

AUTOMATED COST ANALYSIS OF A PARALLEL MAXIMUM SEGMENT SUM PROGRAM DERIVATION

YASUSHI HAYASHI and MURRAY COLE

*Institute for Computing Systems Architecture,
Division of Informatics, University of Edinburgh, UK
{yhay,mic}@dcs.ed.ac.uk*

Received September 2001

Revised January 2002

Accepted by C. Lengauer

Static performance prediction of implicitly parallel functional programs can be facilitated by restricting the source language to be *shapely* [7]. The resulting analyses should provide valuable support for the calculational style of program derivation. We build upon previous work in the area by extending the range of admissible programs, allowing us to demonstrate the first automated analysis of a complete program derivation, that of the well known *maximum segment sum* algorithm of Skillicorn and Cai [11]. We examine the accuracy of our predictions against the run time of real parallel programs.

Keywords: Shape analysis; static performance predictions; program derivation.

1. Introduction

The Bird-Meertens Formalism (BMF) has been advocated as a high level computation model to support the calculational style of parallel program derivation [9]. While support tools for the formal methodology of derivation steps have been actively investigated, the corresponding prediction of the cost of derived programs has been left to the programmer's intuition and at a level of abstraction at which it is difficult to compare the details of cost behaviour among equivalent programs. Static cost analysis of arbitrary programs (whether sequential or parallel) is well known to be intractable. However, when a program is expressed as a composition of suitably constrained *shapely* functions (in the sense that the application cost and shape of the output depends on only the shape of the input rather than being data value dependent) then static analysis becomes possible, essentially by tracking intermediate shape information through what is effectively a process of abstract interpretation.

An early example of using such a cost calculus to guide BMF style derivation was presented by Skillicorn [9, 11]. This informal strategy used intermediate *shape vector* information, deduced by hand, and demonstrated an asymptotically costed *maximum segment sum* (*mss*) program derivation. Subsequent work by Jay [7, 8]

formalised the concepts of shape and cost to allow analysis which is both fully automated and sensitive to the constant factors associated with specific machine characteristics. Recently the authors extended Jay’s framework to give a more detailed and realistic account of communication costs [4, 5] targeting the BSP model [10] for portability. Unfortunately the imposed shape restrictions excluded some intermediate programs which occur in the standard *mss* derivation. In particular, all elements of a vector (the nestable bulk data structure) were required to have the same shape. This made shape expression concise and consequently made automated analysis fast. However, while many practical algorithms (for example in linear algebra) can be expressed within this class, it precludes the use of the standard BMF functions *inits* and *tails* which are common in the early stages of derivations. Noting that the irregularity of these functions is entirely shapely, in the sense of being statically predictable, this paper introduces an extended shape analysis relieving the constraint (but preserving shapeliness) while trying to avoid large increases in analysis cost. This allows us to statically cost analyse all steps of the well known derivation, demonstrating the kind of support which could be made available to programmers in an integrated program development system. It is a well known phenomenon that intermediate derivation steps which increase costs are sometimes required to facilitate subsequent cost reducing steps. Such insights remain the preserve of the programmer’s intuition. Our system would simply act as an assistant, indicating what was happening to costs. As a by product, we also demonstrate the simplicity with which our framework admits new operators. The structure of our analysis means that the effects of these additions are neatly modular.

2. Previous Work in Automated Shapely Cost Analysis

Shapely cost analysis as suggested by Jay [7, 8] aims to translate terms of the source program (using a translation function *cost*) into terms of an analysis program which when executed computes the shape and cost of running the original program. The shape of a datum is defined inductively as an “array length and element shape” pair for non-function terms. For example, the shape of a basic datum constant is determined as *un* (suggesting an indivisible unit) and those of $[1, 4, 2, 3]$ and $[[1, 4], [2, 3], [4, 7]]$ are defined as $\langle 4, \text{un} \rangle$ and $\langle 3, \langle 2, \text{un} \rangle \rangle$ respectively. The shape of a function term is defined as a function from the shape of the argument to a pair of the shape of the result and application cost: *shape of argument* \rightarrow \langle *shape of result*, *application cost* \rangle so that applying it gives the shape of the result and the time involved (both of which are determined by the argument shape and the assumed implementation structure). The cost of a term is accumulated as an element of a *cost algebra* which captures the performance model of a chosen target architecture. For example, for the PRAM this is a function from the number of processors p to time t measured in PRAM steps. During execution of the translated program, shape computation is performed in the first component of a pair and cost is accumulated in the second. This process is captured by the operation *capp* (for “cost of application”)

in the analysis program:

$$\text{capp } \langle f, t \rangle \langle x, t' \rangle = \langle \text{fst}(fx), \text{snd}(fx) + (t \oplus t') \rangle$$

where $\langle f, t \rangle$ are respectively the shape (as a function as explained above) and cost of evaluating the function term to be applied and $\langle x, t' \rangle$ are the analogous shape and cost of the argument to the application. Thus `capp` says that the cost of an application term is a composition of application cost $\text{snd}(fx)$, the cost of the function term t and the cost of the argument term t' . $+$ and \oplus belong to the chosen cost algebra and can be changed to reflect the underlying execution cost model. The parallel cost model in [8] was the PRAM, an abstract model which takes no account of the communication and contention cost incurred on realistic parallel machines.

