A Cost Calculus for Parallel Functional Programming

D.B. Skillicorn*
Department of Computing and Information Science
Queen's University, Kingston, Canada
skill@qucis.queensu.ca

W. Cai
School of Applied Science
Nanyang Technological University
Singapore 2263
aswtcai@ntu.ac.sg

August 23, 1994

Abstract

Building a cost calculus for a parallel program development environment is difficult because of the many degrees of freedom available in parallel implementations, and because of difficulties with compositionality. We present a strategy for building cost calculi for skeleton-based programming languages which can be used for derivational software development and which deals in a pragmatic way with the difficulties of composition. The approach is illustrated for the Bird-Meertens theory of lists, a parallel functional language with an associated equational transformation system.

Keywords: functional programming, parallel programming, program transformation, cost calculus, equational theories, architecture independence, Bird-Meertens formalism.

1 Introduction

The calculational approach to software development uses a set of transformation or refinement rules to derive an efficient, executable specification (that is, a program) from a

*This work was supported by the Natural Sciences and Engineering Research Council of Canada.

naive but obviously correct specification. The transformation rules are constructed to be correctness-preserving. Thus there is no *post hoc* verification requirement, provided that all of the transformations used during a derivation are correctly applied. Derivations are broken up into small steps, usually the application of a single rule, and therefore require (at least in theory) fewer large insights and more small insights or routine steps. Derivations also make the choice points in each design explicit and generate documentation of choices as a side-effect. The collection of transformation rules for a particular context is called a *programming calculus*.

The calculational approach is becoming deservedly popular for sequential software development [6, 24]; and seems essential for parallel software development because of the extra complexity involved. Verifying a massively parallel program after it has been written seems too complex to be a practical approach.

One aspect of calculational software development to which little attention has been paid is the provision of cost information at intermediate stages in a derivation. At points where choices must be made between algorithms, such information can be used to guide the derivation towards efficient programs. In what follows, we take cost to mean parallel execution time, but other costs such as memory or processor requirements can also be considered.

An ideal cost system should have the following three properties:

- It should abstract from the details of implementations sufficiently to allow costs to be computed from only small amounts of information.
- It should be compositional, that is the cost of a program constructed from smaller pieces should depend in a simple way on the cost of the pieces. For this to be possible, the cost of each program fragment should depend only on what that fragment does and not on the context in which it occurs.
- It should be related to the programming calculus, so that transformation rules can be annotated with their effects on costs. The effect of using a rule at some point in the derivation can then be determined before using it.

The first property requires identifying those parameters of programs, compilers, and architectures that are critical to performance. The second two properties require that the cost system be a calculus, coupled with and parallel to the programming calculus.

It is harder to build a useful cost system for parallel computation than for sequential computation because there are many more degrees of freedom. One way to characterise cost systems is to ask what factors must be known in order to compute costs. The fewer factors required, the more abstract and useful the cost system becomes. Some important factors are:

- Details of the structure of the program;
- The size of the input (or the size of the problem);

- The extent to which the work to be done depends on values of the input, rather than their number and sizes;
- The way in which the program is decomposed into threads that can execute on different (virtual) processors;
- The way in which communication between threads and the synchronization rules associated with it are arranged;
- The way in which the threads are mapped to physical processors (which usually involves multiplexing many threads on each processor and therefore introducing contention);
- The mapping of communication actions to the target processor's interconnect (which again involves sharing and hence contention);
- The extent to which the computation exhibits dynamic behaviour;

The range of choices involved here makes a general cost system problematic. At present, the only known way to build cost systems is to restrict the possible choices dramatically.

This paper makes two contributions. The first is a method for getting compositionality at the expense of limited inaccuracy, by providing implementation information indirectly to programmers via annotations to equations. There are two views of programs. Programmers see programs as compositions of second-order functions whose costs are additive (that is the cost of a composition of two functions is the sum of their individual costs). This level of abstraction is exactly what is needed for cost-based program derivation. Implementers view programs as compositions of monolithic, parameterised implementations of a set of basic operations. Because of the existence of parameterised implementations, realistic costs, including a proper accounting for communication, can be expressed as functions of argument size and number of processors available, but *independent of target architecture class*. Information about the relative costs of different implementations of the same function is conveyed by labelling or annotating equations with a cost-reducing direction. Equations are functional tautologies, but implementation inequalities. Such annotation conveys cost information to programmers without violating the abstraction they see. Situations where the costs of implementations violate the simple additive view that programmers have are fixed by adding annotated equations to the software development transformation system.

The second contribution is to work out such a cost system for a particular functional programming language, the Bird-Meertens theory of lists. The approach works for this language because:

- all list homomorphisms can be expressed as compositions of two basic second-order operations, *maps* and *reductions*, so the initial set of operations for which implementations must be built is small;
- there is a rich equational theory that provides, at several points, help with finding implementations of more complex list operations;

- the topological requirements of list homomorphisms are mild, so that architecture independent implementations can be built for a wide range of realistic architectures.

As a side-effect, we produce a set of novel parallel implementations of common list operations, and their complexities.

In the next section we briefly survey existing approaches to computing the cost of programs, particularly in a parallel setting. Not a great deal has so far been done. In Section 3 we present the general strategy for handling costs so that the abstraction of compositionality is maintained for programmers. Section 4 reviews the Bird-Meertens theory of lists. Section 5 defines a notation that captures information about the size of arguments (which is needed to reason about the cost of operations, at least in the implementation view). Sections 6 and 7 give details of the implementation and costs of typical operations for the simple case where the size of the argument list matches the number of processors available on the target architecture. Section 8 repeats this for the harder, general case where list size and target processors do not match. Section 9 shows how the implementation view is used to direct equations to convey cost information indirectly to the software development level. Section 10 shows how a programmer would use the equational system, by deriving a new parallel version of the maximum segment sum problem.

2 Existing Parallel Cost Systems

For sequential computation, several cost systems have been developed. The RAM (Random Access Memory) model is the standard cost system for sequential imperative programming. Basic instructions are assumed to take unit time, memory references to take zero time, and space used to be the maximum number of named locations in use at any step during the computation. Only program loops and recursions can affect the order of the execution time derived using this accounting, so it is straightforward to compute. Furthermore, the times computed are (up to a constant factor) the times programs take on real machines because the cost system is an accurate representation of real costs.

For functional programming, this cost system does not work, because the programming model abstracts from flow of control and, in the presence of non-strictness, from whether some parts of the program are executed at all. The standard functional cost model is to count function calls as a surrogate for execution time. Techniques for doing this, which can be largely automated, have been investigated by Le Métayer [19], and Sands [28, 29], who also deals with non-strictness. Bellegarde [2] has investigated using cost information to reduce redundant steps in programs in FP.

Cost information is also used informally in programming calculi to argue about the improvements in efficiency that result from derivations. Examples may be found in the work of Bird [3, 6], especially in the derivation of the Knuth-Morris-Pratt algorithm [7], and in the Refinement Calculus of Morgan [24].

There are five main existing approaches to parallel cost systems: complete analysis,

the PRAM model used in complexity theory, Valiant's Bulk Synchronous Parallelism, the polytope model, and non-standard interpretations of functional languages.

If all of the decisions listed at the end of the previous section are made explicitly as part of the programming task, then cost measures are possible since, at a low enough level, program execution is deterministic. There are three reasons why this is not practical for even moderately sized programs:

- Programmers do not want to (and probably cannot) work at this level of detail;
- The amount of detail that has to be used to work out costs is too large, especially for massive parallelism;
- The complexity of finding the optimal solution to even a part of the problem, the mapping of a logical program structure to a physical system (the *Mapping Problem*), is known to be exponential.

Also such cost systems can only be applied when programs have been completely developed, and so can play no role in calculational development.

The PRAM model is used as the basis of the complexity theory of parallel algorithms. In the PRAM (Parallel Random Access Machine) model, the execution of basic instructions and access by a processor to a shared memory are assumed to take unit time. Communication between threads is modelled by having the first thread write to a preagreed location and the second thread read from it. Thus inter-thread communication is counted as taking $O(1)$ time. The cost of a computation is given by stating the number of processors it uses and the number of (parallel) unit time steps it requires.

Computation of cost in the PRAM model requires knowing the complete structure of the program, and its decomposition into threads, but does not depend on mapping of data to memory (since all parts of memory are assumed to have the same latency). The problem with the PRAM model is that communication in the real world cannot be realized in constant time, ultimately because of the finite speed of light. PRAM costs underestimate the real costs of execution but, worse, do so in an unpredictable way. Algorithm A and Algorithm B may have the same PRAM cost but their real costs may differ by orders of magnitude because one's communication demands are higher than the other's. Worse still, Algorithm A may appear cheaper than Algorithm B in the PRAM framework, while Algorithm B is cheaper than Algorithm A on some useful set of parallel architectures. Examples where this occurs are found in [8, 25, 32]. While the PRAM model has been used extensively by algorithm designers, its use in practice is problematic.

The Bulk Synchronous Parallelism model of Valiant [22, 33–35] achieves a workable parallel cost system by assuming uniform communication, that is the use of randomization techniques to bound the message delivery time across communication networks, even in the presence of a bounded number of competing messages [23, 33]. This is more realistic than the PRAM model since it accounts for the true cost of communication, and is accurate for new machines that provide the abstraction of uniform communication.

A BSP program is a parallel program consisting of a number of phases or supersteps. At the beginning of each superstep, each thread may issue a global memory operation. For the remainder of the superstep it computes with values held local to it. After the end of the superstep, the global memory operations become effective locally, ready for the next superstep. Thus a read from a remote memory must be requested in the superstep before the one in which the value is required.

The performance of a BSP program can be computed based on two program properties, the problem size and the duration of supersteps, and three architectural properties, the number of processors used, the ratio of computation time to communication time, and the latency of communication across the machine. This is an attractive level of abstraction, requiring limited knowledge of program structure, and only a handful of architectural parameters.

The weakness of the BSP approach is that it forms a cost system but not a cost calculus. Having determined the cost of a program, there is no obvious way to use that information to improve the program should the cost be unsatisfactory. Costs can also only be computed when the whole program has been developed, so the BSP cost system does not provide guidance during development.

Another practical cost system is the polytope model [20] which computes the costs of a restricted class of programs: perfectly nested *for* loop programs with constant time loop bodies in which the bounds of each loop are linear expressions in the indices of the enclosing loops. Alternatively, the class of programs can be viewed as a restricted class of recurrence equations. Such computations can be regarded as forming a finite convex set with flat surfaces (a polytope) with dependencies that are regular and local. Linear programming techniques are used to schedule the computations in ways that minimize execution time, or number of processors used. This is used in the design of systolic arrays and in compile-time transformations for loops in imperative programming languages [1].

