Process And Dataflow Control In Distributed Data-Intensive Systems

William Alexander and George Copeland

MCC
3500 West Balcones Center Drive
Austin, Texas 78759

Abstract

In dataflow architectures, each dataflow operation is typically executed on a single physical node. We are concerned with distributed data-intensive systems, in which each base (i.e., persistent) set of data has been declustered over many physical nodes to achieve load balancing. Because of large base set size, each operation is executed where the base set resides, and intermediate results are transferred between physical nodes. In such systems, each dataflow operation is typically executed on many physical nodes. Furthermore, because computations are data-dependent, we cannot know until run time which subset of the physical nodes containing a particular base set will be involved in a given dataflow operation. This uncertainty creates several problems.

We examine the problems of efficient program loading, dataflow-operation activation and termination, control of data transfer among dataflow operations, and transaction commit and abort in a distributed data-intensive system. We show how these problems are interrelated, and we present a unified set of mechanisms for efficiently solving them. For some of the problems, we present several solutions and compare them quantitatively.

1 Introduction

This paper examines the problems of efficient program loading, dataflow-operation activation and termination, control of data transfer among dataflow operations, and transaction commit and abort in a distributed data-intensive system. By "data-intensive" we mean any system in which the large size of the base data causes significant storage and performance problems. In this section, we first describe our assumptions about the hardware organization and data distribution in the system, present our model of computation, and define some terms. We then define the problems in more detail.

1.1 Hardware Organization

We assume a highly parallel machine for data-intensive applications. It is designed to include on the order of 1,000 intelligent repositories (IRs). Each IR has a main microprocessor, a disk controller, a communication processor, a large main memory, and a disk. Its design philosophy is shared-nothing [Sto86]; neither memory nor disks are shared between IRs. Bubba also has several **interface processors** (**IPs**) to handle interaction with users and some centralized functions. The IRs and IPs are connected by an interconnect, so that any IR or IP may send messages to any other IR or IP. This interconnect is the only shared resource; it makes a network out of the set of IRs and IPs. This network is a single machine; the IRs and IPs are physically close and message delays are small. The high-level hardware organization is illustrated in Figure 1.1. A similar hardware organization is used in [Ter85], [DeW86] and [Tan87].

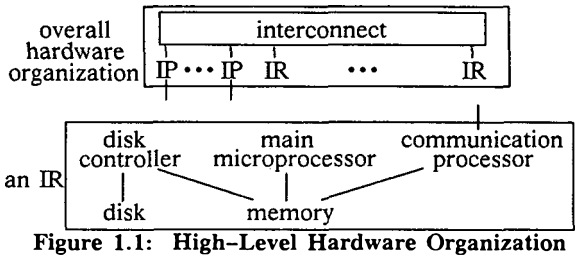

Figure 1.1: High-Level Hardware Organization

1.2 Base Set Data Distribution

Even though message delays are small, the shared-nothing assumption means that this architecture has some features in common with distributed systems. In particular, load balancing among IRs is critical to system performance. Our most important mechanism for balancing the load among IRs is *declustering*, or spreading each base set over multiple IRs [Liv87]. The set of IRs which base set A is declustered over is called the *home* of A. In general, IRs belong to many homes; that is, each holds portions of many base sets. To achieve load balancing, we intentionally try to destroy locality by distributing a base set over its home IRs based on a hash of its key value.

We assume that each base set is indexed, and each IR in A's home holds a portion of A indexed by a contiguous range of the hash of its key values. Each IR has an identical copy of a *global directory* telling which IRs hold

base set A, and within A's home which IRs hold which range of the hash of key values. Thus, every IR knows which IR holds a given item of a base set, associatively by key value, but not where it is within the IR (cached in RAM, on which disk block, etc.). Every IR maintains a distinct *local directory* for the portions of the various base sets that live there.

1.3 Computational Model

A *transaction program* is a dataflow program which implements a particular type of query and/or update of the base set. A transaction program typically has parameters which must be bound for execution. A *transaction* is a particular execution of a transaction program with these parameters specified. Each transaction has a unique identifier. We assume that transaction programs are compiled, and that the compiler, using conventional techniques based on knowledge of base set sizes and selectivities, has provided a partial ordering of the operations [Sel79]. This partial ordering determines the topology of a large-grained dataflow graph.

Figure 1.2 illustrates a transaction program. Each node of the dataflow graph, called a *component*, is a program which accesses at most one base set. A component may produce temporary results which are used as input to other components (e.g., Tw, Tx, Ty and Tz), which establish dataflow dependencies between components. A component may also produce results which are returned to the user (e.g., Ra and Rb). Finally, a component may update a base set (e.g., C3 updates base set C). When a component has a dataflow dependency on other components (e.g., C5 is dependent on C2 and C4), its start and completion is logically constrained by its predecessor components. Components which have no dataflow dependencies (e.g., C1 and C4) are called *start components*. A start component may start and finish independently of any other component.

