Formal Behavioural Synthesis of Handel-C Parallel Hardware Implementations from Functional Specifications

Ali E. Abdallah and John Hawkins
Centre for Applied Formal Methods South Bank University,
103 Borough Road,
London, U.K.
Email: { A.Abdallah, John.Hawkins}@sbu.ac.uk

Abstract

Enormous improvements in efficiency can be achieved through exploiting parallelism and realizing implementation in hardware. On the other hand, conventional methods for achieving these improvements are traditionally costly, complex and error prone. Two significant advances in the past decade have radically changed these perceptions. Firstly, the FPGA, which gives us the ability to reconfigure hardware through software, dramatically reducing the costs of developing hardware implementations. Secondly, the language Handel-C with primitive explicit parallelism which can compile programs down to an FPGA. In this paper, we build on these recent technological advances and present a systematic approach of behavioural synthesis. Starting with an intuitive high level functional specification of a problem, given without annotation of parallelism, the approach aims at deriving an efficient parallel implementation in Handel-C, which is subsequently compiled into a circuit implemented on reconfigurable hardware. Algebraic laws are systematically used for exposing implicit parallelism and transforming the specification into a collection of interacting components. Formal methods based on data refinement and a small library of higher order functions are then used to derive behavioural description in Handel-C of each component. A small case study illustrates the use of this approach.

1 Introduction

In certain application areas, software executed on a single sequential processor no longer meets our ever increasing efficiency requirements. Parallelism and implementation in hardware provide us with two alternatives that can often deliver very dramatic improvements in efficiency, but at a cost - complexity. Parallel implementations of algorithms are often far from intuitive, and hardware design requires extremely low level technicalities to be resolved that can detract seriously from the clarity of the design. Additionally, hardware implementation has traditionally been a very expensive task because of the high costs of the physical processes involved. The FPGA has been a revolution in this field, as it has provided for the developer a hardware element that can be reconfigured by software. For some time FPGA implementations were configured using languages such as VHDL, until a further breakthough came in the form of Handel-C [11]. This is an ANSI C based language with scope for parallelism and CSP [12] style communication that can compile down to an FPGA. In short we now have a quick, low cost path from a high level language to an efficient, parallel implementation in hardware.

This methodology is not complete, however. Parallelism in Handel-C is explicit, and so we are relying entirely on the programmer to specify the parts of the algorithm that can operate in parallel. As such this is likely to be assigned in a fairly ad-hoc method, leading to an implementation that is potentially neither accurate, nor error free, nor optimal. Much talk is made of behavioural synthesis, the 'art' of deriving a parallel algorithm from one where parallelism is not strictly specified. For this it is vital to choose the right starting point. Imperative languages are perhaps not ideal for this task. The direct manipulation of state makes it both difficult to reason about programs and to perform safe substitutions algebraic modifications. Functional languages [5] such as Haskell [6], however, do not allow assignment and direct manipulation of state, and as such gain the property of referential transparency. Any sub-expression of an algorithm can be substituted

for any other equivalent sub-expression without affecting the meaning of the algorithm. This allows a rich set of algebraic laws such as described in BMF [4], along with a wealth of related work on parallel functional programming [2, 3, 8, 9, 10, 13, 14, 15]

In this work we present a methodology for taking intuitive specifications of algorithms in the functional style, and systematically transforming them into scalable, efficient, parallel implementations in Handel-C. This technique exposes implicit parallelism in such algorithms, both of the functional and data parallel varieties.

2 Data Refinement

Truly scalable efficiency in highly parallel systems can only be achieved with some kind of message passing archictecture. Systems that rely purely on shared memory will inevitably reach bottlenecks as the number of processing elements increases. In a message passing environment, it is important to consider different methods for the communication of values and how this may effect the performance of the network. We shall look at some of the alternatives below.

Functional languages generally employ lists as a built-in type. The list is a good abstraction for many linear datatypes, including the array, as often encountered in an imperative setting. Semantically, the list is simply a collection of items with a mapping from an index to a single item, where repetitions are possible. We denote lists with square brackets: [and]. The empty list is written [], and elements in the list are separated by commas, such that a list containing 3 values x_1 through to x_3 might be written $[x_1, x_2, x_3]$. Lists are constructed with the operator :, so the expression $(x : xs)$ represents a single item x at the head of a list xs. We can concatenate two lists together with the operator $+\!\!+$.

