Reprint from the proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures SPAA93, July 1-4, 1993, Schloss Velen, Westfalia, Germany.

Supporting sets of arbitrary connections on iWarp through communication context switches

Anja Feldmann Thomas M. Stricker Thomas E. Warfel

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

In this paper we introduce the *ConSet* communication model for distributed memory parallel computers. The communication needs of an application program can be satisfied by some *arbitrary set of connections* which are partitioned into discrete phases. A *communication context switch* is used to select the active phase.

We present an implementation of the ConSet model on the iWarp and describe its performance characteristics, contrasting it to a *message passing* implementation on the same machine. Our implementation demonstrates how one existing parallel computer can function as a "reconfigurable network" without needing a new processor interconnect technology.

The ConSet model works best when communication patterns can be optimized at compile time. We examine the interactions of the target architecture with the algorithmic problems encountered designing a *communication compiler* to effectively partition, route, and schedule connections. We built a prototype communication compiler for our iWarp implementation, and are using it to generate iWarp code. Looking at basic communication patterns as well as patterns generated by an iterative finite element PDE solver, we compare ConSet's performance (using the compiler's schedules) to that of message passing. Our experiments suggest that ConSet communication offers a performance advantage over message passing in applications where the communication pattern is known at compile time.

1 Introduction

In connection-based communication a persistent channel is opened from one processor to another to satisfy an application's communication needs. A *connection* provides unidirectional communication between any two nodes and offers guaranteed bandwidth and latency. Each *active connection* requires physical network resources; thus, only a limited number of connections may be active at one time. Over an application's lifetime, more connections may be required than the physical machine can simultaneously support. The *ConSet* model therefore promotes the notion of *arbitrary sets of connections* which are split into *phases*. A *phase* consists of a subset of connections which can all be held active by the available hardware resources; exactly one phase is active at any time. Which particular phase is active is under application control; the application requests a *communication context switch* to swap between active phases. All connections belonging to the active phase are themselves active and hence usable by the application. For most applications, restricting the number of active connections does not inflict a large performance penalty. Our observations indicate that connection usage by applications shows a high degree of temporal locality; only a limited number of connections are actually in use at any one time. We refer to this as *communication locality*, similar in concept to locality of reference.

In the iWarp architecture ([BCC+88], [BCC+90]), *connections* between processors are established by chaining multiple *logical channels* (see Section 4.1.2) to form a pathway. Once a pathway is established, data can flow without interruption. Each iWarp node, referred to as a cell, can support 20 logical channels. Individual logical channels on one cell may be linked to logical channels on a neighbor so that arriving data is automatically forwarded; this mechanism forms pathways. iWarp cells can have multiple active connections; the exact number depends on the logical channel allocations. The total pattern of logical channel linkages across the array, combined with the state of the associated channel buffers, constitutes the *communication context*. A *communication context switch* is a global operation which efficiently switches the pattern of logical channel linkages over the whole array. When an application switches phases, precompiled channel linkage information is loaded into the communication agent from the computation agent at each cell, thereby causing a new set of connections to become active.

Snyder suggested a highly configurable parallel computer, the CHiP architecture, based on a similar concept back in 1982 [Sny82] and created Poker, a programming environment originally motivated by this architecture [Sny84, NSS+88]. The major distinctions are that his processors were embedded into a larger switch lattice, and his connections are directly established by these switches rather than with logical channels. CHiP provides no sharing of the physical links among active connections. iWarp's logical channels enable us to support a greater number of active connections and hence construct more complex networks per phase.

We contrast implementations of the ConSet and message passing models on the iWarp, which provides reasonable hardware support for both models. *Message passing* is an alternative programming

*Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972–90–C–0035 issued by DARPA/CMO to Carnegie Mellon University.

model where information which needs to be communicated between arbitrary processors gets wrapped, addressed and sent out to the communication network, which in turn is responsible for routing and delivering the messages. The iWarp communication agent supports message passing with short-lived pathways and wormhole routing.

ConSet was motivated by a common engineering application, namely, a finite element problem solver. The finite element method (e.g. [Joh87]), approximates solutions to partial differential equations by solving algebraic systems with large, irregular, sparse coefficient matrices. These irregularities make load balancing difficult, and achieving maximum performance on distributed memory parallel computers requires precise knowledge of the target architecture together with careful partitioning of the problem. Fortunately, domain-based problems such as this have an underlying combinatorial graph with several useful properties derived from the geometric structure, among them the fact that all communication is known in advance [SBF+92]. This information can be used by a *communication compiler* to order, route, and schedule the necessary communication, in effect providing a globally optimal communication scheme. The communication compiler automatically generates the communication code for the parallel program. It partitions the set of arbitrary connections into distinct phases, then routes and schedules the communication within each phase. In contrast to the work of Bianchini and Shen [BS87], the ConSet model schedules connections rather than packet traffic. The approach of Smitley and Lee [SL89], to embed the set of connections onto any r-regular network, is of little use to us since embedding this network on iWarp would be as hard as the original problem.

Interest in reconfigurable networks is not new, and a complete overview is beyond the scope of this paper. (For more complete information see [LS91]). The simplest ones allow a program to specify the desired network type (within some hardware restrictions) at load time (e.g. [FLM+92]). Most complex SIMD proposals can reconfigure their network topologies at every instruction; most notable is the polymorphic-torus [LM89]. While this is a nice theoretical model [BAPRS91], a bus-based architecture is inherently non-scalable with conventional technology. We show how an existing parallel computer, the iWarp, can function today as a reconfigurable network to support phased connection sets.

2 Sets of arbitrary connections (ConSet) as a communication model

Parallel applications are usually constructed from a sequence of discrete algorithms. Each of these building blocks may use different communication networks, or more precisely, different *sets of arbitrary connections*. Neighborhood operations tend to use meshes, convolutions use butterflies, and permutations need a fully-connected graph. Some connection arrangements may exceed the available hardware resources and are therefore split into more than one phase. A *phase* consists of a subset of connections that can be held active simultaneously. Usually, there is a communication schedule for each communication phase, generated by automatic tools. Figure 1 illustrates the terms of our model graphically.