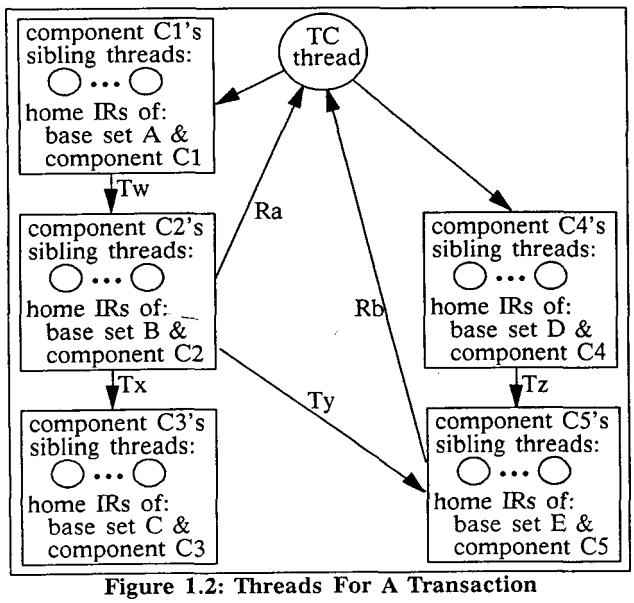

Figure 1.2: Threads For A Transaction

It is usually cheaper to move intermediate results over the interconnect between IRs than to move base sets, because base data is typically much larger and because it is not always needed in its entirety. Therefore, we do the work where the base set data resides. That is, the transaction execution will be distributed across the network, with the operation that accesses a given base set executing on IRs in the home of that base set. We refer to the home of the base set accessed by a component as the home of the component.

A transaction is implemented by assigning a *thread* (a single-threaded process) to every component within each IR and to a special thread called the *transaction coordinator (TC)*. Threads which execute copies of the same component in parallel on different IRs within the component's home on behalf of the same transaction are called *sibling threads*. Threads of start components are called *start threads* and are located on *start IRs*.

The TC acts as the interface between the user and system for a transaction. The TC is compiled with the rest of the transaction program and is the thread which is initiated by the system to begin a transaction in response to a user request. The TC knows which components/threads in its dataflow graph are start components/threads, and which base set each component accesses. Through the global directory, the TC can direct start-up messages containing parameter values to the start IRs, which comprise the home of these start components/threads. Sometimes, the values of query parameters are base set key values, in which case messages can be directed to exactly the relevant subset of IRs in the home. If the query parameters are not key values, or if the parameters are unbound, then threads must be started at all IRs in the home. Results to be returned to the user are sent to the TC.

Once a start component/thread has performed its operations on its base set and computed its intermediate results, the communication procedure is repeated. This component/thread in turn must send its intermediate result to some subset of the IRs in the home of the next component in the dataflow graph. If the intermediate data set contains values for the key of the base set in the receiving home, then a subset smaller than the whole home can be determined by using the global directory; otherwise, the result must be sent to all IRs in the home. We use the notation that for a given data transfer (a given arc on the dataflow graph), the number of IRs in the sending home is S, and the number of IRs which actually have data to send is s, $1 \leq s \leq S$. Similarly, the size of the receiving home is D, and d IRs will actually receive data, $1 \leq d \leq D$.

In general, the work of each component will need to be carried out at some subset of the IRs which comprise its home, but which IRs are in the subset cannot be determined until runtime because it is data-dependent. This uncertainty is at the heart of the problems considered in this paper.

1.4 The Problems

Chronologically, the first problem is when and how to load the transaction program which executes the query. Load images for each component can be built

immediately after compilation and stored on all the IRs in the home of the component. Within the IR, they can reside in RAM or disk. Alternatively, loading can be done at each IR on demand after the first data message arrives at each IR. This problem is examined in Section 2.

The second problem is when to activate the threads which executes each component. The two choices are to preactivate a thread at each IR in the component's home before any data is sent to it, or to delay activation until we know a component really needs to be executed on a particular IR. This problem is examined in Section 3.

The third problem is assuring that a given copy of a component knows when it has received data from all the threads of sending components from which it is ever going to hear. Once a thread has received the first packet of data from another thread, the communication protocol tells it when *that* thread has no more data to send, but because thread execution is asynchronous and because the number and identity of senders is only determined at run time, it cannot know *a priori* that there is not still some sender who has not yet sent its first packet. We call this the *dataflow control* problem.

There are many correct solutions to the dataflow control problem. We could, for example, require that component threads always be started at every IR of the component's home, and further require that every thread of a sending component send messages to every thread of the receiving component, even if only a null message [Kho87]. This solution is too inefficient for transactions involving a subset of the home IRs (i.e., s<S or d<D). In Section 4, we describe three solutions which have more reasonable performance. In Section 5, we compare the three solutions quantitatively according to certain performance metrics.

In Section 6, we look at the related problems of transaction commit and abort in the same distributed environment. We show how the mechanisms we propose to solve the dataflow control problem can be extended to help with these as well.

2 Program Loading

Transaction execution requires the component program to be loaded into the IRs which will need to run them. In general, it is not known until run time which IRs will actually need to execute a particular component. In most cases, prior to run time, we only know which home set of IRs might need to execute a particular component. In this section, we describe different techniques for loading components into the appropriate IRs, and describe qualitatively under what conditions each is better. In [Ale87], we use an analysis similar to the 5-minute rule [Gra87] to quantify this comparison.