2.1 Streams

A *stream* is a sequence of messages on a single channel. As we will be generally concerned with the communication of finite sequences, we will require some means of signalling the end of transmission. Several strategies might be employed here, including the obvious use of rogue values, and additional bits with each message. A third strategy would introduce a second channel on which a single bit could be transmitted to denote the end of transmission. This message would only be sent once per stream, after all values have been communicated on the primary channel. Although this introduces the potential overhead of using twice as many channels as the previous two, we believe it gives a more generic and elegant model than the alternatives, and so this is the model we shall adopt for the purposes of this work. Note also the stipulation that the value communicated on the EOT channel is always true (1), as this makes several implementations slightly more efficient. We shall declare streams in this work using the following syntax. This is in fact a simple macro which creates a structure containing two channels.

```
Stream (type,name);
```

This will create a Handel-C structure of the following form:

```
struct {
  chan type value;
  chan Bool eot;
} name;
```

Let us also introduce a second definition here, a simpler construct where we need a channel to communicate just a single value. In this case we won't need the EOT channel. We shall call this an item. We can declare a shot similarly as follows:

```
Item (type,name);
```

This will create a Handel-C structure of the following form, Note the lack of an EOT channel here.

```
struct {
  chan type value;
} name;
```

Note it is still convenient to have this channel encased in a structure as it means we can manipulate it in the same way as we would a stream. We shall see later where this is convenient.

2.2 Vectors

A *vector* is a means of communicating a list on more than one channel. We will assume there are as many channels in the vector as there are items in the list, such that each item is communicated on its own channel. This method maximises the scope for data parallelism, but also makes an assumption that sufficient processing elements are available to deal with each item of the list independantly.

We shall declare vectors in this work in one of two ways. Firstly, a vector of items. This is a macro which creates an array of channels of size n. This would be a suitable means to communicate a list where the elements are simple values of constant size (a list of integers for example).

```
VectorOfItems (n,type,name);
```

This creates an array of item structures, and is effectively the same as:

```
Item (type,name) [n];
```

Secondly, a vector of streams. This would be used for communicating lists of lists - lists where the elements are lists themselves. This macro creates an array of streams, so in effect creates $2n$ channels.

```
VectorOfStreams (n,type,name);
```

Again, this is equivalent to:

```
Stream (type,name) [n];
```

This opens up possibilities for further communications structures. For example, a vector of vectors would allow us to communicate quadratic sized structures (for example, lists of lists) in constant time.

A stream of vectors might be also be a useful mechanism for communicating lists of lists - at each step we communicate a whole set of values in parallel, until finally at the end we signal the end of transmission for the whole structure on a single channel. Note the distinction between this and a vector of streams - with a vector of streams we require n channels to signal EOT, and the individual streams might all cease transmission at different times.

For the purposes of this paper, however, we shall be concerned only with items, streams, vectors of items and vectors of streams.

3 Processes

We will find convenient a number of simple processes that will deal with communication. This will also help to introduce some of the concepts and syntax used later in this work.

3.1 Produce

Produce communicates a list of items on a single channel as a stream. In CSP, we might define this as follows:

$$\begin{array}{lcl} PRODUCE_{out}([]) & = & out.eot\,!\,True \rightarrow SKIP \\ PRODUCE_{out}(x:xs) & = & out.value\,!\,x \rightarrow \\ & & PRODUCE_{out}(xs) \end{array}$$

Briefly, we pass to $PRODUCE$ the name of the channel (or in fact the two channels) on which we are to produce the stream, and the list we actually wish to communicate. If the list is empty, we can just output $True$ on the eot channel then terminate. If the list isn't, we first output the first value on the appropriate output channel, and then repeat the process for the remainder of the list.

Considering the Handel-C equivalent, we should pass the items as an array, `xs`, for which we must also specify the length, `n`. Additionally we must inform the process of the output stream with which we are to communicate, `streamout`. The implementation is then a simple `for` loop, familiar to any imperative programmer. This uses a counter variable `i` which we declare to be the same type as the parameter `n` using Handel-C's `typeof` operator. Evidently, `n` will have to be of some integer type, but Handel-C allows integers of any width, and it is obviously preferable to have `i` and `n` of the same width, in this case. We shall see this technique frequently. The `!` symbol is used in Handel-C, just as in CSP, to denote outputting a value on a channel. We give the channel on the left hand side, and the expression to be communicated on the right. Once all values from `xs` have been communicated on the primary channel for `streamout`, we then output `True` on the corresponding EOT channel.

```
macro proc PRODUCE (n, streamout, xs)
{
  typeof (n) i;
  for (i=0;i<n;i++)
  {
    streamout.value ! xs[i];
  }
  streamout.eot ! True;
}
```

In a vector setting, the implementation would be quite similar. Firstly, considering an implementation in CSP. We effectively change the sequential composition (;) for parallel composition (||), and remove the expression to output on the EOT channel.

