assumed implementation model does not generate accurate absolute value prediction in these experiments.

Accuracy is inferior in the case of algorithm (2) when n is fixed (the upper two curves in figure 6.10). We note that when m is large in (2), the final sequential folding process performed by the master processor is dominant. Our calculator seems to underestimate that cost, suggesting that our modelling of sequential computation (rather than parallel interaction) is less successful for this algorithm.

6.3.2 Maximum Segment Sum

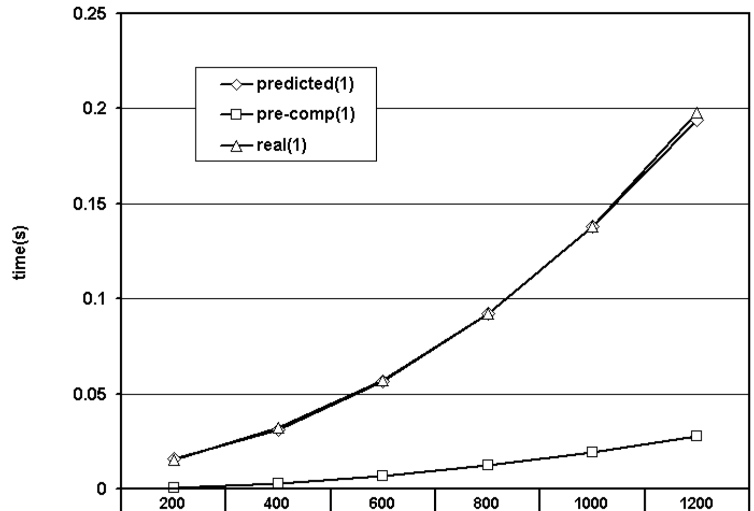
Figure 6.13 - figure 6.15 plot the predicted costs of the three mss algorithms and the real run times of the BSP programs in Oxford BSPlib on the Sun machine when the list size n varies in increments of 800000 up to 4800000. For both Skillicorn and Cai's and Cole's algorithms, figure 6.14 and figure 6.15 also plot the predicted computation costs by setting BSP parameters g and l to 0 for the same purpose as for the matrix multiplication examples. Notice that as Bird' algorithm is sequential, the predicted BSP cost is equal to the predicted computation cost.

For Bird's algorithm and Skillicorn and Cai's algorithm, the real cost and the predicted curves are very close. Accuracy is a little inferior in the case of Cole's algorithm. As most of the computation costs in Cole's algorithm are maximum operation costs, we infer that our cost calculator tends to overestimate maximum operations. Again, counting only computation costs for our assumed implementation model does not generate an accurate absolute value prediction for Skillicorn and Cai's and Cole's algorithms.

6.4 Chapter Conclusion

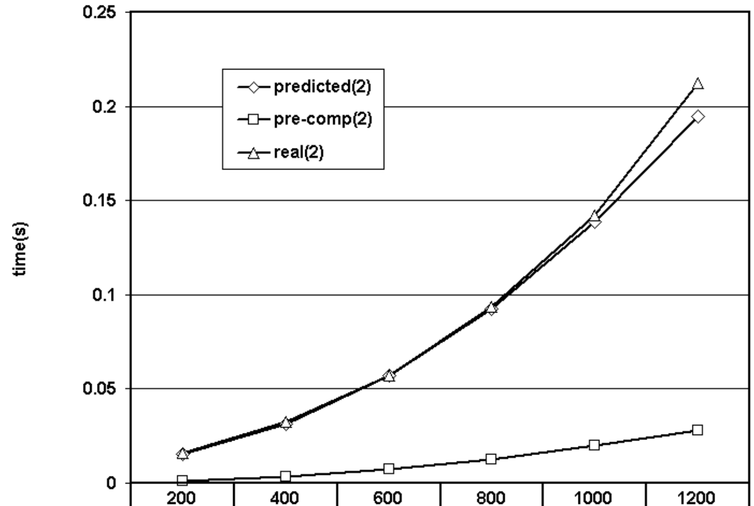
Ad-hoc analysis is a hard task even for a simple algorithm. Our cost calculator can automatically perform the analysis of any arbitrarily complex programs for arbitrary specified parameters, considering the effect of underlying message passing performance. This allows us to make detailed comparisons of algorithms which have the same intuitive asymptotic complexity.

The accuracy of our prediction is encouraging. In general, our accuracy also depends on how the g and l values experienced by the computation patterns and communication patterns used in an application program are matched by those in the benchmark program used to determine the BSP parameters (in other words how robust the BSP framework is itself). Although we used the benchmark program provided with BSPlib, developing a benchmark program more suitable for the computation and communication patterns used in our more restricted computational model should further improve accuracy.



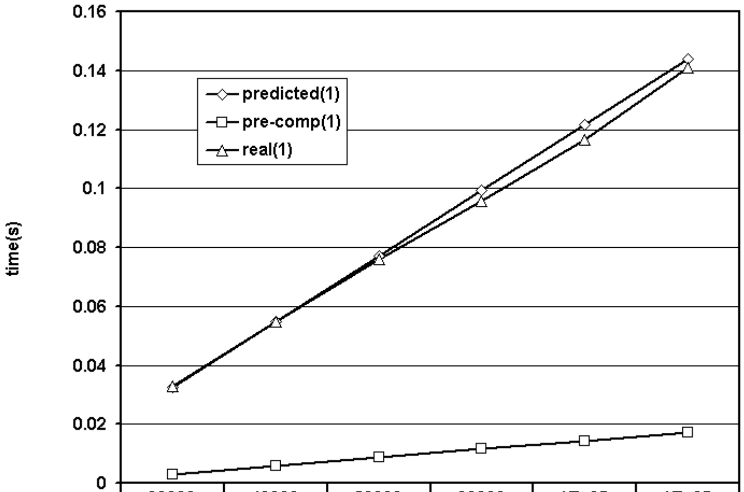
	200	400	600	800	1000	1200
predicted(1)	0.0156	0.031022	0.056599	0.092329	0.138214	0.194252
pre-comp(1)	0.000767	0.003073	0.006917	0.0123	0.019221	0.02768
real(1)	0.015484	0.032234	0.057117	0.092148	0.137751	0.198065

Figure 6.7: Accuracy of (1) when m=n, p=8



	200	400	600	800	1000	1200
predicted(2)	0.015694	0.031211	0.056882	0.092706	0.138685	0.194817
pre-comp(2)	0.000862	0.003262	0.0072	0.012677	0.019692	0.028246
real(2)	0.01611	0.032624	0.057037	0.093367	0.142252	0.212574

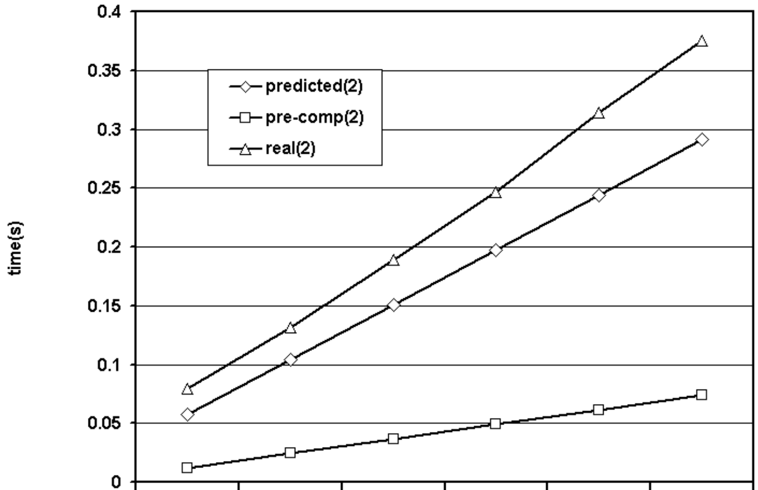
Figure 6.8: Accuracy of (2) when m=n, p=8



	20000	40000	60000	80000	1E+05	1E+05
predicted(1)	0.032607	0.054876	0.077145	0.099415	0.121684	0.143953
pre-comp(1)	0.002885	0.005769	0.008654	0.011538	0.014423	0.017308
real(1)	0.033047	0.054771	0.075999	0.09586	0.116596	0.140966

m

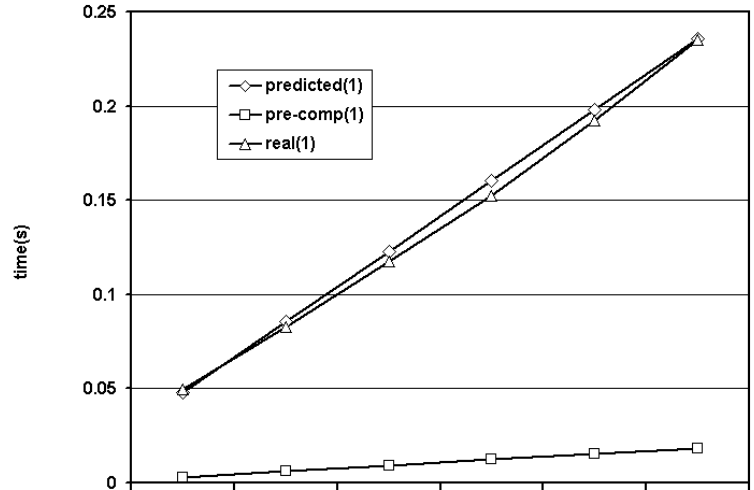
Figure 6.9: Accuracy of (1) when n=8, p=8



	20000	40000	60000	80000	1E+05	1E+05
predicted(2)	0.057101	0.10387	0.150639	0.197408	0.244178	0.290947
pre-comp(2)	0.012308	0.024615	0.036923	0.049231	0.061538	0.073846
real(2)	0.079412	0.131146	0.188744	0.246801	0.314281	0.375309

m

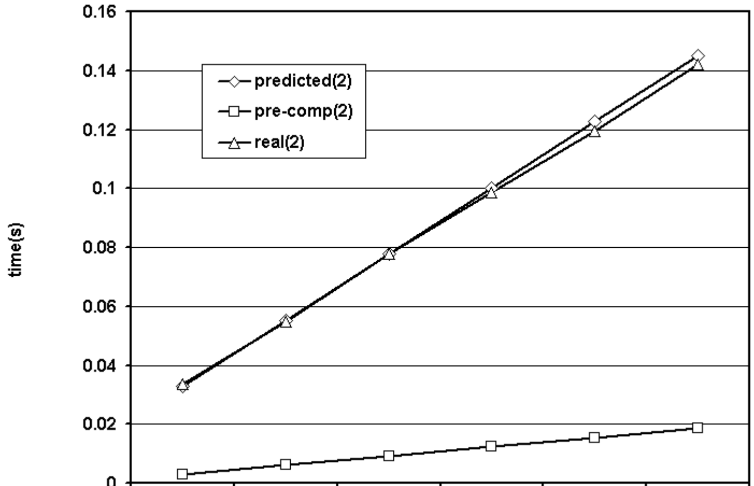
Figure 6.10: Accuracy of (2) when n=8, p=8



	20000	40000	60000	80000	1E+05	1E+05
predicted(1)	0.04787	0.085408	0.122947	0.160485	0.198029	0.235562
pre-comp(1)	0.003077	0.006154	0.00923	0.012308	0.015385	0.018461
real(1)	0.049433	0.082918	0.117705	0.152316	0.19209	0.235004

n

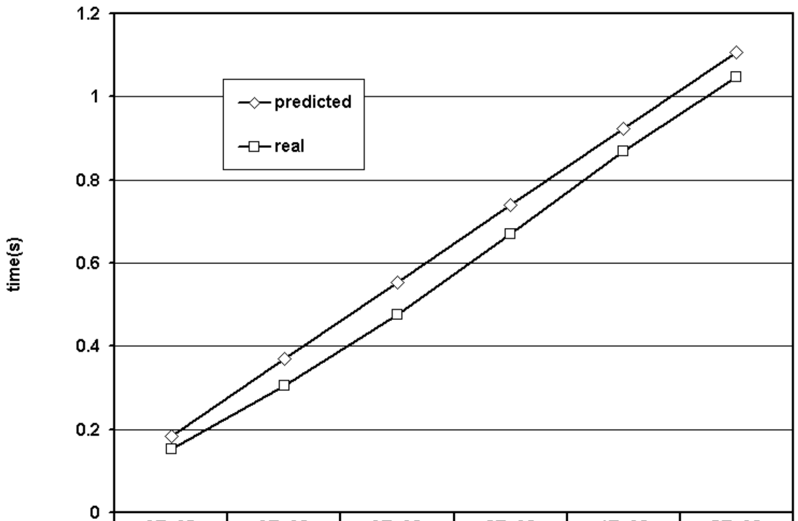
Figure 6.11: Accuracy of (1) when m=8, p=8



	20000	40000	60000	80000	1E+05	1E+05
predicted(2)	0.032803	0.055264	0.077726	0.100188	0.122649	0.145111
pre-comp(2)	0.003081	0.006158	0.009234	0.012311	0.015388	0.018465
real(2)	0.033733	0.054835	0.077826	0.098563	0.119468	0.142025

n

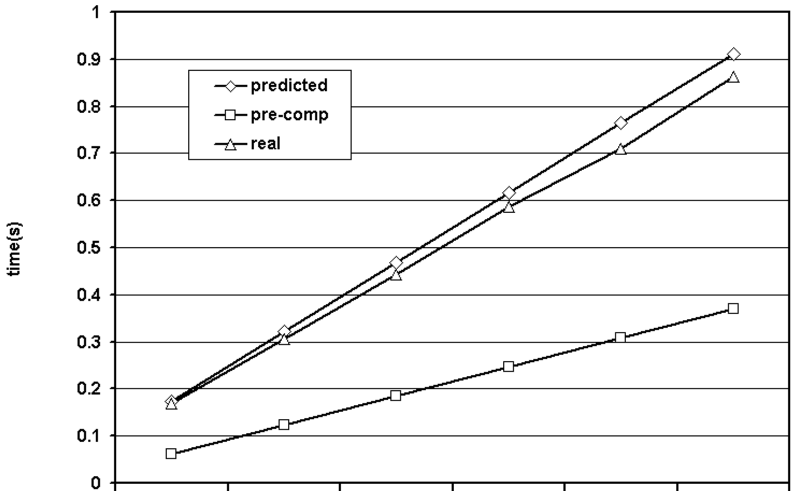
Figure 6.12: Accuracy of (2) when m=8, p=8



