"The Four-Stroke Reduction Engine" by C. Clack and S.L.P. Jones from *Conference Record of the 1984 ACM LISP and Functional Programming Languages*, August 6-8, 1984, pages 220-232. Copyright 1986, Association for Computing Machinery, Inc.

The Four-Stroke Reduction Engine

Chris Clack and Simon L Peyton Jones

Department of Computer Science
University College London,
Gover St, London WC1E 6BT, England

Abstract

Functional languages are widely claimed to be amenable to concurrent execution by multiple processors. This paper presents an algorithm for the parallel graph reduction of a functional program. The algorithm supports transparent management of parallel tasks with no explicit communication between processors.

1. Background

A major challenge facing computer science today is the effective exploitation of parallelism. Functional languages offer a powerful lever on the programming of parallel machines, and the most promising model for implementing these languages is graph reduction.

Many current research projects are investigating parallel architectures [Kell85] [Darl81] [Hudak85] [Peyt85]. Parallel hardware can readily be built using conventional technology, but designing the system so that parallelism gives worthwhile gains, and so that harnessing this parallelism is easy for the programmer, represents a greater challenge.

This paper is concerned with the design of a parallel graph reduction algorithm that will transparently administer the scheduling and synchronisation of concurrent tasks in a parallel functional-programming machine. The algorithm that we present can be coded as a finite state machine. This enables us to produce a simple and efficient implementation.

1.1 Functional languages and parallelism

Why not program in a conventional language which supports multiple tasks, such as Ada? In a conventional language, the programmer must explicitly code the program for parallel evaluation. The behaviour of the program will depend on the scheduling of the tasks, and the programmer must ensure that parallel evaluation does not alter the semantics of the program.

The absence of side effects in a functional language means that the execution of one task cannot affect the outcome of the execution of another task. This implies that task administration and synchronisation are inherently easier for a functional language.

Functional programs do not require explicit parallel programming. No extra language constructs are needed to write parallel functional programs, and the result of the program is guaranteed to be independent of task scheduling (though this may have an impact on efficiency).

There should be no need for the programmer to give directions concerning details of scheduling and communication, but the programmer must still design the algorithm with concurrency in mind; we can only get the full benefits of parallelism if the algorithm is coded to give gross parallel structure (e.g. using a divide-and-conquer approach).

An automatic compile-time analysis of the source (called "strictness analysis") will often detect much of the inherent parallelism [Clack85], but this is still a research area and programmer annotations may be used in preference or to give additional hints to the compiler.

The resultant parallel tasks can be managed automatically at run-time, and our aim is that the entire business of administering parallel tasks should be hidden from the programmer.

1.2 Graph reduction

A functional program is a single expression which has a natural representation as a syntax tree. In general, there will be sharing of nodes, and so the syntax tree will be a graph, which may be cyclic.

A functional program may be evaluated by manipulating the syntax graph. The evaluation proceeds by means of simple steps, each of which performs a local transformation to the graph. Each step is called a reduction, and the process is known as "graph reduction" [Turn79]. A reducible expression is often referred to as a redex.

Reductions may take place concurrently, since they cannot interfere with each other, and evaluation is complete when there are no further reducible expressions (normal form).

The curried application of a function "f" to two arguments x and y is represented like this:

```
        @
       / \
      @   y
     / \
    f   x
```

where "@" represents an application cell, containing pointers to function and argument.

Consider the following functional program:

```
LET    f x  = AND x x
IN     f (NOT TRUE)
```

(the LET introduces a definition of the function f, which takes a single argument x. Function application is denoted by juxtaposition, thus (NOT TRUE) denotes the function NOT applied to the argument TRUE).

The figure below shows how it would be evaluated. Notice that after each reduction the root of the redex is overwritten with the result of the reduction.

```
   @          =>        @           =>        @         =>      FALSE
  / \                  / \                   / \
 /   \                /   \                 /   \
f     @              @     \               @     \
     / \            / \     \             / \     \
    /   \          /   \     \           /   \     \
  NOT   TRUE     AND    \ \             AND   \ \
                          @                    FALSE
                         / \
                        /   \
                      NOT   TRUE
```

2. A model for parallel graph reduction

Our proposal for parallel graph reduction has the following features:

(i) The reduction of the graph is performed by the concurrent execution of many **tasks**, each of which has access to any part of the graph.