$$VPRODUCE_{outs}(xs) \quad = \quad \mathop{||}_{i=1}^{\#xs} outs[i].value!xs[i]$$

In Handel-C the analogy between the two implementations is just as apparent - we essentially replace the `for` with a `par`.

```
macro proc VPRODUCE (n, vectorout, xs)
{
  typeof (n) i;
  par (i=0;i<n;i++)
  {
```

```
    vectorout[i].value ! xs[i];
  }
}
```

3.2 Consume

Consume, the natural opposite of produce, consumes a list of items on a single channel as a stream. This is usually used when we have a process which outputs on more than one channel, but where we effectively wish to discard the values on one of the channels (we shall see an example of this later with the process `TAILSP`). A possible CSP definition of this process might be as follows:

$$\begin{aligned} CONSUME_{in} \quad = \quad & \mu X \bullet \\ & (in.value?x \rightarrow X \mid \\ & \ in.eot?eot \rightarrow SKIP) \end{aligned}$$

The `?` symbol is used in Handel-C, again as in CSP, to denote input of a value from a channel. We give the channel on the left hand side, and the variable which will store the value on the right. This has the obvious implication that we will need to locally declare a variable of the same type as the messages on the channel. This is achieved with `MessageType`, which is a simple macro, again making use of Handel-C's `typeof` operator.

Note the use of Handel-C's `prialt` construct here. This is effectively the same as CSP's external choice operator. We can read this as saying, at each stage of the loop, if the primary channel of `streamin` is willing to communicate, then read a value from it and store it in `x`. If the EOT channel for `streamin` is willing to communicate, read a value from that and store it in `eot`. Recall in standard C that the "not" operator is written `!`, so the expression `(!eot)` below simply checks that `eot` is false - not to be confused with any kind of communication activity. So, after a value is received on the EOT channel, given that this value should always be true, the loop will exit.

```
macro proc CONSUME (streamin)
{
  MessageType (streamin) x;
  Bool eot;
  eot = False;
  do {
    prialt
    {
    case streamin.value ? x:
      break;
    case streamin.eot ? eot:
      break;
    }
  } while (!eot);
}
```

4 Process Refinement

It is important to formally define what we mean by a valid refinement of a function. Let us consider a function f which takes in a list of values and outputs a list of values. Should we wish to refine this to a process P using streams, we can state that P is a valid refinement of f if:

$$PRODUCE(xs) \rhd P \quad = \quad PRODUCE(f\ xs)$$

Here, $\rhd$ is the 'feeding' operator - in which the output channel of the left hand process becomes the input of the right hand process. So, we are saying quite simply, that if the output of process P when fed with the values of xs is the same as directly producing the values of f applied to xs, then P is a valid refinement of f. We can also expand this to a vector based refinement of f. Thus a process Q, which both inputs and outputs using vectors, could be said to be a valid refinement of f if:

$$VPRODUCE(xs) \rhd_n Q \quad = \quad VPRODUCE(f\ xs)$$

Here we make use of a generalised version of $\rhd$ which feeds through a vector of n (in this case the length of xs) channels.

5 Higher-Order Functions

Functional programming environments facilitate clarity of expression and reusability through higher order functions. This stems from the observation that many algorithms can be built from components which are simply problem-specific instances of some more general scheme. In this section we look at the core of this work; how these higher-order functions, taken from a declarative environment can be represented and implemented in a behaviourial way, and how this allows us to expose the inherant parallelism within them. Additionally we must also consider how the composition of these components in the functional specification relates to their combination in a parallel setting.

5.1 Map

Map applies a function to a list of items. We can denote map as an infix operator with the symbol $*$. Thus, in the functional setting, we have:

$$f * [x_1, x_2, ..., x_n] \;=\; [f\ x_1, f\ x_2, ..., f\ x_n]$$

Map is intuitively data parallel. The function f in the above expression can clearly be applied to any given item totally independantly of all the other items. Conveniently also, the size of the input list will always equal that of the output list. Thus it is sensible to consider an implementation of map with vectors as input and output. First however, we might wish to consider a sequential implementation of map, one which takes a stream as input, and also produces a stream as output. This is illustrated in Figure 1.

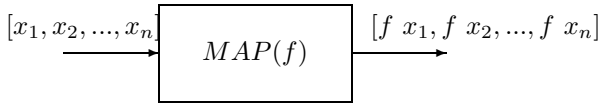


Figure 1. The Process MAP.

