THE CLASS TRANSPUTER ROUTER

MARCO KESSELER
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

ABSTRACT

This paper presents an adaptive routeing algorithm that 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 routeing 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 machines that do not have dedicated routeing processors. We focus on issues related to an efficient implementation of this router for the Inmos T800 transputer called the Class Transputer Router (CTR). The changes made to the original algorithm will be presented and their correctness with respect to avoidance of deadlock and starvation will be shown. Performance figures clearly indicate that the CTR is significantly faster - and more reliable - than the communication primitives of the Helios operating system.

Keywords: routeing, class climbing, deadlock, starvation, performance measurement.

1. Introduction

This paper presents a router based on buffer classes that 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 that meet the following requirements:

- 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. It is based on the routeing 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.

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 or are inefficient at higher network loads.

We will start with a short description of the DOOM router below. 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. 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.

2.1. Processor 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.

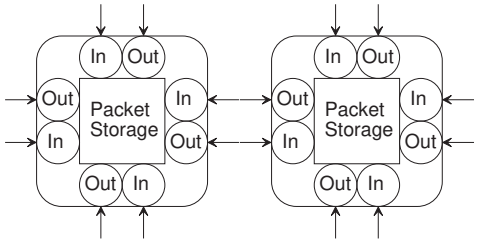

Figure 1 DOOM communication processor overview

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 destination and a class. The class is used to avoid deadlocks.

2.2. *Interaction of Processes*

The communication processes repeat a small number of actions infinitely, as depicted below. On a single processor they interact by means of a shared packet store to which they all have mutually exclusive access. They operate quite independently and employ an intricate administration to efficiently use the available buffers.

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 the packet store for all packets that match this request (= all packets of class ≥ c that may be sent over this link). 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 the packet store 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.

The initiative for the transport of a packet is taken by the receiving process. In this way 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 routeing 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). The DOOM algorithm uses this method and its correctness has been proved [10]. We will not concentrate on this proof, but limit ourselves to general remarks about the algorithm itself.

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. This implies that the minimum amount of buffers that must be allocated is equal to the distance a packet can travel, which is related to the diameter of the network. This does not have to be problematic (contrasting the arguments of Son and Paker [13]) as it is possible to build large networks with a diameter of O(log n). Nonetheless the DOOM router uses a different way to increase classes. It 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 allows to limit the number of buffers for networks that have few cycles but a relatively large diameter (e.g. an array, a tree, or a ring).

2.4. Starvation

In addition to deadlock avoidance the DOOM algorithm provides the following method to prevent starvation. Buffers of a certain class can be allocated by any input process as long as there are more buffers available than the number of input processes. When less buffers are available each buffer is dedicated to exactly one input process at any moment in time, which means that only this input process may allocate this buffer. After it has used its privilege the input process passes it on to the next input process that does not have this privilege. This algorithm relies on the fact that no communication process ever 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. 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 introduce problems when implementing it on a network of transputers. First of all it relies on inter-processor connections that consist of two independent links each. Secondly the communication processes are not blocked ever. We will take at closer look at these problems below.

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. This imposes a problem as a single link cannot be used by multiple processes simultaneously. If two processes at the same transputer start transmitting 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 [14]). 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 (user) process to run.

3.2. Multiplexing

To overcome the link problem mentioned above messages from different processes that have to be sent over the same transputer link are multiplexed over that link. 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).

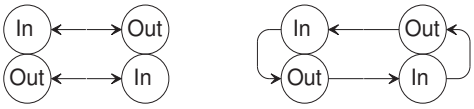

Figure 3 The DOOM links (left) versus the CTR links (right).

It will be clear that this change of inter-process connectivity may not influence the original behaviour of the algorithm. Two issues need to be considered. Firstly the ring of unidirectional links in the picture above should support two entirely separate data streams. Furthermore, the use of a ring introduces the possibility of a classical store-and-forward deadlock. Such a deadlock arises when all processes in the ring start to send a message while none of these processes is able to accept a message. If one looks at the DOOM algorithm, though, one can see that there are never more than two messages travelling simultaneously between two given processors, so this seems no problem. We will see later that additional changes in the algorithm introduce some messages and consequently this deadlock problem. Both problems can be solved in a rather simple way.

