Efficient Routing using Class Climbing

Marco Kesseler
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract. This paper concentrates on a routing algorithm that is adaptive and avoids deadlocks and starvation. It is a general store-and-forward packet router that has been developed to be used as part of a distributed runtime system for Concurrent Clean, an experimental, lazy, higher-order parallel functional programming language based on term graph rewriting. The routing algorithm is based on the one used by the communication processor of the Decentralised Object Oriented Machine (DOOM) architecture, which uses a class climbing algorithm to avoid deadlocks. It has been altered in such a way that it can be used for the Inmos T800 transputer. Efficiency issues will be focused on. Performance figures clearly indicate that algorithm is significantly faster - and more reliable - than the communication primitives of the Helios operating system.

1. Introduction

The T800 transputer hardware allows each transputer to be connected to no more than four others, which means that networks of more than five transputers are not fully connected. This implies that messages between two transputers that are not neighbours have to be routed via some intermediate transputers. Various ways exists to implement such a router. This paper presents one based on buffer classes.

The router that will be described has been devised for a transputer implementation of Concurrent Clean, an experimental, lazy, higher-order parallel functional programming language based on graph rewriting [1,2]. Efficient single-processor implementations of this language have been realised at the University of Nijmegen for the Motorola 680x0 processor family and the Sparc [3,4]. A prototype implementation for the T800 transputer is being developed, which consists of a transputer code generator and a runtime system [5,6,7,8,9]. The latter needs asynchronous point-to-point communications of packets (of limited size) between arbitrary nodes in the network (resembling a global block move). The following requirements must be met:

- absence of deadlock
- absence of starvation
- independence of network topology and size
- efficient usage of buffer space
- high data throughput

Additionally the delay of messages must be kept as small as possible, but this is not a very important issue in the Concurrent Clean system. The effects of relatively large delays (i.e. processes waiting for some result) can be kept to a minimum by allowing multiple processes to run on the same processor, so that useful work can be done while some processes are waiting.

We have chosen to implement an adaptive packet router that meets these requirements. We will see below that adaptive routing is better than routing over fixed paths when the network load is high. The router is based on the routing algorithm that was used for the special purpose router processor of the Decentralised Object Oriented Machine (DOOM) architecture [10]. This algorithm had to be altered in order to be usable for a network of transputers. As the algorithm uses a class climbing algorithm to avoid deadlocks we have called our router the Class Transputer Router (CTR for short). In this article we will refer to it as the Clean Router as well, to avoid confusion between the original algorithm and the new one. Note that packets have some maximum size. This means that large messages have to be broken up in smaller pieces that must be sent over the network separately. This reduces delay of large messages because packets of a single message are pipelined this way. Also, adaptive routing may increase throughput for broken-up messages by using multiple paths for a single message, whereas fixed routing cannot.

There are two reasons to develop a custom transputer router. On the one hand a lot of research has been devoted on a theoretical level to avoidance of deadlock and starvation in packet switched networks, without designing and testing concrete algorithms on concrete machines. On the other hand a reasonable number of routers have been developed for the transputer, but many of them do not consider problems of deadlock and starvation thoroughly. Those that do sometimes use a lot of memory, limit network topology - by disallowing cycles in routing paths for instance -, or are inefficient at higher network loads. Apart from these drawbacks not all routers (in particular their sources) are (or were) easily obtainable.

First we will give a description of the DOOM algorithm. Next, it will be shown that this algorithm cannot be used for a network of T800 transputers unmodified and we will proceed with the presentation of a modified algorithm. We will mainly focus on efficiency issues. Finally some performance figures will be presented.

2. The DOOM Algorithm

The algorithm that has been used to implement the Clean router is based on the one presented in [10]. It has been developed as part of the DOOM project and we will refer to it as the DOOM router. It has been proved to prevent deadlocks, provided that each packet that reaches its destination is consumed within a finite amount of time. To accomplish this, a fixed number of buffers and a class-climbing algorithm are used. Starvation is avoided by a mechanism that passes around privileges for buffers. It is adaptive, i.e. it does not use a fixed path to route packets, so hot spots in the network are avoided to some extent. The transputer version conforms to this algorithm as much as possible.

2.1. Structure

In the original algorithm each communication processor has a number of connections to other communication processors. Each connection consists of two physical links (wires) and each processor has two communication 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 physical link.

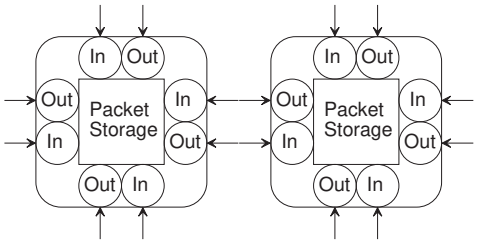

Figure 1 DOOM communication processor overview

Every physical link is able to carry messages in either direction but not in both directions at the same time. Part of these messages contain control information to regulate communication; The rest consists of data messages that contain the actual information that has to be transported. Data messages are not allowed to exceed a certain maximum size and will be referred to as data packets, or packets in short. Packets always travel from an output process to an input process.

In addition to normal data, each packet contains its class and its destination. These are the only two values within a packet that are used by the router. The class is used to avoid deadlocks; packets with higher class have more privileges. This will be explained later. The destination is used to determine over which links a packet should be transmitted. The router uses a routing table for this: it indicates for each destination which links can be used for transmission.

2.2. Process Interaction

The communication processes repeat a small number of actions infinitely, as depicted below. They operate quite independently. On a single processor they interact by means of a shared packet store to which they all have mutually exclusive access.

The packet store consists of a fixed number of fixed size buffers. The buffers are structured as a number of output queues. For each output process such a queue exists, but buffers are *not* dedicated to a certain queue. The same packet may in fact reside in distinct queues simultaneously, meaning the packet is shared among several queues (so it is not copied physically). In this way a single packet can be sent over multiple links. Each output process is allowed to pick packets out of its own queue to transmit over its link.

