Tasks and Connection Sets: Choreographed Communication on a Reconfigurable Connection-Based Parallel Computer

Thomas E. Warfel

April 1996
CMU-CS-96-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Thesis Committee:
H.T. Kung, Chair
Thomas Gross
David R. O'Hallaron
Daniel P. Siewiorek
Jay Strosnider

This research was sponsored in part by the Defense Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152, and in part by the Air Force Office of Scientific Research under contract F49620-92-J-0131.

Keywords: Parallel computing, reconfigurable networks, barrier synchronization, communication context switch, virtual connections, virtual circuits, bandwidth reservation, communication scheduling

Abstract

High-bandwidth, high-throughput applications with hard latency constraints are difficult to implement on a general-purpose parallel computer. Multiple developer-controlled “trial-and-error” cycles are usually needed before applications can reliably meet throughput and latency constraints, even on platforms having ample network bandwidth and computation power. Not only is reliable execution difficult to achieve for code developed in this manner, the code itself is difficult to modify or reuse without upsetting the delicate timing balance achieved.

Local computation performance can usually be bounded, but communication performance is often more difficult to predict. While hardware-supported connections can offer minimal quality-of-service bandwidth and latency guarantees, limited connection resources make scheduling the full application difficult. This thesis introduces a new approach: use multiple sets of connections, and allow tasks to perform ***local communication context switches*** and dynamically swap, within tasks, between statically scheduled sets of connections.

The mechanics of swapping connection sets, starting a task, and ending a task can be encapsulated into a small set of control primitives built upon fast, efficient ***barrier synchronization***. If the control primitives are constructed to give predictable performance, the tasks created using those primitives will have predictable performance as well. Most important, complex tasks can be hierarchically constructed by assembling simpler tasks into larger structures while still maintaining predictable performance.

To demonstrate this scalable predictability, the ***TCS*** (**T**asks and **C**onnection **S**ets) programming model is introduced and implemented on a real target machine, iWarp. The prototype is used to implement a variety of communication patterns and then compared with fast message-passing implementations on the same machine. Finally, the scalable, hierarchical nature of TCS tasks is demonstrated by implementing a portion of a real-time computer vision application. TCS is shown to be well-suited not only for this application, but also for similar applications requiring continuous high-bandwidth input, low-latency output, and multiple computations per datum.

Acknowledgements
I would like to thank a number of people who enabled my completing this work. H.T. Kung, my advisor, for giving me the chance. Thomas Gross and Dave O’Hallaron, for the late-night/weekend meetings and discussions. Carl Love, Elaine Lawrence, and Lynn Philibin for guiding me through myriad vital paperwork over the years. Joseph Furman, who enabled me to keep my “dual identity” at Pitt while working at CMU. The support of friends here, especially Tom Stricker and Anja Feldmann, with whom many ideas were hatched and nurtured over the years and coffee. The members of the Nectar/iWarp research group, my office mates through the years, and the Carnegie Tech Amateur Radio club, for both critiques and moral support. Last but not least, my family, for their unflagging love, support and encouragement.

Table of Contents

Chapter 1 - Introduction

1.1 Introduction

High-bandwidth, high-throughput applications with hard latency constraints are often difficult to implement on a general-purpose parallel computer. Hardware-supported connections can offer minimal quality-of-service bandwidth and latency guarantees, but finite connection resources makes application scheduling difficult; machines usually lack sufficient connections to enable statically scheduling a complex application. Conversely, a purely dynamic connection resource allocation scheme may not be able to guarantee resource availability at run-time, which could lead to missed latency constraints. A hybrid approach that can work for many of these applications is to use multiple sets of connections, allowing tasks to perform ***local communication context switches*** and dynamically swap, within tasks, between statically scheduled sets of connections.

The mechanics of swapping connection sets, starting a task, and ending a task can be encapsulated into a small set of control primitives built upon fast, efficient ***barrier synchronization***. Expressing the application using these primitives exposes the application’s potential runtime communication complexity to the linker, which can then make globally-optimal communication resource allocations. One knows at link time whether or not sufficient resources exist to meet the run-time demands; there are no surprises with run-time resource unavailability. Furthermore, if the control primitives are constructed to give predictable performance, the tasks created using those primitives will have predictable performance. Most important, complex tasks can be hierarchically constructed by assembling simpler tasks into larger structures while still maintaining predictable performance.

