Chapter 17
Realizing the PABC machine

This chapter discusses the concrete realization of the PABC machine. For this purpose one at least needs a good *sequential* code generator. Such a code generator can be developed using the techniques described in Part 4. First, Section 17.1 treats in general terms the many additional problems one has to solve when 'going parallel'. Suitable representations of the different new PABC machine components are given. In particular, attention is given to the rather long list of run-time support that has to be offered on the concrete target machine in either hardware or software. Garbage collection on a distributed machine architecture is discussed separately (Section 17.2).

The remaining part of this chapter treats the realization of the PABC machine on a specific machine, a 32 node transputer architecture. The transputer processor is explained in Section 17.3. A concrete representation of each of the PABC components on this processor is given (Section 17.4). Finally, the most important concrete aspects of actual code generation are treated, including a discussion of the obtained performance (Section 17.5).

17.1 Representing the PABC machine components

The PABC machine has been specifically designed to be realized on a concrete MIMD architecture in which each processor has its private memory and processors are (sparsely) connected via some special communication network. This section briefly explains how each of the PABC machine components can be realized on concrete architectures.

The most complex component of the PABC machine is clearly the abstract operating system. The PABC machine abstracts from a lot of problems for which a solution has to be found on a concrete architecture. So one not only has to provide facilities like lazy copying and communication between arbitrary processes and processors: one also has to take care of the collection of garbage that can now be distributed over the different processors and of the memory management for the allocation of the dynamically created stacks. Furthermore, the distribution of tasks has to be dealt with such that all processors are utilized in the best possible way. All these activities have to be performed by the run-time system of the application in combination with the support offered by the concrete operating system of the machine. The complexity of the concrete run-time system therefore depends heavily on the facilities offered by the processor and the concrete operating system.

17.1.1 Representing the reducer components

The program store and the program counter

Each processor in a distributed memory machine will need its own copy of the program in its private memory. The easiest way to accomplish this is by copying the complete program to a particular location in the store of each of the processors *before* the execution starts (**static loading**). Most concrete machines have instructions to support this. Another strategy is to load precisely that part of the program into each memory that is actually being used by the corresponding reducer (**dynamic loading**). This saves memory occupation but it may become an inefficient strategy if process creation occurs very frequently and different pieces of code have to be (re)loaded often. Also the addressing may become more complex, especially when the architecture has no facility to support dynamic loading.

The program counter of an abstract PABC reducer can, of course, be directly represented by the program counter of the concrete processor the reducer is running on.

The A-stack, B-stack and C-stack

Because several reducers may run on one processor, *several* stacks have to be allocated in memory instead of just *three* stacks as in the sequential case. Furthermore, stacks are created dynamically when a new reducer is created. Consequently, the stack (frame) management is much more complicated than in the sequential case. In general, it is not known in advance how large a stack a reducer needs. If (too) small stack frames are allocated initially, problems arise when the stacks become too large. If too big stacks are allocated initially, heap space is wasted. One solution is to use linked stacks, but then stack addressing becomes more complicated. Another solution is to allocate small stacks initially and to *reallocate* these when necessary. In that case a new and larger stack frame is allocated in a contiguous piece of memory to which the contents of the old one are copied. This copying seems to be a quite inefficient solution, but it often works satisfactorily in practice. Normally, reducers only use small stacks and reallocations are not often needed.

When a reducer dies, its stacks become garbage. They have to be collected such that the space they occupy can be reused. An obvious solution for this problem is to handle stacks as normal heap objects, such that their space can be reclaimed by an ordinary garbage collector. When stacks are incorporated into the heap one has the additional advantage that no distinction has to be made between the amount of memory available for stacks and the amount of memory available for graphs. This division can automatically be tuned at run-time. If stacks are allocated randomly in the heap (just as ordinary nodes) they will always be copied during a garbage collection when a copying or compacting garbage collector is activated. To avoid this problem it is better to reserve a special part of the heap to accommodate stack frames (which makes the heap management a bit more complicated).

As discussed in Chapter 12, one can combine the stacks of a reducer in an actual implementation. Merging the B- and C-stacks is certainly advisable for parallel environments. It makes the management of stack frames easier and reduces the amount of boundary checks. For garbage collection it is convenient to keep a separate A-stack such that pointers to nodes can easily be distinguished from other data.

The N-queue and the waiting list register

The implementation of an *N*-queue can be realized in the same way as the implementation of stacks, e.g. as a special relocatable object in the heap space. For the waiting list register any free machine register or memory location can be used.

The graph store

A new aspect of the graph store is that one now has to deal with waiting lists and with defer attributes. Defer attributes can either be represented by a special indirection node (as suggested in the PABC machine) or by a tag in the node itself.

Waiting lists are more problematic. The easiest way to represent them is by inserting an additional field in the fixed part of a node containing a pointer to the actual waiting lists. The waiting list itself can be represented in the heap. Adding an additional field has the disadvantage that all nodes become larger. This gives rise to increased memory consumption. However, in practice it is possible to (mis)use one of the existing fields that is not being used when the waiting list has to be constructed. For instance, it is possible to use one of the argument fields of a node because a node with a waiting list is certainly being reduced and for a node being reduced the original arguments have already been saved on the stack.