Roe, in his thesis [26], explored the parallel implementation of a higher-order functional language. He concluded that a parallel constructor was needed to explicitly describe opportunities for parallelism. He also analysed several parallel operations, at a number of levels of detail, and noted a number of examples of places where poor analysis of programs or mistaken assumptions led to programs with poor parallel performance. There are other examples of the problems caused by mismatches between the cost system apparent to the programmer and the real costs of programs in [32]. A number of others have taken the line of using non-standard interpretations of functional languages as an approach to obtaining costs [17]. Linear logic is another approach that offers some hope of a general approach.

As these examples show, it is possible to get useful cost information from programs as long as both the program structure and the possible implementations are sufficiently restricted. However, these approaches all depend on computing the cost of an entire program, and can therefore not be used during modular developments. Developing a cost system in which the cost of pieces can be used to compute the cost of the whole, and which can therefore be applied modularly, is more difficult.

3 A Strategy for a Parallel Cost Calculus

The central problem in building a cost system is to provide the right level of abstraction — one that hides much of the underlying complexity, but that reveals enough for useful decisions about one choice of algorithm over another. There are two parts to this abstraction. The first is to reduce the amount of detail required to a manageable level. This involves restricting the form of programs, the flexibility of the compiler to map computations, and the form of target architectures modelled. We have seen several examples of such restrictions in the previous sections.

Unfortunately the details that are hidden by the cost system are precisely those that are needed when designing programs. So the second part of the abstraction involves finding a way of making *relative* cost information available to programmers without requiring absolute cost information (which needs too much detail).

The Bird-Meertens theory of lists is a useful starting place because the form of programs and their mapping to processors is restricted. The theory is based on list data structures and homomorphic operations on them (called *catamorphisms*). These catamorphisms are higher order functions, parameterised by list constructors and sets of functions cognate to them. Every catamorphism can be evaluated using a single standard recursive (and potentially parallel) schema. Thus all programs can be evaluated using a single skeleton.

There are two specialised forms of this schema that are of particular importance — maps and reductions. Every catamorphism can be expressed as the composition of a map and a reduction, so that these two specialised operations form a basic set in which every program can be written. The computation and communication patterns of maps and reductions are straightforward, and versions of them suitable for restricted parallelism can easily be derived. Few basic implementations are therefore required.

This might seem, at first glance, to be quite a restricted programming language. However, the class of catamorphisms includes all injective functions on lists, and many other list functions turn out to be simple extensions. So the class of the functions that can be computed within the framework is large — more restricted than the BSP model, but less restricted than the polytope model. Models of parallel computation based on sets of skeleton operations such as these are becoming popular [10, 14–16, 31].

Having restricted the programming language to the composition of a fixed set of second-order functions or skeletons, the approach consists of two steps: choosing implementations for the basic operations, and solving the problems of composition.

Any implementation for the basic operations is acceptable, provided the work it does is equivalent to the work of the PRAM implementation. These implementations must be parameterized by the number of processors used, since we cannot in general expect the virtual parallelism of an operation to match the physical parallelism of an implementing architecture.

Composition introduces problems into any parallel cost calculus. To be useful, a cost calculus must have the property that the cost of $g \cdot f$ should be computable from the cost of

g and the cost of f, preferably in some simple way independent of architectural properties. The obvious choice is that the cost of a sequential composition should be the sum of the costs of its components. Without some simple compositionality property, we cannot use modularity within a derivational setting, because there is not enough information available to derive each piece without knowledge of the derivations of all of the other pieces. The additivity of costs can fail in two ways:

- There is the possibility of beginning the computation of g before f has been completely evaluated, overlapping processor usage on the two computations. Simply summing the costs of g and f will overestimate the cost of $g \cdot f$.
- The function $g \cdot f$ may be equal to another composition, say $s \cdot r$, and the new algorithm this composition represents may be cheaper.

The failure of additivity is fundamental — to find the optimal algorithm to compute something, it is necessary to check the costs of all other compositions computing the same function, and this will be expensive.

We propose the following general approach. Equations within the transformation system are initially assumed to be cost-neutral from the programmer's point of view. Situations in which implementers discover that the additive view of costs that programmers maintain is wrong (because it overestimates costs) are fixed by annotating equations involving that composition to indicate which expression implies the cheaper implementation. The equations express functional equalities, but are now understood to express implementation inequalities (that is, two sides of an equation may be implemented differently, with different costs). Annotations are done by implementers, who understand architectures in detail, rather than by programmers who would rather not.

When the composition of two operations has an implementation, involving overlapping, whose cost is lower than the sum of the costs of its components, a new operation representing the composite is defined, a defining equation is added, and this equation is labelled as cost-reducing. Thus, for the example above, we would add an equation

$$\text{newop} \mathrel{\hat{=}} g \cdot f$$

and an annotation to indicate that it is cost-reducing right to left.

When there are different compositions that compute the same function, we annotate the equation connecting them to indicate which side is the cheaper. This induces a preorder on the different representations that compute the same function and allows the equations expressing their functional equality to be used as cost-reducing rewrite rules. If the annotations of enough equations are known, this can be used as the basis for an automated optimizer.

The effect of these two enhancements is to separate the views that programmers and implementers have of composition. From the perspective of the programmer, compositions in programs are regarded as barrier synchronizations of the corresponding computations. Although *newop* and $g \cdot f$ are equivalent as functions, they have different behaviours –

newop is a single schedulable operation, while $g \cdot f$ is the execution of an operation f, upon whose termination another operation g is executed. For the programmer, this means that costs behave additively.

The extent to which the cost that the programmer computes is accurate is determined by how many equations have been labelled. There is always the potential that some long, hitherto unconsidered, composition of functions has a cheap implementation, and so the process of annotating equations does not terminate. However, the programmer's cost calculus and real execution costs can be brought into arbitrarily close agreement by considering longer and longer compositions.

Implementers regard composition as a statement about a data dependency between the entire result object of one operation and the argument of the next. This may be an opportunity to find a cheaper implementation for the composite if possible.

Thus there are effectively two views of operations and equations. Programmers view operations as monolithic operations on some data type, and composition as barrier synchronization. Equations are true equalities but they are labelled with a direction, revealing that one side is cheaper to implement than the other.

Implementers, on the other hand, view equations as detailed schedules of simple steps on a set of processors, with defined communication between the actions on different processors. Composition is an opportunity to overlap the execution of successive operations. Equations are functional identities but behavioural inequalities. Implementers may choose to add extra equations to the programming calculus for the purpose of labelling them, so making the programmers' view of costs more accurate. New named operations, and new equations, provide a way for relative cost information to be conveyed to programmers without breaching the architecture- and processor-independent view of costs that programmers must maintain.

This two-level approach seems to be a good pragmatic approach to a theoretically intractable problem: providing programmers with an abstraction of costs without forcing them to become aware of implementation details. It applies to any parallel programming calculus based on skeletons for which there is a deterministic solution to the mapping problem. This certainly includes language such as Parallel SETL [18], scan vectors [9], P^3L [14] and other skeleton-based languages [13, 16].

In subsequent sections we show the working out of this approach in the context of the Bird-Meertens theory of lists.

4 Bird-Meertens Background

The Bird-Meertens model of parallel computation is based on a fixed set of second-order operations for computing with each data type. These operations are generated, along with a program transformation system, by a categorical construction [21]. All programs are compositions of these second-order operations, and are thus single-threaded, that is, from the programmer's point of view, there is only a single function being evaluated at any given time.

We concentrate on the theory of join or concatenation lists. Join lists have three constructors. The first makes an empty list, the second converts a scalar value to a list of length one, and the third, concatenation or $+\!\!+$, joins two lists to give a new list. Concatenation is associative. Although lists might seem a trivial data type, experience with data-parallel languages suggests that many computations can be reasonably expressed within this framework [9]. We will ignore lists constructed with *cons* or *snoc* since these do not give any opportunity for parallelism (although they do permit pipelining).

A homomorphism on lists is a function that respects the list structure. A function h is a homomorphism if

$$h(a +\!\!+ b) = h(a) \otimes h(b)$$

for some associative binary operation $\otimes$ and

$$h([]) = e$$

where e is the identity of $\otimes$. (If a and b are themselves lists, this equation applies recursively; since the computations of h on the right hand side are independent, homomorphisms naturally create opportunities for parallelism.) By a result known as the First Homomorphism Theorem [4], any homomorphism on lists can be expressed as the composition of a map followed by a reduction. Thus maps and reductions are the basic skeleton operations on lists. Because there are only a fixed set of these operations, it is possible to use only a fixed set of communication and computation patterns. Thus the mapping problem need only be solved for a few cases.

The union of the communication patterns required by the second-order operations defines the total communication needs of any Bird-Meertens list program of any size. We call this union the *standard topology* for the data type. The mapping problem now becomes the problem of embedding the standard topology in the topology of a set of target machines without dilation, that is without implementing single-edge communication in the standard topology by non-constant path length communication in the target interconnect. This will not always be possible for every target architecture, but is possible for many reasonable ones and, more importantly, for some architectures of all classes [30]. The mapping of the standard topology to each new target is done once by a system implementer. For lists, the standard topology is a binary tree.

Once the standard topology has been embedded, all single-edge communication is implemented by constant length paths and therefore takes constant time. Thus we can act as if communication is free, much as the PRAM model does, but with better justification.

Each of the second-order operations is strict and recursion is encapsulated within them as necessary so that issues relating to dynamic behaviour and non-termination do not arise.

The first basic operation is the map of a function f and is written $f*$. It is of type $List(\alpha) \rightarrow List(\alpha)$. Its effect is

$$f*[a_1, a_2, \ldots, a_n] = [fa_1, fa_2, \ldots, fa_n]$$

A variant of the map operation that lifts a binary operation is called *zip* and is defined by

$$[a_1, a_2, \ldots, a_n] \curlyvee_{\oplus} [b_1, b_2, \ldots, b_n] = [a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n]$$

The second basic operation, *reduce*, is written $\oplus/$ (where $\oplus$ is the operation of some monoid with carrier α and e is its identity), has type $List(\alpha) \rightarrow \alpha$, and computes

$$\begin{aligned} \oplus/\,[a_1, a_2, \ldots, a_n] &= a_1 \oplus a_2 \oplus \ldots \oplus a_n \\ \oplus/\,[\,] &= e \end{aligned}$$

With only these two second-order operations we can construct all homomorphisms on lists. Many other, apparently necessary, operations are expressible in terms of these two.

