Chapter 16
The Parallel ABC machine

This chapter presents the PABC machine, a parallel version of the abstract ABC machine (see Chapter 10). This PABC machine serves as an abstraction of the class of concrete parallel machines we focus on: MIMD machines with distributed memories. These concrete machine architectures are much more complex than their sequential counterparts. There are many difficult implementation problems that have to be solved before parallel programs can be executed efficiently on them. For example, one has to control the distribution of work such that each processor has enough work to do, one has to take care of the routing of messages over the communication network, garbage has to be collected that can be spread across several processors, and so on. Many of these problems are inherent to these MIMD architectures. They have to be solved for *any* parallel program, regardless of the language.

The PABC machine level abstracts from most of these implementation issues. The abstract machine would otherwise become much too complex. We will focus on the essential aspects of parallel graph rewriting only, such as *process creation*, *lazy copying* and the *communication* between arbitrary processes and processors.

For the description of the PABC machine the specification of the ABC machine presented in Chapter 10 is taken and extended. In this way again an executable specification is obtained. It is remarkable that the concurrency aspects of the machine can still be described understandably in a sequential functional language.

First, Section 16.1 discusses some basic ideas behind the architectural design of the PABC machine. Then, it will be described in detail how {P} and {I} annotations are handled. The design of the PABC machine architecture is such that the machine is also capable of dealing with more sophisticated annotations, such as {Par} and {Self}. In

Section 16.2 *interleaved* reduction is described, while *parallel* reduction is treated in Section 16.3. How the execution of the different (parallel) machine components is handled is explained in Section 16.4. Section 16.5 contains a small example showing how Concurrent Clean's process annotations are straightforwardly translated into PABC code.

16.1 About the PABC machine

In Section 13.1 we have assumed that concurrent functional programs at least have to run conveniently on MIMD machines with a distributed memory architecture. The main goal of the PABC design (Nöcker *et al.*, 1991a) is to introduce an abstract machine which can be efficiently implemented on those kinds of architecture. The PABC machine is defined as an extension of the ABC machine such that the sequential optimizations can still be applied in the parallel case.

The machine components of the PABC machine are more complex than those of the ABC machine. Instead of having one reducer, there are now *several reducers* possible on one single processor, while there are *several processors* as well. All processes and processors can communicate with each other. Processes are created *dynamically*, have to be scheduled and can be active or suspended (locked). In order to let this whole machinery work, it is no longer possible to have only *passive* machine components. As in real life an *operating system* is required as one of the processes on a processor. Furthermore, a *communication process* is provided on each processor to take care of the communication with the other processors.

16.1.1 The basic architecture of the PABC machine

The PABC machine consists of a number of *processors*. On each processor several processes are running interleaved with each other: zero or more *reducers*, an *operating system* and a *communication process*.

In principle, each reducer requires a private ABC machine (as introduced in Chapter 10). However, some of the ABC machine components can be *shared* by reducers executing on the *same* processor. The contents of the *program store* and *descriptor store* do not change during execution and are identical for each reducer. They can therefore be shared without problems. Interleaved reducers all have direct access to the graph, so the graph store has to be shared as well. The contents of this store can be changed by the reducers running on the processor. So one has to be careful and make sure that the graph store always remains in a consistent state. It will be clear that the *A*-, *B*- and *C-stacks*, as well as the *program counter*, cannot be shared between the reducers. Each reducer gets a private local version of these storages (see Figure 16.1).

Compared with the sequential ABC machine, there are also some differences. Input/output handling, as well as all interaction with the other processors, will now be handled by the *operating system* in collaboration with the *communication process*. Hence, *input/output channels* are no longer needed. There are also two new components needed for each reducer. One new machine component, the *N-queue*, is used by a reducer to store the node-ids of subgraphs that have to be reduced by it (the *N*odes to be reduced). This is convenient for the creation of reducers that reduce to normal form or that use some other reduction strategy. Another new component, the *waiting list register*, is a register used by a reducer to temporarily store the list of locked reducers that have to be awakened when the root normal form is reached. Finally, there is a small *administration* store located on each processor that, among other things, is used to identify the current reducer running on it.