Another solution is not to use waiting lists at all. Instead, waiting reducers can execute a polling loop in which they examine whether the node they are waiting for is updated or not. With clever scheduling scheme polling can be cheaper than a waiting list administration with explicit reactivation of locked reducers (Van Groningen *et al.*, 1992).

The descriptor store

The descriptor store can be treated in the same way as the program store: a copy of (a part of) the store has to be loaded either statically or dynamically into each of the processor stores.

17.1.2 Operating system and communication process

The abstract parallel reducer instructions are effectively all handled by the operating system located on the corresponding abstract processor. The operating systems communicate with each other via the communication processes. In practice, the tasks performed by these abstract components are realized by the *run-time system* of the program with the help of the concrete operating system running on top of the concrete machine. Furthermore, many other tasks which we have abstracted in the PABC machine have to be performed by these systems.

Process creation, scheduling and context switching

Not all kinds of processor have direct hardware support for process creation, scheduling and context switching. In most cases additional software support is needed. The PABC machine instructions specify when a context switch between processes is allowed. The scheduling has to be fair. This means that the generated code must be such that context switches indeed take place regularly and that all reducers eventually will get a time slice. Some concrete processors (like, for example, the transputer) support an automatic context switch (consisting of a move of merely two pointers) on certain instructions (e.g. after each jump instruction). This makes switching reasonably efficient. Other systems offer a possibility to raise interrupts at a specified time interval. When the interrupt occurs it can be used by the interrupt handler to schedule another process. In that case one has to ensure that context switches only take place when the stores are in a consistent state and no critical actions are being undertaken (e.g. garbage collection). A context switch can then also be done rather efficiently, but special code is needed to prevent switching at unwanted moments. If the individual processors are of a more conventional type (like Motorola's) special code has to be inserted at certain places that forces a switch on a regular basis. Context switches are probably a bit more expensive in this way. Anyhow, every context switch will cause a certain amount of overhead for saving the old context and restoring a previous one. So on the one hand one should switch regularly for fairness reasons and on the other hand one should not switch unnecessarily to avoid wasting of CPU time.

Communication between processors

The PABC instructions require that any two processes located on arbitrary processors can directly communicate with each other. In practice, the processors in a distributed memory architecture are only sparsely connected in some network topology. Some systems provide the possibility for any two processors to communicate with each other even if there is no direct hardware link between them. Other systems only provide communication between processors that have a direct link. In that case additional communication software (a **router**) is needed to provide reliable and fast communications between arbitrary processors.

Load balancing

Parallel reducers can be created on arbitrary processors. The operating system is free to use any information to select such a processor. In order to obtain ultimate performance the system should distribute the work evenly over the available processors. This distribution of work is called **load balancing.** In an *optimal* balanced system all processors have an equal amount of work to do: they all have enough free memory while the communication overhead is minimal due to the fact that processes which communicate heavily with each other are ideally located.

Load balancing is an important task of the operating system. However, in general it is undecidable to determine an optimally balanced distribution of tasks. To find a good approximation, ideally all processors should always be kept completely informed about the status of the whole machine. This is hard to realize on a distributed memory architecture without introducing an unacceptable amount of communication overhead. Therefore, in practice a suitable processor has to be chosen based on only *partial* knowledge about the workload of other processors. Neighbouring processors or processors with which communication is already taking place are good candidates to obtain the above information without facing too much communication overhead.

Process migration

It is a waste of machine capacity if the load is not well distributed and some processors have too little work or even nothing to do at all. Things become problematic when processors have so much work to do that they are running short of memory space (even after garbage collection has been performed). In that case one either has to stop the whole machine or one has to try to get the system in balance by transferring work from overloaded processors to other processors. Moving work effectively means moving a reducer by making a lazy copy of its graph. This is a delicate task since there is probably not much heap space on the overloaded processors to prepare the lazy copy.

Channels and global nodes

A channel node refers to a global node located on a remote processor. Such a reference cannot simply be the address of that node on the remote machine. Global nodes are, like ordinary nodes, sometimes moved by the garbage collector. An *indirection table* on the remote machine can be used to obtain a fixed address location. A channel node then refers to a fixed location in the table in which the varying address of the global node is stored. The table can also be used to store a reference count (representing the number of channels pointing to the graph) used for global garbage collection (see next section).

When a processor has to answer a request it must know to which processor it has to send the graph such that the channel node on the requesting reducer can be overwritten with the result. Clearly, channel nodes can be moved around by the garbage collector as well. Therefore, the address of a requested channel node has to be stored in an indirection table as well.

Lazy copying

A lazy copy of a graph must preserve the graph structure. One cannot simply apply an algorithm like those often used by copying garbage collectors: forwarding pointers (see Section 12.2) violate the original graph. Furthermore, garbage collection cannot be allowed during a lazy copy action: the garbage collector should not find the heap in an inconsistent state. There are several possible solutions, each with their own space-time behaviour. An important aspect to keep in mind is that graphs to be copied are expected to be relatively small, since otherwise the communication overhead would probably be too large.