(ii) A task performs a **sequence of reductions**, performed in **normal order**.

(iii) The purpose of a task is to reduce a particular (sub)graph to a normal form in which there is no top level redex - we call this Weak Head Normal Form (**WHNF**) [Peyt86]. A task is therefore completely defined by a pointer to the root of this subgraph.

(iv) During its execution a task may anticipate that it will require the value of a subgraph. In this case it may create a new task to evaluate the subgraph concurrently. To create a new task, a task descriptor is placed in a task pool. We call this **sparking** a task.

(v) There are a number of **agents**, each of which executes a task. Typically an agent will be implemented by a physical processor, although one processor may be timesliced to implement more than one agent.

(vi) There are one or more **task pools**, which may be accessed by the agents. A task pool holds a number of tasks which are ready for execution. These tasks may have been created by other tasks, or they may be tasks that have been temporarily suspended and subsequently resumed. Unemployed agents will send requests to the task pool for work to be done.

2.1 Parallelism

One of the major issues that must be faced by any parallel implementation is the generation of new tasks. When should a new task be sparked? There are two broad approaches:

(i) Spark a new task to evaluate a sub-graph when it is **certain** that the sub-graph will eventually be evaluated (**conservative parallelism**). This ensures that all tasks are doing useful work.

(ii) Spark a new task to evaluate a sub-graph when it is **possible** that the sub-graph will eventually be evaluated (**speculative parallelism**). This offers maximum opportunities for parallelism.

The danger of speculative parallelism is that machine resources may be consumed evaluating pieces of graph which will eventually be discarded. Speculative tasks need particularly complicated management, since vital tasks must be distinguished from non-vital tasks, and since tasks that are no longer needed must be garbage-collected [Peyt86]. We therefore assume that only conservative parallelism will be exploited.

2.1.1 Creating new tasks in a conservative regime

Suppose that a function F was known to (eventually) require the value of its argument (that is, it is "strict" in that argument). Then a conservative scheduler could safely spark concurrent evaluation of its argument whenever F is applied.

In the case of primitives (functions such as "+" that are built in to the implementation) it is easy to know which arguments will be required. However, in the case of user-defined functions matters are not so clear. Certainly we do not want to work it out at run-time, so we **annotate the function** with information describing which arguments it is sure to need.

These annotations may either be introduced by the programmer or added by a compiler pass which uses strictness analysis to deduce which arguments the function is sure to need

[Clack85]. Annotations should be added with care; if the evaluation semantics are altered, non-termination may result.

Merely annotating functions in this way is not sufficient. Some functions may be strict in a particular argument for one application, but not for another application. Consider the definition of a function F as:

```
F x y  =  y 3 x
```

Clearly, F is not strict in x, because the function argument y may not be strict in its second parameter. Now suppose that elsewhere there occurs the expression

```
...(F E +)...
```

where E is some complex expression. In this application of F the second argument is +, so E will certainly be evaluated subsequently. In this particular context, therefore, we can safely spark evaluation of E, and we can indicate this fact by annotating the **application node**.

At first it appears that the second sort of annotation subsumes the first. However, there are situations in which each is uniquely appropriate. The two forms of annotation are complementary, and neither can be omitted without loss [Peyt86] [Hank85] [Burn85]. For example, consider the expression

(IF E_1 f g) E_2

where f is strict in its argument but g is not. We cannot know until run-time whether f or g will be applied to E_2, so we cannot annotate the application node. We can, however, annotate the function f so that E_2 will be evaluated in parallel if f is eventually applied to it.

To summarise, we derive parallelism from

(a) **primitives** - we use innate knowledge about the strictness of primitives, and we require that the implementation has some method for annotating primitive functions with this strictness information.