	8E+05	2E+06	2E+06	3E+06	4E+06	5E+06
predicted	0.184615	0.369231	0.553846	0.738462	0.923077	1.107692
real	0.153068	0.30511	0.475572	0.671029	0.870188	1.047206

n

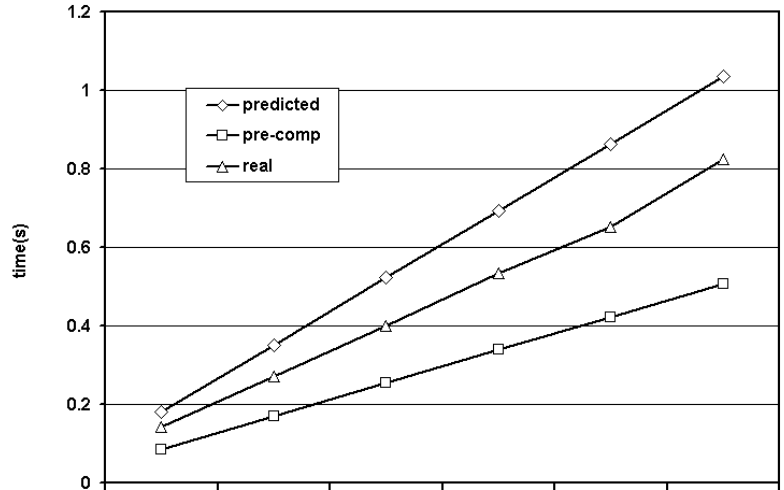
Figure 6.13: Accuracy of Bird's algorithm



	8E+05	2E+06	2E+06	3E+06	4E+06	5E+06
predicted	0.17354	0.321248	0.468957	0.616667	0.764374	0.912085
pre-comp	0.061556	0.12311	0.184665	0.246219	0.307774	0.369328
real	0.16854	0.304909	0.442978	0.586785	0.710103	0.862944

n

Figure 6.14: Accuracy of Skillicorn and Cai's algorithm



	8E+05	2E+06	2E+06	3E+06	4E+06	5E+06
predicted	0.181115	0.351894	0.522672	0.693449	0.864231	1.035008
pre-comp	0.084628	0.169252	0.253875	0.3385	0.423124	0.507748
real	0.142746	0.271711	0.399435	0.53311	0.651277	0.823869

n

Figure 6.15: Accuracy of Cole's algorithm

Chapter 7

Expansion: Costing Algorithm Derivation Steps

7.1 Introduction

Built on the cost analysis described in chapter 3 and 4, the main aim of this chapter is to augment our framework to partially relax our strict requirements on data structure regularity (but without losing static predictability). This arose because Jay's original work and our initial calculus required all elements of a vector (our nestable bulk data structure) to have the same shape. This made shape expression concise and consequently made automated analysis fast. However, although many practical algorithms (for example in linear algebra) can be expressed within this class, the restriction can be a big obstacle when applying the analysis to compare the costs of intermediate algorithms of a BMF style algorithm derivation because we often encounter an algorithm which cannot be expressed with it. For example, in the derivation of the *maximum segment sum* algorithm given below, while the final algorithm has the required property the algorithms at intermediate steps do not because they use the standard BMF functions *inits* and *tails*. However, this irregularity is entirely shapely, in the sense of being statically predictable. In this chapter we attempt to relieve the constraint (but preserve shapeliness) of our analysis while keeping the property of being automatable and avoiding increase in analysis cost (because irregular shaped vector data has more information to express its shape, consequently, the cost analysis using these shape expressions also tends to become more expensive). After this amendment we present our first analysis of a complete derivation of the *maximum segment sum* algorithm, and examine the accuracy of our predictions against the run time of real parallel programs as previous examples.

mss algorithm derivation

Remember from section 2.3.3 that Skillicorn and Cai derived a parallel algorithm from the specification by the following calculation.

$$
\begin{aligned}
mss &= \uparrow/\circ +/*\circ segs && (1)\\
&= \uparrow/\circ +/*\circ +\!\!+/\circ tails*\circ inits && (2)\\
&= \uparrow/\circ +\!\!+/\circ +/**\circ tails*\circ inits && (3)\\
&= \uparrow/\circ \uparrow/*\circ +/**\circ tails*\circ inits && (4)\\
&= \uparrow/\circ (\uparrow\circ +/*\circ tails)*\circ inits && (5)\\
&= \uparrow/\circ (+/_0\uparrow)*\circ inits && (6)\\
&= \uparrow/\circ +//_0\uparrow && (7)
\end{aligned}
$$

7.2 Expanding Shape Analysis

An important feature of our original source language, VEC-BSP, was that it constrained vector elements to have the same shape. This not only makes shape expression concise but also makes shape analysis much quicker than source program evaluation because it avoids purely data dependent computation. For example, computation of $\mathsf{map}\,(+1)\,v$ where v is a vector of length 1000 performs 1000 binary operations, but the corresponding shape analysis concerns only the shape $(1000, 1)$. This characteristic is a key point to keep cost analysis time reasonably small in spite of the extra computations of evaluation information. However, when we try to use this cost analysis to compare algorithms in BMF style derivations (using vectors to represent lists) we often encounter an algorithm which cannot be expressed with this constraint. Therefore, we need to relax our requirement of shape regularity to express this kind of algorithm. Because application of *tails* or *inits* generates a vector of vectors which still has some kind of regularity, the ith vector has $\frac{1}{2}i(i+1)$ elements, and this "triangular shape" can be characterised by the length of the last element, it might seems that only relaxation that allows a triangular shape in addition to the length-element pair shape would be enough. However, the observation of the following example of the intermediate real data structures of the initial version of the mss algorithm with $[1,2,3,4]$

$$\begin{array}{l}
[1,2,3,4] \\
\quad \Downarrow inits \\
[[1],[1,2],[1,2,3],[1,2,3,4]] \\
\quad \Downarrow *tails \\
[[[1]],[[2],[1,2]],[[3],[2,3],[1,2,3]],[[4],[3,4],[2,3,4],[1,2,3,4]]] \\
\quad \Downarrow +\!\!+/ \\
[[1],[2],[1,2],[3],[2,3],[1,2,3],[4],[3,4],[2,3,4],[1,2,3,4]] \\
\quad \Downarrow +/* \\
[1,2,3,3,5,6,4,7,9,10] \\
\quad \Downarrow \uparrow/ \\
10
\end{array}$$

reveals that the second intermediate data has the triangular shape, the third intermediate data is a vector of triangular shape and the fourth data has neither uniform shape nor triangular shape. Therefore, we relax the restriction further to allows vectors whose sub-vectors can have arbitrary length. When we relieve the constraints of the uniformity of vector elements, the intermediate shape information which is required to compute cost becomes extremely complicated and we need to introduce a new way to express it. In Skillicorn's calculus [77, 78] a *shape vector* is introduced to express shape information. For example, a shape vector $[n,m,p]$ denotes a list of n elements, each of which is a list of no more than m elements, each of which is an object of size no more than p. Except for the top level, the shape vector entry gives the maximum length of list at each level, and the last entry in a shape vector gives the total size of any substructure. This shape vector is annotated to indicate intermediate shape expressions in the algorithm in order to calculate the cost. For example, the initial version of the mss algorithm is

$$^{[1]} \uparrow / \circ^{[n^2]} + / * \circ^{[n^2,n]} +\!\!+ / \circ^{[n,n,n]} \, tails * \circ^{[n,n]} inits^{[n]}$$

When the argument vector is $[1,2,3,4]$ (that is $n=4$), these shape vectors are

$$\begin{array}{l} [4] \\ [4,4] \\ [4,4,4] \\ [16,4] \\ [16] \\ [1] \end{array}$$

This shape vector expression is concise, but the calculation of shape is done by hand because no attempt for automation has been made. Even if we can automate the calculation, using only information of this "no more than" type is inaccurate. In this example, the real resulting shape of the second last step is $(10,1)$ in our expression, but reduced shape vector is $[16]$.

Another solution would be to translate vectors to vectors keeping their form and simply replacing unknown real data with a dummy value. For example,

$$\begin{array}{l} [1,2,3] \rightarrow [1,1,1] \\ [[1],[1,2],[1,2,3]] \rightarrow [[1],[1,1],[1,1,1]] \\ \mathsf{map}\,(+1)\,[1,2,3] \rightarrow \mathsf{map}\,(\lambda x.1)\,[1,1,1] \\ \mathsf{fold}\,(+)\,[1,2,3] \rightarrow \mathsf{fold}\,(\lambda x,y.1)\,[1,1,1] \end{array}$$

Automation of shape deduction would be possible in a similar framework to that in chapter 3. The corresponding shapes of the above example are

$$[1,1,1,1]$$

$$[[1],[1,1],[1,1,1],[1,1,1,1]]$$
$$[[[1]],[[1],[1,1]],[[1],[1,1],[1,1,1]],[[1],[1,1],[1,1,1],[1,1,1,1]]]$$
$$[[1],[1],[1,1],[1],[1,1],[1,1,1],[1],[1,1],[1,1,1],[1,1,1,1]]$$
$$[1,1,1,1,1,1,1,1,1,1]$$
$$1$$

However this would involve similar memory usage and computation time to that of the original source program and is consequently unattractive.

Our solution in this chapter is to try to use the (length, element shape) pair as much as possible, that is, wherever it may be statically deduced (in reasonable analysis time) that sub-vector elements have the same shape. Otherwise, we use the vector constructor [ ] as the shape, allowing it to include both (,) and [ ] expressions as sub-vectors. This means that two kinds of shape expressions for vectors can exist in one shape expression.

The corresponding shape expressions should be

$$(4,1)$$
$$[(1,1),(2,1),(3,1),(4,1)]$$
$$[[(1,1)],[(1,1),(2,1)],[(1,1),(2,1),(3,1)],[(1,1),(2,1),(3,1),(4,1)]]$$
$$[(1,1),(1,1),(2,1),(1,1),(2,1),(3,1),(1,1),(2,1),(3,1),(4,1)]$$
$$(10,1)$$
$$1$$

respectively with the difference in data size becoming more significant as the length of the input vector grows. The memory and time required for analysis of a program will depend upon the degree of uniformity of its vector elements. As more vectors in the source program can be expressed in the (length, element shape) form, so analysis costs become smaller. The triangular shapes of the second intermediate data and the element vectors of the third intermediate data might be expressed more concisely by introducing new pair expression like $\{a,b\}$ where a is the length of the last element vector and b is the shape of an element of the element vectors. Its shape expression would be

$$
\begin{aligned}
&(4,1)\\
&\{4,1\}\\
&[\{1,1\},\{2,1\},\{3,1\},\{4,1\}]\\
&[(1,1),(1,1),(2,1),(1,1),(2,1),(3,1),(1,1),(2,1),(3,1),(4,1)]\\
&(10,1)\\
&1
\end{aligned}
$$

However the automatic reduction from the third expression to the fourth expression, for example, seems to be more difficult than that of the above shape expression for which a relatively straightforward reduction strategy can be defined as explained below.

The basic idea to make the shape analysis (using only vector shape and pair shape) with new expressions automatable is to define each shape function so that it has a function for each kind of expression, reflecting the corresponding shape change. The shape function f takes the form of

$$
\begin{array}{rcll}
f\,x & = & f_1\,x, & \text{if } x \text{ is pair}\\
 & & f_2\,x, & \text{if } x \text{ is vector}
\end{array}
$$

Each function shape distinguishes which expression is used for the argument shape from its type and returns an appropriate result shape. For example, the shape function of hd is

$$
\begin{array}{rcll}
\text{shape\_hd} \quad x & = & \mathsf{snd}\,x, & \text{if } x \text{ is pair shape}\\
 & & \mathsf{hd}\,x, & \text{if } x \text{ is vector shape}
\end{array}
$$

The shape function of map is

$$
\begin{array}{rcll}
\text{shape\_map} \quad f\,x & = & (\mathsf{t\_len}\,x, \mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,x))), & \text{if } x \text{ is pair shape}\\
 & & \mathsf{map}\,f\,x, & \text{if } x \text{ is vector shape}
\end{array}
$$

This amendment makes it possible to automate the new shape deduction and its information can be used to compute cost. However, in some cases, shapes which have uniform element shape are expressed as a vector rather than a pair. For example, shape_map which was defined above cannot detect that the elements of the vector at the second last step in the derivation have regular sized elements. From

$$[(1,1),(1,1),(2,1),(1,1),(2,1),(3,1),(1,1),(2,1),(3,1),(4,1)]$$

it reduces [1,1,1,1,1,1,1,1,1,1] rather than (10,1). Although both expressions are correct and have the same effect on the resulting cost, the latter is preferable in terms of both memory usage and time for shape analysis. One solution would be to check the result to determine whether its shape has uniform element shape by using some operation like fold eq, where eq is an operation to check if the two shapes are equal or not, after every function application. However this would be very expensive in term of analysis time. Our solution is to add one more information component *vector level* to the cost tuple and an information component *level change* to the application tuple. The vector level of a real data item is (the number of nested levels) + 1, e.g. 1 for a non-nested vector and 2 for a vector of a vector of a non-nested vector. We set the vector levels of a datum constant and a primitive function term to 0. The level change is a function that captures the change of the vector level after function application. Its type is $\mathsf{sz} \rightarrow \mathsf{sz}$. The new cost tuple takes the form

$$\langle \text{shape, data size, data pattern, vector level, cost} \rangle$$

and the new application tuple takes the form

$$\begin{array}{l}\text{argument shape}\\ \rightarrow \langle \text{result shape, application pattern, level change, application cost} \rangle\end{array}$$

When the result shape has a vector shape and the nested level of the result is 1, the vector is converted to a pair shape using topair in the new bspapp operation.

$$\mathsf{topair}\; n\, x = \text{if } \; (n = 1 \text{ and } x \text{ has vector type}) \; \text{ then } \; (\mathsf{length}\, x, \mathsf{hd}\, x) \; \text{ else } \; x$$

Note that this solution can detect regularity of the elements only when the vector level of a vector is 1. Finding a good solution which can deal with the general case remains future work. The definition of the new shape expressions is as follows. The shape of datum terms is 1. The shape of a tuple is a tuple (denoted with $\langle\, , \ldots ,\, \rangle$) of the shape of the components. The shape of a vector is a vector if its element shapes are not the same, or (length, element shape) pair if its element shapes are the same. Their types are

$$
\begin{array}{lcl}
tycost_C\,(D) & = & \mathsf{sz} \\
tycost_C\,(\mathsf{un}) & = & \mathsf{sz} \\
tycost_C\,(\theta_1 \times \cdots \times \theta_n) & = & tycost_C\,(\theta_1) \times \cdots \times tycost_C\,(\theta_n) \\
tycost_C\,(\mathsf{vec}\,\theta) & = & \mathsf{sz}\ \overline{\times} tycost_C\,(\theta)\ \text{ or }\ \mathsf{vec}\ tycost_C\,(\theta) \\
tycost_C\,(\mathsf{sz}) & = & \mathsf{sz}
\end{array}
$$

The new shape expression which allows sub-vectors to have different shapes affects associated definitions such as data size, application cost and communication cost. Data size can be computed by the operator size defined by

$$
\begin{array}{l}
\mathsf{size}\,\langle x_1, x_2, \cdots, x_n \rangle = \mathsf{size}\, x_1 + \mathsf{size}\, x_2 + \cdots + \mathsf{size}\, x_n \\
\mathsf{size}\,(x, y) = \mathsf{size}\, x \cdot \mathsf{size}\, y \\
\mathsf{size}\,[x_1, x_2, \cdots, x_n] = \mathsf{fold}\ (+)\ (\mathsf{map}\ (\mathsf{size})\ [x_1, x_2, \cdots, x_n]) \\
\mathsf{size}\ n = n
\end{array}
$$

Since the result of the evaluation of an argument could have different sized sub-vectors, the size of data (whose shape is x) which is sent to the processors by the master processor in a scattering is determined by the operator scatsz:

$$
\begin{array}{lcll}
\mathsf{scatsz}\, x & = & \mathsf{size}\, x \cdot (p-1)/p, & \text{if } x \text{ is pair shape} \\
 & & \mathsf{size}\,(\mathsf{drop}\,((\mathsf{length}\, x)/p)\, x), & \text{if } x \text{ is vector shape}
\end{array}
$$

where drop is a function which takes an integer n and a vector xs and removes the first n elements from xs. Computing the communication cost of broadcasting the data in a function is similar. Consequently, the communication cost of the part C which is counted in bspapp is replaced by

$$
\mathsf{comm\_cost}\ (\mathsf{t\_pattern}\,(f\,x))\ d'\ s\ (\mathsf{scatsz}\, x)
$$

where

$$
\begin{aligned}
\mathsf{comm\_cost}\ & ap\_pat\ \ dat\_pat\ \ f\_sz\ \ x\_sz \\
&\tilde{=}\, 0, && \text{if } ap\_pat = \mathsf{SEQ} \\
&= (f\_sz \cdot (p-1) - x\_sz \cdot ((p-1)/p)) \cdot g - l, && \text{if } dat\_pat\ = \mathsf{MAP} \\
&= (f\_sz \cdot (p-1) + x\_sz \cdot ((p-1)/p)) \cdot g, && \text{otherwise}
\end{aligned}
$$

The application costs for combinators are given in the next section. In the following definitions we use t_lchange for taking the level change component from the result of an application of a function shape to an argument shape.