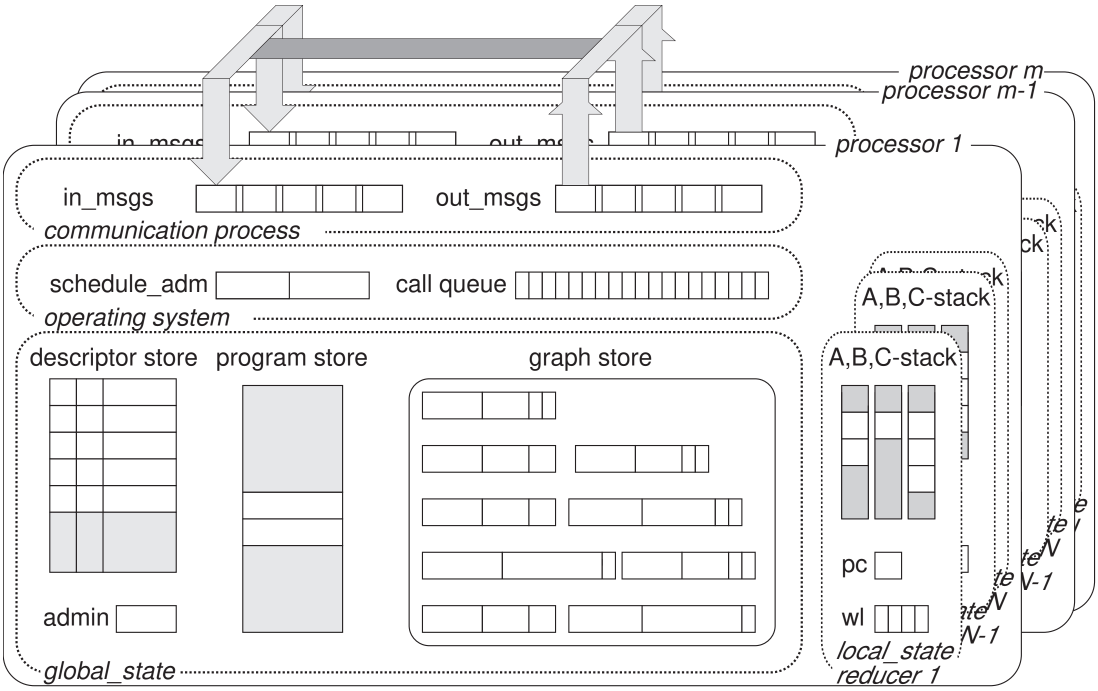

Figure 16.1 The basic architecture of the parallel ABC machine.

The *operating system* deals with the scheduling of the interleaved reducers and the communication requests coming from the reducers or the communication process. All processes can ask assistance from the operating system by putting a request in a queue (the *call queue*) of the operating system. The operating system process handles these requests. It can ask assistance from the *communication process* to handle *communication* between parallel reducers executing on different processors.

16.1.2 The description method

For the description of the PABC machine the specification of the ABC machine presented in Chapter 10 is taken and extended in the same spirit. But there are some differences. As explained above, there are several *kinds* of process on each processor: reducers, an operating system, and a communication process. Each such process performs different kinds of state transitions, and therefore different kinds of instructions are needed. Of course, it would have been possible to define instructions that work on the machine as a whole. But such a design would have led to a very ugly specification. So in the PABC machine specification **active components** are added: processes like the operating system and the communication process that have their own instruction set. Hence, besides ordinary *PABC instructions* (for reducers), there are *operating system instructions* (stored in the call queue) and *communication process instructions* (a fixed program). Each process uses some of the components of the machine for its own purpose (*local* components) while other components are shared with other processes (*global* components). The shared components are used to interface the different processes with each other.

The following sections briefly discuss the specifications of the most interesting parts of the PABC machine. The new instructions are summarized with a short explanation. The full specification of the PABC machine (including micro-instructions) is given in Appendix D.

The reducers

The part of the processor that is seen by a reducer, the **reducer state**, consists of three parts: the first part contains PABC machine storages *local* for the reducer (local_state), the second part contains the *global* storages (global_state) it shares with other reducers, and the last part is the call queue (opsys_callqueue) in which calls to the operating system can be put.

```
instruction   ==  state -> state
state         ==  (local_state, global_state, opsys_callqueue)
local_state   ==  (astack, bstack, cstack, nqueue, pc, wlist)
global_state  ==  (admin, graphstore, descrstore, programstore)
```

The PABC machine in principle incorporates all sequential ABC machine instructions. However, owing to the new state structure the description of each of these ABC instructions has to be slightly adapted.

The PABC specification of the pop_a instruction (see also Section 10.3.2):

```
pop_a:: nr_args -> instruction
pop_a nr_args ((as,bs,cs,nq,pc,wl),glob,cq)
   =  ((as',bs,cs,nq,pc,wl),glob,cq)
      where   as'   =  as_popn nr_args as
```

This actually shows a weakness of a specification in which state transition functions are being used. Even a small change in the state has as a consequence that all functions acting on that state have to be changed as well. A **record**-like data structure with named fields (as is present in ML and in many imperative programming languages) solves this problem. Record structures will be incorporated in a future version of Clean.

Some of the ABC machine instructions have to be changed somewhat further to incorporate context switches (see Section 16.2.2).