The basic operations on lists can be implemented on any architecture into which the standard topology can be embedded. This includes (of course) sequential uniprocessors and vector architectures, SIMD machines (since the same operation takes place on each step in each processor), shared memory multiprocessors, and distributed memory multiprocessors whose interconnect contains a logarithmic depth spanning tree, such a hypercubes and cube-connected-cycles [30].

Other homomorphic list operations arise sufficiently often in applications that they have been given their own names. We will show that these operations also have more efficient direct implementations than their implementations as compositions of maps and reductions. We suspect that, in general, operations added because they have efficient implementations will turn out to be operations that are natural units in which to think about programming. In other words, good implementations and natural more-abstract operations tend to coincide.

The operation *inits* computes the initial segments of a list, that is

$$\text{inits}\ [a_1, a_2, \ldots, a_n] = [[a_1], [a_1, a_2], [a_1, a_2, a_3], \ldots, [a_1, a_2, \ldots, a_n]]$$

A corresponding operation, *tails*, computes the final segments of a list.

The operation *prefix* (sometimes called *scan*), written $\oplus/\!/$, applied to a list is defined by

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

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

$$[a_1, ..., a_n] \otimes /_{b_0} \oplus [b_1, ..., b_n] = b_0 \otimes a_1 \otimes ... \otimes a_n \oplus b_1 \otimes a_2 \otimes ... \otimes a_n \oplus ... \oplus b_{n-1} \otimes a_n \oplus b_n$$

This operation is of practical importance because linear recurrences occur in many scientific computations and because it isn't obvious that a fast parallel implementation exists for an operation that appears so sequentially dependent. Details of the use of this operation as part of the Bird-Meertens theory of lists are found in [11]. The operation is defined with

a seed b_0 so that it takes arguments of the same length. This greatly simplifies checking conformance conditions in derivations.

A related operation is *recur-prefix* (written $\otimes /\!\!/_{b_0} \oplus$), which computes all values generated by the same linear recurrence:

$$[a_1, \ldots, a_n] \otimes /\!\!/_{b_0} \oplus [b_1, \ldots, b_n] = [b_0, b_0 \otimes a_1 \oplus b_1, ..., b_0 \otimes a_1 \otimes ... \otimes a_n \oplus ... \oplus b_{n-1} \otimes a_n \oplus b_n]$$

5 Shape Vectors

We need a notation to describe the nested structure of a list when argument size is an issue. We use a shape vector, that is a list of the maximum size of subobjects at each level of nesting. Thus a shape vector of the form $[n, m, p]$ describes a list of n elements, the largest of which is a list of length m, where the largest element of any sublist is of size p. The last element of a shape vector describes the size of any remaining substructure in the list — the same list could be described by a shape vector $[n, mp]$ if we do not care about the size of the second nesting level structure. Thus a list with shape vector $[n]$ is equivalent to one with shape vector $[n, 1]$.

Since costs depend on the size of list arguments, we must record the size of the result of each operation as a function of the sizes of its arguments. We do this by labelling operations (and programs) with the sizes of their arguments and results whenever it is necessary to compute costs. For example, if f is a function that takes arguments of size m and produces results of size p then an application of $f*$ is labelled

$$^{[n,p]}f*^{[n,m]}$$

This accurately describes what happens even if f is some complex function applied to a deeply nested list structure.

Figure 1 illustrates the effect on shape vectors of some common list operations. These effects are determined from the definitions of these operations in the previous section. Note that each element of a shape vector is the *maximum* size of the element at that nesting level, so will tend to overestimate the amount of work to be done. There seems no straightforward way to avoid this. Also, if the work to be done depends on the *values* of the list elements, rather than their sizes, our cost system will be unable to take that into account. Fortunately, most of the operations of the theory of lists are oblivious (that is, the control flow does not depend on the values of the data), because most of the second-order operations are strict.

6 Basic Operations

In this section we build implementations for the basic operations of map and reduction for the easy case when the size of the argument list matches the number of processors available to compute the required function. These implementations become the building blocks of later, more complex, implementations. We also compute the parallel time complexities of these basic implementations.

6.1 Implementations

We begin with implementations for our basic operations, maps and reductions, implementations for some ancillary operations, and the definition of some combinators for combining implementations in parallel and sequentially. We use lower case names to denote functions or programs, and upper case names to denote atomic implementations of functions or programs. Implementations are imperative, that is they are schedules for the application of functions on particular processors and for communication actions. Nevertheless, transformations on functions are useful preliminary steps in building implementations.

Let $PAR_p f$ denote p parallel applications of f to adjacent list elements. This corresponds to the *doall* or *forall* loop constructor common to several parallel programming languages. Let $REP_m f$ denote m sequential applications of f to adjacent list elements. This corresponds to the ordinary sequential loop constructor. These combinators satisfy the follow equalities:

$$\begin{aligned} PAR_p g \cdot PAR_p f &= PAR_p(g \cdot f) \\ REP_p g \cdot REP_p f &= REP_p(g \cdot f) \\ REP_m(REP_n f) &= REP_{mn} f \\ PAR_p(PAR_q f) &= PAR_{pq} f \end{aligned}$$

that is, each side of an equation denotes a different implementation of the same function.

We now give implementations of the basic operations, under the assumption that a single top-level list element is allocated to each processor (other assumptions are possible; we discuss this issue in more depth later). Such implementations must use only local communication in the target processor's topology. We use the notation f_n for the application of f to a list whose top-level length is n, that is whose shape vector is of the form $[n, \ldots]$.

We denote a basic (atomic) parallel implementation of f using p processors by

$$PARIMPL_p(f)$$

and a basic sequential implementation of f by

$$SEQIMPL(f)$$

PARIMPL and SEQIMPL denote particular atomic implementations that have been chosen

to be useful and efficient and which might be used in implementing more complex functions. These implementations have the status of "the best algorithm known in the absence of lower bounds", that is they are regarded as fixed over the medium term, but might change with advances in understanding.

We use

$$impl_p(f)$$

to denote the function that maps functions or programs f to implementations on an arbitrary number of processors, p. For some functions, $impl_p$ will be recursively defined in terms of implementations of simpler functions; for others $impl_p$ will map functions to atomic implementations.

The implementation of a map is expressed in terms of simpler implementations as

$$impl_n(f*_n) = PAR_n f$$

in which f is evaluated at each processor independently of the others. There is no communication required.

The implementation of a reduction is slightly more complex. Reduction requires computing subexpressions of the result; since $\oplus$ is associative, these subexpressions can be computed in any order, or concurrently. The optimal parallel way to do so is to compute $\oplus$ applied to adjacent pairs on the first step, then $\oplus$ applied to the results of adjacent first step operations and so on. This computes the final result in time logarithmic in the length of the list and requires a communication topology that is a binary tree. We use $PARIMPL_n(\oplus/\,)$ to denote this implementation.

An implementation primitive that occurs often in implementing complex operations, but is not part of the Bird-Meertens language is the operation *shiftright*, which shifts the members of a list one place to the right in a mapping to processors. We use $SHIFTRIGHT_n$ to denote its implementation.

6.2 Costs

We use the notation t_p to denote the parallel time required for a computation on p processors, and n to denote the (top-level) length of a list.

The cost of a second-order operation depends on the cost of the first order operation(s) that it executes in parallel. Since the costs of first order operations are multiplicative in the total cost, they can be factored into a cost computation at any time. At first, we will be careful to mention these costs explicitly. Later it will be convenient to assume that first order operations take unit time, unless we say otherwise.

We begin with the relationship between costs and combinators:

$$t_1(g \cdot f) = t_1(g) + t_1(f) \quad (1)$$

$$t_n(PAR_n f) = t_1(f) \quad (2)$$

$$t_1(REP_n f) = n \cdot t_1(f) \quad (3)$$

Map. The cost of a parallel map is

$$\begin{aligned} t_n(f*_n) &= t_n(PAR_n f) \\ &= t_1(f) \end{aligned}$$

while the cost of a sequential map is

$$\begin{aligned} t_1(f*_n) &= REP_n(f) \\ &= n \cdot t_1(f) \end{aligned}$$

Reduction. The cost of a parallel reduction with constant-sized partial results is

$$\begin{aligned} t_n(\oplus/_n) &= t_n(PARIMPL_n(\oplus/)) \\ &= \log n \end{aligned}$$

while the corresponding cost of a sequential reduction is

$$t_1(\oplus/_n) = n - 1$$

Including the cost of subsidiary operations multiplicatively does not work if the first order operations (such as f and $\oplus$) produce results of different size from their arguments. When this happens, the separation between time and size properties breaks down. A cost system in which both time and size are treated explicitly quickly becomes unmanageably complex. Costs are the solutions of recursive equations in both cost and size, while sizes are themselves determined by recursive equations. Finding closed form solutions in any but the simplest and most regular case is difficult. Instead, we use different execution time equations for first order operations with different effects on sizes. Practically, there seem only to be a few natural ways in which operations produce results of different size from their arguments; and it is only the *reduction* operation and others that are built from it that exhibit the problem.

Let us now examine the behaviour of reduction when it is applied to an operator whose results are of a different size from its operands. The most obvious example is concatenation, for the concatenation of two lists of length n results in a list of length $2n$. The implementation of reduction as a binary tree still works, but we must take into account the communication time required to move the larger and larger results, and this communication time dominates the computation. On the first step, an element of size 1 is moved between processors, on the second step, a list of length 2, on the third step, a list of length 4 and so on. Thus the total parallel time for $+\!\!+/$ (assuming that the concatenation takes constant time) is

$$t_n(+\!\!+/_n) = (2^0 + 2^1 + \ldots + 2^{\log n - 1}) \cdot t_1(+\!\!+)$$

$$= (n-1) \cdot t_1(+\!\!+)$$

which is also the time for the corresponding sequential reduction.

Concatenation can reasonably be assumed to take constant time, but operators that produce results larger than their operands may take longer in general (for example, multiplication of variable length values).

7 More Complex Operations

Many of the standard operations of the theory of lists are examples of operations for which a more efficient implementation exists than that implied by their expression as the composition of a map and a reduction. In this section we examine several such operations.

7.1 Implementations

Prefix. Recall that the operation $\oplus/\!\!/$ applied to a list is defined by

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

Prefix is a homomorphism and so can be computed by composing a map with a reduction (see Section 8.2 for details). However, the following two-phase implementation is more efficient [9]. The upsweep phase of the algorithm computes a reduction over the argument list. The downsweep phase passes global information back to be merged with values computed on the upsweep. Consider a list of length n, and number the positions in the list beginning at 0. The data flow of the algorithm computing

$$+/\!\!/[3, 1, 2, 1, 4, 1, 1, 3]$$

is shown in Figure 2, where the lists in the Figure show the data held at each processor, and the arrows indicate the communication between processors.