The definition of new bspapp is

$$cost\,(t\ t') = \mathsf{bspapp}\ cost\,(t)\ cost\,(t')$$

$$
\begin{aligned}
& \mathsf{bspapp}\,\langle f, s, d, n, T\rangle \langle x, s', d', n', T'\rangle \\
= \quad & \langle \mathsf{topair}\,(\mathsf{t\_lchange}\ (f\,x)\ n')\,(\mathsf{t\_shp}\ (f\,x)),\ \mathsf{data\_sz}\,(\mathsf{t\_apcost}\,(f\,x)), \\
& \mathsf{t\_pattern}\,(f\,x),\ \mathsf{t\_lchange}\,(f\,x)\,n',\ (T + T') \\
& + \lambda\langle p, g, l\rangle.(\mathsf{comm\_cost}\ (\mathsf{t\_pattern}\ (f\,x))\ d'\ s\ (\mathsf{scatsz} x) + l) + \mathsf{t\_apcost}\,(f\,x)\rangle
\end{aligned}
$$

where

$$
\begin{aligned}
\mathsf{data\_sz}\ ap\_c &= s + s', && \text{if } ap\_c \tilde{=} 0 \\
&= \mathsf{t\_size}(f\,x), && \text{otherwise}
\end{aligned}
$$

The main difference from the old bspapp (except for the comm_cost part explained above) is that the new bspapp calculates the vector level of the application result by applying the level change of a function to the vector level of an argument, and then, when it is 0 and the shape of the result is a vector shape, the shape of the result is converted to a pair shape.

Recall that the definition of the application pattern is: SEQ for a sequential function; MAP for a parallel function whose implementation template finishes by gathering local results to the master; FOLD for any other parallel function. The data pattern indicates which application pattern was used to generate the term (0 for atomic term). A translation function *cost* translates source terms to cost tuples. For example,

$$cost\,(d) \quad = \quad \langle 1,1,\mathsf{SEQ},0,0\rangle \text{ where } d \text{ is a datum constant}$$

Note that the second 0 from the right hand end means the vector level of a datum constant is 0.

$$\begin{aligned} cost\,(d) \quad &= \quad \langle \lambda x.\langle \lambda y.\langle 1,\mathsf{SEQ},+0,1\rangle,\mathsf{SEQ},+0,0\rangle,0,\mathsf{SEQ},0,0\rangle \\ &\quad\quad \text{where } d \text{ is a binary datum operation} \end{aligned}$$

Note that the second 0 from the right hand end means the vector level of a binary datum operation itself is 0. The first $+0$ from the right hand means application of d to the first argument does not change the vector level, and the second $+0$ means the application of the resulting function to the second argument also does not change the vector level.

The new cost functions for x and $\lambda x.t$ are

$$\begin{aligned} cost\,(x) \quad &= \quad \langle x,\mathsf{size}\,x,\mathsf{SEQ},\mathsf{vlevel}\,x,0\rangle \\ cost\,(\lambda x.t) &= \quad \langle \lambda x.\langle \pi_1\,(cost\,(t)),\mathsf{SEQ},+(\pi_4\,(cost(t)) - \mathsf{vlevel}\ x),\pi_5\,(cost\,(t))\rangle, \\ &\quad\quad 0,\mathsf{SEQ},0,0\rangle \end{aligned}$$

where $cost(t)$ gives the cost tuple of $t(x)$, which is computed by using the cost tuple of variable x, $cost(x) = \langle x,\mathsf{size}\ x,\mathsf{SEQ},\mathsf{vlevel}\ x,0\rangle$. The function $\lambda x.cost(t(x))$ represents the function which takes s_a: the shape of the argument a, and generates cost tuple of $t(a)$. The shape component of $cost(x)$ is x because it is substituted by the shape of the result of evaluation of an argument s_a. The data size component of $cost\,(x)$ is $\mathsf{size}\,x$ which computes the data size of a when x is substituted by s_a. The data pattern component of $cost\,(x)$ is SEQ because the result of evaluation a is treated as initial data for evaluation of $t(a)$ in the application part. The vector level component of $cost\,(x)$ is $\mathsf{vlevel}\,x$ which computes the vector level of a when x is substituted by s_a. The definition of vlevel which computes the vector level from its shape is:

$$\begin{aligned} \mathsf{vlevel}\ \tilde{}A \quad &= \quad 0 \\ \mathsf{vlevel}\ (A,B) \quad &= \quad \mathsf{vlevel}\,B+1 \\ \mathsf{vlevel}\ [A] \quad &= \quad \mathsf{vlevel}\ (\mathsf{hd}\,A)+1 \end{aligned}$$

The cost component of $cost\,(x)$ is 0 because the result of evaluation a is treated as initial data for evaluation of $t\,(a)$ in the application part.

In the definition of $cost\,(\lambda x.t)$, the result shape of application $(\lambda x.t)$ is equal to the shape component of $cost(t)$, that is $\pi_1\,(cost\,(t))$ because it computes the shape of $t\,(a)$ when x is substituted by s_a. Note that $\lambda x.\pi_1\,(cost\,(t))$ captures the shape change. The application pattern of application $(\lambda x.t)$ to some argument a is SEQ because there is no communication in the redistribution communication part. (Note that the data of the result of evaluation of a is used as initial data stored in the master for evaluation of $t\,(a)$ in the application part and the necessary data in $(\lambda x.t)$ is statically distributed to the processors.) The level change of $(\lambda x.t)$ is $+(\pi_4(cost(t)) - \mathsf{vlevel}\,x)$ because $\pi_4(cost(t))$ computes the vector level of the result $t\,(a)$ when x is substituted by s_a and $\mathsf{vlevel}\,x$ computes the vector level of the argument a when x is substituted by s_a. The application cost of application $(\lambda x.t)$ is equal to the cost component of cost tuple of $t(x)$, that is $\pi_5\,(cost\,(t))$ because it computes cost of $t\,(a)$, (which is evaluated in the application part,) when x is substituted by s_a.

7.3 New Cost Functions for Combinators

Defining *cost* functions for combinators involves defining functions which capture the shape change and determining the application cost and the application pattern based on assumed implementation skeletons. For some combinators which take a vector as the argument we need to define two kinds of functions selected according whether the argument shape expression is a pair (referred to as *pair shape*) or a vector (referred to as *vector shape*). The cost functions which are given below, except for map and fold when x is pair shape, are introduced for the first time in this chapter. In particular, the introduction of foldconcat, inits and tails are made possible by the new shape expression.

map

The modelled implementation of map is: apply the function sequentially on the vector segments in each processor then gather the results to the master. Its cost function is:

$$
\begin{aligned}
&cost\,(\mathsf{map}) = \langle \lambda f.\langle \lambda x.\text{shape\_map}, \mathsf{SEQ}, +0, 0\rangle, 0, \mathsf{SEQ}, 0, 0\rangle,\\
&\text{shape\_map}\\
= \ &\langle(\mathsf{t\_len}\,x, \mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,x))), \mathsf{MAP}, \mathsf{t\_lchange}\,(f\,x), \mathsf{t\_apcost}\,(f\,(\mathsf{t\_eshp}\,x))\cdot\\
&(\mathsf{t\_len}\,(x)/p) + \mathsf{t\_size}\,(f\,(\mathsf{t\_eshp}\,x))\cdot(\mathsf{t\_len}\,(x)/p)\cdot(p-1)\cdot g + l\rangle, \quad \text{if } x \text{ is pair shape}\\
&\langle \mathsf{map}\,(\mathsf{t\_shp})\,tuples, \mathsf{MAP}, \mathsf{t\_lchange}\,(f\,x), \mathsf{maxsum}\,(\mathsf{map}\,(\mathsf{t\_apcost})\,tuples)\\
&+\mathsf{gath\_sz}\,(\mathsf{map}\,(\mathsf{t\_size})\,tuples)\cdot g + l\rangle, \quad \text{if } x \text{ is vector shape}\\
&\text{where } \ tuples = (\mathsf{map}\,f\,x)
\end{aligned}
$$

The second 0 from the right hand end means the vector level of map is 0. The $+0$ means the application of map to a given function t_f does not change the vector level. The level change function of map t_f is the same as the level change function of t_f, that is $\mathsf{t\_lchange}\,(f\,x)$. When x is a pair shape the cost function of map is the same as that in chapter 4 except for this additional information of the vector level and the level change. When x is a vector shape the analysis performs map $f\ x$, that is the shape function f is applied to each element of x generating each application tuple, which becomes an element of the resulting vector. The result shape of map $t_f\ t_x$ is obtained by taking the result shape component of each application tuple, that is $\mathsf{map}\,(\mathsf{t\_shp})\ tuples$. Since the data size of the vector elements allocated to each processor can be different, the local computation cost to apply t_f to the segments in each processor can also be different. Therefore the cost of this parallel computation part is the maximum of the local computation cost in any processor. It is computed by taking the application cost component from each application tuple (that is map $(\mathsf{t\_apcost})\ tuples$), computing the summation for every $(\mathsf{length}\ x)/p$ elements, and then taking the maximum among them. The last two steps are expressed by the operator maxsum. The communication cost is computed using the message size to gather the local result, that is taking the size of each result shape of element, (that is map $(\mathsf{t\_size})\ tuples$) and computing the summation of them excluding the first $(\mathsf{length}\ x)/p$ elements which are already kept in the master. The last step is expressed by the operator gath_sz. The definitions of maxsum and gath_sz are

$$
\begin{aligned}
\mathsf{maxsum}\quad x \quad &= \quad (\mathsf{sum}\,(\mathsf{take}\,(\mathsf{t\_len}\,(x)/p)\,x)) \uparrow \\
&\qquad (\mathsf{if}\quad \mathsf{length}\,(\mathsf{drop}\,(\mathsf{t\_len}\,(x)/p)\,x) > 0 \\
&\qquad \mathsf{then}\quad \mathsf{maxsum}\,(\mathsf{drop}\,(\mathsf{t\_len}\,(x)/p)\,x)\quad \mathsf{else}\quad 0), \\
\mathsf{gath\_sz}\quad x \quad &= \quad \mathsf{sum}\,(\mathsf{drop}\,(\mathsf{length}\,(x)/p)\,x)
\end{aligned}
$$

where take is a function which takes an integer n and a vector xs and returns the first n elements from xs.

fold

The modelled implementation of fold is: the sub-vectors are folded sequentially on each processor, with results then transferred to the master processor which folds them together. As in Skillicorn's calculus, fold is used only with operators which take constant space, that is the shapes of their results are same as the shapes of their arguments. Fold operations with a non-constant space operation are defined individually (e.g. fold-concat defined below). Its cost function is:

$$
\begin{aligned}
&\mathit{cost}\,(\mathsf{fold}) \;= \langle \lambda \oplus .\, \langle \lambda x.\, \mathsf{shape\_fold}, \mathsf{SEQ}, +0, 0\rangle, 0, \mathsf{SEQ}, 0, 0\rangle, \\
&\mathsf{shape\_fold} \\
= \;&\langle \mathsf{t\_eshp}\,x, \mathsf{FOLD}, (-1), \mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,x))\,(\mathsf{t\_eshp}\,x)) \cdot (\mathsf{t\_len}\,(x)/p + \\
&p-2) + \mathsf{size}\,(\mathsf{t\_eshp}\,x) \cdot (p-1) \cdot g + l\rangle, \quad \text{if } x \text{ is pair shape} \\
&\langle \mathsf{hd}\,x, \mathsf{FOLD}, (-1), \mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{hd}\,x))(\mathsf{hd}\,x)) \cdot (\mathsf{length}\,(x)/p + p - 2) \\
&+ \mathsf{size}\,(\mathsf{hd}\,x) \cdot (p-1) \cdot g + l\rangle, \quad \text{if } x \text{ is vector shape}
\end{aligned}
$$

The second 0 from the right hand end means the vector level of fold is 0. The $+0$ means the application of fold to a given function $t_\oplus$ does not change the vector level. The level change function of fold $t_\oplus$ is (-1) because fold $t_\oplus$ reduces the vector level by 1. When x is a pair shape, the shape of the result of fold $t_\oplus\, t_x$ is t_eshp x because of the assumption of constant space. The application cost of fold $t_\oplus\, t_x$ is the computation cost for $\mathsf{t\_len}\,(x)/p+p-2$ applications of fold $t_{\oplus}$ (in which $\mathsf{t\_len}\,(x)/p-1$ are for the parallel part and $p-1$ are for the sequential part), that is

$$\mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,x))\,(\mathsf{t\_eshp}\,x))\cdot(\mathsf{t\_len}\,(x)/p+p-2),$$

the communication cost for gathering the local results, that is $\mathsf{size}\,(\mathsf{t\_eshp}\,x)\cdot(p-1)\cdot g$ and the synchronisation cost l. The case when x is a vector shape is similar, but the computation of the shape and the application cost is performed using hd and length instead of t_eshp and t_len to take the element shape and the length respectively.

