Yale University
Department of Computer Science

Explicit Routing in the Fluent Machine

Björn Lisper

YALEU/DCS/TR-645
August 1988

This work has been supported by the Office of Naval Research under contract N00014-86-K-0564.

Explicit Routing in the Fluent Machine

Björn Lisper
Computer Science Department
Yale University
New Haven, CT06520

August 28, 1988

Abstract

Fluent routing is an efficient way to emulate shared memory in a processor network with distributed memory. This allows for a conceptually simple parallel computation model but the simplicity is bought for a communication cost that is logarithmic in the number of processors. This is wasteful for computations that have such a structure that they can use *local communication*, with communication time independent of the size of the given network, efficiently. Thus, it is of interest to use local or near-local communication whenever it is beneficial in a fluent parallel computer. We define *explicit routing* where full control is given over the routing path. Explicit routing can be used to implement local and near-local communication, and we show how to implement it so that it interferes in a minimal way with the fluent routing scheme. Finally we isolate some problems that can occur when explicit routing is employed and we discuss ways of solving them.

1 Introduction

Programming parallel computers is a complex and error-prone task. Different approaches have been taken to ease the burden of the programmer and simplify the transfer from program specification to implementation. One way to go is to provide hardware support for a relatively high-level programming model. If this is done the "semantic gap" between the high-level program and the machine is diminished which makes the translation easier.

A particular, imperative programming model for parallel computing is the Concurrent-Read Concurrent-Write Parallel Random-Access Machine (CRCW PRAM) model. In this model an arbitrary number of synchronous processors share a common memory. Any processor can read or write to any memory location at any time, and these operations take one unit of time to complete. Memory access conflicts are resolved by some given rule. This model has become quite popular for expressing parallel algorithms since it hides details of communication and synchronization. Communication simply takes place through the common memory.

CRCW PRAM's cannot possibly be built directly in hardware, however, given the restrictions of available technology in the foreseeable future. Therefore, they have to be emulated on technically feasible parallel hardware if this programming model is to be of any practical use. A promising way to do this is Ranade's *fluent routing*, [15,16]. In this model a CRCW PRAM is emulated by a butterfly network of message-passing processors with local memory. The common memory of the PRAM model is physically distributed among the processors in the network. Memory accesses are implemented by sending messages in the network, from the requesting processor to the one holding the accessed memory cell and back. Fluent routing is an efficient scheme for handling the message passing. If the butterfly network has N processors the time for a fluent memory request to be completed will be $O(\log N)$ with very high probability [15].

Whenever high-level concepts are supported in hardware a penalty has to be paid. This penalty may be in the form of extra hardware, longer execution time or both. Therefore there is another trend in parallel architectures that emphasizes raw hardware speed at the cost of conceptual simplicity. As a matter of fact this way of building parallel machines has been predominant so far, due to the need for high performance and technological limitations. In order to bridge the semantic gap between hardware and the programming model, a higher demand is placed on either the programmer or compilers. In the case of compilers a quite interesting development has taken place during the last couple of years. Notable is *trace scheduling* for VLIW architectures [4] and vectorizing compilers for conventional supercomputers [12]. These techniques are common in that they isolate code segments that consist of straight-line code or that can be unraveled into such code, e.g. simple loops. They try to find a good *scheduling* of the events generated by the code segment such that the communication pattern fits well with existing communication facilities while processor utilization is kept high. On the theoretical side, the *space-time mapping* paradigm in its general form [9] offers a new unified conceptual framework for these methods as well as a theory in which the correctness of them can be ensured. This paradigm, while hitherto mostly applied to synthesis of fixed special-purpose parallel hardware [10,14], may also lead to new compiler techniques for parallel computers [2,7,8].

In the case of fluent routing the additional hardware requirements are quite minimal [16] but the $O(\log N)$ time memory accesses may be far from optimal for certain types of computations whose structures map well to the butterfly network used to implement the fluent PRAM. If, for instance, the computation can be scheduled so that communication only needs to take place between adjacent processors, then access time will be $O(1)$, i.e. independent of the size of the network. Theoretical results exist about what kind of computation structures that can be efficiently scheduled on butterfly networks [1]. Simulations indicate that the difference in time between fluent and local accesses will typically be around two magnitudes on a 13–dimensional butterfly network with 114688 processors [16]. Therefore, if the true peak performance of the hardware is to be achieved then local or near-local routing must be utilized when advantageous to do so. This is especially true for areas such as scientific computing where performance is a main issue and the algorithms often exhibit a large degree of structure that can be determined at compile-time and thus scheduled in advance to yield a minimum of control overhead.

Therefore it seems likely that we will want to combine explicit routing with fluent routing. A compiler, for instance, may isolate parts of a program that are suitable to

execute using fixed-connection, local or near-local communication in the network while leaving other parts to use the general routing scheme. In this paper, we define explicit routing and show how to implement it in the fluent network in a way that does not interfere more than necessary with the fluent routing scheme.

The rest of this paper is organized as follows: Section 2 contains a brief primer of the Fluent Abstract Machine and we show how it is supposed to be implemented. In section 3 we define explicit routing and present a way to implement it. Section 4 identifies some problems that may occur in a system where explicit and fluent routing is used at the same time. Here we also propose some ways to cope with these problems and we conclude with a few remarks.

2 The Fluent Machine

In this chapter we give a brief introduction to the Fluent Machine as described elsewhere [16] in order to enable the reader to understand the following discussion. We first describe the Fluent Abstract Machine that is the CRCW PRAM model supported and then we discuss how it is implemented.

2.1 The Fluent Abstract Machine

The Fluent Abstract Machine has N processors represented by the integers $\{1, \ldots, N\}$ which are connected to a shared memory. The processors operate synchronously. In each cycle every processor can issue either a *multiprefix operation* or a *set operation.* A multiprefix operation is of the form $MP(A, v, \otimes)$ where A is a shared variable, v is a value, and $\otimes$ is a binary associative operator. Let $P = \{p_1, \ldots, p_k\}$ be a set of processors such that $p_1 < p_2 < \ldots < p_k$. Assume at a given cycle that every $p_i \in P$ executes the operation $MP(A, v_i, \otimes)$ and that no other processor refers to A. Let v_0 be the value of A at the beginning of the cycle. Then, at the end of the cycle, every $p_i \in P$ will receive the value $v_0 \otimes v_1 \otimes \cdots \otimes v_{i-1}$ and A will have the value $v_0 \otimes v_1 \otimes \cdots \otimes v_k$.

We introduce two more primitives that are special cases of multiprefix: $READ(A)$ that returns the value of A and $WRITE(A, v, \otimes)$ that is equivalent to $MP(A, v, \otimes)$ except that no values are returned to the processors executing it. Note that a standard overwrite is a special case of $WRITE$; it can be expressed as $WRITE(A, v, \circ)$ where "$\circ$" is the binary, associative "overwrite" operator defined by $x \circ y = y$ for all x, y. Thus, the Fluent Abstract Machine supports ordinary concurrent reads and writes to the common memory and it can be programmed exclusively with these if desired.

We can note that the behavior of the Fluent Abstract Machine is totally deterministic, also in the case of memory conflicts. If, for instance, several processors write to a particular memory location in the same cycle, the writes will always take place in the order the processors are numbered.