The concept of *phases* arises naturally. A parallel application itself is composed of smaller, self-contained building blocks with distinct communication needs. The sets of arbitrary connections within one building block must be further decomposed to fit the communication resources provided by the hardware. Our model relies on the following principles.

Principle 2.1 *A connection c between two processors has a well specified minimum bandwidth b_c and maximum latency l_c. Connections are used on a per element basis, where each element can be a single number.*

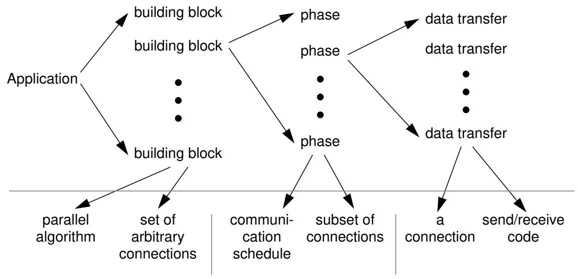

Figure 1: Overview over the model of phased-connections

Primitives for using connections:

```
send_element(connection_ID,element);
element = receive_element(connection_ID);
```

To achieve predictable bandwidth and latency, connections must have dedicated hardware to support efficient data forwarding, fair allocation of bandwidth among multiple connections, and proper flow control.

Principle 2.2 *The hardware can only support a limited number of connections at one time. Communication patterns which exceed this limit must be partitioned into phases.*

Primitive for switching phases:

```
instantiate_phase(phase_name);
```

Principle 2.3 *Global information and synchronization are used to switch between subsets of active connections as a whole rather than on a per-connection basis. Communication patterns are pre-evaluated at compile time and carefully scheduled by a communication compiler.*

In many parallel algorithms the communication occurs on an element-by-element basis. Unlike message passing, ConSet does not force applications to block data into large chunks. Our model is much less constrained in the connections permitted than traditional systolic computing. Non-physical neighbors can communicate arbitrary amounts of data with predictable bandwidth and latency; they just cannot talk to everybody at once.

The performance of ConSet can be characterized by the following parameters.

Cost of a communication context switch: The algorithm is charged the necessary synchronization costs t_{sync} which is related to the size of the machine, plus the cost of initializing the new phase t_{phase}.

Cost of send and receive: The algorithm is charged time $t = bk$ to send or receive an item of length k, where $1/b$ is the bandwidth of the network access.

Connection latency: The latency l is the network travel time for a data item traveling between two furthest-apart processors.

2.1 Communication compiler for the connection-based communication model

To support phased connection sets we provide a *communication compiler* that automatically generates the communication code for a parallel program. The compiler's challenge is to efficiently use the global knowledge of the application's communication patterns to minimize communication time. See Section 4.4 for an outline of the algorithms.

3 Message passing as an alternative communication model

An alternative model for communication between arbitrary processing nodes of a parallel computer is *message passing*. Parallel threads communicate shared data elements through explicit messages. Implementations of the message passing model are the default communication mechanism in most current distributed memory parallel computers [Nug88], [Lil90], [Fel88], [LAD$^+$92].

3.1 A simple model of message passing

Our simple model for message passing relies on the subsequent principles and defines the following primitives:

Principle 3.1 *Each thread of parallel computation identifies the data elements to be sent to another thread, composes them (together with size and address information) into a message, and invokes a mechanism to send the message out to the network. The mechanism may or may not block the computation until the network is able to accept further messages.*

Primitive for sending a message:

```
msg_send(destination,data_block);
```

Principle 3.2 *A receiving thread explicitly invokes a mechanism to receive any message addressed to it. The mechanism may or may not block the computation until a message is available from the network.*

Primitive for receiving a message:

```
msg_receive(data_block);
```

Principle 3.3 *If two threads run on different processors, the content of a message is moved over the network to the destination node; the structure of the network and the route taken by the message are not exposed to the programmer.*

Message passing systems have similar appearances across many distributed systems. Assuming a message length of k words (or data elements), their performance can be characterized by the following parameters.

Cost of send The algorithm is charged $t_s = a_s k + c_s$ for a send operation, where c_s is a constant overhead per message, and $a_s k$ the transfer time for sending a message of length k. The transfer time depends upon the length k of the message length and the bandwidth of the network access $1/a_s$.

Cost of a receive The algorithm is charged $t_r = a_r k + c_r$ for a receive operation, where c_r is the constant overhead per message, and $a_r k$ is the transfer time for receiving a message of length k. Transfer time depends on both the message length k and the network access bandwidth $1/a_r$.

Network latency The network latency l is the greatest time necessary for the head of a message to travel between two arbitrary nodes in a *congestion free* (i.e. unloaded) network.

Network delay The congestion in the network leads to another parameter, the potential delay of a message, due to wormhole routing. This delay parameter must be incorporated to compute the average usable bandwidth between any send-receive pair of processors, but how to accomplish this is still an open research issue.

With the exception of the network delay, good estimates (or even tight bounds) for all parameters can be derived from the hardware specification and the implementation details of a message passing system. Models with similar parameters have been postulated recently by [Ka93], [Val90]. Attempts to model the congestion and network load succeeded in the case of randomized, short messages. Although a problem definition and first approach is presented in [UFR92], little is truly known about the congestion which results from sending long messages on networks using wormhole routing.

For many basic algorithms, the combination of latency, network delay, and software overhead are the relevant parameters for characterizing communication performance visible to the application. Some blocked algorithms can amortize overheads and successfully hide latency by pipelining multiple messages or by overlapping communication and computation.

3.2 Contrasting the two communication models

While the connection based model and the message passing model both offer arbitrary connectivity to the application, the two models can be distinguished by the following three characteristics:

Off-line vs. on-line routing decisions: In the ConSet model the routing decisions are made off-line, based on precise knowledge of communication resources available and with a global view of the congestion. In contrast, a message passing system's routing decisions are quickly made *on-line* at every node with only a local view of network congestion. The routes cannot be optimized as there is no global knowledge of the communication pattern, and deadlock considerations restrict the available routing choices.