foldconcat

In the expanded version described in this chapter, foldconcat (fold with concatenate) is added to the primitive functions in order to express one of the intermediate algorithms in the mss derivation. The modelled implementation of foldconcat is to concatenate sequentially in the master processor rather than in parallel to avoid distribution cost.

$$
\begin{aligned}
& \mathit{cost}\,(\mathsf{foldconcat}) = \langle \lambda x.\mathrm{shape\_foldconcat}, 0, \mathsf{SEQ}, 0, 0\rangle, \\
& \mathrm{shape\_foldconcat} \\
= \; & \langle(\mathsf{t\_len}\;x\cdot\mathsf{t\_len}\,(\mathsf{t\_eshp}\,x),\mathsf{t\_eshp}\,(\mathsf{t\_eshp}\,x)),\mathsf{SEQ},(-1), \\
& \mathsf{concatConst}\cdot(\mathsf{t\_len}\,x-1)\rangle,\;\; \text{if } x \text{ is pair shape} \\
& \langle\mathsf{foldconcat}\,x,\mathsf{SEQ},(-1),\mathsf{concatConst}\cdot(\mathsf{length}\,(x)-1)\rangle,\;\; \text{if } x \text{ is vector shape}
\end{aligned}
$$

Working from the right hand end, the cost of foldconcat itself and vector level are 0. The data pattern is SEQ and the message size is 0. When x is a pair shape, the result shape of $\mathsf{foldconcat}\,x$ is $(\mathsf{t\_len}\;x\cdot\mathsf{t\_len}\,(\mathsf{t\_eshp}\,x),\mathsf{t\_eshp}\,(\mathsf{t\_eshp}\,x))$. The application pattern is SEQ since it is sequential function. The level change is (-1) because foldconcat reduces the vector level by 1. The application cost is some constant time for concatenation multiplied by $\mathsf{t\_len}\,x-1$. When x is a vector shape, the result shape is $\mathsf{foldconcat}\,x$ and the other components are the same as those when x is a pair shape.

scan

The implementation of scan is: the assigned block of elements is scanned sequentially; the final value of local scan is scanned across processors in parallel using the obvious tree algorithm; the result of the global scan on processor $i\,(< p)$ is sent to processor $i+1$. In each processor, $t_{\oplus}$ is applied to the pair of the result of global scan and the local results; the result in each processor is gathered to the master processor. We make the same restriction concerning constant space operators as for fold. Its cost function is:

$$
\begin{aligned}
& \mathit{cost}\,(\mathsf{scan}) \ = \langle \lambda f.\langle \lambda x.\text{shape\_scan}, \mathsf{SEQ}, +0, 0\rangle, 0, \mathsf{SEQ}, 0, 0\rangle, \\
& \text{shape\_scan} \\
= \ & \langle x, \mathsf{MAP}, 0, \mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(x)))\cdot(2\cdot(\mathsf{t\_len}\,(x)/p)-1+ \\
& \log\,(p)) + (\mathsf{size}\,(\mathsf{t\_eshp}\,(x))\cdot(\log\,(p)+1)+(\mathsf{size}\,(x)/p)\cdot(p-1))\cdot g \\
& +(\log\,(p)+2)\cdot l\rangle, \quad \text{if } x \text{ is pair shape} \\
& \langle x, \mathsf{MAP}, 0, \mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{hd}\,(x)))(\mathsf{hd}\,(x)))\cdot(2\cdot(\mathsf{length}\,(x)/p)-1+ \\
& \log\,(p)) + (\mathsf{size}\,(\mathsf{hd}\,(x))\cdot(\log\,(p)+1)+(\mathsf{size}\,(x)/p)\cdot(p-1))\cdot g \\
& +(\log\,(p)+2)\cdot l\rangle, \quad \text{if } x \text{ is vector shape}
\end{aligned}
$$

The second 0 from the right hand end means the vector level of scan is 0. The $+0$ means the application of scan to a given function $t_{\oplus}$ does not change the vector level. The level change function of scan $t_{\oplus}$ is 0 since scan does not change the vector level. When x is a pair shape, the shape of the result of $\mathit{scan}\ t_{\oplus}\ t_x$ is x because of the assumption of constant space. The application cost of scan $t_{\oplus}\ t_x$ is the computation cost for $2\cdot(\mathsf{t\_len}\,(x)/p)-1+\log(p)$ applications of $t_{\oplus}$ (in which $(\mathsf{t\_len}\,(x)/p)-1$ are for the local scan, log (p) are in the tree algorithm and $\mathsf{t\_len}\,(x)/p$ are for between each element of the local scan result and the global scan result), that is

$$
\mathsf{t\_apcost}\,(\mathsf{t\_shp}\,(\oplus\,(\mathsf{t\_eshp}\,(x)))(\mathsf{t\_eshp}\,(x)))\cdot(2\cdot(\mathsf{t\_len}\,(x)/p)-1+\log\,(p)),
$$

the communication cost is $\mathsf{size}\,(\mathsf{t\_eshp}\,(x))\cdot\log\,(p)\cdot g$ in the tree algorithm, $\mathsf{size}\,(\mathsf{t\_eshp}\,(x))\cdot g$ for sending the local result to the next processor and $\mathsf{size}\,(\mathsf{t\_eshp}\,(x))\cdot$

$(p-1)$ for gathering the local results, in total,

$$(\mathsf{size}\,(\mathsf{t\_eshp}\,(x))\cdot(\log(p)+1)+(\mathsf{size}\,(x)/p)\cdot(p-1))\cdot g$$

and the synchronisation cost is $\log(p)\cdot l$ in the tree algorithm, l after sending the local result to the next processor and l for gathering, in total, $(\log(p)+2)\cdot l$. The case when x is a vector shape is similar, but the computation of the shape and the application cost is performed using hd and length instead of $\mathsf{t\_eshp}$ and $\mathsf{t\_len}$ to take the element shape and the length respectively.

inits and tails

The modelled implementation of inits begins with each processor computing the local initial segments of its part of the list. The last element of this local result is then passed to the processor immediately to its right, where it is prepended to each of the partial initial segments held by that processor. After $p-1$ steps, the values from the first processor are prepended to each of the segments in the last processor, and then the local results from all processors are gathered to the master processor. Its cost function is:

$$
\begin{aligned}
& \mathit{cost}\,(\mathsf{inits}) \;=\langle \lambda x.\,\mathrm{shape\_inits}, 0, \mathsf{SEQ}, 0, 0\rangle,\\
& \mathrm{shape\_inits}\\
= \;& \langle\,[(1,\mathsf{t\_eshp}\,(x)),(2,\mathsf{t\_eshp}\,(x)),\cdots,(\mathsf{t\_len}\,(x),\mathsf{t\_eshp}\,(x))],\\
& \mathsf{MAP}, +1, \mathsf{concatConst}\cdot(\mathsf{t\_len}\,(x)/p)\\
& +(\mathsf{size}\,(\mathsf{t\_eshp}\,(x))\cdot(\mathsf{t\_len}\,(x)/p)\cdot g+(\mathsf{t\_len}\,(x)/p)\cdot\mathsf{concatConst}+l)\cdot(p-1)+\\
& \mathsf{size}\,(\mathsf{drop}\,(\mathsf{t\_len}\,(x)/p))\,[(1,\mathsf{t\_eshp}\,(x)),(2,\mathsf{t\_eshp}\,(x)),\cdots,(\mathsf{t\_len}\,(x),\mathsf{t\_eshp}\,(x))]\cdot g\\
& +l\rangle,\ \text{ if } x \text{ is pair shape}\\
& \langle\,\mathsf{inits}\,x, \mathsf{MAP}, +1,\ \mathsf{concatConst}\cdot(\mathsf{length}\,(x)/p)\\
& +((\mathsf{size}\,(\mathsf{take}\,(\mathsf{length}\,(x)/p)\,x))\cdot g+\mathsf{concatConst}+l)\cdot(p-1)\\
& +\mathsf{size}\,(\mathsf{drop}\,(\mathsf{length}\,(x)/p)\,\mathsf{inits}\,x)\cdot g+l\rangle,\ \text{ if } x \text{ is vector shape}
\end{aligned}
$$

Working from the right end, the cost of inits itself and vector level are 0. The data patten is SEQ. The message size is 0. When x is a pair shape, the result shape of $\mathsf{inits}\,x$

is

$$[(1, \mathsf{t\_eshp}\,(x)), (2, \mathsf{t\_eshp}\,(x)), \cdots, (\mathsf{t\_len}\,(x), \mathsf{t\_eshp}\,(x))]$$

which is a vector shape. The level change function is $(+1)$ since inits increases the vector level by 1. The application pattern is MAP since the modelled implementation ends by gathering the local results. The application cost is: $\mathsf{concatConst} \cdot (\mathsf{t\_len}\,(x)/p)$ for computing the local initial segments; $(\mathsf{size}\,(\mathsf{t\_eshp}\,(x)) \cdot (\mathsf{t\_len}\,(x)/p) \cdot g + (\mathsf{t\_len}\,(x)/p) \cdot \mathsf{concatConst} + l) \cdot (p-1)$ for the $p-1$ steps of passing the last element of the local result followed by prepending to each element; and

$$[(1, \mathsf{t\_eshp}\,(x)), (2, \mathsf{t\_eshp}\,(x)), \cdots, (\mathsf{t\_len}\,(x), \mathsf{t\_eshp}\,(x))] \cdot g + l$$

for gathering the local results followed by a synchronisation. When x is a vector shape, the result shape is inits x. The application cost is $\mathsf{concatConst} \cdot (\mathsf{t\_len}\,(x)/p)$ for computing the local initial segments, $((\mathsf{size}\,(\mathsf{take}\,(\mathsf{length}\,(x)/p)\,x)) \cdot g + \mathsf{concatConst} + l) \cdot (p-1)$ for the $p-1$ steps of passing the last element of the local result followed by prepending to each element and $\mathsf{size}\,(\mathsf{drop}\,(\mathsf{length}\,(x)/p)\,\mathsf{inits}\,x) \cdot g + l$.

tails is an analogue of inits, which computes the suffix segments of its argument vector and its cost function is the same as that of inits.

7.4 Costing Derivation Steps

Reflecting the new shape expression, our Haskell implementation, which was outlined in chapter 5 has been modified. In this section, we first rewrite the BMF expression of each intermediate algorithm as the corresponding VEC-BSP program. Next we use our cost calculator to predict the cost of five different algorithms in the derivation steps, comparing one to another for each transformation step. Finally, we test the accuracy of the predicted costs against the real run time of hand compiled BSP program in Oxford BSPlib. The comparison of efficiencies depends on the values of the BSP benchmark which capture performance characteristics of computation, communication and synchronisation of the target systems. Our real target system is the same as that is used in chapter 6, that is an 8-processor Sun HPC 3500 UltraSPARC II machine.

As before, BSP parameters obtained by running a benchmark program provided by Oxford BSPlib are $p = 8$, $g = 1.6$, $l = 67150$. The binary operator constant is set at 1 and the total calculated cost in operations is converted into seconds by dividing by 13 million as directed by s, the benchmark returned factor which normalises l and g to the single processor computational speed.

7.4.1 VEC-BSP Programs of Derivation Steps

First, we express the mss derivation in terms of VEC-BSP program using the new operations that were introduced above.

$$
\begin{aligned}
& \mathsf{fold}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{fold}\,(+))\,(\mathsf{foldconcat}\,(\mathsf{map}\,(\mathsf{tails})\,(\mathsf{inits}\,x)))) && (8)\\
= \;& \mathsf{fold}\,(\uparrow)\,(\mathsf{foldconcat}\,(\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))\,(\mathsf{map}\,(\mathsf{tails})\,(\mathsf{inits}\,x)))) && (9)\\
= \;& \mathsf{fold}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{fold}\,(\uparrow))\,(\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))\,(\mathsf{map}\,(\mathsf{tails})(\mathsf{inits}\,x)))) && (10)\\
= \;& \mathsf{fold}\,(\uparrow)\,(\mathsf{map}\,(\lambda v.\,((\mathsf{fst}\,A) \uparrow (\mathsf{snd}\,A)))(\mathsf{inits}\,x)) && (11)\\
& \text{where}\\
& A = \mathsf{fold}\,(\oslash)\,(\mathsf{map}\,(\mathsf{pair}\,0)\,v)\\
& \oslash = \lambda xy.\mathsf{pair}\,(\mathsf{fst}\,x + \mathsf{fst}\,y)((\mathsf{snd}\,x + \mathsf{fst}\,y) \uparrow 0)\\
= \;& \mathsf{fold}\,(\uparrow)\,(\mathsf{shiftright}(0)\,(\mathsf{map}\,(\uparrow)\,(\mathsf{scan}\,(\oslash)\,(\mathsf{map}\,(\mathsf{pair}\,0)\,v)))) && (12)\\
& \text{where}\\
& \oslash = \lambda xy.\mathsf{pair}\,(\mathsf{fst}\,x + \mathsf{fst}\,y)((\mathsf{snd}\,x + \mathsf{fst}\,y) \uparrow 0)
\end{aligned}
$$

Since recur-reduce and recur-scan can be expressed as a reduce and a scan respectively (chapter 8 in Skillicorn [77]) the BMF expressions of algorithms (6) and (7) can be expressed in VEC-BSP as (11) and (12), where $\mathsf{shiftright}(0)$ rotates the entire list right one place, moving a single element from each processor to the next and inserting 0 at the left end. shiftright is now added to the set of primitive functions. The cost function of shiftright is defined as

$$
\begin{aligned}
cost\,(\mathsf{shiftright}) \;=\; & \langle \lambda e.\langle \lambda x.\langle (\mathsf{t\_len}\,(x) + 1, \mathsf{t\_eshp}\,x), \mathsf{SEQ}, 0, \mathsf{size}\,(\mathsf{t\_eshp}\,(x)) \cdot g\rangle, \mathsf{SEQ}, 0, 0\rangle,\\
& 0, \mathsf{SEQ}, 0, 0\rangle
\end{aligned}
$$

7.4.2 BSP Implementations of Derivation Steps

We outline the BSP implementation of each BSP program (8)-(12) based on the implementation strategy given in 3.3 and implementation skeletons for combinators in 6.3.

