Constructing Skeletons in Clean

The Bare Bones

Marco Kesseler
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract

Skeletons are well-suited to structure parallel programming. They allow the easy use of some well-known parallel programming paradigms to construct portable, efficient programs. Much research has been focused on the use of skeletons in functional programming languages, because they can be expressed elegantly as higher order functions. On the other hand, little attention has been paid to an elementary weakness of skeletons: how to implement them. In this paper we will show that functional languages with some low level constructs for parallelism can be used to efficiently implement a range of high level skeletons. We will construct skeletons for data parallelism, for parallel I/O, and for stream processing. Our experiments demonstrate that no performance penalty needs to be paid, compared to more restrictive solutions.

1. Introduction

The development of efficient parallel programs is an important non-trivial problem that programmers and compiler writers have to deal with. The use of functional languages solves part of the problem, because referential transparency allows arbitrary expressions to be evaluated in parallel, without changing the outcome of the program. This makes reasoning about the result of parallel functional programs as easy as for sequential ones. Unfortunately, knowing we cannot compute the wrong result, does not imply we are doing it efficiently. Resource allocation has great effect on parallel performance. Finding the best one for a certain program on a given parallel machine model is difficult. In addition, as there are so many different models, portability is hard to maintain.

Should the compiler allocate resources implicitly, or is it a programmers task to do it explicitly? Clearly, a fully implicit approach would be ideal. It would keep the programmer from making mistakes and it retains portability. However, at this moment no method is known that will automatically derive efficient parallel programs in all circumstances. This means that compilers currently need some form of guidance from the programmer.

The concept of skeletons [7] forms an interesting idea for structuring this guidance. Skeletons can be seen as predefined templates that are used to control parallel execution of programs. The use of these has the advantage that it enables the programmer to exploit certain well known parallel programming paradigms, without having to resort to low level language constructs. In addition, the implementor of a skeleton is able to construct the most efficient one for each platform by taking full advantage of specific machine features. And finally, a certain degree of portability is ensured if the same set of skeletons is provided for all platforms (true portability requires the skeletons to be implementable equally efficiently on different machines).

Many have advocated the use of skeletons in parallel programming environments and in particular, research has focused on functional languages [1, 3, 4, 8, 9, 14, 17]. This is mainly because skeletons can be expressed elegantly as higher order functions. These are a natural part of functional languages and have already been used widely in functional programming, not so much because they may embody parallelism, but because they provide a concise way of expression. Similar concepts form the basis of the idea to use the Bird-Meertens Formalism as a parallel programming model [18].

On the other hand, little attention has been paid to the elementary weaknesses of skeletons: first of all, a set of skeletons has to be implemented on every platform and secondly, a given set may not be very suited to solve some problems efficiently or elegantly. It has already been pointed out by Cole that new skeletons will have to be developed and that ‘the “ad hoc” implementation of each skeleton from scratch on each new architecture would result in much wasted effort’ (although this effort is relatively little compared to that put into application

development using crude tools). Considering this, it is not surprising that most experiments with some form of skeletons have been produced with a small set of data parallel languages on a limited set of machines that support these languages well [1, 6]. So far, we have gained only little experience with skeletons in more common functional languages on more general purpose machines [3, 4, 14]. This indicates that an intermediate level of abstraction is desirable.

In this paper, some examples will illustrate how Concurrent Clean - a lazy, higher order functional language with a few low level constructs for parallelism [15, 17, 19] - can be used to implement a range of high level skeletons. In contrast to Darlington [9], not only the *meaning* of each skeleton will be established by its functional language definition, but also its *behaviour*. Thus, we will use a lazy functional programming language with general mechanisms for unstructured parallelism only, to capture structured parallelism. This not only allows to build a structured system that can be extended easily, but it also permits mixing of structured and unstructured parallel programming.

Power of expression is important, but actual speed is crucial in a parallel system. Some experiments on a general purpose machine with distributed memory[1] will make clear that no performance penalty needs to be paid, compared to parallel functional (Concurrent Clean) programs without skeletons [5, 16], and compared to other functional languages that provide skeletons in more restrictive ways [4, 14]. With respect to imperative languages like C our results suffer from sub-optimal sequential compilation techniques. It is not the use of the skeletons that causes problems. If we manually optimise the code (only a little) Concurrent Clean programs become competitive with C. But before we take a look at these examples, we will present the building blocks of our language.

2. The Bones: Concurrent Clean

Concurrent Clean is a lazy, higher order functional language with support for parallelism. It has a Haskell-like syntax and enables the programmer to control parallelism explicitly by means of annotations. These cannot influence the outcome of a program, they merely change the order in which redexes are reduced. The following annotations are available.

- **{I}**: This annotation starts up a new process on the current processor. It will reduce the annotated expression to root normal form (RNF). No communication takes place, instead, the new process shares nodes with other processes. Processes on the same processor will run interleaved and scheduling is fair.
- **{P}**: This annotation starts up a new process at some other processor (if there exists one) that will reduce the annotated expression to RNF. Only the annotated function will be evaluated at the remote processor. The arguments will always be evaluated locally and transportation of arguments will always take place *after* evaluation. If they are strict, they will be evaluated before starting the new process, else they will be evaluated lazily when (and if) the new process needs its arguments.
- **{P at** processor**}**: The same as **{P}**, only now the expression 'processor' evaluates to a processor-id and the new process will be started at the given processor. Basically, all annotations are some form of the **{P at ...}** annotation.