Resolution of congestion in the network: With ConSet, congestion is managed by three mechanisms: by dividing conflicting connections into separate phases, by routing connections so that they avoid conflicts, and (on iWarp) by the fine time-division sharing of the physical network links (logical channels). Message passing, on the other hand, resolves the network congestion by blocking and queuing messages rather than sharing link bandwidth. The delay for long messages in heavily congested networks is therefore not predictable.

Element vs. blocked communication: Phases allow us to amortize the cost of the connection setup over multiple messages. In message passing, the data must be blocked to pay for the significant overhead required to send and receive messages.

Note that in one aspect the message passing model is more flexible than phased connection sets. Message passing can deal with data dependent routing patterns directly without incurring any extra overhead, while the ConSet model can only embed such a pattern into an expensive all-to-all communication. Our ConSet model depends on communication locality for efficiency, namely, the reuse of phases within an application and/or connections within one phase.

4 Implementing the communication models

The *ConSet* model is strongly motivated by several architectural features of the iWarp component, so we briefly present the features relevant to our model, then explain how our model was implemented with the communication context switch. We examine how these architectural features can also support message passing, and give an overview of the off-line scheduler used to partition the connection sets into phases.

4.1 The underlying communication architecture

The architecture of the iWarp system has been described in detail in [BCC$^+$88] and [BCC$^+$90].

4.1.1 Processor overview

The iWarp system is based on a single chip VLSI processor developed jointly by Carnegie Mellon and Intel Corporation. The iWarp component contains both a 20 MIPS, 20 MFLOPS *computation agent* as well as a *communication agent* for transferring data to and from other iWarp processors. An iWarp *cell* consists of an iWarp component, fast static memory, and support circuitry. Physical links connect the cells in a 2-dimensional torus mesh configuration such that each communication agent has four incoming and four outgoing physical links with a maximum transfer rate of 40MBytes/sec each. The close coupling between the communication and computation agents allows the control logic for the communication agent to be directly mapped into the control register space of the computation agent. The processor can determine the status of the communication agent by simply reading a register, or can send data to the network merely by writing a register.

4.1.2 Logical channels and iWarp pathways

The single concept most relevant to the ConSet model is the *logical channel* [BCC$^+$88] [Gro89]. We establish connections in the iWarp architecture by chaining logical channels from cell to cell creating a *pathway*. Logical channels are a form of demand-driven, time-division multiplexing of the physical links combined with channel-specific data buffers in each communication agent. A total of 20 logical channels may be routed through any given cell; their data may be carried over any of the 4 physical links. The bandwidth of a physical link is evenly shared among those logical channels waiting to send by means of a fast hardware round-robin scheduler in the communication agent.

The communication agent can be configured to automatically forward all data arriving in a particular logical channel on to a different logical channel in a neighboring cell; in this manner pathways are built. Routing and flow control are handled for each logical channel by hardware at each cell. At the destination end of a pathway the computation agent retrieves the values from the logical channel buffer either by reading a special register which serves as a *gate* between the buffer and computation agent, or by means of a background direct memory access (DMA) operation known as a *spool*.

4.1.3 Reconfiguring the logical channels

The reassignment of logical channels requires both software intervention and mutual assertions/assurances between neighbors. It is essential that two non-adjacent cells do not attempt to create a new pathway through a common neighbor using the same logical channel on that neighbor. iWarp's standard mechanism of setting up and tearing down pathways is by sending control words that are "tagged" with extra state bits. These tagged words control the state in the communication agent as they arrive or pass through, without the involvement of the computation agent. This mechanism works on a per-channel basis. If more logical channels are needed than are available in a particular direction, pathway construction blocks until a needed logical channel becomes available.

4.2 Implementation of ConSet

While the normal method of switching logical channel assignments is by sending special control words, it is also possible for a computation agent to directly assign a state to its communication agent by writing to the control registers. We define the *full communication context* to be the set of logical channel assignments, including all forwarding and buffering information, required to establish a set of connections across the array. While there are many bits of state associated with the communication agent when it is forwarding data, a quiescent (i.e. no data in the buffers) node has a fairly small amount of significant state. This small amount of state can be quickly changed via direct intervention from the computation agent. A *communication context switch* consists of letting all logical channel buffers drain, then changing the state on all communication agents at once. Our *phases* in the connection model are implemented as a series of communication contexts, and our applications switch from one phase to another by performing a communication context switch.

Synchronization is an obvious concern. A particular cell may be finished with its communication needs for a given phase, but some of its logical channels may still be supporting connections used by other cells. It would be disastrous for it to change its logical channel forwarding assignments before the other cells were done with their communications. Thus, some means of synchronization is necessary so that all cells know when all others are finished communicating and it is safe to reconfigure the array. The sequence of events is shown below.

algorithm COMMUNICATION CONTEXT SWITCH
1. disable global events to prevent the runtime system (RTS) from interfering with the switch;
2. synchronize all processors;
3. put the communication agents to sleep, freezing any RTS data that may be in the system;
4. write a new logical channel forwarding pattern into the communication agent;
5. wake up the communication agent, allowing the RTS to continue;
6. enable global events;
7. resume program in a new context;

4.2.1 ConSet performance

ConSet performance can be analyzed using the parameters of Section 2. The following table captures most of them for a k word message on an N processor machine. All times are specified in clocks; on a 20 MHz iWarp each clock is 50ns.

It's worth noting that roughly 60% of the time required for a communication context switch is spent synchronizing. Were a separate,

Primitive	time
synchronization time t_{sync} in clocks	352
phase initialization time t_{phase} in clocks	176
total time context switch in clocks	528
access bandwidth (b in clocks/word)	$1/b$ ($b \geq 2$)
time send_item t_s in clocks	$2k$
time receive_item t_r in clocks	$2k$

fast global synchronization scheme available, the communication context switch would run even faster.

The network access rate actually depends on the way an application uses the connections. Our performance models fail to account for two hardware parameters: the per-hop forwarding latency in the physical network, and the slowdown due to shared physical links. In our experiments with large applications, we observed that neither were significant since we had a fast machine with only 64 cells, and computation was overlapped with communication.