Our previous work [4, 5] built upon the foundations of [8] by adopting BSP as the parallel target model in order to benefit from its portable but pragmatic attention to cost details, especially communication cost. This required both extensions of and changes to the structure of the analysis. Most notably, a number of new concepts were added to the intermediate information and the details of the cost algebra became interwoven with the analytic machinery. We now briefly review this work in order to place our new results in context. The reader is referred to the corresponding papers [3, 4, 5] for a complete exposition.

Our original BSP analysis framework models an implementation in which one of the processors is used as a “master processor” storing the necessary data at the beginning of computation and the result at the end. A complete computation of a program $t t'$ has a nested structure consisting of four ordered 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. We assume 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.

The corresponding analysis uses more complex tuples called *cost tuples*, adding more evaluation information to the original shape-cost pairs of [8]. These take the form

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

with the shape of a function being a function from the shape of the argument to a similarly expanded *application tuple*,

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

Data size is a measure of the quantity of data required to represent an evaluated term. It can be computed from the term's shape and is used to compute communication cost. The application and data patterns are indications of the data distribution strategy required by the term's evaluation and are recorded in order that communications between evaluation phases can be optimised. The application pattern is 0 for a sequential function, 1 for a parallel function whose implementation template finishes by gathering local results to the master and 2 for any other parallel function. The data pattern indicates which application pattern was used to generate the data. Cost is 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. The heart of the analysis is the construction of an operator `bspapp`, which computes the cost of a BSP implementation of an application term using information from the cost tuples of its components. It is expressed as follows, with readability improved by using the shorthands `t_shp(fx)`, `t_pattern(fx)`, `t_apcost(fx)`, `t_size(fx)` to represent the shape of the result, the application cost, the application pattern and the size of the result from `fx` (which are respectively the shape components of the costs of the function and argument terms)

$$\text{cost}(t t') = \text{bspapp } \text{cost}(t) \text{cost}(t')$$

where

$$\begin{aligned} & \text{bspapp } \langle f, s, d, T \rangle \langle x, s', d', T' \rangle \\ &= \langle \text{t_shp}(fx), \text{data_sz}(\text{t_apcost}(fx)), \text{t_pattern}(fx), \\ & \quad (T + T') + \lambda(p, g, l) \cdot ((\text{comm_cost}(\text{t_pattern}(fx)) d' s s') + l) + \text{t_apcost}(fx) \rangle \\ & \text{data_sz } t = s + s', \quad \text{if } t = 0 \\ & \quad = \text{t_size}(fx), \quad \text{otherwise} \\ & \text{comm_cost } x_1 x_2 x_3 x_4 = 0, \quad \text{if } x_1 = 0 \\ & \quad = (x_3 * (p - 1) - x_4 * ((p - 1)/p)) * g - l, \quad \text{if } x_2 = 1 \\ & \quad = (x_3 * (p - 1) + x_4 * ((p - 1)/p)) * g, \quad \text{otherwise} \end{aligned}$$

Note that `data_sz` captures the fact that if the result of an application is a function then there is no application cost (for example, in our model it costs nothing at run time to apply `map` to `hd` - the costs come when the result is applied to some data) and so the message size of `t t'` is just the sum of `s` and `s'` (the data required to represent the terms dynamically). When the result is not a function the message size is `t_size(fx)`. 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(fx)`. Note that `+` indicates point wise function addition in this context. The communication cost, captured by `comm_cost`, depends on the application pattern `t_pattern(fx)` and the argument data pattern `d'`. If the application pattern is not 0 and the data pattern is 1, 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. The Maximum Segment Sum Problem

The maximum segment sum (*mss*) problem is widely used as an example of BMF style program derivation (e.g. [1, 2, 6, 11]). The problem is to find the sum of the contiguous segment of a list whose members have the largest sum among all such segments. For example,

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

Skillicorn and Cai [11] gave a derivation of an implicitly parallel program. As reformulated for our analysis framework it is:

$$\text{reduce}(\uparrow)(\text{map}(\text{reduce}(+))(\text{redconcat}(\text{map}(\text{tails})(\text{inits } x)))) \quad (1)$$

$$= \text{reduce}(\uparrow)(\text{redconcat}(\text{map}(\text{map}(\text{reduce}(+)))(\text{map}(\text{tails})(\text{inits } x)))) \quad (2)$$

$$= \text{reduce}(\uparrow)(\text{map}(\text{reduce}(\uparrow))(\text{map}(\text{map}(\text{reduce}(+)))(\text{map}(\text{tails})(\text{inits } x)))) \quad (3)$$

$$= \text{reduce}(\uparrow)(\text{map}(\lambda v.(\lambda z.(\text{fst } z) \uparrow (\text{snd } z))A)(\text{inits } x))$$

where

$$A = \text{reduce}(\otimes)(\text{map}(\lambda y.(y, 0))v)$$

$$\otimes = \lambda xy.\text{pair}(\text{fst } x + \text{fst } y)((\text{snd } x + \text{fst } y) \uparrow 0) \quad (4)$$

$$= \text{reduce}(\uparrow)(\text{map}(\uparrow)(\text{prefix}(\otimes)(\text{map}(\lambda y.(y, 0))x))) \quad (5)$$

where, \uparrow and redconcat represent the maximum operation and the reduce with concatenation respectively. In the following sections we give a mechanism to automatically compute the shapes and BSP costs of programs like (1)-(5). This involves further expansion of shape expressions and cost functions for the new operations.

