Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

Derivation of Scalable Message-Passing Algorithms Using Parallel Combinatorial List Generator Functions

Ali E. ABDALLAH and John HAWKINS

Research Institute for Computing,
London South Bank University,
103 Borough Road,
London SE1 0AA,
United Kingdom.

A.Abdallah@lsbu.ac.uk, John.Hawkins@reading.ac.uk

Abstract. We present the transformational derivations of several efficient, scalable, message-passing parallel algorithms from clear functional specifications. The starting algorithms rely on some commonly used combinatorial list generator functions such as $tails$, $inits$, $splits$ and cp (Cartesian product) for generating useful intermediate results. This paper provides generic parallel algorithms for efficiently implementing a small library of useful combinatorial list generator functions. It also provides a framework for relating key higher order functions such as map, $reduce$, and $scan$ with communicating processes with different configurations. The parallelisation of many interesting functional algorithms can then be systematically synthesized by taking an "off the shelf" parallel implementation of the list generator and composing it with appropriate parallel implementations of instances of higher order functions. Efficiency in the final message-passing algorithms is achieved by exploiting data parallelism, for generating the intermediate results in parallel; and functional parallelism, for processing intermediate results in stages such that the output of one stage is simultaneously input to the next one. This approach is then illustrated with a number of case studies which include: testing whether all the elements of a given list are distinct, the maximum segment sum problem, the minimum distance of two sets of points, and rank sort. In each case we progress from a quadratic time initial functional specification of the problem to a linear time parallel message-passing implementation which uses a linear number of communicating sequential processes. Bird-Meertens Formalism is used to concisely carry out the transformations.

1 Introduction

The design of efficient algorithms for many interesting programming problems often relies on the generation of combinatorial rearrangements of their input lists. A generic description of such an algorithm can then be seen as taking a list of values, generating useful rearrangements of the input list and processing those intermediate lists, possibly in parallel, in order to construct the final result. Examples of combinatorial list generator functions include: $inits$, $tails$ and $segs$ which generate, from a given list, the list of all its prefixes, suffixes, and segments (contiguous sublists) respectively. An algorithm which operates on a combinatorial list generator function, say gen, usually has the following form:

$$alg\ s = (phase_n\ \circ phase_{n-1} \circ \cdots \circ phase_1)\ \ (gen\ s)$$

where $phase_i$ is usually an instance of a higher order library function. The sequential implementation of such an algorithm normally exhibits, at least, a quadratic time behaviour. This is mainly because the size of the intermediate data generated by $(gen\ s)$ is, at least, quadratic. In [1, 2] it was shown that the above form can be correctly refined to a pipe of $(n+1)$ communicating processes, as shown in Fig 1. Here the first process $GEN(s)$ generates sequentially the result of $(gen\ s)$ and passes it to the process PH_1; and in turn to each of the processes PH_i (where $1 \leq i \leq n$), which implements the corresponding function $phase_i$.

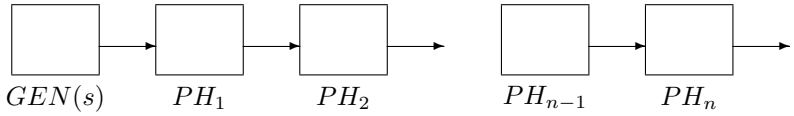


Figure 1: A pipelined implementation of $ALG(s)$

The above refinement obtained solely by exploiting pipelined parallelism is still quadratic, at best, on account of the size of the output of $GEN(s)$. Therefore, for this kind of algorithm pipelining alone may not lead to significant improvement in terms of speed. Hence, the only way to make a substantial improvement is to remove the bottleneck imposed by the sequential generation of $(gen\ s)$ and proceed by generating all the elements of $(gen\ s)$ in parallel, in a data-parallel fashion, each element on a separate channel. This ensures that all the sublists are generated in parallel in linear time. The structure of the pipelining solution stays unchanged but this time each phase in the pipe has internal parallelism; it takes inputs and produces outputs on a $vector$ of channels as opposed to a single channel.

This paper provides a number of scalable and efficient message-passing algorithms for implementing a small library of list generator functions. It exploits both data parallelism, for generating all the elements of the resulting list in parallel; and pipelining parallelism, for processing intermediate results in stages such that the output of one stage is simultaneously input to the next one. To ensure scalability, global communications are eliminated and replaced by efficient local communications for routing shared data to all the relevant processing elements.

We also associate each of several key higher order functions such as map, $reduce$, and $scan$ with two communicating processes which have different layouts: one, operating on a vector of values, which is suitable for data parallelism; and the other, operating on a stream of values, which is suitable for functional (or pipelined) parallelism. We show how the composition of higher order functions can be correctly implemented as (generalized) piping of the processes which implement each of these functions.