2.1 Preloading

Preloading is a technique which loads the components of a transaction program into all IRs of the component's home. This can be done prior to run time, because each component's home is known prior to run time.

Qualitatively, preloading is efficient for the following types of applications:

A) A single transaction which involves nearly all of the IRs in a home. This would typically be caused by poor selectivity (a large fraction of a base set's values are selected), which can be estimated at compile time.

B) A transaction program which has frequent executions. Even if only a few IRs in each home are involved in each execution, frequent executions will cause practically all of the home's IRs to be active within a short time period.

Preloaded programs can be stored either in memory or on disk. In case A, it is more efficient to store preloaded programs in memory than on disk, because they will be used immediately after loading. In case B, whether memory or disk storage is more efficient depends on the frequency of executions of that transaction and the program size.

2.2 Dynamic Loading

Dynamic loading is a technique which loads the components of a transaction program into an individual IR upon request by that IR if and when it is needed at run time. When an IR receives the first data message for a component thread at run time, there will be no thread for the message. When such an orphan message is detected, the IR knows that the program will be needed and requests the TC to load the program at that time. This request requires the IR to send an additional message to the TC of that transaction.

Dynamic loading is similar to dynamic activation described in Section 3. Because of their similarity, dynamic loading and dynamic activation can share the same mechanism for detecting orphan messages and using them to decide when to load and activate a program.

Qualitatively, dynamic loading is efficient for a single transaction which involves only a few of the IRs in a home.

3 Thread Activation

Our model of computation includes wavefronts of partial results advancing from one base set home to the next, where further computation is done. The arrival of data or parameters at one IR of a home logically implies the initiation of a thread to do (a subset of) the next component of the transaction. As a practical matter of system thread management, this initiation may precede or follow the data arrival.

Thread activation costs in at least two dimensions. It requires system computational and storage resources. It also requires real time, during which this thread of this component of the transaction does not proceed. Thus, the choice involves a throughput vs. response time tradeoff; it is similar to choosing between lazy and speculative parallel execution models for programming languages [Hud84] [Tig85].

3.1 Preactivation

One alternative is to activate threads of the component on all IRs which might execute it, which is all IRs in the component home. This could be done with a special thread start-up message sent to all the IRs in the home when the transaction is initiated.

Preactivation may reduce transaction response time by taking thread initiation out of the time-critical path; threads of all components are initiated in parallel at the beginning of the transaction. This is most significant for deep dataflow graphs. The disadvantage of preactivation is that it often causes the system to do the unnecessary work of activating, maintaining and terminating unnecessary threads. In general, the base set data in only a subset of the IRs in a home will actually be accessed by a component. This disadvantage is reduced when messages and all aspects of thread management are very cheap, when component execution needs to be synchronized anyway or when few IRs are involved in the transaction. In summary, preactivation is suitable when transaction response time is of paramount importance.

As we will see in the next section, in some cases it is possible that no one thread of a component knows how many threads there are or which IRs they are on; this is true of the complement of the useful set as well. The extra centralization (and serialization) required to know which threads are useful and which are not is no more than is required to solve the dataflow problem. Termination of the useless threads still requires extra messages and system overhead.

It makes no sense to use preactivation with dynamic loading because preactivation requires the presence of the transaction program.

3.2 Dynamic Activation

To reduce system overhead, we can initiate only those threads which are actually needed by each component of a transaction. We can accomplish this by activating a thread only when the first data message addressed to it arrives at an IR.

Dynamic activation requires that the message handler be able to cope with messages addressed to threads which do not yet exist. This requires that the component identifier contained in the data message must sufficiently identify the thread to enable the system to find or create the proper load image. In either case, the message handler will presumably turn the relevant information over to the thread manager, who will do the actual thread activation.

The advantage of dynamic activation is reduced system overhead because only useful threads are activated. The disadvantages of dynamic activation are that more sophistication is required of system components such as the message handler and thread manager, and that there is a real time delay between when each thread is logically enabled and when it actually begins to process its incoming data. The response time disadvantage is considerably reduced if preloading is used or if the dataflow graph is shallow.

Dynamic activation is compatible with either preloading or dynamic loading.

4 Three Techniques For Dataflow Control

In this section, we describe three techniques for dataflow control. All of the techniques are functionally complete; any of them could be used exclusively if performance were not important.

We describe the techniques as they would be used with dynamic activation. Any of them can also be used with preactivation. The main difference is that with preactivation, someone must kill the unneeded threads. We show in Section 6 how this can be done by the "responsible thread".

One problem which all dataflow control techniques must solve is knowing how many sending threads there are at each component. This is necessary so that receiving threads know how many different threads to expect messages from so that they know when they have received all the data they are going to get. For code robustness and for commit and abort efficiency (see Section 6), it is desirable to convey to the receiving threads the identities of the sending threads {senders}, instead of just the number of them.

We assume that the number of senders, s, is equal to the number of threads which received data in the previous component. In the case where a thread receives data but produces none for the next operation, we will require that it send control message(s) stating that fact. Thus, the problem of knowing {senders} is equivalent to the problem of determining the set of threads which receive messages {receivers} at each component; {receivers} for one component becomes {senders} at the next component.