Input Process:	Output Process:
reserve buffer	
send request	receive request
	retrieve matching packet
receive packet or cancel	send packet or cancel
store packet or free buffer	

Figure 2 Actions of communication processes.

The communication processes continuously perform the following actions. Whenever an input process is able to reserve a buffer of a certain class c, it sends a request over its link. The corresponding output process then starts searching in its own output queue for all packets that match this request (= all packets of class ≥ c). From these packets it takes the one that has arrived first. In other words: the criteria to arbitrate between packets are time of arrival and class (where the class of the chosen packet is not necessarily the highest one). After a packet has been chosen the output process removes it from all output queues it was placed in and sends it over the link to the requesting input process. If no packet could be found it sends a cancel message. On reception of a cancel message the input

process frees the buffer. Conversely, when a packet arrives it is stored in the reserved buffer and put in all output queues that are indicated by the routing table.

It can be seen that the initiative for the transport of a packet is taken by the receiving process, not by the sending one. As a consequence of this one can postpone the decision to which neighbour each packet should be forwarded until it is certain that at least one neighbour is willing to accept the packet.

2.3. Class Climbing

Class climbing is a well-known method to avoid deadlocks [10,11,12]. It can be used for routing algorithms that are able to force a packet to travel over a path without any cycles (i.e. a packet may travel in cycles, but in critical situations it must be possible to reach the destination via a non-cyclic path). Like some other deadlock-avoiding algorithms class climbing is based on the consideration that deadlocks are the result of cyclic (buffering) dependencies between different messages. Unlike others that limit the number of possible physical paths a message may travel it avoids these dependencies by means of sophisticated buffer management. The DOOM algorithm uses this method and its correctness has been proved [10].

As the name suggests a class climbing algorithm works with classes: at any moment in time each packet has a certain class. During transportation this class may be increased, but never decreased. Buffers have a class as well and for every possible class at least one buffer exists. Packets of class n can pass through buffers of class n or lower only. This means that packets of a higher class have more buffers available to travel through than packets of lower class.

In some class climbing algorithms a packet gets a higher class each time it hops over a link. The DOOM router uses a different approach. The ‘increase every hop’ approach implies that the minimum number of buffers that must be allocated is equal to the distance a packet can travel, which is related to the diameter of the network. In contrast, the DOOM router superimposes an acyclic graph on the physical communication network. This means each connection gets a direction. As long as a packet travels along (respectively against) this direction its class remains the same, but when it changes direction its class is increased. This means that the number of buffers can be limited considerably for networks that have few cycles but a relatively large diameter (e.g. an array, a tree, or a ring).

Note that the number of buffers needed is linear with the network diameter at worst. In [13] it is argued this is problematic. One should keep in mind however that one can easily build a large network with a diameter of O(log n). In our view this increase in buffer space for large networks does not constitute any problem.

2.4. Buffer Management

In order for the algorithm to work efficiently the request messages, which have been mentioned above, should always request packets for as low a class as possible. As stated in [10] a rather intricate administration is used to accomplish this efficiently. One of the things that is carefully avoided is copying the contents of buffers to others, by connecting buffers to a certain class only logically, never physically. The following example shows the disadvantages of buffers that are physically connected to a certain class. Suppose a processor has 2 buffers, one of class 0 and one of class 1. Both are filled with a class 1 packet. When the class 1 buffer is freed the router should be able to send a request for a class 0 packet, but there is no buffer available to store it in. The router does not solve this by moving the class 1 packet from the class 0 buffer to the class 1 buffer, but by adjusting some counts and flags that change the logical association of the buffers with a certain class.

In the rest of this paper we will talk about a buffer of class x, instead of a buffer that is logically associated with class x.

It is important that buffers are dedicated to certain classes or links as little as possible. This allows for the most efficient utilisation of resources under all circumstances. Class 0 buffers are most general as they can be used for packets of any class. For this reason every class that is bigger than 0 has only a single buffer associated with it, so that the rest of the buffers is associated with class 0. This is reflected in the buffer administration, which is split in two parts: a zero class part and a non-zero one, enabling the use of a more efficient administration for zero class buffers. In the next section we will see the same holds for the fairness administration that prevents starvation.

2.5. Starvation

In addition to deadlock avoidance the DOOM algorithm provides a method to prevent starvation. This is done in a reasonably straightforward way. As noted above the buffer administration is split in two parts: for zero class buffers and for non-zero ones. For both kinds a different method of fairness administration has been devised.

Zero class buffers can be allocated by any input process as long as there are more zero class buffers available than the number of input processes. When less buffers are available they are distributed according to an array of Booleans. When the n^{th} Boolean is true input process n may allocate a zero class buffer. If it does so, it should clear the corresponding Boolean, thereby throwing away its privilege. When a zero class buffer is freed, a round-robin method (by means of a global variable) is used to choose some Boolean that is false and make it true again.

Each non-zero class buffer is dedicated to exactly one input process at any moment in time, which means that only this input process may allocate a buffer of that particular class. After it has used its privilege the input process passes it on to the next input process. It should be noted that this relies on the fact that each input process never suspends itself, so that privileges are passed around quickly and never get ‘stuck’ at a sleeping process.

To be clear, these methods to avoid starvation are not needed to ensure absence of deadlock. The class climbing algorithm alone suffices. This is important, as we will see later we had to adopt a different fairness administration for the CTR.

3. The Clean Router

The DOOM algorithm has been designed for a special router processor. It is not possible to implement it directly on transputer hardware (T800), but we have conformed the CTR to the original algorithm as much as possible. For example: the CTR acts as if it is running on a separate processor with 5 links. Four of these links are transputer links and one is a connection with ‘the transputer itself’. This section will focus on the problems that are related to a transputer version of the algorithm and the solutions we have adopted.