Algorithm (8): $\mathsf{fold}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{fold}\,(+))\,(\mathsf{foldconcat}\,(\mathsf{map}\,(\mathsf{tails})\,(\mathsf{inits}\,x))))$

- inits phase:
  1. the contents of x in the master processor are scattered to the p processors;
  2. synchronisation;
  3. each processor computes the local initial segments;
  4. the last element of the local initial segment in processor i is sent to processor $i+1$ and prepended to each of the partial initial segments held by processor $i+1$ followed by synchronisation. this is repeated p times.
- $\mathsf{map}\,(\mathsf{tails})$ phase:
  5. each processor computes the tail segments for each partial initial segment held by the processor;
  6. the local result in each processor is gathered to the master processor;
  7. synchronisation.
- $\mathsf{foldconcat}$ phase:
  8. the master processor computes foldconcat for the result of 7.
- $\mathsf{map}\,(\mathsf{fold}\,(+))$ phase:
  9. the contents of the result of 8 in the master processor are scattered to the p processors in vector block manner;
  10. each processor computes the $\mathsf{fold}\,(+)$ for each vector held by the processor.
- $\mathsf{fold}\,(\uparrow)$ phase:

11. each processor computes $\mathsf{fold}\,(\uparrow)$, that is takes the maximum of the result of 10 held by the processor;

12. the local result in each processor is gathered to the master processor;

13. synchronisation;

14. the master processor computes $\mathsf{fold}(\uparrow)$, that is takes the maximum of the gathered local results.

Algorithm (9): $\mathsf{fold}\,(\uparrow)\,(\mathsf{foldconcat}\,(\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))\,(\mathsf{map}\,(\mathsf{tails})\,(\mathsf{inits}\,x))))$

- inits phase:
  This is the same as the step 1.-4. of algorithm (8).
- $\mathsf{map}\,(\mathsf{tails})$ phase:

5. each processor computes the tail segments for each partial initial segments held by the processor.

- $\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))$ phase:

6. each processor computes the $\mathsf{fold}\,(+)$ for each inner vector held by the processor;

7. the local result in each processor is gathered to the master processor;

8. synchronisation.

- $\mathsf{foldconcat}$ phase:

9. the master processor computes foldconcat for the result of 7.

- $\mathsf{fold}\,(\uparrow)$ phase:

10. the contents of the result of 9 in the master processor are scattered to the p processors;

11. each processor computes $\mathsf{fold}\,(\uparrow)$ for the result of 10;

12. the local result in each processor is gathered to the master processor;

13. synchronisation;

14. the master processor computes $\mathsf{fold}\,(\uparrow)$ for the gathered local results.

The BSP implementation of the algorithm (9) is similar to algorithm (8). The difference between (8) and (9) is the timing of the use of $\mathsf{foldconcat}$. We look at the difference using an example in which the input vector is $[1,2,3,4]$ and the number of processors is 4. In (8), after the computation of $\mathsf{map}\,(\mathsf{tails})$ (step 5), the local results are

$P0 : [[1]],$
$P1 : [[2],[1,2]],$
$P2 : [[3],[2,3],[1,2,3]],$
$P3 : [[4],[3,4],[2,3,4],[1,2,3,4]]$

The local result in each processor is gathered into the master processor (step 6) followed by $\mathsf{foldconcat}$ (step 8), and redistributed among the worker processors evenly in vector-block manner (step 9). The data distribution at this time is

$P0 : [1],[2],[1,2]$
$P1 : [3],[2,3],[1,2,3],$
$P2 : [4],[3,4],$
$P3 : [2,3,4],[1,2,3,4]$

and then $\mathsf{fold}\,(+)$ is applied in each vector, resulting in

$P0 : 1,2,3,$
$P1 : 3,5,6,$
$P2 : 4,7,$
$P3 : 9,10$

As we can see, after the application of $\mathsf{map}\,(\mathsf{tails})\,(\mathsf{inits}\,x))$ the outermost elements have different number of inner vector elements which become the outermost elements after application of the next $\mathsf{foldconcat}$ in the master processor. Since redistribution of these for the next $\mathsf{fold}\,(+)$ is made in terms of these new outermost elements, the load imbalance caused by inits and $\mathsf{map}\,(\mathsf{tails})$ is improved and the computation costs of the following operations are reduced, but the gather and redistribution cost to perform $\mathsf{foldconcat}$ is introduced.

In contrast, in the algorithm (9), for the result of $\mathsf{map}\,(\mathsf{tails})$ (step 5)

$P0 : [[1]],$
$P1 : [[2],[1,2]],$
$P2 : [[3],[2,3],[1,2,3]],$
$P3 : [[4],[3,4],[2,3,4],[1,2,3,4]]$

$\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))$ is applied immediately (step 6), resulting in

$P0 : [1],$
$P1 : [2,3],$
$P2 : [3,5,6],$
$P3 : [4,7,9,10]$

The local result in each processor is now gathered into the master processor (step 7) followed by foldconcat (step 9), and redistributed among the worker processors evenly in vector-block manner in step 10, resulting in

$P0 : 1,2,3,$
$P1 : 3,5,6,$
$P2 : 4,7,$
$P3 : 9,10$

As we can see, the application $\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))$ is applied to the unbalanced distributed data, which could introduce a considerable parallel computation cost. However, the gather and redistribution costs to perform foldconcat are relatively small because the data size transmitted for the communication is small after the application of $\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))$.

Algorithm (10): $\mathsf{fold}\,(\uparrow)\,(\mathsf{map}\,(\mathsf{fold}\,(\uparrow))\,(\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))\,(\mathsf{map}\,(\mathsf{tails})(\mathsf{inits}\,x))))$

- inits phase:
  This is the same as the step 1.-4. of algorithm (8) and (9).

- $\mathsf{map}\,(\mathsf{tails})$ phase:

  5. each processor computes the tail segments for each partial initial segments held by the processor;

- $\mathsf{map}\,(\mathsf{map}\,(\mathsf{fold}\,(+)))$ phase:

  6. each processor computes the $\mathsf{map}\,(\mathsf{fold}\,(+))$ for each vector held by the processor.

- $\mathsf{map}\,(\mathsf{fold}\,(\uparrow))$ phase:

  7. each processor computes the $\mathsf{fold}\,(\uparrow)$ for each vector held by the processor.

- $\mathsf{fold}\,(\uparrow)$ phase:

  8. each processor takes the maximum of the result of 7;

  9. the local result in each processor is gathered to the master processor;

  10. synchronisation;

  11. the master processor takes the maximum of the gathered local results.

The BSP implementation of algorithm (10) is similar to (8) and (9). In algorithm (10), $\mathsf{redconcat}$ is not used and gathering occurs only in the last phase $\mathsf{fold}\,(\uparrow)$.

Algorithm (11): $\mathsf{fold}\,(\uparrow)\,(\mathsf{map}\,(\lambda v.\,((\mathsf{fst}\,A) \uparrow (\mathsf{snd}\,A)))(\mathsf{inits}\,x))$

- inits phase:
  This is the same as the step 1.-4. of algorithm (8),(9) and (10).

- $\mathsf{map}\,(\lambda v.\,((\mathsf{fst}\,A) \uparrow (\mathsf{snd}\,A)))$ phase:

  6. each processor computes A, that is, makes a pair with 0 for each vector segment and folds with $\oslash$;

  7. each processor takes the maximum of each resulting pair in 6.

- $\mathsf{fold}\,(\uparrow)$ phase:

  8. each processor computes $\mathsf{fold}\,(\uparrow)$ for the result of 7;

  9. the local result in each processor is gathered to the master processor;

  10. synchronisation;

11. the master processor takes the maximum of the gathered local results.

Algorithm (12): $\mathsf{fold}\,(\uparrow)\,(\mathsf{shiftright}(0)\,(\mathsf{map}\,(\uparrow)\,(\mathsf{scan}\,(\oslash)\,(\mathsf{map}\,(\mathsf{pair}\,0)\,v))))$

- $\mathsf{map}\,(\mathsf{pair}\,0)$ phase:

  1. the contents of x in the master processor are scattered to the p processors;

  2. synchronisation;

  3. each processor makes a pair with 0 for local element.

- $\mathsf{scan}\,(\oslash)$ phase:

  4. each processor computes local scan with $\oslash$;

  5. the final element of the local scan in each processor is scanned in parallel across the processors with $\oslash$ using the obvious tree algorithm;

  6. the result of the global scan in the processor $i\,(< p)$ is sent to processor $i+1$;

  7. synchronisation;

  8. each processor applies $\oslash$ to the pair of the value sent to the processor in 6 and each element of the results of 4.

- $\mathsf{map}\,(\uparrow)$ phase:

  9. each processor takes the maximum element of each pair of the result of 8.

- $\mathsf{shiftright}\,(0)$ phase:

  10. $\mathsf{shiftright}\,(0)$ rotates the entire list right one place, moving a single element from each processor to the next and inserting 0 at the left end.

- $\mathsf{fold}\,(\uparrow)$ phase:

  11. each processor computes $\mathsf{fold}\,(\uparrow)$ for the result of 10;

  12. the local result in each processor is gathered to the master processor;

13. synchronisation;

14. the master processor takes the maximum of the gathered local results.

7.4.3 Predicted Results of Derivation Steps

We now look the predicted result for each derivation step that is computed by our cost calculator with the architecture parameters of our target machine. Experiments are made varying the input vector size n up to the value which is large enough to show significant differences of efficiency. Figure 7.1 plots the predicted results of (8) and (9) varying the input vector size n up to 160. It shows that the cost of (8) increases much faster than (9). Its complexity looks like $O(n^2)$ or faster. Figure 7.2 plots the predicted results of (9) and (10) varying n up to 224. In this case, (10) is a little more efficient than (9) but their plots draw similar curves. It seems that the difference of efficency would not change significantly even if n becomes large. The complexity of both algorithms looks like $O(n^2)$ or faster. Figure 7.3 plots the predicted results of (10) and (11) varying n up to 256. It reveals that complexity of (10) is $O(n^2)$ or faster as predicted above. The costs of (10) and (11) are similar up to about 100, but the difference of the costs becomes significantly large as we increase the value of n. (11) looks very efficient and its complexity looks linear, but it is not clear without checking the case when n is larger. Figure 7.4 plots the predicted results of (11) and (12) of varying n up to 2400. After $n = 400$, the cost of (11) increases rapidly drawing a curve which suggests that the complexity of (11) would be $O(n^2)$ or faster. (12) appears almost constant and its complexity looks linear, but it is not clear again without checking in the case when n is larger. Figure 7.5 shows the cost behaviour of (12) when n varies up to 400000. It appears that the complexity of (12) is linear.

Overall, we can predict not only the difference of order of complexity but also the size of input vector when the difference of order is beginning to be significant. In this example, $(8) \rightarrow (9)$ reduces cost dramatically even if n is very small. It implies that the redistribution cost caused by $\mathsf{foldconcat}$ between $\mathsf{map}\,(\mathsf{tails})\,(\mathsf{inits}\,x)$ and $\mathsf{map}\,(\mathsf{fold}\,(+))$ is very expensive. $(9) \rightarrow (10)$ seems to reduce the cost slightly, but does not depend on the value of n. $(10) \rightarrow (11)$ reduces the cost significantly when n is larger than about 100. $(11) \rightarrow (12)$ also reduces the cost significantly when n is larger than about 500.

The analysis time for (12) is a few seconds which does not depend on n. The analysis time for (8)-(11) at the largest value of n in the above experiment can take a few minutes, although it depends on the speed of the machine. Whether its efficiency is good enough for practical use would also depend on the range of problem sizes of interest as well as machine speed.

7.4.4 Accuracy Test

To test the accuracy of our cost analysis against performance on a real machine we hand compiled the BSP program in Oxford BSPlib for each of the five algorithms according to the implementations that are outlined in 7.4.2 and ran them on an 8-processor Sun HPC 3500. Following the same sequence of experiments using the same range of n as for the predictions, figure 7.6 - figure 7.10 plot predicted and real run times for (8) - (12) respectively for the range of n used for the predictions. The tables of difference between predicted and real run times also given in those figures. Although our calculator seems to tend to underestimate a little (except for the case (8)), those curves capture the characteristics of the behaviours of real run time costs. The reason why only the estimated cost for (8) overestimates might be that our cost calculator overestimates its communication costs which take a considerable part in the total cost when input data is large.

7.5 Experiments with Different Number of Processors

In order to show another use of our cost calculator, we examined the impact of the changing the number of processors. We predict the costs of algorithm (12) when $p = 1, 2, 4, 8$ and $n = 400000$, and then test the accuracy of the predictions, comparing to real run time costs. First we look again the implementation details of (12) specifying the case when $p = 1, 2, 4, 8$ individually.

Algorithm (12): $\mathsf{fold}\,(\uparrow)\,(\mathsf{shiftright}(0)\,(\mathsf{map}\,(\uparrow)\,(\mathsf{scan}\,(\oslash)\,(\mathsf{map}\,(\mathsf{pair}\,0)\,v))))$

when p=1:

- $\mathsf{map}\,(\mathsf{pair}\,0)$ phase:

  1. master processor makes a pair with 0 for the elements.

- $\mathsf{scan}\,(\oslash)$ phase:

  2. master processor computes sequential scan with $\oslash$.

- $\mathsf{map}\,(\uparrow)$ phase:

  3. master processor takes the maximum element of each pair of the result of 2.

- $\mathsf{shiftright}\,(0)$ phase:

  4. master processor inserts 0 at the left end of the result of 3.

- $\mathsf{fold}\,(\uparrow)$ phase:

  5. master processor computes $\mathsf{fold}\,(\uparrow)$ for the result of 4.

when p=2:

- $\mathsf{map}\,(\mathsf{pair}\,0)$ phase:

  1. the second half of contents of x in the master processor are sent to the worker processor;

  2. synchronisation;

  3. each processor makes a pair with 0 for the local elements.

- $\mathsf{scan}\,(\oslash)$ phase:

  4. each processor computes local scan with $\oslash$;

  5. the final element of the local scan in the master processor is sent to the worker processor;

  6. synchronisation;

7. the worker processor applies $\oslash$ to the data sent in 5 and the final element of the local scan in the worker processor;

8. the final element of the local scan in the master processor is sent to the worker processor;

9. synchronisation;

10. the worker processor applies $\oslash$ to the pair of the value sent to the processor in 8 and each element of the results of 4.

- $\mathsf{map}\,(\uparrow)$ phase:

11. each processor takes the maximum element of each pair of the result of 10.

- $\mathsf{shiftright}\,(0)$ phase:

12. master processor send the last element of the local result to the left end of the local result in the other processor and inserting 0 at the left end.

- $\mathsf{fold}\,(\uparrow)$ phase:

13. each processor computes $\mathsf{fold}\,(\uparrow)$ for the result of 12;

14. the local result in the worker processor is sent to the master processor;

15. synchronisation;

16. the master processor takes maximum of the local result in the master and the local result in the worker which was sent to the master in 14.

when p=4:

- $\mathsf{map}\,(\mathsf{pair}\,0)$ phase:

1. the contents of x in the master processor are scattered to the worker processors;

2. synchronisation;

3. each processor makes a pair with 0 for the local elements.

- $\mathsf{scan}\,(\oslash)$ phase:

4. each processor computes local scan with $\oslash$;

5. the final element of the local scan in each processor is scanned in parallel across the processors with $\oslash$ using the obvious tree algorithm;

6. the result of the global scan in the processor $i\,(< 4)$ is sent to processor $i+1$;

7. synchronisation;

8. each processor applies $\oslash$ to the pair of the value sent to the processor in 6 and each element of the results of 4.

