COMMUNICATION ISSUES REGARDING PARALLEL FUNCTIONAL GRAPH REWRITING

Marco Kesseler
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
E-mail: marcok@cs.kun.nl
September 1992

Abstract

Efficient communication is the basis of an implementation of a parallel language on a distributed machine architecture. This article handles about the communication mechanism that is needed to support parallel graph rewriting in Concurrent Clean. The danger of deadlock is imminent at several levels of such a system. Communication problems and their solutions are closely related to memory management issues, garbage collection in particular. In fact the communication mechanism presented here is a global memory management system. It is part of a distributed runtime system that has been moulded to fit parallel graph rewriting as implemented in Concurrent Clean.

Introduction

Parallel reduction of graphs in a distributed architecture requires extensive communication between processors. In this paper we'll take a look at communication issues in Concurrent Clean. Despite differences between various parallel architectures a number of common communication problems can be distinguished. A general communication mechanism will be presented to deal with a number of these problems.

Concurrent Clean is translated to abstract PABC code. First we'll resume most primitives for parallel execution as defined in this abstract machine. We'll refer to this machine throughout this article. Next, a number of important communication issues will be determined. The rest of the article will deal with these in more detail.

PABC primitives for parallelism

Most PABC instructions influence sequential execution only. Only a few PABC instructions are used for parallelism. These parallel ones can be divided into two groups: instructions that influence the current processor only and instructions that require communication with other processors. The first group performs tasks that are basically needed for internal parallelism, i.e. interleaved execution of processes. With these it is possible to create, suspend, stop and wake up processes on the current processor. The other group is used to create reducers on other processors and to send graphs from one processor to another.

Internal parallelism

The main instructions to support interleaved execution of processes are: new_int_reducer, stop_reducer, suspend, copy_graph, set_wait, getWL and release. A short informal description of these instructions will be given below.

new_int_reducer start_label a_offset -> creates a new reducer running interleaved on the same processor. It starts at the start label and its argument is the graph at the given offset in the a-stack.

stop_reducer -> stops the current reducer (it becomes garbage)

suspend -> stop the current reducer temporary.

set_wait a_offset -> put 'yourself' (= the reducer currently executing) in the waiting list of the given node.

copy_graph a_offset -> duplicate the given graph in the heap and leave a reference to it on top of the a-stack

getWL a_offset -> get the waiting list of the given node and keep it in some 'special purpose register'.

release -> wake up all reducers in the waiting list that has been obtained by getWL.

It takes three abstract PABC instructions to update a node that has a waiting list:

```
getWL      node
fill       node ...
release
```

The basic reduction machinery for interleaved graph reduction works as follows. Each reducer is started up with some special reduction code that determines how the reducer behaves. At this moment only one kind of reducer is supported: RNF reducers. As the name suggests, these reducers reduce a graph to RNF. They are started with this code:

```
_rnf_reducer:
   jsr_eval                     || reduce node on top of stack to RNF
   pop_a        1               || pop result
   stop_reducer                 || and die
```

Where possible, reducers share graphs (to be clear: copy_graph is NOT used in this context). If a reducer starts to reduce a node it reserves this node by putting the special reduction code '_reserve' in it. The reserve code is a predefined piece of PABC code that looks as follows:

```
_reserve:
    set_wait     0              || put yourself in waitlist in node
    suspend
    rtn                         || node is in RNF, return to caller
```

This means that any reducer that subsequently tries to reduce the node will suspend itself after having put itself in the waiting list of the node. The reducer that is actually reducing the node will eventually overwrite the node with a RNF node, while releasing all suspended reducers.

external parallelism

Real parallelism is achieved by means of a small set of powerful instructions. On one hand there are instructions to send graphs and start up new reducers on remote processors, of which create_channel, send_graph and send_request are most important. On the other hand some instructions allow a process to get information about network topology (and, in the future, about processor load as well). Of the latter only some simple ones are supported at this moment.

create_channel code_label -> create a channel node to the processor of which the id is on top of the b-stack. The new channel node must be left on top of the a-stack. Its evaluation address should be set to the given label.

send_graph start_code graph_offset channel_offset -> send the given graph over the given channel and start a new (remote) reducer on it. This reducer should be started at the given start code.

send_request channel_offset -> send a request over the given channel.

<< examples of instructions for process placement >>

randomP -> push a random processor id on the b-stack

ItoP -> convert integer to processor id

Starting a new remote reducer involves two steps. It starts by creating a channel to the processor that has been chosen to accommodate for the new one. After achieving this, a start-up code and a copy of some graph are sent over the channel. These are the arguments of a 'new_int_reducer' instruction that will automatically be executed at the remote processor.

The result of a graph that is being reduced at another processor is never returned, unless a request for it has been received for it. Requests are sent whenever a reducer evaluates a channel node, which means it needs the result of the graph the channel node points to. Each channel node holds the following code:

```
_channel_code:
    push_node     _reserve 0                || reserve channel
    setwait       0                         || put yourself in waiting list
    send_request  0
    suspend
    rtn
```

When one or more requests have been received for a graph it will be copied back to each requesting process as soon as it has been reduced to RNF. The channel nodes that were used for the request will be overwritten by the requested graph and all reducers waiting for it are woken up.

Although easy to understand, these instructions are substantially more difficult to implement than the ones for sequential execution. Below we'll take a look at problems that are related to the implementation of these instructions.