4. Expanding our Shape Analysis

An important feature of Jay's VEC language and our previous language VEC-BSP was that they 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 $\text{map}(+1)v$ where v is a vector of length 1000 performs 1000 binary operations, but the corresponding shape analysis concerns only the shape $\langle 1000, \text{un} \rangle$. However, when we try to use this cost analysis to compare programs in BMF style derivations (using vectors to represent lists) we often encounter programs which cannot be expressed with this constraint. The principles of the costing mechanism in this paper are similar to those for VEC-BSP, but with the fundamental difference that sub-vectors are allowed to have different lengths. When we relax the constraint of 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, a shape vector is introduced to express shape information. The 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 . The program is annotated with shape information to assist informal cost calculation. This shape vector expression is concise, but the calculation of shape is done by hand. Our new approach tries 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, the shape is expressed using pair notation $(,)$. Otherwise, we use the vector constructor $[]$ as the shape, allowing it to include both $(,)$ and $[]$ expressions as sub-vectors. For example, the intermediate real data structures of the initial version of the *mss* program with argument vector $[1,2,3,4]$ are:

$$\begin{array}{l}
[1, 2, 3, 4] \\
\downarrow \text{inits} \\
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]] \\
\downarrow \text{map(tails)} \\
[[[1]], [[2], [1, 2]], [[3], [2, 3], [1, 2, 3]], [[4], [3, 4], [2, 3, 4], [1, 2, 3, 4]]] \\
\downarrow \text{redconcat} \\
[[1], [2], [1, 2], [3], [2, 3], [1, 2, 3], [4], [3, 4], [2, 3, 4], [1, 2, 3, 4]] \\
\downarrow \text{map(reduce(+))} \\
[1, 2, 3, 3, 5, 6, 4, 7, 9, 10] \\
\downarrow \text{reduce}(\uparrow) \\
10
\end{array}$$

and the corresponding shape expressions should be

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

with the difference in data size between source program evaluation and shape deduction becoming more significant as the length of the input vector grows. Note that we use 1 as the shape of a datum constant instead of `un` for the convenience of computing its size from the shape. 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 challenge is thus to automate this new shape analysis. The basic idea is to define a shape function for each combinator which captures the corresponding shape change in either situation. Such a function f takes the form

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

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

$$\begin{aligned} \text{cshape_hd } x &= \text{t_eshp } x, && \text{if } x \text{ is a pair} \\ &= \text{hd } x, && \text{if } x \text{ is a vector} \\ \text{cshape_map } f x &= (\text{t_len } x, \text{t_shp } (f (\text{t_eshp } x))), && \text{if } x \text{ is a pair} \\ &= \text{map } f x, && \text{if } x \text{ is a vector} \end{aligned}$$

where `t_len` and `t_eshp` represent the operations which take the length component and element shape component from the shape respectively. This amendment makes it possible to automate the new shape deduction and so to compute cost. However, in some cases, shapes which have the same element shape are expressed as a vector rather than a pair. For example, `cshape_map` which was defined above reduces the shape of the second last step in the derivation to $[1, 1, 1, 1, 1, 1, 1, 1, 1]$ 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. Our solution is to add one more information component *vector levels* to the cost tuple and an information component *level change* to the application tuple. The vector level of a real vector data item is (the number of nested levels +1), e.g. 1 for a non-nested vector and 2 for a vector of a non-nested vectors. We set the vector level of datum constants and primitive function terms to 0. The level change is a function that captures the change of the vector level after function application. The new cost tuples take the form

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

and the new application tuples take the form

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

When the result shape is a vector and the vector level of the result is 1, the vector can be converted to a pair using `topair`.

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

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 $\langle \dots \rangle$) of the shapes of the components. The shape of a vector is a vector (referred to as a *vector shape*) of the shapes of the elements if these are not uniform or a pair of the length and element shape (referred to as a *pair shape*) if they are. We must also amend the associated definitions of data size, application cost and communication cost. Data size can be computed from the shape by the operator `size` defined by

$$\begin{aligned} \text{size } \langle x_1, x_2, \dots, x_n \rangle &= \text{size } x_1 + \text{size } x_2 + \dots + \text{size } x_n \\ \text{size } (x, y) &= \text{size } x * \text{size } y \\ \text{size } [x_1, x_2, \dots, x_n] &= \text{reduce } (+) (\text{map } (\text{size}) [x_1, x_2, \dots, x_n]) \\ \text{size } n &= n \end{aligned}$$