Since implementations of these library functions are readily available from a repertoire of parallel designs, the design of many message passing algorithms can then be systematically derived by simple program transformations and refinements. The parallelisation of many interesting functional algorithms is directly obtained from "off the shelf" parallel implementations of list generators and composition with appropriate parallel implementations of instances of higher order functions. This approach is then illustrated with a number of case studies which include: testing whether all the elements of a given list are distinct, the maximum segment sum problem, the minimum distance of two sets of points, and rank sort. In each case we progress from a quadratic time initial functional specification of the problem to a linear time parallel message-passing implementation which uses a linear number of communicating sequential processes. Bird-Meertens Formalism is used to concisely carry out the transformations.

The rest of this paper is organized as follows. Section 2 introduces some notation based on BMF and briefly explains some concepts for associating key higher order functions with communicating processes. Section 3 introduces several combinatorial list generator functions

and provides efficient scalable parallel message-passing algorithms for implementing them. Section 4 illustrates how the concepts and techniques of the previous two sections can be used to systematically derive efficient parallel solutions to a number of small case studies. Section 5 briefly describes related work and, finally, Section 6 concludes this paper.

2 Notation and Basic Concepts

Throughout this paper, we will use the functional notation and calculus developed by Bird and Meertens [3, 4, 5] for specifying algorithmics and reasoning about them and will use a CSP style environment (as developed by Hoare [8]) for specifying processes and reasoning about them. We give a brief summary of the notation and conventions used in this paper. The reader is advised to consult the above references for further details.

2.1 Lists, Streams and Vectors

Lists are finite sequences of values of the same type. The list concatenation operator is denoted by $+\!\!+$ and the list construction operator is denoted by $:$. The elements of a list are displayed between square brackets and separated by commas.

Conventionally we have modelled lists in our parallel implementation as a *stream*, a serial sequence of messages on a channel. So for a list $[x_1, x_2, ..., x_n]$, we first send x_1 along our channel, then x_2 and so on up to x_n which is then followed by the special message eot to denote the end of the transmission. However, as has been explained, this can, in certain cases, introduce unacceptable bottlenecks into a network. Thus we have the alternative to streams which we call *vectors*. Here a separate channel is used for each item in the list and as such the whole list can then be communicated in parallel. This not only alleviates this bottleneck for many algorithms but also introduces scope for some data parallelism in our networks.

2.2 The Map Operator

The operator $*$ (pronounced "map") takes a function on the left, a list on the right, and applies the function to each element of the list. Informally, we have:

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

We can associate with this function two different processes, the first, MAP, corresponds to functional or task parallelism. It takes a stream of inputs on one channel, say in, and produces a stream of output on a channel, say out. This can be pictured as seen in Figure 2.

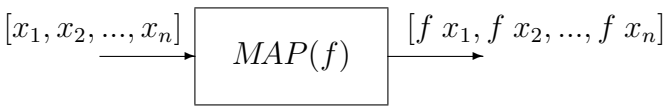


Figure 2: The Process MAP.

The second implementation, $VMAP$, corresponds to data or vector parallelism. It takes the list of values as inputs on a list of channels (one channel per value) and produces the resulting list on a list of channels one value per channel. This can be pictured as seen in Figure 3.

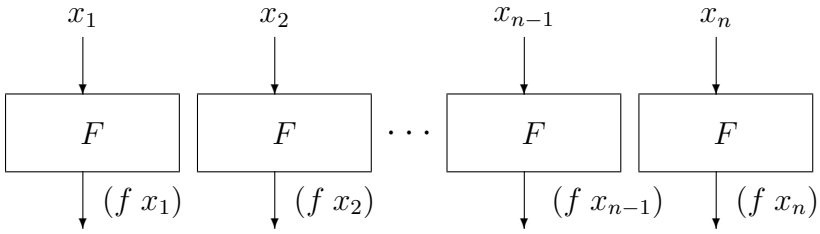


Figure 3: The Process $VMAP$.

2.3 Reduction Operators

The operator / (pronounced "reduce") takes an associative binary operator on the left, a list of values on the right and returns the "summation" of all the elements of the list. This can be informally described as follows

$$(\oplus)/\,[a_1, a_2, \cdots, a_n] \;=\; a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

The left reduction operator $(\oplus\!\!\not\rightarrow e)$ (also known as $foldl$) corresponds to a specific interpretation of reduction in which the computation of a list starts with e, as an initial value, and the result is gradually accumulated by successively applying the operator $\oplus$ while traversing the list from left to right. Informally, we have:

$$(\oplus\!\!\not\rightarrow e)\,[a_1, a_2, \cdots, a_n] \;=\; (\cdots((e \oplus a_1) \oplus a_2) \oplus \cdots) \oplus a_n$$

The right reduction operator $(\oplus\!\!\not\leftarrow e)$ (also known as $foldr$) is similar to $(\oplus\!\!\not\rightarrow e)$ except that the computation proceeds by traversing the list in the opposite direction, that is, from right to left.

$$(\oplus\!\!\not\leftarrow e)\,[a_1, a_2, \cdots, a_n] \;=\; a_1 \oplus (a_2 \oplus\ (\cdots \oplus (a_n \oplus e)\cdots)\ )$$

Note that the operator used with directed reductions (both left and right) may not be associative.

