Static Performance Prediction of Skeletal Parallel Programs

Yasushi Hayashi and Murray Cole
Institute for Computing Systems Architecture
Division of Informatics, University of Edinburgh
JCMB, King's Buildings, Edinburgh EH9 3JZ.

Abstract

We demonstrate that the run time of implicitly parallel programs can be statically predicted with considerable accuracy when expressed within the constraints of a skeletal, shapely parallel programming language. Our work constitutes the first completely static system to account for both computation and communication in such a context. We present details of our language and its BSP implementation strategy together with an account of the analysis mechanism. We examine the accuracy of our predictions against the performance of real parallel programs.

Keywords: Skeletons, shapes, cost modelling, BSP.

Computing Reviews Categories: D.1.3, D.3.3

1 Introduction

Deducing interesting dynamic characteristics of programs (and in particular, run time) is well known to be an intractable problem in the general case. In practice, programmers rely upon a combination of common sense, intuition and profiling to make the important algorithmic decisions which will affect performance. The situation is further complicated when parallelism becomes an option. 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 [3, 4, 12, 13] 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. A common theme has been the use of templates and their associated cost models to guide implementation and semi-automated performance prediction.

The shapely programming methodology [8, 9, 10] 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, even for programs in which shape is varied dynamically. This property is independent of parallelism but information on shape is fundamental to the computation of communication costs.

This paper investigates the use of both skeletal and shapely restrictions to produce a parallel programming language for which static performance prediction is completely automatable. 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 for an SPMD model, choosing BSP [7, 15, 17] to introduce parallelism because of its explicit support for dependably stable performance analysis across diverse architectures. Our work extends previously reported results in the field [6, 11].

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 [1, 5]. 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.

In section 2 we present an overview of our approach's structure and terminology, relating the work presented here to earlier papers. Section 3 presents our language, its implementation strategy and the definition of our analytic technique. This is completed in section 4 with the detailed implementations and costings of the skeletons. Section 5 demonstrates our analysis in action, comparing the performance of two algorithms for the same problem against each other and against the run time of equivalent hand compiled BSP programs on a real machine.

2 Overview and Terminology

Static shape analysis to support compilation and cost prediction for parallel programs was originally suggested by Jay [8] and first applied in detail to the cost analysis of VEC, a small shapely functional language [11]. This work 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). Preliminary work addressing this issue was presented by Hayashi and Cole in [6], with BSP replacing the PRAM. This required a number of amendments to the analytic framework and the assumed implementation mechanism (compiling VEC programs to BSP) was kept deliberately simple in order to focus on these structural changes. In [10], Jay sketches the application of skeletal and shapely techniques to the cost analysis of a parallel extension of the FISh programming language.

In this paper, we further develop the model of [6] and in particular capture the behaviour of an optimisation (concerning the re-distribution of intermediate data) which would be made by any realistic compiler. This requires further refinement of the analysis. We also add new operators (and their implementation skeletons) 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. Finally, for the first time in this research thread, we report on a comparison of predicted and actual performance.

In essence, our approach is a form of abstract interpretation. The application of a VEC-BSP program to real data is modelled by translating the VEC-BSP program into the closely related language MSIZE, representing the data term by a simpler term storing only its shape, then applying the former to the latter to produce a prediction of execution time. The translation strips out all data and data-oriented operations, replacing them with operations on shapes and machinery to gather the communication overheads which would be incurred by the real computation. We now outline the structure of this process.

Source Language

VEC-BSP is 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). VEC-BSP terms use standard functional terminology and have the expected semantics. A small number of built-in second order functions have parallel implementations. Its terms and types are discussed in more detail in section 3.1.

Target Language

MSIZE is essentially a restriction of VEC-BSP in which types, terms and operators which represent and manipulate real data have been removed, leaving only those which handle shapes. It also includes a small number of new components, not present in VEC-BSP, to allow manipulation of cost information. These are discussed in section 3.4.

Type Framework

Types of VEC-BSP terms are exactly those that would be 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) and types sz and un. sz is used to denote vector lengths, indices and other shape oriented quantities, while un denotes the shape of a primitive datum.

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, and that there are two forms of pairing, used to distinguish between pairs which are derived from pairs in the original VEC-BSP program and those which correspond to shape information about vectors (consisting of a pair recording vector length and element shape).

Evaluation costs are modelled as 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. The translation extends that which would be derived for a simple shape analysis which for data terms would remove primitive data, leaving only a condensed description of the structure, and for function terms would produce a function from the shape of the argument type to the shape of the result type. For example, the shape of a vector of vectors of integers would be a nested pair $(m, (n, \mathsf{un}))$ where m is the length of the outer vector and n the length of the inner vectors (which must all be the same, as discussed above).

Our translation augments the resulting terms with evaluation information. For data terms this involves adding

- a measure of the quantity of data which would have to be communicated to describe the term;
- an indication of the data distribution strategy required by the term's implementation (in order that communications between evaluation phases can be optimised);
- an evaluation cost function for the term, mapping from the BSP performance parameters to time (so that evaluation time for the term can be computed, given the performance characteristics of the specific target architecture).

Notice that this quite correctly means 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_1 and t_2 which evaluate to the same vector of length n. Suppose t_1 computes its result in parallel, while t_2 is entirely sequential. The costs of the terms will take similar forms $(((n, \mathsf{un}), n), (t, s))$ and $(((n, \mathsf{un}), n), (t', s'))$ indicating that both results have the same shape (n, un) and data size n. Meanwhile, the cost functions t and t' and distribution strategies s and s' are distinct, distinguishing the implementations.