Communication issues

Important issues that have to be addressed when dealing with communication networks are speed and reliability. In particular deadlocks and starvation have to be avoided. The way this is achieved best, depends on the demands that are made on the network. These are very diverse for arbitrary functional programs however.

Graph copying

Graphs need to be copied in Concurrent Clean. To allow graphs to be sent from one processor to another it should be possible build concise representations of graphs to be used by the communication system. This introduces the following problems.

Each node contains code addresses, references to other nodes and possibly a waiting list. When a node is copied to another processor these references change. Code may not be located at the same location for different processors. Nodes that are locally accessible for the original graph may not be for the copy and vice versa.

When a graph is sent away, as much nodes as possible should be sent in one go, for obvious performance reasons. Additionally, Concurrent Clean allows cycles and shared nodes. These should be preserved when a graph is copied to another processor. Duplication of work should be avoided as much as possible.

communication pattern

It is difficult to predict in what way Concurrent Clean programs will use the communication network. The amount of processes and their placement varies considerably for different programs. The same is true for the amount of communication each process needs.

A lot of messages are made out of graphs and their size varies notably as a result of this. In theory it might take anything between a few bytes and the entire memory space of a processor to send a graph. In reality sizes between some bytes and several Kilobytes are common. Graph messages are dispersed by small ones, such as requests and messages to create channels. There's no general pattern for the routes and distances these messages travel.

This means that the basic communication mechanism that is used to implement the communication primitives of the PABC machine must be very general. It should at least provide a (virtual) fully connected network.

Memory limitations

With unlimited memory, deadlock-free communication would be trivial to implement. Apart from deadlock problems when implementing the virtual fully connected network mentioned above there are several additional communication problems in a functional language that are related to memory management, in particular garbage collection.

It may not be possible to handle a message at the moment it arrives. For example, if a graph is received when a garbage collection is going on, the heap is in an inconsistent state and the graph cannot be stored. It is not sufficient to simply buffer these messages at the destination as buffer overflows are unavoidable at some point. The return of error messages is equally dangerous as this requires buffer space as well (at the processor that generates the messages), whereas time-outs on messages would introduce other problems (e.g. unnecessary re-transmission of messages, tuning of time-outs for different network sizes).

Garbage collections move nodes. This applies to nodes that are being sent/received and nodes that have external references to them. The communication mechanism must be able to keep track of those.

Virtual fully connected network

As stated above the basic communication system should be very general. A (virtual) fully connected network should at least be provided. At the same time this system must be kept as simple as possible as this allows for easy implementation on a variety of parallel systems.

At the lowest level it isn't essential messages can have any size. If packets of some maximum size can be sent from one processor to another, it is easy to implement arbitrary sized messages on top of this. To keep the system as simple as possible and for reasons that will be explained later on (see message relocation problems) our basic communication mechanism will only provide communication of packets that are not larger than some maximum size.

As we'll see, the order in which packets arrive at the destination isn't important for normal distributed graph rewriting. On the other hand it is very useful for some other uses to maintain the ordering of packets. For instance, if packets arrive out of order, IO routines would require extensive buffering in certain situations (and speed isn't extremely important in this case). For this reason it should at least be possible to force packets to arrive in the same order as they have been sent. The 'random order' arrival of packets is not mandatory for correct execution of programs, so this is optional. It should be added when it is clear this allows for faster communications. The same can be said for broadcasting of packets. At this moment broadcasts aren't useful in the Concurrent Clean system, but this will probably change in future.

There are two ways to handle incoming packets. First of all one could put all incoming packets in some queue and enable processes to get packets out of this queue explicitly. On the other hand it's possible to assign some specific packet handling (interrupt) routine to each kind of packet that will be executed whenever a packet arrives. Both are equally powerful, but the latter doesn't suffer from inherent buffering problems and offers a very direct way of specifying what has to be done when a packet arrives. This means that incoming packets are regarded as events. Each event invokes a special event-handling routine and passes the event to this routine.

All this can be translated in a small number of functions that should be offered by the basic communication mechanism. None of these routines should ever fail. Once a packet has been accepted by the system, it may be considered sent, although it may not have arrived yet. Packets shouldn't get lost ever.

```
<< this is an incomplete specification of the routines provided by the
   lowest level of the communication mechanism >>

TYPE

PACKET        -> (PACKET_KIND,CONTENTS);
PACKET_ACTION -> (PACKET STATE -> STATE);

RULE

<< a send_packet routine for point to point communication of a single
   packet. The ordering requirement is >>

:: send_packet DESTINATION PACKET BOOL STATE -> STATE;
:: send_packet destination packet order state
   -> send_and_maintain_order dest packet state, IF order
   -> send_in_arbitrary_order dest packet state;

<< a broadcast routine to send a packet to all processors >>

:: broadcast PACKET STATE -> STATE;

<< a routine to set the routine to be called at arrival of a certain
   kind of packet.  >>

:: set_packet_action PACKET_KIND PACKET_ACTION STATE -> STATE;
```

The use of sets of 'packet actions' offers useful possibilities to easily hook monitoring routines to the reception of certain packets. A 'debugging'-mode or a 'garbage collection proceeding'-mode can be entered easily by switching an entire set at once.