(b) **user functions** - we use strictness analysis to determine this information, and we require that the implementation has some method to annotate user functions.

(c) **applications** - we use strictness analysis to determine those situations where an application is strict: we require a method to annotate an application node.

2.2 Colouring the graph to synchronise tasks

During the course of evaluation, two tasks may attempt to evaluate a common subgraph simultaneously. Notice that no mutual exclusion is required (since they will both arrive at the same result). In practice, however, it is highly desirable that only one task should evaluate a piece of graph at a time, to avoid duplicated work. The main aim of our algorithm is to achieve this mutual exclusion painlessly.

The idea behind the algorithm is that that as a task traverses the graph it "paints" the nodes that it is working on. After working on a section of graph, a task will "unpaint" the nodes that are no longer being used.

If one task attempts to access a node that has been painted by another task, the intruding task will be **blocked** until the required cell has been unpainted. Thus, if two tasks share the same subgraph, there will be no duplicated effort.

Consider, for example, the following program:

```
LET f   = g 6
    g x = + (-x)
IN
    + (f 3) (f 1)
```

We might spark two tasks to evaluate the (f 3) and (f 1) sub-graphs, which share a common sub-graph f:

```
            @
           / \
          /   \
         /     \
        @       \
       / \       \
      +   @#      @#
         / \     / \
        /   3   /   1
        |______/
      f:@
       / \
      g   6
```

The "+" might spark the nodes marked "#" thus creating two new tasks to evaluate the arguments to the "+". The first of these tasks to try to evaluate the node labelled f will paint it (let us suppose it is the left hand task in the picture). When the second task tries to evaluate this node it will be blocked. Meanwhile the first task will reduce the f node to WHNF by applying g to 6, and overwriting the node with the result (+ (-6)). Then, having evaluated the arguments (-6 and 3) it will add them, remove the paint from the f node as it pops the node from its stack, and overwrite the left node marked "#" with the result (-3). Now the second task can proceed, so it will access the f node, where it will see the (+ (-6)). It will never know that there was once a (g 6) redex there.

We intend that parallel tasks should be blocked and later resumed in a way that is entirely transparent to the agents, and at low overhead to the implementation. There is no explicit communication between agents or between tasks - synchronisation between tasks is mediated entirely through the graph. The result of a reduction is communicated to the graph as a single, indivisible operation, (that is, the overwriting of the root node of the redex) and the reduction appears to all other tasks to take place instantaneously. The graph never appears in an intermediate state, and the interlocking of agents becomes totally hidden from the programmer.

2.2.1 Blocking and resumption

As mentioned above, for efficiency reasons we would like it to be possible for one task to be blocked by another. We will now consider the blocking mechanism in more detail, with the help of an example. Suppose a task is evaluating the expression

$$(+\ E_1\ E_2)$$

where E_1 and E_2 are complicated expressions. Now, we know that "+" will need the values of both of its arguments, so the task can spark a child to evaluate one argument (say, E_1) and evaluate the other argument (E_2) itself.

When it has finished evaluating E_2, the parent task will look again at E_1, to make sure that it has been evaluated. E_1 can now be in one of three possible states:

(1) It has not been evaluated yet (perhaps because the system has been busy executing other tasks). In this case, the parent task should proceed to evaluate E_1 itself. Eventually, another agent will try to evaluate E_1 (remember, E_1 was sparked, so the child task is in the task pool) but this child task will die immediately upon discovering that E_1 has already been evaluated.

(2) It has been evaluated already. In this case, the parent task can immediately proceed to apply the function "+" to the two evaluated arguments.

(3) It is still in the process of being evaluating. Now the root node of E_1 will have been "painted" by the child task and the parent task will be blocked until E_1 has been evaluated.

What should happen to a task when it is blocked? There are two main alternatives:

(a) It could simply be returned to the pool of tasks awaiting execution. In due course an unemployed agent looking for work will resume execution of the task. It will very soon encounter the node that blocked the task before. If this node is still painted, then the task is again blocked, and returned to the pool, otherwise it can continue to execute normally.