3.1. Incompatibilities

Two aspects of the DOOM algorithm need careful consideration. First of all it relies on inter-processor connections that consist of two independent links each. Secondly the input process is not blocked ever: when it receives a cancel message it goes on requesting and when it cannot reserve a buffer it starts polling for one. The output process is blocked only at moments in which it waits for a request to arrive. In practice this means it waits only during the time the corresponding input process is polling for buffers. This polling is rather exceptional: normally both processes are continuously running. These aspects introduce two important problems when implementing this algorithm for a network of transputers:

- Although each T800 transputer connection consists of two separate links (wires), these links cannot operate independently and a single link cannot send messages in both directions: each transputer connection consists of two unidirectional links. When data are sent over a link, the opposite link is used for acknowledgements. This imposes a problem as a single link cannot be used by multiple processes simultaneously. 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.

- On the transputer the router processes should run at high priority. Not only does this facilitate the use of shared data structures - high priority processes are not time-sliced - but much more important: the use of high priority processes is vital when good response times have to be achieved (as noted in the ZAPP experiments [15]). In the original DOOM algorithm the router processes run without any interruption. Running these processes at high priority on the transputer would not allow any low priority process to run.

3.2. Changes

To overcome the problems mentioned above the algorithm has been changed in two ways. In the first place the messages from the input processes (requests) and the messages from the output processes (packets and cancel messages) that have to be sent over the same transputer link are multiplexed over that link. Secondly the router processes will suspend themselves in certain situations. The latter solution introduces an additional problem that is related to fairness administration: processes might fall asleep while holding some privilege and this would prevent others to access certain buffers. We will take a closer look at these problems below.

3.2.1. Multiplexing

The multiplexing of messages is done as follows. To avoid inappropriate use of transputer links output processes are not allowed to read from a link and input processes are not allowed to write to a link. Instead each input process delegates write actions to the corresponding output process and when it receives a message for an output process it passes it on (see picture below).

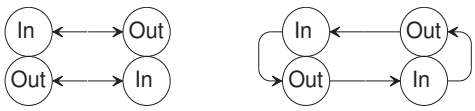

Figure 3 The DOOM links (left) versus the CTR links (right).

As can be seen in the picture above, request messages now have to travel through two intermediate communication processes to reach their destination. This does not introduce a performance penalty as messages from an input process to a local output process are not copied in reality. They are globally known to both processes.

It will be clear that this change of inter-process connectivity may not influence the original behaviour of the algorithm. As explained in [14] the transputer is able to provide exactly the same communication primitives with this interconnection scheme as the DOOM hardware, which means the original algorithm is not violated in any way. Below we will see how we have solved the other problems.

3.2.2. Sleeping Processes

In the DOOM algorithm the router processes are running even when there are no packets to be transported: the input processes are continuously issuing requests and the output processes cancel these. As explained above this is a serious problem for a transputer implementation. One would like to suspend the routing processes when there is no message to be sent. However, suspended processes do not pass on privileges, so we will abandon the original fairness administration and introduce a different one later. Now we can suspend processes at some moments. We have implemented the following solution.

When an input process receives a cancel message for a certain class this means that the other side of the connection has no packet available that matches the previously issued request. Because of this it is no use to go on requesting packets of this class or higher. If no request of lower class can be sent the input process falls asleep instead of issuing useless requests. It will be woken up in two cases: (a) when the input process is able to send a request of lower class than the one cancelled and (b) when the other side tries to send a packet that matches the previously cancelled request.

In case (a) occurs it is easy to wake up an input process. Case (b) involves a special 'wake-up' message, which we will call a signal message. When the output process on the 'other side' gets a packet that matches the class of a previously cancelled request, it sends a signal message to the sleeping input process to make it send a new request. Each signal contains the highest class of all messages that match the previous cancel. This is used to avoid sending superfluous requests: when a signal arrives and no suitable buffer is available, the input process does not wake up. In effect each class n signal simulates a request of class n+1 followed by a cancel: it just increases the class that has been cancelled and only if this means a 'matching' request can be sent, the input process is woken up.

If we abandon the original method to avoid starvation it is easy to show that suspending the input process in this way does not violate the mechanism of the original algorithm to avoid deadlock (see [14]). We will take a closer look at issues related to starvation below.

3.2.3. Starvation

Although it is possible to build a deadlock free router with the techniques presented above, they invalidate the DOOM method to avoid starvation. To be more specific: if a privilege is granted to a sleeping process it might not get passed on to others quickly. Even worse: it may not get passed on at all. This has led to a different fairness administration for the CTR that does not allow sleeping processes to have any privileges [14]. This avoids the problem of the original fairness administration.

For reasons of efficiency this fairness algorithm requires the number of zero class buffers to be equal to or greater than the number of connections per processor. For the transputer this means at least 5 zero class buffers are needed: 1 for each transputer link and 1 for the connection between the 'router processor' and the transputer itself. For each non-zero class buffer only a single buffer is needed.

3.3. Efficient Class Climbing

The routing algorithm that has been presented so far prevents deadlocks and starvation, but it contains some inefficiencies. In this section a few optimisations will be presented that improve the execution speed of the router. In general these optimisations work best at higher loads of the network.

3.3.1. Variable packet size

To avoid having to send a lot of bytes for small packets, packets have variable size up to some maximum. It is not entirely trivial to implement this on a transputer as the communication hardware requires both sender and receiver to agree on the number of bytes that have to be transferred. For this reason the router splits packets in two parts. The first part has a fixed length and contains four fields: the kind of the message (to distinguish packets from requests, cancels and signals), the class of the packet, the length of the packet, and finally a request field, which will be explained below. This header layout is not only used for packet headers, but also for signals, requests and cancels. In the following we will use the word 'header' to indicate both packet headers and control messages. In contrast to control messages, packet headers are followed by a body of the exact length that is mentioned in the header.