The amount of work needed to implement this obviously depends on the communication primitives that are provided by the hardware and/or operating system. For the new Connection Machine (CM5) a suitable communication mechanism is readily available, as will be the case for the T9000 transputer if it is used in conjunction with some router chips. For architectures that use a bus, all this is easy to implement. The T800 transputer on the other hand provides nearest neighbour communication primitives only. The C40 of Texas Instruments has much resemblance to the T800 in this respect.

Note that it may be beneficial to copy graphs in (virtual) shared memory architectures as well, as copying can be used to improve the locality of a graph. In these systems communication of packets (or messages) would take a single (global) block move. On the other hand a global block move can be implemented with the primitives above in a trivial way, as clearly these notions are equivalent.

T800 transputer implementation

At Nijmegen a Supercluster of 64 T800 transputers is available. It runs Helios. Unfortunately the Helios Programming philosophy is quite different from that of Concurrent Clean and this is reflected in its message passing system (messages might be thrown away as a result of congestion). Because of this and historical reasons Helios has not been used for the current implementation. Instead, we have implemented a custom packet router.

The algorithm that has been used to implement this router is based on the one presented in (Annot et al (1987)). It prevents deadlocks, provided that each packet that reaches its destination is consumed within a finite amount of time. To accomplish this a fixed amount of buffers and a class-climbing algorithm is used. 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. It doesn't use a fixed path to rout packets, so hot spots in the network are avoided to some extend.

The original algorithm has been designed for a special router processor. It cannot be used for a network of transputers unmodified. I’ll give a *very* short description of it below. The complete algorithm can be found in (Annot et al (1987)).

original algorithm

In the original algorithm each communication processor has a number of connections to other communication processors. Each connection consists of 2 physical links (wires) and each processor has two processes per connection: an input process and an output process. Each output process is connected to an input process on another processor via a dedicated link. Every link is able to carry messages in either direction (but not in both directions at the same time). The processes continuously perform the following actions:

Input Process	**Output Process**
reserve a buffer of the lowest class possible	
request a packet of the reserved class	receive request
	find packet of requested (or higher) class
receive packet or CANCEL	send packet or CANCEL

The input process isn't blocked ever: when it receives a CANCEL it goes on requesting and when it can't reserve a buffer it starts polling. As a consequence of this the output process isn’t descheduled either.

transputer modifications

There are two problems when implementing this algorithm for a network of transputers:

- On the transputer the router processes should run at high priority to achieve a good response time when packets are routed (as noted in the ZAPP experiments (Goldsmith et al (1991)). In the original algorithm this would mean that low priority processes (reducers) wouldn’t be allowed to run (as the router processes are continuously running).

- Although each transputer connection consists of two separate links (wires), these links cannot operate independently and a single link cannot send messages in both directions. If an input process sends a request and an output process (at the same transputer) sends a packet over the same link at the same time the transputer link hardware gets confused.

The original algorithm has been changed such that requests of input processes and packets of output processes (at the same transputer) for a certain link are multiplexed over that link. They are combined in a single message if possible. Furthermore each input process gets suspended as soon as it gets a CANCEL message or when there are no buffers available. It is woken up again (by means of a SIGNAL message) as soon as the other side gets a packet that matches the request that has been cancelled, or when the input process is able to send a request for a packet of a lower class than the one that has been cancelled before (when buffers of lower priority become available).

The algorithm has been optimised for a high packet load. If the frequency packets are offered is low, the overhead per packet is high. In worst case four (small) messages are needed in addition to the packet itself (SIGNAL-REQUEST-PACKET-REQUEST-CANCEL). Overhead is smallest when a link continuously sends packets in both directions simultaneously. In this case request messages are piggy backed on packets and no extra messages are needed.

The router tries to suppress situations where all buffers are in use. Such situations slow down the router as they decrease the number of links that can be used simultaneously (and bidirectionally), resulting in a decreased throughput. One can picture such a situation as each processor having only one 'hole' to receive a packet in (instead of at least one per link). At any moment in time only one neighbour can fill such a hole, thereby transferring the hole to itself. If no measurements are taken, this situation might persist for a very long time. A slowdown of a factor of 2 has actually been measured during tests of an early version on a ring where this situation was created artificially. Due to (a bit too rigid) fairness administration in combination with a heavy load it didn't resolve at all. This problem has been solved by relating fairness measurements to the amount of free buffers. when a lot of buffers are in use at some processor new packets are accepted less frequently, in favour of through-routing and delivery of packets at this processor. This increases the overall amount of routing-holes.

The current implementation investigates the network topology during start-up and builds a table of all shortest paths. Packets are routed over one of these paths. Given a destination and an intermediate processor it is possible to use a fixed link to forward a packet or the one that is ready first. The latter is more efficient in general while the former enforces packets to arrive in the same order they have been sent. The current implementation doesn't handle packet broadcasts yet.

The resulting algorithm is quite general. It should work on any network of processors that are connected by bidirectional links.

Low level Concurrent Clean messages

The primitives presented above are used to allow arbitrary processors to communicate. A general way to enable processes to transmit messages of arbitrary size will be presented here. This communication mechanism is only accessible by some PABC instructions mentioned above. There's no way to send messages explicitly in Concurrent Clean.

messages and queues

Messages will be stored in the heap to allow reducers to construct messages easily. This eliminates the need to implement separate message buffers as well. Each message will be represented by a number of standard nodes, so it can be carried around easily on the a-stack of each reducer without any danger of messages being destroyed by the garbage collector.