On the first step, the value of all even-numbered elements of the list are passed to odd-numbered elements. During the upsweep phase, the operation carried out at the nodes is shown in Figure 3(a). The value received from the left is kept, and the $\oplus$ operator applied to it and the value calculated at the last step. The result of this reduction operation is then passed right.

For the downsweep, the operations performed at each node are illustrated in Figure 3(b). The values passed from the right are copied to the left, and that value and the value at the beginning of the list at that node are added. One value is transmitted on the first step, three values on the second, and so on, with all but one processor active on the last step. Call this implementation $PARIMPL_n(\oplus/\!\!/)$.

Inits. The *inits* operation is another complex operation for which an efficient direct implementation exists. Consider the computation of the initial segments of a list of length n, stored with one element on each processor. If list elements are copied and shifted repeatedly to the right (with the empty list inserted at the left hand end), then prepending each value to those already present in each processor computes the required list. This is expressed as

$$
\begin{aligned}
&impl_n(inits_n\ x)\ \text{is}\\
&\quad xs := [\cdot]{*}\ x;\\
&\quad ys := xs;\\
&\quad REP_n\{ys := SHIFTRIGHT_n\, ys;\ xs := ys \curlyvee_{+\!+} xs\}
\end{aligned}
$$

Call this implementation $PARIMPL_n(inits)$. Note that it requires adding a path through the leaves to the standard topology (see Section 8.2 for further details).

7.2 Costs

We now consider the complexity of the prefix operation. First we assume that partial results are the same size as operands. The time taken for the upsweep is

$$t_n(\oplus /\!\!/_n)_{upsweep} = (\log n) \cdot t_1(\oplus)$$

as we have already seen, since it is a slight variant on the reduction computation. The time for the downsweep is also

$$t_n(\oplus /\!\!/_n)_{downsweep} = (\log n) \cdot t_1(\oplus)$$

so the total execution time for prefix is

$$t_n(\oplus /\!\!/_n) = (2 \log n) \cdot t_1(\oplus)$$

Prefix is an operation, like *reduce*, whose complexity changes when it is applied to a first order function whose results is larger than its arguments. Consider the implementation of $+\!\!+/\!\!/$ as shown in Figure 4. The analysis of the upsweep is exactly the same as for $+\!\!+/$ so the time for the upsweep is

$$t_n(+\!\!+ /\!\!/_n)_{upsweep} = (n-1) \cdot t_1(+\!\!+)$$

Consider the data flow on the down sweep. The amount of information required by the rightmost processor is large, all of the list elements. On the other hand, this information has accumulated in the last processor during the upsweep. The second-to-last processor needs less information, but it takes longer to get it since some of it must be transmitted by the last processor.

Divide the processors into groups, depending on their distance in the tree from the last

processor – thus the last processor is in group 0, the second last processor in group 1, the next two processors preceding it in group 2, and so on. Then the time taken to correctly deliver the required data to a processor in group i is

$$t_n = (n - 2^i)(i)$$

where the term $n - 2^i$ is the volume of data to be sent to a processor in group i, and i is the distance the data must travel from the last processor. This quantity is maximised when $2^i = n/2$, that is the most expensive processor to reach is the one holding the initial elements of the second half of the list. Thus the time for the downsweep is obtained by substituting in the equation above to give

$$t_n(+\!\!/\!\!/_n)_{downsweep} = \frac{n}{2}(\log n - 1)$$

and the time for both phases is

$$t_n(+\!\!/\!\!/_n) = \frac{n}{2}(\log n + 1) - 1$$

The cost of *inits* is obtained by summing the steps of the corresponding implementation and so is

$$t_n(inits_n) = 1 + 2n$$

8 Unmatched Size Implementations

We now consider implementations parameterised by the number of processors in the target architecture. Eventually we will show that equations can be directed independently of this parameter, and so it can be factored out of considerations of (relative) cost.

Suppose that the argument list length (at the top level) is n and the number of processors in the target architecture is p. There are two different dimensions along which allocation choices can be made:

1. Allocate list objects to processors in a round robin fashion; or
2. Allocate list objects to processors by segments, so that the first n/p of them are placed in the first processor, the next n/p in the second processor and so on.

The second dimension is:

a. allocate top level objects, that is if the list has shape vector $[n, m, q]$, allocate n objects each of size up to qm; or

b. allocate bottom level objects, that is allocate nmq distinct objects

Alternative (2) is better than alternative (1) because it reduces the interconnect traffic during operations where adjacency in the list is important (which includes reductions). Choosing alternative (a) is better than (b) because our list operations work from the outer level of list structure inwards, and this arrangement spreads the top level uniformly across the available processors. We therefore assume this embedding of the standard topology of the program in the standard topology of the target architecture.

We must now give implementations for the two basic operations assuming this embedding. Since more than one element of a list is held by each processor, processors must carry out steps related to more than one list element.

We assume that the number of processors used is less than or equal to the length of the outermost level of the list. It is not difficult to avoid this assumption but it does make explanations more complicated. For simplicity, we also assume that p divides n.

The implementation of a function actually has a different type from the function itself, because the implementation acts on a list that has been evenly spread across processors. For example, if $f*$ is applied to a list with shape $[n] +\!\!+ s$ then its implementation is applied to a list whose shape is $[p, n/p] +\!\!+ s$. Thus we must convert a program expecting a list as input into a program expecting a list of lists.

We begin by defining an operation $distribute_p$ which takes a list and breaks it up into p segments, by implication distributing them to p processors. An implementation for $distribute_p$ is shown in Figure 5; we use $DIST_p$ to denote this implementation. The parallel time to distribute a list depends on the communication associated with each step in the Figure, that is

$$\begin{aligned} \frac{m}{2} + \frac{m}{4} + \ldots + \frac{m}{p} &= m \sum_{i=1}^{\log p} \frac{1}{2^i} \\ &= m(1 - \frac{1}{p}) \end{aligned}$$

This distribution operation must of course occur once at the beginning of each program, and we can arguably not include its cost in the cost of the program. In the first place, its cost is hidden in the cost of loading the program. In the second place, as disk arrays become more common, the argument list(s) may already be stored in segments local to each processor.

A corresponding operation is required to reassemble the result list into a list of the proper type. No new operation is required for this – it is just $+\!\!+/$ for, when applied to a list with shape $[p, n/p] +\!\!+ s$, $+\!\!+/$ returns a list of shape $[n] +\!\!+ s$. Furthermore, $+\!\!+/$ is a postinverse to *distribute*, that is

$$+\!\!+ / \cdot distribute = id \tag{4}$$

(The usefulness of this identity was pointed out by Paul Roe in [27].)

Thus a program of the form

$$S3 \cdot S2 \cdot S1$$

is equivalent to

$$(+\!\!+/ \cdot distribute_p) \cdot S3 \cdot (+\!\!+/ \cdot distribute_p) \cdot S2 \cdot (+\!\!+/ \cdot distribute_p) \cdot S1 \cdot (+\!\!+/ \cdot distribute_p)$$

or, rearranging parentheses,

$$+\!\!+/ \cdot (distribute_p \cdot S3 \cdot +\!\!+/) \cdot (distribute_p \cdot S2 \cdot +\!\!+/) \cdot (distribute_p \cdot S1 \cdot +\!\!+/) \cdot distribute_p$$

This form suggests that, rather than implementing f, we should implement

$$distribute_p \cdot f \cdot +\!\!+/$$

A program then becomes a composition of these implementations, preceded by a distribution step at the beginning of each program, and ending with a collection step. The implementation of each operation assumes that its argument has already been distributed across the processors and is responsible for ensuring that its result is similarly distributed.

If h is an arbitrary homomorphism then an implementation must define a function that can be applied to the argument spread over the set of processors. There must be a function h' which depends on h and whose effect on a part of the argument is like the effect of h. The theory of lists provides some help with finding such a specialisation of h, further support for the argument that skeleton-based languages should be defined mathematically rather than as *ad hoc* collections of operations.

An important result of the theory of lists, called the Promotion Theorem, is expressed by the following identity (5). Suppose that h is a list homomorphism. We have already noted that h can be expressed as the composition of a map and a reduction, say

$$h \;=\; g/ \cdot f*$$

Then the Promotion Theorem is the identity

$$h \cdot +\!\!+/ \;=\; g/ \cdot h* \tag{5}$$

Thus an implementation of h can instead be an implementation of the right hand side of this equation:

$$distribute_p \cdot h \cdot +\!\!+/ \;= distribute_p \cdot g/ \cdot h* \tag{6}$$

which applies h in parallel p times (so each application of h is sequential), followed by a reduction with g (which depends on h), followed by a redistribution of the result of this reduction. Thus any list homomorphism can be implemented by applying it to sublists, then joining the results of this application together using a gluing function, g, that depends on the homomorphism, and then redistributing. This gives an immediate implementation for all list homomorphisms.

This reassures us that highly parallel implementations always exist and gives us a tech-

nique for building a distributed implementation of an arbitrary list homomorphism. Notice that the form of identity (5) ends with a reduction step. This will tend to collect the result in a single processor, necessitating an expensive *distribute* to spread list elements across the processors again. This makes such simple implementations expensive and leads us to look for implementations that leave list segments in place.

Two particular instances of equation (5) that are of interest to us are given by the following equations

$$\begin{aligned} f* \cdot +\!\!+/ &= +\!\!+/ \cdot f** \\ \oplus/ \cdot +\!\!+/ &= \oplus/ \cdot (\oplus/)* \end{aligned}$$

A special case of the second equation, namely

$$+\!\!+/ \cdot +\!\!+/ = +\!\!+/ \cdot (+\!\!+/)* \tag{7}$$

will be of particular use in building implementations in Section 9.

Some examples will illustrate the use of the identity (5). We assume that f and $\oplus$ map operands of size q to size q. In each case we transform a program into a form that can be directly implemented.

$$\begin{aligned} & {}^{[n,q]}f* \cdot^{[n,q]} (\oplus/)^{*[n,m,q]} \\ &= f* \cdot +\!\!+/ \cdot distribute_p \cdot (\oplus/) * \cdot +\!\!+/ \cdot distribute_p \\ &= f* \cdot +\!\!+/ \cdot distribute_p \cdot +\!\!+/ \cdot (\oplus/)** \cdot distribute_p \\ &= f* \cdot +\!\!+/ \cdot (\oplus/)** \cdot distribute_p \\ &= {}^{[n,q]}+\!\!+/ \cdot^{[p,n/p,q]} f** \cdot^{[p,n/p,q]} (\oplus/)** \cdot^{[p,n/p,m,q]} distribute_p^{[n,m,q]} \end{aligned}$$

Here each map operation is implemented by a nested map operation. The $+\!\!+/$ moves to the end of the program, where it becomes the final collection step. Note that we discover that one of the inserted copies of the identity (4) turned out to be redundant and was removed.