One solution uses pointer reversal to traverse the original graph in advance to compute the size of the graph before copying starts. Next, the garbage collector is called (if necessary), directly followed by the (indivisible) copy action itself using forwarding addresses. Finally the original graph is restored, again using pointer reversal to traverse it. An advantage of this algorithm is that it does not require more memory than needed to store the lazy copy itself. A disadvantage is that it requires three scans of the graph to be copied.

Another solution uses the end of the heap temporarily as a stack to keep track of the original nodes. Whenever a node is encountered for the first time during copying its address is pushed on this stack. After copying has taken place the stack is traversed and all nodes encountered are restored. Finally, the stack is removed. If the garbage collector has to be called during copying the original graph must be restored in advance (again using the stack at the end of the heap). The copy that has been built so far has to be considered lost. After the garbage collection has been performed the copying must start all over again, increasing the cost of a copy action considerably (in the worst case it takes twice as long). But this situation is not likely to happen very often.

Sending of graphs

Sending a graph to another processor is usually realized as follows. Locally a lazy copy of the graph is made, as explained above. This copy is provided with relocation information. The copy cannot be sent away immediately. On the destination processor enough heap space must be available as well. This implies that first enough heap space on the remote processor has to be reserved. Further sending has to be suspended until an acknowledgement has been received that the amount of memory indeed could be reserved. Hereafter the graph can be sent away and, finally, when the graph arrives at the destination, all pointers in the graph have to be adjusted using the relocation information.

Deadlock detection

It is possible to write functional programs that contain deadlock (see Chapter 14). It would be nice if deadlock could be detected by the system. This is certainly important in a multi-user environment where a deadlock blocks (a whole) part of the machine. Deadlock that involves reducers located on one processor is not difficult to detect: one has to check whether locked reducers are mutually waiting on each other. Of course, deadlock will be much harder to detect without considerable communication overhead when the reducers involved are located on different processors.

Input/output

The treatment of I/O highly depends on the kind of I/O (file I/O, screen I/O) and the architecture of the concrete machine. In most concrete architectures only a few of the available processors will have a direct connection to secondary memory (disk) or to a video processor. Most of the time only one processor offers such facilities, sometimes indirectly via a connection with a host machine. This means that all I/O actions have to be realized by sending messages across the communication network to this particular processor. I/O is therefore typically a facility offered by the operating system of the concrete machine.

17.2 Garbage collection on a distributed machine

It is difficult to accomplish efficient garbage collection on a distributed machine. The possibly cyclic graph is now distributed across the different processors. To determine whether or not a node has become garbage references to that node from other processors have to be taken into account. One certainly would not like to stop the whole machine because one of its processors needs a garbage collection. Therefore, generally the collection process is first split into a local part and a global part (Kingdon *et al.*, 1991). The **global garbage collection** tries to determine which of the *global nodes* are garbage. The **local garbage collection** basically removes garbage locally, knowing which of its global nodes are garbage or not. Often for local and global garbage collection different techniques are combined. To recover all cyclic structures, information has to be passed from the local to the global collector and vice versa. There still does not exist a completely satisfactory solution to recover *all* distributed garbage efficiently.

Local garbage collection

In principle, local garbage collection can be done by using a standard copying garbage collector/mark–scan technique (see Chapter 12). Each of the non-garbage global nodes must now be regarded as a root of a subgraph that cannot be collected. The global nodes themselves are collected differently (see below).

In a sequential implementation the garbage collector is only invoked when a processor runs out of heap memory. In a parallel implementation garbage collection needs to occur more regularly (for instance with the help of a timer) even when there is enough local memory available. Suppose a processor p_1 holds a node n_1 that contains a reference to a node n_2 on another processor p_2 and suppose that n_2 is the root of a large graph. At some point in time n_1 may become garbage but p_1 will not discover this until it performs a local garbage collection. Consequently n_2 will not be removed before this time either.

In a parallel implementation it is very important that collection takes place as soon as possible. When a processor cannot reclaim its heap space because it is unaware of the status of its global nodes, none of its reducers can continue either and eventually the whole parallel machine will get blocked. Furthermore, if a process has been created on a node that has become garbage (a **garbage reducer**), it has to be stopped quickly because such a process is actively creating garbage, wasting the resources of the computer.

Global garbage collection

For garbage collection of global nodes a reference count garbage collector has certain advantages over copying/mark–scan. With a reference count collector a global node (and the possibly large subgraph with this node as root) can be collected locally as soon as its reference count becomes zero. When a channel node containing a reference to a global node is collected by its local garbage collector, a message is sent to the corresponding processor to decrement the reference count of the global node. When a channel node is copied or requested, a message is sent to increase the global reference count. A disadvantage of reference counting is that it cannot reclaim cyclic structures (see Chapter 12).

The straightforward implementation of global reference counting may cause severe problems. When a decremental message arrives before an incremental one it can result in removal of a global node that is not garbage at all. This problem can be solved by a **weighted reference count** algorithm (Bevan, 1987; Watson and Watson, 1987). The advantage of this algorithm is that *only* decremental messages are needed. Initially, a huge power of two is stored as reference count in both the channel node and the corresponding global node. When a channel node is copied its reference count is divided by two. This new reference count is stored in the original channel node as well as in its copy. When the reference count drops down to one (which is not very likely to happen often in practice) a new indirection node to the channel is created, again containing a huge power of two. When a channel node is deleted its reference count is sent to the processor it is referring to, such that the reference count of the global node can be decremented by that number. When this reference count drops down to zero the global node has become garbage and it can be collected.