There are of course also several new machine instructions added for the control of process creation and communication. These new instructions are listed below (see also Sections 16.2 and 16.3).

```
new_ext_reducer  :: instrid -> a_src -> instruction  || create a {P} reducer
new_int_reducer  :: instrid -> a_src -> instruction  || create a {I} reducer
stop_reducer     :: instruction                      || stop current reducer

newP           :: instruction           || get processor-id for new process
randomP        :: instruction           || get a random processor-id
currentP       :: instruction           || get current processor-id
neighbourP     :: instruction           || get processor-id of a neighbour
channelP       :: a_src -> instruction  || get processor-id stored in channel

create_channel :: instruction                    || create new channel
send_graph     :: a_src -> a_src -> instruction  || lazy copy graph
send_request   :: a_src -> instruction           || request for lazy copy

is_empty_n     :: instruction           || are there node-ids in the N-queue?
get_node       :: instruction           || fetch node to reduce from N-queue
set_continue   :: a_src -> instruction  || store node to be reduced in N-queue

set_wait       :: a_src -> instruction  || administer reducer waiting for rnf
suspend        :: instruction           || lock current reducer
getWL          :: a_src -> instruction  || store locked reducers in register wl
release        :: instruction           || unlock reducers stored in wl
```

The operating system

The new parallel reducer instructions generally end with a request via the call queue (opsys_callqueue) to the operating system to do the actual work. Such an **operating system call** is an *instruction* that can be executed by the operating system (opsys_instruction). If the operating system gets control, it executes these calls, removing them from the queue.

As in real life, the operating system can be seen as the software extension of the machine. So the complete state of the processor is determined by the state of the operating system itself (opsys_state). The operating system has complete control over all processes on the processor. It therefore manages its own administration (schedule_adm) that gives the possibility to control (schedule) all the reducers living on the machine. To control communication the operating system also has access to the shared global components of the reducers. To communicate with the communication process it furthermore has access to the input and output list of messages (in_msgs, out_msgs).

```
processor       ==  opsys_state
opsys_state     ==  (schedule_adm,
                       in_msgs, out_msgs, global_state, opsys_callqueue)
schedule_adm    ==  ([ active ], [ passive ])
active          ==  reducer
passive         ==  reducer
reducer         ==  (redid, local_state)
```

A full list of the *operating system instructions* is given below.

```
opsys_instruction   ==  opsys_state -> opsys_state

os_suspend_reducer  ::  opsys_instruction
os_schedule         ::  opsys_instruction
os_reducerprio      ::  opsys_instruction

os_send_newreducer  ::  instrid   -> nodeid   -> opsys_instruction
os_new_reducer      ::  instrid   -> nodeid   -> opsys_instruction
os_stop_reducer     ::  opsys_instruction
os_halt             ::  opsys_instruction

os_send_newchannel  ::  procid    -> nodeid   -> opsys_instruction
os_send_graph       ::  nodeid    -> nodeid   -> opsys_instruction
os_send_request     ::  nodeid    -> opsys_instruction
os_setwait          ::  nodeid    -> opsys_instruction
os_release          ::  wlist     -> opsys_instruction

os_print            ::  string    -> opsys_instruction
```

The communication process

The communication process handles all interaction with other processors. The communication process is a simple process which repeatedly performs the *same* single instruction (cp_accept_msg). This instruction will see to it that a message is delivered to the intended processor.

```
comproc_instruction   ==  comproc_state -> comproc_state
cp_accept_msg::   msg_contents -> comproc_instruction
```

The communication process has access to the global structures of the processor such that it can read from and write to the graph store to ship lazy copies. A request for an outgoing message is done by the operating system. It puts a message into the output message list (out_msgs) of the communication processor. The incoming messages are fully dealt with by the communication process (in_msgs). For this purpose, the communication process can call the operating system if necessary via the call queue.

```
comproc_state   = =   (in_msgs, out_msgs, global_state, opsys_callqueue)
in_msgs         = =   [ msg_contents ]
out_msgs        = =   [ message ]
message         = =   (proc_src, proc_dst, msg_contents)
```

The communication network

An outgoing message sent away by a communication process has to be sent across a *communication network* such that it can be received as an incoming message by the communication process of the indicated processor. A simple communication network is chosen: all processors are connected with each other. Another option would have been to model a sparsely connected network topology as well as the routing of messages across such a network. This is assumed to be out of the scope of our abstract specification. Of course, if one is interested in modelling these aspects, they can be specified in any detail.

```
network           = =   [network_element]
network_element   :: =  P processor
```

16.2 Interleaved reduction

16.2.1 Creation and abortion of interleaved reducers

The PABC instruction new_int_reducer creates a new interleaved executing process reducing a graph of which the root is assumed to be on the A-stack (a_depth). The current reducer executing this instruction asks the operating system to perform the creation (os_new_reducer is put in the call queue with the micro-instruction os_call). The operating system will administer the new reducer as *active* and initialize the local storages such as the A-, B- and C-stacks, as usual.

Reducers created by an {I}-annotation reduce the graph to root normal form. To support other kinds of reducers, new_int_reducer is parametrized with an entry (instrid) to the code that drives the execution of the reducer to (root) normal form.

```
new_int_reducer:: instrid -> a_src -> instruction
new_int_reducer code a_depth ((as,bs,cs,nq,pc,wl),glob,cq)
   = ((as,bs,cs,nq,pc,wl),glob,cq')
     where  nid   =  as_get a_depth as
            cq'   =  os_call cq [ os_new_reducer code nid ]

os_new_reducer code nid (rs,in,out,(ad,gs,ds,ps),instrs)
   = (rs',in,out,(ad',gs,ds,ps),instrs)
     where  rs'                       =  (act_reds ++ [ new_red ], pass_reds)
            (act_reds, pass_reds)  =  rs
            (ad', new_redid)  =  ad_new_redid ad
            new_red           =  (new_redid,(as,bs,cs,nq,pc,wl))
                                 where  as   =  as_init
                                        bs   =  bs_init
                                        cs   =  cs_init
                                        nq   =  nq_new nid
                                        pc   =  pc_update code pc_init
                                        wl   =  wl_init
```

A reducer can quit with the instruction stop_reducer. This means that the reducer is stopped and all its stacks frames are released. This is effectively achieved by the operating system instruction os_stop_reducer.