For a description of the set operations supported we refer to Ranade, Bhatt, Johnsson [16].

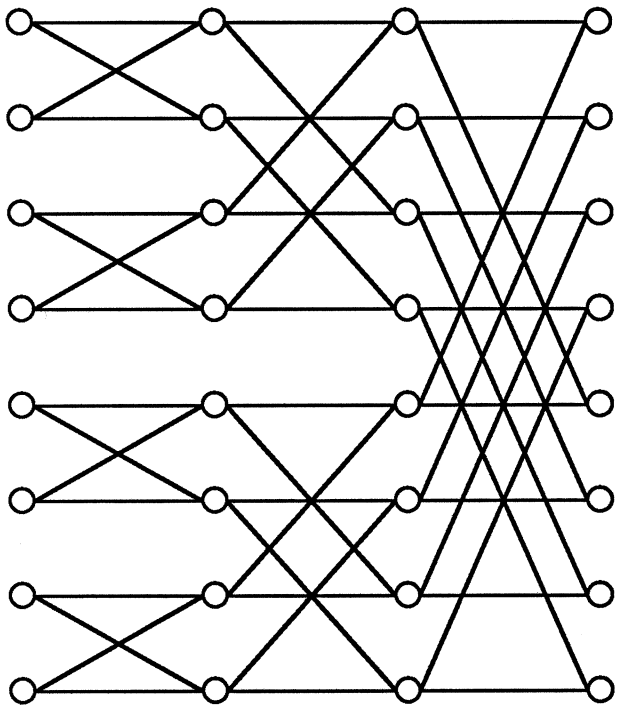

Figure 1: A butterfly network

2.2 Implementing The Fluent Abstract Machine

The Fluent Abstract Machine is implemented on an n-dimensional butterfly network with $(n+1)2^n$ nodes. In such a network every node can be given a two-dimensional coordinate $\langle c, r\rangle$ where $0 \leq c \leq n$ and $0 \leq r < 2^n$. We will say that a node with label $\langle c, r\rangle$ is at *level* c. A node $\langle c, r\rangle$ has a *node address* that consists of the binary representation of c concatenated with the binary representation of r. Every node $\langle c, r\rangle$, except for the ones at level n, are connected by forward links to the two nodes $\langle c+1, r\rangle$ and $\langle c+1, r \oplus 2^c\rangle$ where "$\oplus$" denotes bitwise exclusive or. Thus, each node in level c, except for the levels 0 and n, has two connections to level $c-1$ and two to level $c+1$.

Each node consists of a processor with memory and six routing switches. Each switch has two inputs and two outputs. Every switch is used in a distinct *routing phase* [15,16].

The fluent memory space is physically distributed among the local memories of the nodes. Every physical adress is of the form $\langle p, a\rangle$ where p is the address of a node and a is an address in the local memory at p. Fluent adresses are translated to physical by a hash function that randomizes the access pattern and ensures that all nodes hold about the same number of fluent memory locations.

Fluent memory requests are implemented by messages that are sent from the requesting processor and forwarded to the node holding the requested address. In the case of reads and multiprefixes, a reply message with the requested data is sent back the same way. The routing takes place in six phases [15,16]. In every phase the message is passed through the corresponding switches throughout the network. Every switch has two FIFO queues, one

for each input line, where messages in transit are buffered. If a queue becomes full, flow control is provided that causes any further messages over that line to be held until there is space in the queue again.

A fluent request from $\langle c, r\rangle$ to $\langle c', r'\rangle$ is routed as follows: in the first phase it is sent forward to $\langle n, r\rangle$. In phase two it is directed the unique path back to $\langle 0, r'\rangle$. In phase three it is sent forward again to $\langle c', r'\rangle$. The next three phases simply retrace the message back through the network in case the message required a reply.

The path followed in phase two deserves a closer examination. In every step the switch looks at the appropriate bit of the binary representation $r'_0 r'_1 \cdots r'_{n-1}$ of r' and directs the message accordingly. This leads to a path where the message after i hops will be at node $\langle n-i, r'_0 r'_1 \cdots r'_{i-1} r_i \cdots r_{n-1}\rangle$.

Messages have three fields: DEST, TYPE and DATA. DEST holds the physical address of the requested variable. The TYPE field contains the type of request, for instance READ, WRITE or MP.

Consider the action of the network during the emulation of one cycle of the Fluent Abstract Machine. All processors will issue memory requests of some type during the cycle. Any two processors may or may not issue requests to the same fluent address. The propagation of the requests is governed by two key ideas of fluent routing: *message sorting* and *message combining*. Every switch always maintains its input queues sorted so that messages with smaller destination addresses are stored first. At every time three situations can occur: first, both queues may be non-empty and the first messages have different destinations. Then, the message with the lower destination is forwarded. Second, if the first messages in both queues have the same destination they are *combined* and the combined message is transmitted. If, finally, any queue is empty no message is forwarded (since possibly a message with lower destination may arrive to the empty input later and thus the sortedness would be violated if the first message is prematurely forwarded).

The combination principle and the messages being sorted implies that only one request to a given fluent address is transmitted over any connection. This greatly reduces the risk of message congestion when many processors request the same fluent address. On the other hand, this puts some restrictions on the kind of requests that can be issued to the same fluent address in the same cycle [16]. The combination principle is also the reason why binary operators in multiprefix operations must be associative; since messages may combine at different stages depending on where they are issued, this is required to guarantee the uniqueness of the result.

Fluent routing as sketched above is provably fast; the following theorem is due to Ranade [16]:

Theorem 1 (Ranade) *Assuming a perfect random access map, the probability that any memory reference takes more than* $15 \log N$ *steps is less than* N^{-20}.

So far we have discussed the implementation of a single abstract memory cycle. Requests belonging to different cycles are kept separate from each other by *end-of-stream messages* (EOS). Immediately after a processor has sent a memory request it also issues an EOS message. Such a message has destination ∞. Thus, all nodes issue an EOS in each cycle so every node will eventually receive one. Because switches keep messages sorted the EOS will be the last request to go out from any switch during a particular cycle.