- To support two separate data streams both input and output processes must be able to accept any mixture of messages from the two streams: no communication process may at any time commit itself to accept only one kind of message.

- To avoid the store-and-forward deadlock problem processes should not block when sending a message. This is problematic for the messages travelling from an output process to an input process as the transputer hardware implements synchronous communication. It is sufficient however to ensure that the input process does not block on sending a message. Note that the routeing algorithm is such that this does not introduce buffering problems.

When link communications are implemented this way the transputer is able to provide exactly the same communication primitives 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 problem.

3.3. Suspend and Signal

In the DOOM algorithm the router processes are running even when there are no packets to be transported. This is a serious problem for a transputer implementation. One would like to suspend the routeing 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 safe moments, i.e. when they expect input from another process. We have adopted a solution that is based on the following considerations.

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 may just as well do nothing instead of issuing useless requests. It needs to 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.

The Clean router implements this as follows. When a cancel message is returned the input process 'falls asleep'. It does not really stop itself though, as it should continue reading messages from the link. In this case 'falling asleep' means that it frees the buffer it has reserved and withholds a new request message, preventing a subsequent cascade of cancels and requests. There are two ways to reintroduce a request in the network. First of all, on the processor that contains the 'sleeping' input process, each time a buffer becomes available of lower class than the one that has been cancelled, the input process is notified to issue a new request. Conversely, when the output process on the 'other side' gets a packet that matches the class of the 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.

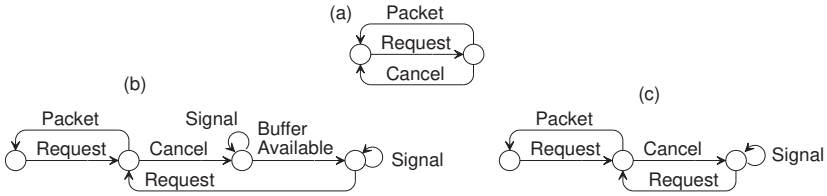

Figure 4 State transitions of the router processes. 'Buffer available' means a buffer can be allocated of lower class than the one cancelled. (a) represents both the input process and the output process in the DOOM algorithm. (b) and (c) depict processes of the CTR. (b) stands for the input process and (c) models the output process. (c) is an abstraction of (b): $signal^* \cdot buffer\ available \cdot signal^*$ is represented by $signal^*$ as freeing a buffer is invisible for the output process. (b) is in fact a process graph of input process ‖ output process. (a) is an abstraction of (c) - and therefore of (b): signals are non-existent in the DOOM algorithm.

Note that a signal may be sent to an input process that has already been woken up. This can be seen in graph b above: at some point a signal is possible while the input process is trying to send a request. This requires the store-and-forward deadlock mentioned above to be considered.

If we abandon the original method to avoid starvation it is easy to show that suspending the input process does not violate the mechanism of the original algorithm to avoid deadlock. Two scenarios are conceivable after an input process has fallen asleep: (1) it will be woken up in the future and (2) it will not. We will take a look at both below.

It is trivial to show that the first scenario is safe: when the input process wakes up it proceeds at the exact point it had fallen asleep. In the original algorithm this can simply be seen as a case where the input process runs extremely slow for a while. As

process speed plays no role in the proof of the original algorithm, the first scenario is perfectly safe.

Suppose the second scenario occurs, i.e. an input process falls asleep and is never woken up again. Consider that a suspended process does not hold any shared resources: it has previously freed the buffer it had reserved, and it does not hold any privileges as no fairness administration is used. Furthermore, it continues passing messages from other processes. This means that it does not influence any other process except the output process on the neighbouring processor, which will not get new requests.