16.2.2 Scheduling

Several reducers can exist on one processor. But only one of them can be executing: this is called the **current reducer**. All others have to wait until it is their turn for execution. At a certain point in time a **context switch** takes place: the execution of the current reducer is interrupted and the execution can proceed with one of the other reducers, which then becomes the new current reducer. Some kind of **scheduling** mechanism is needed: a mechanism that decides, when a context switch occurs, *which* of the active reducers on a processor is chosen for execution. Such a scheduling mechanism has to fulfil several requirements.

- Some kind of *fairness* should be assured (see Section 14.1). A fair distribution of the execution power avoids unnecessary locking of reducers and is furthermore needed because one can create reducers that do not contribute to the computation of the final result. So one should switch between interleaved processes *regularly*.
- The semantics prescribes that a context switch is only allowed after a rewrite step. And indeed, context switches cannot be allowed at arbitrary moments. When a process is manipulating globally accessible information (such as the graph store) it is possible that this information is temporarily in an inconsistent state. A context switch at such a moment is prohibited. Since graph rewriting on the

(P)ABC machine is optimized, there are some places in the code where it can be shown that a context switch cannot do any harm (for instance, a node is only rewritten when a root normal form is reached (see Chapter 11)).

- The sequential parts of the parallel ABC code should be amenable to the *same* optimizations as the sequential ABC code. In a basic block various kinds of optimization are performed (see Chapter 12). These optimizations would no longer be applicable if context switches within a basic block were allowed.
- A context switch is an *expensive* operation on a concrete machine (although the costs may vary from machine to machine). It should therefore be *avoided* as much as possible. Furthermore, on some machines context switches are only possible at *specific points* (e.g. transputers generate switches on jump instructions only).

Therefore, in the PABC machine the current reducer will never get interrupted by another reducer unless it executes an instruction in which *explicitly* a scheduling request is made to the operating system (using the instruction os_schedule). This happens in all instructions that end a sequential basic block (jmp, jsr, print, print_symbol) and furthermore in a few new instructions added for parallel processing (explained later on).

```
jsr address ((as,bs,cs,nq,pc,wl),glob,cq)
   =  ((as,bs,cs',nq,pc',wl),glob,cq')
      where  pc'  =  pc_update address pc
             cs'  =  cs_push (pc_get pc) cs
             cq'  =  os_call cq [os_schedule]

print string (loc,glob,cq)
   =  (loc,glob,cq')
      where  cq'  =  os_call cq [ os_print string, os_schedule ]
```

Given the generated code it can be shown that, when these instructions are being used and a context switch is requested, all the globally accessible stores on a processor will be in a consistent state.

The abstract operating system uses a very simple scheduling algorithm. The reducers living on a processor are divided into **active reducers** (wanting a time slice) and **inactive reducers** (that are **locked** or **suspended** because they are waiting for information). A context switch means that the current reducer is put at the end of a queue of active reducers and the next active reducer is fetched from the start of the queue. This **round robin** scheduling algorithm is just one method of dividing processor time fairly between active reducers. Any other algorithm can also be used as long as it obeys the fairness condition discussed above.

```
os_schedule ((act_reds,pass_reds),in,out,glob,instrs)
   = ((act_reds,pass_reds),in,out,glob,instrs),    if act_reds = [ ]
   = (rs',in,out,glob,instrs).                     otherwise
     where  rs'                  =  (new_act_reds, pass_reds)
            new_act_reds         =  rest_act_reds ++ [ red ]
            red : rest_act_reds  =  act_reds
```

16.2.3 Locking of reducers

If a reducer wants to reduce a node that is already under reduction, it has to be *locked* until the reduction of that node is finished (see Section 14.3). The first reducer that takes a node under reduction will *reserve* it before doing anything else. Of course, such a reservation can be done by a mark in the node, but it is more elegant and efficient to use the code field in the root node. In that case no flags nor tests are necessary. For this purpose, the node entry of the reduction code of each function has to be changed (see Section 11.1).

```
[       Label
"n_entry"                        ,  || The node entry
        Set_entry "_reserve" arity  ,  || Reserve this node
        Push_args 0 arity arity     ,  || Proceed as usual with the node entry
        ...
```

As a result, another reducer accessing a node with _reserve as code field (such a **node** is said to be **reserved** or **locked**) will enter the _reserve code instead of entering its node entry. The reserve code takes care of the suspension (or locking) of other reducers executing this code.

```
[       Label
"_reserve"            ,  || Code for locked nodes
        Set_wait 0    ,  || Put this reducer in the waiting list of this node
        Suspend       ,  || Lock this reducer
        Rtn           ]  || Return when awake again
```

By executing the set_wait instruction of the reserve code the current reducer is administered as being waiting for the result of the redex under reduction, after which the reducer locks itself with the suspend instruction. The operating system will put the reducer in the list of inactive reducers and perform a context switch to one of the active ones.

