Proceedings of the Third International Workshop on Implementations of Functional Languages on Parallel Architectures (1991). Technical Report Series, CSTR 91 91-07, June 1991, University of Southampton, United Kingdom.

IMPLEMENTING THE PABC MACHINE ON TRANSPUTERS

Marco Kesseler
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
E-mail: marcok@cs.kun.nl
June 1991

Abstract

In this paper an efficient implementation of the PABC machine on transputers is presented. The PABC machine is an abstract machine that is used to implement Concurrent Clean, an experimental lazy functional language. For a single transputer this means an 'unbounded' graph store has to be implemented and a set of reducers, each having a couple of stacks. In a network of transputers each reducer should be able to send graphs to other transputers, start new reducers on these remote graphs and to get reduced graphs back. At the moment the PABC machine runs on a single transputer only. Efficient and reliable implementation on a network of transputers has started.

Introduction

Concurrent Clean is an experimental functional language suited for evaluation on parallel machine architectures (Plasmeijer & van Eekelen (1989)) It is based on Parallel Functional Graph Rewriting Systems (PFGRS's) such that the notion of sharing of computation and the notion of processes are explicitly present in the language instead of being an option for the implementation. Concurrent Clean is generally used in two ways: as a programming language and as an intermediate language (i.e. for other functional languages). It is not translated directly to concrete machine code but to the machine code of an abstract machine called the Parallel ABC-machine (PABC machine). This abstract code is either interpreted by a software PABC machine or compiled to concrete machine code of the host machine.

At the moment a compiler (Smetsers (1989)) is available that translates Concurrent Clean into PABC code. Also a PABC simulator (Nöcker (1989)) has been built, such that PABC programs can be interpreted. It includes facilities for tracing, debugging and statistics. Furthermore a codegenerator (van Groningen (1990)) has been implemented that translates PABC code into very efficient sequential assembly code (i.e. concurrent processes are not supported) for the Motorola 68000 family.

In this paper an implementation of the PABC machine for a transputer based architecture is presented. First a short description of the PABC machine will be given. Next I'll give a short explanation of the kind of transputer hardware that is available at the university of Nijmegen. I won't go into details of the transputer chip itself. The main part of this paper is meant to clarify the way the PABC machine has been implemented until now. Also some of the things that yet have to be done will be discussed.

At the moment of this writing a prototype for the interleaved execution of a set of PABC processes on a single transputer has been implemented (except for some small things that yet have to be implemented and/or tested). A lot of effort has been put in optimisations for sequential processing. Although this can be improved still further, we have recently focussed our interest at the implementation of real parallel execution of the PABC machine on a network of transputers. This will be the main topic of interest in the very near future.

A number of problems that haven't been solved yet will be left unattended. It should also be noted that the PABC machine itself is still subject to changes. Despite of this some results about the sequential execution of the PABC on the transputer can already be given.

The PABC machine

The PABC machine consists of a set of processors that are interconnected by some communication mechanism. Each processor has a local graph store and a set of reducers. Each graph store contains a (probably different) part of the graph that must be reduced.

Each reducer has three stacks[1]:

- the argument stack (A-stack). This stack holds references to nodes in the graph store.
- the basic value stack (B-stack). This stack holds basic values, such as integers, booleans and characters.
- the control stack (C-stack). This stack holds return addresses.

Most PABC instruction operate on these stacks. Some instructions are designed to conveniently express graph rewriting, others have more resemblance with instructions of traditional machine architectures. Opposed to the graph-rewriting instructions these run relatively fast on an actual machine. The modification of a graph leads to complex memory management problems, involving garbage collection of previously allocated memory that is no longer in use.

Instructions for parallelism comprise instructions to start up new reducers (either on the same processor or on a remote one) and instructions to copy graphs from one processor to the other.

Reducers on the same processor run interleaved and share the graph in the graph store. To keep different reducers from reducing the same part of the graph a locking mechanism is used. When a reducer starts to reduce a free node of the graph it locks the node and goes on reducing. If the node is locked the reducer suspends itself on that particular node. As soon as a locked node is updated by the reducer that is reducing it, all reducers that are suspended on this node are released.

When part of the graph must be reduced on a different processor, it is sent to it and a new reducer is started on the other processor. If a reducer needs (part of) a reduced graph that resides on another processor, it sends a request for it and suspends itself. As soon as the result is copied to the requesting reducer it is awakened.