For function terms we additionally annotate the translated term with an indication of the parallel application structure involved. When the function is applied, this is propagated as the data distribution strategy of the result.

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 3.4. Essentially, this must 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 function and arguments to compute these are of course bound up in the cost information for t and t' themselves.

The type information involved in translation is presented in section 3.4 and the translation itself is discussed in detail in section 3.5.

3 VEC-BSP Cost Analysis

3.1 VEC-BSP: A Shapely Skeletal Language

Our language is based on the small shapely functional language VEC defined in [11], with the addition of new parallel skeletons. We summarise its features here. The types are

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

where D can include other simple *datum* types, and the type 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. 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). A full list of constructors appears in section 3.5. VEC-BSP has four skeletal combinators which introduce parallelism:

- 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.

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

- 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 $\tilde{}0$ 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. This is enforced by statically checked shape rules. 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 iter combinator allows bounded iteration, controlling repeated application of a function to data. The number of repeats must be statically determined. We exclude unbounded iteration and recursion, since our goal is full automation.

3.2 BSP

In the BSP model [15] a parallel computer consists of three components: a set of processors each with a local memory, a communication network and a mechanism for globally synchronising the processors. A BSP program consists of supersteps each of which contains three phases: local computation, global communication and synchronisation. BSP has a cost model which is attractive by virtue of its conceptual simplicity and pragmatic accuracy. 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. The g and l values for several typical parallel computers are reported in [15]. The effects of contention and congestion on communication are captured in parameter g. When the communication pattern requires at most h words 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. The Oxford BSPlib [7] provides a BSP interface for the parallelisation of C programs.

3.3 A Vec-BSP Implementation Strategy

In our earlier paper we outlined a simple implementation strategy for Vec on the BSP computation model [6]. We now review this and explain its efficiency problem.

Because BSP has no shared memory the main issue is to specify placement and movement of data in the implementation while keeping things simple enough to predict cost automatically. We address this by using processor 0 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 parts: an evaluation of the argument t' which we call $E_{t'}$; an evaluation of the function t which we call E_t; a communication C, in which the data of the results of both component evaluations are redistributed for the next process followed by a barrier synchronisation; and A, an application, in which the result of E_t is applied to the result of $E_{t'}$. Nesting arises because $E_{t'}$ and E_t can themselves be application terms. The application phase A may be either *sequential* or *parallel*. A *sequential* application is executed only in the master processor, so there is no communication in C because the necessary data already resides in the master processor. A *parallel* application, dictated by the use of one of the parallel skeletons, requires that any data component of the result of E_t be broadcast to all processors. Note that we use a parallel implementation template in which all processors perform the same operation. The data describing the result of $E_{t'}$ is scattered to all processors evenly. Figure 1 illustrates the *scatter* and *broadcast* $-$ *scatter* communication patterns. In our execution diagrams time

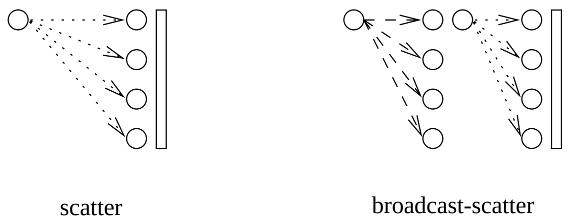

Figure 1: communication patterns

progresses from left to right with activities in a processor proceeding horizontally. Thus, 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. The narrow vertical box denotes machine wide synchronisation.

Combinators map and fold are typical examples of *parallel* application. 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 2 illustrates the application patterns, with solid lines indicating computation.

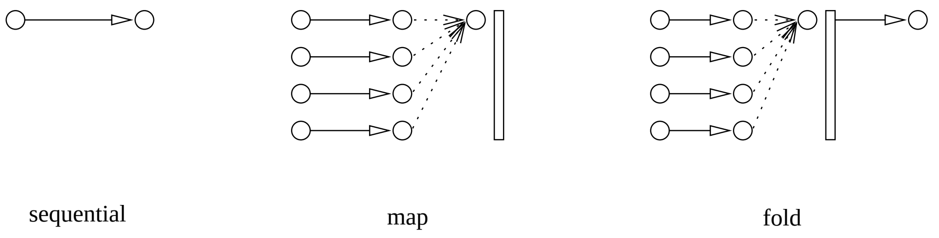

Figure 2: application patterns

We now introduce our communications optimisation. In [6] 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 were also always stored in the master processor, since these were either themselves nested parallel applications abiding by the same rule, or were already sequential. This rule simplified implementation by providing a common interface across nested terms. However, it also causes an efficiency problem. For example, if a parallel application process finishes by gathering results 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 illustrates the structure of such a computation, for a term of the form map f (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.

There are several possible resolutions. We could define several versions of a skeleton, with implementations differing only in data distribution and expect the programmer to choose one of them to optimise each C. This would make programming less abstract and more difficult. Similarly, we could predefine combining skeletons following [16]. Instead, we choose an automated route, demonstrating that our static analysis can be extended to analyse the interface between communication patterns. The next subsection describes how our model detects and resolves such inefficient cases. The lower half of figure 3 illustrates the resulting structure.