Reducers that hit a locked node are collected in a waiting list. A pointer to this list is stored in the node on which they are waiting. When such a node is overwritten with its root normal form, the locked reducers have to be released and made active. This has to be done *after* the update because otherwise the node is not in a consistent state yet. However, when the update takes place, the pointer to the waiting list is lost.

Therefore, the pointer is temporarily stored (using the getWL instruction) in a special register (the **waiting list register** wl) *before* the update takes place. *After* the update the reducers stored in this register are released by the release instruction.

```
getWL:: a_src -> instruction
getWL a_depth ((as,bs,cs,nq,pc,wl),(ad,gs,ds,ps),cq)
   = ((as,bs,cs,nq,pc,wl'),(ad,gs,ds,ps),cq)
      where   wl'   =  n_wl_get (gs_get nid gs)
              nid   =  as_get a_depth as

release:: instruction
release ((as,bs,cs,nq,pc,wl),glob,cq)
   = ((as,bs,cs,nq,pc,wl),glob,cq')
      where   cq'   =  os_call cq [ os_release wl ],  if ~ wl_empty wl
                    =  cq,                            otherwise
```

An example of the use of these instructions for an integer node:

```
GetWL a_depth          ,  || Get waiting list from node in wl
FillI 3 a_depth        ,  || Update the node with integer value 3
Release                ,  || Release locked reducers
…
```

16.2.4 *N*-queue

The ***N*-queue** is a special store in the PABC machine containing the node-ids of the root nodes of subgraphs that have to be reduced to root normal form by the corresponding reducer. A reducer stops if its queue is empty. For a root normal form reducer ({P} and {I} annotations) one node is stored in the *N*-queue when the reducer is created.

```
[        Label
"_rnf_reducer"           ,  || Code for a root normal form reducer
         Get_node        ,  || Get the node from the N-queue
         Jsr_eval        ,  || Reduce that node
         Pop_a 1         ,  || Clean up stack
         Stop_reducer    ]  || Rnf reached: stop
```

The *N*-queue store is added to the PABC machine to make the implementation of other kinds of reducer (such as normal form reducers) possible. The code for a normal form reducer very much resembles a driver in the sequential case (see Section 11.2) recursively invoking root normal form reducers. Each node that has to be reduced to normal form is stored in the queue. When the root normal form is reached the corresponding node is removed from the queue, while its arguments are stored. In their turn they will be evaluated in the same way until a normal form is obtained. The parallel context offers the opportunity to reduce the nodes in the *N*-queue in any order, even interleaved. This makes it possible to increase the effective parallelism. For instance, when a normal form reducer is locked it can still continue with any of the other nodes in the *N*-queue. Also, when a request from another reducer comes in, a normal form reducer can choose to create a new internal reducer to reduce the corresponding node in the *N*-queue such that the request can be answered swiftly. This requires a special code generation scheme for normal form processes. For optimal performance it might be better to create more special PABC machine instructions to deal with such situations. Accessing the *N*-queue requires two new instructions: get_node, which moves the first node in the *N*-queue to the A-stack (and pushes a boolean on the B-stack that indicates this was possible), and set_continue which appends a node to the *N*-queue.

```
get_node:: instruction
get_node ((as,bs,cs,nq,pc,wl),glob,cq)
   =  ((as',bs,cs,nq',pc,wl),glob,cq)
      where  as'   =  as_push n as
             n     =  nq_first nq
             nq'   =  nq_remove nq

set_continue :: a_src -> instruction
set_continue depth ((as,bs,cs,nq,pc,wl),glob,cq)
   =  ((as,bs,cs,nq',pc,wl),glob,cq)
      where  n     =  as_get depth as
             nq'   =  nq_add n nq
```

16.2.5 Representing deferred nodes

The semantics of the {P} and {I} annotation prescribes that all processes are created on a deferred node. In the PABC machine a node with a deferred attribute is represented by a special indirection node. So all pointers to a node with a deferred attribute will actually point to such a **deferred indirection node**. The advantage of this representation is that such a deferred indirection node can easily be treated specially by using a special descriptor (important for lazy copying). When the indirection node is overwritten the deferred attribute is automatically removed. Processes that are trying to evaluate a deferred node which is already under reduction have to be locked, as is explained in Section 16.2.3.

```
[  Label                              ,  || Code for evaluating a
"_defer_code"                         ,  || deferred indirection node
         Set_entry "_reserve" 1       ,  || Reserve this node
         Push_args 0 1 1              ,  || Get graph to evaluate
```

```
Jsr_eval                              ,  || Evaluate graph
GetWL 1                               ,  || Save locked reducers in wl
Fill_a 0 1                            ,  || Update node with result
Release                               ,  || Release locked reducers
Pop_a 1                               ,  || Clean up stack
Rtn                                   ]  || End of defer code
```

16.3 Parallel reduction

16.3.1 Creation of parallel reducers

The creation of a parallel reducer is split into several *phases* to keep the machine as flexible as possible:

(1) *build the graph* that has to be reduced in parallel;
(2) *find a processor* on which the reducer can be created;
(3) *create a channel* to that processor;
(4) make a *lazy copy of the graph to be reduced*;
(5) start *a new parallel reducer* on the remote graph.