Small messages can be sent as a single packet. Large ones have to be split up in a number of packets. To allow this each large message is divided into a header and a body as depicted below (see routing conventions). The header contains routing information and the body contains the actual data to be sent. As message bodies consist of ordinary string nodes they may be shared. In this way it is easy to send large messages to several processors without having to duplicate them in the heap. Strings can be sent without copying them in the heap: only a header needs to be built. At the moment large messages are needed only to send graphs or for IO involving strings.

```
TYPE

SMALL_MESSAGE -> STRING;
LARGE_MESSAGE -> (HEADER,BODY);

HEADER        -> STRING;
BODY          -> STRING;
```

A message can be sent by appending it to a special list that is maintained by the router. This list consists of normal list constructors (Cons and Nil) and contains references to messages in the heap. Note that the type of elements in this list may differ. When a reducer has put a message in this list, it may assume the router will deliver the message eventually and thus it may proceed. As the message itself is in normal form there's no danger it will be overwritten. The garbage collector takes care the message queue isn't deleted which means the queue is subject to relocation. When the heap becomes filled and a single garbage collection doesn't free enough memory, reducers simply wait until following collections free more memory.

routing conventions

The router splits up large messages. It divides the contents of the message body into m parts ($p_0..p_{m-1}$) each part smaller than max, where max = max packet size - header size - word size, and m as small as possible. For each part p_i the router builds a packet that contains the header, the position of p_i within the message and part p_i itself. Those packets are sent over the network separately.

Each message should obey a few conventions. First of all the size of a small message shouldn't exceed packet size. A header should be smaller than packet size. Apart from this, a reducer is completely free to choose sizes for header, body and (small) messages. In order to be routed properly each message has to contain the destination address and the message kind. The first 4 bytes of each message (or header) are reserved for the message kind whereas the next 4 bytes should contain the destination. Note that the size of each message is stored implicitly in the string node. The rest of each message can be anything a reducer wishes.

message relocation problems

Messages may be relocated due to a garbage collection. This introduces some problems. A possible solution would be to treat messages differently during garbage collections and keep them at a fixed location. This would complicate a compacting garbage collector (both copying and mark-sweep) notably. Instead we've chosen to allow relocation of messages and deal with the problems related to this.

The contents of a message isn't adjusted in any way when it is relocated (it is a string). This is no problem as a message shouldn't contain references to local objects (either addresses or offsets), except from offsets to fields within the message itself. Clearly direct references to objects at the source processor have no meaning at the destination, so there's no use for them inside a message.

The transputer links operate independently of the CPU: communication is a divisible action, which will be the case for many other distributed systems as well. This means a garbage collection might occur while a message is being sent. This would clearly be disastrous if the communication hardware would read (or write) from the heap itself. For this reason the low level communication mechanism has to use private non-relocatable buffers not only for through-routing of packets, but also to send (and receive) messages that are stored in the heap. If variable sized buffers would be used for this, memory management would get considerably more difficult (if possible at all without limiting message size). It is better to use fixed size packet buffers instead and copy only one packet at a time (for the transputer implementation the normal router buffers are used for this). In other words: when transmission of a message is divisible there is no use for variable sized messages at the lowest level of communication.

Channels

On PABC level all communication takes place via channel nodes. These are special indirection nodes that point to nodes that are located on remote processors. Channel nodes are created explicitly at runtime by means of a create_channel instruction. They are destroyed when they are overwritten by a RNF node (after issuing a request) or implicitly by the garbage collector. During the lifetime of a channel node it is used to send graphs, start up new reducers and to request results.

indirection table

In addition to channel nodes the actual implementation of channels uses indirection tables. Each processor contains one such indirection table. It is an array that holds references to nodes in the heap that have a channel node pointing to it. In fact this is the same as the request nodes in the PABC simulator or the 'input table' mentioned in (Kingdon et al. (1991)), whereas the set of channel nodes on a single processor can be seen as the 'output table'. The remote addresses in each channel node consist of two parts: a processor id and an index in the indirection table. This allows the garbage collector to move nodes around in the heap without having to update channel nodes pointing to them. Only the indirection table has to be adjusted.

The indirection table itself is located in the heap as well. When it turns out to be too small to hold all references, a larger one is allocated and the old one becomes garbage, much like the reallocation of stacks. It will be clear the table cannot shrink beyond the highest index in use. Therefore references are always allocated as low as possible within the table.

Apart from providing a way to implement global addresses the indirection table is used for various administrative tasks. As we'll see it is used for garbage collection, reservation of memory at remote processors, buffering of copied graphs and for administration of requests.

garbage collection

As inter-processor references are composed of channel nodes and indirection tables these play an important role with respect to garbage collection. We'll briefly indicate in what way. Garbage collection is not the main issue of this article, however.

Local garbage is collected by an ordinary copying (or mark-sweep) garbage collector. A weighted reference counting garbage collector is used on inter-processor references (Bevan (1987), Watson & Watson (1987)). Inter-processor cycles will be removed in a different way (this hasn't been solved yet). Weighted reference counts prevent racing conditions between messages to increase or decrease a reference count: reference counts need only be decreased in this scheme. Channel nodes and indirection tables are used as follows to implement this. All channel nodes on a single processor are linked to each other and they have a weight. Every element in the indirection table contains a reference count.