3.4 The Cost Translation Framework

The cost calculus in [11] computes PRAM cost and so does not model communication effects. In [6] we extended the analysis to detect inter-processor communication costs in terms of the BSP cost model. In this section, we show how the analysis is further extended to include

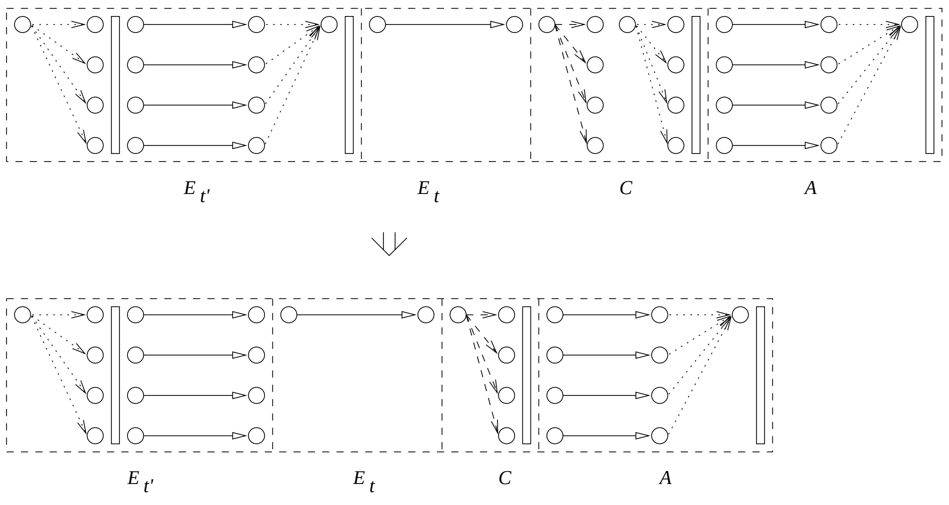

Figure 3: removal of unnecessary communication

information on communication patterns in order that these can be optimised.

As in earlier work, the translation is defined by a function *cost* which is now from Vec-BSP to Msize. The translated program, when run, will compute the shape of the result and the run-time cost of the corresponding compiled BSP program. For a Vec-BSP term $t : \theta$, the corresponding Msize term has type

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

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. T reflects the BSP cost of term t. The final sz reflects the *data pattern*, which we introduce in this paper to analyse the interface between communication patterns of individual skeletons.

The type translation $tycost_C$ is defined as follows:

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

We assign˜1 to the shape of D and un. The shape of a vector is a pair comprising its length and the common shape of its elements. Unlike Vec-BSP, Msize distinguishes between pairs which represent the shape of Vec-BSP pairs, denoted by $\langle\ ,\ \rangle$ and those which represent the shape of vectors, denoted $(\ ,\ )$, corresponding to $tycost_C(\theta) \times tycost_C(\theta')$ and $\mathsf{sz}\ \bar{\times}\ tycost_C(\theta)$ respectively.

The shape of a function is defined by attaching an application cost and an application pattern to the conventional shape, that is

$$\begin{aligned}
&tycost_C(\theta \rightarrow \theta') = tycost_C(\theta) \rightarrow C(tycost_C(\theta')) \\
&where, \quad C\theta = \theta \times (T \times \mathsf{sz})
\end{aligned}$$

The shape types $tycost_C(\theta)$ and $tycost_C(\theta')$ reflect the change of shape, T captures the application cost of the function and sz reflects the application pattern ($\tilde{}0$ for sequential, $\tilde{}1$ or 2 for parallel as discussed below). A term by term definition of *cost* is presented in the next section.

The information on data size, application cost and application pattern is added to allow calculation of communication costs. Communication occurs in two situations, firstly in C when a parallel pattern is used, and secondly in A when a parallel template is involved. In the first case, the communication cost is determined as a function of the number of words transmitted by processor 0, that is

$$s * (p - 1) * g$$

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

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

for scattering the s' words of the result of $E_{t'}$ to the worker processors. To compute s and s' from a shape expression we define an operator size:

$$\begin{aligned}\mathsf{size}\ \langle x, y\rangle &= \mathsf{size}\, x + \mathsf{size}\, y\\ \mathsf{size}\ (x, y) &= \mathsf{size}\, x * \mathsf{size}\, y\\ \mathsf{size}\ \tilde{}n &= \tilde{}n\end{aligned}$$

To optimise communication between component evaluation and A, the data distribution of the result of $E_{t'}$ and data distribution 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 the application process like map, from the other parallel patterns like fold. We label the former pattern $\tilde{}1$ and the latter pattern $\tilde{}2$. The *sequential* pattern is labelled $\tilde{}\ 0$. The data pattern is added to the MSIZE term to indicate which application pattern was used to generate t (with $\tilde{}0$ also denoting the case that t is primitive). The communication interface can then be optimised statically by removing unnecessary communication, using the information on the data pattern of the result of $E_{t'}$ and the application pattern of the function. The data pattern of the application result is set to be equal to the application pattern used to create it.

The cost of the application process, including this optimisation, is captured in our translation by the MSIZE function bspapp.

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

To improve the readability of its definition we introduce some additional notation. If x is the shape of a vector then the length of the vector and the shape of its elements are denoted $\mathsf{t\_len}\ x$ and $\mathsf{t\_eshp}\ x$ instead of $\mathsf{fst}\ x$ and $\mathsf{snd}\ x$ respectively. Similarly, if f is the shape of a function and x is the shape of an argument, then fx takes the form $\langle$*shape of result*, $\langle$*application cost*, *application pattern*$\rangle\rangle$. We use the shorthand $\mathsf{t\_shp}(fx)$, $\mathsf{t\_apcost}(fx)$, $\mathsf{t\_pattern}(fx)$, $\mathsf{t\_size}(fx)$ (which is equivalent to $\mathsf{size}(\mathsf{t\_shp}(fx))$) to represent the shape of the result, the application cost, the application pattern and the size of the result of fx respectively. The definition of bspapp is