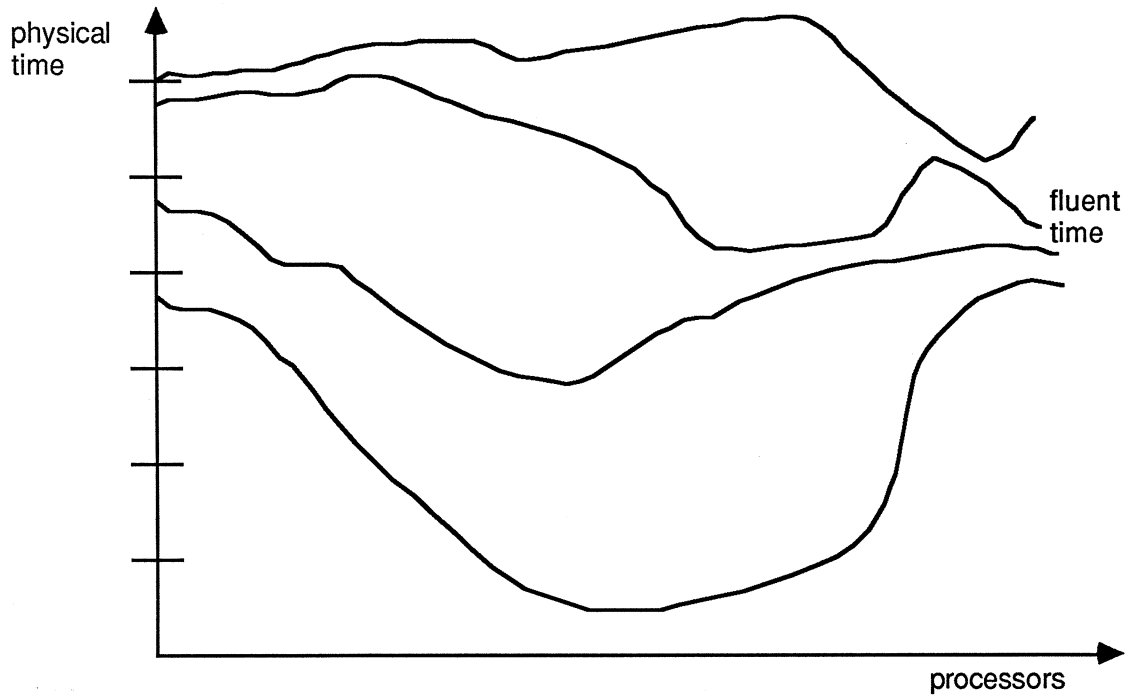

Figure 2: Fluent time vs. physical time

This property of EOS messages can be used to implement a global, distributed clock. Since only one EOS is forwarded through a switch during a cycle, the switch can keep a count of how many have passed. This count is a global time since it enumerates the current cycle for which the switch transmits messages. Guided by this the processor at the node in question can inject its proper request for that cycle. A processor can also use this mechanism to stop the global clock simply by withholding the EOS.

3 Explicit routing

Let us first define what we mean by explicit routing:

Definition 1 *A memory request that is* explicitly routed *(or* e-routed*, for short) has its path completely specified through the network.*

Thus, we may for instance route a message between $\langle c, r\rangle$ and $\langle c-2, r\rangle$ explicitly as $\langle c, r\rangle \rightarrow \langle c-1, r\rangle \rightarrow \langle c-2, r\rangle$. E-routing should not be mixed up with the following concept:

Definition 2 *A memory request that uses* absolute-addressed fluent routing *has a physical destination address (processor ID + local address) but uses the fluent routing scheme.*

An absolute-addressed request is a fluent request in all respects except that its address is in the physical address space instead of the fluent address space. To illustrate the difference between absolute-addressed requests and e-routed requests, consider an absolute-addressed

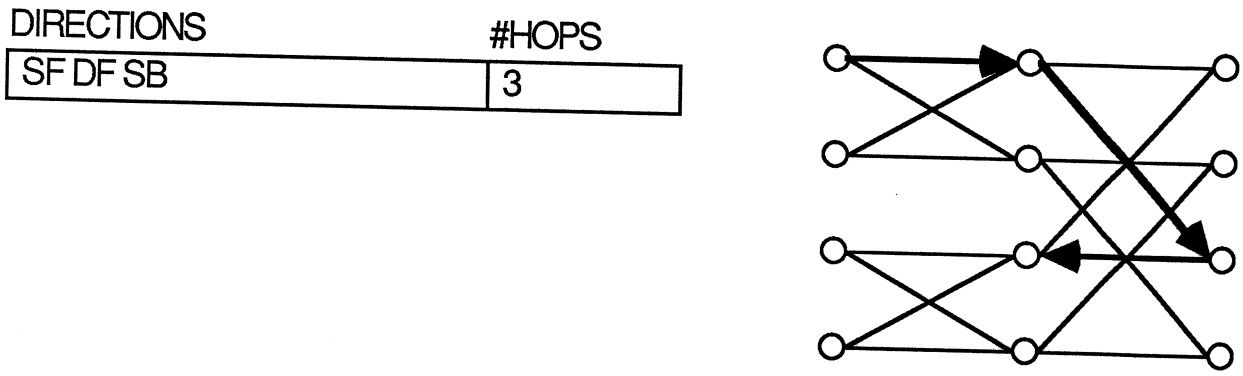

Figure 3: An example of a path representation

request from $\langle c, r\rangle$ with destination $\langle c-2, r\rangle$. According to the fluent routing scheme [15,16] this message will be sent first using forward links to $\langle n, r\rangle$ (where n is the dimension of the butterfly), then backwards to $\langle 0, r\rangle$ and finally forward to $\langle c-2, r\rangle$. This should be compared with the path of the e-routed request between the same two nodes above.

Absolute-addressed requests pose no problems whatsoever in the fluent environment since they are treated exactly as fluent requests. The only thing that differs is that no address translation is necessary before the request is sent out. E-routed requests is another matter. Typically they will be used for computation structures that have good embeddings in the network. These embeddings will in general imply communication routes that are different from the fluent ones. It is therefore clear that e-routed requests should *not* be treated by the switches in the same way as fluent and absolute-addressed requests.

3.1 Implementing explicit routing

We propose three new types of requests distinguished by new codes in the TYPE field: E-WRITE, E-READ and E-REPLY. E-WRITE and E-READ are e-routed writes and reads, respectively. An E-REPLY is the reply message to an E-READ carrying the requested data.

The DEST field for an e-routed request is different from that of a fluent request. Instead of an absolute physical address – a pair consisting of a processor address and a local memory address, the DEST field contains a *path representation* and a local memory address. The path representation holds the explicit route to the destination node and the local memory address refers to the memory of that node.

A path representation consists of two fields: #HOPS that contains the number of hops, say n, in the route and DIRECTIONS that holds n *direction pointers* $\mathrm{DP}(1), \ldots, \mathrm{DP}(n)$. For every hop i in the route $\mathrm{DP}(i)$ tells which neighbor of the current node the message should be sent to. This is very akin to self-routing using control tags in permutation networks [6].

Since a node $\langle c, r\rangle$ in a butterfly network has at most four neighbors, a direction pointer can have four possible values; straight forward (SF) to $\langle c+1, r\rangle$, diagonally forward (DF) to $\langle c+1, r \oplus 2^c\rangle$, straight backward (SB) to $\langle c-1, r\rangle$ and diagonally backward (DB) to $\langle c-1, r \oplus 2^{c-1}\rangle$. See figure 4. Thus, it can be encoded with two bits.

The butterfly network has the property that if a is a neighbor of b of a certain type, say

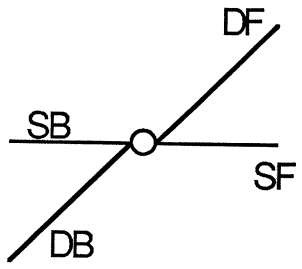

Figure 4: Directions from a butterfly network node

SF	$\mapsto$	SB
SB	$\mapsto$	SF
DF	$\mapsto$	DB
DB	$\mapsto$	DF

Table 1: Conversion of neighbor types when a path is reversed

X, then b's neighbor type vs. a is uniquely given by X *independent of* a *and* b. If a is a forward (backward) neighbor of b, then b is always a backward (forward) neighbor of a. If a is a diagonal (straight) neighbor of b, then b is always a diagonal (straight) neighbor of a. This property is important when reversing paths, see below.