As regards implementation of the fold operators, again we have two choices. The first two processes, $FOLDL$ and $FOLDR$, again, correspond to functional parallelism. $FOLDL$ is depicted in Figure 4, and $FOLDR$ would have a similar appearance. Here the process takes in a stream and returns either a stream or a single result, depending on the nature of the function used.

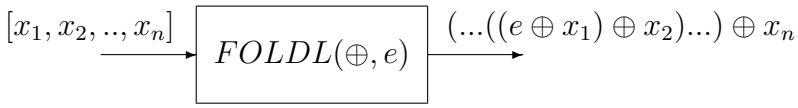


Figure 4: The Process $FOLDL$.

As before, the introduction of vectors requires that we also have a second view of the operators when the issue of implementation arises. Here we need to envisage a process that takes in a vector and produces the same result as the previous processes. Thus we have the process VFOLDL (as seen in Figure 5). Again, VFOLDR has a similar layout.

We may also require fold operators that do not take a base value, often termed as the functions $foldl1$ and $foldr1$, named as such due to only being defined on lists of length 1 or greater. This can be achieved in the implementation simply by removing the input of e, and,

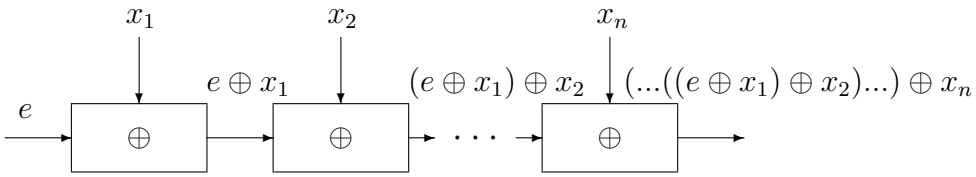


Figure 5: The process $VFOLDL$

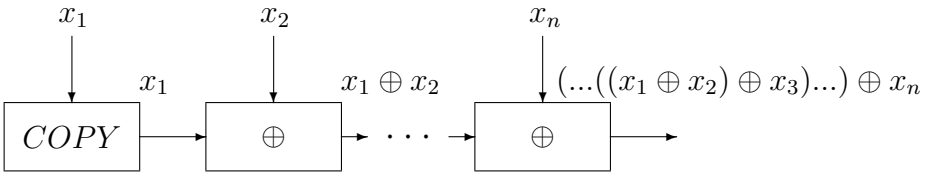


Figure 6: The process $VFOLDL1$

to compensate, replacing the first instance of the folding function with $COPY$, which simply mimics its input as its output. This can be seen in Figure 6 for a refinement of $foldl1$ applied to a vector. A similar implementation can be achieved for $foldr1$ via the same techniques.

2.4 Sections and Function Composition

Binary operators can be *sectioned*. This means that $(a\oplus)$ and $(\oplus b)$ both denote functions. Thus, if $\oplus$ has type $\oplus : A \to B \to C$, then we have

$$\begin{array}{lcl} (a\oplus) & : & B \to C \\ (\oplus b) & : & A \to C \end{array}$$

for all $a \in A$ and $b \in B$. The definitions of these sections are:

$$\begin{array}{lcl} (a\oplus)\ b & = & a \oplus b \\ (\oplus b)\ a & = & a \oplus b \end{array}$$

For example, $f*$ denotes a function which takes a list of values and maps f to each element of the list; but $(*xs)$ denotes a function which takes a function as input and applies it to each element of the list xs.

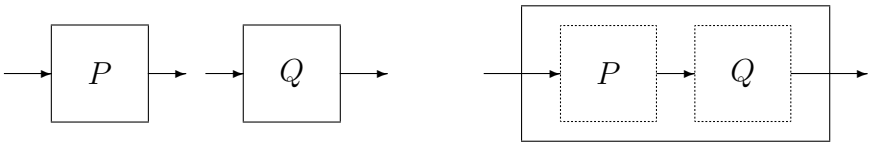
Function composition is denoted by $\circ$. This operator has lower precedence than all other operators. Thus, $f \circ g*$ denotes $f \circ (g*)$ and not $(f \circ g)*$.

2.5 Refinement to Processes

Function composition corresponds to functional parallelism and can be realized in a concurrency framework (for example CSP) by process piping ($\gg$). Careful checking must be done to ensure the correctness of this realization that the output of one process must match (have the same type as) the input of the next process in the pipe. If the common type is a stream, say [A], we will denote piping by $\gg$; but if it is a vector of length p, we will denote piping by the operator ($\gg_p$).