All communications take place implicitly, except for the ones implied by the annotations above. Objects are represented by graphs. If a function needs one that is stored at another processor, this graph will automatically be transported to the function. For this, a graph copying mechanism is used that copies as much of a graph as possible at once. However, only evaluated parts will be transported directly. If a function needs a structure that has not yet been reduced, it will not be shipped immediately. Instead, a new lazy process will be started on it and only after it has been reduced to RNF will it be transported, again for as far as it has been reduced. The effect is that only normal forms can travel implicitly between processors, closures are always evaluated before copying and act as 'copy-stoppers'. This 'Lazy Normal Form Copying' [13] provides the concept of distributed processes, all reducing lazily to normal form, although in reality each actual process only reduces to RNF. Amongst others, this has the advantage that work cannot get copied implicitly and that the programmer can easily determine what gets copied by looking at the type of functions.

2.1. Auxiliary Functions

With the annotations presented above, it is possible to introduce and control parallelism at a very low level. Nonetheless, it turned out to be useful to add a few auxiliary functions. The implicit communication mechanism transports evaluated arguments to functions that need them, where they can be used for further processing.

[1] Our results have been obtained with a Concurrent Clean compiler for a network of T800 transputers.

It is a little cumbersome to achieve the converse, that is: transport a function to an argument and apply it to this argument there. For this reason we have defined the following functions and types in Clean[2].

```
:: R x = Remote x

Ap :: (x -> y) (R x) -> R y
Ap f (Remote x) = ApAt (ArgId x) f x
where
  ApAt :: ProcId (x -> y) x -> R y
  ApAt p f x = Remote ({P at p} f x)

GetRemote :: (R x) -> x
GetRemote (Remote x) = x

Id :: x -> x
Id x = x
```

First of all, we have devised a polymorphic higher order function 'Ap' that is able to transport a function to a remote argument and apply it to this argument. The result is again a remote object. It uses the low level function 'ArgId' that returns the location of its argument; more precisely, it returns the position of the root node of its argument. This may seem to violate referential transparency, but, as processor ids can only be used in annotations, pureness is preserved in this 'para-functional' programming style. Even so, the 'ArgId' function is completely hidden in the definition of the 'Ap' function. It is not available for programmers. In a sense, 'Ap' can be seen as a very basic skeleton that hides these confusing details.

Secondly, we have introduced a new algebraic type that indicates - or rather suggests - that a data structure is at another processor. This is useful for various reasons. Firstly, it helps the programmer to keep track of remote structures. In Clean there is no obvious difference at the language level between a local object and a remote one. By declaring some objects to be remote, the programmer is reminded of their special characteristics. Secondly, the extra constructor 'Remote' allows functions - always reducing to RNF - to return a truly remote result: a reference to a result at another processor. This becomes apparent in the definition of the 'ApAt' function above. Without the extra constructor it would not only apply a function at a remote argument, but it would also subsequently evaluate the result, so that it does not remain remote. This would not be the intended behaviour. And finally, functions that have remote arguments need to perform a pattern match on the 'Remote' constructor to get hold of the actual reference to the argument. This ensures that any indirections that may have been added after constructing the remote object will conveniently be removed automatically. This makes it easy for the 'ArgId' function to find out the exact location of its argument. It will not accidentally return the location of some selector function for example.

And finally, two simple functions have been defined. The 'GetRemote' function turns a remote object into a local one, using the standard evaluation and communication mechanism. A 'PutRemote' function is not very useful, as this can easily be provided by the standard **{P}** annotation in conjunction with the 'Remote' constructor. The identity function is sometimes necessary to act as a copy-stopper. This is useful in case the 'Remote' constructor is produced by a remote function, and not locally, as is the case in the 'ApAt' function above. If so, we need to insert a copy-stopper between the constructor and the object, so that evaluation and transmission of the constructor does not trigger transmission of the object as well.

Combined with these functions, the standard Clean annotations are powerful enough to easily define a range of high level skeletons, as we will see in the following sections. We will construct skeletons that provide efficient data parallel constructs, skeletons to perform parallel I/O, and finally, skeletons that describe stream processing.

3. Skeletons for Data Parallelism

Unlike Sisal [2, 6], Concurrent Clean is not a strict language dedicated to data parallelism. One of its major shortcomings has been the lack of suitable constructs for this kind of parallel processing. In this section we will show that functional languages not only can be used *as* data parallel languages, but also to *implement* data parallel constructs efficiently.