When a switch detects an incoming e-routed request (by checking the TYPE field) it lets the request bypass the queues for the fluent requests. It then decreases #HOPS by one. If #HOPS then is zero the request has reached its target and the local memory access is executed. Otherwise, the request is redirected to the neighbor node given by DP(#HOPS).

An E-REPLY message is routed backwards the same way as the E-READ request whose requested data it carries. To make it possible to restore the path the E-READ request carries the original value of its #HOPS in its DATA field. When the E-READ reaches its destination its DATA field is copied to the #HOPS field of the E-REPLY, thus restoring the length of the path. The DIRECTIONS field of the E-REPLY is created as follows: all DP(i) of the E-READ are converted according to table 1 and copied to position DP(#HOPS$-i+1$) of the DIRECTIONS field of the E-REPLY. It is not hard to convince oneself that this is the correct reversed path representation. Finally the DATA field is filled with the locally retrieved data, the TYPE is set to E-REPLY and the request is sent back to the node issuing the original E-READ request.

3.2 Additional hardware requirements

In order to handle e-routed requests the switches will require additional circuitry for the following:

- Recognizing the types for e-routed requests.

- Letting e-routed requests bypass the queues for fluent messages and putting them on an output line while withholding the fluent message that would have used that line otherwise.
- Decoding direction pointers and decrease #HOPS fields.
- Restoring path representations for E-REPLY's from E-READ's.

Depending on how fast the switches will handle the e-routed requests, a separate queue may be necessary to hold waiting e-routed requests. Whether to have a queue or not is a matter of performance, the flow control for fluent messages will work also for e-routed requests and guarantee that no messages are lost. It is also possible that it is beneficial to make the queues for fluent requests longer if fluent messages are frequently withheld because of e-routed messages passing.

Some more local memory will probably be needed if e-routing is used. This is because a computation using e-routing is likely to use its own explicit local addressing as well, and we do not want to have fluent variables overwritten. Thus, it seems necessary to partition the local memory into a fluent part and a "direct-addressed" part. As a matter of fact this can be beneficial for processors issuing fluent requests only too, since they may need some fast (compared with fluent access) local memory to hold temporary values during computations. A processor accessing its own local memory directly can be seen as issuing an e-routed request with zero hops.

This idea of partitioning the memory resembles the partitioning of memory in IBM's experimental multiprocessor system RP3 into a global and a local part [13]. As in RP3 it can be advantageous to have a "movable boundary" between the two memory types, so that memory can be re-partitioned to meet the demands of a particular application. Note, though, that the non-fluent part of the memory in a fluent node is accessible also to other nodes via e-routed requests. Thus the memory is partitioned in a non-fluent and a fluent part rather than in a local and shared part.

3.3 Message sizes

If explicitly routed requests are added it may be necessary to have slightly longer messages. First, an additional bit may be necessary for the TYPE field since there will be three more types to encode. Second, depending on how long we want the paths to be, the DEST field may have to be larger.

Assume that we allow a maximum of m hops. A path representation of m hops requires $2m + \lceil \log m \rceil$ bits, $2m$ for the DIRECTIONS field and $\lceil \log m \rceil$ for the #HOPS field. The diameter of an n-dimensional butterfly network, consisting of $n + 1$ levels of 2^n nodes each, is $2n$. Thus, if any node is to be able to send e-routed requests to any other node, then $4n + \lceil \log 2n \rceil$ bits are maximally needed in the path representation. This should be compared with the size of a node address for a fluent request that is $\lceil \log(n + 1) \rceil + n$ in the same network. The proposed Fluent-I [16] has a 13–dimensional butterfly network. In this machine a node address requires 17 bits while the longest required representation path needs 57 bits.

E-routed requests are intended to be used primarily for local or lear-local communication, however. If we allow variable-size DIRECTIONS fields then the size of the path representation for a k-hops request is $2k + \lceil \log 2n \rceil$. With $n = 13$ a one-hop request needs 7 bits and a two-hop request needs 9 bits. Another possibility is to have fixed-size fields but to restrict the longest path to some small number m. If we require the path representation to fit into the 17 bits of the node address field for fluent requests in the Fluent-I, then m can be 7 at the most.

3.4 Passing explicit messages in a pipelined manner

So far explicit routing has been described as a strict store-and-forward process, where the whole message is stored completely in a node as a single packet before it is transmitted to the next node in the path. As a matter of fact explicitly routed messages can be transmitted in a pipelined fashion, analogous to *wormhole routing* [3].

In this scheme every explicit message consists of several packets. During a transmission these packets are spread out over a sequence of nodes in the path. The first packet contains the path representation. The direction pointers are used as flow control digits. Therefore the DIRECTIONS field should come first in the path representation. This field should be lead by the direction bits and these should in turn be arranged so that direction bits specifying early hops in the path come before those specifying later hops. With the address arranged in this fashion the first direction pointer can simply be "chopped off" when the first packet enters a new node. This pointer is then stored in the switch while the packets pass and it is used to direct them to the correct output line. A special marker at the end of the last packet tells the switch that the message has passed completely; it can then discard the direction pointer and start transmitting a new message.

This implementation of explicit routing has the advantage of being compatible with the proposed bit-serial implementation of the fluent routing switch [16].

3.5 Operations that can be implemented by explicitly routed requests

Three different operations can be implemented using explicitly routed requests: *write*, *read with wait for completion* and *read without wait for completion.* The implementation of write is straightforward: an E-WRITE is simply issued to the desired destination. The read operations send out an E-READ and retrieve the E-REPLY that comes back. A read with wait (for completion) issues an E-READ and halts the processor until the E-REPLY comes back. A read without wait allows the processor to proceed without waiting for the E-REPLY. In this case some kind of interrupt mechanism is necessary that interrupts the processor and invokes the proper code when the E-REPLY returns.

E-REPLY messages have an important property: *no E-REPLY will be sent to a processor that did not issue an E-READ to the address from which the E-REPLY carries data.* This is because E-REPLY messages are sent only in response to E-READ requests and every E-REPLY follows exactly the same path backwards as the corresponding E-READ came. Thus, we can state the following:

Proposition 1 *If a processor p has at most one outstanding E-READ request at any time, then any E-REPLY arriving to p will be the answer to the latest E-READ issued by p.*

We immediately get the following result for reads with wait:

Corollary 1 *If a processor p only executes read requests* with *wait, then any E-REPLY arriving to p will be the answer to the latest E-READ issued by p.*

If, however, a processor has several outstanding E-READ requests at the same time, a situation that may occur if it executes reads without waiting for completion, then there is no guarantee that the corresponding E-REPLY's will come back in the same order as the E-READ's were issued. This is because different requests may follow paths of different lengths, with a different amount of interfering traffic. We do, however, have the following result:

Proposition 2 *Assume that switches send out e-routed messages in the same order as they arrive. Then, if the E-READ request r_1 is issued before the E-READ request r_2 by the processor p* and *if the path of r_1 is subsumed by the path of r_2, the E-REPLY of r_1 will arrive to p before the E-REPLY of r_2.*

Proof. Assume that r_1 is issued before r_2 and the path of r_1 is subsumed by the one of r_2. Because switches preserve the order between messages r_1 will arrive to its destination d before r_2 arrives to d. Thus, the E-REPLY for r_1 will leave d before r_2 is forwarded by d. This implies that r_2 will reach *its* destination after r_1's reply is issued, and thus, r_2's reply will also be issued after r_1's reply is. Since the reply paths are the reversed request paths the path of r_1's reply will be subsumed by the path of r_2's reply. The reasoning above that showed that r_1 arrives to d before r_2 can now be applied to the replies to show that the reply to r_1 arrives to p before the reply to r_2 does. ■

The proposition for E-WRITE's below is proved in the same way as proposition 2.