At the moment each header field consists of 32 (!) bits. This can be decreased considerably but in our prototype this allows for some additional tags in each header. Also, for small messages, size has proved not to be very important. Even so, headers will probably be smaller in future versions of the CTR.

3.3.2. Overlapping data and control messages

Having received a packet header, one does not have to wait for the packet body to issue a new request or a signal. The CTR overlaps the sending of a new request with the transmission of a packet body. In this way the next request will have arrived by the time the current packet has been transmitted entirely (if it is not very small). This minimises the delay between subsequent packets. Signal messages are not (yet) overlapped with packet bodies. It would impose some problems for large packets bodies. A request for a packet may arrive when its body has not yet been received entirely and this would result in additional cancels and signals unless the router becomes more complex.

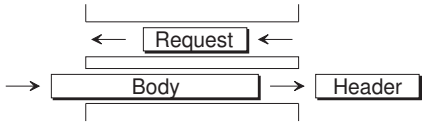

Figure 4 Overlapped transmission of request and packet body

3.3.3. Combination of messages

On the transputer it is more efficient to send a small number of large messages instead of a large number of small ones. On each connection two opposite data streams are implemented by the router. Packets, cancels and signals are transported along the direction of data and request messages go against it. The router uses this to piggy back requests of one data stream on the non-request messages of the other stream. For this the request field of each header is used: it contains the class that is requested, in contrast to some special value if no request is piggy-backed. Due to this optimisation no control messages are needed when a connection is used continuously in a bi-directional way: only packets containing piggy backed requests are needed (in contrast to the worst case overhead when traffic is low: signal - request - packet - request - cancel, four additional - but small - messages)

3.3.4. Traffic control

When a lot of buffers are in use the router is slowed down due to a decrease in the number of links that can be used simultaneously (and bidirectionally), resulting in sub-optimal

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 measures 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 fairness administration in combination with a heavy load it did not resolve at all. A heuristic has been used to suppress this problem: fairness measurements are related to the number of free buffers. When a lot of buffers are in use at some processor, new packets are accepted less frequently at this processor, favouring through-routing and delivery of packets. This increases the overall number of routing-holes in the network.

3.3.5. Code optimisations

The router has been written in Par.C. The Par.C compiler does not generate very efficient code, though. A few routines have been optimised by hand and this slightly improved performance. It remains to be seen what speed can be obtained with an optimising C compiler. We have ported the router to PACT parallel C, but for this version no extensive measurements have been taken yet.

The current implementation wakes up routing processes eagerly. Sometimes processes are woken up that fall asleep immediately thereafter. By inserting additional tests when waking up some process this may be prevented in future versions (at the cost of an increase in code size).

3.3.6. Lazy requests and lazy signals

Some additional changes to the algorithm might decrease control overhead in case the load is not extremely high. First of all it may be better to wait a little before returning a cancel message. A suitable packet might arrive shortly. This avoids an additional cancel - signal - request sequence. One should not wait too long, though, as the requesting process may have some lower class buffer available meanwhile: a cancel message would trigger a request for this lower class.

Secondly, when a packet arrives that can be sent over multiple links, all these links are woken up if they are asleep. This means several signal messages are generated, which is rather superfluous when traffic is low. A single one is probably sufficient. If this does not result in a suitable request within some time other links could be signalled after all. For high loads this is somewhat inappropriate as one would like to activate as many links as possible in this case. It is not clear whether these changes are very useful (they have not yet been implemented).

3.4. Routing Strategy

The CTR investigates the network topology during start-up and builds a table of all shortest paths. An algorithm of $O(n^2)$ is used for this, where n is the number of processors in the network. Packets are routed over one of the shortest paths found in this way. For point-to-point communications one can use either an adaptive or a fixed strategy. The former routes packets over an arbitrary shortest path and does not guarantee packets to be delivered in the same order they were sent. The fixed strategy uses a fixed shortest path to route packets: this is somewhat less efficient with high packet loads in general, but it keeps packets in the right order. The ability to keep packets ordered can be very useful for some kinds of communication (file IO for instance). One-to-all broadcasting has been implemented as well, but for this no performance figures are available yet.

4. Performance

In this section we will present the results of some tests that were carried out with the CTR. These will be compared with tests of the Helios communication mechanism [16] and the Parix communication mechanism. It will be shown that Helios is not only less reliable - it is not deadlock free -, but that it is slower as well, in particular for larger networks and higher traffic loads. The tests of the Parix communication primitives had to be carried out on a different architecture and must be treated with some care. Parix is very fast for small messages, but relatively slow for large messages, especially for larger networks and more complicated communication patterns. At the moment of writing, we did not have enough information about the performance of other routers under the same circumstances on a comparable machine to relate these to the CTR properly.

For the Virtual Channel Router (VCR [17]) some figures about message latency over 1, 2 and 3 hops in a quiet network have been published. Looking at the latency for messages of different size one can deduct the raw link speed is approximately 1500 Kbyte/sec. For small messages in a quiet network the VCR outperforms the CTR. Apart from differences in link speed this is mainly due to the complexity of the CTR. Small messages do not benefit from overlapped requests for instance, which results in relatively large delays. Apart from this the CTR performs sub-optimal in quiet networks. It remains to be seen how well the VCR performs with high loads on larger networks. Apart from this the VCR needs large amounts of memory: using default settings (160 bytes per buffer) the VCR kernel needs about 100 Kbyte plus 6400d bytes of buffer space per node, where d is the network diameter.

Tiny is able to avoid deadlocks by eliminating cycles from routing tables. It allows routing without deadlock-avoidance as well. In [18] one can find some timing information about Tiny. It presents the travel times over 1 to 12 hops in a quiet network for messages of 4, 16, 64, and 256 bytes. It does not state whether these figures apply to deadlock-free routing or not, but this is probably not important. The raw link speed of the architecture used is about 770 Kbyte per second. For Tiny the same holds as for the VCR: it is fast for small messages in a quiet network, but it is not clear how efficient it is for high loads in a large network. It would be interesting to known if the elimination of cycles from routing tables affects the efficiency of Tiny in certain situations.