The Nijmegen SuperCluster

At Nijmegen a Parsytec multi-transputer system is available. This is a member of the so-called supercluster family. These machines consist of a number of hardware modules that can be extended and adjusted according to the specific requirements of the user. A number of superclusters can be cascaded to form larger transputer systems.

---

[1] For the sake of completeness: each reducer actually has four stacks. In addition to the a-, b- and c-stack there's a so-called continuation stack (n-stack). This stack holds references to graphs that the reducer should reduce after it has finished reducing the current one. Although this stack is available it isn't used seriously at the moment.

The NSC currently consists of 32 T800 transputers, each having 4 Mbyte of RAM and a clock frequency of 25 MHz. At the end of 1991 the system will be extended to the total of 64 transputers.

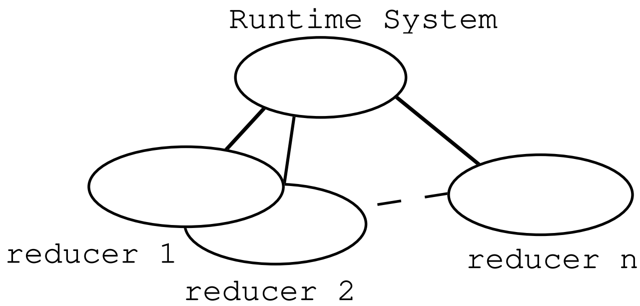
The transputers in the supercluster are interconnected by a number of Network Configuration Units (NCU's). These make it possible to build different network topologies with the transputers available.

Process structure

In this section a description will be given of the process structure that is used to implement the PABC machine on a single processor. The only processes available in the PABC machine are reducers. For an actual implementation some additional processes are needed that perform several services needed by reducers.

The transputer supports concurrent processing via a hardware scheduler which performs round-robin scheduling at two priority levels (i.e. only low priority processes are time-sliced). This mechanism is very suited to implement the interleaved execution of a set of reducers on a single processor: every reducer is implemented as a low priority transputer process.

Additional to these reducer-processes a runtime system is resident. This runtime system is currently implemented as a single low priority process. It is used for one purpose only, i.e. to perform I/O. Whenever a reducer wishes to perform I/O, it sends a message to the runtime system, telling it what to do. In future versions the runtime system will also be used to route messages between transputers. For this purpose it will be split into several processes.

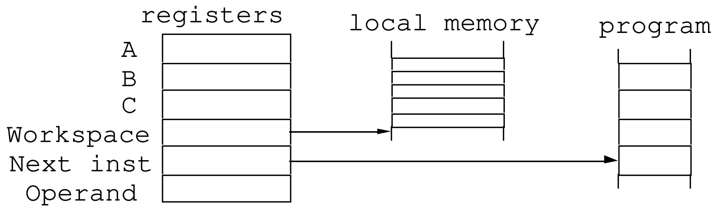


The runtime system is written in ParC. This is basically normal C with some extensions for parallelism. The code of the reducers is generated by a codegenerator that translates PABC code into transputer assembly. A runnable program is obtained by linking the runtime system to the generated code.

Implementation of stacks and the graph store

Each PABC reducer has three stacks. Efficient implementation of operations on these stacks is essential. Furthermore it should be possible create nodes in the heap very fast. These things can be implemented fairly easy on machines that have a reasonable amount of registers, e.g. registers can be used to point to the top of each stack and to the first free byte in the heap. For the transputer this approach isn't possible.

Transputer registers



The transputer has six registers (see picture above), of which three are ‘general purpose’ registers. However, these three registers (called Areg, Breg and Creg) cannot be used to store values over a longer period of time, i.e. when a context switch occurs the values in them will not be saved and must be considered lost afterwards. Furthermore: Areg, Breg and Creg are used as a kind of evaluation stack by the transputer. In fact almost every transputer instruction uses them (e.g. for address calculations). The transputer implements this in such a way that it is not possible to leave a stack (heap) pointer in any of these registers.

Let us now consider the other registers. First of all there’s the workspace pointer. It can be made to point to any word-aligned piece of memory. It is saved upon context switches and memory locations in the neighbourhood of it can be accessed easily. It could indeed be used to implement a single stack (e.g. for languages like ‘C’). However, multiple stacks cannot be implemented this way.

The remaining two registers act as instruction pointer and as a so-called operand register. They are both used by the transputer hardware for very special purposes and cannot be used for anything else.