Proposition 3 *Assume that switches send out e-routed messages in the same order as they arrive. Then, if the E-WRITE request r_1 is issued before the E-WRITE request r_2 to the same processor p* and *if the path of r_1 is subsumed by the path of r_2, r_1 will arrive to p before r_2.*

In general, however, some other mechanism must be used to distinguish between E-REPLY's than to rely on them going the same way. One possibility is to restrict the read-without-wait instruction so that after such an instruction is executed the processor continues, with one outstanding read request, *only until another read instruction is encountered.* When this happens the processor halts and waits for completion of the first request until the new read instruction is executed. This assures that at most one read request is outstanding at any time which totally determines the arriving order of the replies according to proposition 1.

Another possibility is to add an *ID field* to each E-READ and E-REPLY message and let the switch of the destination node copy the ID from the E-READ to the returned E-REPLY. If the ID field is i bits wide a processor can have a register whose contents is

incremented by one modulo 2^i and copied to the ID field every time an E-READ is issued. In this way there can be 2^i consecutively issued outstanding E-READ requests at the same time whose E-REPLY's all are uniquely identifiable. Note, though, that the administration of these replies will require some enhancements of the hardware; either the switch must be able to queue and sort incoming E-REPLY's with respect to their ID so that the processor is guaranteed to recieve them in order, or the processor itself must be equipped to handle multiple choices when an E-REPLY interrupt occurs.

3.6 Synchronization of explicitly communicating processors

When two processors communicate through fluent variables there is a strong synchronization given by the global, distributed clock implemented by the EOS messages. Each processor knows which fluent time slice it is in and can, if it has knowledge about what other processors are doing in which time slice, use that information to determine whether a desired value is available or not. The fluent clock does *not*, however, apply to explicitly routed messages since they are forwarded independently of the fluent EOS messages. Thus, processors using e-routed requests to communicate cannot make assumptions about the result of requests based on this clock.

Instead processors communicating via e-routed requests will have to rely on conventional methods for synchronization. These differ depending on the type of program execution that the fluent machine will use. The type of execution is not currently fixed. If the communicating processors execute in SIMD mode, for instance, then there is some global controller that has information about the instruction stream and holds the next instruction until the previous one has completed. If the processors run as a MIMD system, then they can synchronize via semaphores or test-and-set, some of which probably should be provided as a separate type of e-routed request.

4 Problems with mixing explicit and fluent routing

In this section we will identify some problems that may occur when fluent routing and explicit routing are used at the same time.

4.1 EOS depletion

If a processor only sends out e-routed requests and does not issue any fluent requests, then it will not issue any EOS messages either. Since the global distributed clock relies on that all processors send EOS regularly this means that the global fluent clock will stop and all processors that are doing fluent computations will eventually stop too, when they reach the last defined fluent time slice. This is a not a problem specific to e-routed communication; a fluently communicating processor that runs a computation-bound program and issues few fluent requests per time unit will also cause the same problem. It seems likely, though, that the problem will be particularly serious when processors utilize e-routed communication because of the type of computations that are likely to be done using such routing: long, heavy computations with a largely predetermined structure that is embedded in the network and where communication almost exclusively is accomplished through e-routed messages.

Some general methods to handle EOS depletion have been suggested [11] and they are applicable to processors sending e-routed requests. The idea is to have a "non-communicating" mode for nodes (where "non-communicating" means "not fluently communicating"). A node in non-communicating mode will simply pass on every EOS that arrives without injecting its own fluent message in that fluent time-slice as otherwise would be required. This has two consequences. The first is a positive one: fluent requests will not be halted at the switch anymore. The second consequence is negative, the node will lose its fluent synchronization. Whenever it will want to issue fluent requests again it must resynchronize in some way. This can be done using standard techniques like semaphores implemented with fluent variables.

4.2 Slowdown of fluent requests

In the implementation of e-routing sketched in the previous chapter, e-routed requests will always bypass the fluent queues and thus e-routed requests will always have priority over fluent requests. If many e-routed requests per time unit are transmitted over a particular line, then very few, if any fluent requests will be able to pass that line. As a matter of fact there is *no* guarantee that *any* fluent request will ever be able to be completed in such a system because of this possibility of livelock.

A possible remedy is to *interleave* fluent and e-routed messages when necessary. That is, when there are both fluent and e-routed messages waiting to be transmitted over a certain line then the switch alternates the type of the messages sent every time step until one of the queues are empty. (This solution definitely will require a separate queue for e-routed messages in each switch.)

It is also possible to have strategies where n_e e-routed messages are transmitted for every n_f fluent message transmitted when conflicts arise. The selection of n_e and n_f is a matter of what priority is desirable to give the different types of communication; maybe this should be possible to set programmatically in the switch.

Tuning the priorities for requests

Under some simplified assumptions we can analyze the tradeoff and determine the optimal ratio n_f/n_e. Our objective is to minimize the total execution time. Assume the following:

1. Our fluent machine has N processors. At time 0 we reserve a group P of k processors to execute E operations each using e-routing. The execution pattern is such that each of these processors completes its task at the same time, which is E if no e-routed request are delayed. All processors in P transmit n_e e-routed requests for any n_f fluent requests they transmit in case of conflicts.

2. We also have F fluent operations to execute. Each fluent operation can be scheduled to be executed at any time, either in one of the $N-k$ "fluent" processors not in P while the "non-fluent" processors in P are executing e-routing operations or in any of the N processors after P has completed.

3. All operations are communication-bound, so that the operations are slowed down the amount the communication is slowed down.

4. Communication lines are saturated all the time so that the n_e/n_f ratio of transmitted requests actually holds for non-fluent processors.

5. All e-routed requests are routed so that they never pass a fluent processor not in P.

6. Define the (normalized) speed of a fluent request to be $v_1 = 1$ when it passes a fluent processor and $v_2 = n_f/(n_e+n_f)$ when it passes a non-fluent processor. We assume that any fluent message that passes p processors passes $l_1 = p(N-k)/N$ fluent processors and $l_2 = pk/N$ non-fluent processors, including EOS messages.