The router presented by Son and Paker [13] has been tested on various network topologies, but the account of its performance is too short to draw any conclusions. The maximal average throughput appears to be about 260 Kbyte/sec. The router cannot force packets to be delivered in the correct order, which can be a problem for some kinds of communication. It is interesting to see that this router performs best on a cross mesh, because this topology has many (but long) paths between any two processors. The cross mesh is not exceptionally well-suited to the CTR, as the number of different shortest paths is small.

Except for the tests of the Parix system, all tests presented below were run on a Parsytec supercluster that contains 64 T800 transputers running at 25 MHz with 4 Mbyte of 5-cycle memory each (with error detection and correction) [9,19]. In this system all transputers are connected via a network of crossbar switches (Inmos C004 chips) that enable different network topologies to be used easily. The raw unidirectional link speed is about 1330 Kbyte/sec. For links that have to be set up via multiple cross-bars this number is even smaller. The tests do not make use of the on-chip memory of the transputer: one usually needs this for other purposes (the Concurrent Clean system uses it as a set of ‘registers’). The initialisation code of the CTR takes about 4 Kbyte of code and the router itself about 8 Kbyte (excluding buffers and routing table). All tests of the CTR use adaptive routing with 10 zero class buffers per processor, except when stated otherwise. Additionally, a single

buffer is allocated for each non-zero class. The number of classes equals the network diameter, even for topologies where less would be sufficient. The buffer size equals the maximum packet size. Using buffers of 1 Kbyte and a network with diameter d the amount of buffer space per node equals (10 + d) Kbyte.

4.1. Helios versus CTR

In this section we will compare the performance of the CTR with that of the Helios (version 1.2) communication primitives. We will see that the CTR is significantly faster - especially for large networks - and that Helios is not entirely reliable.

The figure below depicts performance figures on an array of processors. Both CTR and Helios were tested with 2, 8 and 16 transputers. It can be seen that Helios slows down when larger arrays are used. The CTR is not influenced by the array size in any way.

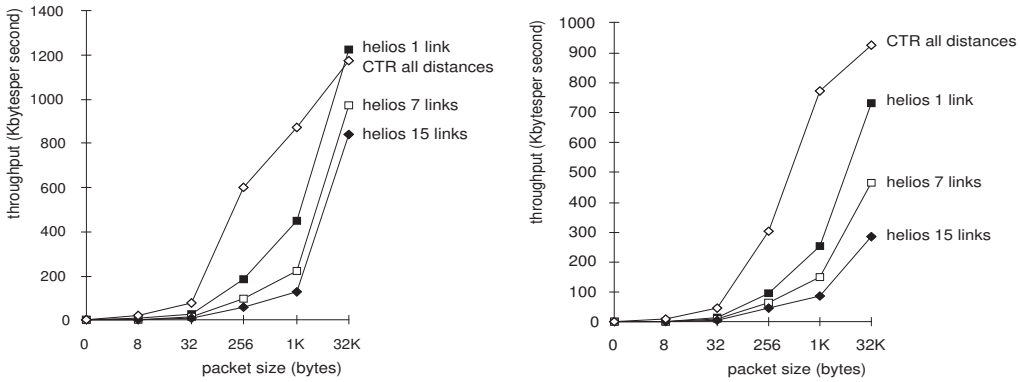

Figure 7 Communication over an array of processors. In the figure on the left all packets travel in the same direction from one end of the array to the other. In the figure on the right packets travel from one end of the array to the other end in both directions simultaneously. The picture shows the throughput in a single direction. The total throughput is twice this value.

The following pictures show information about some tests on a grid and a ring. Helios does not take advantage of the network topology here. In contrast, the CTR is able to increase bandwidth by using multiple paths simultaneously.

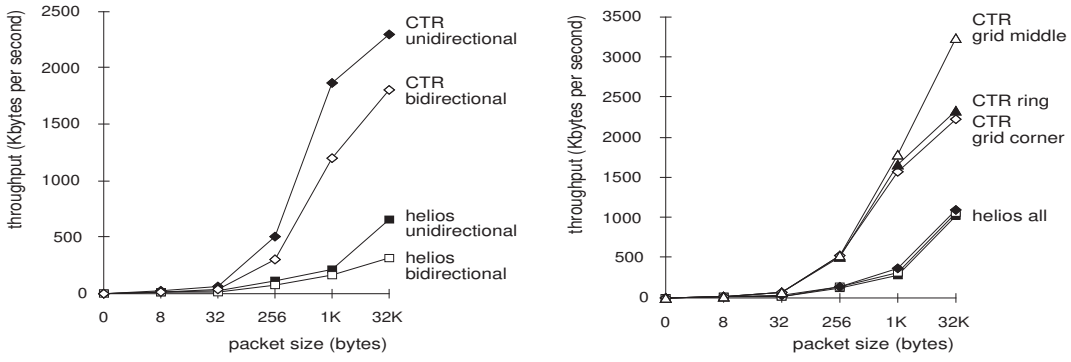

Figure 8 Communication over a 2-dimensional grid of 16 processors and over a ring of 16 processors. The figure on the left shows the throughput from one corner of the grid to the opposite one for both unidirectional and bi-directional communications between the two corners. The value for bi-directional communication indicates the throughput for one direction only: the total throughput is twice this value. The figure on the right shows the throughput from some source processor to random destinations (the number of bytes the source can send away randomly per second). This has been measured for a 2-dimensional grid of 16 processors and for a ring of 16. In case of the grid the source was either a corner processor or one in the centre.

The picture below shows the performance of the CTR and Helios when all processors are sending messages to random destinations, which means communications are frequent and chaotic in this case.