Let us assume that at some point in time the input process stopped due to cancellation of a request of class n and no signal of a class ≥ n will be received in the future (note such a class exists). This implies (a) that the processor that contains the input process will never free a buffer of class < n *and* (b) the output process at the other side never has a packet available of class ≥ n. In the original algorithm (a) implies that all future requests will be of class ≥ n, whereas (b) means these can only be answered by a cancel as there will never be matching packets at the other side. In other words, the DOOM algorithm would poll for packets for an infinitely long time. This is equivalent to never waking up the input process.

Having seen that the changes presented above keep the routeing algorithm deadlock-free, we will take a closer look at issues related to starvation below.

3.4. Starvation

Although it is possible to build a deadlock free router with the techniques presented above, they invalidate the DOOM method to avoid starvation. This has led to a different fairness administration for the CTR.

To avoid starvation it suffices to make sure that each process eventually gets buffers of arbitrary low class within certain needs: sleeping processes need a buffer of lower class than the one that has been cancelled, whereas processes that are awake are in need of a buffer of any class. The original algorithm passes around privileges for all buffers without taking into account these needs. Using induction over the classes it is easy to show the original algorithm avoids starvation. To prove the new algorithm, one needs to add that while an output process tries to forward a packet of class n, the input process on other side will never get a cancel of a class ≤ n (respectively n+1 if the class should be increased), and thus will stay eligible for a privilege of a class n buffer (respectively a class n+1 buffer).

The CTR fairness administration maintains privileges for buffers of all classes. When a buffer of a certain class is freed the privilege for this class is passed to the next process that needs a buffer of this class. If a process is found that already has a privilege, the process keeps the lowest privilege and the highest privilege is passed on. If no process needs the buffer it becomes free to be allocated by any process. If an input process receives a cancel for class n while it has a privilege for a buffer of a class ≥ n, it passes on this privilege.

This means that a process never has more than one privilege: it keeps the best privilege it gets. This is sufficient to avoid starvation and it allows to efficiently pass on the privilege when a process falls asleep. Note that sleeping processes cannot get any privileges without being woken up as well, which avoids the problem of the original fairness administration.

3.5. Optimisations

The routeing 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.

3.5.1. Variable packet size

To avoid having to send a lot of bytes for small packets, packets have variable size up to some maximum. To accomplish this on a transputer the router splits packets in two parts: a header and a body. The header has a fixed length and contains - amongst others - the length of the body.

3.5.2. Overlapped requests

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. This minimises the delay between subsequent packets. Signal messages are not (yet) overlapped with packet bodies.

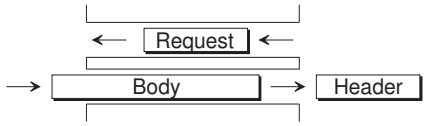

Figure 5 Overlapped transmission of request and packet body

3.5.3. 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 piggy backs requests on the non-request messages of the opposite stream.

3.5.4. 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 heuristic has been used to suppress this problem. When a lot of buffers are in use at some processor, new packets are accepted less frequently at this processor, favouring through-routeing and delivery of packets. This increases the overall number of routeing-holes in the network.

3.5.5. Future efficiency considerations

Some 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. Secondly, when a packet can be forwarded over multiple links that are asleep, several signal messages are

generated. A single one would probably be sufficient when traffic is low. It is not clear whether these changes are very useful.

3.6. Routeing Strategy

The CTR investigates the network topology during start-up and builds a table of all shortest paths. Packets are routed over one of these shortest paths. 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. 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 [15]. 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. 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.

For the Virtual Channel Router (VCR [16]) some figures about message latency over 1, 2 and 3 hops 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 and this number of hops the VCR outperforms the CTR. Apart from differences in link speed this is mainly due to the complexity of the CTR, from which a few small messages do not benefit. 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.

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.

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) [9,17]. All transputers are connected via a network of crossbar switches (Inmos C004 chips). 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 router takes about 4 Kbyte of code and the router itself about 8 Kbyte (excluding buffers and routeing table). All tests of the CTR use adaptive routeing 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 Clean

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.