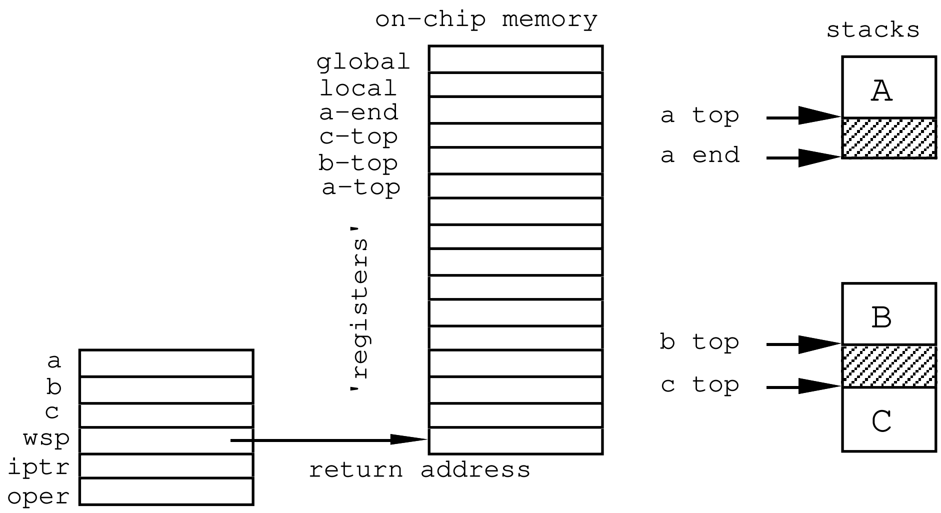
Multiple stacks versus single stack

The transputer hardware made us reflect on the PABC machine. One of the most important reasons the PABC machine has several stacks is that this enables the garbage collector to distinguish node addresses from other data easily. Furthermore: the abstract machine code is far more elegant and flexible if several stacks are used instead of just one. We were not very willing to give that up, just because of the transputer architecture.

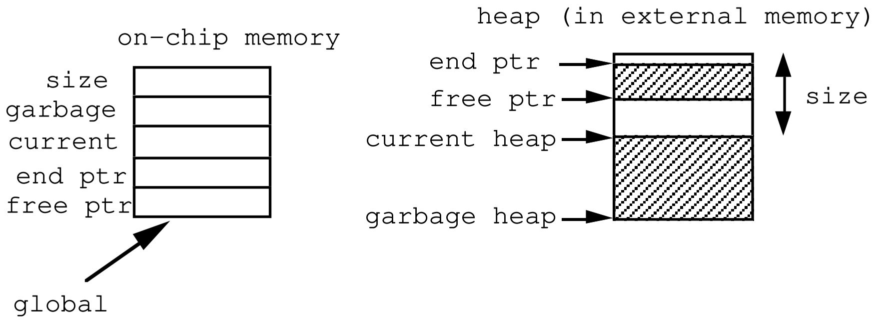
So perhaps it would be possible to use the PABC machine anyway, but let the codegenerator combine the three abstract stacks into one actual stack. This would introduce stack frames and/or the tagging of stack entries. Thinking of the overhead this would bring to access stack entries we decided to use a different approach which is quite straightforward at first sight, but incorporates possibilities for several optimisations.

Locals for registers

Each transputer has some on-chip static RAM (4 Kbytes for the T800). Its access time is around three times smaller than for external memory (depending on the speed of the external memory). We decided to use this on-chip memory to give each reducer a kind of private register set. We’ll refer to this set as ‘the locals of a reducer’ from now on. The workspace pointer of each reducer is made to point to the start of these locals (see picture below).



Information about the heap is shared by all reducers. For this purpose they all have a pointer (the 'global' field in the picture above) to a shared structure that contains the following information:



Stack size

One of the decisions that had to be made was how big each stack should be. It isn't realistic to allocate large stacks for each reducer as this would waste too much memory. Allocating small stacks would prohibit certain programs to be run. Both is highly undesirable. The solution chosen is this: at first every reducer gets small stacks. Whenever a stack turns out to be too small a larger one is allocated and the contents of the old stack are copied to the new one. The old stack becomes garbage.