A CSP implementation of the stream map process could proceed along the following lines:

$$\begin{array}{l} MAP_{in,out}(f) = \\ \quad \mu X \bullet (\ \ in.value?x \rightarrow out.value\,!\,(f\ x) \rightarrow X \mid \\ \qquad\qquad in.eot?eot \rightarrow out.eot\,!\,True \rightarrow SKIP) \end{array}$$

We input values on the designated input channel, until we receive a message on the corresponding EOT channel. Each time a value is input, we apply the function f to it, and output the result. The tail recursion specification of map in the functional setting translates directly into a simple loop in the imperative setting. This can therefore be implemented similarly in Handel-C as follows:

```
macro proc MAP (streamin,streamout,f)
{
  MessageType (streamin) x;
  Bool eot;
  eot  = False;
  do
  {
    prialt
    {
    case streamin.value ? x:
      streamout.value ! f(x);
      break;
    case streamin.eot ? eot:
      streamout.eot ! True;
      break;
    }
  } while (!eot);
}
```

Here the first two parameters represent the input and output streams on which this process is to communicate. The parameter f is a function (a macro expression in Handel-C) which should be a valid refinement of the function originally used in the functional setting. The operation of the process is then just a case of inputting values from the input channel repeatedly until EOT is signalled, and for each value input, outputting on the designated output channel, the result of f applied to that value.

Returning to the vector setting, a data parallel implementation of map can be implemented as follows:

```
macro proc VMAP (size, vectorin, vectorout, p)
{
  typeof (size) c;
  par (c=0;c<size;c++)
  {
    p(vectorin[c],vectorout[c]);
  }
}
```

The process can be visualised as in Figure 2. Here the parameters correspond closely to those used in the process `map`. One important difference is that we must state explicitly the size of the input, with the parameter `size`, and this value must be a constant. The first two parameters are references to arrays of channels, both of length `size`, which form the input and output of this process.

Finally we pass in a process (not a function) `p`, which should be a valid refinement of the function f. The implementation specifies that `n` instances of the process `p` should execute in parallel, each operating on a different input and output channel.

5.2 Fold Operators

The fold family of functions is used to 'reduce' a list by inserting a binary operator between each neigbouring pair of elements. The basic fold operator / has no concept of direction and as such requires an associative binary operator to be well defined.

$$\oplus/[x_1, x_2, ..., x_n] = x_1 \oplus x_2 \oplus ... \oplus x_n$$

If we can produce the input list as a vector, we can implement this in parallel using a 'funnel' network. It is widely known that this method can calculate the result in $O(\log n)$ time, assuming of course the operator being used works in constant time. However, we will frequently find this particular method is not of such great advantage - where the network as a whole runs in linear or slower time (because one or more components

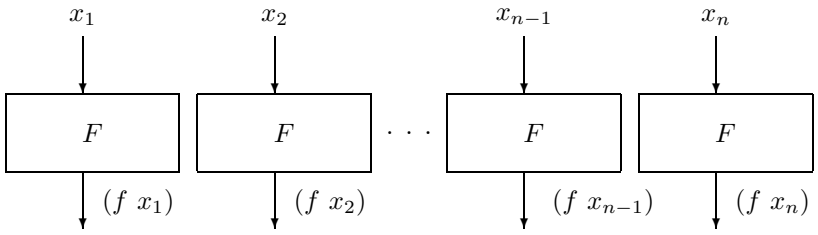


Figure 2. The Process $VMAP$.

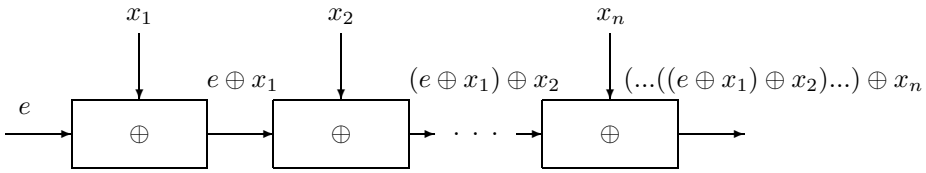


Figure 3. The process $VFOLDL$.

cannot be implemented any more efficiently than this), there is usually no great benefit to a single component operating any better than linear time.

We may encounter two specific interpretations of the fold operator, $foldl$ and $foldr$, which reduce the list from left to right and from right to left respectively, starting at some base value e. We shall consider these both below.

Usefully, both $foldr$ and $foldl$ are valid refinements for $fold$, given that with an associative operator, reduction in either direction will produce the same result. This of course requires that we have some value e that is an identity element for the operator $\oplus$. To avoid this requirement, we also have two commonly found variations, $foldr1$ and $foldl1$. These operate similarly to the two previously mentioned, but do not take a base value, and as such are undefined for the empty list. In fact, it is quite common in a concrete implementation to implement $fold$ in terms of $foldr1$ or $foldl1$. We shall look at these in a little more detail later.