4.3 The implementation of iWarp message passing

iWarp message passing is implemented by constructing short-lived pathways on an as-needed basis. These pathways are established on-the-fly from a fixed set of logical channels reserved exclusively for message passing in each of the four torus directions. Tagged pathway control words in the header and the trailer of a message signal the hardware to perform proper message forwarding as they pass through each communication agent. Messages are forwarded word-by-word through intermediate nodes, beginning as soon as the message header is detected; this scheme is known as *wormhole routing* .

The present iWarp message passing software uses *deterministic, non-adaptive routes* specified by the sender. *Routing deadlock* is prevented by establishing a directed acyclic graph of routing dependencies as previously described in [DS87] and [Str91]. The logical channels are divided into two reservation pools. Messages are always launched using logical channels reserved from pool 0. As messages cross a predetermined "dateline" on the torus, they switch to using logical channels from pool 1. This breaks the cyclic dependency and prevents routing deadlocks.

Hardware flow control is provided at each cell along a pathway. If a receiver fails to promptly remove its incoming messages from the pathway, the body of a message can back up in the channel buffers and block traffic between unrelated processors. This is just a form of congestion since, as long as progress can be made on one receiver no deadlock occurs. A few programming precautions are all that is needed to prevent deadlocks. This built-in hardware flow control, combined with the intrinsically reliable network, obviate the need to send acknowledgments for received messages.

In our current implementation we use *sender-buffered* messages as a way to decouple sending from receiving. The receiver actively receives and processes messages in the foreground; the transmitter sends them as a background task. This technique of decoupling, buffering and copying is distinct from the methods previously described in the literature, where buffering is usually done on the receiver side. Other approaches include a door-to-door message passing, as pointed out in the early iWarp architecture papers [BCC+88], and active messages [ECGS92] where data from the network is received immediately by a user written interrupt handler.

On the current iWarp system the overhead for sending and receiving messages is relatively high prohibitively so for short messages. This limits the effectiveness of *randomization* to balance the congestion but simplifies message handling at the receiver because all messages transferred between two nodes arrive in order.

4.3.1 The iWarp message passing performance

iWarp message passing can be characterized using the parameters of Section 3.1. The following table captures most of them for a k word message on an N processor machine. (The header and the trailer of a message consist of four additional data words.) All times are specified in clocks.

Primitive	send/receive ($i \in \{s, r\}$)
constant message overhead c_i	400
network access bandwidth $1/a$ (a in clocks/word)	$1/a_i$ ($a_i = 2$)
total time t_i in clocks	$2k + 416$

The per-message sending overhead is the composite of the latencies for event handlers, queue management, route lookup, and setting up the spooling units. The receiver overhead is somewhat smaller since no event handlers are invoked. Note that overlapping of sending and receiving can at best reduce the message passing time by one transfer time.

In a two dimensional torus the network latency l is $\Theta(\sqrt{N})$. The latency of each message is bounded from below by the network access time $a = 2$ clocks and from above by $4\sqrt{N}$.

4.4 The communication compiler

In this section we examine the interaction of the algorithmic problems of routing and scheduling with the architecture.

Few algorithms contained in a complex parallel application impose a complete order on the use of the connections. For example, our sparse matrix vector multiply kernel does not impose any restrictions on the ordering of the communication pattern occurring during execution. Some algorithms (such as the bitonic sort) might use the connections of a hypercube in a specific order, grouped in sets according to the cube dimension.

The communication compiler addresses two problems. First, each connection in the algorithm must be assigned to one or more phases during which the connection will be active. Each phase's set of connections must be routed within the constraint of a limited number of logical channels. Second, for each processor the partial order (which describes the connection use in the algorithm) must be extended to a complete order.

Optimally partitioning a connection pattern into phases and finding the best communication schedule is a complex NP-complete problem, but it can be decomposed into subproblems resembling others with well-known heuristic solutions. The first subproblem, the automatic phase partitioner and router, is very similar to VLSI circuit layout. This type of problem has been extensively studied and a comprehensive description of methods and heuristics are found in [Len90]. The second subproblem, the automatic scheduler, contains well-known graph problems [GJ79].

4.4.1 The automatic partitioner and router

The input to the phase partitioner and router (APR) is the set of connections needed within a task. This set of connections should be embedded onto the iWarp communication network with a minimal number of phases without exceeding the number of logical channels available. In some cases, ordering constraints between connections need to be satisfied as well.

The iWarp communication network is modeled as a graph: each processor is represented by a node in the graph having a capacity equal to the number of logical channels available on the processor. Each physical link in the machine corresponds to an edge in the graph. Embedding a connection onto the iWarp thus means finding a *route* (path) in this graph while ensuring that the number of routes which pass through a node is less than or equal to the node's capacity. This is an important change from traditional measures used when choosing a set of routing paths: the *maximum congestion* of a set of routing paths is of far greater importance than its *maximum dilation*.

Several heuristics could be adapted to our problem [NRT87] but the most promising results so far were achieved using a sequential routing technique based on shortest path search. These heuristics are very robust, and given the relatively small problem size (compared to circuit layout problems) we can avoid most of the well-known limitations. The basic sequential routing algorithm which solves the routing problem for one phase is outlined in the algorithm below.

```
algorithm APR (one phase)
  1  order connections
  2  initialize node costs to 1
  3  for route (i.a.)
       find shortest path (e.g. use Dijkstra's algorithm)
       update node costs
  4  repeat  k times: ripup and reroute
```

If all connections have been routed after step 3, we have found a solution, but not necessarily a good one. Since the quality can depend on the initial ordering, the communication compiler continues with a fixed number k of ripup and reroute iterations. During these phases some number of previously assigned routes are selected to be ripped up and the node costs are decremented to reflect this change; after ripup the connections are rerouted one by one. In this manner it is often possible to find a route for a previously unroutable connection.

In the compiler this scheme is generalized to multiple phases by modeling each phase with a copy of the graph, searching for a shortest path in all copies of the graph, and then selecting the shortest one which obeys the partial ordering constraints. Our current version of APR is tailored to finite element applications (which have no ordering constraints) but it can easily be extended to handle ordering constraints as well.