It is important that reallocation of stacks doesn't occur too often as this will seriously decrease performance. In practice this means that a stack must be increased by a sufficiently large amount each time it is reallocated. On the other hand this amount may not be too large as this would waste too much memory. At the moment the size of the new stack is derived from the size of the old one and the number of entries that are needed by the reducer that is reallocating the stack:

```
new_size = old_size + old_size/10 + constant + entries_needed
```

Basically the stack is increased by 10% each time it is reallocated. The constant takes care the new size isn't too small when the old stack is small. The term 'entries_needed' ensures that the reducer that is reallocating the stack has enough entries available afterwards.

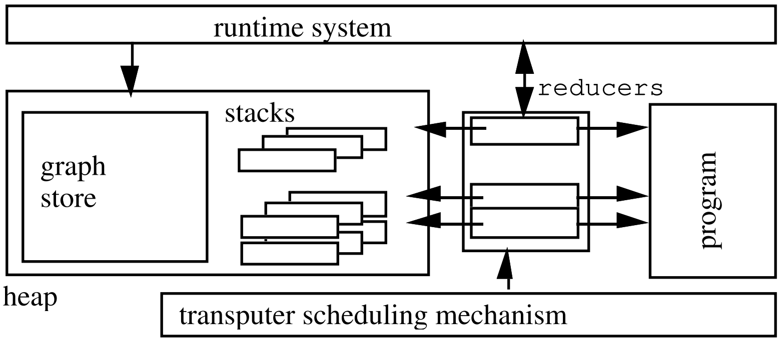
This seems to work quite well. The actual reallocation of stacks is hardly noticeable. Despite this, some (constant) overhead is imposed on each program because stacks should be checked when something is pushed upon it. Compared to programs that don't check stacks the decrease in speed is noticeable, but not too serious.

At the moment the size of a stack is never decreased. This can easily be implemented by a garbage collector: whenever it detects a stack that has few entries in use compared to its size it may decide to shrink the stack. One should be careful not to increase and decrease the size of a stack too often. It remains to be seen what strategy should be used to accomplish this.

We've chosen to place all stacks in the heap. In this way is prevented a division has to be made between the amount of memory available for stacks and the amount of memory available for graphs. Without special precautions such a division would mean that a program would fail if either of the two spaces ran out.

global overview

The structures presented above lead to the following overview:



Code optimisations

The Concurrent Clean compiler uses several techniques to produce efficient PABC code. An important one is the use of an automatic strictness analyzer. Where possible only stacks are used and graph rewriting is avoided. The programmer may also help the compiler to produce efficient code by inserting some program annotations at the appropriate places (e.g. strictness annotations or annotations for load distribution).

The codegenerator divides the PABC code into basic blocks and generates optimised code for each block. Only the last instruction of a basic block may influence the flow of control (instructions that may cause a context switch are regarded as being instructions that may influence the flow of control). This means that once a basic block is entered at runtime, control will inevitably arrive at the last instruction of that basic block (without interruptions), which will pass control to the next basic block to be executed, etc.

Within a basic block the codegenerator tries to maximise the use of the registers and locals, while minimising stack access. Locals 1 - 9 are used to store intermediate results within a basic block and for passing 'parameters' to other basic blocks. In this way access to stacks is avoided, as this costs an extra indirection. The exact way that code is optimised will not be discussed here. For this I refer to (Smetsers et al. (1991)).

Garbage collection

Implementation of garbage collection on a network of transputers is not an easy task. First we'll focus on local garbage collection, i.e. the way garbage collections can be performed on a single processor system. The methods described are in fact the ones that are used for the current implementation. Next, we'll take a look at some problems that arise from the need to implement global garbage collection on a system that consists of multiple processors that each have some private memory.

Local garbage collection

Whenever a reducer creates a node, or when it reallocates one of its stacks it needs memory. As long as enough memory is available in the (local) heap, memory can be reserved by increasing the pointer to the first free byte by the number of bytes needed. A copy of the old pointer can then be used as a pointer to the newly reserved memory.

If not enough memory is available, the garbage collector should be called. This will remove garbage and compact the heap. Note that not only garbage nodes must be removed, but garbage stacks as well. After the garbage collection has succeeded the reducer may try again to reserve the memory it needs.

At this moment a copying garbage collector is used for the transputer implementation. This turns out to be a good solution as long as the amount of garbage is relatively large. To overcome the disadvantage that only one half of the available memory can be used we are planning to switch to a compacting mark-sweep garbage collector as soon as the amount of garbage becomes less than a certain threshold. This threshold depends on the relative speed of the mark-sweep collector compared to the copying collector.