Reference counts are decremented in two ways. First of all, after a local garbage collection has taken place, the list of channel nodes is examined. For each channel node that has not been visited by the collector a decrement reference count message is sent to the graph it points to (and it's removed from the list). In addition to this, reference counts are decremented automatically when a request is sent. As soon as the reference count of a table entry becomes zero, the corresponding graph becomes garbage and the table entry is freed to be reused for creation of channels.

Weighted reference counting is being implemented at the moment.

Copying of graphs

Graphs can be copied, which means it should be possible to duplicate graphs without loss of sharing. Duplication of work should be avoided as much as possible. Two distinct situations can be distinguished. First of all a graph can be copied locally, that is, the copy of some graph will be built in the same heap as the original. Secondly a graph might be copied globally. The latter case requires the copy to be built in a remote heap. We'll first look at some node-attributes to help prevent duplication of work: the defer attribute and the copy attribute. For a detailed description I refer to (van Eekelen (1990)).

Deferred nodes

Each closure can be 'deferred'. For RNF nodes this attribute has no meaning. It indicates that the node may not be copied. By deferring a node it gets glued to the current processor, until it becomes updated by a RNF node. The Clean compiler defers certain nodes automatically, in particular the root node of each new reducer. Note the possibility of deadlock when a deferred node will not be reduced by some internal reducer and it is referenced from another processor.

The defer attribute can either be implemented by a special 'Defer' indirection node, or by some flag in the node itself. In the current implementation the latter solution has been chosen: deferred nodes have a special node entry that jumps to the normal entry. The copying algorithm distinguishes these entries by looking at a descriptor (arity etc.) that is stored directly before each node entry.

Copy nodes

All nodes may have a copy attribute. It means that after such a node has been evaluated, a copy should be made of the result. This copy will then be used by the reducer that evaluated the node. Copy nodes are implemented as special indirection nodes. The evaluation code of such a node looks as follows.

```
_copy_code:
       push_node  _reserve 1        ;push argument and reserve node
       jsr_eval                     ;evaluate argument
       copy_graph 0                 ;copy result and push on a-stack
       getWL      2                 ;get waiting list of copy node
       fill_a     0 2               ;update copy node with copied result
       release                      ;release waiting list of copy node
       pop_a      2
       rtn
```

Note the correspondence between channel nodes and copy nodes. Channel nodes can be seen as global copy nodes.

Local copying

Copying a graph is quite a large task. The structure of the original graph should be preserved in the copy, while the original graph shouldn't be affected in any way by the copying algorithm. This section will show how this can be achieved.

The copying algorithm presented here avoids duplication of work to some extend by treating channel nodes, copy nodes, deferred nodes and reserved nodes in a special way. Copying stops at each of these nodes. For channel nodes and copy nodes this is all that happens: the copy is just a new channel/copy node. The copy of a deferred/reserved node consists of a new copy node that points to the original deferred/reserved node. In this way the copying of these nodes is deferred. Note that not all duplication of work can be avoided this way: if a closure is shared among two (or more) reducers and it isn't deferred (nor reserved), copying this node will introduce duplication of work if the result is needed by both processes.

Several ways exist to implement copying algorithms that preserve sharing and there's a clear trade-off between space requirements and speed. We've chosen to favour speed (see description below). We use forwarding addresses to detect sharing. This means the original graph becomes contaminated by these addresses. Consequence of this is that the original graph must be restored afterwards Thus, two phases are needed.

The first phase builds a copy of some graph. The graph is traversed and copied in the same way a copying garbage collector does: by subsequently copying the argument nodes of a formerly copied node (scavenging) until no unscavenged nodes are found. When a node is encountered for the very first time, it is copied to the first free byte in the heap. A forwarding address is then put in the original node, while the address of the original is pushed on a special purpose stack at the end of the heap. The second time a node is encountered the forwarding address is used and nothing is pushed on the stack. Special nodes at which copying should stop are copied as well, but their arguments are not. They will be adjusted during the second phase.

After the first phase has taken place the original graph has been copied completely. However: the original graph is messed up by forwarding addresses and some of the copied nodes have to be replaced by copy nodes. The second phase will solve this. It uses the stack at the end of the heap to find all original nodes. The forwarding addresses in the original nodes lead to the corresponding copies and thus to the correct contents of the originals. After a node has been restored, the copy is replaced by a copy node to the original if necessary. After having restored all nodes, the stack is deleted.

When not enough memory is available during copying the garbage collector should be invoked. This is not possible without restoring the partially copied graph first. Fortunately this doesn't require any additional memory at all, so this imposes no problem. If the garbage collector succeeds to free enough memory copying may 'proceed', which means it should start all over again. This doesn't happen often though, so this overhead no serious drawback.

Clearly the stack imposes some spatial overhead. On the other hand it allows the original graph to be restored quickly. Note that all nodes of the copy occupy a contiguous piece of the memory until the next garbage collection takes place. This property will be used to implement global copying of graphs.

Global copying

Global copying should have the same effect as local copying. The 'only' difference is the eventual placement of the copy. Like the local case, sharing should be preserved and in addition to this as much nodes as possible should be sent at once. As mentioned above the local copying algorithm delivers a copy that occupies a contiguous area within the heap. We will use this property to send graphs to other processors. The following steps are needed.

temporary copy

Before a graph is sent, it is copied locally. In this way all nodes that have to be sent are gathered and stored in a format that contains exact information about the structure of the original graph. There is one difference with the original copying algorithm however: copy nodes, deferred nodes and reserved nodes aren't replaced by copy nodes during the second phase of the copying, but by channel nodes. These channel nodes point to the original nodes and are sent to the destination processor, so it can access nodes of which copying (sending) has been delayed. channel nodes don't have to be replaced of course (note that by sending an existing channel node, it may get to point to a locally accessible node).