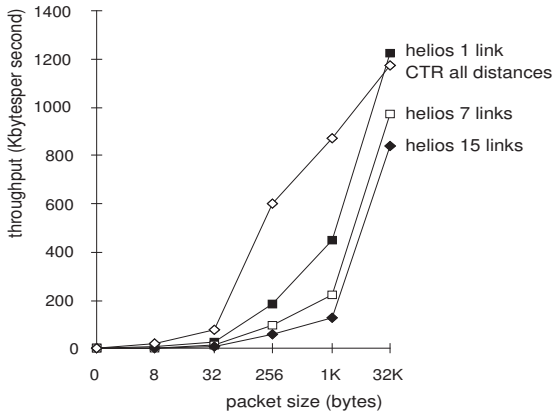

Figure 6 Communication over an array of processors. All packets travel in the same direction from one end of the array to the other.

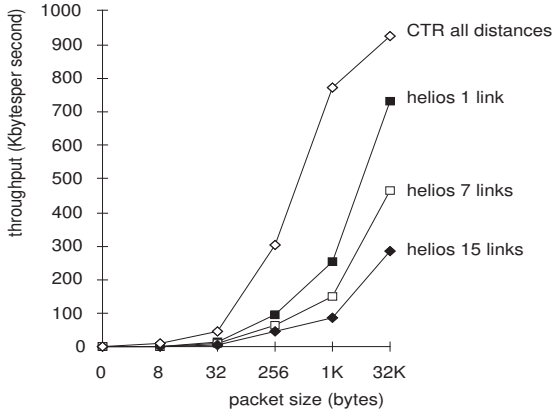

Figure 7 Communication over an array of processors. Packets travel from one end of the array to the other end in both directions simultaneously. The picture shows the throughput in a single direction. The total throughput is twice this value.

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.

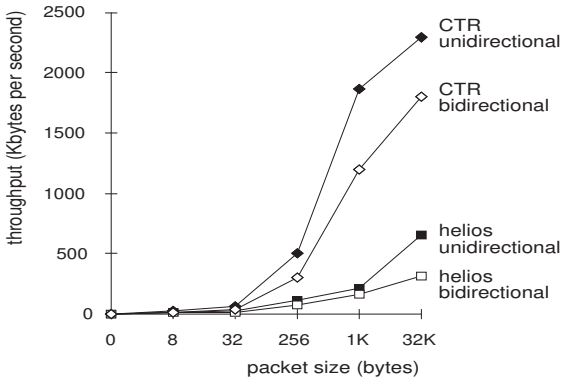

Figure 8 Communication over a 2-dimensional grid of 16 processors. The throughput from one corner of the grid to the opposite one has been measured for both unidirectional and bi-directional communications between the two corners. The value for bi-directional communication indicates the throughput for one direction only: the total throughput is twice this value.

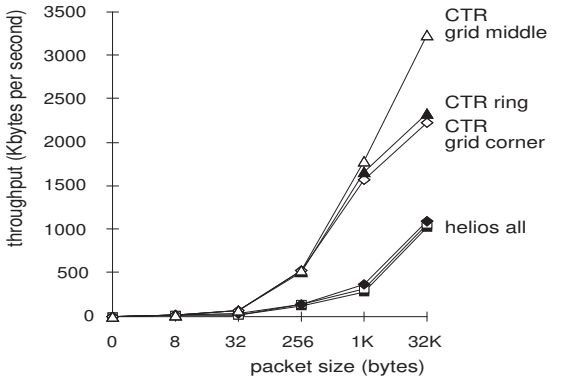

Figure 9 The throughput from some source processor to random destinations (the amount of bytes the source can send away randomly per second). This has been measured for a 2-dimensional grid of 16 processors and for a ring of 16. In case of the grid the source was either a corner processor or one in the 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.