The next example illustrates the use of an extra insertion of the identity to redistribute a list that has been collected into a single processor. After the reduction the list of shape $[m]$ exists in a single processor. It is necessary to redistribute this new smaller list across the available processors to make parallelism available for the following map operation.

$$\begin{aligned} & {}^{[m]}f* \cdot^{[m]} (+/)^{[n,m]} \\ &= f* \cdot +\!\!+/ \cdot distribute_p \cdot (+/) \cdot +\!\!+/ \cdot distribute_p \\ &= {}^{[m]}+\!\!+/ \cdot^{[p,m/p]} f** \cdot^{[p,m/p]} distribute_p \cdot^{[m]} (+/) \cdot^{[p,m]} (+/)* \cdot^{[p,n/p,m]} distribute_p^{[n,m]} \end{aligned}$$

Observe that the term

$$distribute_p \cdot +\!\!+/$$

although not an identity in general, is one when the shapes match. So we have

$$^{[p,n/p]}distribute_p \cdot^{[n]} +\!\!+/^{[p,n/p]} = id$$

under these particular circumstances. We will make use of this identity to remove redundant data movement operations in the analysis that follows.

8.1 Basic Operations

For maps we have that

$$\begin{aligned} & impl_p(distribute_p \cdot f*_n \cdot +\!\!+/_p) \\ &= impl_p(distribute_p \cdot +\!\!+/_p \cdot f*_{n/p}*_p) \\ &= PAR_p(REP_{n/p} f_1) \end{aligned}$$

so

$$\begin{aligned} t_p(f*_n) &= t_1(REP_{n/p} f_1) \text{ by equation 2} \\ &= \frac{n}{p} \cdot t_1(f_1) \end{aligned}$$

For reductions in which the size of partial results is the same as the size of arguments, we have

$$\begin{aligned} & impl_p(distribute_p \cdot \oplus/_n \cdot +\!\!+/_p) \\ &= impl_p(distribute_p \cdot \oplus/_p \cdot (\oplus/_{n/p})*_p) \\ &= DIST_p \cdot PARIMPL_p(\oplus/) \cdot PAR_p(SEQIMPL_{n/p}(\oplus/)) \end{aligned}$$

from which we compute

$$\begin{aligned} t_p(\oplus/_n) &= \log p \cdot t_1(\oplus) + t_1(SEQIMPL_{n/p}(\oplus/)) \\ &= (\log p + (\frac{n}{p} - 1)) \cdot t_1(\oplus) \end{aligned}$$

to which the cost of $DIST_p$ is added if necessary.

For reductions in which the first order operation increases the size of partial results, the cost on p processors is made up of the cost of doing sequential reductions on segments of the list of length n/p; this takes time $n/p - 1$ and produces partial results of size n/p. The parallel reduction then takes time $(p-1)n/p$ for p processors on operands of initial size n/p. Thus the total execution time is

$$t_p(\circledast/_n) = (p-1)\frac{n}{p} + (\frac{n}{p} - 1)$$

$$= \quad n - 1$$

There is one interesting special case for an operation of this kind:

$$^{[n]}\!+\!\!+\!/^{[p,n/p]}$$

can reasonably be assigned a cost of zero, since it requires no data movement, but simply a different view of the data present in each processor.

Note that we are using explicit constants in the cost computations. These constants are obviously not of any practical interest since implementations will conceal constants that are often of greater magnitude. However, making constant values explicit often makes it easier to see how a cost formula was computed and so we leave them in.

8.2 More Complex Operations

We now turn to implementations on p processors of the more complex operations we have already examined.

Prefix. Prefix is expressed as a homomorphism like this:

$$\odot/\!/(x +\!\!+ y) = (\odot/\!/x) \oplus (\odot/\!/y)$$

where

$$u \oplus v = u +\!\!+ ((last\ u) \odot)*v$$

Since prefix applied to a singleton list computes a list with a single singleton element, it is easy to see that

$$\odot/\!/ = \oplus/ \cdot ([\cdot])*$$

This immediately gives us an implementation of prefix in terms of the implementations of maps and reductions. However, the reduction with $\oplus$ is an expensive one, for two reasons: first it contains a reduction with $+\!\!+$, which we have already seen takes linear time even in parallel, and second, it leaves its result in a single processor so that it would have to be followed by an expensive distribution operation.

If we are willing to make a small addition to the standard topology, namely adding the requirement that there is a path through the leaves of the spanning tree, then a better implementation of this and other operations can be constructed. This additional requirement is not a strong one — most topologies of practical interest (hypercube, cube-connected-cycles) already contain such a cycle.

A better implementation for prefix is computed by:

- computing a sequential prefix in each processor on the segment of length n/p it contains;

- computing the parallel prefix of the last value in each processor in a parallel way, using the tree algorithm, and then shifting the resulting values to the right (inserting zero at the left and discarding the rightmost value).
- adding the shifted value at each processor to each of the values in that processor sequentially.

This is illustrated on the following list, stored in four processors. The initial value of the list is

P0	P1	P2	P3
1	4	7	10
2	5	8	11
3	6	9	12

After the first sequential prefix, the values are

P0	P1	P2	P3
1	4	7	10
3	9	15	21
6	15	24	33

A parallel prefix is computed for the values $[6, 15, 24, 33]$ which are then shifted right to get

P0	P1	P2	P3
0	6	21	45
1	4	7	10
3	9	15	21
6	15	24	33

and these values are added to the values below them to give

P0	P1	P2	P3
1	10	28	55
3	15	36	66
6	21	45	78

The implementation becomes

$$
\begin{aligned}
&impl_p(distribute_p \cdot \oplus/\!/_n \cdot +\!\!+/\,) \text{ is} \\
&\quad xs := PAR_p(SEQIMPL(\oplus/\!/)_{n/p}); \\
&\quad ys := PARIMPL_p(\oplus/\!/) \cdot last *_p xs; \\
&\quad zs := SHIFTRIGHT_p\, ys; \\
&\quad ts := zs \curlyvee_{\circledast} xs; \\
&\qquad \text{where } a \circledast bs = (a\oplus) * bs
\end{aligned}
$$

Notice that this implementation absorbs both the flatten and the distribute operations, that is it assumes its argument is distributed over processors and *it leaves its results distributed* in the same way. The resulting costs are

$$\begin{aligned} t_p(xs) &= \frac{n}{p} \\ t_p(ys) &= 2\log p + 1 \\ t_p(zs) &= 1 \\ t_p(ts) &= t_1(z \circledast x) = \frac{n}{p} \\ t_p(\oplus /\!\!/_n) &= 2(\frac{n}{p} + \log p + 1) \end{aligned}$$

Since a prefix computation generates n values, its execution time is bounded below by n/p; since one of these values depends on all the other values in the list, and hence requires communication from all the other processors, its execution time is also bounded below by $\log p$. Hence the time derived for the implementation above is (up to constants) as good as we can do.

Inits. Recall that the operation *inits* computes the initial segments of a list, that is

$$\text{inits } [a_1, a_2, \ldots, a_n] = [[a_1], [a_1, a_2], [a_1, a_2, a_3], \ldots, [a_1, a_2, \ldots, a_n]]$$

The *inits* function is expressed as a homomorphism like this:

$$inits(x +\!\!+ y) = (inits\ x) \oplus (inits\ y)$$

where

$$u \oplus v = u +\!\!+ ((last\ u) +\!\!+)* v$$

(Note the similarity to the computation of prefix.) Thus we can express inits as

$$inits = \oplus/ \cdot ([\cdot])*$$

from which we can immediately get an implementation. It has the same flaws as the implementation of prefix — it moves too many values, and it leaves them in a single processor.

A better way to implement this operation is by circulating values along the path added to the standard topology, with each processor accumulating the values it needs to form its segments. On the initial step each processor computes the local initial segments of the part of the list it contains. The concatenation of its local part 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. When the values from the first processor have reached the last processor

(after p steps) the computation is complete. The process is illustrated below.

$P0$	$P1$	$P2$	$P3$
a_1	a_3	a_5	a_7
a_2	a_4	a_6	a_8

After the first step, each processor has computed the following partial initial segments:

$P0$	$P1$	$P2$	$P3$
a_1	a_3	a_5	a_7
$a_1\,a_2$	$a_3\,a_4$	$a_5\,a_6$	$a_7\,a_8$

Each processor then passes its sublist to its right neighbour, where it is prepended to give:

$P0$	$P1$	$P2$	$P3$
a_1	$a_1\,a_2\,a_3$	$a_3\,a_4\,a_5$	$a_5\,a_6\,a_7$
$a_1\,a_2$	$a_1\,a_2\,a_3\,a_4$	$a_3\,a_4\,a_5\,a_6$	$a_5\,a_6\,a_7\,a_8$

The same values are then passed on a further step to a neighbour two steps from their original processor and the prepending repeated, for a total of p steps. We can write this as

$impl_p(inits_n\ xs)$ is
$ys := inits_{n/p} *_p\ xs$;
$zs := last *\ ys$;
$REP_p\{zs := SHIFTRIGHT_n zs;\ \{ys := zs \curlyvee_{\circledast}\ ys,$ where $a \circledast bs = (a +\!\!+) * bs\}\}$

The total cost is

$$t_p = p\left(\frac{n}{p} + 1\right)\frac{n}{p} + \frac{n}{2p}(\frac{n}{p} + 1)$$

because the cost of computing the zs is overlapped with shifting them right, or

$$t_p = \frac{n^2}{p} + n + \frac{n^2}{2p^2} + \frac{n}{2p}$$

Since *inits* requires the computation of about n^2 values, it is clear that its execution time is bounded below by n^2/p. Thus the implementation above is of the right order. The analogue, *tails* has the same cost.