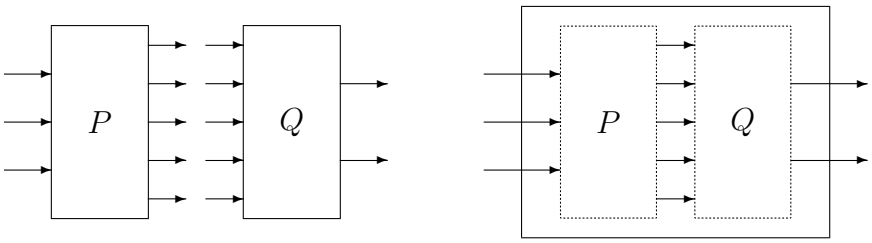
2.6 Algebraic Laws

One important asset of BMF is its richness in algebraic laws which allow the transformation of a program from one form to another while preserving its meaning. Here is a short list



The Processes P and Q. $P \gg Q$

Figure 7: Stream Piping



The Processes P and Q. $P \gg_5 Q$

Figure 8: Piping Generalized.

of frequently used algebraic rules which will be used in later examples. Historically, the "promotion rules" are intended to express the idea that an operation on a compound structure can be "promoted" into its components.

$$
\begin{array}{lcll}
(f \circ g)* & = & (f*) \circ (g*) & \text{map distributivity} \\
f* \circ +\!\!+/ & = & +\!\!+/ \circ (f*)* & \text{map promotion} \\
\oplus/ \circ +\!\!+/ & = & \oplus/ \circ (\oplus/)* & \text{reduce promotion}
\end{array}
$$

3 Combinatorial List Generator Functions

3.1 Segments

A list s is a *segment* of t if there exist u and v such that $t = u +\!\!+ s +\!\!+ v$. If $u = []$, then s said to be an *initial segment* or a *prefix* of t. On the other hand, if $v = []$, then s is called a *final segment* or a *suffix* of t.

3.2 Inits and Tails

The function $inits$ returns the list of initial segments of a list, in increasing order of length. The function $tails$ returns the list of final segments of a list, in decreasing order of length. Thus, informally, we have

$$
\begin{array}{lcl}
inits\ [a_1, a_2, \ldots, a_n] & = & [[], [a_1], [a_1, a_2], \ldots, [a_1, a_2, \ldots, a_n]] \\
tails\ [a_1, a_2, \ldots, a_n] & = & [[a_1, a_2, \ldots, a_n], [a_2, a_3, \ldots, a_n], \ldots, []]
\end{array}
$$

The functions $inits^+$ and $tails^+$ are similar, except that the empty list does not appear in the result.

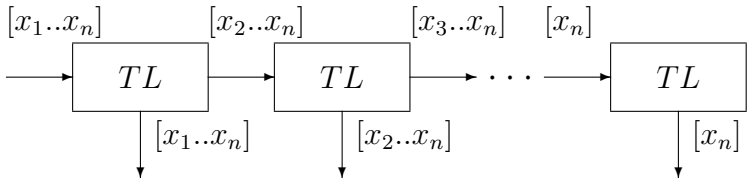


Figure 9: The process $TAILS$

We can define the functions $tails$ and $inits$ by explicit recursion equations (see [5]). For example, the suffixes of a list.

$$\begin{array}{lcl} tails\ [] & = & [[]] \\ tails\ (x:xs) & = & (x:xs) : (tails\ xs) \end{array}$$

For the examples given in this paper we use $tails^+$, rather than $tails$, so we will define a process corresponding to that function only for now. Here each item of the resulting list is produced on a separate channel- the output is modelled as a vector. The resulting network, $TAILS$, is depicted in Figure 9.

The function $inits$, which gives us the prefixes of a list can similarly be defined as follows (see [5]):

$$\begin{array}{lcl} inits\ [] & = & [[]] \\ inits\ (x:xs) & = & [[]] +\!\!+ ((x:) * (inits\ xs)) \end{array}$$

This could be implemented with a similar network to that in Figure 9, except that the flow of data would come from the right instead of the left. This underlines the basic symmetry that exists between $inits$ and $tails$.

3.3 Segs

The functions $segs$ returns a list of all segments of a list, and $segs^+$ returns a list of all non-empty segments. A convenient definition is

$$\begin{array}{lcl} segs & = & (+\!\!+)/\ \circ\ (inits *)\ \circ\ tails \\ segs^+ & = & (+\!\!+)/\ \circ\ (inits^+ *)\ \circ\ tails^+ \end{array}$$

For example,

$$\begin{array}{lcl} segs\ [1,2,3] & = & [[],[1],[1,2],[1,2,3],[],[2],[2,3],[],[3],[]] \\ segs^+\ [1,2,3] & = & [[1],[1,2],[1,2,3],[2],[2,3],[3]] \end{array}$$

Notice that the empty list [] appears four times in $segs\ [1,2,3]$ (and not at all in $segs^+[1,2,3]$). The order in which the elements of $segs\ x$ appear is not important for our purposes and we shall make no use of it. In effect, we shall reduce over $segs$ with commutative operators only.