- $\mathsf{map}\,(\uparrow)$ phase:

9. each processor takes the maximum element of each pair of the result of 8.

- $\mathsf{shiftright}\,(0)$ phase:

10. $\mathsf{shiftright}\,(0)$ rotates the entire list right one place, moving a single element from each processor to the next and inserting 0 at the left end.

- $\mathsf{fold}\,(\uparrow)$ phase:

11. each processor computes $\mathsf{fold}\,(\uparrow)$ for the result of 10;

12. the local result in each processor is gathered to the master processor;

13. synchronisation;

14. the master processor takes the maximum of the gathered local results of 12.

when p=8:
The description for this case is similar to that for the case when $p = 4$.

Next, we predict the cost of algorithm (12) when $p = 1,2,4,8$ using our cost calculator. BSP parameters used in the prediction are obtained by running the benchmark program provided with BSPlib on the Sun HPC 3500, and given in table 7.1. Note that as system conditions of our target system has changed since when the previous experiments were done, the values of these parameters when $p = 8$ are different from those for the previous experiments.

p	g (flops/word)	l (flops)	s
1	0.06	810	16
2	1.09	5479	16
4	1.37	25671	16
8	1.63	104230	16

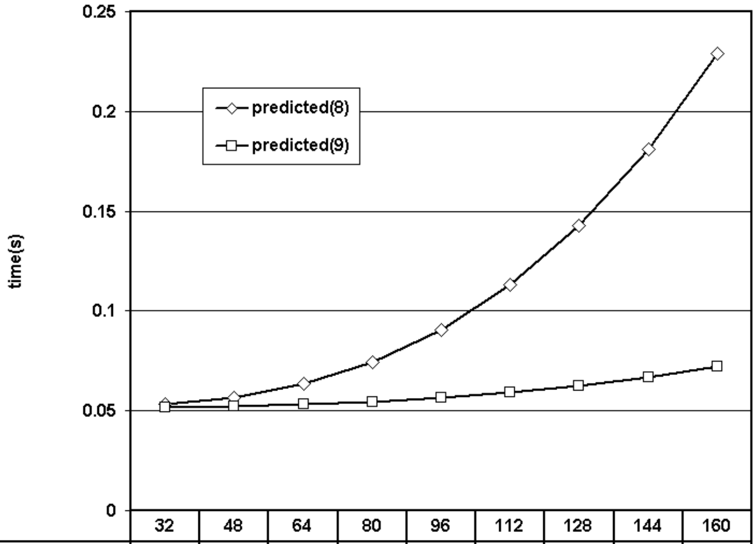
Table 7.1: BSP parameters when p = 1, 2, 4, 8

Figure 7.11 plots the predicted and real run time results of algorithm (12) when $p = 1, 2, 4, 8$. The table of difference between predicted and real run time is also given with the graph. The predictions when $p = 1$ and $p = 4$ are very close and the predictions are a little inferior when $p = 2$ and $p = 8$. As a result, the best case ($p = 4$) and the worst case ($p = 1$) in real run time results were successfully predicted from the results obtained by our cost calculator. The second best case ($p = 2$) and third best case ($p = 8$) in real run time results does not correspond with the predicted order partly because the calculator overestimated when $p = 2$ and underestimated when $p = 8$.

Although we would need to make more experiments and improve accuracy from the investigation of the results, the calculator seems to have promise as a tool to predict an optimal number of processors.

7.6 Chapter Conclusion

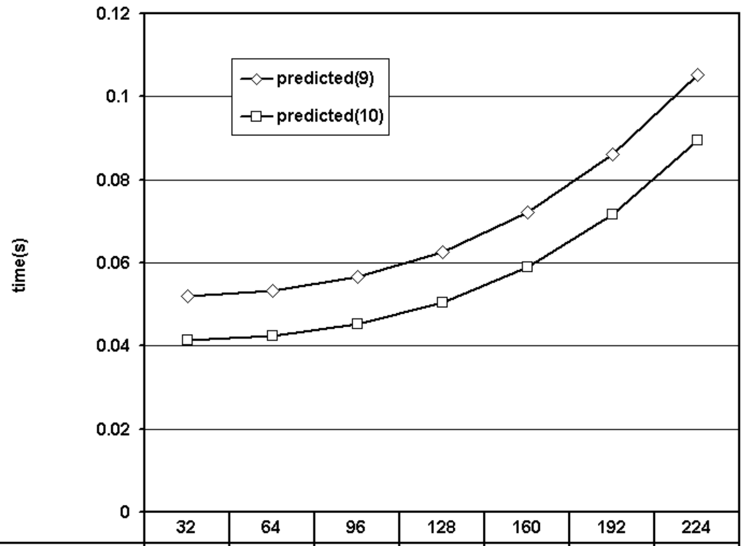
We have demonstrated the first example of application of our cost calculator to a complete algorithm derivation. It required a new shape expression in which two types of shape expression are mixed in order to express a shape which has different shaped elements while trying to avoid a large increase of the analysis time. It required us to introduce new operators such as inits and tails. Our cost analysis mechanism was augmented so that it can deduce the new shape expressions and costs automatically. Our cost calculator can automatically perform the analysis for arbitrary specified parameters. This allows us to make detailed comparisons of algorithms, which would be difficult in the traditional order analysis. The accuracy test of our predictions against real run time costs shows that the predictions are accurate enough to capture the behaviours of each run time cost when the input data size grows. Future development would include the improvement of both accuracy of prediction and efficiency of analysis cost, and the addition of more combinators which would enable us to apply the tool to a wide ranger of application problems.



	32	48	64	80	96	112	128	144	160
predicted(8)	0.05318	0.05666	0.06332	0.07421	0.09041	0.11291	0.14276	0.18106	0.22878
predicted(9)	0.05193	0.05241	0.05327	0.05461	0.05653	0.05913	0.06252	0.06681	0.07209

n

Figure 7.1: Comparison of predicted cost between (8) and (9)



	32	64	96	128	160	192	224
predicted(9)	0.051926	0.053274	0.056529	0.062525	0.072093	0.086065	0.105274
predicted(10)	0.041479	0.042485	0.045172	0.050375	0.058923	0.071651	0.089388

n

Figure 7.2: Comparison of predicted cost between (9) and (10)

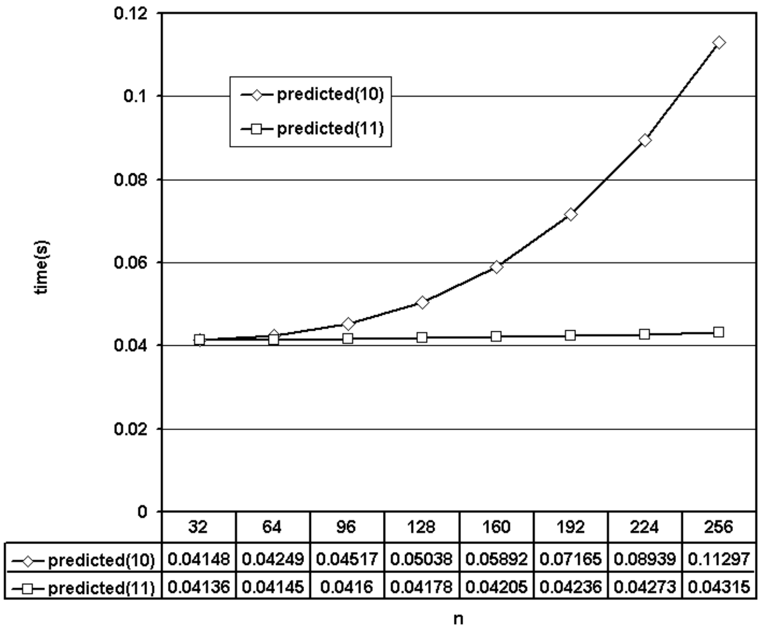


Figure 7.3: Comparison of predicted cost between (10) and (11)

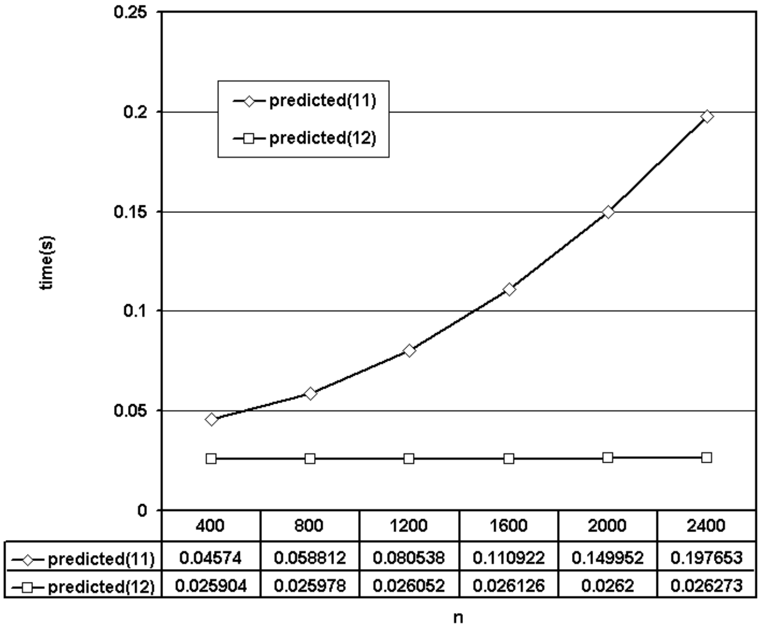


Figure 7.4: Comparison of predicted cost between (11) and (12)

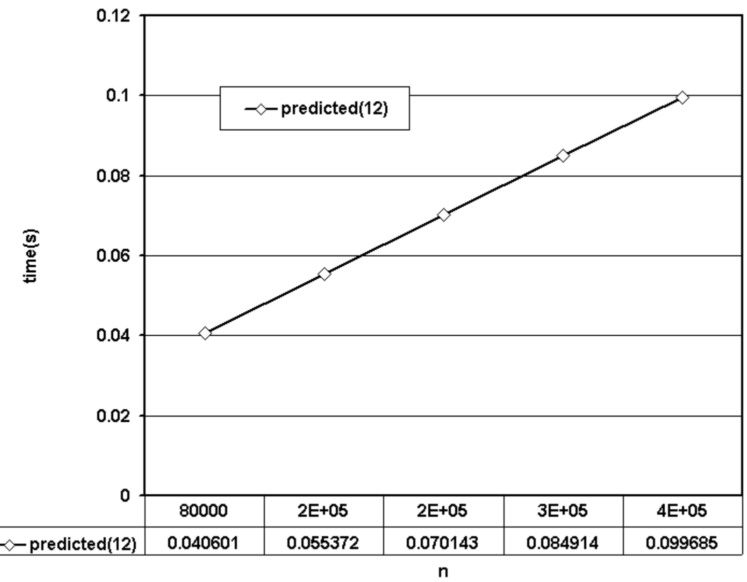
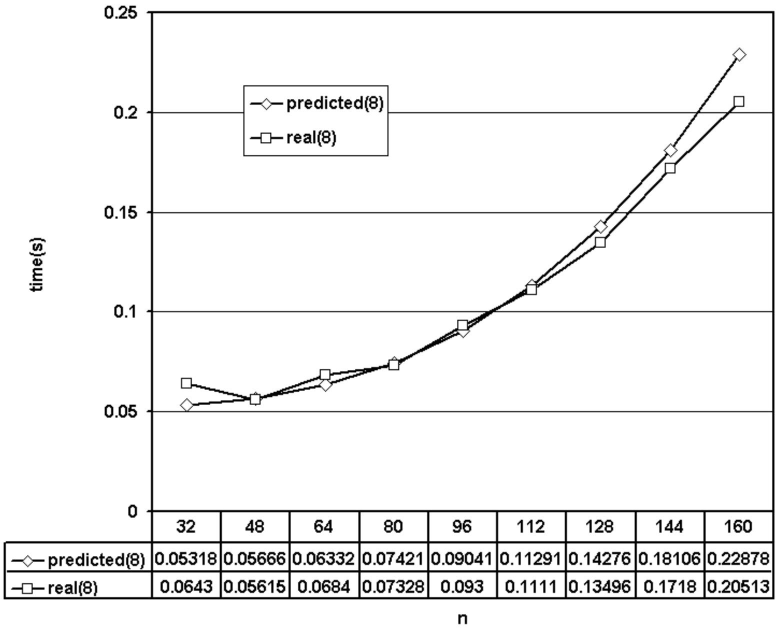
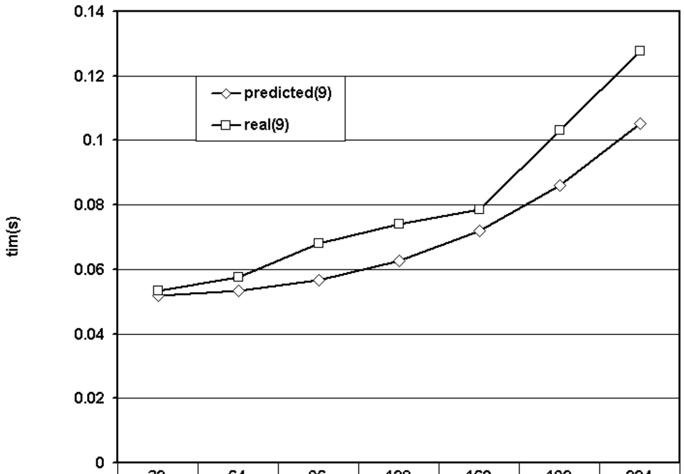


Figure 7.5: Predicted cost of (12)



n	32	48	64	80	96	112	128	144	160
real − predicted	0.01112	−0.0005	0.00508	−0.00093	0.00259	−0.000181	−0.0078	−0.00926	−0.02265

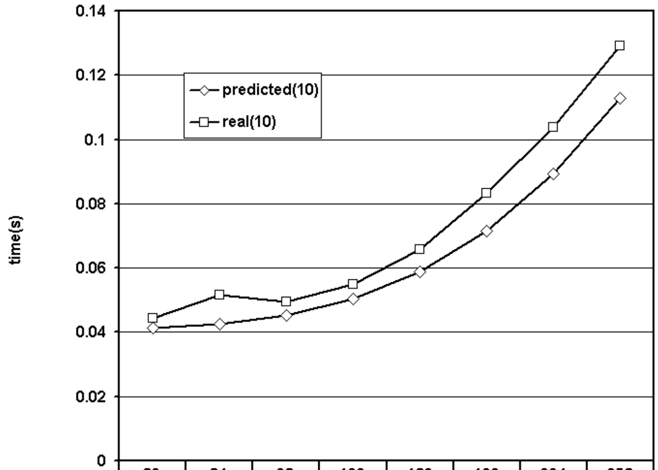
Figure 7.6: Comparison of predicted and real cost of (8)