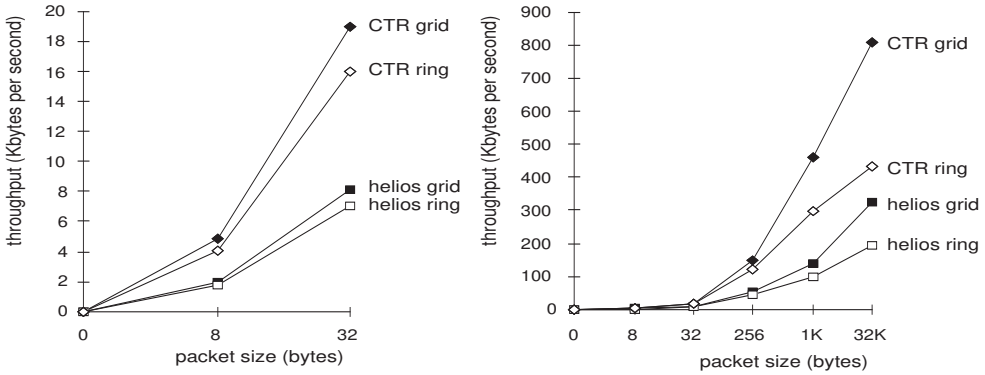

Figure 9 Throughput per processor when all processors are sending messages to random destinations (i.e. the number of bytes each processor can send away randomly per second). These results were obtained on a ring of 16 processors and on a 2-dimensional grid of 16 processors. The picture on the left is an enlarged part of the one on the right. To obtain the total network throughput these values should be multiplied by 16: for the CTR using 1 Kbyte packets this is about 7 Mbyte per second on a grid, and about 5 Mbyte per second on a ring. Using a grid of 16 the average distance a packet travels is about 2.7, which means that about 19 Mbyte of data makes a hop per second, indicating every link transports approximately 790 Kbyte per second bidirectionally on average. For the ring the average distance is about 4.27 so that approximately 21 Mbyte traverses a link per second, resulting in an average bi-directional throughput of 1300 Kbyte per second per link.

In addition to the differences in speed between Helios and the CTR, Helios is not deadlock-free. Helios allocates routing buffers at runtime. This means that communications might fail when memory is low on some intermediate processor. The Helios manual [16] states that, when a message gets lost due to congestion, an exception message is generated which is sent back to the 'reply port'. Unfortunately congestion may cause exception messages to get lost as well and there is no guarantee that subsequent tries will succeed. Tests have shown that programs that allocate a lot of memory at start-up may deadlock at the moment (heavy) communication takes place.

Looking at the results presented above it is clear that the CTR outperforms Helios. This may not be very problematic for 'normal' Helios applications, as the Helios programming philosophy is quite different from that of Concurrent Clean, but for our purposes Helios is not very well suited. In the next section we will take a closer look the communication mechanism of Parix.

4.2. Parix versus CTR

We have been able to run some tests with the Parix communication mechanism, which is based on virtual links. Unfortunately this had to be done on a different architecture as Parix has not been installed on our Supercluster. The CTR on the other hand has not (yet) been ported to this other architecture, so we had to relate communication figures of two different hardware platforms. Nonetheless, they give some indication of the relative merits of both communication systems.

The Parix tests have been run on a network of 512 T805 transputers running at 30 MHz each having 4 Mbyte of memory. Only grid networks are possible in this system. Using Parix one is able to achieve a maximum link bandwidth of about 1100 Kbyte/sec, but it is not clear whether this is the true hardware limit.

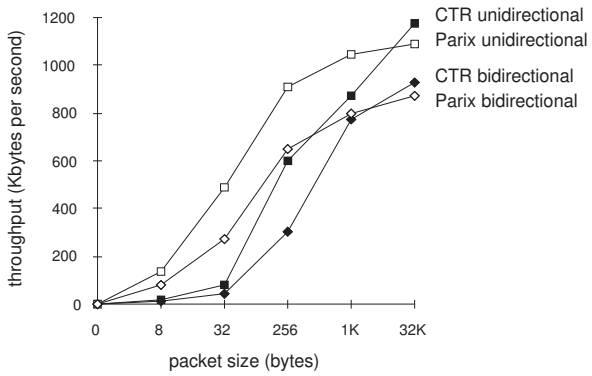

Figure 5 Throughput over a single link in a quiet network. Parix is clearly faster than the CTR. Both unidirectional and bi-directional communication over a link has been measured.

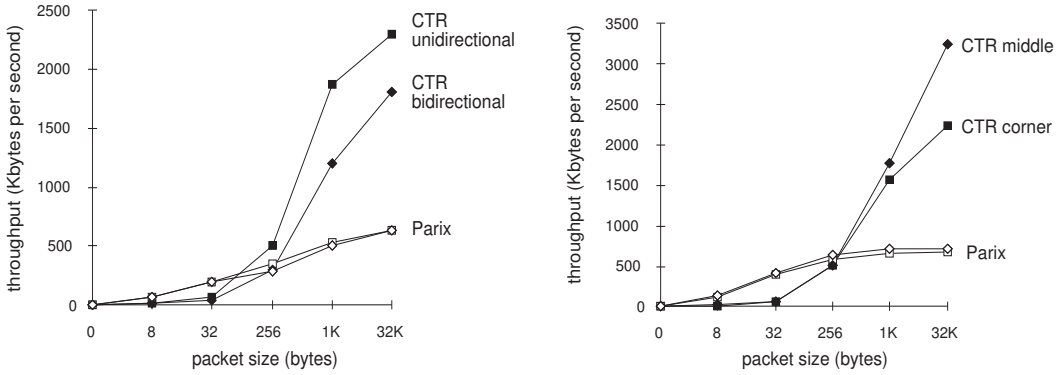

Figure 6 Communication over a 2-dimensional grid of 16 processors. The figure on the left shows the throughput from one corner of the grid to the opposite one for both unidirectional and bi-directional communications between the two corners. The value for bi-directional communication indicates the throughput for one direction only: the total throughput is twice this value. The figure on the right shows the throughput from some source processor to random destinations (the number of bytes the source can send away randomly per second). The source was either a corner processor or one in the centre.