(To give an indication about the relative speed of both garbage collectors: for the 680x0 processor a mark-sweep garbage collector turned out to be about 3 times slower than a copying garbage collector. This has lead to the conclusion that a copying garbage collector should only be used when less than 25% of the total heap is non-garbage.)

global garbage collection

Basically global garbage collection is yet unsolved. Some problems are:

- local garbage collectors cannot remove global garbage
- local garbage collectors may change local addresses (annoying when these nodes have to be addressed by another processor).
- some garbage collectors, such as reference count garbage collectors cannot remove cyclic structures.
- detection of garbage reducers.

A possible solution would be to perform some kind of mark-sweep garbage collection over all processors simultaneously. This would mean that each time a garbage collection has to be performed, all reducers have to be stopped, the global garbage collection performed and next the reducers should be started again. Clearly this is very inefficient, certainly for large networks so this solution will not be chosen.

It seems better to combine different garbage collectors, e.g. a local copying garbage collector and a global reference count garbage collector (but then of course cyclic structures remain problematic). An important point is that one should try to solve problems locally as much as possible. As long as a local garbage collector is able to free enough memory other processors don't have to be bothered. Note that when a local garbage collection doesn't free enough memory, this doesn't mean the whole program should halt (as in the sequential case).

The problem that local garbage collectors may change local addresses can be solved with the help of translation tables that link global addresses to local addresses. The local garbage collector should take care the translation table is updated.

Nodes

In the PABC machine nodes consist of two parts: a fixed sized part and a variable sized part. This representation has been chosen to make overwriting of nodes possible. The fixed sized part contains three fields. The first field contains a pointer to the code that reduces the node to Head-Normal-Form (HNF). The second field contains a pointer to the descriptor of the node. This descriptor contains additional information about the node, e.g. the symbol of the node. Descriptors are shared between nodes of the same type. The third field contains a pointer to the variable sized field of the node. In this variable sized field the arguments of the node are stored.

Furthermore it should be possible to store a so-called waiting list in each node. This is basically a list of reducers that are suspended on the node.

In reality is deviated from this representation. For an efficient implementation it is important that nodes are as small as possible. In this way programs that require a lot of heap space are able to run, the number of garbage collections needed is decreased and graph rewriting becomes cheaper as less fields have to be updated.

Careful consideration of the PABC code that is needed to implement graph rewriting revealed the following properties that enabled us to shrink nodes:

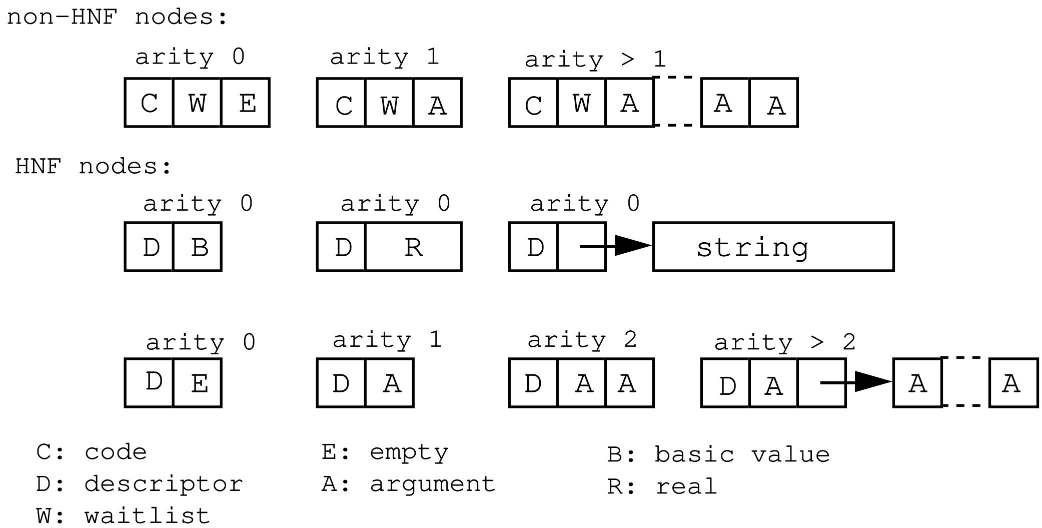
- The descriptor of a node is only needed when it is in HNF.
- Reducers will never be suspended on an HNF node.
- Only non-HNF nodes risk to be overwritten.
- When a node is overwritten, it is always overwritten by an HNF node.