	32	64	96	128	160	192	224
predicted(9)	0.051926	0.0532737	0.056529	0.062525	0.072093	0.086065	0.105274
real(9)	0.053412	0.057527	0.067914	0.074024	0.078654	0.102995	0.127829

n	32	64	96	128	160	192	224
real − predicted	0.001486	0.004249	0.011385	0.011499	0.006561	−0.01693	0.022555

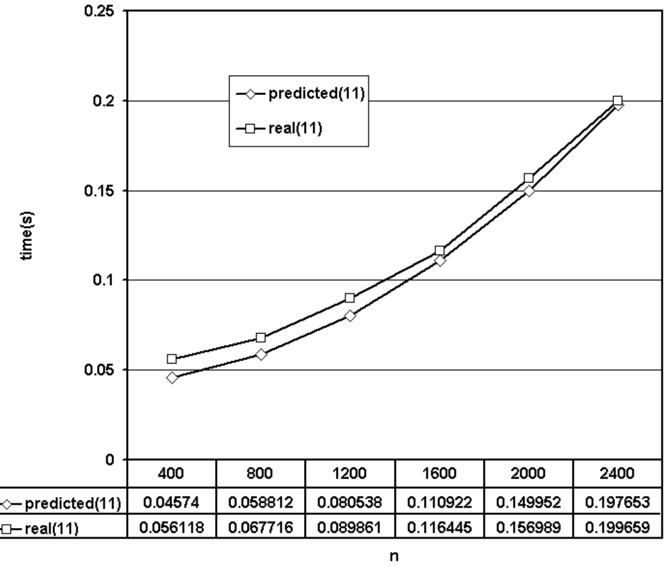
Figure 7.7: Comparison of predicted and real cost of (9)



	32	64	96	128	160	192	224	256
predicted(10)	0.04148	0.04249	0.04517	0.05038	0.05892	0.07165	0.08939	0.11297
real(10)	0.04431	0.05152	0.04951	0.05501	0.06564	0.08332	0.10371	0.12912

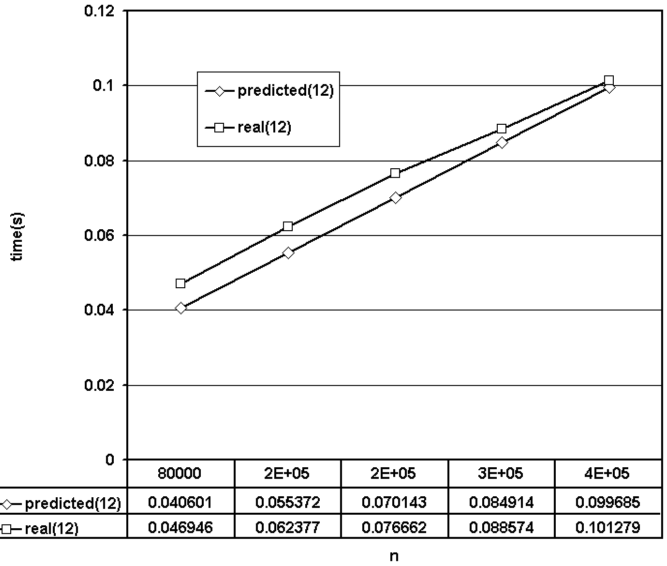
n	32	64	96	128	160	192	224	266
real − predicted	0.00283	0.01913	0.00444	0.01563	0.00772	0.01167	0.01432	0.01615

Figure 7.8: Comparison of predicted and real cost of (10)



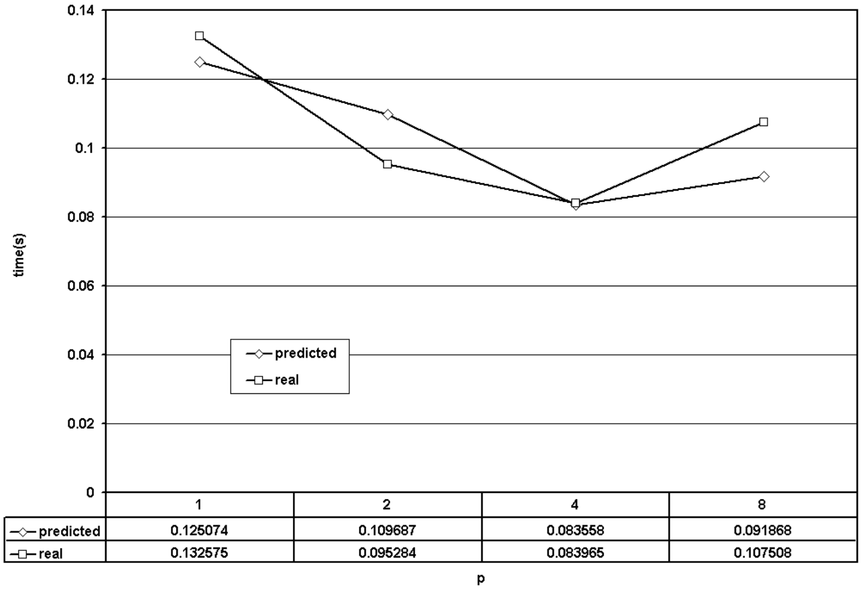
n	400	800	1200	1600	2000	2400
real − predicted	0.020378	0.008904	0.009323	0.05523	0.017037	0.002006

Figure 7.9: Comparison of predicted and real cost of (11)



n	80000	160000	240000	320000	400000
real − predicted	0.006345	0.007005	0.006519	0.00366	0.001594

Figure 7.10: Comparison of predicted and real cost of (12)



p	1	2	4	8
real cost − predicted cost	0.006501	− 0.014403	0.000407	0.01564

Figure 7.11: Predicted and real cost of (12) when p=1,2,4,8

Chapter 8

Summary and Directions for Future Research

8.1 Thesis Summary

We first give a summary of each chapter, and then summarise the whole thesis.

- **Chapter 1** explained the motivation for this work and gave a short overview of our analysis and a list of contributions of the thesis.
- **Chapter 2** first surveyed the main theoretical low level cost models for parallel programming. Next, we explained the concept of the "skeleton" methodology and investigated three works on cost analysis for BMF style parallel skeletal programming in detail. Then we gave a short survey of more related works.
- **Chapter 3** was the central part of the thesis which gave the definition of VEC-BSP, implementation strategy and cost analysis frame work.
- **Chapter 4** gave the BSP implementation templates of the built-in second order functions with their application costs, which complete the cost analysis presented in chapter 3.
- **Chapter 5** explained the Haskell implementation of our cost analysis.

- **Chapter 6** demonstrated that our analysis allows us to compare performance of alternative algorithms for the same problem in a concrete way. Comparisons between the predicted costs and the run times of the equivalent hand-compiled BSP programs on a real machine were also given.

- **Chapter 7** augmented our analysis framework to partially relax our strict requirements on data structure regularity. We demonstrated that the modified frame work allows the cost analysis of a derivation of the mss algorithm;

The initial motivation of this work was to modify the shape-based analysis that was presented in [55] so that it can account for communication cost as well as computation cost by changing the assumed target implementation model, because the communication cost which is incurred in real parallel computation could have a significant effect on the total execution cost. Since their analysis was based on the PRAM model which ignores communication cost, we tried to replace PRAM by BSP as the target implementation model because it has a simple, pragmatically accurate, and machine performance sensitive mechanism for costing the communication process. The main issue for a BSP implementation of VEC-BSP was to specify the placement and movement of data, while maintaining simplicity of the mechanism to compute cost automatically. The basic implementation structure has a nested structure for the implementation of the application term $t\,t'$ that consists of the four parts, $E_{t'}$: evaluation of the argument, E_t: evaluation of the function, C: communication part and A: application part. To compute the cost of the communication part with the BSP cost model, we need to determine the largest message size h sent or received by any processor. This requires information on the size of data placed in each processor, how the data is distributed among the processors, and how processors communicate the data. This is too complicated in the general case for automatic analysis. We addressed this issue by using one of the processors as the master processor and imposed the rule that the data of the result of each part is eventually stored in the master processor. This makes the data distribution and data size of the result in each part quite simple. The data size of the result is computed from the shape and added to the shape-cost pair as in the form of a tuple. We also need to determine the data distribution pattern at the beginning of the application parts. If the function is sequential, we use the master processor for the application part so that there is no communication in C, since the necessary data all resides in the master processor. For a parallel function, we addressed this issue by restricting the parallel application templates to those in which the data of the argument are always distributed evenly among the processors and all processors perform the same operation. Thus, the communication pattern in C in the case of parallel application is determined uniquely that is, the data of result of $E_{t'}$ is scattered to the processors evenly and that of $E_{t'}$ is broadcast to the processor. We added information on the application pattern to the shape of a function to indicate if the function is sequential or parallel. This new information, that is, message size and application pattern, is sufficient to determine h for the computation of the communication cost in C.

However, this strategy caused efficiency problems. When an application process ends by gathering the local results and it is the argument of another parallel function, just gathering for the next scattering is apparently redundant. To remedy this, we distinguished such application patterns from the others. The information on which application pattern was used to generate the intermediate results is recorded as a new component of cost tuple, data pattern, which is propagated by the information on the application pattern. Thus the analysis can detect redundant communication cases by checking the combination of the data pattern and the application pattern for each application and give the optimised cost.

The analysis has been implemented in Haskell. Automated cost analysis is useful especially for parallel programs because counting the number of instructions and deducing the shape and the message size of the intermediate results by hand is a complicated task. The analysis produces absolute value cost prediction based on the factors of benchmark results of the target architecture. Changing the target machine involves just the change of the values of the parameters without changing the source VEC-BSP program or the cost calculator program. In contrast with conventional order analysis, our analysis allows us to examine the cost behaviour on a particular range of the problem sizes or number of processors. It is possible to compare the efficiency of algorithms which have the same asymptotic complexity. Our experiments testing accuracy against real programs show that the predictions have enough accuracy to capture the trends of the cost behaviours for our example programs.

VEC and our initial shapely language impose restrictions on the uniformity of the element shape of the vectors. That is a key point which makes its shape expression concise and so the shape analysis time is much quicker than the execution time of the source program itself. However it is also a drawback when we try to apply our cost calculator to a BMF style program derivation since we often encounter an intermediate algorithm which does not satisfy this restriction. Our approach to address this issue was to prepare two kinds of shape expression. When the data has different shaped elements we express its shape as a vector. When the data has a common element shape we express its shape as a pair of its length and the element shape. The analysis distinguishes the kind of shape expression from its data type and generates an appropriate shape which is used to compute cost information. The efficiency of cost analysis depends on the degree of uniformity of the shape. This amendment allows our analysis to cost the complete derivation of Skillicorn and Cai's mss algorithm.

Finally, we assess how far our aims described in the introduction have been met by these achievements. Our initial motivation was to develop a cost analysis as an alternative to conventional asymptotic analysis (typically, for PRAM model) which suffers from

- lack of ability to cost communication
- difficulty in counting the number of instructions
- lack of ability to model the cost behaviour for a modest number of processors p and a particular range of problem sizes.

We chose Jay's shape-based cost analysis as the start point because its automatic and absolute value cost analysis has already solved the second and third problems. By incorporating the BSP approach to shape-based cost analysis, we achieved a communication sensitive shape-based cost analysis while keeping the characteristics of automatability and absolute value prediction. We also mentioned in chapter 1 that one of main problems of parallel computing is lack of portability as well as cost predictability. Although we have not yet achieved an implementation of the language, our language takes the skeletal approach, which enhances portability. The source language is implicitly parallel and assumed to be compiled to a BSP target which can be implemented on wide range of architectures through the existing communication libraries. Furthermore, as the impact of architecture change, including difference of communication performance characteristics, is reflected in our cost analysis results through the BSP parameters, our programming language could serve as a basis for the implementation of a programming language which is not only portable but also *performance portable*. Another aim was to use cost analysis to predict the effect of performance change in program transformation. An obstacle to this was that our source language is often too restrictive to express all intermediate algorithm in derivation steps. To alleviate the problem we partially relaxed our strong restriction on uniformity and showed the example of complete derivation of the mss program. Further research for this direction will involve applying our cost analysis to more examples and to identifying functions which are often used in the program transformational technique so that so that our language has a rich set of primitive function to express derivation steps. Improving the efficiency of the analysis developed in the work might be necessary.

8.2 Contributions of Thesis

The main contributions of this thesis can be summarised as follows.

- We demonstrated the first completely automated, communication sensitive shape-base cost analysis system for an implicitly parallel skeletal programming language of nested arrays. This builds on earlier work by Jay et al. [54] in the area by quantifying communication as well as computation costs, with the former being derived by changing the target implementation model from PRAM to BSP;

- We added several built-in second-order functions, each of which has a parallel implementation template and predefined application cost which is parameterised by the argument shape, in order to enhance the skeletal approach of parallel programming and to broaden applicability of our analysis;
- We extended Jay's shape-based analysis framework with cost tuples which contain useful static information as their components, and illustrated how this information is used for costing the communication process, optimising interface communication and eventually computing BSP cost.
- We partially relaxed our strict requirements on data structure regularity (but without losing static predictability) by introducing new shape expression in our analysis frame work;
- We presented the first analysis of a complete derivation, the well known *maximum segment sum* algorithm of Skillicorn and Cai;
- We illustrated skeletal programming in VEC-BSP by implementing several example programs. The accuracy of predictions made by our cost calculator against the run time of real parallel programs was tested experimentally.

8.3 Limitations

The main limitations of our analysis constitute the trade-offs involved in obtaining the static predictability and automatability of the analysis and a simple programming model that provides a high level of abstraction. These limitations can be summarised as follows.

- Only programs which are shapely, that is, for which the shape of the result is determined by the shape of the input can be expressed and analysed. Non-shapely function such as *filter* cannot be used.
- The program is expressed by combining predefined constructs. Although a number of types of data parallel programs can be expressed within this restriction,

some parallel algorithmic techniques cannot, or are difficult to express. This limitation of expressiveness could be partly alleviated if the language incorporated more useful functions as primitive functions. It would also possible to give the language a facility whereby the user can define a new function which cannot be composed from other function, but it requires the user to give information which is necessary for the cost analysis such as how it changes the shape of an input, the application cost and application pattern. It might be a considerable burden for the programmer.

8.4 Avenues for Future Research

The main future directions are summarised as follows

- **Improving Efficiency**
  We kept the assumed implementation mechanism deliberately simple in order to focus on its structural mechanism. Improving efficiency of the implementations of source programs would involve investigation of the possibility of different implementation templates for the built-in second order functions. Optimisation considering possible implementations of nested skeletons would also be possible. More complex communication patterns such as multi-cast would improve the efficiency of communication costs. Since improving the efficiency of the implementations of source programs tends to make the implementation mechanism more complicated, it would also increase the complexities of the cost analysis itself. Improving efficiency of both implementations of source programs and their cost analysis would require further refinement of analysis including choice of information components in the cost tuple and investigation of their interaction.