3.4 Splits

The function $splits$ returns the list of all possible ways of splitting a list into two parts such that the second part is non-empty. Informally, we have:

$$\begin{array}{lcl} splits\ [a_1,a_2,\cdots,a_n] & = & [\ ([],\ [a_1,a_2,\cdots,a_n]),\ ([a_1],\ [a_2,\cdots,a_n]), \\ & & \quad ([a_1,a_2],\ [a_3,\cdots,a_n]),\ \cdots,\ ([a_1,a_2,\cdots,a_{n-1}],\ [a_n])\ ] \end{array}$$

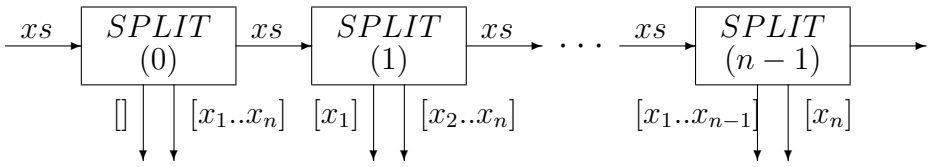


Figure 10: The process $SPLITS$

We can construct functional definition of $splits$ using the functions $take$ and $drop$ (see [5]):

$$splits\ s \quad = \quad [(take\ i\ s, drop\ i\ s) \mid i \leftarrow [0..\#s - 1]\ ]$$

With this function we can associate the network $SPLITS$ which produces each item of the resulting list on a separate pair of channels. This is depicted in Figure 10.

3.5 Cartesian Product

The function cp returns a Cartesian product of its two input lists xs and ys, i.e. a list where every element of xs is paired with every element of ys. This can be defined quite simply as follows:

$$cp\ xs\ ys \quad = \quad [(x, y) \mid x \leftarrow xs; y \leftarrow ys]$$

Production of this list, as in the previous cases will require $O(n^2)$ steps. However, if we consider the result as a list of lists, each an element from xs paired with every element of ys, the result can then be produced in parallel in linear time. Thus we have a slight redefinition. We shall call this new function dcp for distributed Cartesian product.

$$dcp\ xs\ ys \quad = \quad [(pair\ x) * ys \mid x \leftarrow xs]$$

So, for example:

$$\begin{array}{rcl} dcp\ [1,2,3]\ ['a','b','c'] & = & [[(1,'a'),(1,'b'),(1,'c')], \\ & & [(2,'a'),(2,'b'),(2,'c')], \\ & & [(3,'a'),(3,'b'),(3,'c')]] \end{array}$$

We can clearly define cp in terms of dcp to illustrate the relation between the two functions.

$$cp\ xs \quad = \quad (+\!\!+)/ \ \circ\ dcp\ xs$$

This function can now be associated with the process CP, pictured in figure 11.

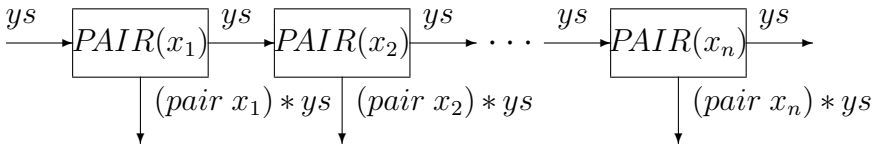


Figure 11: The process CP

3.6 Accumulations

The operator $\oplus\!\!\!\!\!\!\rightarrow e$ (or $scanl$) takes a binary operator, say $\oplus$, on the left; a list of values on the right; and applies the function $(\oplus\!\!\!\!\!\!\rightarrow e)$ to all the initial segments of the list. This function is often being refered to as "prefix sum" when the operator $\oplus$ is associative. We have

$$(\oplus\!\!\!\!\!\!\rightarrow e)\ s \quad = \quad (\oplus\!\!\!\!\!\rightarrow e) * (inits\ s)$$

For example:

$$(+\!\!\!\!\!\!\rightarrow 0)\ [1,2,3,4,5] \quad = \quad [0,1,3,6,10,15]$$

In some cases we may not want to include the base value in the resulting list. For this purpose, in a similar manner to the $fold$ functions, we also introduce a function $scanl1$. This can be defined as follows:

$$scanl1\ (\oplus)\ e\ s \quad = \quad (\oplus\!\!\!\!\!\rightarrow e) * (inits^{+}\ s)$$

For example:

$$scanl1\ (+)\ 0\ [1,2,3,4,5] \quad = \quad [1,3,6,10,15]$$

For each of these functions we are given two choices for implementation, in each case either using streams or vectors. The function $scanl1$ can be implemented with stream parallelism using the process $SCANL1$, as shown in Figure 12. This process will both input and output a stream. Alternatively, we have a vector implementation, and this is demonstrated by the process $VSCANL1$, shown in Figure 13. This process inputs a vector and outputs a vector.

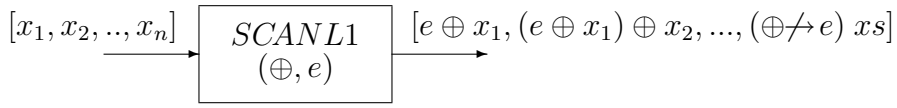


Figure 12: The Process $SCANL1(\oplus, e)$.