The data size transmitted from the master processor to the other processors for scattering the data of an argument (whose shape is x) is determined by the operator `scatsz`:

$$\begin{aligned} \text{scatsz } x &= \text{size } x * (p - 1) / p && \text{if } x \text{ is a pair} \\ &= \text{size } (\text{drop } ((\text{length } x) / p) x) && \text{if } x \text{ is a vector} \end{aligned}$$

Consequently, the communication cost counted in `bspapp` is replaced by

$$\text{comm_cost } (\text{t_pattern } (f x)) d' s (\text{scatsz } x)$$

where

$$\begin{aligned} \text{comm_cost } x_1 x_2 x_3 x_4 &= 0, && \text{if } x_1 = 0 \\ &= (x_3 * (p - 1) - x_4) * g - l, && \text{if } x_2 = 1 \\ &= (x_3 * (p - 1) + x_4) * g, && \text{otherwise} \end{aligned}$$

The definition of the new `bspapp` is

$$\text{cost } (t t') = \text{bspapp } \text{cost } (t) \text{cost } (t')$$

where

$$\begin{aligned} &\text{bspapp } \langle f, s, d, n, T \rangle \langle x, s', d', n', T' \rangle \\ &= \langle \text{topair } (\text{t_lchange } (f x) n') (\text{t_shp } (f x)), \text{data_sz } (\text{t_apcost } (f x)), \\ &\text{t_pattern } (f x), \text{t_lchange } (f x) n', (T + T') \\ &+ \lambda \langle p, g, l \rangle. (\text{comm_cost } (\text{t_pattern } (f x)) d' s (\text{scatsz } x) + l) + \text{t_apcost } (f x) \rangle \\ &\text{data_sz } x_1 = s + s', && \text{if } x_1 = 0 \\ &= \text{t_size } (f x), && \text{otherwise} \end{aligned}$$

where, `t_lchange` represents the operation returning the level change components from $f x$.

5. New Cost Functions for Operators

In our model a translation function *cost* translates source terms to cost tuples. For example,

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

It indicates that its shape, data size, data pattern and vector level are 1, 1, 0, 0 respectively and that it takes no time to evaluate the term *d* itself.

$$\text{cost}(d) = \langle \lambda x. \langle \lambda y. \langle 1, 0, +0, 1 \rangle, 0, +0, 0 \rangle, 0, 0, 0, 0 \rangle$$

where *d* is a binary datum operation

Working from the right hand end of the expression in, it indicates that it takes no time to evaluate the term *d* itself, its vector level, data pattern, data size are all 0 and that its shape is $\lambda x. \langle \lambda y. \langle 1, 0, +0, 1 \rangle, 0, +0, 0 \rangle$. From the right hand end of the shape, it indicates that it takes no time to apply *d* to a first argument, it does not change the vector level and that application pattern involved is 0. Application cost for applying the resulting function to a second argument is set to 1, its level change, application pattern and resulting shape are +0, 0, 1 respectively.

The cost functions given below, except for *map*, *reduce* and *prefix* when *x* has pair shape, which are given in the previous papers, are introduced for the first time in this paper. In particular, the introduction of *redconcat*, *inits* and *tails* is made possible by our extended shape formulation. They take a common structure, $\langle \lambda x.f, 0, 0, 0, 0 \rangle$, that is, the components of the cost tuple of the function terms themselves, except for shape, are all 0. We use the notation $\lambda'x.f$ for this.

5.1. Map

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

$$\text{cost}(\text{map}) = \lambda'f. \langle \lambda x. \text{shape_map}, 0, +0, 0 \rangle$$

where, *shape_map*

$$\begin{aligned} &= \langle (\text{t_len}x, \text{t_shp}(f(\text{t_eshp}x))), 1, \text{t_lchange}(f x), \text{t_apcost}(f(\text{t_eshp}x)) * (\text{t_len}(x)/p) \\ &\quad + \text{t_size}(f(\text{t_eshp}x)) * (\text{t_len}(x)/p) * (p - 1) * g + l \rangle, \text{ if } x \text{ is a pair} \\ &= \langle \text{map}(\text{t_shp})\text{tuples}, 1, \text{t_lchange}(f x), \text{maxsum}(\text{map}(\text{t_apcost})\text{tuples}) \\ &\quad + \text{gath_sz}(\text{map}(\text{t_size})\text{tuples}) * g + l \rangle, \quad \text{if } x \text{ is a vector} \end{aligned}$$

where *tuples* = $(\text{map } f x)$

The level change function of *map f* is the same as the level change function of *f*, that is $\text{t_lchange}(f x)$. When *x* is a vector shape the resulting shape is the vector obtained by $\text{map}(\text{t_shp})(\text{map } f x)$, in which each element shape is computed. The computation cost is the maximum time of local computation in any processor, obtained by taking the application cost of the function for each element (that

is $\text{map}(\text{t_apcost})(\text{map } f x)$, computing their summation for every $(\text{length } x)/p$ elements, and then taking the maximum among them. The last two steps are defined as an operator maxsum . The communication cost is computed from the message size involved in gathering the local results. This is the sum of the sizes of all result elements, except the first $(\text{length } x)/p$ which were computed directly by the master, and is denoted by the operator gath_sz .