Recur-reduce. Recall that *recur-reduce* (written $\otimes/_{b_0}\oplus$), is defined by

$$[a_1, ..., a_n] \otimes/_{b_0} \oplus [b_1, ..., b_n] = b_0 \otimes a_1 \otimes ... \otimes a_n \oplus b_1 \otimes a_2 \otimes ... \otimes a_n \oplus ... \oplus b_{n-1} \otimes a_n \oplus b_n$$

Recur-reduce is expressed as a reduction as follows:

$$x \otimes /_{b_0} \oplus\ y = \begin{cases} b_0, & \text{if } \#x (= \#y) = 0 \\ b_0 \otimes \pi_1 A \oplus \pi_2 A, & \text{if } \#x (= \#y) \neq 0 \end{cases}$$

where

$$\begin{aligned} A &= \oslash / \ (x \curlyvee_{\otimes}\ y), \\ a \otimes b &= (a, b), \\ (a, b) \oslash (c, d) &= (a \otimes c, b \otimes c \oplus d), \\ \pi_1(a, b) = a, &\quad \text{and} \quad \pi_2(a, b) = b \end{aligned}$$

This set of equations immediately gives an implementation from which we compute, using the cost of implementation of reduction, that

$$\begin{aligned} t_p(x \otimes /_{b_0} \oplus\ y) &= 4 + t_p(A) \\ t_p(A) &= t_p(\oslash / \ ) + t_p(x \curlyvee_{\otimes}\ y) \\ &= 3(\log p + \frac{n}{p} - 1) + \frac{n}{p} \end{aligned}$$

Thus the total cost is

$$\begin{aligned} t_p &= 4 + 3(\log p + \frac{n}{p} - 1) + \frac{n}{p} \\ &= 1 + 3 \log p + 4n/p \end{aligned}$$

Recur-prefix. Recall that *recur-prefix* (written $\otimes /\!\!/_{b_0} \oplus$), is defined by

$$[a_1, ..., a_n] \otimes /\!\!/_{b_0} \oplus\ [b_1, \ldots, b_n] = [b_0, b_0 \otimes a_1 \oplus b_1, ..., b_0 \otimes a_1 \otimes ... \otimes a_n \oplus ... \oplus b_{n-1} \otimes a_n \oplus b_n]$$

Recur-prefix is expressed as a prefix as follows:

$$x \otimes /\!\!/_{b_0} \oplus\ y = \begin{cases} [b_0], & \text{if } \#x (= \#y) = 0 \\ [b_0] +\!\!+ \ (b_0 \circledast) * (\oslash /\!\!/ (x \curlyvee_{\otimes}\ y)), & \text{if } \#x (= \#y) \neq 0 \end{cases}$$

where

$$\begin{aligned} a \otimes b &= (a, b), \\ (a, b) \oslash (c, d) &= (a \otimes c, b \otimes c \oplus d), \\ b_0 \circledast (a, b) &= b_0 \otimes a \oplus b \end{aligned}$$

As before we use the implementation of prefix and compute

$$\begin{aligned} & t_p(x \otimes /\!\!/_{b_0} \oplus y) \\ = \;& \frac{n}{p} + t_p((b_0 \circledast)*) + t_p(\oslash /\!\!/) + t_p(x \curlyvee_{\otimes} y) \\ = \;& \frac{n}{p} + 2\frac{n}{p} + 3 \cdot 2(\frac{n}{p} + \log p + 1) + \frac{n}{p} \\ = \;& 10\frac{n}{p} + 6 \log p + 6 \end{aligned}$$

where appending b_0 to the beginning of the list requires shifting the list one place to the right, moving an element of size n/p.

For simplicity, when $[b_1, ..., b_n] = [id_{\otimes}, ..., id_{\otimes}]$ and $b_0 = id_{\otimes}$, we write

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

The complexity of these operations is the same as the general forms except for the first step of each, and the map and concatenation steps at the end of the prefix computation.

By way of summary we give the costs of the operations we have discussed in Figure 6. This will be useful for the subsequent sections.

9 Using Costs with Equations

We now show how cost information is used to direct equations of the theory.

Example 1

$$(f \cdot g)* = f* \cdot g* \tag{8}$$

The cost of the left hand side is given by

$$LHS \; t_p = \frac{2n}{p}$$

and the cost of the right hand side by

$$RHS \; t_p = \frac{n}{p} + \frac{n}{p}$$

This equation is cost-neutral, with both sides requiring the same number of operations. (Note, though, that the left hand side is to be preferred on many architectures since it requires only local synchronizations. See the paper [12] for an extensive discussion of this and related issues.)

Example 2

$$f* \cdot \#/ \; = \#/ \; \cdot f** \tag{9}$$

To analyse this equation we look at the implementations of each side, omitting the initial $distribute_p$ from the beginning of both sides. The left hand side becomes

$$\begin{aligned} & distribute_p \cdot f* \cdot \#/ \; \cdot distribute_p \cdot \#/ \; \cdot \#/ \\ = \; & {}^{[p,nm/p,q]}f** \cdot^{[p,nm/p,q]} distribute_p \cdot^{[nm,q]} \#/ \; \cdot^{[p,nm/p,q]} (\#/ \,) *^{[p,n/p,m,q]} \\ = \; & {}^{[p,nm/p,q]}f** \cdot^{[p,nm/p,q]} (\#/ \,)*^{[p,n/p,m,q]} \end{aligned}$$

Note the use of equation 7 on the first step. The cost of this implementation is nmq/p for the $f**$, and $(n/p-1)mq$ for the $(\#/\,)*$, so

$$t_p(LHS) = \frac{2nmq}{p} - mq$$

The right hand side implementation becomes

$$\begin{aligned} & distribute_p \cdot \#/ \; \cdot \#/ \; \cdot distribute_p \cdot f** \cdot \#/ \\ = \; & {}^{[p,nm/p,q]}(\#/ \,) * \cdot^{[p,n/p,m,q]} distribute_p \cdot^{[n,m,q]} \#/ \; \cdot^{[p,n/p,m,q]} f***^{[p,n/p,m,q]} \\ = \; & {}^{[p,nm/p,q]}(\#/ \,) * \cdot^{[p,n/p,m,q]} f***^{[p,n/p,m,q]} \end{aligned}$$

The cost of this implementation is $(n/p-1)mq$ for the $(\#/\,)*$, and $n/p \cdot mq$ for the $f***$. The total cost is

$$t_p(RHS) = \frac{2nmq}{p} - mq$$

so this identity is cost neutral.

Example 3

$$+/ * \cdot \#/ \; = \#/ \; \cdot +/ * * \tag{10}$$

This equation is a special case of the equation above; it is interesting because it illustrates how the use of a suboperation that reduces argument sizes can affect the cost of two different computation forms.

The implementation of the left hand side is (using Example 2)

$${}^{[p,nm/p,1]}(+/\,)** \cdot^{[p,nm/p,q]} (\#/ \,)*^{[p,n/p,m,q]}$$

The cost of this implementation is $nm/p(q-1)$ for the $(+/\,)*$, and $(n/p-1)mq$ for the $(\#/\,)*$. The total cost is

$$t_p(LHS) = \frac{2nmq}{p} - mq$$

The implementation for the right hand side is

$$^{[p,nm/p,1]}(\#/\,)* \cdot ^{[p,n/p,m,1]}(+/\,)***^{p,n/p,m,q]}$$

with costs $(n/p-1)m$ for the $(\#/\,)*$, and $nm/p(q-1)$ for the $(+/\,)***$, for a total of

$$t_p(RHS) = \frac{nmq}{p} - m$$

so the right hand side implementation is cheaper. We direct the equation from left to right.

Example 4

$$\oplus/ \cdot \#/ \;= \oplus/ \cdot (\oplus/\,)* \tag{11}$$

Note again the use of equation 7 on the first step. The implementation of the left hand side is

$$\begin{aligned}
&distribute_p \cdot \oplus/ \cdot \#/ \cdot distribute_p \cdot \#/ \cdot (\#/\,)* \\
&= {}^{[p,q/p]}distribute_p \cdot^{[q]} \oplus/ \cdot^{[p,q]} (\oplus/\,)* \cdot^{[p,nm/p,q]} distribute_p \cdot^{[nm,q]} \#/ \cdot^{[p,nm/p,q]} (\#/\,)*^{[p,n/p,m,q]} \\
&= {}^{[p,q/p]}distribute_p \cdot^{[q]} \oplus/ \cdot^{[p,q]} (\oplus/\,)* \cdot^{[p,nm/p,q]} (\#/\,)*^{[p,n/p,m,q]}
\end{aligned}$$

The cost of this implementation, ignoring the final redistribution, is $q \log p$ for the final reduction, $(nm/p-1)q$ for the $(\oplus/\,)*$, and $(n/p-1)mq$ for the $(\#/\,)*$, for a total of

$$t_p(LHS) = q \log p + \frac{nm}{p}(q+1) - mq - 1$$

The implementation of the right hand side is

$$\begin{aligned}
&distribute_p \cdot \oplus/ \cdot \#/ \cdot distribute_p \cdot (\oplus/\,)* \cdot \#/ \\
&= {}^{[p,q/p]}distribute_p \cdot^{[q]} \oplus/ \cdot^{[p,q]} (\oplus/\,)* \cdot^{[p,n/p,q]} distribute_p \cdot^{[n,q]} \#/ \cdot^{[p,n/p,q]} (\oplus/\,)**^{[p,n/p,m,q]} \\
&= {}^{[p,q/p]}distribute_p \cdot^{[q]} \oplus/ \cdot^{[p,q]} (\oplus/\,)* \cdot^{[p,n/p,q]} (\oplus/\,)**^{[p,n/p,m,q]}
\end{aligned}$$

The cost of this implementation is $q \log p$ for the final reduction, $(n/p-1)q$ for the $(\oplus/\,)*$, and $n/p(m-1)q$ for the $(\oplus/\,)**$, for a total of

$$t_p(RHS) = \frac{nq}{p}(m+1) - q - \frac{nq}{p}$$

The right hand side implementation is cheaper so we direct the equation from left to right.

Example 5

$$\oplus/\!/ = (\oplus/\,)* \cdot \mathrm{inits}$$

The cost of the left hand side is

$$t_p(LHS) = 2(\log p + n/p + 1)$$

The implementation of the right hand side is

$$\begin{array}{l} distribute_p \cdot (\oplus/\,) * \cdot +\!\!+ /\ \cdot distribute_p \cdot inits \cdot +\!\!+ / \\ = \quad {}^{[p,n/p,1]}(\oplus/\,)** \cdot^{[p,n/p,n]} impl_p(inits)^{[p,n/p]} \end{array}$$

The cost of this implementation is $m/p(n-1)$ for the $(\oplus/\,)**$, and $n^2/p + n + n^2/2p^2 + n/2p$ for the implementation of *inits*, for a total of

$$t_p(RHS) = \frac{2n^2}{p} - \frac{n}{2p} + \frac{n^2}{2p^2} + n$$

We direct the equation from right to left.