5.2.1 Foldl

Let us first consider the functional version of $foldl$:

$$(\oplus \not\rightarrow e)[x_1, x_2, ..., x_n] = ((e \oplus x_1) \oplus x_2) \oplus ... \oplus x_n$$

Accumulation from the left is a useful characteristic when we consider a behavioural implementation of this function, which employs streams as input and output. We can calculate as we go along, and need only store the current result being received and the accumulated result. The Handel-C implementation might then proceed as follows:

```
macro proc FOLDL (streamin, itemout, f, e)
{
  MessageType(itemout) result;
  MessageType(streamin)  x;
  Bool eot;
  eot = False;
  result = e;
  do {
    prialt
    {
    case streamin.value ? x:
      result = f(result,x);
      break;
    case streamin.eot ? eot:
      break;
    }
  } while (!eot);
  itemout.value ! result;
}
```

Note here that the parameter `f` should again (as with `MAP`) be a Handel-C macro expression which is a valid refinement of the operator $\oplus$ used in the original functional specification.

A vector implementation of this function, `VFOLDL`, places n independant instances of process p together in parallel, where n is the size of the input vector. This can be seen in Figure 3. The Handel-C definition is given below:

```
macro proc VFOLDL (size,vectorin,itemout,p,e)
{
  VectorOfItems
     (size,MessageType(itemout))) mid;
  typeof (size) c;
  par
  {
    ifselect (size==0)
      streamout.value ! e;
    else
      mid[0].value ! e;
    par (c=0;c<size;c++)
    {
      ifselect (c==size-1)
        p (mid[c],vectorin[c],itemout);
      else
        p (mid[c],vectorin[c],mid[c+1]);
    }
  }
}
```

We have used `VectorOfItems` here as a convenient means of defining the internal channels used. Note the use of Handel-C's `ifselect` construct. This is similar to an if statement, but is evaluated at compile time, and the appropriate code is generated. Hopefully there should be a close correspondence apparent between this definition and the diagram in Figure 3. In general, each processing element takes an input from the vector, the result so far from its left hand neighbour, and outputs the result to its right hand neighbour. This behaviour differs slightly at either end - thus the need for the `ifselect` statements. At the left hand side the base value `e` has to be communicated to the first processing element. On the right hand side, the result is output to the output channel, rather than the next processing element within `VFOLDR`.

5.2.2 Foldr

Similarly, to reduce the list from right to left, we can make use of the function $foldr$:

$$(\oplus \not\rightarrow e)[x_1, x_2, ..., x_n] = x_1 \oplus (x_2 \oplus ... \oplus (x_n \oplus e)...)$$

This presents us with slightly more difficulty when we consider a stream implementation. We cannot begin calculation until the last element of the list has been received. As such we need to store all the elements of the list internally, and perform the actual calculation after the end of transmission. For reasons of space we shall not include the full definition, save to say it bears resemblance to that of `FOLDL`, with the addition of an extra stage at the start to read the stream into an array, before the actual calculation of the result takes place.

Similarly, as with $foldl$ we can also come up with an implementation that takes a vector as input. The definition of this in Handel-C is quite similar to that of `VFOLDL`. A graphic depiction of this process would appear similar to that of `VFOLDL` in Figure 3, but essentially with the flow of data going right to left rather than left to right.

5.3 Inits and Tails

We will often find useful certain list processing functions often provided in functional languages. Two such functions are *inits* and *tails*. The first, *inits*, produces all of the initial segments of a given list. We have:

$$\begin{aligned} &inits\ [a_1, a_2, \ldots, a_n] \\ &\quad = [[a_1, a_2, \ldots, a_n], [a_1, a_2, \ldots, a_{n-1}], \ldots [a_1], []] \end{aligned}$$

As a natural companion to this, *tails* produces all of the final segments of a list:

$$\begin{aligned} &tails\ [a_1, a_2, \ldots, a_n] \\ &\quad = [[a_1, a_2, \ldots, a_n], [a_2, a_3, \ldots, a_n], \ldots [a_n], []] \end{aligned}$$

Two further variations on these functions may be seen, $inits_+$ and $tails_+$. These operate in an almost identical fashion to *inits* and *tails*, except that they don't include the empty list in their output. Let us consider the Handel-C implementation of just one of these functions, $tails_+$.

```
macro proc TAILSP (size, streamin, vectorout)
{
  VectorOfStreams
    (size,MessageType(streamin), mid);
  Stream (MessageType(streamin), streamout);
  typeof (size) c;
  par
  {
    par (c=0;c<size;c++)
    {
      ifselect (c==0)
        TLP (streamin,mid[0],vectorout[0]);
      else ifselect (c==(size-1))
        TLP (mid[c-1],streamout,vectorout[c]);
      else
        TLP (mid[c-1],mid[c],vectorout[c]);
    }
    CONSUME ( streamout );
  }
}
```

Here `TLP` is a simple process with one input channel and two output channels. It consumes a stream on its input channel, outputs the stream in full on one output channel, and the tail of the stream (all values except the first) on the other.