5.2. *Reduce*

In the modelled implementation of *reduce*, sub-vectors are reduced sequentially on each processor, with results then transferred to the master processor which reduces them together. As in Skillicorn's calculus, *reduce* is used only with operators which take constant space, that is the shape of their results are same as the shape of their arguments. Reductions with non-constant space operations are defined individually (e.g. *redconcat* defined below). Its cost function is:

$$\text{cost}(\text{reduce}) = \lambda' \oplus . \langle \lambda x. \text{shape_reduce}, 0, +0, 0 \rangle$$

where, shape_reduce

$$\begin{aligned} &= \langle \text{t_eshp } x, 2, (-1), \text{t_apcost}(\text{t_shp}(\oplus(\text{t_eshp } x))(\text{t_eshp } x)) * (\text{t_len } (x)/p \\ &\quad + p - 2) + (\text{size}(\text{t_eshp } x)) * (\text{t_len } (x)/p * (p - 1)) * g + l \rangle, \text{ if } x \text{ is a pair} \\ &= \langle \text{hd } x, 2, (-1), \text{t_apcost}(\text{t_shp}(\oplus(\text{hd } x))(\text{hd } x)) * (\text{length } (x)/p + p - 2) \\ &\quad + (\text{size}(\text{hd } x)) * (\text{length } (x)/p * (p - 1)) * g + l \rangle, \text{ if } x \text{ is a vector} \end{aligned}$$

The level change function is (-1) because *reduce* reduces the vector level by 1. When x is a vector shape the computation of the shape and the application cost is performed using hd and length instead of t_eshp and t_len respectively.

5.3. *Redconcat*

The implementation of *redconcat* is sequential on the master processor to avoid distribution cost.

$$\text{cost}(\text{redconcat}) = \lambda' x. \text{shape_redconcat}$$

where, shape_redconcat

$$\begin{aligned} &= \langle (\text{t_len } x * \text{t_len}(\text{t_eshp } x), \text{t_eshp}(\text{t_eshp } x)), 0, (-1), \\ &\quad \text{concatConst} * (\text{t_len } x - 1) \rangle, \text{ if } x \text{ is a pair} \\ &= \langle \text{redconcat } x, 0, (-1), \text{concatConst} * (\text{length } (x) - 1) \rangle, \text{ if } x \text{ is a vector} \end{aligned}$$

5.4. *Prefix*

The modelled implementation of *prefix* begins with each processor prefixing its assigned block of elements sequentially. The final values of these local prefixes are then prefixed across processors in parallel using the obvious tree algorithm. The

result of this global prefix on processor $i (< p)$ is now sent to processor $i + 1$. In all processors, \oplus is applied to the pair of the result of the global prefix and the local results. Finally, the results from all processors are gathered to the master processor. We make the same restrictions concerning constant space operators as for `reduce`. The cost function is:

$$\text{cost}(\text{prefix}) = \lambda' \oplus . \langle \lambda x. \text{shape_prefix}, 0, +0, 0 \rangle$$

where, `shape_prefix`

$$\begin{aligned} &= \langle x, 1, 0, \text{t_apcost}(\text{t_shp}(\oplus(\text{t_eshp}(x)))(\text{t_eshp}(x))) * (2 * (\text{t_len}(x)/p) - 1 \\ &\quad + \log(p)) + (\text{size}(\text{t_eshp}(x)) * (\log(p) + 1) + (\text{size}(x)/p) * (p - 1)) * g \\ &\quad + (\log(p) + 2) * l \rangle, \quad \text{if } x \text{ is a pair} \\ &= \langle x, 1, 0, \text{t_apcost}(\text{t_shp}(\oplus(\text{hd}(x)))(\text{hd}(x))) * (2 * (\text{length}(x)/p) - 1 \\ &\quad + \log(p)) + (\text{size}(\text{hd}(x)) * (\log(p) + 1) + (\text{size}(x)/p) * (p - 1)) * g \\ &\quad + (\log(p) + 2) * l \rangle, \quad \text{if } x \text{ is a vector} \end{aligned}$$

5.5. *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. Its cost function is:

$$\text{cost}(\text{inits}) = \lambda' x. \text{shape_inits}$$

where, `shape_inits`

$$\begin{aligned} &= \langle [(1, \text{t_eshp}(x)), (2, \text{t_eshp}(x)), \dots, (\text{t_len}(x), \text{t_eshp}(x))], \\ &\quad 1, +1, \text{concatConst} * (\text{t_len}(x)/p) \\ &\quad + (\text{size}(\text{t_eshp}(x)) * (\text{t_len}(x)/p) * g + (\text{t_len}(x)/p) * \text{concatConst} + l) * (p - 1) \\ &\quad + \text{size}(\text{drop}(\text{t_len}(x)/p))[(1, \text{t_eshp}(x)), (2, \text{t_eshp}(x)), \dots, (\text{t_len}(x), \text{t_eshp}(x))] * g \\ &\quad + l \rangle, \quad \text{if } x \text{ is a pair} \\ &= \langle \text{inits } x, 1, +1 \text{ concatConst} * (\text{length}(x)/p) \\ &\quad + ((\text{size}(\text{take}(\text{length}(x)/p) x)) * g + (\text{t_len}(x)/p) * \text{concatConst} + l) * (p - 1) \\ &\quad + \text{size}(\text{drop}(\text{length}(x)/p) \text{inits } x) * g + l \rangle, \quad \text{if } x \text{ is a vector} \end{aligned}$$

The level change function is $(+1)$ since `inits` increases the vector level by 1. The result shape becomes a vector shape even if x is a pair shape. `tails` is analogous to `inits`, but computes the suffix segments of its argument vector. Its cost function is the same as that of `inits`.