Because the senders are determined at runtime, the only feasible solution is for one or more threads at each component to know {senders} and communicate it to the receiving threads. This in turn requires that one or more threads know {receivers} at each component. The requirement of knowing {receivers} at each component is an important consideration in designing dataflow control techniques. We adopt the rule that the identities of the receivers, {receivers}, are communicated to each of the d receiving threads in all techniques rather than just the number d.

All of these techniques can be used within the same transaction. We are free to use the technique which is most effective between each pair of communicating components provided only that {senders} and {receivers} are always propagated correctly.

4.1 Control Node

The first dataflow control technique we will describe is called *control node* (*CN*). In this technique, one IR is designated the control node and given the extra job of learning and communicating {receivers}, and also of telling the receivers from whom they should have received data.

After each sender has sent its last data message, it sends a special control message to the control node containing the identities of all the receivers to which it sent data messages. The control node knows to wait for s "here is who I sent data to" messages because the senders tell him the value of s. The control node determines {receivers} from the union of these identities and then sends one control message to each of the d receivers telling them that there are no more data messages coming, from whom they should have received data, and their sibling set. The CN technique is illustrated in Figure 4.1, where the shaded circles represent IRs containing

activated threads, the solid lines represent pure data messages, and the broken lines represent control messages.

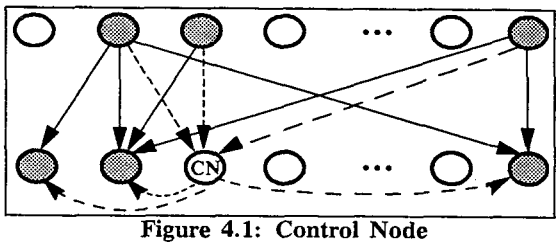

Figure 4.1: Control Node

This technique requires up to s+d extra control messages in addition to the data messages. If the control node is one of the receivers, and if the control message from each sender is piggybacked on the last data packet to the control node, then there could be as few as d-1 extra control messages.

The senders must agree among themselves upon the identity of the control node, preferably without using any messages. One method would be to use a hashing function on the transaction identifier. If the IR chosen to be the control node did not happen to be one which receives any data, then the control messages to it cannot piggyback. In the comparative analysis in Section 5, we assume s+d short extra control messages for the CN technique.

The main advantage to the CN dataflow control technique is its simplicity. As we shall see, in most situations some other technique can do the job with fewer message packets.

4.2 Point To Point

In the *point to point* (*PP*) technique, there are no pure control messages; all control information is piggybacked on data messages. Every sender includes {senders} with one of the packets which it sends to each receiver, and sends its own end-of-data signal piggybacked on the last data message it sends each receiver. This guarantees that each receiver knows from whom to await messages. Each sender must send at least one message to each receiver, even if it has no data for that receiver. PP is illustrated in Figure 4.2, where solid lines indicate data messages and broken lines indicate null messages.

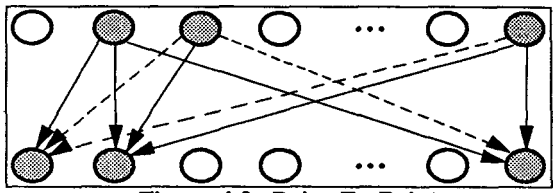

Figure 4.2: Point To Point

The problem with this technique is that {receivers} cannot always be known. In special cases d may be known at compile time because of the nature of the operation the receivers are performing. In other cases, if the number of data values being sent is very large, then d may safely be assumed to be equal to D, the size of the home of the receivers. If s=1, then d is always known. But if s>1, it is possible that no one of the sending IRs knows {receivers}. If PP were the only dataflow technique available, we would have all senders send messages, possibly empty except for the end-of-data signal, to all IRs in the receivers' home, and set d=D; IRs in the receivers' home which got all empty messages would still have to participate in the next component, sending empty end-of-data messages. We consider the cost of these empty messages and useless threads to be unacceptable, and will not use PP in such cases.

In the cases where we will use it, PP will never send more message packets than CN. If either d=1 or s=1, then with suitable choice of control node, CN can be made equivalent to PP.

4.3 Mux/Demux

Whereas CN sends some control messages which contain no data, and PP piggybacks all control information on data, *Mux/Demux* (*MD*) can be thought of as piggybacking data on control.

In this technique, one IR is chosen as a special control node as in CN. However, in MD, the senders send all their data to this control node. The control node re-bundles the data into d messages, in the process discovering {receivers}, and forwards it to the receivers. The control node knows from whom to await messages because the senders tell it {senders}. Receivers know to expect only one big message. MD is illustrated in Figure 4.3.

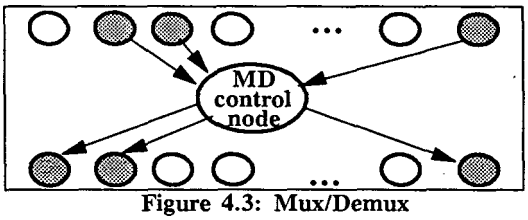

Figure 4.3: Mux/Demux

With MD, there are always s+d messages. The control node can be chosen from among the senders' or receivers' home using the same method as with CN. In the comparative analysis in Section 5, we will not assume that the control node is one of the actual senders or receivers; if it is, there are only s+d-1 messages.