The average speed $\overline{v}$ for any fluent request will then be

$$\frac{v_1 v_2 (l_1 + l_2)}{v_2 l_1 + v_1 l_2} = \frac{N n_f/(n_e + n_f)}{(N-k) n_f/(n_e + n_f) + k}. \tag{1}$$

Consider now the situation when the processors in P are executing non-fluent operations. The e-routed requests sent between processors in P will be slowed down by the factor $n_e/(n_e + n_f)$ since fluent requests are interleaved with e-routed requests. Because of the assumption about operation speed being communication bound P will thus use the time $E(n_e + n_f)/n_e$ to complete the non-fluent task. During this time the speed of the fluent operations will be $\overline{v}$ because of the slowdown of the fluent clock caused by EOS messages passing non-fluent processors. Thus, a maximum of

$$\overline{v} E \frac{n_e + n_f}{n_e}$$

fluent operations are executed while P is executing non-fluent operations. If the total number F of fluent operations is less than this amount, then the total execution time will indeed be

$$E \frac{n_e + n_f}{n_e}. \tag{2}$$

If not, then there are $F - \overline{v} E(n_e + n_f)/n_e$ fluent operations left to execute at this time. All N processors will be available to do this and they will all run at speed 1 again, which yields an additional execution time of

$$\frac{(F - \overline{v} E(n_e + n_f)/n_e)}{N}. \tag{3}$$

If we sum (2) and (3), substitute the expression (1) for $\overline{v}$ and simplify we obtain the following expression for the total execution time as a function of n_f/n_e:

$$\begin{aligned} & E + \frac{F}{N} + \frac{n_f}{n_e} E \left(1 - \frac{1}{(N-k) n_f/(n_e + n_f) + k}\right) \\ = \; & E + \frac{F}{N} + \frac{n_f}{n_e} E(1 - \alpha), \qquad \text{where } \alpha \leq 1. \end{aligned} \tag{4}$$

The total execution time will thus be given by either (2) or (4). In either case it is minimized if we choose $n_f/n_e = 0$, that is: $n_f = 0$ and $n_e \neq 0$.

Therefore, under the assumptions above, the total execution time is minimized if e-routed requests are always given unconditional priority over fluent requests. This may seem

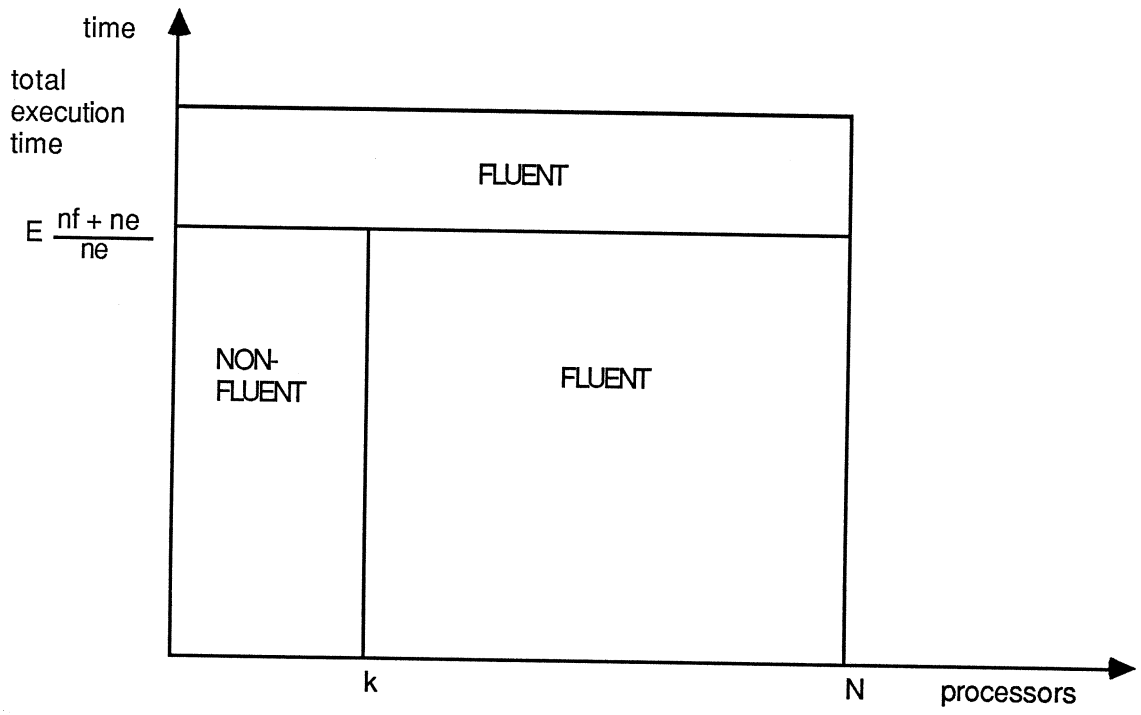

Figure 5: Phases of execution

a bit counterintuitive in the light of the preceding discussion about livelock. Note, that $n_f = 0$ implies that $\overline{v} = 0$, i.e. the fluent clock is completely stopped while the non-fluent processors are executing. One should be aware though, that the assumptions are very idealized and do not reflect the real situation always. Especially assumption 2, that the fluent operations can be freely rescheduled to be executed at any time on any processor, is not entirely realistic. In a real situation it is likely that data dependencies will pose strong restrictions in which order the operations can be executed. Nevertheless, the analysis gives a hint that it might be advantageous to reschedule computations when possible so that there will be phases with mostly fluent computations alternating with phases where most of the machine is executing e-routed computations and e-routed requests are given high priority.

Partitioning the machine

The analysis in the previous section indicates that it is advantageous to separate fluent and non-fluent computations so that communication conflicts between the different types of requests are minimized. One way to do this is to separate the type of computations in time. Another way is to separate them in space by *partitioning* the machine into one part reserved for fluent computations and one part reserved for non-fluent computations. This requires that the mapping of fluent addresses to physical addresses is changed so that no fluent variables are stored in the part of the machine reserved for non-fluent computations. Furthermore, the partitioning should preferably be done in such a way that no (or very few) fluent messages between fluent processors ever will have to pass a non-fluent processor and

vice versa.

The partitions can communicate with each other in the following way: non-fluent processors can send fluent requests to read fluent variables, and fluent processors can send absolute-addressed requests to read memory locations in non-fluent processors. If this is to work the general routing scheme that allows any processor to send fluent messages to any other processor can not be changed. Only the fluent-to-physical memory mapping is to be altered.

Since the fluent memory map is different when the machine is partitioned it seems that a partitioning has to be static throughout a program execution. Changing the memory map on the fly is bound to wreak havoc.

How to partition the network is an interesting problem. Other criteria than minimizing message interference are important, like for instance how well a given computation structure can be embedded in the assigned partition. We do not address this issue here. Instead we will characterize a class of partitions for which it is guaranteed that fluent requests will only pass through fluent processors. First we need a formal definition:

Definition 3 *A subgraph of the butterfly network is* closed under fluent routing *iff the fluent route between any two nodes in the subgraph constains edges in the subgraph only.*

Lemma 1 *A subgraph that contains only a part of a row in the butterfly network can not be closed under fluent routing.*

Proof. Consider a node $\langle c, r\rangle$ in the n-dimensional butterfly network that sends a fluent request to itself. This request will first traverse all nodes $\langle c+1, r\rangle$, $\langle c+2, r\rangle$ etc. up to $\langle n, r\rangle$. Then, it will go back in the network through all nodes in row r, back to $\langle 0, r\rangle$. Finally, it will be routed forwards to $\langle c, r\rangle$.