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

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

$$\begin{aligned}
\mathsf{comm\_cost}\ x1\ x2\ x3\ x4\quad &=\tilde{}\ 0, && \mathit{if}\ x1=\tilde{}0\\
&=(x3*(p-1)-x4*((p-1)/p))*g-l, && \mathit{if}\ x2=\tilde{}1\\
&=(x3*(p-1)+x4*((p-1)/p))*g, && \mathit{otherwise}
\end{aligned}$$

in which data_sz captures the fact that if the result of an application is a function then there is no application cost and the message size of $t\ t'$ is just the sum of s and s'. When the result is not a function the message size is t_size $(f\,x)$. The time function for $t\ t'$ has four parts, namely the costs of the component evaluations T and T', the communication cost, the synchronisation cost l and the application cost t_apcost $(f\,x)$. Note that $+$ indicates point wise function addition in this context. The communication cost, captured by comm_cost, depends on the application pattern t_pattern $(f\,x)$ and the argument data pattern d'. If the application pattern is not $\tilde{}0$ and the data pattern is $\tilde{}1$ (result of a map), then our optimisation applies, allowing the communication cost for gathering the local results and the synchronisation at the end of the evaluation of the argument to be removed and the communication cost for the next scattering of the data to be omitted.

3.5 Detailed Cost Translation Rules

We now present the cost translation rules from VEC-BSP to MSIZE for basic term expressions and functions. The application cost functions for our parallel combinators are too complex to present in-line and so we give them names here (apcost_map and so on), presenting full definitions in section 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.

$$\begin{aligned}
&\mathit{cost}(d) && = && \langle\langle\,1,1\rangle\,,\langle 0,0\rangle\rangle\ \text{ where } d \text{ is a datum constant}\\
&\mathit{cost}(d) && = && \langle\langle\lambda x.\langle\lambda y.\langle 1,\langle\mathsf{binOpConst},0\rangle\rangle,\langle 0,0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle\\
&&&&& \text{where } d \text{ is a binary datum operation}\\
&\mathit{cost}(x) && = && \langle\langle x,\mathsf{size}\,x\rangle,\langle 0,0\rangle\rangle\\
&\mathit{cost}(\lambda x.t) && = && \langle\langle\lambda x.\langle\mathsf{fst}\ (\mathsf{fst}\ (\mathit{cost}(t))),\langle\mathsf{fst}\ (\mathsf{snd}\ (\mathit{cost}(t))),0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle\\
&\mathit{cost}(t\,t') && = && \mathsf{bspapp}\ \mathit{cost}(t)\ \mathit{cost}(t')\\
&\mathit{cost}(\mathsf{if}\ t\ \mathsf{then}\ \ t'\ \mathsf{else}\ t'')\\
&&& = && \mathsf{add}\ (\mathsf{mszmax}\,\mathit{cost}(t')\,\mathit{cost}(t''))\ (\mathsf{fst}(\mathsf{snd}\,\mathit{cost}(t)))\\
&&&&& \text{where }\ \mathsf{add}\,\langle\langle x,s\rangle,\ \langle t,d\rangle\rangle\,t'=\langle\langle x,s\rangle,\langle t+t',d\rangle\rangle
\end{aligned}$$

$$
\begin{array}{lcl}
cost(\mathsf{ifs}\ t\ \mathsf{then}\ \ t'\ \mathsf{else}\ \ t'') & & \\
 & = & \mathsf{add}\,(\mathsf{ifs}\,(\mathsf{fst}\,(\mathsf{fst}\,(t)))\ \mathsf{then}\ cost(t')\ \mathsf{else}\ cost(t''))(\mathsf{fst}(\mathsf{snd}\ cost(t))) \\
cost(\mathsf{length}) & = & \langle\langle\lambda x.\langle\ 1,\langle\mathsf{lengthConst},0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{entry}) & = & \langle\langle\lambda x.\langle\lambda y.\langle\mathsf{t\_eshp}\,x,\langle\mathsf{entryConst},0\rangle\rangle,\langle 0,0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{pair}) & = & \langle\langle\lambda x.\langle\lambda y.\langle\langle x,y\rangle,\langle\mathsf{pairConst},0\rangle\rangle,\langle 0,0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{hd}) & = & \langle\langle\lambda x.\langle\mathsf{t\_eshp}\,x,\langle\mathsf{hdConst},0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{tl}) & = & \langle\langle\lambda x.\langle(\mathsf{fst}\,x-1,\mathsf{snd}\,x),\langle\mathsf{tlConst},0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{fst}) & = & \langle\langle\lambda x.\langle\ \mathsf{fst}\,x,\langle\mathsf{fstConst},0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{snd}) & = & \langle\langle\lambda x.\langle\ \mathsf{snd}\,x,\langle\mathsf{sndConst},0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{map}) & = & \langle\langle\lambda f.\langle\lambda x.\langle(\mathsf{t\_len}\,x,\mathsf{t\_shp}\,(f\ (\mathsf{t\_eshp}\,x))),\langle\mathsf{apcost\_map}\,f\,x,1\rangle\rangle, \\
 & & \langle 0,0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{fold}) & = & \langle\langle\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),\langle 0,0\rangle\rangle(p-1)), \\
 & & \langle\mathsf{ap\_cost\_fold}\ \oplus\ x,2\rangle\rangle,\langle 0,0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
 & & \text{where }\ \mathsf{preiter}\,f\ \langle x,\langle t,d\rangle\rangle=\mathsf{add}_2\,(f\ x)\ t \\
 & & \qquad\quad \mathsf{add}_2\langle s,\langle t,d\rangle\rangle t'=\langle s,\langle t+t',d\rangle\rangle \\
 & & \qquad\quad iter\_rlt=\mathsf{iter}(\mathsf{preiter}(\oplus(\mathsf{t\_shp}(x))))\langle\mathsf{t\_shp}(x),\langle 0,0\rangle\rangle((\mathsf{t\_len}(x)/p)-1) \\
cost(\mathsf{pair\_map}) & = & \langle\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)))), \\
 & & \langle(\mathsf{ap\_cost\_pair\_map}\,f\,x,1\rangle\rangle,\langle 0,0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{c\_prod}) & = & \langle\langle\lambda f.\langle\lambda x.\langle\lambda y.\langle(\mathsf{t\_len}\,x,(\mathsf{t\_len}\,y,\mathsf{t\_shp}(f\ (\mathsf{t\_eshp}\,x)\ \ (\mathsf{t\_eshp}\,y)))), \\
 & & \langle\mathsf{ap\_cost\_c\_prod}\,f\,x\,y,1\rangle\rangle,\langle 0,0\rangle\rangle,\langle 0,0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle \\
cost(\mathsf{iter}) & = & \langle\langle\lambda f.\langle\lambda x.\langle\lambda y.\ \mathsf{iter}\,(\mathsf{preiter}f)\langle x,\langle\mathsf{iterConst},0\rangle\rangle\ y,\langle 0,0\rangle\rangle,\langle 0,0\rangle\rangle,0\rangle,\langle 0,0\rangle\rangle
\end{array}
$$