Removal of cyclic structures

Cyclic structures that are distributed over multiple processors cannot be collected when a reference count collector is being used. An option is to use a mark–scan collector (for the whole network, for all local and global nodes) as a *secondary* collector, solely to remove cyclic structures when memory gets scarce in spite of local and global collecting attempts. For this purpose either a stop-and-collect or an on-the-fly collector can be used.

Removal of garbage reducers

To check whether a reducer is a garbage reducer it is sufficient to check whether all the nodes the reducer has reserved have become garbage.

17.3 The transputer processor

A transputer is a processor with hardware support for concurrent processing. It has instructions to start and stop processes at two priorities. *Low-priority processes* are automatically time-sliced in a round robin fashion by means of a hardware scheduler. *High-priority processes* are not time-sliced. Each transputer has four hardware links with which it can be connected to four other transputers (Figure 17.1). Instructions exist that enable a process to communicate with another process over such a link or with another process on the same transputer. Each transputer has some on-chip memory (static RAM) and a memory interface to connect it to at most 4 Gbyte of external memory. The access time of the static RAM is 3 to 5 times smaller than the access time of the external memory.

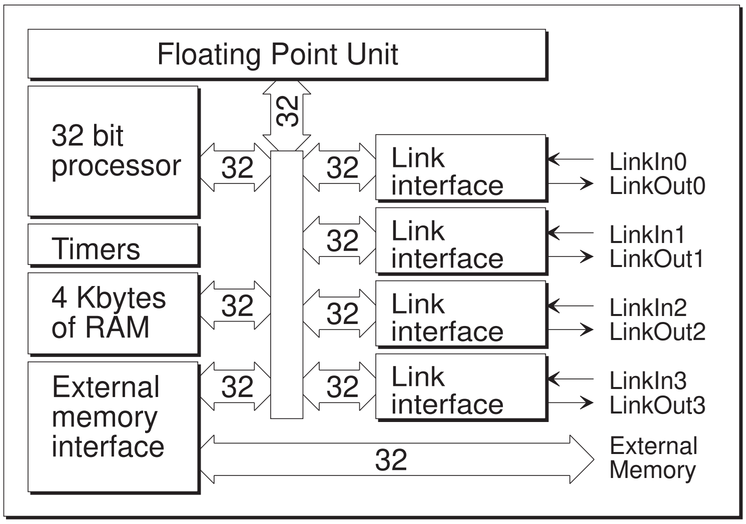

Figure 17.1 Architecture of the transputer processor.

Several types of transputer exist (T212, T414, T800 and T9000). The main differences between these processors can be found in speed, word size, instruction set and size of on-chip memory. The instruction set of newer processors is basically a superset of the older ones.

The transputer has only six registers. None of them is a general purpose register like the ones available in more traditional processors. The *workspace pointer* (wsp) is a register containing a pointer to some piece of memory which is called the workspace of a process. Addressing takes place relative to this workspace pointer. The *instruction pointer* (iptr) indicates the next instruction to be executed. The *operand register* (Operand) is used to construct large operands. Finally, there are three registers (called Areg, Breg and Creg) that form a tiny evaluation stack. These registers are the sources and destinations for most arithmetic and logical operations. Loading a value into this stack pushes B into C and A into B before the value is loaded into A. The contents of C are lost. Storing the value of A into memory pops B into A and C into B.

The value of *C* is left undefined when this happens. The contents of these registers are not saved when a context switch occurs. Only a few instructions exist at which a context switch is possible.

Some important transputer instructions: in the listed instructions '(expression)' stands for the *contents* of the address indicated by this expression. wordlength is the word length of the machine. Only the j instruction can cause a context switch.

```
ldl offset       ; load (wsp + offset * wordlength) into Areg
stl offset       ; store Areg into (wsp + offset * wordlength)
ldnl offset      ; replace Areg by (Areg + offset * wordlength)
stnl offset      ; store Breg into (Areg + offset * wordlength)
ldnlp offset     ; add offset * wordlength to Areg
ldpi             ; add iptr to Areg
dup              ; Creg := Breg, Breg := Areg
rev              ; swap Areg and Breg
mint             ; load –2^(wordlength–1) into Areg
ldc constant     ; load constant into Areg
adc constant     ; add constant to Areg
add              ; Areg := Breg + Areg,   Breg := Creg
sub              ; Areg := Breg – Areg,   Breg := Creg
mul              ; Areg := Breg * Areg,   Breg := Creg
and              ; Areg := Breg & Areg,   Breg := Creg
gt               ; Areg := Breg > Areg,   Breg := Creg
cj label         ; jump to label if Areg == 0
j label          ; jump to label, context switch possible
gcall            ; jump to Areg, Areg := return address
runp             ; start process of which Areg contains the wsp
stopp            ; stop current process
```

17.4 Representing PABC components on transputers

This section discusses a concrete realization of the PABC machine on a ParSyTec transputer architecture (Kesseler, 1991; 1992). The concrete network being used consisted of 32 T800s, each running at 25 MHz and having 4 Kbyte of static RAM and 4 Mbyte of external memory. The discussed implementation describes one of the many ways in which a concrete implementation can be obtained. A different transputer implementation of the PABC machine based on the ZAPP design (Burton and Sleep, 1981) can be found in Goldsmith *et al.* (1993).