Thus all nodes in row r will be traversed. It follows that a subgraph that contains $\langle c, r\rangle$ but not the node $\langle c', r\rangle$ cannot be closed under fluent routing. ■

By lemma 1 all subgraphs closed under fluent routing are formed from full rows in the butterfly network. Therefore, we can equally well consider rows instead of individual nodes when discussing these subgraphs. In the following "$\wedge$" and "$\vee$" denote bitwise "and" and "or", respectively.

Theorem 2 *For any $0 \leq M, M' < 2^n$, the subset $\{\, (r \vee M) \wedge M' \mid 0 \leq r < 2^n \,\}$ of rows of the n-dimensional butterfly is closed under fluent routing.*

Proof. Any row q in $S = \{\, (r \vee M) \wedge M' \mid 0 \leq r < 2^n \,\}$ is such that if M'_i (the i:th bit of M''s binary representation) is zero, then q_i is zero. If M'_i and M_i are one, then q_i is one. M and M' thus fixes the values of certain bits in the binary representation of all rows in S. Consider now the fluent route from the row $q = q_0 q_1 \cdots q_{n-1}$ to $q' = q'_0 q'_1 \cdots q'_{n-1}$. By the bitwise routing in the second routing phase [15,16] this path will look as follows as expressed in the bitwise representations of row coordinates:

$$
\begin{array}{c}
q_0 q_1 \cdots q_{n-1} \\
q'_0 q_1 \cdots q_{n-1} \\
q'_0 q'_1 \cdots q_{n-1} \\
\vdots \\
q'_0 q'_1 \cdots q'_i q_{i+1} \cdots q_{n-1} \\
\vdots \\
q'_0 q'_1 \cdots q'_{n-1}
\end{array}
$$

It is immediately clear that if $q_i = q'_i$ for some $0 \leq i < 2^n$, then all rows on the path between them will have the same value for that bit. Hence, if both q and q' are in S, then all the bits fixed in them by M and M' will be fixed for all rows in the path between them as well and the path will be wholly in S. Thus, S is closed under fluent routing. ■

Theorem 2 characterizes a class of subgraphs that can host fluent computations without interfering with the rest of the network. Since the number of rows in the subgraph is halved every time a bit is fixed, subgraphs of this form can be used to partition the machine into pieces with $(n+1)2^i$ nodes each for any $0 \leq i \leq n$. Different partitions can host either fluent or e-routed computations. Besides giving the capability to run fluent and e-routed computations side by side this also makes it possible for several users to share the machine by using one partition each.

4.3 Processor fragmentation

A well-known problem that can occur in time-sharing systems is memory fragmentation. If memory is assigned to, and released from, different tasks in an uncontrolled fashion, a situation can occur where the free memory is scattered in small pieces throughout the memory space. Thus, a process that requires a certain amount of contiguous memory may not find it even though the total amount of memory needed is free. It is clear that a similar situation can arise in the fluent machine with respect to free processors when a task using a fixed communication pattern is to be allocated. Since such a task requires that the participating processors are configured in a certain way it may very well happen that there is no place in the machine where the task fits even though the total number of needed processors is available. In analogy with the term for the corresponding memory problem we will call this *processor fragmentation.*

Note that processor fragmentation is not a problem when only fluent routing is used. Since fluent memory is arbitrarily distributed throughout the whole network the relative position of processors is irrelevant and free processors can simply be kept in a pool and assigned when needed.

A possible remedy is to adopt a processor assignment strategy similar to the one used to alleviate the corresponding memory problem: namely to use some sort of *paging.* Processors can be assigned and released in predefined chunks rather than in the exact amount required for each task. It should be noted, however, that the processor fragmentation problem is potentially harder to cope with than the corresponding memory problem. A memory is essentially a linear space where tasks need contiguous segments. Processors may on

the other hand, depending on the embedded computation structure, need to be assigned in a bewildering number of possible patterns. Still it seems that a paging strategy that preserves locality within pages is likely to be successful since computation structures often are embedded just so locality can be utilized. A possibility is also to provide a number of standard structures in which processors can be allocated, like for instance paths, cycles, binary trees and sub-butterfly networks.

4.4 Interfacing pipelined computations with fluent data structures

Explicit routing can be used to implement computations that use local point-to-point communication in the butterfly network of a fluent machine with N processors. Such communication takes $O(1)$ time instead of the $O(\log N)$ expected time to access a fluent variable. Typically, a computation that uses local communication is pipelined. Examples are systolic computations of various kinds and pipelined computations on binary trees. For a pipelined computation *throughput* is important. Consider the execution of m identical tasks on a k-stage pipeline. Assume that each stage, including inter-stage communication, takes time s, and that a new task cannot be initiated until d time units after the initialization of the previous one. The total time to execute the m tasks is then $dm + sk$. Typically m is much larger than k. This means that d is an important parameter since the time essentially will be $O(dm)$.

Consider now a pipeline whose indata are stored in fluent variables. This means that at some stage fluent requests must be issued to transmit the data to the processor at the entry of the pipeline. The first solution that comes to mind is to let the processor at the entry of the pipeline issue a fluent read every time a new task is initiated and wait for the read to become completed. This is, however, *not* a good idea since this will cause d to be $O(f \log N)$, where f is the time for a fluent cycle, so the whole computation will take $O((f \log N)m + sk)$ time.

The processor does not, however, necessarily have to wait for the read to be completed. Because fluent messages are kept sorted all the time replies will arrive in the same order as the requests were sent out [16]. If a local buffering mechanism is provided the processor can issue several reads without waiting for completion. The processor can then read from the buffer when data is available. If the buffer is placed in the processor's local memory the reads from it will take $O(1)$ time. Thus, if sufficiently many reads are issued to keep the buffer non-empty all the time, the time from the first read is issued to the computation is completed will be $O(\max(f, s)m + f \log N + sk)$.

The buffer could simply be implemented as a queue in local memory. The switch should be able to insert items in the queue and test if it is full. The processor should be able to perform a variety of operations on the queue: inserting and taking out elements, test if it is full/empty, delete the first/last element and possibly others. The technique for having two processes manipulating a queue concurrently is well known and will cause no problems to implement.

This type of buffer could also be used to interface different pipelines to each other: if the path between them is known one pipeline could issue e-routed writes to the input buffer of the other pipeline. If the writes are sent via the same path they will arrive to the destination in the same order as they were sent according to proposition 3.

The introduction of buffers calls for a new family of requests whose destinations are buffers instead of simple memory locations. The reason is that the switch upon arrival of a request must be able to determine whether just to write the contents into the specified address in the local memory or to invoke the queue insertion mechanisms. Fluent read-to-buffer and e-routed write-to-buffer (or e-routed read-to-buffer, for that sake) are explained above and pose no special problems. Absolute-addressed fluent write-to-buffer, on the contrary, may be somewhat problematic to use.