(b) It could somehow be suspended, so that it is not considered for execution by unemployed agents, and be resumed when the node which blocked it has its paint removed. Reawakening the task would consist of putting it in the pool of tasks awaiting execution.

The first method has the advantage of simplicity, but it is rather inefficient, since repeated attempts are made to execute a task which is still blocked for the same reason.

In order to implement the second method we would somehow have to attach the blocked task to the painted node. Then when the paint is taken off the node, the blocked task can be put back in the task pool as a **resumed** task.

2.3 The task pool

A new task is created by adding a task descriptor to the task pool. What happens when more and more tasks are sparked? If tasks are being added to the task pool faster than they are being taken out, then the task pool may run out of space.

As we showed in section 2.2.1, after a parent sparks a child task to evaluate a strict argument it **always** subsequently returns to evaluate the argument itself. Hence, all **sparked** tasks in the task pool are entirely disposable (even the program root, as long as the I/O mechanism is informed, or automatically tries again later).

Once a task has started, it will block its parent (see section 2.2.1). Therefore, it is **vitally important** that we do not throw away **resumed** tasks.

So a good strategy for administering the size of the task pool would seem to be:

(i) set a limit somewhat below the real limit of the task pool.
(ii) once the task pool has reached that size, **ignore** all sparks until the task pool shrinks again, but
(iii) be careful **not** to **ignore** any resumptions of tasks.

2.3.1 Task scheduling

When an unemployed agent sends a request to the task pool, which task should be returned out of all those in the pool? A decision must be made on how best to schedule the available tasks.

Although the details of scheduling tasks from the task pool are not yet well understood, we can offer the major comfort that with conservative parallelism **the scheduling of tasks is guaranteed not to affect the result.**

However, scheduling may have a considerable effect on efficiency. Most tasks will cause the graph to grow before it can shrink again, so a bad choice of scheduling algorithm could mean that many tasks will expand a subgraph and then be blocked before the shrinking occurs. This might result in the implementation running out of memory, so that the computation grinds to a halt.

Simple strategies like last-in-first-out (LIFO), or first-in-first-out (FIFO) may give acceptable results and merit some investigation. For instance, a FIFO strategy corresponds to breadth-first evaluation of the graph, and may therefore result in the sparking of more parallel tasks than a depth-first LIFO strategy. Indeed, switching between the two strategies could give useful dynamic control of the production and consumption of tasks.

Another strategy may be always to schedule **resumed** tasks first, since we can be sure that they will be doing useful work (whereas a sparked task may find that its subgraph has already been evaluated), and since they could be blocking other tasks.

3. An implementation of parallel graph reduction

The four-stroke reduction engine assumes that we are implementing graph reduction using a **global heap** of cells, each with three fields; **tag, head** and **tail.** The head and tail fields may each contain either a data object or a pointer to another cell. The tag field is used to identify the type of cell - for instance, an application cell or perhaps a "cons" cell. The hardware should also support the **indivisible update** of a heap cell.

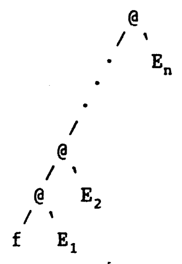
We present an algorithm that is applicable to all forms of graph reduction. We describe how the scheme works both for primitives and for user functions. Our preferred representation for a user function is the supercombinator, although the algorithm is adaptable to other representations.

3.1 Task execution and the 2-stroke cycle

We assume that reductions are carried out in **normal order,** which specifies that **the leftmost outermost redex should be reduced first.** The expression to be evaluated can only be of the form

$$f\ E_1\ E_2\ \ldots\ E_n$$

where f is a data object (such as TRUE), built-in function (such as AND), or user-defined function, and there are zero or more arguments E_i, which denote arbitrarily large expressions. The graph of this expression looks like this:



Suppose that f takes m arguments; the leftmost outermost redex will be the application of f to its arguments E_1, E_2,...E_m. Before the graph reduction machine can reduce this redex it must find f: it goes down the left branch of each application node from the root until it finds a non-application node.