In contrast of the first restoration routine, this one might require some memory: Every time a copy is overwritten by a channel node, a corresponding entry in the indirection table has to be built as well. This may require reallocation of the table and thus a garbage collection could be invoked. Even if a node already has an entry pointing to it a new one should be created. This has to do with weighted reference counting: the reference count of a single table entry cannot be increased. Instead several table entries are allowed to point to the same node. It must be possible to retrace all entries that point to a node using nothing but the node itself (see 'transmission and reception' and 'requests' below). For this reason all entries pointing to the same node are linked, assuming the node itself has a way to find one of them.

packing

Having made a local copy, transmission of the graph becomes relatively easy to implement. But before this is done, the graph must packed for shipping. First of all the copy is put in a string, so all nodes stay together during possible garbage collections.

To make this graph representation relocatable two addresses are included in the string. First of all the address of the copied root node within the string is included: now every node pointer within the string can be transformed into an offset to the root node when the string is unpacked. Secondly, the address of some label is included that is defined on all processors. This allows code addresses (node entries) to be ported between processors provided every processor contains the same code image (which is the case in the current transputer implementation). Exact code addresses may differ among different processors, but the offsets between two labels should remain the same. The code address that is included in the message acts as a reference code: by comparing the remote address to the corresponding local one it is easy to determine the constant that has to be added to each code address.

For some implementations, it may not be possible to load the same code on every processor. In this case the following will be done. Instead of sending node entries, the descriptor id of each node is sent as suggested by John van Groningen (van Groningen (1992)). To use this method each descriptor has to contain the (machine specific) entry code for 'its' node. With the help of a suitable linker identical descriptor tables can be built for different executables. This means the same descriptor id's can be used on different processors and these id's are used to translate code addresses. This offers the possibility to run parallel programs on a network consisting of different processors.

Instead of the graph representation presented above, it's possible to use a more compact representation. It suppresses the need for pointers within the global graph representation. In some situations this can decrease communication costs notably. For this I refer to (van Groningen (1992)). It hasn't been implemented yet for the transputer version.

transmission and reception

It is easy to start sending a graph to another processor once it has been packed in a string node. One only has to use the primitives presented earlier. This will split the graph into packets, and is likely to scatter these over the network. No assumption may be made about the order in which packets arrive. Apart from this, packets of different graphs may arrive interleaved and reducers will probably allocate memory between arrival of two packets, possibly using a garbage collection.

This means it is not entirely trivial to store all packets that compose a single graph in a correct way. One might link non-neighbouring packets and use a sophisticated algorithm to unpack the graph, but this is bothersome (and expensive), especially if it has to deal with variable sized nodes and strings. Therefore we'll ensure all packets that belong to a single graph are stored one after the other in the destination heap. This is accomplished with the help of string nodes to build reserved area's for graphs in advance.

After a graph has been copied locally, its size is known. Before the graph is sent, a RESERVE message is transmitted to the destination to allocate this amount of memory in the remote heap. This is done by allocating a string of the requested size (a reference to the string is stored in the indirection table). The contents of this string are unimportant, so this is left uninitialised. If the reservation fails an error message is returned and the reducer tries again, otherwise an acknowledgement is returned and the reducer starts sending the graph.

Having reserved a piece of memory it becomes easy to store packets correctly. As packets arrive they are stored in the reservation string according to their position within the message. As soon as all packets of a single graph are available, the graph is unpacked: all node-pointers and code addresses are adjusted and the string-package is removed. If the root node of the unpacked graph is not in RNF the indirection index that is used to refer to the graph is stored in the root node itself. The waiting list field is used for this purpose. See the next section (requests) for more information.

The amount of messages needed to send a graph can be decreased. Instead of creating reserved areas before a graph is sent, this is done at the moment the first packet arrives (first in time, not necessarily the first inside the message). This requires each packet to contain the total size of the graph as any packet may arrive first. An acknowledgement/error message is still needed: when packets can't be stored (garbage collection) they are thrown away at the destination and they must be retransmitted. Therefore the sending process must keep a reference to its local copy of the graph as long as it hasn't received an acknowledgement. The receiver only sends an acknowledgement if memory could be reserved at (or before) arrival of the first packet: when it is certain all packets will be stored properly. Normally spoken this protocol means two messages are needed instead of three, although many packets might get retransmitted when an error occurs.

Note that in the optimised scheme some packets might arrive twice if error messages are returned early. This is quite problematic. A packet may arrive for a certain indirection table entry when it is used for other purposes, to receive another graph for example. This can be solved in different ways, but easiest is probably to return error messages only after all packets have been received (and thrown away).

The optimised scheme has not yet been implemented for the transputer: space is reserved in advance.

Communication on PABC level

As has been written earlier, there are not many PABC instructions that involve communication with other processors. We have seen how messages can be sent, in what way channels are represented and how to copy graphs to other processors. In this section we'll see how this is used to implement the PABC instructions create_channel, send_graph and send_request. We'll first show a simple solution, followed by a discussion of some improvements.