3.1. Arrays and Matrices

Arrays and matrices undoubtedly are the most prominent data structures in data parallel processing. Clean does have some support for matrices and arrays, but these are not 'data parallel' in the sense that they are automatically distributed over a number of processors. Instead, each array is associated with a single contiguous block of memory on a single processor, much like arrays in sequential imperative languages. In addition, referential

[2] In this paper, we will use Clean 1.0 syntax, as this conforms to the usual notations, but the actual tests have been performed with Clean 0.8 for technical reasons.

```
Map3 :: [[R x1]] [[R x2]] [[R x3]] (x1 x2 x3 -> y) -> [[R y]]
Map3 [row1 : rows1] [row2 : rows2] [row3 : rows3] f
  = [ Map3Row row1 row2 row3 f : Map3 row1 rows2 rows3 f ]
Map3 [] [] [] f = []

where Map3Row :: [R x1] [R x2] [R x3] (x1 x2 x3 -> y) -> [R y]
      Map3Row [elem1 : elems1] [elem2 : elems2] [elem3 : elems3] f
        = [ Ap (Map3Elem f elem1 elem2) elem3 : Map3Row elems1 elems2 elems3 f
]
      Map3Row [] [] [] f = []

      Map3Elem :: (x1 x2 x3 -> y) (R x1) (R x2) x3 -> y
      Map3Elem f e1 e2 e3 = f (GetRemote e1) (GetRemote e2) e3
```

transparency forces single-threaded access on these arrays if updates are to be done in place[3].

Despite this sequential nature of arrays in Clean, they are very suited for parallel programming. Compared to lists, strict arrays can be considerably more efficient when they are to be transmitted to other processors. For instance, using arrays instead of lists can dramatically speed up parallel matrix multiplication, even if one uses a very straightforward algorithm without any data parallel constructs [12].

Below, we will demonstrate how an efficient implementation of data parallel matrices can be obtained from the standard matrices in Clean. First we will choose the structure of the distributed matrix and define some - rather general - data parallel operations on it. Next, we will use these to construct a data parallel matrix multiplication program that is based on Gentleman's algorithm [10]. And finally, we will present the execution times of this program, which will make clear that building data parallel skeletons in Clean does not result in a performance penalty.

3.2. Operations on Distributed Matrices

The idea for obtaining a distributed matrix, is to define a local structure that refers to remote substructures. Using the definitions given earlier we can simply define parallel matrices to be a double list of remote submatrices, as depicted below. Choosing a list of lists is rather arbitrary though. One could have used a different structure, like for instance a quad-tree, or a doubly linked circular list, or several, while using some natural transformation functions to get from one form to the other.

```
:: PMat    :== [[R Matrix]]
```

Map- and Fold-like operations are crucial in data parallel programming languages. We will give Clean definitions of each kind below. These functions are polymorphic and operate on double lists of general remote objects. They can be applied to objects of type 'PMat', but also to different distributed objects that use a double list as local structure.

The 'Map3' function shown above constructs a new parallel structure out of three. The remote elements of the new structure will contain 'f a b c', where a, b and c are the corresponding remote elements of the argument structures. The function f is applied at the processor that contains 'c' and this will also be the location of the resulting element.

This definition is completely lazy. It is not parallel, which may seem odd. Our aim however, is to provide the kind of *data parallel laziness* that is proposed by Hill [11]. The parallel construct we have defined above is nothing more than a description of what should happen where when it is needed. It is still a bit too eager, because the 'Ap' function embodies a **{P}** annotation that starts up a new process. It would be sufficient to just *place* a function at the correct processor, without starting to reduce it. Having this lazy definition of map, an additional eager mechanism is required to actually introduce parallelism where necessary. This is closely related to standard sequential lazy evaluation, where computations are driven by an eager printing mechanism. In our case true parallelism may be introduced by an eager parallel fold function.

The following fold function applies the function g to each remote element and folds the results with the function f. Normally, g will be a fold function over the structure of the remote elements. Most concurrency can be found in the parallel application of g, but 'Fold' is not as parallel as could be, because there is a sequential

[3] In Clean, destructive updates are allowed if derived *uniqueness* properties indicate that an object is not shared [20].

dependency between the applications of f. A trivial solution would be to split the double list recursively and fold it in a tree-like manner, provided that f is associative.

```
Fold :: [[R x]] (y y -> y) (x -> y) y -> y
Fold [row : rows] f g e
  = Fold rows f g ({I} FoldRow row f g e)
Fold [] f g e = e

where
  FoldRow :: [R x] (y y -> y) (x -> y) y ->
y
  FoldRow [elem : elems] f g e
    = FoldRow elems f g e'
      where
         e' = {I} GetRemote (Ap f' elem)
         f' = Compose (f e) g
  FoldRow [] f g e -> e

  Compose :: (y -> z) (x -> y) x -> z
  Compose f g x = f (g x)
```

Transformations like shifts and rotations can easily be defined on the local structure of the distributed object. They merely perform some permutations on the references to the remote structures. In this way, communications are automatically postponed until needed. This means that the transformation 'RotateLeft (RotateUp x)' does not introduce more communications than some function 'RotateLeftAndUp x'.

```
RotateUp :: [[x]] -> [[x]]
RotateUp [row : rows] = Append rows [row]
RotateUp [] = []
```

And concluding, distributed objects can be created with functions like the map function above. Only now, each occurrence of 'Ap f x' should be replaced by 'Remote **{P AT** ...**}** (CreateElement ...)'. Again, the 'CreateElement' function can be passed as an argument.