This left-branching chain of application nodes is called the **spine** of the expression.

It is therefore rather easy to find the next redex to reduce. We descend the spine, painting the spine nodes as we go, until we find a function. Then, based on the function we find, we go back up the spine, collecting the arguments E_i and unpainting the spine nodes as we go, to find the root of the redex. Now the function and all its arguments are available and the reduction may be carried out; the result overwrites the root of the redex.

In order to minimise the overheads of task-switching, we prefer not to remember the argument stack in the task state descriptor. Thus, collecting the arguments must wait until we go back up the spine, since it is only when ascending that we can guarantee never to be blocked.

After completing a reduction, the task again descends the spine and the process repeats. When the task finds that a function does not have enough arguments on the spine, or the expression is now a data object, then the subgraph has reached WHNF and the task dies.

This "down-up" cycle is somewhat reminiscent of a piston engine, and we call it the **2-stroke cycle.** In fact, we only use this method to reduce user functions and primitives with no strict arguments: for all other primitives we need to use four strokes, as we discuss later.

3.1.1 Representation of a task

When a task is not being executed by an agent it must be represented in some way in store. The representation of a task must contain all the information required to continue executing the task from the point at which it was last blocked. In conventional multi-tasking operating systems this representation is often called a Task Control Block, and contains information such as the task's stack pointer, its program counter and the state of the task's registers.

By contrast, in our parallel reduction model a task could be represented completely by a single pointer to the root of the subgraph it is evaluating. **The complete state of a partially completed task is held in the graph,** so that a pointer to the root of its subgraph suffices to represent a task at **any stage** in its life (not only when it is newly sparked).

At any moment an agent can stop performing reductions on a task, put its root pointer back into the task pool, and begin executing another task.

The only trouble with this representation of a task is that if a task is blocked and subsequently resumed the agent has to descend the spine of the subgraph from the root. This is due to the fact that no information is saved about the state of a task - the agent must look in the graph to see how far the task got before being blocked.

We may choose to save more state information in each task descriptor (held in the task pool), and a technique called **pointer reversal** gives us a way to save enough state to allow the task to continue from where it was suspended, i.e. **without** having to descend the subgraph from the root again. Furthermore, pointer reversal allows us to save this much state using just **two pointers!**

3.1.2 Pointer reversal

An evaluator can descend the spine of an expression without using a stack by reversing pointers in the spine as it goes. This pointer-reversing technique is described by Stoye et al [Stoye84].

For example, to descend the spine of (+ E_1 E_2), we use two pointers "B" (for Backward) and "F" (for Forward). B and F point to two adjacent nodes on the spine; spine nodes below F have undisturbed pointers pointing down the spine to the next node, whereas spine nodes above B have reversed pointers that point up the spine to the previous node. To descend the spine, we read the function pointer in the "F" cell - this becomes the new "F", and we overwrite the cell's function pointer with the old B. The old "F" becomes the new "B":

```
B: ↑                           ↑                         ↑
F:-----> @              B:---> @                         @
        / \                     \                       / \
       /   \                     \                     /   \
      ↓     ↓                     ↓                   ↑     ↓
      @     E2          F:-> @    E2       B:-----> @      E2
     / \                    / \                       \
    /   \                  /   \                       \
   ↓     ↓                ↓     ↓                       ↓
   +     E1               +     E1         F:-> +       E1
```

The act of descending the spine using pointer-reversal is sometimes called **unwinding** the spine. Conversely, ascending the spine is called **rewinding**.

At first it appears that this is totally unworkable in a parallel machine, since the pointer-reversed graph is in a "peculiar state" which will be incomprehensible to other tasks. However, pointer reversal only reverses the painted nodes. No other task will look inside a pointer-reversed node, and it is therefore safe to use this technique.

The complete state of a task can now be represented by two pointers, the F and B pointers. When a blocked task is resumed, the F and B pointers are already pointing to the area of the graph which is of interest.