17.4.1 Representing the reducer components

A PABC machine can conveniently be implemented on a concrete machine that has a reasonable number of general purpose registers.

Clearly, the transputer is not such a machine. To overcome this problem, the on-chip static RAM is regarded as a large collection of registers. Each reducer is implemented as a single low-priority transputer process with a private 'register' set, a fixed amount of memory of the on-chip static RAM: the **workspace frame**. The elements of this set are called the **locals** of a reducer. The workspace pointer of each reducer points to the start of these locals and can be used to identify each reducer uniquely (Figure 17.2). There can be locals to cache top values of the A- and B-/C-stacks, to remember the number of cached values (information used by the garbage collector) and to store stack pointers. Furthermore, two locals can be used to refer respectively to data that is different for each reducer and to data that is shared by all reducers.

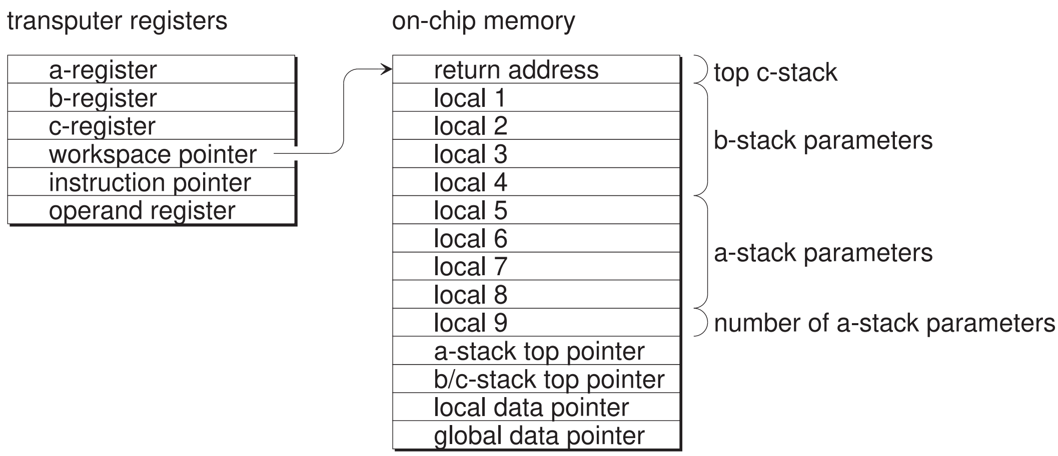

Figure 17.2 Layout of a workspace frame.

The program store and the program counter

The external memory and instruction pointer register are respectively used to represent the program store and the program counter. Into each external memory of the transputer a complete copy of the program is statically loaded when execution is initialized.

The A-stack, B-stack and C-stack

Two of the locals are used as a stack pointer for the A-stack, and the combined B-/C-stack. So each stack access requires an indirection, which is not very efficient. However, the locals provide enough spare registers for each reducer, such that in many cases stack access can be avoided (see Chapter 12). In practice, this solution gives good results.

All stacks are allocated in the heap and occupy one contiguous area of memory. The B- and the C-stacks are merged. A- and B-stacks grow in opposite direction such that their sizes can be checked at the same time by comparing the two stack pointers.

Each reducer gets small stacks initially. When necessary, a larger one (typically 1/8 larger) is allocated and the contents of the old stack are copied to the new one. The old stack then becomes garbage. Boundary checks cause a noticeable but not too serious overhead (worst case about 10%).

The N-queue and the waiting list register

The *N*-queue is stored in the heap. The waiting list register is a local.

The graph store

To store the graph each processor has a heap consisting of a contiguous piece of its external memory. The free space is a contiguous area within the heap: each reducer can reserve memory by taking the shared pointer to the first free word and advancing it the required number of words.

Representation of nodes

The node representation is basically the same as for sequential implementations, albeit that the waiting list has to be taken into account. For this purpose an extra field is added into the fixed part of the node in which a reference to a waiting list can be stored (see Figure 17.3).

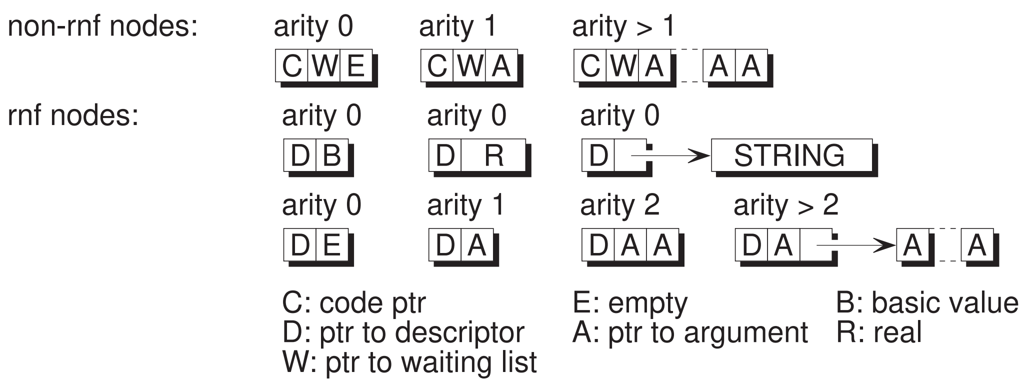

Figure 17.3 Chosen rnf and non-rnf node representations.