Note that the map and fold operations defined above just happen to be specialised to lists of lists. Constructing these operations for other kind of data structures is fairly straightforward. It comes down to walking the particular structure while distributing and inserting functions in a similar way.

3.3. Data Parallel Matrix Multiplication

The functions that we have defined in the previous paragraphs - and a few others for rotating and skewing matrices that can be constructed similarly to the 'RotateUp' function - allowed us to implement Gentleman's algorithm in a straightforward way. Here, we will list the main part of the program.

```
Mul :: PMat PMat -> PMat
Mul a b = Mul' a' b' zero rotate_n
where
  a' = SkewLeft a
  b' = SkewUp b
  zero = NewMat ZeroMat
  rotate_n = Length a

  Mul' :: PMat PMat PMat Int -> PMat
  Mul' a b c 0 = c;
  Mul' a b c n
     = Mul' a' b' (Map3 a b c MulAdd) (n -
1)
        where a' = RotateLeft a
              b' = RotateUp b
```

The 'NewMat' function creates a new distributed matrix and fills it according to its argument function. The skew and rotate functions have been defined in the same way as the 'RotateUp' function above. 'Length' is a function that computes the length of a list. It tells how many rotations are necessary. 'MulAdd' does all the work. It is a sequential function that multiplies two ordinary matrices and adds the result to another.

The whole computation is driven by the function 'Fold' as shown below. This is not part of Gentleman's algorithm, but it is used to guarantee parallel evaluation. If we omit the fold function the whole resulting matrix gets printed on screen. This happens lazily, so we loose parallelism, while gaining I/O overheads. Conversely, if we merely select a single submatrix from the result, only this element will be computed on its own processor. The rest will remain idle. The fold function avoids this. It ensures that all elements of the result are computed in parallel.

```
Fold (Mul a b) (&&) MatrixToBool True
where a = NewMat SomeMat
      b = NewMat AnotherMat
```

3.4. Performance

The following results have been obtained with matrices containing floating point numbers of 64 bits. The tests have been performed on a torus of T800 transputers running at 25 MHz and having 4 Mbyte of memory each. The parallel versions have been compared to a purely sequential matrix multiplication program that contains no overheads for parallelism. Although the actual network topology matches the matrix structure, no effort has been made to place submatrices that are neighbours logically at

physically neighbouring processors (for our hardware provides no clear relation between processor numbering and the physical network). This increases communication overheads, especially for large networks and small matrices.

	Number of Processors			
Matrix Size	1	4	16	64
32 x 32	1.0 sec	0.4 sec	0.4 sec	1.7 sec
64 x 64	7.6 sec	2.1 sec	1.0 sec	1.8 sec
128 x 128	60.3 sec	15.6 sec	5.1 sec	2.9 sec
256 x 256	480.7 sec	121 sec	36.3 sec	11.3 sec
512 x 512	3846 sec*	966 sec*	281 sec	73.8 sec

Fig. 1 Execution times of matrix multiplication in Clean

	Number of Processors		
Matrix Size	4	16	64
32 x 32	2.5	2.5	0.6
64 x 64	3.6	7.6	4.2
128 x 128	3.9	11.8	20.8
256 x 256	4.0	13.3	42.5
512 x 512	4.0*	13.7*	52.1*

Fig. 2 Speed-up of matrix multiplication in Clean

The star indicates where we have used estimations, as matrices of size 512 by 512 are too big to be accommodated on a single processor (we are using a copying garbage collector, which halves the amount of memory available). Matrix multiplication is of $O(n^3)$, so it is not unreasonable to assume it will take 8 times longer to multiply two matrices of size 512 by 512 on one processor, than it takes to multiply two of size 256 by 256. This assumption is strongly supported by the results we have obtained for the sequential program.

The figures presented above are about as good as the ones we have obtained earlier in Concurrent Clean without using data parallel constructs [12], and they compare favourably to the ones presented for π-RED$^+$ [5] and the Data Parallel Functional Language (DPFL) presented in [14]. Both give similar speed-ups, but have worse absolute performance (No absolute performance figures have been presented for π-RED$^+$, but this is an interpreter). Other implementations of functional languages on machines with distributed memory either do not list results for matrix multiplication, or they are considerably less efficient.

	Number of Processors	
Language (Processor)	4	25
Concurrent Clean (25 MHz T800)	902 sec*	169 sec
DPFL (20 MHz T800)	3127 sec	510 sec

Fig. 3 Execution times of Clean and DPFL for multiplication of matrices of size 500 x 500.

More interesting than execution times however, - as this largely depends on sequential code quality -, is that DPFL uses a set of custom data parallel constructs. To implement these, a communication mechanism has been suggested that uses version numbers to maintain the correct ordering of messages and operations. Though messages may pass each other in Clean as well, we do not need such a construct, because the correct order of operations is automatically maintained by data dependencies. In addition, we do not rely on uniqueness properties [20] as heavily as DPFL. Instead of requiring each instance of an entire array to be unique, we only need this property temporarily during sequential *construction* of some remote sub-array. Usually, this requirement will be met by the array-constructing argument functions of the skeletons. We have not (yet) made use of uniqueness properties within the definition of the skeletons themselves.