In a sequential implementation, pointer-reversal is not as efficient as using a stack, since the pointers have to be re-reversed when rewinding the spine. However, in a parallel implementation which uses the graph-colouring scheme to synchronise tasks, nodes have to be "unpainted" as the spine is ascended, and there will probably be little extra cost to re-reverse the pointers as well.

We conclude that pointer reversal may save repeatedly unwinding the spine each time a task is blocked, and adds very little to the overheads of task switching.

3.1.3 Sparking tasks

Recall from section 2.1.1 that the parallelism information is conveyed to the evaluating agents by two forms of annotated graph nodes, namely annotated functions and annotated application nodes. We implement the second form of annotation by means of the tag field in a cell - special kinds of application tags show whether the application node is strict in its argument.

Annotations to application cells can be discovered and sparked on the way down down the spine, whereas annotations to primitives and user functions cannot be discovered until we reach the bottom of the spine, and so strict arguments must be sparked on the way up the spine. Of course, we must be careful not to spark an argument twice, which would be wasteful. This is easy to achieve, by altering the tag of a spine node when its argument is sparked.

3.2 Task synchronisation

We saw above how task synchronisation could be achieved by "colouring" the graph. Now we are in a position to describe our implementation of such a synchronisation scheme:

3.2.1 How to colour the graph

We implement the "colouring" idea with special values of the tag attached to each node. At each stage of reduction, a task will check the cell tag it wishes to reference, and the value of this tag will determine the future computation. We assume that memory is cheap enough that the extra memory space required is not profligate. The scheme need not be especially wasteful of time, since we can design intelligent memory units to implement special read-modify-write instructions that depend on the value of the tag of the cell (thus allowing us to access the cell, check to see if it is already painted, then paint it if it wasn't before - all in one indivisible operation).

3.2.2 The blocking mechanism

How do we achieve the blocking of multiple tasks on one painted node? We could achieve this by adding an extra field to every application node. This points to a list of tasks which should be reawakened when the paint on the node is removed. This is the approach taken by the ALICE machine [Darl81].

Attaching an extra field to every application node seems rather wasteful, since most of them will not have any tasks blocked on them. Our proposal is to overwrite the head of the application node with a pointer to a list of blocked tasks (we call this a **task queue**), and remember the old head in the tail of the list. Some mechanism would then be required to indicate that there were blocked tasks queued up on a painted node - for instance, yet another special value for the cell tag.

It is only the unwind and rewind operations that affect the blocking and resumption mechanisms. When a task unwinds it may be blocked and will be added to a task queue. When the blocking task finally rewinds back up the spine, it will come across a cell with an attached task queue and all blocked tasks in the queue will now be added to the task pool.

One advantage of the mechanism described here is that it is sufficiently simple and low-level that it can be implemented in hardware (e.g. VLSI), and that as such it can form part of an **Intelligent Memory Unit** - if the agents talk to memory via high-level operations there is **absolutely no need for them to know about this blocking mechanism!** If an agent is executing a task that is blocked, it only needs to know that the task cannot continue, and that the task pool should be consulted for more work.

3.3 4-stroke reduction

The four-stroke reduction engine is named after the fashion in which it reduces primitives applied to their arguments. In contrast with user functions (and primitives with no strict arguments), a primitive application like (+ E_1 E_2) cannot be reduced any further until the strict arguments E_1 and E_2 have been evaluated. Therefore, a task must **ensure** that all strict arguments have been evaluated (either by itself, or by another task). To be more specific, any primitive which has at least one strict argument will need four "strokes" (instead of two) to reduce:

(1) **inlet stroke**
unwind the spine to determine which primitive is being applied, sparking any strict applications on the way.