6 Composition

Once we have implementations for our various components we need to consider how they are to be assembled together. The goal is to express our functional specifications in compositional form, as functional composition maps directly to communication in the behavioural setting. This is termed piping - where the output channel(s) of one process form the input channel(s) of another. We have two models of piping, depending on which communication method we are using for the data involved. This is basically facilitated in both cases by Handel-C's ability to treats channels as values, therefore allowing them to be passed as parameters.

6.1 Stream Piping

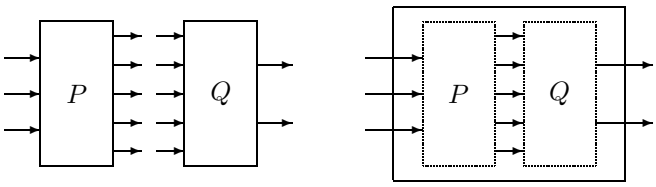
Given a process `P` which outputs a stream, and a process `Q` which inputs a stream, we can achieve a simple stream piping so long as the channels of `P` and `Q` are parameterised. For example:

```
macro proc PIPE ()
{
  Stream (Int, s);
  par
  {
    P (s);
    Q (s);
  }
}
```

The process `PIPE` will create an instance of `P`, and an instance of `Q`, running independantly in parallel, as well as a stream of integers named `s`. This channel is then passed as a parameter to both `P` and `Q`. Assuming processes `P` and `Q` have just a single parameter, and that parameter specifies the output channel (in the case of `P`) and the input channel (for `Q`). We have then effectively created a piping between the two.

6.2 Vector Piping

Vector piping is effectively a generalisation of the above, we just employ arrays of channels rather than single ones. This is depicted in Figure 4. Let us consider the process `VPIPE`. Assuming again we have a process `P` and a process `Q`. This time however, both processes communicate with vectors. It may be that `P` and `Q` use fixed size vectors, but all of the vector processes we have considered so far (`VMAP`, `VFOLDL`, `VFOLDR` and so on) have been scalable to the size of their input, so let us also assume `P` and `Q` are.



The Processes P and Q. P Vector Piped with Q.

Figure 4. Vector Piping.

```
macro proc VPIPE (n)
{
  VectorOfItems (Int, v);
  P (n,v);
  Q (n,v);
}
```

Here `VPIPE` creates an instance of `P` and `Q` as before, in parallel. It also creates a vector, `v`, which is of size `n` - some constant value defined elsewhere (and passed as a parameter to `VPIPE`). We then pass both `n` and `v` as parameters to `P` and `Q`, and again, assuming these parameters specify their output and input respectively, we have a vector piping between the two.

7 Algebraic Laws

One of the biggest advantages of using a functional framework for algorithmic specification is the ability to manipulate the definitions algebraically. This is a product of referential transparency - without side effects it is valid to substitute any expression for any other that we can prove to be equivalent. Particularly we shall find useful the promotion laws. Historically, these are intended to express the idea that an operation on a compound structure can be "promoted" into its components. More details on these can be found in [4].

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

8 Case Study

Let us consider the closest pair problem. Given a set of distinct points ps, in some n-dimensional space, the task is to find the distance between the two closest points. This problem is of quadratic complexity, given that their needs to be $O(n^2)$ comparisons between points if every point is to be compared against