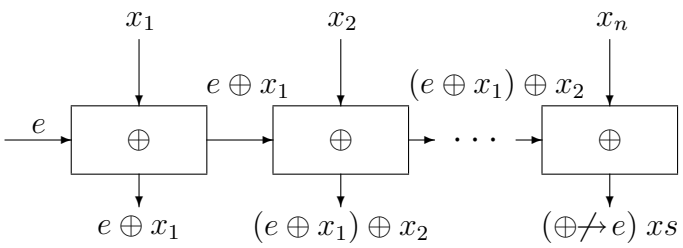


Figure 13: The process $VSCANL1(\oplus)$

4 Case Studies

4.1 The Maximum Segment Sum Problem

There is a famous problem, called the *maximum segment sum* (mss) problem, which is to compute the maximum of the sums of all segments of a given sequence of numbers, positive, negative or zero. For example, $mss\ [2,-3,1,2,-2,3,-1] = 4$ which corresponds to the segment $[1,2,-2,3]$. For details see Bird's paper ... In symbols

$$mss = \max/\ \circ sum* \circ segs^+$$

Direct evaluation of the right-hand side of this equation requires $O(n^3)$ steps on a list of length n. There are $O(n^2)$ segments and each can be summed in $O(n)$ steps, giving $O(n^3)$ steps in all.

$$\begin{array}{rcl}
mss & = & \left\{ \text{ definition } \right\} \\
 & & \max/\ \circ sum* \circ segs^+ \\
 & = & \left\{ \text{ definition of } segs^+ \right\} \\
 & & \max/\ \circ sum* \circ +\!\!+/\ \circ inits^+* \circ tails^+ \\
 & = & \left\{ \text{ map promotion } \right\} \\
 & & \max/\ \circ +\!\!+/\ \circ sum** \circ inits^+* \circ tails^+ \\
 & = & \left\{ \text{ map distributivity } \right\} \\
 & & \max/\ \circ +\!\!+/\ \circ (sum* \circ inits^+)* \circ tails^+ \\
 & = & \left\{ \text{ definition of accumulation } \right\} \\
 & & \max/\ \circ +\!\!+/\ \circ (+\!\!\!\!\!\!/\!\!\rightarrow)* \circ tails^+ \\
 & = & \left\{ \text{ reduce promotion } \right\} \\
 & & \max/\ \circ \max/* \circ (+\!\!\!\!\!\!/\!\!\rightarrow)* \circ tails^+
\end{array}$$

Implementation is now an almost trivial matter of combining 'off the shelf' components that correspond with each stage of our algorithm. This gives us the following network:

$$TAILS \gg_n VMAP(SCANL(+)) \gg_n VMAP(FOLD(max)) \gg_n VFOLDR1(max)$$

where n is the length of the input list. The resulting network can be seen in Figure 14. This algorithm will now run in linear time.

4.2 Minimum Distance

Given two lists of points in three dimensional space, the minimum distance function, md compares every point from the first list with every point from the second list and returns the distance between the closest of these.

Here we need to make use of the Cartesian product function, cp. We simply need to map some function $dist$ to this list, and then find the minimum of this result.

$$\begin{array}{lcl}
md & = & (min/) \circ (dist*) \circ cp \\
dist\ (x1,y1,z1)\ (x2,y2,z2) & = & \sqrt{(x2-x1)^2+(y2-y1)^2+(z2-z1)^2}
\end{array}$$

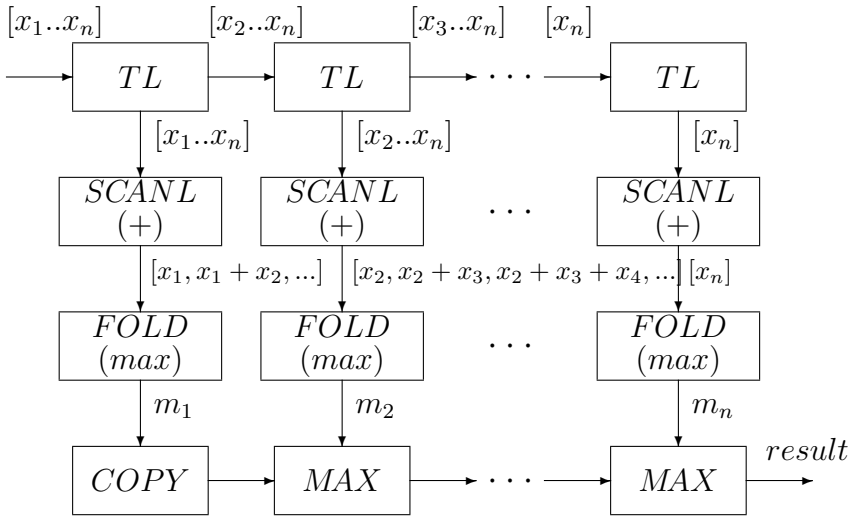


Figure 14: A network to solve the maximum segment sum problem for the list xs

However, given the nature of dcp, producing a list of lists, we need a little remodeling to achieve the actual definition.

$$
\begin{aligned}
md\ xs &= \quad \{\ \text{definition}\ \} \\
&\quad \min /\ \circ dist* \circ (cp\ xs) \\
&= \quad \{\ \text{definition of } cp\ \} \\
&\quad \min /\ \circ dist* \circ +\!\!+/\ \circ (dcp\ xs) \\
&= \quad \{\ \text{map promotion}\ \} \\
&\quad \min /\ \circ +\!\!+/\ \circ (dist*) * \circ (dcp\ xs) \\
&= \quad \{\ \text{reduce promotion}\ \} \\
&\quad \min /\ \circ \min /\ * \circ (dist*) * \circ (dcp\ xs)
\end{aligned}
$$

The implementation can now be constructed as before, giving us the network:

$$CP(xs) \gg_n VMAP(MAP(dist)) \gg_n VMAP(FOLD(min)) \gg_n VFOLDL1(min)$$

Again n is the length of the input list. This is depicted in Figure 15.