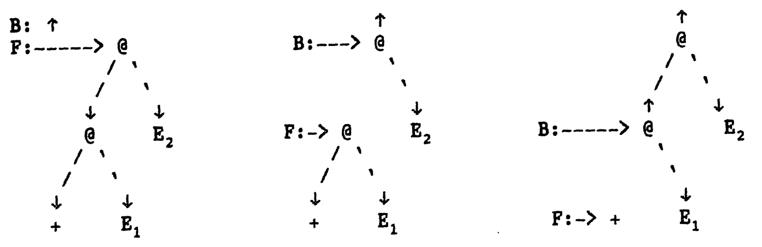
(2) **compression stroke**
rewind to the topmost strict argument, sparking strict arguments on the way

(3) **power stroke**
unwind the spine again, evaluating strict arguments one at a time on the way down (and reversing pointers as before).

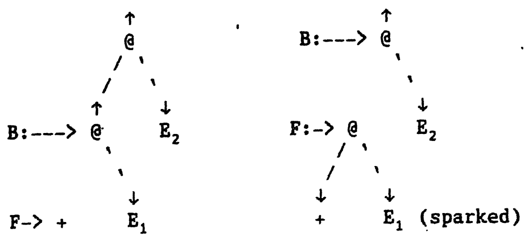
(4) **exhaust stroke**
finally rewind to the root of the redex, collecting the now-evaluated arguments, perform the reduction and overwrite the root of the redex with the result.

We illustrate four-stroke reduction using the example "+ E_1 E_2":

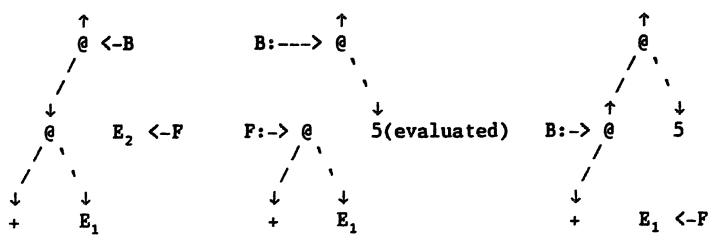
(i) Inlet Stroke



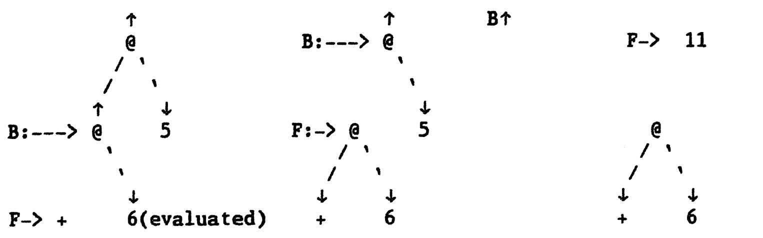
(ii) Compression Stroke



(iii) Power Stroke



(iv) Exhaust Stroke



3.4 Optimisations

A disadvantage of the blocking scheme outlined above is that it risks unnecessary serialisation.

Consider the case of a shared subgraph that is already in WHNF, as might be the case with a commonly used partial application. As one task unwinds into the subgraph it paints the top node, thus blocking any other tasks from unwinding into it. **But if the subgraph is already in WHNF, there is no point in making other tasks block.** It is perfectly safe to allow any number of tasks simultaneous access to the subgraph!

This is a specific instance of a general rule: once a sub-graph is in WHNF it will never be altered, so it is quite safe for many tasks to have (read only) access to it.

Our implementation allows many tasks to access a subgraph in WHNF, by requiring that tasks do not use pointer reversal when traversing WHNF subgraphs (they can never be blocked when doing so).

4. Finite State Machine

Perhaps the most satisfying feature of our algorithm is that it can be represented as a finite state machine. This is possible because of the way that we use graph-colouring in order to synchronise tasks.