With relation to imperative languages and languages like Sisal [2], it is important to know that the tested version of Clean does not optimise 'loops' on arrays, nor does it generate code for array selection. Instead it uses small functions like 'Put' and 'Get' to insert or extract elements. This means that the Clean system used for these tests is not yet as fast as highly optimising data parallel languages, but this is purely a technical matter.

Matrix Size	Helios C	Pact C[1]	Pact C[2]
32 x 32	0.2 sec	0.2 sec	0.3 sec
64 x 64	1.9 sec	1.8 sec	2.2 sec
128 x 128	15.7 sec	14.3 sec	18.2 sec
256 x 256	131 sec	117 sec	146 sec
512 x 512	1046 sec*	938 sec*	1171 sec*

Fig. 4 Execution times of matrix multiplication in C on 1 processor. The figures for the Pact Compiler indicate execution times obtained with the stack in fast on-chip memory (1) and in off-chip memory (2). The latter is more realistic for parallel programs with multiple processes on a single processor. In Clean and in Helios, stacks are always placed off-chip. The Helios compiler produces the best code in this case.

	Number of Processors				
Size	1	16		64	
32 x 32	0.3 sec	0.4 sec	x 0.8	1.7 sec	x 0.2
64 x 64	2.3 sec	0.6 sec	x 3.7	1.8 sec	x 1.3
128 x 128	17.6 sec	1.8 sec	x 9.6	2.5 sec	x 7.0
256 x 256	139 sec	10.2 sec	x 13.6	5.0 sec	x 27.8
512 x 512	1112 sec*	73.9 sec	x 15.1	22.1 sec	x 50.3

Fig. 5 Execution time and speed-up in Clean after inlining the array selection functions inside the innermost vector multiplication loop.

If we manually inline the 'Get' functions for the inner vector multiplication loop we obtain the results above. These are competitive with C. A further improvement can be expected if we employ loop optimisations that are commonly found in languages like FORTRAN and SISAL. Considering the code that the Clean system currently produces, we expect that performance can be improved approximately by factor of two.

We have not been able to compare Clean with SISAL in a sensible way. Mainly because the architectures for which they are available differ greatly. The performance figures obtained on transputers are not conclusive with respect to performance on a Cray (and vice versa). We end up comparing architectures instead of languages. Eventually, an important question will be whether laziness allows the same level of code optimisation (copy elimination) as has been achieved in SISAL.

4. Skeletons for Parallel I/O

I/O can be a bottleneck if only one process is allowed to do it. Typically, machines have many independent devices for I/O that all operate in parallel. A single thread of control may not be able to drive all of these efficiently, either because it is too slow, or because it introduces unnecessary sequentialisation of I/O. The latter is not unlikely in a pure functional language without a non-deterministic merge function. Instead, it would be best to split up the load and let different processes - and processors - perform various I/O tasks in parallel.

To accomplish this, one needs to divide an I/O system at some level into several independent parts. For instance, one could view a picture as a set of independent pixels. Different processes may then colorize these pixels in parallel, which is all right as long as the order in which this happens does not matter. If we are only interested in the final result, which will always be the same, this is no dangerous form of non-determinism.

At some level however, this concept is unmanageable. There are moments one would rather like to manipulate a picture as a whole, or handle a properly ordered sequence of pictures, or deal with some other object in which the pixels are united, without loosing the efficiency of parallelism. In these cases one needs to supply parallel I/O within some framework. This ordered concurrency can be provided by skeletons in exactly the same way as they provided data parallelism. Only now, we are not working with parallel data structures, but with parallel I/O devices.

4.1. Plotting Pixels in Parallel

We will stay with the picture example and show how to use the two skeletons below to build and draw a picture in parallel. Again, these skeletons have been constructed in Clean with the primitives presented above.

```
NewPPict :: Size (Int Int -> Pixel) -> PPict
PlotPPict :: Window PPict -> Window
```

The 'NewPPict' function is almost the same as the data parallel 'NewMat' function. It lazily creates a distributed picture, in which each remote part is a little square of pixels that represents a distinct part of the image. The only difference with the matrix creating skeleton is that it does not build a local list of lists, but instead it constructs a distributed one. We will see below why this is useful. 'NewPPict' has two arguments. The first is a tuple of four integers that describe the size of the picture (respectively, the horizontal and vertical size of the picture and the horizontal and vertical size of each remote square). The second argument is a function that describes the scene. It delivers the colour of each pixel according to its co-ordinates in the image.