A positive field value `D` indicates that the node is in root normal form and a pointer to the descriptor can be computed by taking `–D`. A negative value `C` means the node is not in root normal form and `C` is a pointer to the code of the node (on the transputer the lowest address starts at $-2^{\text{wordlength} -1}$). Note that this only works as long as the transputer has no more than 2 Gbyte of memory. A pointer to a waiting list (Figure 17.4) is stored in a dedicated field of each non-root normal form node. As a consequence of this node representation a reducer has to check whether a node is in root normal form each time it tries to evaluate it.

Code for PABCs Jsr_eval (local 5 is assumed to refer to the node to reduce):

```
        ldl 5               ;   get pointer to node
        ldnl code           ;   get code/descriptor of node
        mint                ;   load hex 80000000
        and                 ;   0 if descriptor
        cj .is_in_rnf       ;   jump if descriptor
        check_stacks        ;   macro call, check if enough space on stacks
        save_               ;   macro call, save all locals in use on stacks
        ldl 5               ;   get pointer to node
        ldnl   code         ;   get evaluation code of node
        gcall               ;   evaluate the node
        restore_            ;   macro call, restore previously saved locals
.is_in_rnf
```

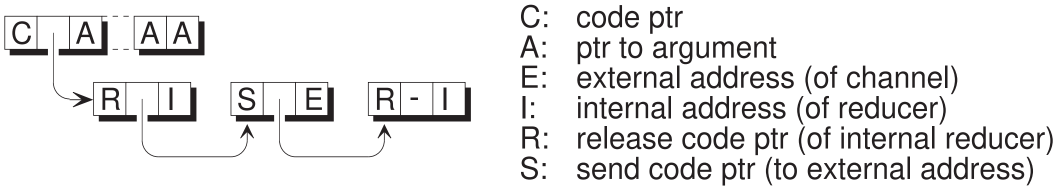

Figure 17.4 Representation of a waiting list.

Each element in the waiting list is implemented as an ordinary graph node with one argument. This argument is either the workspace pointer or the address of the remote channel node. The next element of the waiting list is stored in the waiting list field of each waiting list node, just as is the case for normal nodes. Depending on the argument, the evaluation code either puts a reducer back in the scheduling list of the transputer, or it sends the evaluated node to the requesting processor and updates the remote channel node with it. So all requests stored in a waiting list can be answered by evaluating each node in it.

The descriptor store

Each processor gets a copy of the representation of the descriptor store when program execution is initialized. The representation is conceptually the same as for sequential implementations.

17.4.2 The run-time system

A transputer has primitives for process creation, abortion and context switching. It has a hardware round robin scheduling mechanism. Reducers are automatically time-sliced by the transputer hardware when jump and loop end instructions are executed. However, this feature sometimes gets in the way at those places where context switches are not allowed. On the transputer there are basically two ways to prevent context switches. A process can run at high priority (high-priority processes are not time-sliced at all) or the use of those instructions that cause a context switch is totally avoided. One can replace those instructions by conditional jump instructions. In this way one can prevent other low-priority processes from getting a time slice. Interrupts from high-priority processes are still possible. Since it is relatively expensive to change the priority of a process the second solution is used where possible. In order to obtain a fair scheduling transputer `jump` instructions are used sometimes between adjacent basic blocks to jump across data.

Locally, an ordinary copying garbage collector is used. Globally, a weighted reference count algorithm is implemented. So cyclic structures across process boundaries are currently not removed. The program is aborted when a processor, in spite of garbage collection, runs out of memory space.

Indirection tables are used to store channel node addresses, global node addresses and a reference count. A stack is used for graph copying, since pointer reversal is relatively expensive. Currently, no load balancing information is taken into account to find a suitable processor for an arbitrary created parallel process. A simple pseudo-random generator is used. Running tasks or processes are not migrated to other processors.

17.4.3 Communication between processors

Each of the four transputer links provides two uni-directional links, one in each direction. So a fully connected network of transputers cannot be constructed with more than five transputers. However, the transputer hardware does not support the communication between transputers that have no direct link. So for the implementation of the parallel instructions of the PABC machine a complete communication system has to be provided including support for file I/O. None of the available standard communication software facilities could be used because they were not implemented reliably, efficiently or flexibly enough. Therefore, a dedicated communication system (a router) has been implemented directly on top of the low-level communication primitives of the transputer. The lowest level of the router is a communication layer that provides deadlock-free *routing* of packets to arbitrary destinations. These packets are not allowed to be larger than a certain maximum size. Messages larger than the maximum packet size are broken into a number of packets, which are sent separately. Each packet gets a number that indicates its position in the original message. The order in which packets arrive at the destination is not necessarily the same as the order in which they are sent. Packets cannot get lost.