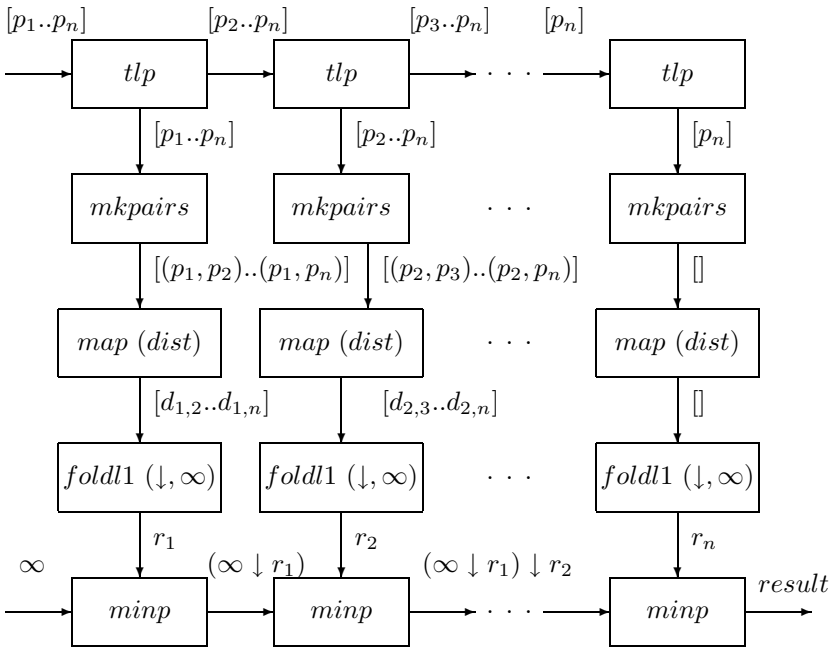


Figure 5. The closest pair network.

every other. Our intuition tells us a solution can be achieved by pairing each point with every other point in the set, calculating the distance between each of these pairs, then finding the minimum of all these distances. In essence we have:

$$closestpair = \downarrow / \circ dist* \circ pairs$$

Here *dist* is defined as follows:

$$dist\ ((x_1,y_1),(x_2.y_2)) = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

There are many different ways of implementing *pairs* here, one possibility is to generate all final segments of the list, and for each segment pair the first element with all elements after this.

$$pairs = +\!\!+ / \circ mkpairs* \circ tails_+$$

The function *mkpairs* can be defined as follows:

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

Here *pair* is a simple function which creates a pair from its two parameters.

$$pair\ x\ y\ =\ (x,y)$$

We can expand this definition of *pairs* into that of *closestpair*, to give us:

$$closestpair = \downarrow / \circ dist* \circ +\!\!+ / \circ mkpairs* \circ tails_+$$

Appealing to map promotion, we can achieve the following definition:

$$\downarrow / \circ +\!\!+ / \circ (dist*)* \circ mkpairs* \circ tails_+$$

Finally making use of reduce promotion, we have:

$$\downarrow / \circ (\downarrow /)* \circ (dist*)* \circ mkpairs* \circ tails_+$$

This is a crucial variation. The function $tails_+$ will create $O(n)$ lists, each of $O(n)$ size, implying an overall data size of $O(n^2)$. Importantly though, the next three compositional stages in the above definition are all map operations. The homomorphic nature of map means each component (in this case each of the lists produced by $tails_+$) can be processed independantly. In short we have exposed data parallelism.

The first stage, $tails_+$ can be refined by the process `tailsp`, introduced earlier. This produces a vector of streams as output, so the next stage should take a vector as input. This next stage in the functional specification is the expression $mkpairs*$. We can refine the $*$ (map) to `VMAP`, and we a require a simple process refinement of the function *mkpairs*. Hopefully we can see that this follows logically from the functional specification:

```
macro proc MKPAIRS (streamin,streamout)
{
  MessageType(streamin) x, y;
  Bool eot;
  eot = False;
  prialt
  {
  case streamin.value ? x:
    MAP (streamin,streamout,pair(x));
    break;
  case streamin.eot ? eot:
    streamout.eot ! True;
    break;
  }
}
```

Given the macro expression `pair` as follows:

```
macro expr pair(p,q) = {p.x, p.y, q.x, q.y};
```

The next stage - the expression $(dist*)*$, can be refined with a combination of `MAP` and `VMAP`. We can use `MAPDIST` as a simple synonym for `MAP` with *dist* as the characteristic function. This can then be passed as a parameter to `VMAP`.

```
macro proc MAPDIST (streamin,streamout)
{
  MAP (streamin,streamout,dist);
}
```

Here `dist` would be a simple expression along the lines of the function *dist* given earlier.

```
macro expr dist(p) = sqrt (square(p.x2-p.x1) +
                           square(p.y2-p.y1));
```

The next stage of the algorithm, $(\downarrow /)*$, can again be implemented with `VMAP`, as the input is still in vector form. Again we need a short synonym process that can be passed to `VMAP`. As noted previously, $foldl$ and $foldr$ are valid refinements of the $fold$ operator, so we could in theory use either here, provided we can come up with some base value e. We shall use the fictitious value `Infinity`, which can be defined as the maximum numeric value allowed.

```
macro proc FOLDMIN (streamin,streamout)
{
  FOLDL (streamin,streamout,min,Infinity);
}
```