Because MD bundles data up into bigger messages, it will result in fewer packets being sent than with the other two techniques when there are relatively few data values to be sent by each sender. The big disadvantages of MD are that all data is sent twice, imposing both a bandwidth and a response time penalty, and that the control node may become a hot spot. Thus, MD is suitable when there are relatively few total data values to send. When s=1 or d=1 and with suitable choice of control node, MD becomes equivalent to the other two techniques.

4.4 Components With Multiple Receivers

Components which may receive data from more than one other component require special treatment to insure correctness when dynamic activation is used. Consider as an example component C5 in Figure 1.3; it will receive data from both C2 and C4. If the compiler knows that C2 and C4 will send messages to exactly the same set of IRs in C5's home, then there is no problem. The compiler might know this because of constant propagation or

because the operation performed by C5 must be done on all IRs in the home. In general, however, C2 and C4 may send data to different subsets of IRs.

Suppose for example that base set E is declustered over IRs 1 through 4, that collectively the sibling threads of C2 send data to IRs 1 and 2, and that the sibling threads of C4 send data to IRs 2 and 3. Sibling threads of C5 get dynamically activated on IRs 1, 2 and 3. The problem is that the thread on IR 1 has no way of knowing that no message from C4 is ever going to come, and the thread on IR 3 does not know not to wait for a message from C2.

Any way of telling the waiting components to wait no longer is logically equivalent to sending them a null message on the port they are waiting on. We will sketch two ways to generate such null messages; both require the compiler to recognize that a component has more than one incoming dataflow arc.

The first solution requires the compiler to always choose the same dataflow control technique on all the arcs into a given component. If the technique chosen is PP, then the required null messages will be generated automatically, and nothing more need be done. If MD or CN is chosen, then all control messages must be addressed to a single control thread serving all the arcs rather than generating separate control threads for each data transfer. This single control thread must form the union of all sibling threads activated by all the transfers when informing the receiving threads of their sibling set. Furthermore, for each incoming arc, the control thread must do a set difference and send a null message to each sibling thread activated which received no data over that arc.

Another solution has the advantage of allowing different dataflow control methods to be used to send data to the same component, but requires the creation of a special thread to coordinate the receivers. All the transfers proceed almost normally, but all control information must be directed to the special thread. If a transfer uses PP, at least one of the senders must send a list of IRs to which data was sent by that transfer to the special thread. The special thread forms the union of all receivers and informs them of their sibling set. It also generates the required null messages based on set differences. One possibility would be for the TC to act as the special thread in these situations.

5 Choosing a Control Technique

There are many possible criteria for choosing among MD, CN and PP. All three techniques are logically correct and reasonably easy to implement, but no one of them is superior in all situations by all measures of performance. This section compares the three techniques using two metrics. In Section 5.1, we compare the three techniques based on the number of required packets. In Section 5.2, we compare the three techniques based on system throughput and transaction response time. For a more detailed comparison of the three techniques, see [Ale88].

In the previous section, we discussed in general terms the number of messages required by each of the three dataflow control methods. However, number of messages is not a good metric for communication work, because the messages generated by the three methods vary greatly in length, and because messages will be broken up into packets for transmission. Protocols, routing, forwarding, and other costs of sending and receiving information between IRs will mostly be done on a per packet basis. Therefore, the total number of packets sent is a better metric of communication work. In this section, we will assume that data messages are always broken into packets, that each packet can hold up to P data values, and that control information is of negligible size compared to data.

5.1 Choosing a Technique Based on Number of Packets

In [Ale88], we used two methods of estimating how many packets of size P would be required to transmit N data values between s senders and d receivers using a given dataflow control technique: simulation and simple approximate formulas. We showed that the two methods of estimating packets agree closely enough that in subsequent analysis we can use the following simple formulas with confidence.

With MD, a first approximation might be that sending N values to the control node will result in N/P packets. This is optimistic because some senders will send more values than others and the final packet from most of them will be only partially filled. If every sender's last packet is half filled, there will be s/2 additional packets for a total of N/P+s/2. Similarly for the second phase, when the control node forwards all the values to the receivers, there will be approximately N/P+d/2 packets. However, when there is only one sender or one receiver, one of the two phases need not be done. Thus, the total number of packets with MD is approximately

$$NP_{MD} = \min(s,d,2) * \frac{N}{P} + \frac{s+d}{2} . \qquad (5\text{-}1)$$

When using the CN technique, there will be up to s*d "final" packets. Thus, we estimate that there are approximately N/P+(s*d)/2 data packets. If neither s nor d is 1, there will be an additional s+d control packets:

$$NP_{CN} = \frac{N}{P} + \frac{s*d}{2} \quad \{+s+d \text{ if}(s>1 \text{ and } d>1)\}. \qquad (5\text{-}2)$$

The simple estimate of the number of packets required to send N values using PP is similar to Formula 5-2 for CN with two differences. PP never has any extra control messages, but it always sends at least s*d packets:

$$NP_{PP} = \max\left[\frac{N}{P} + \frac{s*d}{2}, \quad s*d\right] . \qquad (5\text{-}3)$$

There is an important assumption implicit in Formula 5-3. We *assume* that for large N, if d is not 1 then d=D. In other words, if a large number of data values are being sent, in most typical applications they are going either to exactly one IR or to all the IRs in the receivers' home. If a large number of data values are sent to 1<d<D receivers, then Formula 5-3 may seriously underestimate the number of packets required by PP. We could substitute D for d throughout; we choose not to do this to make comparison with the other two techniques easier, and because we think the assumption is reasonable. The assumption is less reasonable for small N, but it is

unreasonable to use PP at all in the case where N is small and neither s nor d is 1.