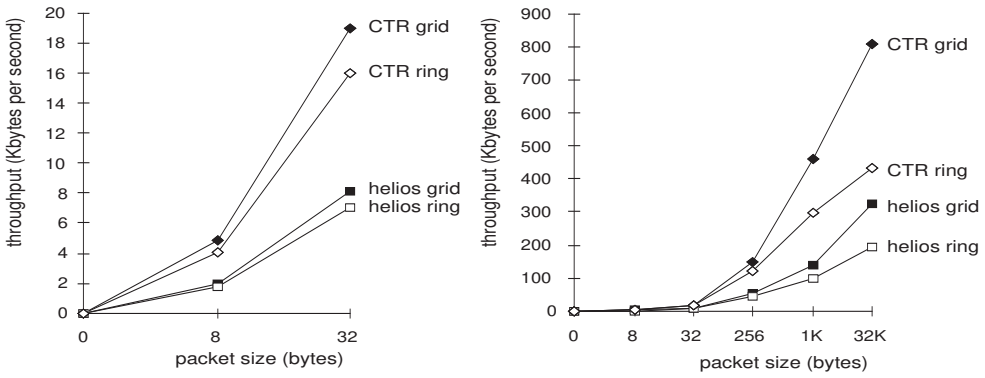

Figure 10 Throughput per processor when all processors are sending messages to random destinations (i.e. the amount of bytes each processor can send away randomly per second). These results were obtained on a ring of 16 processors and on a 2-dimensional grid of 16 processors. The picture on the left 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 routeing buffers at runtime. This means that communications might fail when memory is low on some intermediate processor. 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. It remains to be seen how the CTR relates to other routers, such as the VCR, when these are submitted to the same tests.

4.2. 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 routeing routines.

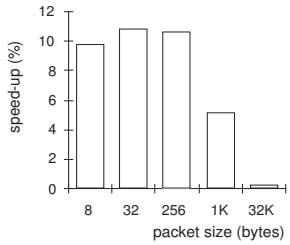

Figure 11 Speedup of router due to slightly optimised code

4.3. 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.

Figure 12 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.

4.4. Adaptive Routeing versus Fixed Routeing

Depending on topology and network load, adaptive routeing is able improve performance. On the other hand it introduces some additional routeing overhead and it does not guarantee that packets arrive in the correct order.

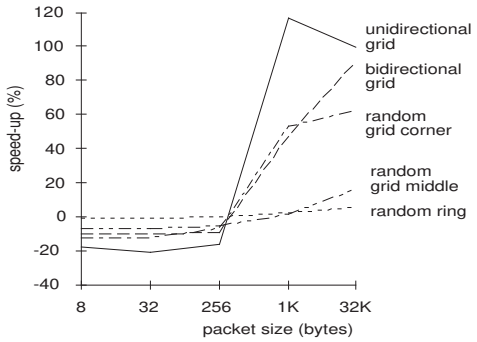

Figure 13 Speedups that have been obtained by using adaptive routeing instead of fixed routeing. The following tests are shown: unidirectional communication between opposite corners on a grid of 16, bi-directional communication between opposite corners on a grid of 16, randomly destined packets from 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.

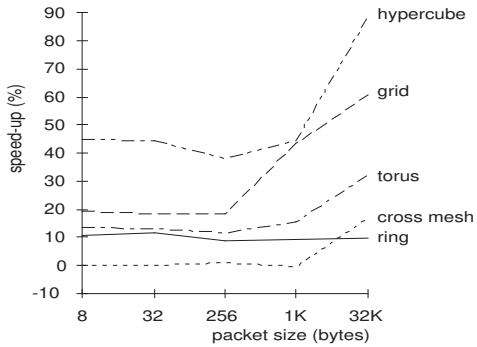

Figure 14 Speedups that have been obtained by using adaptive routeing instead of fixed routeing. The figure pictures tests in which all processors simultaneously send randomly destined messages. This has been done for the following topologies: ring, cross mesh, torus, 2D grid, and hypercube.

The figures above show that adaptive routeing is best for large packets and high loads. Additional tests showed that the effect of adaptive routeing 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.

4.5. *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.