We shall find convenient to have an expression for calculating *min*. Note the question mark here is the C-style selection operator, not Handel-C's channel input operator.

```
macro expr min (x,y) = (x<y) ? x : y;
```

In the final stage of the algorithm, we collect all the intermediate results to obtain a single result, with the expression $\downarrow /$. As before, the fold operator $(/)$ can be refined by an implementation of either of the functions $foldl$ or $foldr$ given the base value infinity. The result of the last stage will be output as a vector, so we are again inputting a vector here. This leads us choose the process `VFOLDL`, as the direction of reduction is unimportant. Note as mentioned previously there is no great advantage to using a funnel type, $O(\log n)$ time process here, as the performance of the network as a whole will be linear. We shall require a process version of the function *min* to pass as a parameter to `VFOLDL`.

```
macro proc MINP (in1, in2,out)
{
  MessageType (in1) x;
  MessageType (in2) y;
  in1.channel ? x;
  in2.channel ? y;
  out.channel ! min (x,y);
}
```

We now have all the components required to assemble the parallel, Handel-C implementation of the algorithm as a whole. We shall leave some of the types abstract here - `Coordinate` and `CoordinatePair` would be defined as c-style `structs` and `Distance` would ideally be some kind of floating point value. We parameterise the network with a single input and output channel, and with the size of the input. All intermediate channels are defined locally and are private to `CLOSESTPAIR`.

```
macro proc CLOSESTPAIR (n,streamin,channelout)
{
  VectorOfStreams (n,Coordinate,     vectora);
  VectorOfStreams (n,CoordinatePair, vectorb);
  VectorOfStreams (n,Distance,       vectorc);
  VectorOfItems   (n,Distance,       vectord);

  par
  {
    TAILSP (n,streamin,vectora);
    VMAP   (n,vectora,vectorb,MKPAIRS);
    VMAP   (n,vectorb,vectorc,MAPDIST);
    VMAP   (n,vectorc,vectord,FOLDMIN);
    VFOLDL (n,vectord,channelout,MINP,Infinity);
  }
}
```

The network is depicted in Figure 5. The original functional specification was quadratic time. Through processing each of the lists produced by $tails_+$ independantly in parallel, with $O(n)$ processing elements, the parallel Handel-C implementation will run in linear time.

9 Conclusion

We have demonstrated a methodology that can take intuitive, high level specifications of algorithms in the functional programming style and derive efficient, parallel implementations in Handel-C that can be compiled into hardware on an FPGA. We have provided a concrete example, where the efficiency was improved by an order of magnitude - systematically achieving a linear time implementation from a quadratic time specification. This has exploited both data parallelism - generating a number of lists in parallel and processing them independantly; and functional parallelism - the pipelined processing of the intermediate results. The implementation was achieved by combining behavioural implementations 'off the shelf' of commonly used components that refine the higher order functions which form the building blocks of the starting functional specification.

In future work we shall explore a number of expansions of this methodology. Firstly, it is quite pragmatic to assume that the number of processing elements (related to the number of cells on the FPGA) may well be exceeded by the size of the input data. We shall look at methods of partitioning data for resolving this. Secondly we shall look towards some means of mechanisation, a tool for automatic behavioural synthesis. Finally we shall look to further expanding the calculus of laws and refinements, such that a wider base of algorithms can be accomodated, and further scope for parallelism can be identified.

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] A. E. Abdallah, Functional Process Modelling. In K Hammond and G. Michealson (eds), *Research Directions in Parallel Functional Programming*, (Springer Verlag, October 1999). pp339-360.

[4] 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.

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

[6] R. S. Bird *Introduction to Functional Programming Using Haskell*, (Prentice-Hall, 1998).

[7] Silvia Breitinger, Rita Loogen, Yolanda Ortega Mallen, and Ricardo Pena. *Eden - Language Definition and Operational Semantics*, Technical Report 96-10, Phillips Universität Marburg, 1996.

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

[9] 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**.

[10] S. Gorlatch and C. Lengauer, Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism, *Formal Aspects of Computing* **7** (6) (1995) 663-682.

[11] *Handel-C Documentation*, Available from Celoxica (`http://www.celoxica.com/`).

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

[13] F. Rabhi, Exploiting Parallelism in Functional Languages: A Paradigm Oriented Approach. In J.R. Davy and P.M.Dew (eds), *Abstract Machine Models for Highly Parallel Computers*, (Addison-Wesley,1999).

[14] C. Runciman, and D. Wakeling, *Functional Programming Applications*, (UCL press,1995).

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