A typical PABC instruction sequence resulting from a {P} annotation:

```
...                                  ,  || Code for building the graph
NewP                                 ,  || Destination proc-id on B-stack
Create_channel                       ,  || Create channel to that processor
Send_graph 1 0                       ,  || Send lazy copy graph to reduce
New_ext_reducer "_rnf_reducer" 0,       || Create external reducer process
```

Build the graph

When a redex is lazy copied to another processor the corresponding subgraph has to exist on the shipping processor. Due to the optimizations this might not always be the case. Then, additional code has to be generated that constructs a proper subgraph on the local processor before it is lazy copied to the destination processor. No new instructions are needed for this graph construction.

Find a processor

In Concurrent Clean processes can be created with a location directive (see Chapter 15). In the PABC machine each processor has a unique identification, the **processor-id**. Special instructions are provided (newP, randomP, currentP, neighbourP and channelP) that push a processor-id on top of the B-stack to support these location directives.

Create a channel

References to nodes on another processor are represented by **channel nodes** (a lazy copy node). On the PABC machine level, a special indirection node is used with an argument field containing an identification of the process on the remote processor. The instruction create_channel assumes the remote processor-id for the channel to be on top of the B-stack. To inform the remote processor that a channel has to be created, a message is sent to it. The remote processor will create a new empty global node. Its node-id will be returned over the communication network. With the processor-id it is stored in the new channel node.

```
create_channel:: instruction
create_channel ((as,bs,cs,nq,pc,wl),(ad,gs,ds,ps),cq)
   = ((as',bs',cs,nq,pc,wl),(ad,gs',ds,ps),cq')
     where bs'        = bs_popn 1 bs
           pid        = bs_getP 0 bs
           (gs',nid)  = gs_newchannel pid gs
           as'        = as_push nid as
           cq'        = os_call cq [ os_send_newchannel pid nid,
                                     os_setwait nid,
                                     os_suspend_reducer ]
```

Make a lazy copy of the graph

send_graph sends the graph that has to be reduced in parallel to its destination. The destination address is fetched from the channel node.

```
send_graph:: a_src -> a_src -> instruction
send_graph graph_depth chan_depth ((as,bs,cs,nq,pc,wl),glob,cq)
   = ((as,bs,cs,nq,pc,wl),glob,cq')
     where chanid  = as_get chan_depth as
           graphid = as_get graph_depth as
           cq'     = os_call cq [ os_send_graph graphid chanid ]
```

The lazy copying algorithm is given in Appendix D. Copying has to preserve the structure of the graph. The algorithm requires two scans of the graph: one scan to build the copy and one scan to remove all the forwarding references and markers that have been created to preserve the shared structure in the copy.

Start a new parallel reducer

Finally, the new reducer will be started on the remote processor (remembered in the channel node) using the instruction new_ext_reducer. A message is sent to the remote processor for creating the new reducer. The reducer is created with a label code indicating the kind of reduction code that has to be executed (e.g. to root normal form or to normal form). The remote processor will create and initialize a new reducer and put the root node of the remote graph in the *N*-queue as usual.

```
new_ext_reducer:: instrid -> a_src -> instruction
new_ext_reducer code chan_depth ((as,bs,cs,nq,pc,wl),glob,cq)
   = ((as,bs,cs,nq,pc,wl),glob,cq')
      where   chanid  =  as_get chan_depth as
              cq'     =  os_call cq [ os_send_newreducer code chanid ]
```

16.3.2 Demanding a lazy copy across a channel

The result of a parallel process is communicated to a demanding reducer when it is needed, i.e. when a corresponding channel node is evaluated. A channel node is evaluated as any other node: by executing its code.

```
[       Label
"_channel_code"               ,  || The code associated with a channel node
        Set_entry "_reserve" 0,  || Reserve the channel node
        Send_request 0        ,  || Send a demand for the result
        Set_wait 0            ,  || Put reducer in waiting list of this node
        Suspend               ,  || Lock the current reducer
        Rtn                   ]  || Return when awake again
```

First, the channel node will be reserved as for any other node: other reducers that need the result just have to wait. Then, the instruction send_request asks the remote processor for the result of the reduction of the graph belonging to the channel node.

```
send_request:: a_src -> instruction
send_request chan_depth ((as,bs,cs,nq,pc,wl),glob,cq)
   = ((as,bs,cs,nq,pc,wl),glob,cq')
      where   cq'    =  os_call cq [ os_send_request chanid, os_reducerprio ]
              chanid  =  as_get chan_depth as
```

After sending the request the demanding reducer is suspended to wait for the result (with the instructions set_wait and suspend). The instruction os_reducerprio guarantees that administration and suspension are done with highest priority, i.e. before the result is already communicated.