Figure 5.1 shows the simple formulas for all three techniques for one choice of values for s, d, and P. The results illustrate our earlier assertions; for small N the number of packets sent over the interconnect can be minimized by using MD, while for large N using PP minimizes the number of packets. Recall that the formulas for MD and CN assume that the control node is not one of the senders or receivers unless s=1 or d=1. If we can manage to make the control node one of the receivers and use piggybacking, then the performance of MD and CN is slightly better relative to PP than is shown in Figure 5.1.

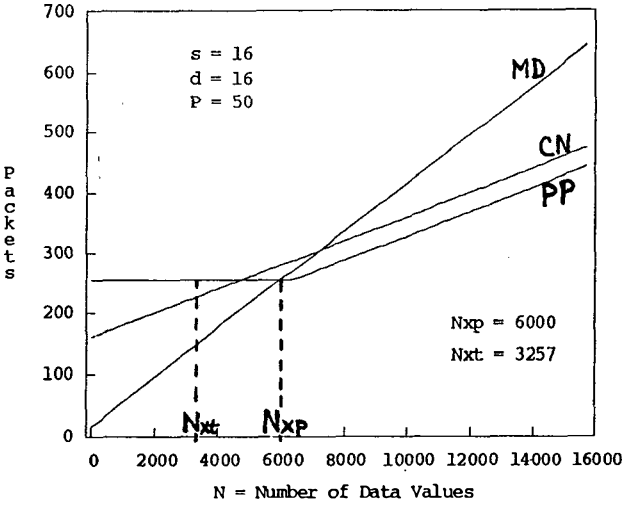

Figure 5.1: Comparing Number of Packets

We are interested in N_{xp}, the crossover point between NP_{PP} and NP_{MD}. It is shown in [Ale87] that N_{xp} always occurs when the s*d term of Formula 5-3 applies, so that for s>1 and d>1

$$N_{xp} = \frac{s*d*P}{2} - \frac{(s+d)*P}{4} \quad (5\text{-}4)$$

An estimate of N at each component in query execution is usually generated and used by a query optimizer. A compiler could use the estimate as a basis for choosing which dataflow technique to encode for each communicating pair of components. The following algorithm will minimize the number of packets sent:

```
if(s=1 or d=1) use PP
else if(N < Nxp) use MD
     else if(d is known) use PP
          else use CN
```

Algorithm 5.0

At the third line of the algorithm, we want to avoid using PP when {receivers} is unknown because we want to avoid having to initiate and communicate with all IRs in the receivers' home if not all of them would really receive data. {receivers} might be known if, for example, the arguments to a component were constants. More generally, if the application justifies the assumption that, for large N, d is nearly always equal to D, then we can always choose PP at this point.

Algorithm 5.0 assumes that we are willing to support three different dataflow control techniques. If we were going to choose only one technique, it would be CN, because while it seldom generates the fewest packets, it is also seldom far from optimal. If only two techniques could be supported, we would choose MD plus either PP or CN.

5.2 Choosing a Technique Based on Throughput and Response Time

Choosing a dataflow control technique based on number of packets unrealistically favors MD because it does not recognize that, for N near N_{xp}, MD may cause a severe load imbalance; the control node must either receive or send every packet and must repartition data in packets. This will lengthen response time and may also reduce throughput.

Whether we should actually use MD or PP (or CN) in a given transfer for better system throughput depends on several factors including T, the average number of data transfers going on simultaneously in the system. If T is large *and* if we can somehow insure that the role of control node for different transactions is spread as evenly as possible among all the IRs, then using MD might result in higher throughput because total load is lower. Notice further that there is probably a loose inverse relationship between T and $\overline{N}$, the average number of data values transferred per dataflow arc. Assuming a system of fixed total capacity, this capacity is an upper bound on $T*\overline{N}$.

Choosing between MD and PP also involves conflicting influences on the response time for a data transfer. On one hand, using MD can increase response time because the single control node has to process every packet, imposing a serialization. On the other hand, if the control node role is evenly distributed and if using MD has resulted in a significantly lower total load, then there will be less queuing within the MD nodes, which decreases response time.

The advantage of MD over PP in reduced total load decreases as N increases toward N_{xp}. Also, the load imbalance caused by MD gets worse as N approaches N_{xp}. Therefore, we seek a point $N_{xt}<N_{xp}$ at which throughput and response time for MD compare favorably with PP. In [Ale87], we derive an approximation for N_{xt} for any given point in the throughput/response time space. This N_{xt} can be used in Algorithm 5.0 instead of N_{xp}. Figure 5.1 illustrates the value of N_{xt} assuming that response time is the same for both MD and PP.