4.4.2 The scheduler

The input to the automatic scheduler (AS) is a set of processor to processor connections for one phase, together with the amount of data to be communicated over each connection, and optionally, a set of ordering constraints. The automatic communication scheduler assumes a specific machine model which defines the way a program can use a set of arbitrary connections (i.e. usable bandwidth per connection, number of connections to be used simultaneously, the amount of data to be transferred, and optional ordering constraints). Since we observe in our application that the amount of data differs significantly from connection to connection (by factors of up to 200 in FEM patterns), it is not appropriate to solve the scheduling problem by graph coloring (e.g. [Viz64]). Rather, we use a greedy heuristic to fill in a global timetable.

In the algorithm given below the value of MIN_TIME is a lower bound on the time needed to execute the schedule on each processor. The maximum of all MIN_TIME values denotes the lower bound on the execution of a whole phase.

```
algorithm AS (simple, without ordering constraints)
   compute the predicted duration d_c of communication on
       each connection c
   initialize global time table
   until all connections are scheduled do
      for the processor with the maximum MIN_TIME do
         find the connection c with the largest d_c
         schedule c in the first sufficiently large
              unfilled time region on both processors
      update MIN_TIME
```

This simple heuristic scheduler gives satisfactory results. An extended scheduler incorporating congestion information will be described in the forthcoming technical report [FSW93].

5 Experimental work

In this section we compare an implementation of ConSet to an implementation of message passing. The experiments were all carried out at Carnegie Mellon University on a 64 cell iWarp system (peak performance 1.2 GigaFLOPS). We compare execution times for two classes of programs: a set of basic communication primitives, and the communication within a parallel iterative finite element solver. The two implementations each received approximately the same level of architectural support from the machine. Out of 16 logical channels per processor available to user programs, the ConSet model used 12 for communication and 3 for fast synchronization. The message passing system also used 12 logical channels for communication, divided into 3 reservation pools. 4 additional logical channels were used to compensate for the lack of a "channel free" interrupt in the processor.

Communication performance is highly dependent on how the data is sent, how it is received, and how it gets merged into the computation. iWarp offers a variety of data transfer modes with different set-up times, peak transfer rates and processor utilizations:

streaming – Data is transferred to or from memory on a word-by-word basis under strict program control, usually within a tight loop. Normal instructions can saturate one link either sending or receiving. LIW coding allows saturating one link sending and another receiving.

systolic – Data elements are sent to the network as the result of a computation or indirect load operation. Data elements received from the network are immediately used as an operand for a computation.

spooling – An asynchronous, background transfer of data from memory directly to the communications network (or vice-versa).

An iWarp cell can readily sustain 80 MBytes/sec I/O with any of the above communication modes. The ConSet model can utilize any of these data transfer modes, while message passing must use spooling at the sender to achieve the buffering necessary for correct operation. The spooling and systolic modes can hide some or all of the communication costs if computation and communication are overlapped. For our experiments, (basic pattern and finite elements), we used the best possible transfer modes in each of our implementations.

The experiments were carried out on an commercially available parallel system rather than on a communication architecture simulator. As such, we had to deal with a number of difficulties arising from the compiler and the run-time system supplied with iWarp. Our results, therefore, have some caveats and should be perceived as first evidence for the performance of ConSet rather than as a final proof. In message passing, the large overheads are mainly due to a cumbersome event dispatch mechanism in the iWarp run-time system. By rewriting some system software and utilizing a few registers for storing message passing state, a 20-30% performance improvement is likely. This improvement would be visible to all programs and applications. The communication part of the finite element solver is presently written without LIW instructions in the inner loops. Software pipelined LIW instructions, by better overlapping communication with computation, could result in a 33-50% communication speedup. The communication compiler of the ConSet model can be improved by integrating scheduling data into the router, which in turn will improve the utilization of the connections.

5.1 Basic communication patterns

We created test programs to exercise each basic communication pattern using both the ConSet and message passing models, and ran them with data sizes of 32, 256, 2K, and 16K words.

We compare our measured performance times to what our models predicted. The quantities of interest are the total execution time in measured and predicted clocks and the aggregate communication bandwidth reached for that pattern. The aggregate bandwidth for 64 cells is computed and graphed as follows:

$$b_{\text{agg.}}(\frac{\text{MB}}{\text{sec}}) = \frac{\text{total amount of data transferred (Bytes)}}{\text{execution-time (clocks)}} \times \text{clock-rate (MHz)}.$$

In ConSet, data is transferred by "streaming" because of its low startup overhead. Message passing used "spooling" at both ends to free up the processor as much as possible for protocol processing.

5.1.1 The torus communication pattern

The torus pattern results from next neighbor communication including the backloops. Neither communication models experiences congestion on the physical torus, hence the performance predictions are fairly accurate.

data words	predicted ConSet	measured ConSet	predicted MsgPass	measured MsgPass
32	776	1,168	3,456	3,760
256	2,568	2,984	5,248	5,720
2048	16,909	17,408	19,584	20,012
16384	131,592	132,768	134,272	135,224

Table 1: Torus communication pattern, with four different data transfer sizes, predicted & measured *execution times* for ConSet and Message Passing on a 64 cell iWarp.

For next neighbor communications, the aggregate bandwidth of our implementations is limited by the network access bandwidth for "useful" the data transfers (64 cells $*$ 40 MB sec^{-1} cell^{-1} $=$ 2560 MB sec^{-1}) rather than the total bandwidth of the physical links (4 links $*$ 64 cells $*$40 MB sec^{-1} cell^{-1} link^{-1} $=$ 10240 MB sec^{-1}).