The reason is that it is a little complicated to guarantee that fluent writes-to-buffer will arrive in the intended order. If the fluent writes w_i are supposed to arrive in the order $w_1, w_2, \ldots$ then they must be issued in fluent time-slices $t(w_i)$ such that $t(w_1) < t(w_2) < \ldots$. To make sure that this is true is not trivial; there are indeed cases where the time slice for the write can be determined in advance and the code can be laid out accordingly. However, generally this is not true and one will have to resort to potentially costly synchronization between fluent processors through semaphores or something equivalent. We can also note that it is not advisable to issue fluent writes to the same buffer in the same fluent time slice: these writes can *not* be combined as ordinary writes or multiprefix requests are unless potentially very long messages are accepted. Therefore, they will cause severe congestion close to the destination. The fluent routing mechanism must also be altered in this case to handle several requests going to the same address in the same cycle without being combined.

Still fluent writes to buffers are of interest. The reason is that buffers possibly can be set up in advance, before a pipeline starts to execute. Fluent writes to such buffers can then be used to pre-fill the buffer so the pipeline when it starts can begin to execute immediately without having to wait for the results of the first reads to come back. This will lower the execution time for the pipeline to $O(\max(f,s)m + sk)$ when applicable. (Or even to $O(sm + sk)$ if the buffer is big enough and writes to it began so well in advance that the pipeline will never empty it and have to wait for more coming in the next fluent cycle.) This technique of pre-filling buffers must however be considered quite advanced and be of use primarily as a tool for sophisticated performance optimization.

5 Conclusions

Fluent routing is an efficient and elegant way to implement a CRCW PRAM on a network of processors. Still there are situations where performance demands require total control over the communication and execution pattern. We have proposed a new type of *explicitly routed* (e-routed) requests in the fluent machine that gives this control. This request type can be implemented with moderate additions to the original hardware required for fluent routing.

A simple analysis indicates that even though e-routed requests and fluent requests can be transmitted in a mixed fashion, with both types of requests alternatingly being sent over the same line, the best performance is achieved if the interference between the request types is kept to a minimum. This can be accomplished by separating the types of communication in either *time* or in *space*. The latter implies that the machine must be statically partitioned in such a way that fluent requests never cross partition boundaries. We characterize such a class of partitions.

It should be noted that explicit routing is not the only possible way to send messages between two given processors. It is rather a "most simplistic" approach where the routing path is explicitly given for every message and it is up to the programmer (compiler, run-time system, ...) to choose the paths so that performance is not degraded. It is, although general, primarily intended to implement local or near-local communication when a good embedding of the computation structure is known. Other routing strategies may for instance be adaptive, that is: they react to the precence of other messages and will try to re-route messages around points of congestion in the network. The possible performance gain is, however, bought for the loss of the total control over the routing that explicit routing gives. We do believe that there are cases where total control over the communication is crucial to achieve maximum performance. Adaptive routing will also require additional administrative overhead. If congestion or "hot spots" (heavily accessed memory locations) is an issue then we recommend that pure or absolut-addressed fluent routing is used.

There are also non-adaptive routing schemes that are not explicit. An example is the routing in the Connection Machine [5], or fluent routing itself for that matter. Both have in common that the actual path is uniquely given by the source and destination addresses. If a routing scheme has this property, then there will be embeddings of computation structures where the routing requirements conflict with the paths given by the source and destination addresses. Complete freedom to experiment with different embeddings is possible only if full freedom is given to specify the message paths. We still have a very incomplete knowledge of how embeddings should be done in a butterfly network. Therefore it seems wrong to limit the possibilities at this stage, especially since the Fluent Machine is a research project and thus likely to be a testbed for different parallel strategies.

We want to mention, finally, that "smart" (but non-adaptive) routing strategies very well can be built on top of the explicit routing. Nothing prevents a program to calculate message paths at execution time. If we add the possibility to have several queues for every input line, then for instance deadlock-free wormhole routing strategies, according to Dally and Seitz [3], can be implemented on top of the pipelined explicit routing scheme proposed in section 3.4.

No matter what the processor-to-processor communication primitives will finally be, developing compiler techniques for the fluent machine will be a very interesting challenge. Compiling to the Fluent Abstract Machine without explicit routing is probably easier than compiling to other parallel architectures since communication then simply takes place through global, shared variables. The really interesting issue is to investigate compiler techniques that use local communication when adequate: how to determine if a program segment should be compiled into an embedded computation structure using processor-to-processor routing or if the fluent model should be used, how to rearrange computations to minimize message interference, how to decide what mechanism should be used to interface embedded computation structures with fluent data. This is a potentially rich field for research.

6 Acknowledgements

I would like to thank Sandeep Bhatt, Lennart Johnsson, Michael Littman and Abhiram Ranade for suggestions and interesting discussions. My thanks also to Eileen Connolly for her editorial efforts. The generous support by the Office of Naval Research under contract No. N00014-86-K-0564 is gratefully acknowledged.

References

[1] S. N. Bhatt, F. R. K. Chung, J.-W. Hong, and F. T. Leighton. Optimal simulations by butterfly networks. In *20th ACM Symposium on Theory of Computing*, pages 192–204, May 1988.

[2] M. C. Chen, Y-I. Choo, and J. Li. *Compiling Parallel Programs by Optimizing Performance.* Research Report YALEU/DCS/TR-633, Dept. Comput. Sci., Yale University, June 1988.

[3] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.*, C-36(5):547–553, May 1987.

[4] J. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Dept. Comput. Sci., Yale University, 1985. Also published in the ACM Dissertation Award Series by MIT Press.

[5] W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[6] S.-T. Huang and S. K. Tripathi. Self-routing technique in perfect-shuffle networks using control tags. *IEEE Trans. Comput.*, 37(2):251–256, February 1988.

[7] P. Hudak, J-M. Delosme, and I. C. F. Ipsen. *ParLance: A Para-Functional Programming Environment for Parallel and Distributed Computing.* Research Report YALEU/DCS/RR-524, Dept. Comput. Sci., Yale University, March 1987.

[8] J. Li, M. C. Chen, and M. F. Young. *Design of Systolic Algorithms for Large Scale Multiprocessors.* Technical Report YALEU/DCS/TR-513, Dept. Comput. Sci., Yale University, February 1987.

[9] B. Lisper. *Synthesis of Synchronous Systems by Static Scheduling in Space-time.* PhD thesis, NADA, KTH, Stockholm, February 1987. TRITA-NA-8701.

[10] B. Lisper. *Time-Optimal Synthesis of Systolic Arrays with Pipelined Cells.* Research Report YALEU/DCS/RR-560, Dept. Comput. Sci., Yale University, September 1987.

[11] C. Metcalf. Fluent routing considerations. April 1988. Unpublished manuscript.

[12] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *CACM*, 29(12):1184–1201, December 1986.

[13] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): introduction and architecture. In *Proc. of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[14] P. Quinton and P. Gachet. *Automatic Design of Systolic Chips.* Research Report RR 450, INRIA, Rennes, October 1985.

[15] A. Ranade. How to emulate shared memory. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 185–194, October 1987. Also available as Technical Report YALEU/DCS/TR-578, Dept. Comput. Sci., Yale University.

[16] A. Ranade, S. Bhatt, and L. Johnsson. *The Fluent Abstract Machine.* Technical Report YALEU/DCS/TR-573, Dept. Comput. Sci., Yale University, January 1988.