Simple version

In a first attempt to implement PABC instructions involving communication using the tools presented earlier one can proceed in a rather straightforward way. Every instruction can be expanded in a sequence of simple messages. No attempt will be made to find the best solution possible, but one that works. (The transputer implementation uses this 'simple' version at the moment).

channel creation

To create a channel to a certain processor, a new entry should be obtained from this processor and a channel node must be created and filled with the index of the remote entry. This is done by sending a REQUEST_ENTRY message to the processor and suspending yourself. The remote processor will answer by returning a message containing a new index (or an error value). The index is pushed on the b-stack of the waiting reducer, which will be woken up directly hereafter. Finally the reducer creates the channel node containing the new index.

send_graph

This routine is essentially implemented as indicated above (global copying). In addition to sending a graph a reducer should be started. The following steps are involved: first a graph is copied locally, next a RESERVE message is sent to allocate space for the graph and the reducer (stacks etc.) and finally the graph is transmitted, together with the start address of the reducer. As soon as the whole graph has been received it is unpacked and the reducer is started on the root node of the graph.

requests

send_request is the most complicated instruction of the ones described. Sending a request requires some negotiation and administration.

First of all, on execution of a send_request instruction, a new channel node is created that points to the (reserved) channel node on which the send_request is executed. It gets the weight of the channel node it points to. This new channel node will be used as a global reference to the node that has to be updated and as a request message: it is sent to the remote processor. In this way a new 'request channel' is created in the opposite direction automatically. It prevents the original channel node from being deleted by the garbage collector before it gets overwritten by the result.

When a request (channel node) arrives there are two distinct possibilities. Either the requested graph is in RNF, in which case it should be returned immediately, or it is not, which means the request should be stored and handled as soon as the graph becomes updated. We'll take a look at the latter one.

Suppose the graph that is requested is not in RNF. It is possible no reducer is reducing it at the moment the request arrives, even worse, it may not even have arrived yet (due to packets arriving out of order). This means the request should be stored. As no graph may be available this isn't done in the graph itself, but in the table entry that references the graph: a special field in this entry is used to refer to all request channels that arrive. The link field within each channel node is used to link the channels to each other. Thus a list is built of all requests that still have to be answered.

As soon as a node becomes updated all requests for it must be answered. This is done by executing a release instruction on the waiting list of the node: the indirection table index that is stored in the waiting list of every globally known closure is used to find all request channels. The result graph is copied over each of these channels (note the possibility to share local copies). Hereafter the reference count of the requested graph is decreased by the weight stored in each request/channel node.

After a graph has been returned successfully the channel node that issued the request is overwritten by the root node of the returned graph and all reducers suspended on it are released.

Some Implementation details

There are some implementation issues that require attention.

In order to copy a result back one has to take into account all issues related to copying a graph to another processor, including reservation of memory. Situations might occur where it's not possible to reserve memory for some result (for a while). At the moment the runtime system keeps trying. This introduces endless loops when memory stays low.

The system presented so far uses return messages to accomplish certain tasks. This introduces buffering problems (and thus a possibility to deadlock). We are using a separate message buffer to 'deal' with this problem. For most programs this works fine, but in certain situations it causes programs to be aborted. This problem will be discussed below.

Due to possible speculative parallelism it is vital that processes are scheduled in a fair way. If this isn't be guaranteed, requests will have to be handled in a different way.

Improvements

Combination of create_channel and send_graph

Normally spoken create_channel and send_graph are used in pairs. This seems to offer possibilities to combine both instructions. One could delay creation of a channel and do this while reserving memory for a graph. This only works if memory is reserved in advance. If it's reserved at delivery of the first packet, the other packets (being transmitted at that very moment) have no way to tell what table entry to use. We think it's better to stay with the optimised reservation scheme and combine this with improved creation of channel nodes as shown below.

Faster Creation of Channel nodes.

Creation of channel nodes is quite bothersome: two messages are needed at the moment. There are several ways to deal with this.

In first place the source processor could pick a locally unique id for its channel nodes each time a channel is created. Send_graph includes this id when it sends a graph. The destination processor should take care the id is 'glued' to the graph when it arrives. To avoid confusion about duplicate id's from different processors the indirection table could be implemented by a hashing table. Disadvantage of this method is the fact that a slightly more complex structure has to be used to implement the indirection table. Big advantage is an extremely fast creation of channel nodes.

A second solution involves buffering of new channel id's. When a channel node is created, a REQUEST_ENTRY message is sent to the destination. Instead of returning only one id, the several of these should be sent back. They will be stored in a special buffer at the source. Subsequent calls to create_channel will take id's out of this buffer. This can be improved still further by returning new id's each time the arrival of a graph is acknowledged. In this way the buffer is maintained without any overhead as long as it doesn't become depleted.

Safe return messages

Sometimes a problem arises during the routines send_request and send_graph. This is due to (temporary) shortage of memory at some destination. Situations occur in which a message that arrives cannot be stored at a particular moment. This introduces the need to return error messages to some sender. Unfortunately this requires memory to be allocated as well. At the moment this problem is alleviated by using a separate buffer space to return messages. In some exceptional situations buffer overflows do occur however.

Before discussing a possible solution, I'd like to make clear that the router as implemented in the transputer version cannot be used to return messages directly after delivery of some packet: it should free each buffer as soon as a packet arrives at its destination. This buffer must then be allowed to be used by some link to accept another packet (for through-routing), otherwise a deadlock situation might arise. In other words: another solution has to be found.