The request is answered as soon as the requested node is in root normal form (regardless of whether the reducer reduces to root normal form or to normal form). The result can be returned immediately as a lazy copy if it is available. Otherwise, the request is stored in the waiting list of the remote node under reduction. In that case the request will be answered when a release instruction is executed for this node. On the demanding processor the actual update with the lazy copy of the result will be performed automatically by the communication process.

16.4 Program execution

In order to obtain an executable specification of the PABC machine one needs an implementation of the abstract data structures and micro-instructions, as in Section 10.4. For the PABC machine this includes a specification of the rather powerful operating system instructions and the instructions of the communication process. Furthermore, one needs to define how communication (and lazy copying) takes place.

This section only treats the PABC *bootstrapper* and *instruction fetch* mechanisms (for the other aspects we refer to Appendix D). A special solution is necessary since real parallelism and non-deterministic interleaving are *not* directly expressible in a functional language. The concurrent behaviour has to be *simulated* in some way.

16.4.1 The instruction fetch cycle

To run the PABC machine one not only has to specify the execution behaviour of a single processor (which involves fair interleaving of operating system, communication process and reducers), but also the behaviour of the network of parallel processors (which involves simulation of parallelism and distribution of messages across the network). Altogether the specification of the behaviour of the machine implies *several* instruction fetch cycles that call each other.

The PABC machine cycle

The PABC machine consists of a number of processors connected via a very simple communication network. In a PABC machine cycle all messages on outgoing lines from the processors are collected, removed and distributed to the incoming lines. Outgoing messages that are the output of the overall reduction process are added to the final output of the PABC machine. The parallelism is simulated by giving each processor a time slice by calling the processor cycle for it.

```
output       ==   [ msg_contents ]

machine_cycle:: network -> output
machine_cycle nw
    =  output                                       , if op_halt output
    =  op_append output (machine_cycle nw')         , otherwise
       where  output  =  nw_output nw
              nw'     =  nw_exec proc_cycle (nw_distribute_msgs nw)
```

The processor cycle

Each processor has to guarantee a fair distribution of CPU time between the processes running on it. The choice has been made to give the communication process and the current reducer each a time slice in each processor cycle. In order to handle operating system calls as soon as possible, the operating system gets a time slice both after the reducer cycle and after the communication process cycle.

```
proc_cycle:: processor -> processor
proc_cycle =  opsys_cycle . red_cycle . opsys_cycle . comproc_cycle
```

The communication process cycle

The communication process handles all incoming messages in one cycle. However, it can happen that the current reducer has requested priority over the communication process (see Section 16.3). In that case communication will not be handled in the current processor cycle.

```
comproc_cycle:: processor -> processor
comproc_cycle (rs, in, out, (ad,gs,ds,ps), cq)
    =  (rs, in, out, (ad',gs,ds,ps), cq),   if ad_prio ad = Prio_Reducer
       where    ad'    =  ad_new_prio No_Prio ad
comproc_cycle (rs, [ ], out, glob, cq)
    =  (rs, [ ], out, glob, cq)
comproc_cycle (rs, (msg : msgs), out, glob, cq)
    =  comproc_cycle (rs, in', out', glob', cq')
       where    (in', out', glob', cq')  =  cp_accept_msg msg (msgs,out,glob,cq)
```

The operating system cycle

The operating system performs all pending calls in one cycle.

```
opsys_cycle:: processor -> processor
opsys_cycle (rs, in, out, glob, cq)
    = (rs, in, out, glob, cq),                              if os_cq_empty cq
    = opsys_cycle (os_first_call cq (rs,in,out,glob, os_rest_calls cq)),otherwise
```

The reducer cycle

The cycle of the current reducer calls fetch_cycle to execute the current reducer. The execution of a reducer is driven by a slightly modified version of the instruction fetch cycle of Section 10.4.1. fetch_cycle terminates when the current reducer calls the operating system.

```
red_cycle:: processor -> processor
red_cycle ((act_reds, pass_reds), in, out, glob, cq)
```

```
    =  ((act_reds, pass_reds), in, out, glob, cq),    if act_reds = [ ]
    =  (rs', in, out, glob', cq'),                    otherwise
       where  rs'                =  ((rid, loc') : reds, pass_reds)
              (rid, loc) : reds  =  act_reds
              (loc', glob', cq') =  fetch_cycle (loc, glob, cq)

fetch cycle:: state -> state
fetch_cycle ((as,bs,cs,nq,pc,wl), (ad,gs,ds,ps), cq)
    =  ((as,bs,cs,nq,pc,wl), (ad,gs,ds,ps), cq),          if ~ os_cq_empty cq
    =  fetch_cycle (currinstr ((as,bs,cs,nq,pc',wl) ,(ad,gs,ds,ps), cq)), otherwise
       where  pc'        =  pc_next pc
              currinstr  =  ps_get (pc_get pc) ps
```