4.3 Lists with Distinct Elements

Consider the problem of testing whether all the elements of a list are distinct. That is, no element of the list occurs more than once in the list.

Essentially we need to compare every element in the list with every other, see if they differ, and then and all these results together. This can be achieved with the following function:

$$distinct \ = \ (\wedge/) \circ noteq* \circ tails^+$$

Given a function $noteq$ which takes a list and dictates if the first item is different to all the others.

$$
\begin{aligned}
noteq &= (\wedge)/ \circ diff \\
diff\ (x : xs) &= (\neq x) * xs
\end{aligned}
$$

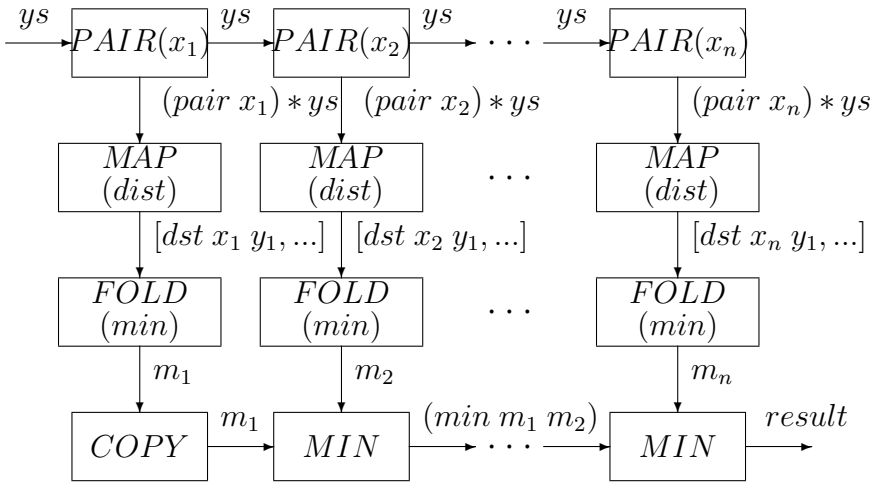


Figure 15: A network to solve the minimum distance problem for the lists xs and ys

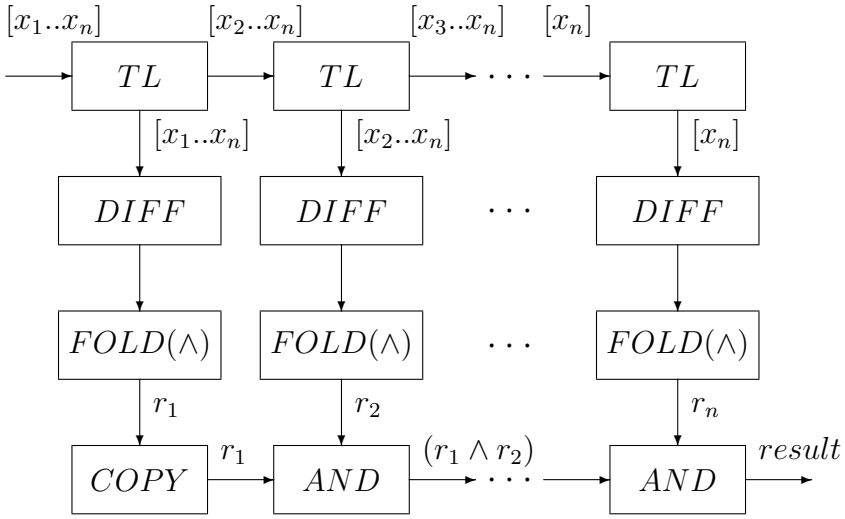


Figure 16: A network to solve the distinct elements problem for the list xs

This allows us to give a slight redefinition of $distinct$:

$$distinct \;=\; (\wedge/) \circ (\wedge/)* \circ diff* \circ tails^{+}$$

The above definition can then be transformed quite straightforwardly to the following network:

$$TAILS \gg_n VMAP(DIFF) \gg_n VMAP(FOLD(\wedge)) \gg_n VFOLDL1(\wedge)$$

where n is the length of the input list. The results can be seen in Figure 16.