The first property has lead to the combination of the code field and the descriptor field. A positive value D indicates that the node is in HNF and a pointer to the descriptor can be computed by taking -D. A negative value C means the node isn't in HNF and C is a pointer to the code of the node.

The second property means that only non-HNF nodes have to contain an extra field to store a waiting list. It is probably possible to misuse another field for this purpose, but we haven't done that because we still have to find out in what ways waiting lists are going to be used in the future.

The final two properties mean that HNF nodes needn't have a minimum size and that non-HNF nodes don't have to consist of two parts.

This has lead to the following node representation:



This node representation means that each time a reducer tries to evaluate a node it should check if the node is in HNF or not. In fact this improves execution speed because reducers very often try to evaluate HNF nodes. First of all, testing for HNF and jumping around some code (if the node is in HNF indeed) isn't less efficient than calling some code that returns immediately. But secondly, in cases as depicted below it is quite advantageous to test for HNF.

```
;this is what happens for the jsr_eval instruction

if (not_hnf(node(a_top)))
{
      check_stacks();
      save_locals();
      eval(node(a_top);
      restore_locals();
} else
      /* do nothing! i.e. all locals are preserved. */;
```

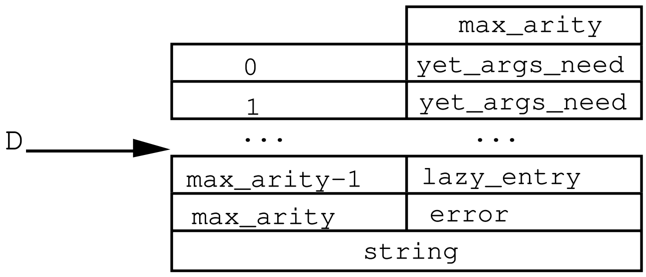
Below will be explained what certain structures a node may point to look like.

Waiting lists

Waiting lists are implemented as lists of workspace pointers (i.e. the workspace pointers of waiting reducers). Suspending a reducer is implemented by simply taking it out of the scheduling list of the transputer. When a node is updated all reducers in its waiting list are put back in the scheduling list.

Descriptors

Descriptors are shared between nodes of the same type. Notwithstanding they are used to store the arity of each node. Each descriptor actually consists of 'max_arity+1' entries. Depending on its arity the node points to one of these entries. Additional to the node-arity each entry also holds a pointer to the code that should be executed when a node with less than 'max_arity' arguments is applied to yet another argument. Finally each descriptor holds the value 'max_arity' and the symbol of the node (a string).



Entry code

the entry code of each node has the following form:

```
      .byte arity             ;arity of node
      rev                     ;used for jmp_eval
G_LABEL
      stl   return            ;store return address
J_LABEL
      ...
```

In each node the address of some 'g_label' is stored. The prefix 'g' stands for 'general call'. The transputer instruction 'gcall' leaves the return address in Areg. It is the only instruction that is able to take the contents of a register and call it. It is used to enter the code of a node, so the first thing to do is to store the return address at the appropriate place. 'j_label' is used by ordinary (conditional) jump instructions.

In order to be able to determine the arity of a node that doesn't have a descriptor, the arity of the node is placed above the entry code of the node (at 'g_label-2').

Furthermore: PABC instructions exist that should jump to the code entry of a node (without pushing the return address). For this they can't use the address (g_label+(j_label-g_label)), because the distance between the two labels may vary for different node entries (for reasons of code optimisation). The solution chosen is the following: the calling routine loads its own return address and gcalls 'g_label-1', so the 'rev' instruction will be executed first. Now the return address that is left by the gcall and the return address of the caller will be swapped. The PABC instructions that need this are very unimportant, so this doesn't make programs less efficient.

Context switches

Context switches 'occur' automatically. It's a bigger problem to prevent context switches at certain places than to 'generate' them. To accomplish this when necessary, care is taken not to use instructions that allow a context switch to occur. This is easier and faster than continuously increasing or decreasing the priority of a process.

Note that it is very important that context switches are impossible at certain places. For example: no context switches may occur within a basic block, as this would destroy the possibility for certain code optimisations. Also: the garbage collector may not be interrupted, as this would leave the heap in an inconsistent state.

Routing

The transputer has hardware that enables a process to communicate with a process on a neighbouring transputer. However, if a process wants to send a message to a process that resides on a transputer that isn't a neighbour, this message should be routed to its destination via some other transputers.