The router employs a modified algorithm based upon the algorithm presented in Annot and Van Twist, (1987). It uses a fixed amount of buffers and a class-climbing algorithm to prevent *deadlock* under the assumption that each packet that reaches its destination is consumed within a finite amount of time. *Starvation* is avoided by a mechanism that locks out links from accessing buffers of a certain class if they recently used such a buffer and others are trying to access it as well. The algorithm does not use a fixed path to route packets, so *hot spots* in the network are avoided to some extent.

17.5 Generating transputer code and performance

Efficient sequential code generated for a single processor is the basis of an efficient parallel implementation. Despite the fact that the transputer architecture is quite different from the M680x0, the code generator presented in Chapter 12 can successfully be retargeted to transputer code. There are some small differences because the transputer has only six registers (but it has an on-chip memory) and a small evaluation stack. Also some adaptations are needed for dynamic stack allocations. However, with a reasonable amount of locals assigned to each process most of the optimizations discussed in Chapter 12 remain applicable.

Example of generated code. Consider again (see also Section 12.5.2):

```
:: F !INT !INT    ->    INT;
   F a b          ->    - (* a b) (+ a 3);
```

The generated Motorola code for this rule alternative:

```
muls.l  d1, d0        ; multiply a and b
addq.l  #3, d1        ; add a and 3
sub.l d1, d0          ; subtract both results
rts                   ; return to caller
```

The corresponding transputer code:

```
ldl 1                 ; push b on evaluation stack
ldl 2                 ; push a on evaluation stack
mul                   ; multiply a and b
ldl 2                 ; push a on evaluation stack
adc 3                 ; add a and 3
sub                   ; subtract both results
stl 1                 ; store result on top of B-stack
ldl 0                 ; get return address
gcall                 ; return to caller
```

This may seem a bit long, but the code actually occupies only 10 bytes of memory. The top of the B-stack is cached in local 2, the second entry resides in local 1. The caller expects the result to be returned in local 1. On entry to this function the return address is stored in local 0. No stack checks are needed since only locals are used.

Performance

Table 17.1 shows some performance results (see also Table 12.3).

nfib 30	each task computes nfib 30.
sieve	10 000 primes, one task for every 20 prime numbers.
queens	counts all solutions for the (10) queens problem.
rnfib 30	each task computes nfib 30.
fastfourier	on an array of 8000 complex numbers.
mandelbrot	calculates Mandelbrot fractals, resolution 530 x 320, depth 128.

Table 17.1 Performance in seconds on respectively 1, 2, 4, 8, 16 and 32 T800 processors of some well-known benchmarks on 32 processors connected in a grid topology.

Program	*Number of processors*					
	1	*2*	*4*	*8*	*16*	*32*
nfib	12.2	6.5	3.5	2.2	1.4	1.1
sieve	19.4	31.2	32.1	23.6	16.9	14.5
queens	47.9	28.5	15.1	9.0	6.2	4.7
rnfib	23.7	12.2	7.1	3.9	2.2	1.6
fastfourier	13.8	11.2	8.7	6.3	5.6	5.6
mandelbrot	147.0	91.0	54.3	34.3	18.2	10.6

The main difference from the sequential implementation discussed in Chapter 12 is that stacks are handled less efficiently in the transputer version. In the Motorola implementation, stack frames are allocated once outside the heap. In the transputer implementation programs are started with an initial stack size of 0 bytes in the heap. The stacks are automatically reallocated and resized, implying that boundary checks have to be performed. So the Motorola programs actually use more memory than their transputer counterpart because their A-, B- and C-stacks are allocated outside the heap. A disadvantage of a T800 is that the processor needs quite some time to fetch instructions and data from off-chip memory. If a program is placed in the 4K on-chip memory (only the code, while keeping the data in the same place) it runs about 1.6 times faster. But, of course, most useful programs are larger than 4K. A T800 produces good results for computations with reals due to the speed of the on-chip floating-point processor.

The parallel results are obtained with medium-sized randomly distributed tasks. A small task size degrades performance by introducing a relatively large communication and process management overhead. Large tasks limit the total number of tasks, thereby increasing the chance that a bad load distribution will occur.