4.4 Parallel Enumerate Sort

For each element in the input list $[a_1, a_2, \cdots, a_n]$, the *rank sort* algorithm aims at computing its final position (rank) in the sorted list. This is simply achieved by counting the number of elements in the list having smaller value ([9]). If j elements have smaller value than a_i then

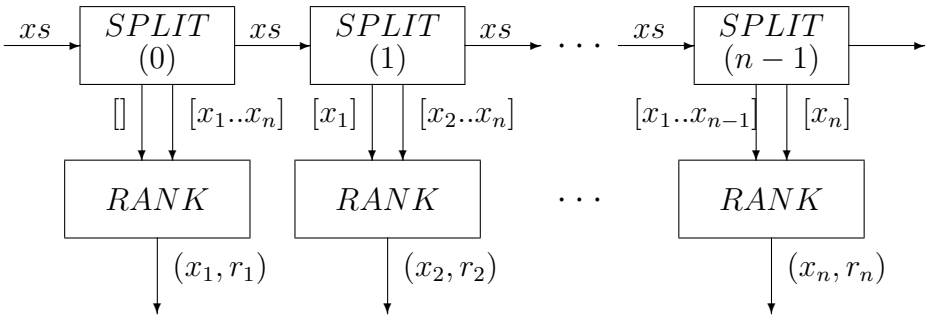


Figure 17: A network performing rank sort on the list *xs*

a_i is the $(j+1)^{th}$ element of the sorted list. If two or more elements have the same value, the algorithm must be slightly amended in order to produce a unique rank for each element in the unsorted list. This is achieved by counting the number of elements having smaller value or the same value and smaller index in the unsorted list. Formally, the rank of the i^{th} element of the list $[a_1, a_2, \cdots, a_n]$ is captured as:

$$rank\ i\ [a_1, a_2, \cdots, a_n] = \#\{j \in \{1..n\} \mid a_j < a_i \vee (a_j = a_i \wedge j < i)\}$$

The functional specification of the *rank sorting* algorithm is:

$$\begin{array}{lcl} rsort :: [\alpha] \rightarrow [(\alpha, num)];\ rank :: ([\alpha], [\alpha]) \rightarrow (\alpha, num) \\ rsort & = & (rank*) \circ splits \\ rank\ (s, x : t) & = & (x, \#filter\ (< x)\ s + \#filter\ (\leq x)\ t) \end{array}$$

The resulting implementation can be found in Figure 17.

5 Conclusion

In this paper we have presented a number of frequently used combinatorial list generator functions and showed how they can be efficiently implemented in parallel as networks of interacting processes. These list generator functions are the building blocks for many interesting algorithms and their proposed parallel implementations can be the basis for systematically parallelising such algorithms.

This paper attempts to combine both data parallelism and functional parallelism in one framework which is founded on concurrency. It also develops some real world algorithms which make use of this framework. Both the functions and algorithms which use them had at least quadratic (sequential) execution time and in every case we have successfully developed linear time parallel implementations.

The focus has been on these combinatorial functions, and how a mixture of data and functional parallelism can be used to remove the previous bottleneck of communicating their results (lists of lists). In the process we have introduced models for data parallelism into a transformational framework which previously concentrated solely on functional parallelism (see [1]).

We have presented an essentially skeletonic approach to implementation. Given a specification composed of several more commonly used functions, we can simply take each function in turn, find a pre-defined parallel implementation known to be efficient, and then link these all together to create our network. This approach has several advantages. The first and perhaps most obvious is speed of development. In addition, as already mentioned, our predefined 'off the shelf' components are already known to be efficient, and, whereas we cannot

always guarantee that the resulting network will be optimal, we will in almost all cases see a substantial increase in execution time. Finally, this approach removes some of the burden of understanding the inherent parallelism from the programmer, which can allow them to concentrate more on what is actually required from an algorithm, rather than how is best to implement it. This highly systematic approach may then even lead to an entirely automatic tool for transformation from specification to parallel implementation. This is an area we are already investigating.

This work could be extended to include a larger class of combinatorial list generator functions, and a further study of the effects of higher order functions on vectors. We may then identify situations quite different to those encountered in this work where this combined parallelism could be used to great effect.

References

[1] A.E. Abdallah, Derivation of Parallel Algorithms from Functional Specifications to CSP Processes, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS 947, (Springer Verlag, 1995) 67-96

[2] A. E. Abdallah, Synthesis of Massively Pipelined Algorithms for List Manipulation, in L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert (eds), Proceedings of the *European Conference on Parallel Processing, EuroPar'96*, LNCS 1024, (Springer Verlag, 1996), pp 911-920.

[3] R. S. Bird, An Introduction to The Theory of Lists, in: M. Broy, ed., *Logic of Programming and Calculi of Discreet Design*, (Springer, Berlin, 1987) 3-42.

[4] R. S. Bird, Functional Algorithm Design, in: Bernhard Möller, ed., *Mathmeatics of Program Construction*, LNCS 947, (Springer Verlag, 1995) 2-17

[5] R. S. Bird and P. Wadler, *Introduction to Functional Programming*, (Prentice-Hall, 1988).

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

[7] J. Darlington, A. Field and P.G. Harrison, *Parallel Programming Using Skeleton Functions*, in: A. Bode, M. Reeve and G. Wolf, eds., Parallel Architectures and Languages Europe (PARLE'93), LNCS 694.

[8] C. A. R. Hoare, *Communicating Sequential Processes*. (Prentice-Hall, 1985).

[9] Donald E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching* Addison-Wesley 1973