Example 6

$$inits = +\!\!+\!\!/\!/ \cdot ([\cdot])*$$

The cost of the left hand side is

$$t_p(LHS) = \frac{n^2}{p} + n + \frac{n^2}{2p^2} + \frac{n}{2p}$$

The implementation of the right hand side is

$$\begin{array}{l} distribute_p \cdot +\!\!+ /\ \cdot (distribute_p \cdot +\!\!+\!\!/\!/ \cdot +\!\!+ /\,) \cdot distribute_p \cdot ([\cdot]) * \cdot +\!\!+ / \\ = \quad {}^{[p,n/p,n]} impl_p(+\!\!+\!\!/\!/) \cdot^{[p,n/p,1]} distribute_p \cdot^{[n,1]} +\!\!+ /\ \cdot^{[p,n/p,1]} ([\cdot])**^{[p,n/p]} \\ = \quad {}^{[p,n/p,n]} impl_p(+\!\!+\!\!/\!/) \cdot^{[p,n/p,1]} ([\cdot])**^{[p,n/p]} \end{array}$$

The cost of this implementation is

$$\begin{array}{l} \frac{n}{2}(\log p + \frac{n}{p} + 1) - 1 + \frac{n}{p} \\ \quad = \quad \frac{n \log p}{2} + \frac{n^2}{2p} + \frac{n}{p} + \frac{n}{2} - 1 \end{array}$$

We direct the equation from right to left.

Example 7

$$\otimes /_{id_\otimes} \oplus \ = \ \oplus/\ \cdot \otimes/ * \cdot tails \tag{12}$$

The cost of the left hand side is

$$t_p(LHS) = \frac{4n}{p} + 3\log p + 1$$

while the implementation of the right hand side is dominated by the cost of computing the *tails* operation, which has complexity of order n^2/p. The left hand side is clearly less costly since it represents a more direct way of computing the desired result. This equation is the general form of Horner's Rule for evaluating polynomials. We direct the equation from right to left.

Example 8

$$\otimes /\!\!/_{id_\otimes} \oplus \;=\; (\otimes /_{id_\otimes} \oplus) * \cdot \mathit{inits} \tag{13}$$

The cost of the left hand side is

$$LHS\ t_p = \frac{10n}{p} + 6\log p + 6$$

while the cost of the right hand side is dominated by the computation of *inits*. Again the direct formulation on the left hand side is less costly. We direct the equation from right to left. Note the parallel with the equation defining *prefix*.

Figure 7 summarizes the cost-reducing directions of the equations in these examples.

It is not necessarily the case that all equations of a theory could be directed in this way. For the Bird-Meertens theory and implementation strategy we have suggested, communication plays little role in costs. Thus operations tend to have costs that increase monotonically with the sizes of their arguments, and with outer shapes more important than inner ones. This helps to make costs well behaved.

10 An Example Derivation

With this information about equations and their effects on costs we are able to illustrate a derivation driven by cost minimization. Of course, simply minimizing cost is not enough to derive all useful programs, so that the existence of a cost system does not immediately imply an automated derivation assistant. Experience suggests that most derivations actually have three phases: the first increases the cost as compact parts of the given specification are "expanded out", the second rearranges without having much effect on the overall cost, while the third minimizes cost by reversing "expansions". However, the example of the maximum segment sum is almost entirely driven by cost minimization.

The Maximum Segment Sum problem is: given a list of integers, find the greatest sum of values from a contiguous sublist. It is of interest because there are efficient but non-obvious algorithms to compute it, both sequentially and in parallel. The derivation of a sequential

algorithm is given in [5]. Here we derive a new parallel algorithm that uses the recur-reduce and recur-prefix operations introduced earlier. It begins from an obviously correct solution: compute all of the subsegments, sum the elements of each, and select the largest of the sums. The symbol $\uparrow$ represents the binary maximum function.

$$
\begin{aligned}
mss &= \quad \{ \text{ definition } \} \\
&\quad \uparrow/ \cdot +/* \cdot segs \\
&= \quad \{ \text{ by definition, } segs = \#/ \cdot \text{tails}* \cdot \text{inits} \ \} \\
&\quad \uparrow/ \cdot +/* \cdot \#/ \cdot tails* \cdot inits \\
&= \quad \{ \text{ equation 10, cost-reducing } \} \\
&\quad \uparrow/ \cdot \#/ \cdot +/** \cdot tails* \cdot inits \\
&= \quad \{ \text{ equation 11, cost-reducing } \} \\
&\quad \uparrow/ \cdot \uparrow/* \cdot +/** \cdot tails* \cdot inits \\
&= \quad \{ \text{ map promotion, equation 8 , cost-neutral } \} \\
&\quad \uparrow/ \cdot (\uparrow/ \cdot +/* \cdot tails)* \cdot inits \\
&= \quad \{ \text{ equation 12, cost-reducing } \} \\
&\quad \uparrow/ \cdot (+/_0\uparrow)* \cdot inits \\
&= \quad \{ \text{ equation 13, cost-reducing } \} \\
&\quad \uparrow/ \cdot +/\!/_0\uparrow
\end{aligned}
$$

We can compute the change in cost between the initial version and the final version. We begin with the expanded initial version, applied to a list of length n and labelled with the shape vectors of the intermediate results.

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

We now sum the costs for each step of this composition beginning from the right.

	t_p
$inits$	$\frac{n^2}{p} + n + \frac{n^2}{2p^2} + \frac{n}{2p}$
$tails*$	$\frac{n(n-1)n}{2p}$
$\#/$	$(n-1)n^2$
$+/*$	$\frac{n^2}{p}n$
$\uparrow/$	$\frac{n}{p} + \log p - 1$

Thus the total cost is $t_p = O(n^3)$, regardless of p, so that this version of the solution takes at least cubic time no matter how much parallelism is used.

For the final version we have

$$[1]\uparrow / \ .^{[n]} \ +/\!\!/_0 \uparrow^{[n]}$$

so that

$$\begin{aligned} t_p &= \frac{n}{p} + \log p - 1 + 6\log p + \frac{10n}{p} + 6 \\ &= \frac{11n}{p} + 7\log p + 5 \end{aligned}$$

a dramatic improvement.

It is clear that the cost calculus we have outlined here could be integrated into a transformation assistant and we have begun some steps in this direction. However, most derivations do not proceed by strict cost minimization, and it will be interesting to see to what extent cost information can be used to inform the early stages of a derivation.

11 Conclusions

A practical cost system for use in calculational style software derivation must abstract from the complexities of actual implementations on parallel machines. In particular, the full complexity of the mapping problem must be avoided, or else there is no hope of subexponential cost computations. To be practical it must also be intimately connected to the programming calculus so that changes in cost can be computed from the application of refinements or transformations rather than computing whole-program costs repeatedly. Because parts of a large program are developed independently, it is also important that a cost system be compositional so that the cost of a piece does not depend on the design of all of the other

pieces.

We have presented an approach to cost calculi which allows costs to be viewed from two perspectives. In an implementation view, costs are based on full information about mapping and scheduling, from which the execution time of operations is computed, parameterized by the number of processors used. Operations are designed to implement communication in constant time and therefore communication can be ignored in cost computation.

Compositions of operations can be examined for opportunities to implement the composites more efficiently than the sum of the component pieces would suggest. Whenever this occurs, a new primitive operation is defined, and its defining equation is annotated with a cost-reducing direction in the programming calculus. Since the number of times such discrepancies can arise is infinite, the exact cost system and the costs seen by programmers cannot be brought into precise agreement. However, as a practical matter all of the simple discrepancies can be repaired; and this is the best that can be done in general. This view of costs is only used by implementers, and many of the optimizations are done once, and then made available to programmers.

Programmers have a different, simpler, and more useful view of costs. Costs are associated with basic operations. The cost of a composition is the sum of the costs of each individual piece. Optimizations found by implementers are made available in this view by annotating each equation with its cost-reducing direction.

We have illustrated the approach by building a cost calculus for the Bird-Meertens theory of lists. The basic operations in this theory arise from a categorical construction which produces an equational transformation as a side-effect. The use of a standard topology allows these operations to be implemented on a wide range of architectures without any loss of efficiency due to communication. We have shown how several common operations, such as *inits* and *prefix*, have interesting implementations with costs lower than their definitions suggest, and have shown how this is made explicit to programmers without revealing implementation details. We have also shown how equations of the theory may be labelled with their cost-reducing direction, and how this is used in transformational development.

There are a number of deficiencies in the cost calculus presented here. Costs would be more accurate if they took into account load balancing, the actual sizes of list elements rather than the maximum at each depth, and if they could be computed for topologies that do not contain the standard topology. Including these improvements is technically feasible, but dauntingly complex; and makes preserving compositionality more difficult. The approach proposed here is a first working attempt to build a compositional cost calculus, but the right trade-off between accuracy and good composition properties remains something of an open question.

References

[1] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, to appear.

[2] F. Bellegarde. Rewriting systems on FP expressions to reduce the number of sequences yielded. *Science of Computer Programming*, 6:11–34, 1986.

[3] R.S. Bird. A calculus of functions for program derivation. Oxford University Programming Research Group Monograph PRG-64, 1987.

[4] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[5] R.S. Bird. Lectures on constructive functional programming. Oxford University Programming Research Group Monograph PRG-69, 1988.

[6] R.S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, February 1989.

[7] R.S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[8] G. Blelloch. Scans as primitive parallel operations. In *Proceedings of the International Conference on Parallel Processing*, pages 355–362, August 1987.

[9] G.E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[10] L. Bougé. The data-parallel programming model: A semantic perspective. Technical Report 92–45, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1992.

[11] W. Cai and D.B. Skillicorn. Calculating recurrences using the Bird-Meertens Formalism. *Parallel Processing Letters*, submitted February 1994.

[12] S. Chatterjee, G.E. Blelloch, and A.L. Fisher. Size and access interference for data-parallel programs. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 130–144, June 1991.

[13] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[14] M. Danelutto, R. di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 1992. Also appears as "The P^3L language: an introduction", Hewlett-Packard Report HPL-PSC-91-29, December 1991.

[15] M. Danelutto, R. di Meglio, S. Pelagatti, and M. Vanneschi. High level language constructs for massively parallel computing. Technical report, Hewlett Packard Pisa Science Center, HPL-PSC-90-19, 1990.

[16] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *PARLE93, Parallel Architectures and Languages Europe*, June 1993.

[17] V. Dornic, P. Jouvelot, and D.K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, pages 33–45, March 1992.