'PlotPPict' is able to draw such a distributed picture in a window, and it does this for all image parts in parallel. It has been defined analogous to the 'Fold' function and drives parallel computation by eagerly applying a plotting function to each remote part of the image simultaneously (similar to 'g' in 'Fold'). For this it not only splits the picture recursively into its basic components, but it also breaks down the window structure into corresponding sets of pixels. The result of the various plotting functions is then combined to form a new window, similarly to the function 'f' in 'Fold'. In this way windows can be treated as a whole, while incorporating parallelism.

```
PlotPPict Window (NewPPict size fill)
where
  size = (560 320 20 20)
  fill = Mandelbrot_color

PlotPPict Window (NewPPict size fill)
```

```
where
  size = (300 300 10 10)
  fill = TraceColour screen scene
```

We have tested these functions in two programs, of which the main call has been listed above. The execution times can be found below. In both cases, the 'fill' function passed to 'NewPPict' is a plain sequential function that does not contain any construct for parallelism. Conversely, 'NewPPict' itself scatters a considerable number of processes over the network: 448 and 900 respectively. One program is quite simple and draws a mandelbrot set. The other is fairly extensive and generates a scene of polygons using a ray-tracing algorithm. To accomplish this, it needs to distribute a collection of objects - the 'scene' - over all processors. This is done automatically by the 'Ap' function contained in 'NewPPict', but to avoid doing it from a single processor for every process, we have kept the 'PPict' structure as distributed as possible (a broadcast may have been better, but the graph copying mechanism does not support this yet).

# processors	1	16	32	64
MandelBrot	153.5 sec	13.6 sec	9.2 sec	6.3 sec
RayTrace	331.3 sec	33.0 sec	17.4 sec	9.9 sec

Fig. 6 Execution time with parallel plotting

# processors	16	32	64
MandelBrot	11.3	16.7	24.4
RayTrace	10.0	19.0	33.5

Fig. 7 Speed-up with parallel plotting

Although it seems that first a picture is generated, and then plotted, this is not the case. As in the data-parallel map, construction of the image is lazy, and therefore will take place at the moment the plotting function requires it: when it starts to plot a basic part of the picture. So reading ';' as 'followed by' and '||' as 'in parallel', instead of $((gen_1 \parallel gen_2 \parallel ... \parallel gen_n) ; (plot_1 \parallel plot_2 \parallel ... \parallel plot_n))$, we get $((gen_1; plot_1) \parallel (gen_2;plot_2) \parallel ... \parallel (gen_n;plot_n))$. This shows better on screen what is going on, while avoiding unnecessary sequentialisation: each part of the image is plotted as soon as it is available. This spreads plotting over time, which is more efficient in many cases.

# processors	1	16	32	64
MandelBrot	155.5 sec	18.5 sec	12.7 sec	9.2 sec
RayTrace	392.5 sec	36.1 sec	22.4 sec	16.8 sec

Fig. 8 Execution time with sequential plotting

In our system parallel plotting was able to improve execution time considerably - up to a factor of 1.7 -, compared to having a single process that draws the image sequentially (see figure 8). The latter causes all pixels to be plotted in a fixed order, as in $(gen_1 \parallel ... \parallel gen_n \parallel (plot_1; ... ;plot_n))$. As a result, most of them are plotted in a single burst after computing the whole picture. This increases execution time notably because the underlying system transfers all pixels from the transputer network to a server via a single transputer link[4]. Using parallel plotting, transferring the pixels gets overlapped with ordinary computation to such an extent that plotting overheads are barely noticeable (compare figure 6 and 9).

# processors	1	16	32	64
MandelBrot	141.5 sec	13.6 sec	9.2 sec	6.3 sec
RayTrace	321.0 sec	31.3 sec	16.7 sec	9.9 sec

Fig. 9 Execution time without plotting

The mandelbrot program in Clean is a factor 1.5 slower than an iterative version in C, again caused by suboptimal code quality. Recursive versions in C on the other hand, perform worse than Clean, even after some tuning. Compared to SkelML [4], which is strict, the raytracer in Clean is a factor 3.5 faster, although the speed-ups are comparable (SkelML uses OCCAM as an intermediate language, whereas the Clean system generates transputer assembly directly). As with the matrix multiplication example, we have not been able to compare our results in a sensible way with other implementations of functional languages on distributed memory machines.

Load imbalance - caused by the irregular structure of these problems - largely explains the limited speed-up. The use of skeletons does not degrade performance here. Generating a distributed pixel map is not more difficult than generating a distributed matrix. This means that speed-ups can be at least as good as in figure 1 if we create an image by filling it with a function that takes constant time for all co-ordinates.

5. Skeletons for Streams

Our last example involves the use of skeletons to construct streams. These provide pipelines and process networks (which can be used to simulate systolic arrays, for example). To some extent, the low level constructs for

[4] This link forms the only connection to the outside world, so we have not been able to test the effects of true parallel I/O.

```
StreamTree :: (Tree (R [x])) ([x] [x] -> [x]) ([x] -> [x]) -> [x]
StreamTree list node_f leaf_f
  = Stream BuffSize (Remote {P at ItoP StartProc} root)
where root = StreamTree' list node_f leaf_f StartProc BuffSize

where
   StreamTree' :: (Tree (R [x])) ([x] [x] -> [x]) ([x] -> [x]) Int Int -> [x]
   StreamTree' (Leaf list) node_f leaf_f proc size = leaf_f (Stream size list)
   StreamTree' (Node left right) node_f leaf_f proc size = node_f arg1 arg2
   where   arg1         = Stream newsize (Remote {P at ItoP leftP} stream_left)
           arg2         =  Stream  newsize  (Remote  {P  at  ItoP  rightP}
stream_right)
           stream_left  = StreamTree' left node_f leaf_f leftP newsize
           stream_right = StreamTree' right node_f leaf_f rightP newsize
           leftP        = NewLeftP proc
           rightP       = NewRightP proc
           newsize        = Half size
```

parallelism in Clean are very well suited to express this kind of parallelism directly. This also holds for certain divide-and-conquer style programs, which means that we may not always need to employ skeletons. Examples are well-known programs like 'queens' and the sieve of Erathostenes. Using basic Clean constructs, we have obtained good speed-ups for both in a rather straightforward way, although the sieve required a special buffering function between distinct filter functions [12]. These examples are very simple however, and inserting special functions, such as buffers, may be hard sometimes. In these cases, skeletons are invaluable.