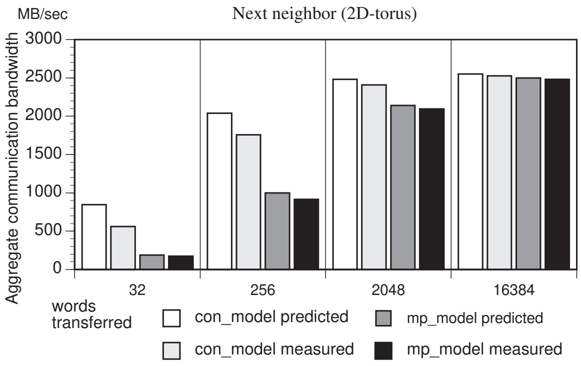

Figure 2: *Aggregate bandwidth* for the torus communication pattern, predicted & measured, four different data transfer sizes, for both ConSet and Message Passing on a 64 cell iWarp.

5.1.2 Hypercube communication pattern

Hypercube connections are required to support FFT butterflies or bitonic compare-and-swap steps. Given a Gray code mapping as input, our communication compiler routes the same optimal communication pattern as we used in previous work on a fast sorter [Str92]. There is a single-phase mapping for 16 logical channels, but since we constrained the communication compiler to 12 logical channels, it routed the pattern as two phases of 8 logical channels each.

data words	predicted ConSet	measured ConSet	predicted MsgPass	measured MsgPass
32	1,424	2,448	5,184	6,208
256	4,112	6,952	7,872	10,792
2048	25,616	43,182	29,376	48,320
16384	197,216	332,848	201,408	343,192

Table 2: Butterfly communication pattern, with four different data transfer sizes, predicted & measured *execution times* for ConSet and Message Passing on a 64 cell iWarp.

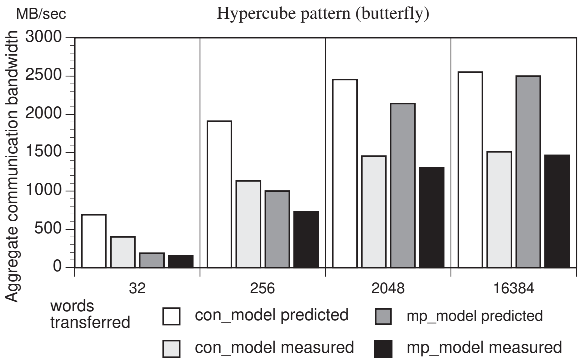

Figure 3: *Aggregate bandwidth* for the butterfly communication pattern, predicted & measured, four different data transfer sizes, for both ConSet and Message Passing on a 64 cell iWarp.

On iWarp tori the bisection bandwidth (16 links $*$40 MB sec^{-1}

link^{-1} = 640 MB sec^{-1}) is smaller than the network access bandwidth. Both routers, on-line message passing and the off-line communication compiler, have to deal with congestion. Our simple performance models with (just overhead and transfer time) predict execution time less accurately for the hypercube pattern.

5.1.3 All-to-all communication pattern

The all-to-all communication pattern represents a fully-connected network. The pattern is used by parallelizing compilers for distribution alignments, permutations and row-column transpose operations. The communication compiler partitioned the all-to-all pattern into 30 phases. The discrepancy between our simple model and measurements is that the routers have to resolve a large amount of congestion, a parameter not captured in the predictions.

data words	predicted ConSet	measured ConSet	predicted MsgPass	measured MsgPass
32	19,631	54,112	55,296	120,960
256	47,856	184,544	83,968	274,136
2048	273,648	1,258,616	313,344	1,358,632
16384	2,075,448	9,973,024	2,148,352	11,210,048

Table 3: All-to-all communication pattern, with four different data transfer sizes, predicted & measured *execution times* for ConSet and Message Passing on a 64 cell iWarp.

To alleviate congestion in message passing all messages were sent in random order. The execution time of the all-to-all communication is limited by the bisectional bandwidth. This limitation was confirmed by an experiment with a surface router (no routes across the backloops). As expected the transfer rates were approximately half of the torus router.

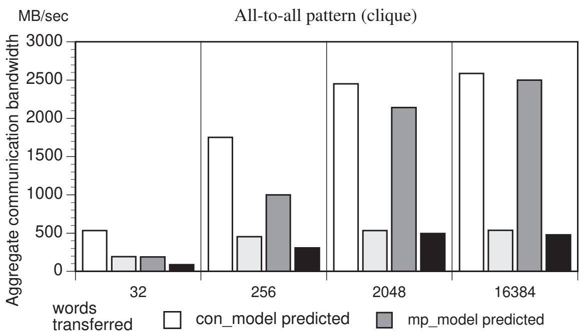

Figure 4: *Aggregate bandwidth* for the all-to-all communication pattern, predicted & measured, four different data transfer sizes, for both ConSet and Message Passing on a 64 cell iWarp.

5.1.4 Observations

Neither implementation achieves the available bandwidth, each suffering from a different cause. The ConSet communication compiler cannot completely balance the communication within nor among the phases. This hurts because a phase persists until the final transfer completes, and all the other processors are kept waiting. Our performance model accounts for this effect, but then ignores slowdown due to multiple logical channels sharing the bandwidth of one physical link.

In message passing, some messages block others and can not be re-routed even if bandwidth is available on an alternate route. Since the congestion delay for long messages is a highly dynamic, timing dependent process, it is not included in performance predictions.

For large block transfers (16K-words), execution times accurately reflect bandwidth considerations. The significant performance difference between the two implementations is observed with smaller (32 word) messages. In this case, the performance of message passing is strongly dominated by its high per-message processing overhead. ConSet has no per-message overhead but there is a per-phase synchronization and switch cost.

5.2 Partitioned finite element graphs

These experiments were motivated by a finite element solver for parallel distributed machines currently under development at Carnegie Mellon [SBF$^+$92].

In our finite element solver the super-graph of the partitioned finite element mesh exactly describes the communication pattern in the kernel of the iterative solver. The pattern results in a set of connections which must be established for every parallel sparse matrix vector multiplication in the kernel.