[18] S. Flynn Hummel and R. Kelly. A rationale for parallel programming with sets. *Journal of Programming Languages*, 1:187–207, 1993.

[19] D. le Métayer. Mechanical analysis of program complexity. *Proceedings of the SIGPLAN '85 Symposium*, pages 69–73, July 1985.

[20] C. Lengauer. Loop parallelization in the polytope model. In *CONCUR '93*, Springer Lecture Notes in Computer Science, 1993.

[21] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, September 1990.

[22] W.F. McColl. General purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation, pages 337–391. Cambridge University Press, Cambridge, 1993.

[23] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulation of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[24] C. Morgan. *Programming from Specifications*. Prentice-Hall International, 1990.

[25] A.G. Ranade. *Fluent Parallel Computation*. PhD thesis, Yale University, 1989.

[26] P. Roe. *Parallel Programming Using Functional Languages*. PhD thesis, Department of Computer Science, University of Glasgow, February 1991.

[27] P. Roe. Derivation of efficient data parallel programs. Technical report, Queensland University of Technology, December 1993.

[28] D. Sands. Complexity analysis for a higher-order language. Technical report, Department of Computing, Imperial College, Technical Report 88/14, London, December 1988.

[29] D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Imperial College, London, September 1990.

[30] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

[31] D.B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133–158, April 1991. Actually appeared in 1992.

[32] L. Snyder. Type architectures, shared memory and the corollary of modest potential. *Annual Review of Computer Science 1986*, 1:289–317, 1987.

[33] L.G. Valiant. Optimally universal parallel computers. *Phil. Trans. Royal Society Lond. Series A*, 326:373–376, 1988.

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

[35] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*. Elsevier Science Publishers and MIT Press, 1990.

if	*maps argument(s) of size*	*to*	*then*	*maps shapes*	*to*
f	m	q	$f*$	$[n, m] +\!\!+ s$	$[n, q] +\!\!+ s$
$\oplus$	m	m	$\oplus/$	$[n, m] +\!\!+ s$	$[m] +\!\!+ s$
$\oplus$	m	$2m$	$\oplus/$	$[n, m] +\!\!+ s$	$[nm] +\!\!+ s$
			inits	$[n] +\!\!+ s$	$[n, n] +\!\!+ s$
$\oplus$	m	m	$\oplus/\!\!/$	$[n, m] +\!\!+ s$	$[n, m] +\!\!+ s$
$\oplus$	m	$2m$	$\oplus/\!\!/$	$[n, m] +\!\!+ s$	$[n, nm] +\!\!+ s$

Figure 1: Effects of Operations on Shape Vectors (s is an arbitrary, possibly empty, shape vector)

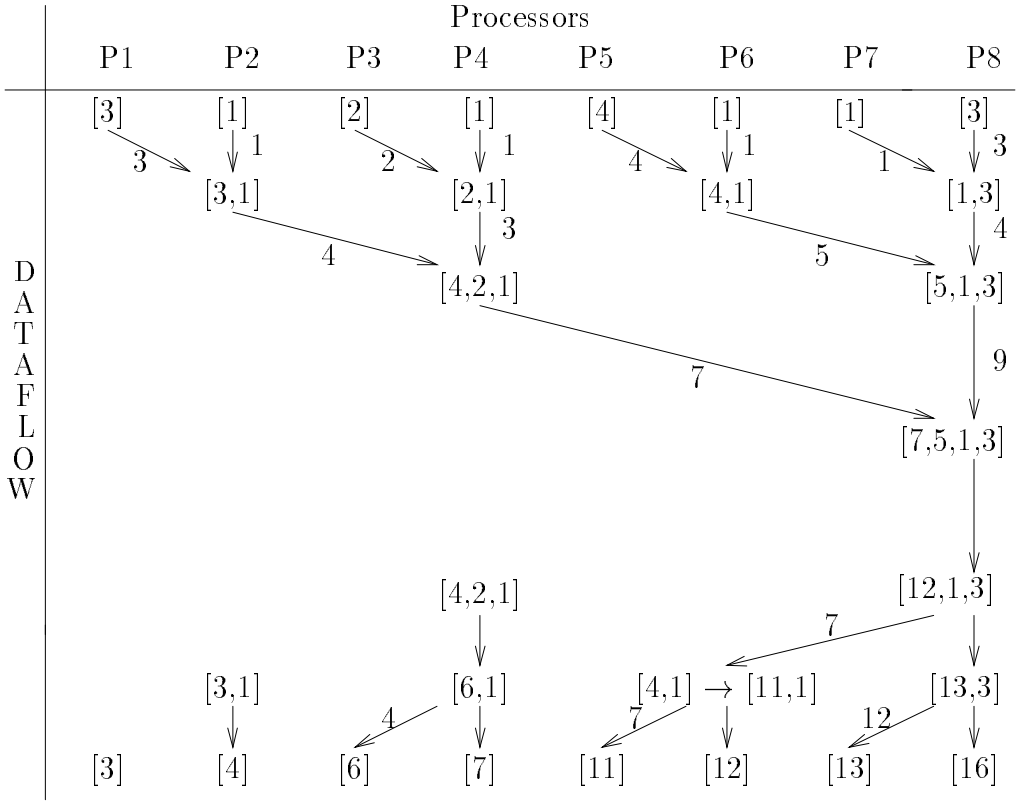
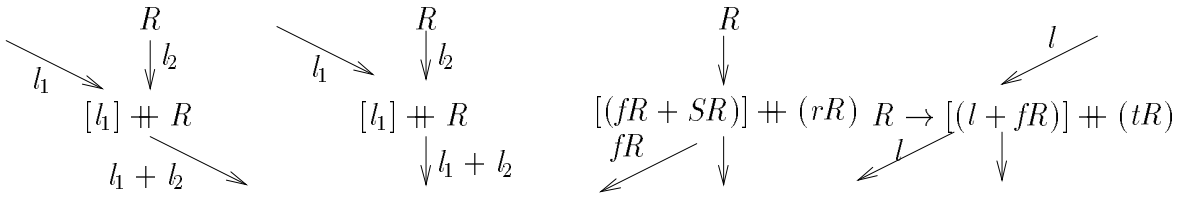


Figure 2: Data Flow for Prefix (using a list of length 8)



R is a list, l_1 and l_2 are list elements, and $\rightarrow$ means "change to"
$f[a1, a2, ..., an] = a1$, and $s[a1, a2, ..., an] = a2$
$r[a1, a2, ..., an] = [a3, ..., an]$, and $t[a1, a2, ..., an] = [a2, ..., an]$

Figure 3: Operations Performed at Each Processor

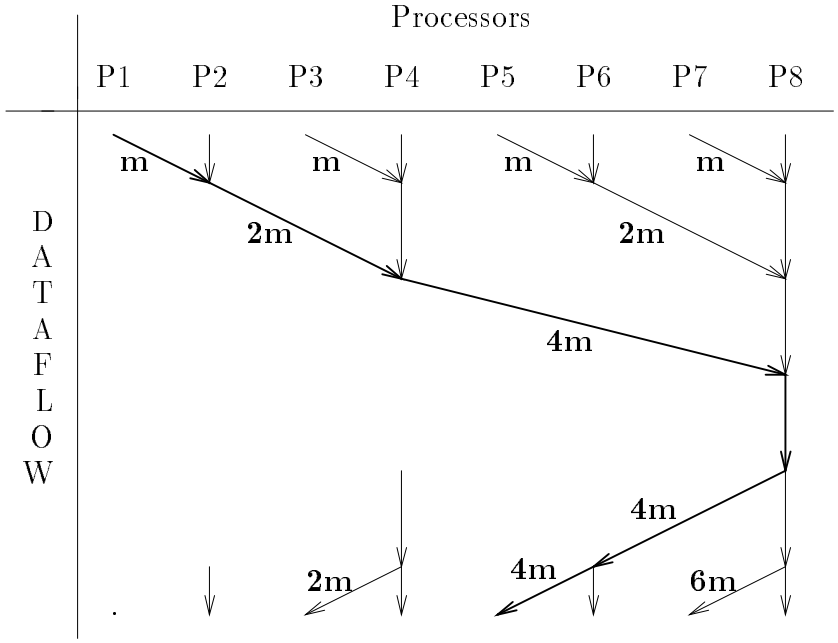


Figure 4: Data Flow for Concatenation Prefix (using a list of length 8 where initial elements are of size m, edge labels are the sizes of data flowing on arcs). Critical path shown bold.

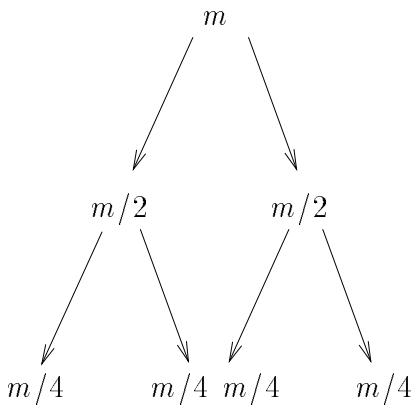


Figure 5: The *distribute* Operation

Operation	t_p
$f*$	n/p
$\oplus/$	$\log p + n/p - 1$
$\#/$	$(n-1)$
inits	$n^2/p + n + n^2/2p^2 + n/2p$
$\oplus/\!/$	$2(\log p + n/p + 1)$
$\#/\!/$	$n/2(\log p + n/p + 1) - 1$
$\otimes/_{b_0}\oplus$	$4n/p + 3\log p + 1$
$\otimes/\!/_{b_0}\oplus$	$10n/p + 6\log p + 6$

Figure 6: Summary of Operation Costs

$$
\begin{array}{rcl}
(f \cdot g)* & \rightleftharpoons & f* \cdot g* \\
f* \cdot \#/ & \rightleftharpoons & \#/ \cdot f** \\
+/* \cdot \#/ & \rightharpoonup & \#/ \cdot +/** \\
\oplus/ \cdot \#/ & \rightharpoonup & \oplus/ \cdot (\oplus/)* \\
\oplus/\!/ & \leftharpoondown & \oplus/* \cdot \mathrm{inits} \\
\mathrm{inits} & \leftharpoondown & \#/\!/ \cdot ([\cdot])* \\
\otimes/_{id_\otimes}\oplus & \leftharpoondown & \oplus/ \cdot \otimes/* \cdot \mathit{tails} \\
\otimes/\!/_{id_\otimes}\oplus & \leftharpoondown & (\otimes/_{id_\otimes}\oplus)* \cdot \mathit{inits}
\end{array}
$$

Figure 7: Cost-Reducing Annotations for some Equations