The parallel mergesort program below is a divide-and-conquer style program that uses streams. It exploits both horizontal and vertical parallelism to sort a tree of unsorted lists. First it applies a sequential mergesort to each list in parallel, and then it merges the resulting lists into one big sorted list. Streams are used to overlap these computations as much as possible. The whole program has been based on the 'StreamTree' Skeleton. This function has been defined in the frame above.

'StreamTree' gets a tree of remote lists as an argument and two functions: 'node_f' and 'leaf_f'. It applies the function 'leaf_f' to each list in the tree (at the leaves), and it uses the function 'node_f' to combine the results recursively (at the nodes). Note the similarity with the fold and map functions defined earlier. The main difference is that we are now dealing with trees instead of lists of lists. Additionally, it inserts the buffering function 'Stream' where appropriate, that is, between functions executing on different processors. The size of the buffer gets halved each step down the tree. With the 'StreamTree' function we can easily construct the following mergesort algorithm.

```
MergeSort :: (Tree (R [Int])) -> [Int]
MergeSort list = StreamTree list Merge Sort

Sort :: [Int] -> [Int]
Sort [x]   = [x]
Sort list  = Merge (Sort left) (Sort right)
where (left, right) = Split list [] []

Merge :: [Int] [Int] -> [Int]
Merge alist=:[a : as] blist=:[b : bs]
  | Less a b   = [a : Merge as blist]
  | Otherwise  = [b : Merge alist bs]
Merge alist [] = alist
Merge [] blist = blist

Split :: [Int] [Int] [Int] -> ([Int],[Int])
Split [a : [b : rest]] left right
  = Split rest [a : left] [b : right]
Split [a] left right = ([a : left],right)
Split [] left right  = (left,right)
```

The figures below lists the results of this program for different input sizes. The tree of unsorted lists is generated at a single processor. Two versions have been tested. One that uses the standard integer comparison, and one that employs a complex comparison. Compared to earlier experiments with special stream functions in Clean [16], the use of buffers gives better performance, while improving memory usage.

# processors	1	8	16	32
Standard 20000	15.6 sec	4.6 sec	3.5 sec	3.9 sec
Standard 40000	59.8 sec	11.2 sec	8.0 sec	9.1 sec
Complex 20000	79.2 sec	19.4 sec	12.2 sec	10.9 sec
Complex 40000	196.0 sec	42.6 sec	25.3 sec	21.7 sec

Fig. 10 Execution time for parallel mergesort

# processors	8	16	32
Standard 20000	3.4	4.5	4.0
Standard 40000	5.3	7.5	6.6
Complex 20000	4.1	6.5	7.3
Complex 40000	4.6	7.8	9.0

Fig. 11 Speed-up for parallel mergesort.

The reason that speed-ups are limited is two-fold. First of all, - despite the use of buffering functions -, delays can be substantial. Our buffering mechanism is too simple to avoid all delays, which may be caused by complex data dependencies. And secondly, the algorithm contains only a limited amount of inherent parallelism. About half of the available processors is used to sort the remote lists in parallel and the rest for merging the results. One cannot be faster than the time it takes to sort one sublist, which is a problem for small networks. Conversely for large networks, having a tree of processes, the root becomes a bottleneck. At a certain point, adding more processors does not help. It could even harm, as it may cause more overhead at some processors (for example, at the list generating processor), while also introducing longer data paths. The effects can be quite complex and account for the speed-down for 32 processors compared to 16 with the standard integer comparison.

Having poor speed-ups for this particular program does not mean that stream-like processing is not useful at all. First of all, better speed-ups can be expected for programs that have more regular data dependencies, such as systolic arrays of the form $f_1(f_2(...f_n))$, where n is sufficiently large. Even the sieve of Erathostenes gives better speed-ups - up to 16 on 32 processors -, although the first filtering process forms a bottleneck. Secondly, many large computations contain compositions of functions working on streams. These computations can be overlapped to some extent. There may not be enough overlap to keep many processors busy, but the speed-up may be high enough nonetheless.