Four meshes were studied: *m11k* is a synthetically generated 2 dimensional discretization of a solid plate with 11000 random nodes. *c10k and c100k*, are refined meshes to model the stress zones around a crack in a solid plate. *q20k* is a 3 dimensional mesh representing a alluvial valley surrounded by hard rock, used by the scientific community for earth quake simulations. The structure of the meshes, their partitioning and the routing by our communication compiler is summarized in Table 5.2.

	m11k	c10k	c100k	q20k
Elements	20998	20141	204873	20000
Nodes	11000	10240	105000	22451
Elements per proc.	min/max 307/346	min/max 294/341	min/max 2987/3444	min/max 286/330
Comm. per proc.	min/max 47/133	min/max 46/189	min/max 116/638	min/max 357/1070
in/out-degree	8	5	13	22
logical chan.	12	12	12	12
phases	2	2	3	5

Table 4: Some structural parameters of the partitioned finite element mesh graphs and the number of phases determined by our communication compiler for routing with a given set of resources.

The amount of data exchanged after each solver iteration between any pair of processors depends on the number of shared FEM nodes. The histogram in Figure 5 illustrates the quantitative distribution of the data to be moved. A better description of the structure of the partitioned mesh graphs and the resulting communication patterns was omitted due to space limitations and can be found in [FSW93].

The execution times are reported for a single matrix vector multiplication. In the conjugent gradient method, this is the major computation and communication done in a single iteration. With good conditioning, $O(\sqrt[3]{\text{nodes}})$ iterations are required to solve a problem.

The communication step in the finite-element code sends and

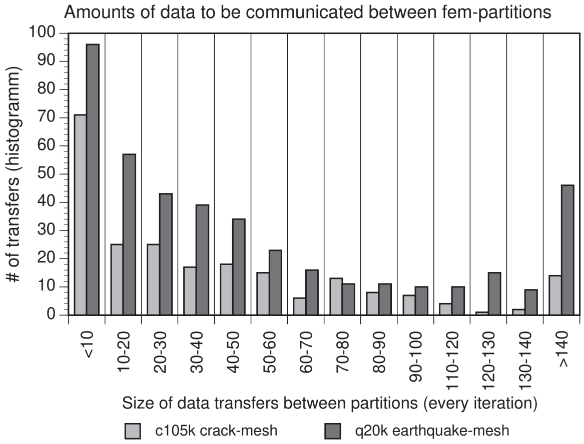

Figure 5: Histogram of the amount of data to be communicated between each pair of processors during the communication step of the iterative FEM solver.

FEM mesh	comp. cycles	comp. MFLOPS	comm. ConSet	comm. bandwidth
m11k	22,696	189.44	5,352	127.23
c10k	21,064	199.04	5,816	166.38
c100k	186,632	200.96	20,432	159.87
q20k	118,096	273.92	37,152	147.75

Table 5: Per iteration execution times for local computation and communication in the kernel of a simple conjugent gradient method solver measured *execution times* for ConSet on a 64 cell iWarp.

receives data that is stored in sparse matrix data structures. The send and receive primitives in ConSet (one floating-point number at the time) allow us to use a systolic transfer mode to gather the data from an indexed vector structure in the send operation, and to do a scatter and sum operation at the receiver. In the message passing code the data is copied to a buffer before sending rather than transferred directly. The receiver ends of both implementations are identical.

Analyzing the present code in the send and receive loops, we recalculated the usable access bandwidth ($\frac{1}{b}$) to be ($\frac{1}{10}$) in the model. The network access rate is no longer at peak speed which results in link congestion of up to three becoming invisible. Therefore, the performance models are fairly accurate for the communication patterns of all four finite element meshes.

fem mesh	predicted ConSet	measured ConSet	predicted MsgPass	measured MsgPass
m11k	4,786	5,352	8,378	8,730
c10k	5,066	5,816	6,752	10,968
c100k	19,774	20,432	19,508	24,164
q20k	35,910	37,152	32,852	47,700

Table 6: Finite element mesh pattern, with four different data transfer sizes, predicted & measured *execution times* for ConSet and Message Passing on a 64 cell iWarp.

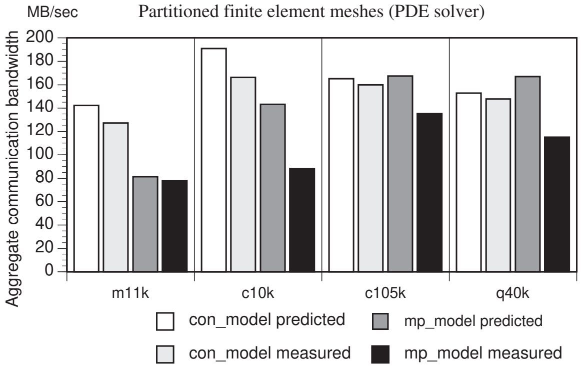

Figure 6: Finite element mesh pattern, with four different mesh sizes, predicted & measured *aggregate bandwidth* for ConSet and Message Passing on a 64 cell iWarp.

5.2.1 Observations

The measured communication times for message passing indicate that for small to mid-range finite element problems, large communication overheads noticeably affect application performance. The connection-based implementation does not suffer from these overheads and therefore performs better on all meshes we investigated.

The gap between the two models closes as the problem size increases, but given the computation and memory requirements of the finite element solver, we do not expect that much larger problem sizes will be viable for current parallel computers.

The data transfers occuring within real applications often include gathering and scattering the values from sparse matrix data structures. Our measured bandwidth for the sparse matrix vector multiply is nowhere near the bandwidth achieved in the basic communication patterns. We attribute this to the indirection needed to access the packed sparse data structure. While the sustained bandwidth is only a fraction of the peak attainable bandwidth, iWarp's fast communication network ameliorates the effects of congestion and latency. Furthermore, it allows us to overlap the address computation with the data communication.

6 Conclusion

We proposed ConSet, a flexible model for connection-based communication on distributed memory parallel machines, motivated by our experiences with an iterative finite element solver. We successfully implemented ConSet on the iWarp parallel computer using the mechanism of a communication context switch. This mechanism allows us to extend the number of connections available to an application. We believe that our model and its implementation contributes evidence that "reconfigurable networks" could exist today although our model is limited in that it does not provide bus based broadcasts.