6. Cost Analysis of the *mss* Derivation

We have implemented our cost analysis framework as a Haskell program based on that originally developed for the PRAM cost analysis of VEC by Jay's group [4]. We now present an investigation of the *mss* derivation. Firstly, in section 6.1, we play the role of a derivational programmer, comparing the *predicted* performance of the various programs. Then, in section 6.2, we investigate the accuracy of these predictions by comparing them with the actual run times of corresponding programs on a real parallel machine. In order to make comparisons we must instantiate our model with the BSP characteristics of a target machine, in this case an 8-processor Sun HPC 3500 (shared memory, symmetric multi-processor system with UltraSPARC II processors). The BSP cost parameter values 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 value which normalises l and g to the sequential processor speed.

6.1. Predicting the intermediate programs

We now present the predicted results for each derivation step, plotting the predicted costs against varying input vector size n . We choose ranges of n which are large enough in each case to illustrate the emerging trend, thereby demonstrating that our tool can be used to investigate performance in specific areas of practical interest rather than simply asymptotically.

Figure 1 plots the predicted results of (1) and (2) varying n up to 160. It shows that the cost of (1) increases much faster than that of (2). The curves seem to match the asymptotic complexity of $O(n^2)$ which would be derived by informal analysis. Figure 2 plots the predicted results of (2) and (3) varying n up to 224. In this case, (2) is a little more efficient than (3) but their curve is almost same. It seems that the difference in efficiency would not change significantly even if n were large.

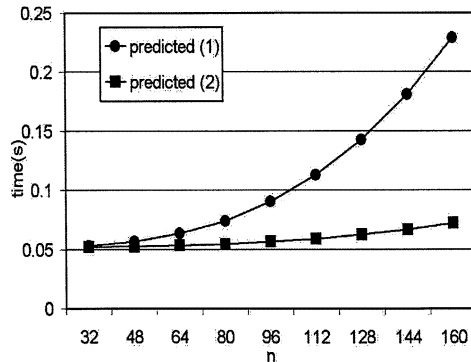


Fig. 1. Predicted costs of (1) and (2).

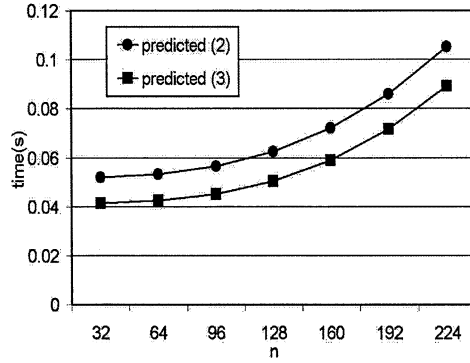


Fig. 2. Predicted costs of (2) and (3).

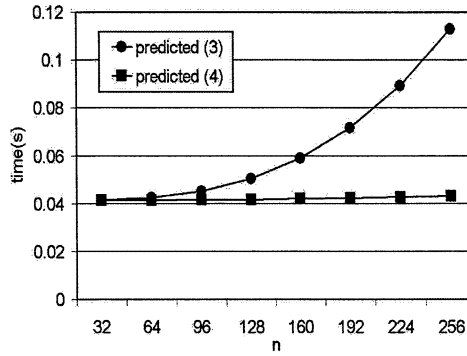


Fig. 3. Predicted costs of (3) and (4).

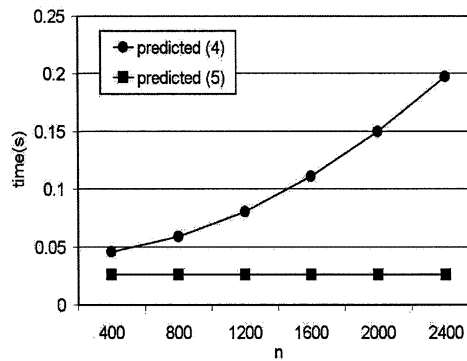


Fig. 4. Predicted costs of (4) and (5).

Figure 3 plots the predicted results of (3) and (4) varying n up to 256. These are similar up to about 100, but the difference in cost becomes significantly large as n grows. Figure 4 plots the predicted results of (4) and (5) while varying n up

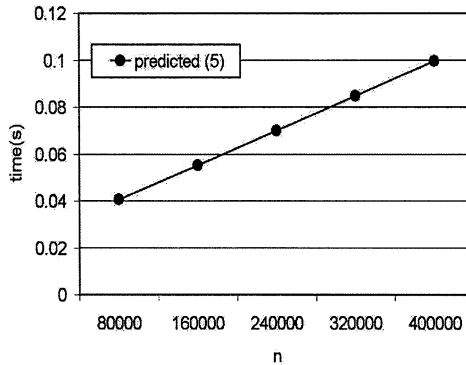


Fig. 5. Predicted cost (5).

to 2400. After $n = 400$, the cost of (4) increases, while (5) seems to be almost constant. Finally, Figure 5 shows the cost behaviour of (5) with n varying up to 400000. It seems that the complexity of (5) is linear.