As can be seen in the pictures above, in general, Parix is very fast for small messages, about as fast as the CTR for messages around 256 bytes, and rather slow for messages exceeding 256 bytes. It is not clear how much memory each virtual link takes. On the one hand Parix is faster than the CTR for communication over a single link in a quiet network, but on the other hand performance of Parix decreases with increasing distance. We were planning to test how well Parix performs when all processors send randomly destined messages, but unfortunately we have not yet been able to set up a fully connected network of virtual links over a grid of 16 transputers.

Concluding, we can say that it depends on the communication pattern needed which kind of routing mechanism should be used. For the tests presented above Parix is clearly best when small messages have to be routed. The CTR may be better for large messages. As we have not been able to test Parix when all processors send messages to random destinations it is not yet clear how Parix compares with the CTR in this respect. It remains to be seen how the CTR relates to other routers, such as the VCR, when these are submitted to the same tests.

4.3. Optimised Code

The router has been written in C. To obtain better performance we have optimised some routines by hand. The effect of this can be seen in the picture below. The speedup for unidirectional communication over a single link has been measured for different packet

sizes. For large packets the effect of these optimisations is small - in this case the link speed is more important than the speed of the routing routines.

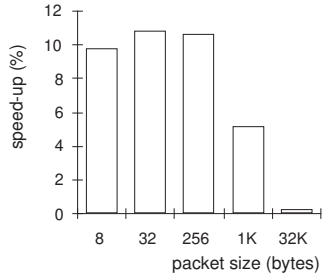

Figure 10 Speedup of router due to slightly optimised code

4.4. Number of Buffers

The number of zero class buffers per processor influences the performance of the router. The more buffers are available the smaller the chance that requests need to be cancelled. When very few buffers are available it is difficult to use multiple links simultaneously. The figures below present some tests with packets of 256 bytes on a few simple topologies (arrays).

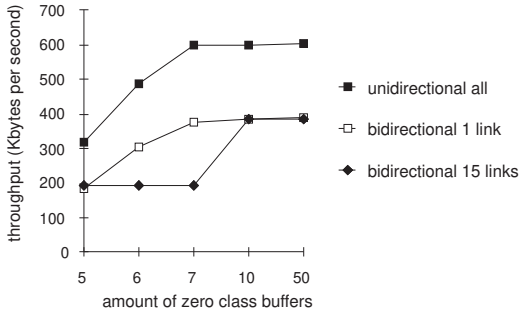

Figure 11 Performance on arrays for different numbers of zero class buffers (with packets of 256 bytes). For reasons of efficiency the current implementation requires the number of zero class buffers to exceed the number of transputer links (4).

These tests - and others on different topologies - have indicated that 10 zero-class buffers are sufficient in general. Adding more buffers will sometimes increase performance slightly for larger topologies, but we have not yet encountered an example where this resulted in a significant speed-up.

4.5. Adaptive Routing versus Fixed Routing

Depending on topology and network load, adaptive routing is able to improve performance. It does introduce some additional overhead though, so that it will not always be better to use adaptive routing. Also, adaptive routing does not guarantee that packets arrive in the correct order. For some kinds of communication this is rather troublesome. The CTR implements both: for each packet one can specify whether it should be routed over a fixed shortest path, or over any shortest path.

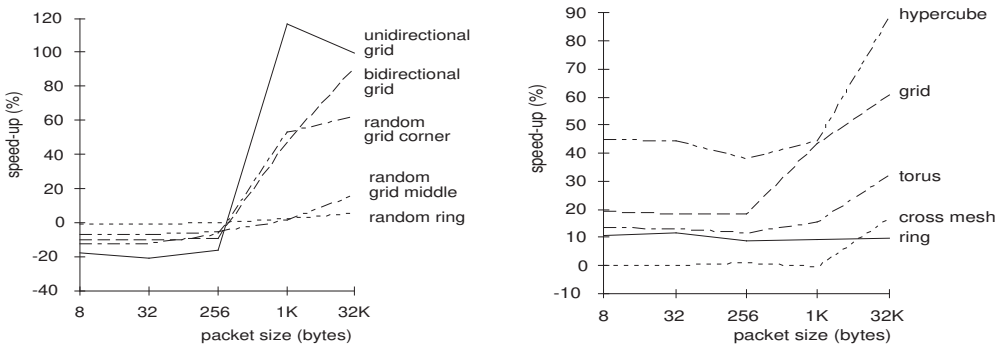

Figure 12 Speedups that have been obtained by using adaptive routing instead of fixed routing. The following tests are shown in the figure on the left: unidirectional communication between opposite corners on a grid of 16, bi-directional communication between opposite corners on a grid of 16, randomly destined packets from a single processor on a ring of 16, randomly destined packets from a corner processor on a grid of 16, and randomly destined packets from a 'middle' processor on a grid of 16. The figure on the right pictures tests in which all processors simultaneously send randomly destined messages. This has been done for the following topologies: ring, cross mesh, torus, 2D grid, and hypercube.

The figures above show that adaptive routing is best for large packets and high loads. Additional tests have shown that the effect of adaptive routing for large packets diminishes when the input load on every processor is decreased. When packets are small or the network is not very busy - which means relatively much control information is needed - the link speed is not the limiting factor and a single path is able to do the job just as well as multiple paths, sometimes even better as a single reasonably well-used path introduces less useless control messages (signals followed by cancels) than several barely used paths.