- **Improving Accuracy**
  Accuracy depends on how the g and l values experienced by the computation patterns and communication patterns used in an application program are matched by those in the benchmark program used to determine the BSP parameters (in other words how robust the BSP framework is itself). Although we used the

benchmark program provided with BSPlib, developing a benchmark program more suitable for the computation and communication patterns used in our more restricted computation model should further improve accuracy.

- **Testing More Application Problems**
  More example problems that have different parallel structures could be tested on the scheme. This would assist in identifying more useful functions to be added to the set of built-in functions.

- **Implementing the Language**
  An obvious important future work would be the construction of a source to source compiler to automatically generate C/BSP code from VEC-BSP according to the assumed implementation strategy using the static information which is extracted from the analysis system. To examine if our cost analysis can be applied when a source language is translated to a target language with other message passing communication libraries such as C/MPI would be interesting. As pointed out in [46], it is possible to program in a BSP style in MPI, although it has been found that such systems are rarely optimised for the small number of primitives that are necessary for BSP programming. It would be possible to develop a similar cost analysis if we can find a parameter which corresponds to g in BSP for broadcast and gather communication.

- **Application to Transformational Program Development**
  We demonstrated that our analysis technique is useful in the example of mss algorithm derivation steps. Tools to support the validation of transformation step already exist (e.g. [66]). Integration with automatic cost modelling would provide the programmer with immediate feedback on the performance implications of transformation decisions and could also assist with automated or semi-automated heuristic driven searches through the transformation space.

- **Future Research Beyond VEC-BSP**
  VEC-BSP and its shape-based cost analysis has limitations which come from intrinsic nature of static analysis, that is, it is not possible to predict the cost of programs which varys for different data values. This fact excludes the use of

non-shapely functions like filter and prevent the accurate prediction of programs which includes some kind of branch point for conditionals. More powerful language support by a cost analysis requires some extended environment beyond the static approach. The introduction of profiling techniques in the analyser could overcome the problem to some extent. Profiling could be used to capture run-time information at the points where shape and cost are dynamically determined. Effective cooperation of a static component and a dynamic component of a cost analyser would a key point for such a future parallel programming system.

Bibliography

[1] A. Abel. Specification and Verification of a Formal System for Structurally Recursive Functions. In *Types for Proofs and Programs, International Workshop, TYPES '99*, number 1956 in Lecture Notes in Computer Science, pages 1–20. Springer-Verlag, 2000.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LoGP: Incorporating Long Messages into the LogP Model. In *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architecture*, pages 95–105, 1995.

[4] B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and Transformations in an Integrated Parallel Programming Environment. In *Parallel Computing Technologies (PaCT-99)*, LNCS 1662, pages 13–27. Springer-Verlag, 1999.

[5] J. Backus. Can Programming be Liberated from the von Neumann Style? *Communications of the ACM*, 21(8):613–641, August 1978.

[6] O. Ballereau, F. Loulergue, and G. Hains. High Level BSP Programming: BSML and BSlambda. In G. Michaelson and P. Trinder, editors, *Trends in Functional Programming*, pages 29–38. Intellect, 2000.

[7] C. R. Banger. *Arrays with Categorical Type Constructors*. PhD thesis, Queen's University, Kingston, Ontario, Canada, 1992.

[8] G. Bilardi, K. T. Herley, A Pietracaprima, G. Pucci, and P. Spirakis. BSP vs LogP. In *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architectures*, pages 25–32, 1996.

[9] R. S. Bird. An Introduction to the Theory of Lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[10] R. S. Bird. Lectures on Constructive Functional Programming. *Constructive Methods in Computing Science*, volume F55 of *NATO ASI*:151–216, 1988.

[11] R. S. Bird. Algebraic Identities for Program Calculation. *the Computer Journal*, 32(2):122–126, 1989.

[12] G. E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[13] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[14] G. E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-95-170, Carnegie Mellon University, 1995.

[15] O. Bonorden, B. Juulink, I. von Otto, and I. Rieping. The Paderborn University BSP (PUB) Library-design, Implementation and Performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*, April 1999.

[16] G. H. Botorog. *High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms*. PhD thesis, Aachen University of Technology, 1998.

[17] G. H. Botorog and H. Kuchen. Efficient Parallel Programming with Algorithmic Skeletons. In *Euro-Par'96 Parallel Processing*, number 980 in Lecture Notes in Computer Science, pages 718–731. Springer-Verlag, 1996.

[18] T. A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, 1994.

[19] P. Brinch Hansen. *Studies in Computational Science*. Prentice Hall, 1995.

[20] A. Bundy, G. Grosse, and P. Brna. A Recursive Techniques Editor for Prolog. *Instructional Science*, 20:135–172, 1991.

[21] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam System. In M. E. Stickel, editor, *In Proceedings of 10th International Conference on Automated Deduction*, number 449 in Lecture Notes in Artificial Intelligence, pages 647–648. Springer-Verlag, 1990.

[22] R. M. Burstall. Proving Properties of Programs by Structural Induction. *The Computer Journal*, 12(1):41–48, 1969.

[23] R. M. Burstall. Inductively Defined Functions in Functional Programming Languages. *Journal of Computer and System Sciences*, 34(2/3):409–421, 1978.

[24] D. Busvine. *Detecting Parallel Structures in Functional Programs*. PhD thesis, Heriot-Watt University, 1993.

[25] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[26] M. Cole. Parallel Programming with List Homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.

[27] D. E. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, Subramoinian, and T. V. Eiken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. 10th symp. Theory Computing*, pages 114–118. ACM, 1993.

[28] M. Danelutto, R. D. Meglio, S. Orlando, S. Palagatti, and M. Vanneschi. A Methodology for the Development and the Support of Massively Parallel Programs. In D. Skillicorn and D. Talia, editors, *Programming Languages for Parallel Processing*, pages 205–220. IEEE Computer Society Press, 1994.

[29] J. Darlington, A. Field, P. Kelly, and R. Wu. Parallel Programming Using Skeleton Functions. In *PARLE'93*, number 694 in Lecture Notes in Computer Science, pages 146–160, Munich, June 1993. Springer-Verlag.

[30] J. Darlington, Y. K. Guo, H. W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In *Euro-Par'95 Parallel Processing*, number 966 in Lecture Notes in Computer Science, pages 55–69, Stockholm, August 1995. Springer-Verlag.

[31] H. Deldarie, J. R. Davy, and P. M. Dew. The Performance of Parallel Algorithmic Skeletons. In *Proceedings of ZEUS '95*, pages 65–74. IOS press, 1995.

[32] V. Dornic, P. Jouvelet, and D. K. Gifford. Polymorphic Time Systems for Estimating Program Complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, 1992.

[33] D. Feldcamp, H. V. Streekantaswamy, A. Wagner, and S. Chanson. Towards a Skeleton-based Parallel Programming Environment. In A. Veronis and Y. Paker, editors, *Transputer Research and Applications*, volume 5, pages 104–115. IOS press, 1992.

[34] D. Feldcamp and A. Wagner. Parsec - A Software Development Environment for Performance Oriented Parallel Programming. In S. Atkins and A. S. Wagner, editors, *Transputer Research and Applications*, volume 6, pages 247–262. IOS press, 1993.

[35] C. Foisy and E. Chailloux. Caml Flight: A Portable SPMD Extension of ML for Distributed Memory Multiprocessors. In *Workshop on High Performance Functional Computing*, pages 83–96, April 1995.

[36] S. Fortune and J. Wyllie. Parallelism in Random Access Machine. In *Conference record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, California, May 1978.

[37] G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[38] J. Gibbons. *Algebras For Tree Algorithms*. PhD thesis, University of Oxford, 1991.

[39] S. Gorlatch. Toward Formally-Based Design of Message Passing Programs. *IEEE Transactions on Software Engineering*, 26(3):276–288, 2000.

[40] Sergei Gorlatch, Christoph Wedler, and Christian Lengauer. Optimization Rules for Programming with Collective Operations. In Mikhail Atallah, editor, *IPPS/SPDP'99. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing*, pages 492–499, 1999.

[41] M. W. Goudreau, K. lang, S. B. Rao, T. Suel, and T. Tsantilas. Towards Efficiency and Portability: Programming with the BSP Model. In *Proceedings of 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[42] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(2):12–21, August 1993.

[43] M. Hamdan. *A Combinational Framework for Parallel Programming using Algorithmic Skeletons*. PhD thesis, Heriot-Watt University, 1999.

[44] Y. Hayashi and M. Cole. BSP-based Cost Analysis of Skeletal Programs. In G. Michaelson and P. Trinder, editors, *Trends in Functional Programming*, pages 20–28. Intellect, 2000.

[45] Y. Hayashi and M. Cole. Static Performance Prediction of Skeletal Programs. *Parallel Algorithms and Applications*, 2002. to appear.

[46] J. Hill. Portability of Performance in the BSP Model. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 267–287. Springer-Verlag, 1999.

[47] J. M. D. Hill, S. R. Donaldson, and D. B. Skillicorn. Portability of Performance with the BSPlib Communications Library. In *Programming Models for Massively Parallel Computers, (MPPM'97)*, pages 33–42. IEEE Computer Society Press, 1997.

[48] J. M. D. Hill, B. McColl, D. C. Stefanescu, K. Lang, S. B. Rao, T.Suel, T. Tsantilas, and R. Bisseling. BSPlib: The Programming Library. Technical Report

PRG-TR-29-97, Oxford University Computing Laboratory, 1997.

[49] W. A. Howard. The Formulae-as-types Notion of Construction. In J. P. Seldin and J. R. Hindle, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda calculus and Formalism*, pages 479–490. Academic Press, 1980.

[50] L. Huelsbergen, J. R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *LEF'94 - Conference on Lisp and Functional Programming*, pages 79–90. ACM Press, 1994.

[51] R. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive System Using Sized Types. In *POPL'96 - Symposium on Principles of Programming Languages*, pages 21–24. ACM Press, 1996.

[52] C. B. Jay. Shape Analysis for Parallel Computing. In J. Darlington, editor, *Proceedings of the fourth international parallel computing workshop: Imperial College London, 25–26 September, 1995*, pages 287–298. Imperial College/Fujitsu Parallel Computing Research Centre, 1995.

[53] C. B. Jay. Shape in Computing. *ACM Computing Surveys*, 28(2):355–357, 1996.

[54] C. B. Jay. Costing Parallel Programs as a Function of Shapes. *Science of Computer Programming*, 37:207–224, 2000.

[55] C. B. Jay, M. I. Cole, M. Sekanina, and P. A. Steckler. A Monadic Calculus for Parallel Costing of a Functional Language of Arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, number 1300 in Lecture Notes in Computer Science, pages 650–661, Passau, August 1997. Springer-Verlag.

[56] H. W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, University of Glasgow, 1997.

[57] H. W. Loidl and K. Hammond. A Sized Time System for a Parallel Functional Language. In *Glasgow Workshop on Functional Programming 1996*, 1996.

[58] F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect, 2001.

[59] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[60] D. Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), 1988.

[61] G. Michaelson, A. Ireland, and P. King. Towards a Skeleton Based Parallelising Compiler for SML. In C. Clack, K. Hammond, and T. Davie, editors, *Proceedings of 9th International Workshop on the Implementation of Functional Languages*, number 1467 in Lecture Notes in Computer Science, pages 539–546. Springer-Verlag, 1997.

[62] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing*, 16:181–206, 2001.

[63] R. Miller. A Library for Bulk Synchronous Parallel Programming. In *Proceedings of the BCS Parallel Processing Specialist Group Workshop on General Purpose Parallel computing*, pages 100–108, December 1993.

[64] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[65] J. M. Nash, P. M. Dew, M. E. Dyer, and J. R. Davy. Parallel Algorithm Design on the WPRAM Model. In J. R. Davy and P. M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, pages 83–102. Oxford University Press, 1995.

[66] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A Calculational Fusion System HYLO. In *IFIP TC2 Working Conference on Algorithmic Language and Calculi*, pages 76–106. Chapman&Hall, 1997.

[67] S. Pelagatti. *Structured Development of Parallel Programming*. Taylor & Francis, 1997.

[68] F. A. Rabhi. Exploiting Parallelism in Functional Languages: a Paradigm-Oriented Approach. In T. Lake and P. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, pages 118–139. Oxford University Press, 1993.

[69] F. A. Rabhi. A Parallel Programming Methodology Based on Paradigms. In P. Nixon, editor, *Transputer and Occam Developments*. IOS Press, 1995.

[70] F. A. Rabhi and J. Schwarz. POPE: Paradigm-Oriented Design of Parallel Programming Environment for SIT Algorithms. In J. G. W. Glauert, editor, *6th Workshop on the Implementation of Functional Languages*, Norwich, September 1994. Springer.

[71] R. Rangaswami. *A Cost Analysis for a High-order Parallel Programming Model*. PhD thesis, University of Edinburgh, 1996.

[72] J. Reed, K. Parrott, and T. Lanfear. Portability, Predictability, and Performance for Parallel Computing: BSP in Practice. *Concurrency: Practice and Experience*, 8(10):799–812, December 1996.

[73] B. Reistad and D. Gifford. Static Dependent Costs for Estimating Execution Time. In *LFP'94 - Conference on Lisp and Functional Programming*, pages 65–78. ACM Press, 1994.

[74] M. Rosendahl. Automatic Complexity Analysis. In *FPCA'89 - Conference on Functional Programming Language and Computer Architecture*, pages 144–156. ACM Press, 1989.

[75] N. Scaife, P. Bristo, and G. Michaelson. Engineering a Parallel Compiler for Standard ML. In *Proceedings of the 10th International Workshop on Implementations of Functional Language*, pages 213–225, 1998.

[76] D. B. Skillicorn. Architecture-independent Parallel Computation. *IEEE Computer*, pages 38–50, December 1990.

[77] D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[78] D. B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *Journal of Parallel and Distributed Computing*, 28(1):65–83, July 1985.

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

[80] A. J. Telford and D. A. Turner. Ensuring Termination in ESFP. *Journal of Universal Computer Science*, 6(4):474–490, 2000.

[81] H. W. To. *Optimizing the Parallel Behaviour of Combinations of Program Components*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, 1995.

[82] L. M. Tucker and A. Mainwaring. CMMD: Active Messages on the CM-5. *Parallel Computing*, 20(4), 1995.

[83] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[84] C. Walther. Argument-Bounded Algorithms as a Basis for Automated Termination Proofs. In *Proceedings of of the 9th International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 602–621. Springer-Verlag, 1988.

[85] J. Whittle, R. Boulton A. Bundy, and H. Lowe. An ML Editor Based on Proofs-as-Programs. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 166–173. IEEE Computer Society Press, 1999.

[86] A. Zavanella. *Skeletons and BSP: Performance Portability for Parallel Programming*. PhD thesis, University of Pisa, 1999.