By way of elaboration, we now explain the expression for the cost of map. Working from the right hand end of the expression in, the pair $\langle 0,0\rangle$ indicates that it takes no time to evaluate the term map itself and data pattern of the term map itself is 0. 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 pair $\langle 0,0\rangle$ indicates that it similarly takes no time to apply map to a given function and that the application pattern involved is sequential (pattern 0). The 1 indicates that the application of map f to some data x uses the application pattern 1. The apcost_map computes the cost of such an application as the sum of computation time and communication time as given in the section 4. Finally, the $(\mathsf{t\_len}\,(x),\mathsf{t\_shp}\,(f\,(\mathsf{t\_eshp}\,x)))$ captures the shape of the resulting vector, which has the same length as x and an element shape reflecting the result of an application of f to an element of x.

The term for fold is similarly structured with $iter\_rlt$ denoting the result of the initial local folding phase. Notice that in the term for iter, items f, x and 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. Finally, in the translation of a data conditional, mszmax gives the maximum of two MSIZE terms, taking point wise maximum for functions and returning 2 for data pattern or application pattern terms.

4 Implementation and Costing of the Parallel Combinators

We now introduce our BSP implementation skeletons for the parallel combinators of VEC-BSP and deduce their cost functions to complete the definition of *cost*. In conventional BSP, computing the size of messages would be a programmer's task. Our cost analysis allows this to be automated. 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)[7].

4.1 map

map 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 [4] and plays a central role in other paradigms. Its implementation strategy was presented in section 3.3 and the corresponding SPMD pseudo-code is:

```
bsp_get(data describing f from P0);
bsp_get(local share of x from P0);
bsp_sync();
for each local item
   apply f to this local element of x;
bsp_put(result to P0);
bsp_sync();
```

while the application cost is:

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

Note that the application cost of map does not include the costs of the first two bsp_gets and the first bsp_sync. These are counted by the bspapp operation as a communication cost of C.

4.2 fold

Recall that fold combines the elements of a vector using an associative binary operator. The combination of map and fold forms the important "map and reduce" paradigm in BMF [14]. Its implementation was presented in section 3.3. The corresponding SPMD pseudo-code is:

```
bsp_get(data describing f from P0);
bsp_get(local share of x from P0);
bsp_sync();
for each local item
   combine this item into emerging local result;
bsp_put(local result to P0);
bsp_sync();
if (bsp_pid() == 0)
  sequentially fold together collected sub results;
```

The application cost of fold is

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

where $iter\_rlt = \mathsf{iter}(\mathsf{preiter}(\oplus(\mathsf{t\_shp}(x))))\langle \mathsf{t\_shp}(x), \langle 0,0\rangle\rangle((\mathsf{t\_len}(x)/p)-1)$, corresponding to the local folding on each processor. The iter combinator is required to model the repeated application of the $\oplus$ to allow for situations in which the resulting shape is not a simple multiple of the shape of the original elements.

4.3 pair_map

pair_map applies a function f to pairs of elements drawn from a pair of vectors of the same length. The implementation of the pair_map skeleton broadcasts the data in f and scatters the data in fst x and snd x (the two vectors) to all processors followed by synchronisation. f is then applied to elementwise pairs drawn from the segments of fst x and snd x in each processor. Finally local results are gathered to the master processor followed by synchronisation. In SPMD pseudo-code this is:

```
bsp_get(data describing f from P0);
bsp_get(local share of fst x from P0);
bsp_get(local share of snd x from P0);
for each local item from x
    apply f to (fst x) and corresponding local (snd y);
bsp_put(results to P0);
bsp_sync();
```

with an application cost of

$$\begin{aligned}& \mathsf{ap\_cost\_pair\_map}\, f\, x\,\langle p,\langle g,l\rangle\rangle \\ = \quad & \mathsf{t\_apcost}(f(\mathsf{t\_eshp}(\mathsf{fst}\ x))(\mathsf{t\_eshp}(\mathsf{snd}\ x))) * (\mathsf{t\_len}(\mathsf{fst}\ x)/p) \\ & + \mathsf{t\_size}(f(\mathsf{t\_eshp}(\mathsf{fst}\ x))(\mathsf{t\_eshp}(\mathsf{snd}\ x))) * (p-1) * g + l\end{aligned}$$

4.4 c_prod

c_prod applies a function to each member of the cross-product of the elements of two vectors x and y. It is used for a class of algorithms in which each object interacts with every other and corresponds to Brinch Hansen's All-Pairs Paradigm [2] and Darlington's RaMP (Reduce-and-Map-over-Pairs) skeleton [4].