- The execution of each task is governed by a finite state machine.
- Each agent executes a finite state machine (physically, the same code will run on all processors in a multiprocessor machine).
- The tasks can each be in any one of a fixed, small number of states.
- The first action of an agent at each stage of the finite state machine will be to access a cell in the shared graph. Each agent holds the B and F pointers that represent the current task. If the current state is one that descends the spine, then the cell accessed will be the F cell. In an ascending state, the B cell will be accessed.
- The value of the tag of the accessed cell will determine the subsequent action (such as sparking the tail of the cell just read, and the painting or unpainting of a cell) and will specify the next state transition (often this will be to stay in the same state).
- In a real implementation, knowledge of actions and state transitions can be incorporated in the intelligent memory. Thus, with a knowledge of the particular high-level memory operation being requested by a processing element, and of the value of the tag of the cell being accessed, the intelligent memory can itself take care of administrative details such as painting and unpainting.

It is important to realise that the value of both tags is required to determine the next state transition; the value of one of the tags is implied by the current state, and the value of the other tag must be determined by reading from the graph.

How do we define the different states? They are derived from a combination of

(i) the direction of pointer-reversal (are we unwinding or rewinding?)
(ii) the value of the "known" tag (B if going down the graph: F if going up).
(iii) the number of strict arguments still to be collected (compression stroke only), and
(iv) the total number of arguments left to be rewound past (exhaust stroke only).

What happens when a task is resumed? Since a task can only be blocked during the inlet or power strokes, then (iii) and (iv) above do not apply. It also follows that the task must be unwinding. The only condition left is (ii), so directly after resuming a task the first thing that an agent must do to establish the state of the task is to read the value of the B tag.

Our finite state machine currently has a total of 12 states (including the optimisation of using a stack to traverse shared subgraphs in WHNF), and there are 22 different kinds of tag - 11 of which are application nodes in various guises!

5. Conclusion and project status

This algorithm has been implemented as part of the Alvey-funded GRIP project. GRIP (Graph Reduction In Parallel) is a parallel machine based on about 120 processing elements, a fast asynchronous bus, and about 30 intelligent memory units. The finite state machine has now been coded (in C) and we have a running simulator of a parallel graph reduction machine. It is intended that the code executed by an agent should be ported without change onto the actual processing elements in GRIP. Our experience has shown that implementing parallel graph reduction is by no means trivial, and we could not have progressed as far if the algorithm had not been represented as a finite state machine. As evidence of this, we found that debugging the simulator has consisted almost entirely of remedying typing errors rather than changing the algorithm. The simulator is now producing results which will help us with the final stages of the hardware design.

References

[Burn85] Burn G, Hankin CL and Abramsky S, "Strictness analysis of higher order functions", Science of Computer Programming (to appear); also DoC 85/6, Dept Comp Sci, Imperial College London, April 1985.

[Clack85] Clack CD and Peyton Jones SL, "Strictness analysis - a practical approach", Functional Programming Languages and Computer Architecture, ed Jouannaud, LNCS 201, Springer Verlag, pp35-49, August 1985.

[Darl81] Darlington J, Reeve M, "ALICE - a multiprocessor reduction machine for the parallel evaluation of applicative languages", Proc ACM Conf on Functional Programming Languages and Computer Architecture, New Hampshire, pp65-75, Oct 1981.

[Hank85] Hankin CL, Burn GL and Peyton-Jones SL, "An approach to safe parallel combinator reduction", Dept Comp Sci, Imperial College, Oct 1985.

[Hudak85] Hudak P, "Functional programming on multiprocessor architectures - research in progress", Dept Comp Sci, Yale University, November 1985.

[Kell85] Keller RM, "Rediflow architecture prospectus", UUCS-85-105, Dept Comp Sci, University of Utah, Aug 1985.

[Peyt85] Peyton Jones SL, Clack CD, Salkild J and Hardie M, "GRIP - a parallel graph reduction machine", Dept Comp Sci, University College London, November 1985.

[Peyt86] Peyton Jones SL, "Implementation of Functional Programming Languages", Prentice Hall (to be published), 1986.

[Stoye84] Stoye WR, Clarke TJW, Norman AC, "Some practical methods for rapid combinator reduction", ACM Symposium on Lisp and Functional Programming, Austin, pp159-166, August 1984.

[Turn79] Turner DA, "A new implementation technique for applicative languages", Software Practice and Experience 9, pp31-49, 1979.