To demonstrate this scalable predictability, the ***TCS*** (**T**asks and **C**onnection **S**ets) programming model is introduced and implemented on a real target machine, iWarp. TCS allows parallel tasks to perform ***local communication context switches***, reliably swapping (in predictable time) between predefined sets of connections having guaranteed worst-case latency and bandwidth. Three key machine features are required to support TCS:

(1) the network switches must allow their connection state to be directly configured by the local processing elements,

(2) connections must be reliable and offer guaranteed worst-case bandwidth and latency, and

(3) some form of fast, reliable barrier synchronization must be available.

Unlike message-passing communication (which handles all communication resource assignments at runtime), TCS requires that the connection sets (but not their usage patterns) be known at compile time; communication resource assignment is resolved at link time. This link time global foreknowledge of the permissible runtime connection states allows the TCS toolchain to make communication resource assignments that will meet the requested bandwidth criteria, or else return an error message at link time. If a TCS application successfully links, the requested connection sets are guaranteed to be available at runtime. Realtime problems having deadlines on the order of milliseconds can be addressed by solutions with execution times predictable to within a few microseconds.

To demonstrate the utility and validity of the idea, the TCS prototype was used to implement a variety of communication patterns representative of real application patterns. For comparison, the same communication patterns were also implemented using a fast message-passing system on the same machine. While message-passing and TCS both can provide fast, predictable performance for uncongested patterns, dense communications patterns (such as all-to-all) lead to unpredictable link congestion which causes message-passing to lose both performance and predictability. TCS is shown to maintain good, predictable performance even with dense communication patterns.

Finally, the scalable, hierarchical nature of TCS tasks is demonstrated by implementing a portion of a real-time computer vision application. The vision application is realized as a TCS task constructed by assembling smaller TCS tasks. TCS is shown to be well-suited not only for this application, but also for similar applications requiring continuous high-bandwidth input, low-latency output, and multiple computations per input datum.

1.2 Why TCS?

Consider the following problem: a ball is thrown through the field of view of a watching camera. A computer attached to the camera locates the ball in several consecutive frames, then plots a predicted trajectory for the ball. Current ball position and predicted ball trajectory are superimposed over a display showing the live camera video (Figure 1.1). No hardware-supported frame-buffers are used; the only special hardware is a fast, unbuffered analog-to-digital converter (which converts the incoming video pixels to binary numbers), a comparitor to detect video sync edges, and a digital-to-analog converter to convert an output stream of pixel values to an NTSC video output. The real-time nature of this problem is apparent in that the incoming video pixels must be sampled, forwarded through the system, and output to the video monitor in a timely manner. Latencies are additive,