The implementation of c_prod proceeds by broadcasting the data in f and x and scattering the data in y to all processors followed by synchronisation. Next, in each processor, f is applied to the all members of the cross product of x and the local segment of y. Finally the local results are gathered to the master processor, followed by synchronisation. In SPMD pseudo-code this is:

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

with an application cost of

$$\begin{aligned} & \mathsf{ap\_cost\_c\_prod}\, f\, x\, y\, \langle p, \langle g, l\rangle\rangle \\ = \; & \mathsf{t\_apcost}(f(\mathsf{t\_eshp}(x))(\mathsf{t\_eshp}(y))) * (\mathsf{t\_len}(y)/p) * (\mathsf{t\_len}(x)) \\ & +(\mathsf{t\_size}(f(\mathsf{t\_eshp}(x))(\mathsf{t\_eshp}(y))) * (\mathsf{t\_len}(y)/p) * (\mathsf{t\_len}(x)) * (p-1)) * g + l \end{aligned}$$

5 Implementation and Experiments

In this section we describe our experimental framework for automatic cost prediction. We consider two algorithms for matrix-vector multiplication 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.

5.1 Automating the Cost Analysis

Our cost calculus has been implemented as a Haskell program based on that developed for the PRAM by Jay's group [11]. It consists of nearly one thousand lines of Haskell and takes a few seconds to analyse the short examples presented here.

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

$$\mathsf{dummyvec} : (\mathsf{sz} \times \tau) \rightarrow \mathsf{vec}\ \tau$$

which allows the programmer to directly specify the input shapes. This would be replaced by calls to IO operations in the executable program. The cost function for $\mathsf{dummyvec}$ is simple, as the programmer provides the shape directly, while the associated costs are all zero.

$$\mathrm{cost}(\mathsf{dummyvec}) = \quad \langle\langle \lambda x. \langle\, x, \langle 0, 0\rangle\rangle, 0\rangle, \langle 0, 0\rangle\rangle$$

5.2 Example Programs

In this section, we show how our tool can be used to compare the behaviours of different algorithms for a simple problem using real BSP machine parameters. The example problem is a matrix vector multiplication Mv, where M is $m \times n$ 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 sections 3.3 and 3.4 is applicable in the second algorithm. 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. The first algorithm expressed in Vec-BSP is:

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

where, $v = \mathsf{dummyvec}(n, 1)$ and $M = \mathsf{dummyvec}(m, \mathsf{dummyvec}(n, 1))$ for analysis purposes. The parallel structure of algorithm (1) is illustrated in figure 4.

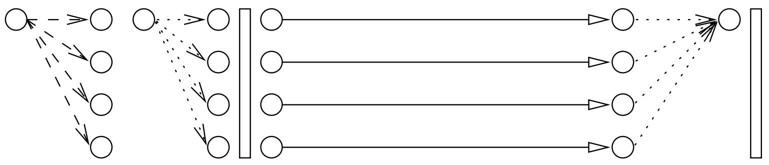

Figure 4: Parallel structure of algorithm (1)

The function $\lambda x.(\mathsf{fold} + (\mathsf{pair\_map} * (\mathsf{pair}\, v\ x)))$ takes a vector and returns its inner product with v. The data size of this function term is the size of v. The elements of v in the master processor are broadcast to the p processors. M's contents, consisting of mn integers in the master processor, are scattered to the p processors in vector-block-wise manner followed by synchronisation. Each processor computes the inner product of v and each distributed vector. Finally, the local result on each processor is gathered to master processor.

An intuitive BSP cost analysis of (1) is made by counting the number 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. The 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:

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

where, $L = M^t = \mathsf{dummyvec}(n, \mathsf{dummyvec}(m, 1))$ and $v = \mathsf{dummyvec}(n, 1)$ as before. The implementation of this skeletal program has two parallel phases as illustrated in figure 5.
In the first, L and v are scattered to the p processors followed by synchronisation. The effect of the application of $\mathsf{pair\_map}\, (\lambda x\, y.\mathsf{map}\, (\lambda z.(y * z))x)$ to $(\mathsf{pair}\, L\, v)$ is that each element of i^{th}

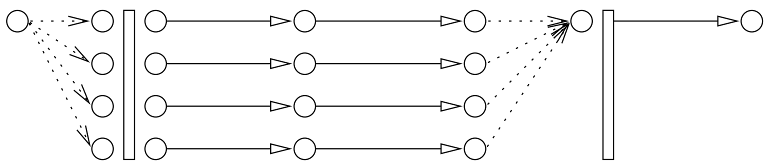

Figure 5: Parallel structure of algorithm (2)

vector of L is multiplied by i^{th} element of v. The communications implied by the gather step in this phase and the broadcast-scatter step in the next can be optimised away by our analysis, leading directly to the computation step of the second phase, fold $(\lambda x\, y.(\mathsf{pair\_map}+ (\mathsf{pair}\, x\, y)))$, in other words element-wise addition of all local vectors, gathering of the local results to the master processor, and then element-wise addition of the gathered vectors on the master processor.

An intuitive BSP cost analysis of (2) reveals that 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 $2l$ as before.