Another consideration is pipelining. Our computational model allows pipelining among dataflow operations. However, in our performance comparisons, we have assumed that such pipelining is limited to the extent that it does not require additional packets. That is, when data is being sent to a particular IR, a packet is not sent until it is either full or the sending IR has no more data to send. This restriction causes pipelining using MD to begin at about N = max(s, d)*P and causes pipelining using CN or PP to begin at about N = s*d*P. Thus, MD allows pipelining to begin at lower values of N.

6 Transaction Commit And Abort

In this section, we outline techniques for committing and aborting a transaction in a distributed shared-nothing system which uses the activation and dataflow control techniques we have described.

Regardless of the concurrency control (CC) method used, at commit time it is necessary for a single commit coordination thread to conduct a dialogue with all active threads of the transaction that have accessed base data, culminating with a message to each of them (or at least the writers) to commit or abort the transaction. Any thread that has not accessed base data can be terminated immediately upon completion of that thread. This CC dialog requires that the coordinator know the identity of all threads of all transaction components which have accessed base data. The problem is made somewhat simpler by the fact that this knowledge is required only after all components have completed their work and are awaiting the commit dialogue.

A slightly more difficult problem arises if a transaction must be aborted before it has completed (e.g., a user directed abort or a deadlock). Again we need to send a message to all threads of the transaction that have accessed base data, but in this case we may not yet know which of the components have been started.

Recall that under preactivation, threads are initiated at every IR in a component's home whether they will do useful work on behalf of the transaction or not. CC or abort messages should be broadcast to every IR in every home accessed by the transaction. A thread either has been or will be initiated at each of these IRs, so messages will match up with threads, and there are no unnecessary messages or loose ends. But it is desirable to kill unneeded threads as soon as possible. (For example, they could be killed by the "responsible thread" described in Section 6.1.) If the unneeded threads have already been killed when the CC or abort messages need to be sent, then the situation is exactly the same as with dynamic activation.

With dynamic activation, we could still broadcast to all the IRs in the receivers' home. However, this wastes messages, and it also creates an awkward situation at IRs which do not and never will have a thread active for this component. We describe one solution to the problem of sending messages only to the active threads in a transaction. It is certainly not the only possible solution, but it is correct and reasonably efficient.

6.1 Transaction Coordinator and Responsible Threads

In Section 1.3, we defined a special TC thread to interface with the user and initiate the transaction. We propose that the TC also act as the commit coordinator, and be the thread to whom transaction abort messages are sent. All the roles we propose for the TC typically occur at different times in the life of a transaction, so little is lost by serializing them in one thread. Because the TC is compiled with the program, it knows the information represented in the transaction program's dataflow graph such as the one in Figure 1.2.

Recall from Section 4 that with every dataflow control technique, at least one thread among every pair of communicating components knows the identity of all the receivers for that data transfer. We call this thread the *responsible thread* (*RT*). With CN and MD, the control node is the RT. With PP, any of the IRs in, say, the receiving home can be chosen arbitrarily in the same way control nodes are chosen under CN or MD. If more than one data transfer is directed to the same component, there should be only one RT for the receiving component, not one for every transfer; this will be the same thread which formed the union of receivers for all the transfers (see Section 4.4). We cannot assure that the RT is one of the actual receivers, but this does not matter.

Notice that the RT is chosen algorithmically by, for example, hashing into the receiving home based on the transaction's identifier. The TC can perform exactly the same computation, and therefore knows the identity of each RT at any time. We propose that each RT tell the TC the set of active threads for its component, and that either the TC communicate with them directly (for transaction commit) or that the TC communicate with the RTs and that they communicate with their siblings (for transaction abort).

6.2 Transaction Commit

When the TC receives the results from the result threads, it must initiate a two-phase commit protocol with all the threads that accessed base data during the course of the transaction. Every RT will send one message to the TC as soon as the RT knows the complete set of its receivers. The TC knows which RTs from which to expect these messages based on its knowledge of the dataflow graph. After hearing from all of them and after getting the transaction results, the TC can begin the commit protocol with exactly the correct threads.

It may be that threads of more than one component of a transaction end up executing on the same IR because the homes of the relevant base sets overlap. In this case, we can reduce the number of messages required to implement transaction commit as well as reduce system overhead at each IR by grouping threads belonging to the same transaction for certain purposes. For example, if they share a common workspace, then that space need be allocated only once, and the TC can carry on a single dialogue with the group or an elected thread to commit their combined updates rather than with each thread individually. Clearly, the system thread manager and message handlers have to support such a grouping. The Mach operating system [Acc86] has such a grouping of threads into a "task".

6.3 Transaction Abort

There are many workable schemes for aborting an incomplete transaction. The simplest scheme is for the TC to wait for the result IR to send it the answer, and then send termination messages instead of transaction commit messages using the transaction commit protocol as outlined above. An alternative scheme is for the TC to send a special abort message to the transaction's start IR directing it to forward the termination directive to everyone to whom it sent data messages, and then die. Each thread receiving this special message forwards it to its receivers, thus assuring that all and only the threads

dynamically activated are killed. With these two schemes, the transaction may complete all its work before aborting.