Notice that we can predict not only differences in absolute value of costs but also the size of input vector at which the differences begin to be significant. In this example, (1) \rightarrow (2) reduces cost significantly even if n is very small suggesting that the redistribution cost caused by `redconcat` between `map(tails)(inits x)` and `map(reduce(+))` is expensive. (2) \rightarrow (3) reduces cost slightly, independent of n . (3) \rightarrow (4) reduces the cost significantly when n is larger than about 100. (4) \rightarrow (5) reduces the cost significantly when n is larger than about 500.

6.2. Accuracy test

To test the accuracy of our cost analysis against performance on a real machine we hand compiled the BMF expressions into C and Oxford BSPlib for the five programs above and ran them on an 8-processor Sun HPC 3500. For example, the BSPlib pseudo code for the core of algorithm (5) is given in Figure 6. Following the same sequence of experiments using the same ranges of n as for the predictions, Figure 7 shows comparisons of predicted and real run times. Although our calculator seems to tend to underestimate a little, our predicted curves capture the characteristics of the behaviour of real run time with high accuracy.

7. Future Work

It would be interesting to extend the range of “regular irregularity” which the system can handle, while maintaining or improving the efficiency with which analysis is performed. The time currently taken by our analysis program to make a prediction for program (5) is a matter of a few seconds on a contemporary workstation, independent of n . The time for programs (1)-(4) at the largest value of n in the above experiments is a few minutes (because of the irregular vector shapes

```

bsp_get(0,list,myoffset,fst_local_list,n_over_p * sizeof(int)); /* scatter the */
bsp_sync(); /* input list */
for (i=0;i<n_over_p;i++)
    snd_local_list[i] = 0; /* set second element of each pair to 0 */
fst_local_result[0] = fst_local_list[0];
for (i=1; i<n_over_p; i++){ /* local prefix */
    fst_local_result[i] = fst_local_result[i-1] + fst_local_list[i];
    snd_local_result[i] = snd_local_result[i-1] + fst_local_list[i];
    if (snd_local_result[i] < 0) snd_local_result[i] = 0;
}
fst_last = fst_local_result[n_over_p-1];
snd_last = snd_local_result[n_over_p-1];
for (i=1; i<P; i*=2) { /* global prefix */
    if (bsp_pid()+i < P){
        bsp_put(bsp_pid()+i,&fst_last,&fst_left,0,sizeof(int));
        bsp_put(bsp_pid()+i,&snd_last,&snd_left,0,sizeof(int));
    }
    bsp_sync();
    if (bsp_pid()>=i) {
        snd_last = snd_left + fst_last;
        if (snd_last < 0) snd_last = 0;
        fst_last = fst_left + fst_last;
    }
}
if (bsp_pid()>0){ /* send the result of global prefix to the next processor */
    bsp_get(bsp_pid()-1,&fst_last,0,&fst_shift,sizeof(int));
    bsp_get(bsp_pid()-1,&snd_last,0,&snd_shift,sizeof(int));
}
bsp_sync();
if (bsp_pid()==0){
    fst_shift = 0;
    snd_shift = 0;
}
for (i=0; i < n_over_p; i++) { /* apply the function to */
    fst_result[i] = fst_shift + fst_local_result[i]; /* the local result and */
    snd_result[i] = snd_shift + fst_local_result[i]; /* the global result */
    if (snd_result < 0) snd_result[i] = 0;
}
for (i=0; i < n_over_p; i++) { /* take the maximum of each pair */
    if (fst_result[i] > snd_result[i])
        pair_max[i] = fst_result[i];
    else
        pair_max[i] = snd_result[i];
}
local_max = pair_max[0];
for (i=1; i < n_over_p; i++){ /* take the maximum in each processor */
    if (pair_max[i] > local_max)
        local_max = pair_max[i];
}
bsp_put(0,&local_max,g_max,bsp_pid()*sizeof(int),sizeof(int)); /* gather local */
bsp_sync(); /* results */
if (bsp_pid()==0) { /* take the maximum of the local results */
    result = g_max[0];
    for (i=1; i<P; i++)
        if (result < g_max[i])
            result = g_max[i];
}

```

Fig. 6. BSP code for algorithm (5).