16.4.2 Booting the machine

To boot the PABC machine not only do program and descriptors have to be supplied, but also the number of processors.

```
boot:: nat -> ([ instruction ], [ descr ]) -> network
boot nr_proc (program, descriptors)
    =  nw_init nr_proc (program, descriptors)
```

16.4.3 Running a program

So the PABC machine starts evaluating a program as follows.

machine_cycle (boot *nr_of_processors* (assembler (*program*)))

16.5 Translating Concurrent Clean into PABC code

The PABC instructions are on a high level, very close to the process annotations available in Concurrent Clean. So the translation is straightforward. The important aspects are already covered in the previous sections. In this section an example is given of code generated accordingly.

Take the second rule alternative of Sieve in an example of Section 15.2.3:

```
Sieve [ p | s ]  ->     [ p | {P} Sieve ({I} Filter p s) ] ;
```

The generated code for the right-hand side of this alternative (extra instructions needed for the interleaved and parallel evaluation are emphasized):

```
[  Label
"Sieve2"                                   ,  || Entry for the second alternative
   Push_args 0 2 2                         ,  || Push the arguments
   Create                                  ,  || Node for defer attribute
```

```
Create                                  ,  || Node for result of Filter
Push_a 2                                ,  || Push second argument of Filter
Push_a 4                                ,  || Push first argument of Filter
Fill "Filter" 2 "n_Filter" 2            ,  || Fill node for Filter
Fill "_Defer" 1 "_defer_code" 1         ,  || Fill deferred indirection node
Create                                  ,  || Node for result of Sieve
Push_a 1                                ,  || Push argument of Sieve
Fill "Sieve" 1 "n_Sieve" 1              ,  || Fill node for Sieve
NewP                                    ,  || Destination proc-id on B-stack
Create_channel                          ,  || Create channel to that processor
Send_graph 1 0                          ,  || Send lazy copy graph to reduce
Update_a 0 1                            ,  || Update Sieve node with channel
Pop_a 1                                 ,  || Remove channel from A-stack
New_int_reducer "_rnf_reducer" 1,          || Create internal reducer process
New_ext_reducer "_rnf_reducer" 0,          || Create external reducer process
Push_a 0                                ,  || Push second argument of Cons
Push_a 3                                ,  || Push first argument of Cons
GetWL 7                                 ,  || Save locked reducers in wl
Fill _Cons 2 "_rnf" 7                   ,  || Update root node of process
Release                                 ,  || Release the locked reducers
Pop_a 5                                 ,  || Clean up stack
Rtn                                     ]  || End of alternative
```

Summary

- The PABC machine is an abstract distributed machine which is an extension of the sequential ABC machine such that the sequential optimizations can still be applied.
- The PABC machine consists of the following components:
  - a *network* of *processors* that communicate messages;
  - a *communication process* on each processor that deals with outgoing and incoming messages;
  - an *operating system* on each processor which can be called via a queue of pending operating system instructions. The operating system deals with lazy copying, scheduling, creation and locking of reducers etc.;
  - a number of *reducers* on each processor. A single reducer performs sequential reduction like the sequential ABC machine.
- The instruction set of the PABC machine extends the sequential ABC machine for communication and synchronization with a relatively small number of instructions based on lazy copying and parallel graph rewriting.
- The PABC machine can be specified elegantly in a functional language in very much the same way as the sequential ABC machine.

- A Concurrent Clean program can be translated to PABC code quite straightforwardly. Code has to be generated for locking and releasing of reducers. Furthermore, each process annotation gives rise to a sequence of specific PABC machine instructions.

EXERCISES

16.1 Compile the parallel programs written in the previous exercises to PABC code. Study the generated code. Some PABC instructions are different. Which ones? Can you explain what the advantages of these changes are?

16.2 Refine the PABC machine specification such that processors are connected via a sparsely connected communication network instead of a fully connected network. Define the simulated topology in a table such that it can be changed easily. Also define in a table how messages have to be routed to the intended destination. Change the communication processor such that it can take care of the routing of messages. Then, define the instructions newP, randomP, currentP, neighbourP and channelP properly within the PABC specification.

16.3* Suppose that you had to design an *S*ABC machine to model a *shared* memory architecture instead of a *distributed* memory architecture.

(a) Is lazy copying still necessary? Is it necessary to make a distinction between {P} and {I} processes? Which PABC components would you still need? Which ones can be shared? Which of the PABC instructions are not needed any more? Are there any new instructions needed?

(b) The SABC machine has a different architecture from the PABC machine. This means that all PABC machine instructions have to be adapted such that they can be applied on the new state. Are there also non-trivial changes in instructions necessary? Hint: remember that parallel evaluation is only allowed when the rewritings cannot interfere with each other.

(c) Give a picture similar to Figure 16.1 depicting the SABC architecture.

16.4* Specify the SABC machine.