In an experimental comparison we observed that the connection-based communication may offer a performance advantage over message passing on a system with support for both styles. We showed the benefit of globally optimizing communication patterns at compile time rather than making the routing decisions at run-time in the interconnect network itself. The mechanism of a communication context switch suggests that it might be simpler and faster to swap a whole set of connections by loading the state of the communication hardware than it would be to establish them on a per connection basis. Limiting factors to our implementation on the iWarp included the lack of a fast global synchronization mechanism and the difficulty of programming communication state. This is not surprising since we used the communication agent of the iWarp machine in an unforeseen way.

We acknowledge that message passing provides a simpler programming paradigm and offers more flexibility than long lived connections. Still, we think that architects of future parallel machines should consider supporting connections by providing logical channels, fine-grain communication access, and an atomic mechanism for fast connection reconfiguration to increase application performance.

Acknowledgments

We would like to thank Jonathan R. Shewchuk for assistance with the FEM application, Gary Miller for general guidance, Eric Schwabe for early work on the communication compiler, and Thomas Gross, David O'Hallaron, and the rest of the CMU/ Intel iWarp group for their support.

References

[BAPRS91] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster. The power of reconfiguration. *Journal of parallel and distributed computing*, Vol. 13, 1991.

[BCC+88] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, FL, USA, November 1988. IEEE Computer Society and ACM SIGARCH.

[BCC+90] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proceedings of 17th Int. Symposium on Computer Architecture*, May 1990.

[BS87] R.P. Bianchini and J.P. Shen. Interprocessor traffic scheduling algorithm for multiple-processor networks. *IEEE Transactions on Computing*, Vol. 36, April 1987.

[DS87] W. Dally and C. Seitz. Deadlock free message routing. *IEEE Transactions on Computers*, Vol C-36, May 1987.

[FSW93] A. Feldmann, T.M. Stricker and T.E. Warfel. Supporting sets of arbitrary connections on iWarp through communication context switches. To appear as Tech. Rep. CMU-CS-93-142, School of Computer Science, Carnegie Mellon University, 1992.

[Fel88] E.W. Felten. Generalized signals: an interrupt-based communication system for hypercubes. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 563–568, Jan 1988.

[FLM+92] R. Funke, R. Lüling, B. Monien, F. Lücking, and H. Blanke-Bohne. An optimized reconfigurable architecture for transputer networks. In *Proc. of 25th Hawaii Int. Conf. on System Sciences (HICSS)*, 1992.

[GJ79] M.R. Garey and D.S. Johnson. *Computer and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, 1979.

[Gro89] T. Gross. Communication in iWarp systems. In *Proceedings of Supercomputing '89*, pages 436–445, Reno, NV, USA, November 1989. IEEE Computer Society and ACM SIGARCH.

[Joh87] C. Johnson. *Numerical Solutions of Partial Differential Equations by the Finite Element Method*. Cambridge University Press, 1987.

[Ka93] R. Karp et al. The log P model: towards a more realistic model for Parallel Computation, Westcoast Video Conference Lecture Series, January 1993.

[LAD+92] C.E. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynmann, M. Ganmukhi, J. Hill, W. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. In *1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 122–129, June 1992.

[Len90] T. Lengauer. *Combinatorial Algorithms for Circuit Layout*. John Wiley & Sons, 1990.

[Lil90] S.L Lillevik. Delta: a 30 gigaflop parallel supercomputer for touchstone. In *Northcon. Conference Record*, pages 294–304, Oct 1990.

[LM89] H. Li and M. Maresca. The polymorphic-torus network. *IEEE Transactions on Computing*, Vol. 38, 1989.

[LS91] H. Li and Q.F. Stout. Reconfigurable SIMD massively parallel computers. *Proceedings of the IEEE*, Vol. 79, April 1991.

[NRT87] A.P.-C. Ng, P. Raghavan, and C.D. Thompson. Experimental results for a linear program global router. *Computers and Artificial Intelligence*, 6:229–242, 1987.

[NSS+88] D. Notkin, L. Snyder, D. Socha, M.L. Bailey, B. Forstall, K. Gates, R. Greenlaw, W. Griswold, T. Holman, R. Korry, G. Lasswell, R. Mitchell, and P.A. Nelson. Experiences with Poker. In *Proceedings ACM/SIGPLAN PPEALS*, 1988.

[Nug88] S.F. Nugent. The iPSC/2 direct-connect communications technology. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 51–60, Jan 1988.

[SBF+92] E.J. Schwabe, G.E. Blelloch, A. Feldmann, O. Ghattas, J.R. Gilbert, G.L. Miller, D.R. O'Hallaron, J.R. Shewchuk, and S.-H. Teng. A separator-based framework for automated partitioning and mapping of parallel algorithms for numerical solution of PDEs. In *Proceedings of the 1992 DAGS/PC Symposium*, pages 48–62, June 1992.

[SL89] D. Smitley and I. Lee. Synthesizing minimum total expansion topologies for reconfigurable interconnection networks. *Journal of Parallel and Distributed Computing*, 7:178–199, 1989.

[Sny82] L. Snyder. Introduction to the configurable highly parallel computer. *Computer Magazine*, 15, Jan 1982.

[Sny84] L. Snyder. Parallel programming and the poker programming environment. *Computer Magazine*, 17, July 1984.

[Str91] Stricker, T.M. Message routing on irregular 2D-meshes and tori. In *Proceedings of the Sixth Distributed Memory Computing Conference*, April 1991.

[Str92] Stricker, T.M. Supporting the hypercube programming model on mesh architectures (a fast sorter for iWarp). In *1992 ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 1992.

[UFR92] E. Upfal, S. Felperin, and P. Raghavan. A theory of wormhole routing in parallel computers. In *Proceedings of the 33st Annual Symposium on Foundations of Computer Science*, Oct 1992.

[Val90] L.G. Valiant. General purpose parallel architectures. In *Handbook of theoretical computer science*. Amsterdam; New York: Elsevier; Cambridge, Mass.: MIT Press, 1990.

[ECGS92] T. Von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Int. Symp. on Computer Architecture*, pages 256–266, 1992.

[Viz64] V.G. Vizing. On an estimate of the chromatic class of a p-graph. *Diskret. Analiz.*, 3:25–30, 1964.