The PABC machine has instructions that enable an arbitrary graph to be sent to any processor in the network. It should also be possible to let any reducer start a new one on any processor. This means a router must be implemented for a (non fully connected) network of transputers. The efficiency of this router will greatly determine the overall efficiency of a Concurrent Clean program.

At the moment a simple router has been implemented. It sends all messages via the shortest path to the destination. Unfortunately it isn't deadlock free, so we've recently started to implement one that is. It will be based on the class-climbing algorithm that can be found in (Annot et al(1987)). This routing algorithm is deadlock free, provided that each message that arrives at its destination is 'consumed' within a finite amount of time.

Copying graphs

For the implementation of parallel graph reduction, we need to be able to send (parts of) graphs to other processors. The overhead needed for this should be kept as small as possible. In practice this means (amongst others) that one should try to send graphs in as large chunks as possible. At the same time care should be taken that only those parts of the graph are sent that actually need to be reduced on the other processor.

We aren't planning to send graphs node for node. This would result in a lot of overhead and additional to that, sharing of graphs would be lost completely. Instead we'll try to send the whole graph at once. This isn't always possible or smart: e.g. when part of the graph is already being reduced by some reducer (possibly on another processor), this part of the graph should not be sent, but stay at the original processor.

We haven't determined yet in what way exactly this will be accomplished. Below we'll discuss a possible solution.

When a graph is going to be sent to another processor, first a local copy is made of it (preserving sharing). This local copy will be an exact copy of the original graph, with the exception that some nodes, such as reserved nodes will not be copied. Furthermore, all nodes that are copied will be placed directly behind each other, so the final result will be a graph that resides in a contiguous area of memory. While copying, all pointers are changed to offsets (e.g. to the copy of the root node). In this way we are able to send the graph as if it were an ordinary sequence of bytes. The destination processor only has to update all pointers by adding a constant (e.g. the new address of the root node).

Such a local copying algorithm may be quite complex (and thus expensive). While copying, it might leave the original graph in an inconsistent state. This is very annoying for the garbage collector, so the size of the graph to be copied would have to be computed in advance. In this way it is possible to call the garbage collector before copying starts.

When a graph is sent, absence of deadlock in the routing algorithm must be guarantied. One way to accomplish this is to reserve memory for the graph on the destination transputer before the graph is sent away (and if some processor doesn't have enough memory available another destination might be chosen).

Performance

The following figures are a measure for the performance of the system in comparison to the implementation on a M68030. All programs are 'normal' sequential versions that are executed on a single transputer. The numbers indicate the execution time of the program in seconds.

	MacII-si	single T800
nfib	5.2	12.7
tak	5.3	11.3
sieve	9.4	16.4
queens	41	63.4
reverse	63	128
twice	1.5	2.5
rnfib	7.5	3.7
fastfourier	23	26.8

Queens gives a solution of the queens problem on a board of 10x10 fields. Sieve is a Concurrent Clean version of the sieve of Erathostenes. It computes the first 10,000 prime numbers. Nfib is a realisation of the following function:

```
nfib(x) =           if (x < 2 )
                    then 1
                    else (1 + nfib(x-1) + nfib(x-2))
```

The number above indicates the execution time of nfib(30). Tak is defined as follows:

```
tak(x,y,z) =        if  (y >= x)
                    then z
                    else (tak(tak(x-1,y,z),tak(y-1,z,x),tak(z-1,x,y))
```

The execution time needed to compute tak(24,16,8) is printed above. Reverse reverses a list of 3000 elements 3000 times. Twice is the function twice twice twice twice inc 0, where twice f x is defined as f (f x). Rnfib is a version of nfib using real numbers. Here rnfib 26.0 is computed. Fastfourier computes the fastfourier transformation over 8*1024 complex numbers.

Except for fastfourier, these programs have been run with the heap size set to 2Mb. Fastfourier is tested with 3Mb of heap. For the Macintosh programs the stacks had to be set to a sufficiently large size, as the M680x0 implementation doesn't reallocate stacks. In contrast, the transputer programs all started with an initial stack-size of 0 bytes. Comparing the two implementations one should also bear in mind that the Macintosh programs actually use more memory than 2Mb, because they don't allocate their stacks in the heap.

It should be noted that the T800 is slower than we expected. We have run some tests and these indicate that the T800 needs quite some time to fetch instructions from off-chip memory. If a program is placed in on-chip memory (and I mean: only the code, while keeping all data at the same place) it runs about 1.6 (!) times as fast. Unfortunately most useful programs are larger than 4K. In either case: the numbers presented above are the result of programs running in off-chip memory.