We now apply our cost calculator to the two algorithms. As a concrete example we investigate the case in which $p=8$ and M is square (ie $m=n$), with m varying. Figure 6 shows the predicted result of varying m in increments of 200 up to 1200. We can see that the

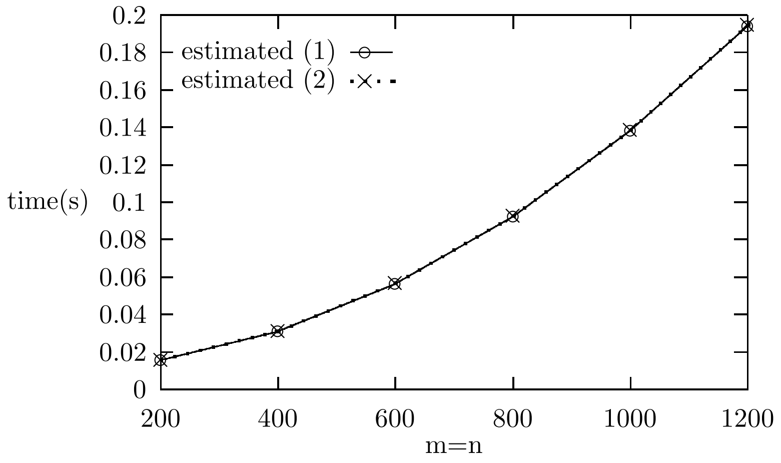

Figure 6: prediction when m=n, p=8

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, which predicts 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 same when $m=n$. Therefore, while the BSP cost complexity of both programs is $O(m^2)$, the actual difference in BSP cost of $m(p+1/p-2)$ has complexity of $O(m)$. This means that the difference is not significant when $m(= n)$ is large.

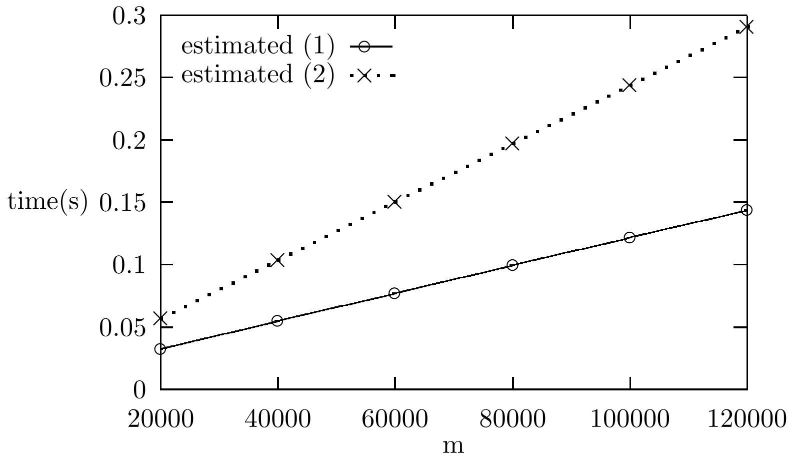

Figure 7: prediction when n=8, p=8

Now we consider the case in which n is fixed and m varies. Is there any significant difference in efficiency between (1) and (2)? Figure 7 shows the costs predicted by our calculator when

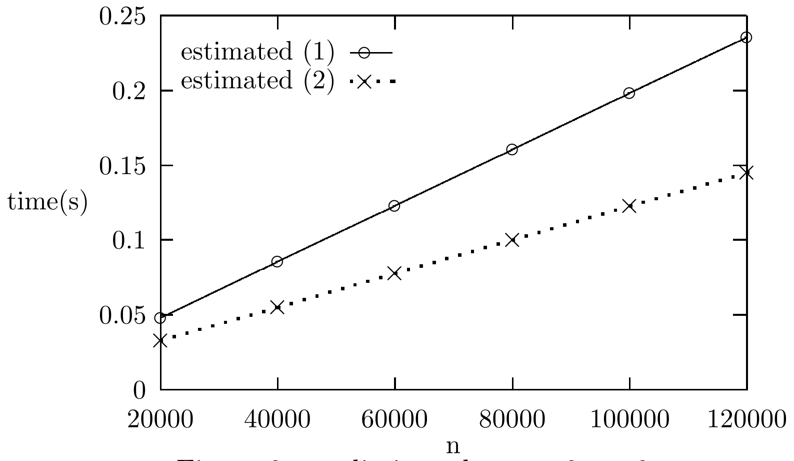

Figure 8: prediction when m=8, p=8

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 $O(m)$, the overall difference of costs also has complexity $O(m)$. This could be significant, and the results from figure 7 predict that this is indeed the case.

Finally we consider the case when m is fixed and n is varied. Figure 8 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 above, both algorithms 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 $O(n)$. As before this could be significant and the predications of figure 8 again show this to be the case.

Ad-hoc analysis is a hard task even for a simple algorithm. Our cost calculator can automatically perform the analysis of 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.

5.3 Comparison with Real Programs

To test the accuracy of our static cost analysis against performance on a real machine we hand compiled BSP programs in Oxford BSPlib for the two algorithms above and ran them on an 8-processor Sun HPC 3500.

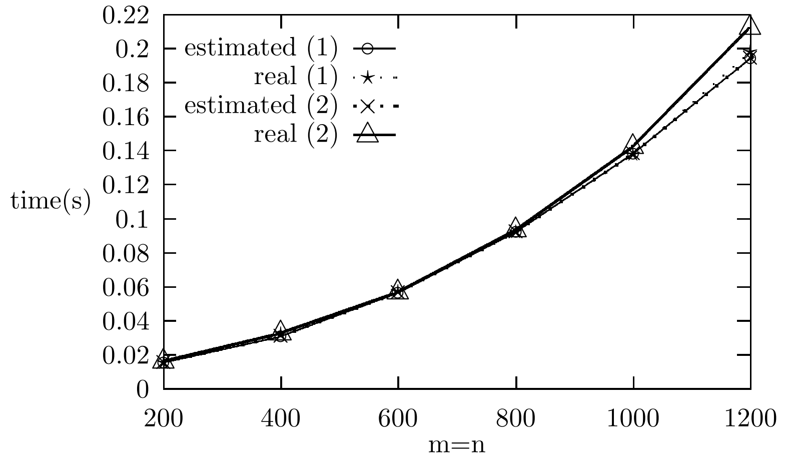

Figure 9: accuracy when m=n, p=8

Following the same sequence of experiments as for the predictions, figure 9 plots predicted and real run times for both programs when $m = n$, figure 10 plots times when n is fixed and m varies, and figure 11 plots times when m is fixed and n varies. In five out of six cases, real and predicted curves are very close.