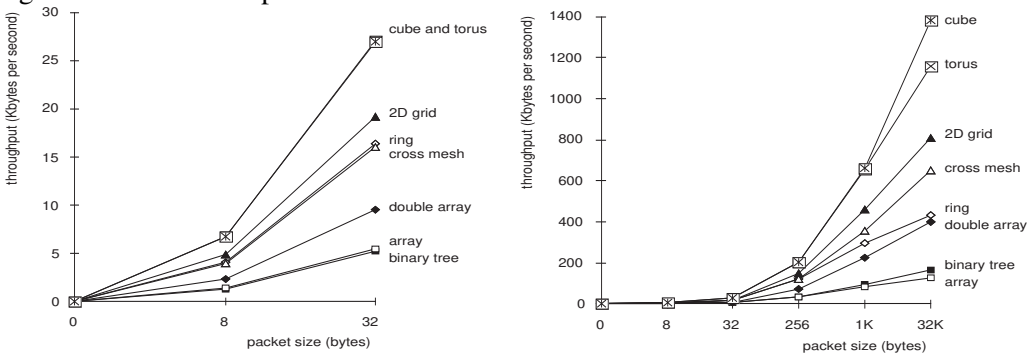

Figure 15 Throughput for various topologies using adaptive routeing. The picture on the left is an enlargement of the one on the right. All networks consist of 16 transputers. Most topologies are self-explanatory. The double array is an array of which all connections consist of 2 transputer links instead of 1. The cross mesh is depicted below.

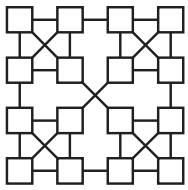

Figure 16 The cross mesh

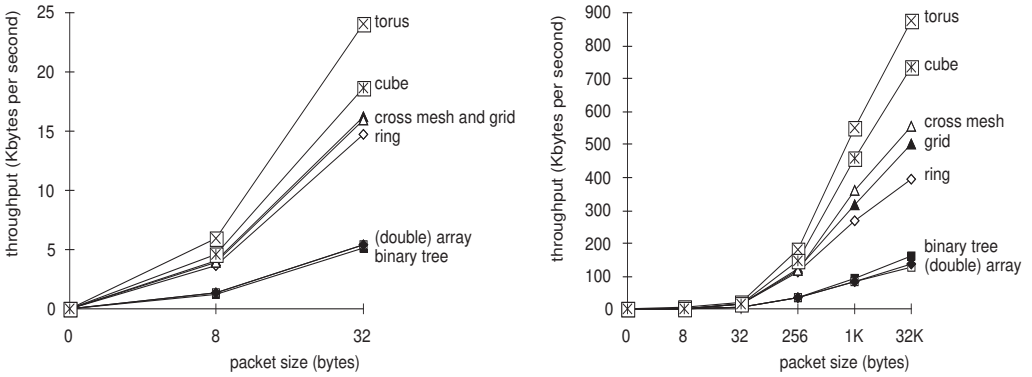

Figure 17 Throughput for various topologies using fixed routeing. The picture on the left is an enlargement of the one on the right. All networks consist of 16 transputers. See figure 15 and 16 for an explanation of the topologies used.

5. Conclusions

A an adaptive router was presented that is suited for a network of transputers and uses class climbing to avoid deadlock. This router is based on the algorithm that was used for the routeing chip of the DOOM architecture. We have given a brief description of the original algorithm, followed by a presentation of the changes that were needed to adapt the algorithm to the transputer hardware. It has been shown this modified algorithm is free of deadlock and starvation and some efficiency considerations have been discussed. Performance figures indicate the router is faster than the Helios communication primitives and 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. 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).

15. Perihelion Software LTD (1989). ‘The Helios Operating System’. Prentice Hall International, 1989.

16. Debbage M., Hill M., Nicole D., (1991). ‘Virtual Channel Router Version 2.0 User Guide’. Technical Report, University of Southampton, October 1991.

17. Parsytec GmbH (1989). ‘Supercluster Technical Documentation’. Revision 1.2, April 1989.

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