The importance of this becomes more apparent if we consider that data-parallel operations can easily be combined with stream-like processing. The latter may be used for consuming small numbers of processors, while the former is responsible for the big speed-up factors. For instance, one could map a stream-like parallel signal processing function - consisting of several filters - over a distributed database of signals. Each signal processing function will account for only a limited speed-up, but it does effect a better utilisation of processing power. Skeletons as defined above are powerful tools to accomplish this. Being higher order functions, skeletons for stream-like processing can be passed to skeletons for data parallelism (and vice versa), so that a new skeleton is composed out of old ones.

```
MapFilters :: (Struct s) [s -> s] -> Struct
s
MapFilters database filters
  = Map database (ComposeStream filters)
```

6. Conclusions

We have used the functional programming language Concurrent Clean to construct a number of skeletons. In this way we easily obtained efficient skeletons for data parallelism, for parallel I/O, and for streams. They have been tested and show good performance compared to more restrictive solutions. Constructing skeletons in this way makes it easy to extend a system with new skeletons, tailored for specific needs.

7. Future Work

The primitives presented above allow functions to ‘travel’ over distributed structures easily. This offers an interesting starting point for research on distributed functional databases, using skeletons for searching and updating. These skeletons may then be parametrised with functions that perform local transformations on the database.

Acknowledgements

I would like to thank Herbert Kuchen for providing the latest information on DPFL [14].

References

[1] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha “Implementation of a Portable Nested Data-Parallel Language” In *Principles of Programming Languages*, pages 102--111, May 93.

[2] A.P.W. Böhm, J. Sargeant, “Code Optimisation for Tagged Token Dataflow Machines”, *IEEE Transactions on Computers*, Vol 38, Nmb 1, January 1989.

[3] Tore A. Bratvold, “Parallelising a Functional Program Using a List-Homomorphism Skeleton”. *The First International Symposium on Parallel Symbolic Computation, PASCO ’94,* Hagenberg/Linz, Austria, pages 44-53, World Scientific, 1994.

[4] Tore A. Bratvold, “A Skeleton-Based Parallelising Compiler for ML”. In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages,* Nijmegen, the Netherlands, pages 23-33, September 1993.

[5] Torsten Bülk, Achim Held, Werner Kluge, Stefan Pantke, Carsten Rathsack, Sven-Bodo Scholz, Raimund Schröder, “Preliminary Experience with a π-RED$^+$ Implementation on an nCUBE/2 System”, In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages,* Nijmegen, the Netherlands, pages 101-113, September 1993.

[6] David Cann “Retire Fortran? A debate rekindled” *Communications of the ACM* , 1992.

[7] Murray Cole, “Algorithmic Skeletons: Structured Management of Parallel Computation”. Research Monographs in Parallel and Distributed Computing. Pitman/MIT, 1989.

[8] Marco Danelutto and Susanna Pelagatti. “Parallel Implementation of FP using a Template-based approach”. In *Proceedings of the Fifth International Workshop on the Implementation of Functional Languages,* Nijmegen, the Netherlands, pages 7-21, September 1993.

[9] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While “Parallel Programming Using Skeleton Functions” In *Parallel Architectures & Languages Europe, PARLE ’93,* pages 146-160 LNCS 694 Springer-Verlag, 1993

[10] W.M. Gentleman, “Some Complexity Results for Matrix Computations on Parallel Processors”, *Journal of the ACM,* vol. 25, pages 112-115, 1978.

[11] Jonathan M.D. Hill “The AIM is laziness in a data-parallel language” In K. Hammond, and J. T. O'Donnell, editors, *Glasgow Functional Programming workshop*, Springer-Verlag, 1993.

[12] Marco Kesseler, “Reducing Graph Copying Costs - Time to Wrap it Up”. *The First International Symposium on Parallel Symbolic Computation, PASCO ’94,* Hagenberg/Linz, Austria, pages 244-253, World Scientific, 1994.

[13] Marco Kesseler, “Uniqueness and Lazy Graph Copying - Copyright for the Unique”, In *proceedings of the 6th International Workshop on the Implementation of Functional Languages,* University of East Anglia, Norwich, UK, 1994.

[14] H. Kuchen, R. Plasmeijer, and H. Stoltze “Distributed Implementation of a Data Parallel Functional Language” In *Parallel Architectures & Languages Europe, PARLE ’94* , pages 464-477, LNCS 817, Springer Verlag, 1994.

[15] Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen M.C.J.D. van, Plasmeijer M.J., (1991). “Concurrent Clean”, Proceedings of the conference on Parallel Architectures and Languages Europe (PARLE'91). Springer Lecture Notes in Computer Science 505, Vol. II, page 202-219.

[16] Eric Nöcker, “Efficient Parallel Functional Programming - Some Case Studies”. In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages,* Nijmegen, the Netherlands, pages 51-67, September 1993.

[17] Plasmeijer M.J., Eekelen M.C.J.D. van (1993). *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[18] David B. Skillicorn “The Bird-Meertens Formalism as a Parallel Model” In *NATO ARW “Software for Parallel Computation*,” 1992.

[19] Smetsers J.E.W., Nöcker E.G.J.M.H., Van Groningen J.C., Plasmeijer M.J. (1991). “Generating Efficient Code for Lazy Functional Languages”. *Proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture (FPCA’91)*, U.S.A., Springer Lecture Notes on Computer Science.

[20] S. Smetsers, E. Barendsen, M. van Eekelen, R. Plasmeijer (1993). “Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs”. In *Graph Transformations in Computer Science*. Dagstuhl Castle, Germany, Springer LNCS 776, pages 358-379.