Accuracy is inferior in the case of algorithm (2) when n is fixed (the upper two curves in figure 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.

Overall, 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.

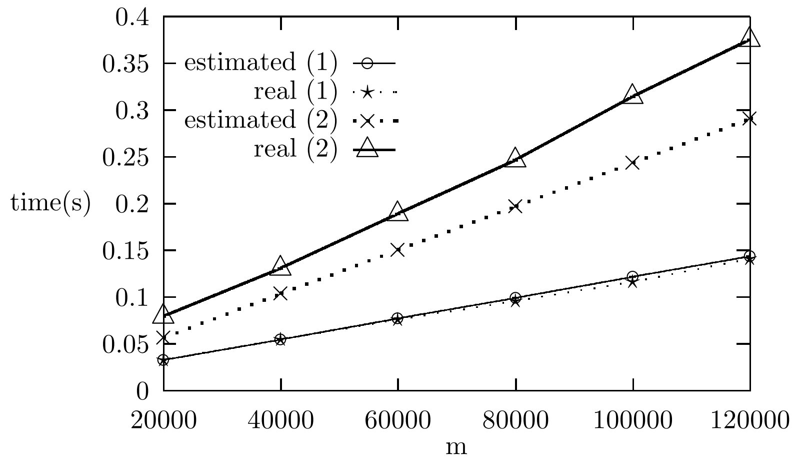

Figure 10: accuracy when n=8, p=8

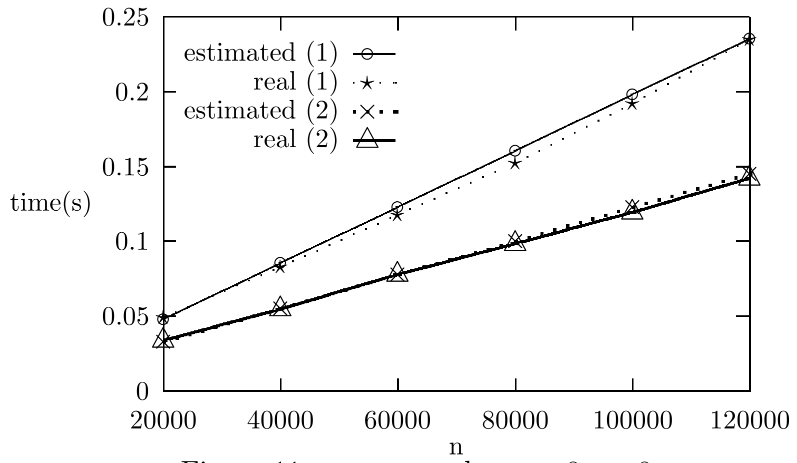

Figure 11: accuracy when m=8, p=8

6 Future Work

We have demonstrated the first completely automated, communication sensitive analysis of a skeletal parallel programming language. A number of opportunities for further development now present themselves. Firstly, there is scope for incremental improvement of our cost calculator. This could involve the addition of new skeletal combinators, increased sophistication of the modelled implementation mechanisms and consideration of further optimisations. Secondly, the VEC-BSP to C/MPI translation could be automated with a source-to-source compiler. Finally, we hope that our analysis technique will prove useful in the application of transformational program development methodologies. Tools already exist to support the validation of transformation steps. 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.

7 Acknowledgement

We 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 us with the source code for the original PRAM cost calculator from which our own system derives.

References

[1] Bruno Bacci, Sergei Gorlatch, Christian Lengauer, and Susanna Pelagatti. Skeletons and transformations in an integrated parallel programming environment. In *Parallel Computing Technologies (PaCT-99)*, LNCS 1662, pages 13–27. Springer-Verlag, 1999.

[2] P. Brinch Hansen. *Studies in Computational Science.* Prentice Hall, 1995.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[4] 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, 1993. Springer.

[5] Sergei Gorlatch. Toward Formally-Based Design of Message Passing Programs. *IEEE Transactions on Software Engineering*, 26(3):276–288, 2000.

[6] Y. Hayashi and M. Cole. BSP-based Cost Analysis of Skeletal Programs. In G. Michaelson and P. Trinder, editors, *Trends in Functional Programming.* Intellect, 2000.

[7] 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.

[8] 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.

[9] C. B. Jay. Shape in Computing. *ACM Computing Surveys*, 28(2):355–357, 1996.

[10] C. B. Jay. Costing Parallel Programs as a Function of Shapes. *Science of Computer Programming*, 37(1–3):207–224, 2000.

[11] 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.

[12] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications*, 2001. To appear in a special issue on High Level Models and Languages for Parallel Processing.

[13] S. Pelagatti and M. Danelutto. *Structured Development of Parallel Programs.* Taylor & Francis, 1997.

[14] D. B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

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

[16] H. W. To. *Optimizing the Parallel Behaviour of Combinations of Program Components.* PhD thesis, Department of Computing, Imperial College, 1995.

[17] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.