We believe it will usually be more efficient to immediately send the abort message to each responsible thread in an attempt to prevent the transaction from proceeding with work which is going to be wasted. If the TC's abort message arrives before the RT has been activated, this message causes the activation (as always with preactivation). As soon as it knows the identities of its siblings, the RT kills them (under PP and CN) or never activates them (under MD). Under CN and MD, the transaction's progress down this particular path in the dataflow graph is halted; execution may proceed across arcs using PP until an arc using CN or MD is reached.

The responsible threads activated by the TC's abort message act as a "fire break", preventing the transaction from proceeding and thus preventing unnecessary work from being done. Some RTs "downstream" in the dataflow graph from where the transaction is eventually halted may never receive any data messages and never have any siblings to kill. These lonely RTs can be terminated by a dialogue with the TC, or they can simply be allowed to time out.

From a correctness point of view, it does not matter whether we are successful in stopping the transaction early or not; it cannot commit, because the TC is also the commit coordinator, and it knows not to commit. Thus, the transaction may safely be assumed aborted as soon as the TC is known to have received the abort request.

7 Summary

Our architecture and computational model imply that each dataflow operation of a program can be executed on potentially a very large number of physical nodes (IRs). Exactly which IRs is not known until run time. This uncertainty gives rise to the control and performance problems which were addressed in this paper.

The control problems include how to correctly load and activate threads, when a thread can safely stop receiving on a port, and how to correctly commit and abort transactions. The associated performance problems are how to do these things efficiently. We described an integrated set of mechanisms to correctly and efficiently handle these.

We described three dataflow-control techniques called MD, CN and PP. We described the conditions under which each technique provides better efficiency and response time. We also provided an algorithm for choosing among MD, CN and PP, which assumes that we are willing to support all three techniques. If we were going to choose only one technique, it would be CN, because while it seldom generates the fewest packets, it is also seldom far from optimal. If only two techniques could be supported, we would choose MD plus either PP or CN.

For all of these problems, an important measure of efficiency is the number of communication packets which must be sent. We prefer decentralization, and tried to achieve necessary control with the minimum message traffic without introducing any extra synchronization delays not already in the application. We observe that a single thread is necessary to serve as the user interface in any distributed computation; we called this thread the transaction coordinator (TC) and assigned it the other centralized roles necessary to achieve transaction initiation and transaction commit or abort. A common property of the three dataflow control techniques is the use of a single IR which coordinates the receiving activity for a component. We used these threads to help with committing and aborting transactions as well, offloading some of this work from the TC.

We are now implementing these mechanisms on a medium-sized shared-nothing machine. This will allow us to make actual performance measurements.

Acknowledgements

Thanks to Setrag Khoshafian and Marc Smith for their helpful ideas and comments.

References

[Acc86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation For UNIX Development," *Summer Usenix Conference* (July 1986).

[Ale87] W. Alexander and G. Copeland, "Process And Dataflow Control In Distributed Data-Intensive Systems," *MCC Technical Report No. ACA-ST-281-87* (September 1987).

[Ale88] W. Alexander and G. Copeland, "Comparison Of Dataflow Control Techniques In Distributed Data-Intensive Systems," *ACM SIGMETRICS Conference*, Santa Fe, New Mexico (May 1988).

[DeW86] D.J. DeWitt, R.H. Gerber, G. Graefe, M.H. Heytens, K.B Kumar and M. Muralikrishna, "GAMMA---A High Performance Dataflow Database Machine," VLDB *Conference*, Japan (August 1986).

[Gra87] J. Gray and F. Putzolu, "The 5 Minute Rule For Trading Memory For Disc Accesses And The 10 Byte Rule For Trading Memory For CPU Time," *ACM SIGMOD Conference*, San Francisco (May 1987).

[Hud84] P. Hudak and B. Goldberg, "Experiments In Diffused Combinator Reduction," *ACM Symposium on LISP and Functional Programming*, Austin, Texas (1984).

[Kho87] S. Khoshafian and P. Valduriez, "Parallel Execution Strategies for Declustered Databases", *International Workshop on Database Machines*, Japan (October 1987).

[Liv87] M. Livny, S. Khoshafian and H. Boral, "Multi-Disk Management Algorithms," *ACM SIGMETRICS Conference*, Alberta, Canada (May 1987).

[Sel79] P.G. Sellinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie and T.G. Price, "Access Path Selection In A Relational Database Management System," *ACM SIGMOD Conference*, Boston (May 1979).

[Sto86] M. Stonebreaker, "The Case For Shared Nothing," *Database Engineering*, Vol. 9, No. 1 (1986).

[Tan87] The Tandem Database Group, "NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL," *2nd International Workshop on High Performance Transaction Systems*, Asilomar, California (September 1987).

[Ter85] "DBC/1012 Data Base Computer System Manual, Release 1.3," *C10-0001-01, Teradata Corp.*, Los Angeles (February 1985).

[Tig85] S. Tighe, "A Study Of The Parallelism Inherent In Combinator Reduction," MCC Technical Report No. PP-140-85 (November 1985).