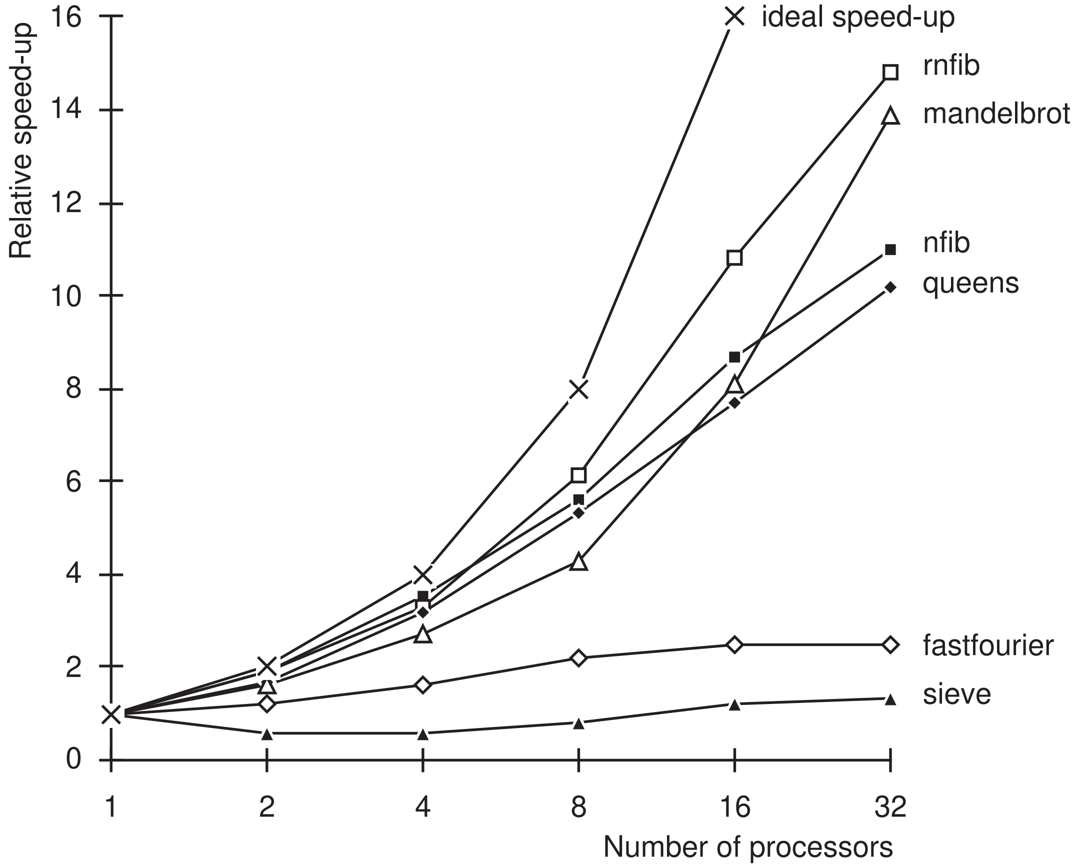

Figure 17.5 A graphical representation of the speed-up obtained for the different test programs.

Figure 17.5 shows that some examples (like sieve and fastfourier) do not have a lot of inherent parallelism, although a lot of processes are generated. The communication overhead is inherently too large for these examples. They serve as an exercise in expressing certain process topologies, but not as examples of good speed-ups. The other examples show a reasonable performance considering that a random process distribution was used.

Further code optimizations

The code as tested (see Table 17.1) can be improved in several ways. Many small improvements for the sequential code (see Chapter 12) are not yet implemented. Besides these standard optimizations for sequential code there are also many possible improvements related to the parallel parts of the machine. Certain PABC instructions that involve communication between processors often appear in a certain order. It is sometimes possible to combine PABC instructions to reduce the amount of communication between these processors. Furthermore, the initialization of a new process can be done more cheaply by reusing the stacks of a previously stopped process. Processes are created randomly. No heuristic for workload balancing is being used. A lazy copy sent across a network can be represented more compactly (Van Groningen, 1992). Process communication and synchronization as well as scheduling can be automatically tuned to optimize the communication between a producing process and a consuming process. All these optimizations are subjects of future research.

Summary

- The sequential implementation technique presented in this book can also successfully be applied on a transputer processor, even though its architecture is quite different from the more traditional architectures such as those offered by a Motorola processor.
- The PABC machine abstracts from many implementation problems. Therefore, the realization of the PABC machine is considerably more work than the realization of the ABC machine. The actual effort will depend heavily on the facilities offered by the concrete MIMD machine hardware and software. Typical features that have to be solved in one way or another include *stack checking*, *stack reallocation*, *local* and *global garbage collection*, *removal of garbage reducers*, *context switching*, deadlock and starvation-free efficient *routing of messages* between arbitrary processors, *load balancing*, *deadlock detection*, *process migration* and *I/O support.*
- Most of the items on the previous list also have to be addressed when other kinds of programming languages (imperative, logical, object oriented) are to be implemented on an MIMD architecture.
- The PABC machine has been realized on a 32 node ParSyTec transputer system. Although a transputer does not have a large set of general purpose registers, the actual implementation on one transputer performs reasonably well.
- MIMD architectures are generally quite expensive. An implementation of Concurrent Clean (and the PABC machine) is also available for a, generally less expensive, heterogeneous network (AppleTalk/Ethernet) of Macintosh and Sun workstations. This will offer the opportunity to use concurrent functional programming not only for gaining speed but also for writing *distributed interactive multi-user applications* (e.g. spreadsheets, databases) combining Clean's event I/O with process annotations.

EXERCISES

17.1 Use the code generator to generate parallel code for the programs of Exercises 15.2 and 15.3. Generate parallel code for one machine. Measure the differences between parallel code and pure sequential code. If it is possible on your system, measure the speed-ups on as many distributed processors as are available. Also measure the communication costs. Compare the actual results with the results predicted by the simulator. Explain the differences.

17.2 Measure the `nfib` example using different allocation strategies and topologies. Does the chosen topology make much difference? Does the chosen allocation strategy make much difference? Explain the results.

17.3* Assume that you would have used `{Par}` and `{Self}` annotations instead of `{P}` and `{I}`. Would the programs of Exercise 17.1 perform better, worse or the same?

17.4 Assume that you would have used a shared memory architecture instead of a distributed memory architecture. What would be the expected performance compared with distributed implementations?