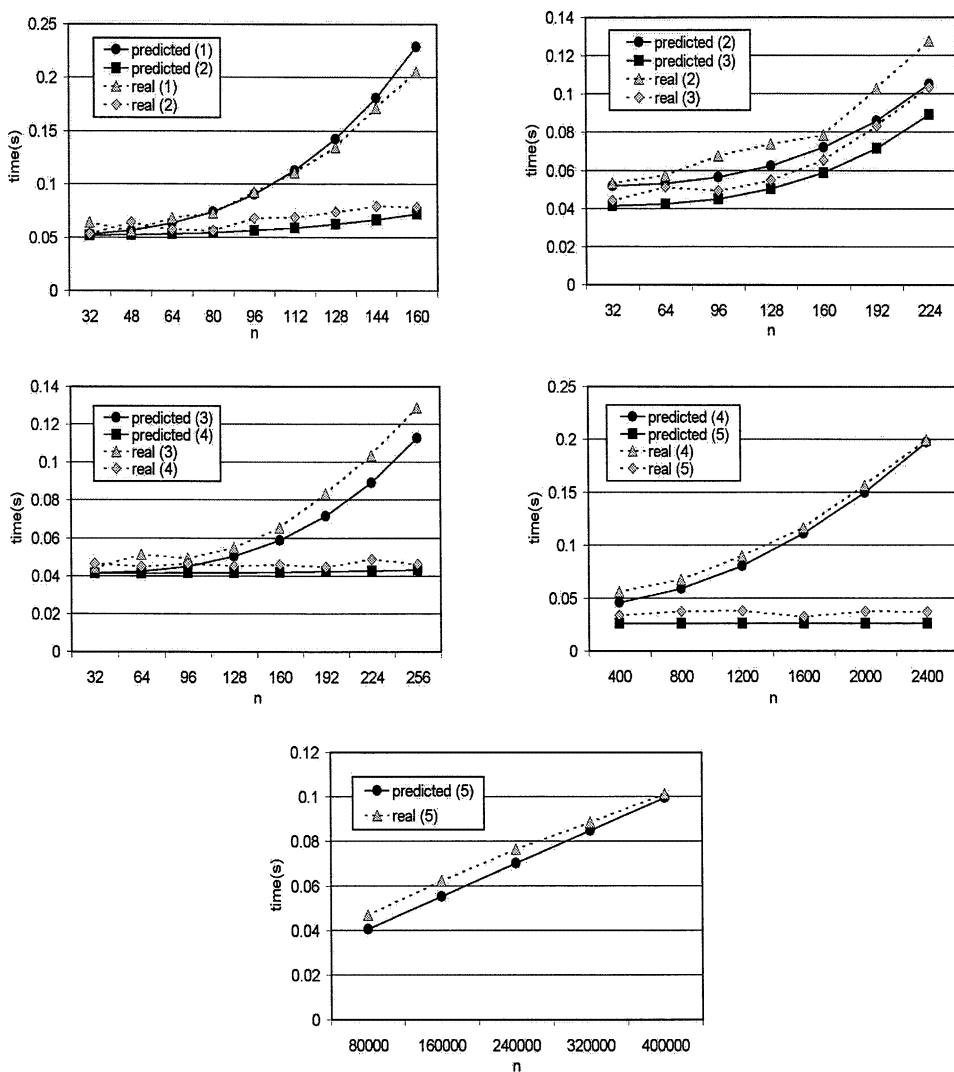


Fig. 7. Accuracy test.

involved). In a complementary direction, our framework will make it possible to add more parallelised combinatorics, enabling us to apply the tool to a wider range of application problems.

Acknowledgement

We thank Barry Jay and his group at the School of Computing Sciences, University of Technology, Sydney for their pioneering work in the area and for access to the source code for the original PRAM cost analyser.

References

1. R.S. Bird, Algebraic Identities for Program Calculation, *Computer Journal*, **32**(2) (1989) 122–126.
2. S. Gorlatch, Toward Formally-Based Design of Message Passing Programs. *IEEE Transactions on Software Engineering*, **26**, (3), (2000), 276–288.
3. Y. Hayashi, Shape-based Cost Analysis of Skeletal Parallel Programs, PhD Thesis, University of Edinburgh, 2001.
4. Y. Hayashi and M. Cole, BSP-based Cost Analysis of Skeletal Programs. In *Trends in Functional Programming*, eds. G. Michaelson and P. Trinder (Intellect, 2000), 20–28.
5. Y. Hayashi and M. Cole, Static Performance Prediction of Skeletal Parallel Programs, to appear in *Parallel Algorithms and Applications*.
6. Z. Hu and H. Iwasaki and M. Takeichi, Formal Derivation of Parallel Program for 2-Dimensional Maximum Segment Sum Problem, Euro-Par'96, LNCS 1123, (Springer, 1996) 553-562.
7. C. B. Jay, Costing Parallel Programs as a Function of Shapes, *Science of Computer Programming* **37** (2000), 207–224.
8. C. B. Jay and M. I. Cole and M. Sekanina and P. A. Steckler, A Monadic Calculus for Parallel Costing of a Functional Language of Arrays, in *Euro-Par'97*, LNCS 1300, (Springer 1997) 650–661.
9. D. B. Skillicorn, *Foundations of Parallel Programming*, (Cambridge U. Press 1994).
10. D. B. Skillicorn and J. M. D. Hill and W. F. McColl. Questions and Answers about BSP, *Scientific Programming*, **6**(3)(1997) 249–274.
11. D. B. Skillicorn and W. Cai, A Cost Calculus for Parallel Functional Programming, *Journal of Parallel and Distributed Computing*, **28**(1) (1995), 65–83.