And further

A lot of problems haven't been solved yet (at least not entirely). Important topics of future research will be:

- routing
- copying of graphs
- distributed garbage collection
- load distribution
- code distribution
- program annotations
- implementation of large and useful parallel programs
- implementation of monitoring facilities
- increase of sequential performance

Conclusions

In this paper an efficient implementation of the PABC machine on transputers was presented. A lot of effort has been put in optimisations for sequential processing. This can be improved still further. It turned out that the need to check stacks and to reallocate them if necessary makes programs somewhat less efficient, but this loss isn't very serious. The same holds for the extra field needed in each non-HNF node to store waiting lists. The difference in performance (on a single processor) between lazy functional languages and traditional imperative languages has become acceptable. Implementation of a multi-processor version has started. We expect that true parallelism will make it possible to increase the performance of functional languages considerably.

References

Annot J.K., Twist R.A.H. van, (1987). 'A Novel Deadlock Free and Starvation Free Packet Switching Communication Processor'. Philips Research Laboratories. Eindhoven. Proceedings of the conference on Parallel Architectures and Languages Europe (PARLE'87). Springer Lecture Notes in Computer Science 258, Vol I, page 68-85.

Eekelen M.C.J.D. van, Plasmeijer M.J., Smetsers J.E.W., (1989). 'Communicating Functional Processes'. University of Nijmegen. Technical Report 89-3.

Eekelen M.C.J.D. van, Plasmeijer M.J., Smetsers J.E.W., (1990). 'Parallel Graph Rewriting on Loosely Coupled Machine Architectures' proceedings of the workshop on CTRS'90. Montreal Canada. To appear in 1991.

Groningen J. van. (1990). 'Implementing the ABC-machine on M680x0 based architectures'. Master Thesis, University of Nijmegen, November 1990.

Groningen J. van, Smetsers J.E.W., Nöcker E.G.J.M.H., (1991). Efficient Heap Management in the Concrete ABC Machine. Proceedings of the Third International Workshop on Implementations of Functional Languages on Parallel Architectures (1991). Technical Report Series, CSTR 91 91-07, June 1991, University of Southampton, United Kingdom.

INMOS, (1988). 'Transputer instruction set'. Prentice-Hall.

INMOS, (1988). 'Transputer reference manual'. Prentice-Hall.

Kesseler M., (1990), 'Concurrent Clean on Transputers', Master Thesis, University of Nijmegen, November 1990.

Koopman P.W.M., Eekelen M.C.J.D. van, Nöcker E.G.J.M.H., Smetsers S., Plasmeijer M.J. (1990). 'The ABC-machine: A Sequential Stack-based Abstract Machine For Graph Rewriting'. Technical Report 90-22, University of Nijmegen.

Nöcker E.G.J.M.H., (1989). 'The PABC Simulator, v0.5. Implementation Manual'. University of Nijmegen, Technical Report 89-19.

Nöcker E.G.J.M.H. (1990), 'Strictness Analysis using Abstract Reduction', Proceedings of the Second International Workshop on Implementations of Functional Languages on Distributed Architectures, University of Nijmegen, June 1990. Technical Report 90-16. page 203-222.

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 the conference on Parallel Architectures and Languages Europe (PARLE'91). Springer Lecture Notes in Computer Science 505, Vol. II, page 202-219.

Nöcker E.G.J.M.H., Plasmeijer M.J., Smetsers J.E.W. (1991). The PABC Machine. Proceedings of the Third International Workshop on Implementations of Functional Languages on Parallel Architectures (1991). Technical Report Series, CSTR 91 91-07, June 1991, University of Southampton, United Kingdom.

Parsec Developments, (1989). 'Par.C System Manual'.

Plasmeijer M.J., Eekelen M.C.J.D. van (1991). Functional Programming and Parallel Graph Rewriting. Lecture notes, University of Nijmegen, to appear at Addison Wesley 1991.

Sansom P.M. (1991). Dual-mode garbage collection. Proceedings of the Third International Workshop on Implementations of Functional Languages on Parallel Architectures (1991). Technical Report Series, CSTR 91 91-07, June 1991, University of Southampton, United Kingdom.

Smetsers J.E.W., (1989). 'Compiling Clean to Abstract ABC-Machine Code'. University of Nijmegen, Technical Report 89-20.

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. to appear.