At the moment it seems best to limit the amount of sending/requesting a processor may do to a remote processor. This allows the remote processor to keep a small set of small 'message return buffers' for each possible source. As all 'dangerous' messages require some message to be returned it is easy to keep the 'allowance' of each processor to send messages up to date. At times the allowance is zero, the source processor just has to postpone sending. Although safe, it's not yet clear whether this would introduce to much synchronisation between processors. Clearly synchronisation can be dealt with by using a bigger allowance, but this would require precious memory space.

Current Status of the Transputer Implementation

At the moment a parallel code generator has been implemented that generates code that is comparable to its sequential counterparts. It is based on the (sequential) code generator that has been implemented for the M680x0 and the Sparc (van Groningen (1990,1991)). The code that is generated has been improved compared to earlier versions. This can be seen in the table below that lists some results for sequential programs. A slight speedup has been achieved by including guards in Concurrent Clean, but most is due to optimisations within the code generator. As can be seen the biggest differences have been measures for programs that use a lot of heap.

	old version	new version
nfib	12.7	12.2
tak	11.3	11.3
reverse	128	78.4
queens	63.4	47.9
fastfourier	26.8	13.8

In addition to the code generator, a runtime system has been devised that allows parallel graph rewriting. It implements the 'simple' version of the communication mechanism as presented in the previous sections (i.e. deadlock free routing, channels etc.). A global garbage collector has not been implemented yet, although all structures to accomplish weighted reference counting are available. Some preliminary results are given below. They suffer from various problems that still have to be solved and the absence of certain mechanisms (global garbage collection, a more tightly packed graph representation, improved creation of channels etc.). For some programs the algorithm or problem size implies too much overhead. For all programs the processes have been allocated in a random way. Note that the ZAPP results (Goldsmith et al (1991)) are better for nfib. This is mainly due to the mechanisms used in ZAPP to control load distribution and scheduling of processes.

	1 processor	16 processors	speedup
nfib	12.2	1.4	8.7
float_nfib	23.7	2.2	10.8
mandelbrot	147	18.2	8.1
quicksort	9.7	10.1	0.96
fastfourier	13.8	5.6	2.5
queens	47.9	6.2	7.7
sieve + output	20.6	17.5	1.2
warshall	12.2	11.2	1.1

All programs have been run with a heap of 2 Mbyte, except fastfourier (3 Mbyte). Thresholds have been used to limit the total amount of processes. Both nfib and float_nfib compute nfib 30, the latter using floating point arithmetic. Mandelbrot computes a well known fractal in a resolution of 530 by 320 and depth 128 (without plotting). quicksort sorts 10,000 elements. fastfourier transforms 8K complex numbers. queens uses a board of 10 by 10. The sieve is a straightforward 'sieve of Erathostenes' program on 10,000 numbers (The parallel version starts up new processes for every 20 numbers). The warshall algorithm has been executed on an adjacency matrix of 64 by 64.

Conclusions and Further Work

In this paper an efficient and reliable communication mechanism has been presented that supports parallel functional graph rewriting as implemented in Concurrent Clean. Some design decisions have been addressed in conjunction with implementation problems, memory management and deadlock avoidance in particular. First tests have indicated that absolute speedups due to parallelism can be achieved with this system, at least with a subset of possible algorithms. These speedups have been accomplished using an optimising code generator on a distributed memory system. The actual implementation of the communication primitives presented here can still be improved. This indicates that communication doesn't (necessarily) constitute a bottleneck if it is used in combination with state-of-the-art compiled code.

A lot of questions are still left unanswered. In the near future we'll focus on some of them. First of all the suitability of Concurrent Clean to express parallel computations has to be investigated and (probably) enhanced. This will indicate the requirements that are imposed on global garbage collection, load balancing and controlling task size. Some information can already be extracted from the benchmark results presented earlier. When more complete, this information will be used in a next step to investigate possible solutions to the indicated problems.

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.

Bevan D.I. (1987). 'Distributed garbage collection using reference counting', Proceedings of the conference on Parallel Architectures and Languages Europe (PARLE'87). Springer Lecture Notes in Computer Science 259, Vol II, page 176-187.

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.

Goldsmith R., McBurney D.L. and Sleep M.R. (1991). 'Concurrent Clean on ZAPP'. Proceedings of the Semagraph '91 Symposium on the Semantics and Pragmatics of Generalized Graph Rewriting., Nijmegen, the Netherlands, (10-12 December), to appear 1992.

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.

Groningen J. van (1992). 'Some Implementation Aspects of Concurrent Clean on Distributed Memory Architectures', Proceedings of the Fourth International Workshop on the Parallel Implementation of Functional Languages, Aachen, September 1992.

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

Kesseler M.H.G. (1991). 'Implementing the PABC machine on transputers', Proceedings of the Third International Workshop on the Parallel Implementation of Functional Languages', Southampton, June 1991.

Kingdon H, Lester D.R., Burn G.L. (1991). ‘The HDG-machine: a highly distributed graph-reducer for a transputer network’, The Computer Journal, Vol. 34, No. 4, 1991.

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.

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.

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

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.

Watson P., Watson I., (1989). ‘An efficient garbage collection scheme for parallel computer architectures’, Proceedings of the conference on Parallel Architectures and Languages Europe (PARLE’87). Springer Lecture Notes in Computer Science 259, Vol II, page 432-443.