4.6. Different Topologies

The next pictures show the throughput for different topologies when all processors send messages to random destinations. All measurements have been performed on networks of 16 transputers. They indicate the number of bytes a single processor was able to send away per second. For the total network throughput this figure should be multiplied with 16.

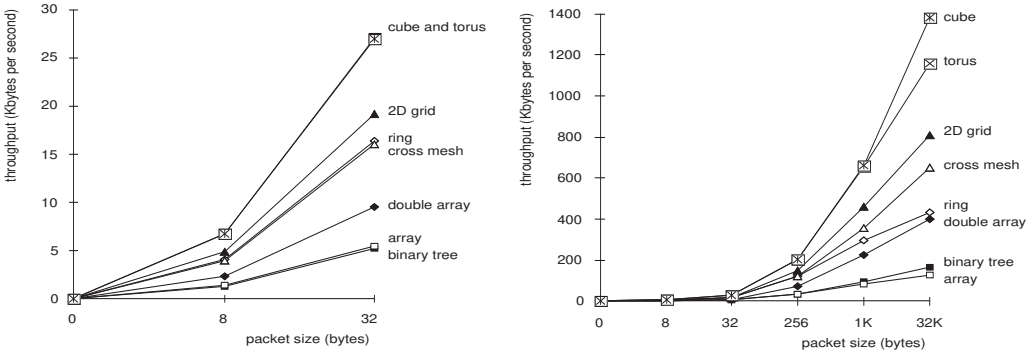

Figure 13 Throughput for various topologies using adaptive routing. The picture on the left is an enlargement of the one on the right. All networks consist of 16 transputers. Most topologies are self-explanatory. The double array is an array of which all connections consist of 2 transputer links instead of 1. The cross mesh is depicted below.

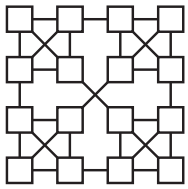

Figure 14 The cross mesh

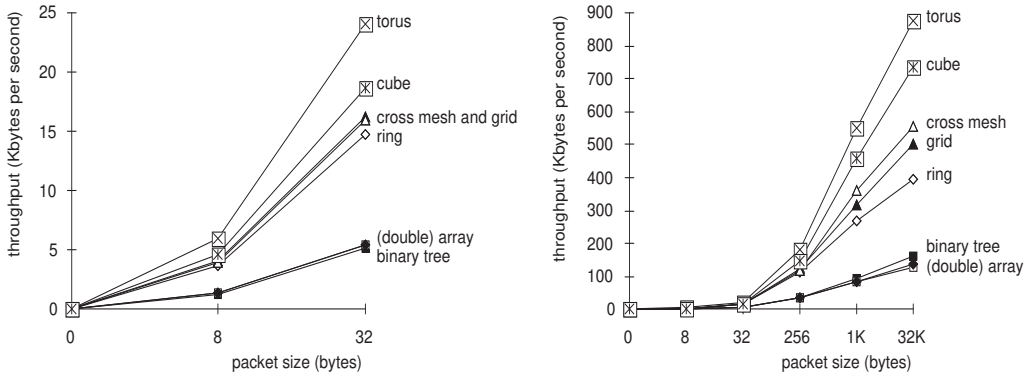

Figure 15 Throughput for various topologies using fixed routing. The picture on the left is an enlargement of the one on the right. All networks consist of 16 transputers. See figure 13 and 14 for an explanation of the topologies used.

5. Conclusions

Issues of the Class Transputer Router have been discussed, an adaptive router that uses class climbing to avoid deadlock. This router is based on the algorithm that was used for the routing chip of the DOOM architecture. We have given a brief description of the original algorithm and the changes that were needed to adapt it to the transputer hardware. Essential efficiency issues have been focused on. Performance figures indicate the router is able to compete with the Parix communication mechanism and is faster than the Helios communication primitives. The communication pattern needed determines largely which kind of routing mechanism is best. In general the Class Transputer Router performs very well for high network loads, taking advantage of network topology.

References

[1] Plasmeijer M.J., Eekelen M.C.J.D. van (1993). Functional Programming and Parallel Graph Rewriting. Addison Wesley, 1993.

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

[3] 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.

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

[5] Kesseler M.H.G., (1990), 'Concurrent Clean on Transputers', Master Thesis, University of Nijmegen, November 1990.

[6] Kesseler M.H.G., (1992), 'Communication Issues regarding Parallel Functional Graph Rewriting', Proceedings of the Fourth International Workshop on the Parallel Implementation of Functional Languages, Aachen, September 1992.

[7] 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.

[8] 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.

[9] INMOS Limited, Transputer Reference Manual, Prentice Hall International, 1988.

[10] 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.

[11] Bertsekas D., Gallager R., (1987). 'Data Networks'. Prentice-Hall International 1987.

[12] Stallings W., (1988). 'Data and Computer Communications'. Macmillan Publishing Company.

[13] Son N. T., Paker Y., (1991). 'Adaptive Deadlock-free Packet Routeing in Transputer-based Multiprocessor Interconnection Networks'. The Computer Journal, Vol. 34, No. 6, 1991, Page 493-502.

[14] Kesseler M., (1993). 'The Class Transputer Router'. Poceedings of the PaCT-93 Conference on Parallel Computing Technologies., Obninsk, Russia, to appear.

[15] 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 Generalised Graph Rewriting., Nijmegen, the Netherlands, (10-12 December).

[16] Perihelion Software LTD (1989). 'The Helios Operating System'. Prentice Hall International, 1989.

[17] Debbage M., Hill M., Nicole D., (1991). 'Virtual Channel Router Version 2.0 User Guide'. Technical Report, University of Southampton, October 1991.

[18] Clarke L., Wilson G., (1991). 'Tiny: an efficient routing harness for the Inmos transputer'. Concurrency: Practice and Experience, Vol. 3(3), Page 221-245.

[19] Parsytec GmbH (1989). 'Supercluster Technical Documentation'. Revision 1.2, April 1989.