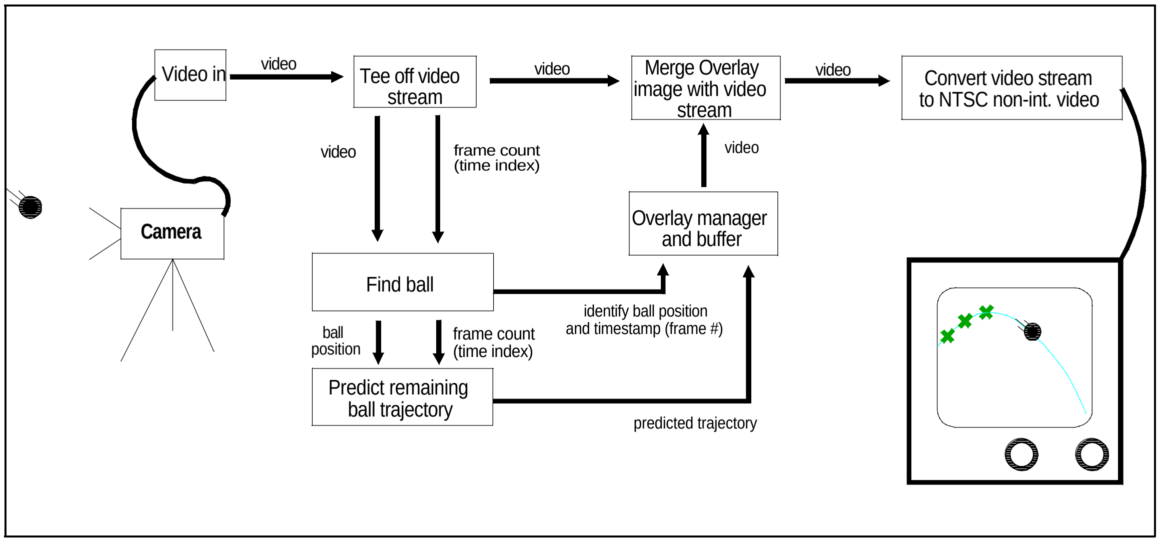

Video in
video
Tee off video stream
video
Merge Overlay image with video stream
video
Convert video stream to NTSC non-int. video
video
frame count (time index)
video
Camera
Overlay manager and buffer
Find ball
identify ball position and timestamp (frame #)
ball position
frame count (time index)
Predict remaining ball trajectory
predicted trajectory

Figure 1.1 Locate a thrown ball in a live video feed, then predict its future trajectory.

and thus for the system to be useful, the ball's position must be detected and future positions predicted and plotted all within one frame time. Otherwise, the result is just a "comet trail" drawn behind the ball on the screen. This is a fairly demanding (but statically schedulable) communication problem. What makes the problem interesting is that the "ball finding" computations and the trajectory prediction occur asynchronously with respect to the incoming video stream.

Other example applications with similar latency, computation, and throughput constraints include:

(1) Phased array multi-sensor acoustic processing, such as an ultrasonic anti-collision system on a car's rear bumper;

(2) Phased array sonar processing [35,36];

(3) Real-time medical imaging, including:

  (a) correcting for patient movements in the imaging plane "on-the-fly" while doing functional (multiple scan) MR imaging;

  (b) precisely quantifying radiation therapy dosages by generating a CT-like image "on-the-fly" from the radiation treatment (realtime noninvasive internal dosimetry), and comparing these against prior, conventional CT (Computed Tomography) scans used for dose planning, so that treatment can be redirected or aborted if sensitive tissues (such as the spinal cord) become overly-irradiated. Off-line portal image evaluation is discussed in [19] and [31] to detect damage inflicted, but on-line realtime 3D internal dosimetry is not yet practiced.

While by no means an exhaustive application list, the scope is broad enough to draw some generalizations. Common features shared by these examples include:

1) Large amounts of computation (multiplication and addition) are required per data point.
2) Real-world data is sampled in high-bandwidth, time-critical bursts.
3) The problems have some inherently parallel aspect, whether it

be multiple sensors acquiring data to be processed, or whether it be the means by which the data itself is processed

4) The output of the process has a time-critical nature; the output is often used as feedback in some sort of control loop which may or may not be completely automated (that is, a human may be in the loop).

1.3 The prototypical parallel target machine

A parallel computer exists as a group of ***cells*** interconnected via a communication network. Each cell is a single functional computer within the larger parallel machine, complete with processor, local memory, specialty I/O devices (if any), and a connection to the machine's communication network. While some architectures may use more than one processor per cell, for the purposes of this thesis the cell is treated as the smallest functional computing unit. Due to the real-time nature of the applications being addressed, stand-alone cells must offer predictable execution times.

Fast, predictable, low-latency interprocessor communication emerges as a requirement for this parallel machine. While not an explicit part of any application definition per se, little is gained if multiple cells can acquire high-speed data in parallel but cannot pass that data on for correlation at the same rate. Buffering can compensate for small discrepancies in bandwidth, but the basic communication capacity needs to be available. Fast communication involves two major issues: the communication protocol used (***how*** two cells talk), and the network implementation (which cells can talk to which other cells, how fast can they talk, and how many can talk at once).

Conventional supercomputers often accept high latencies as the price for high bandwidth, and accordingly pipeline their computations and data transfers in huge blocks[45,47]. For instance, while image N is being computed, data for image N+1 is being loaded, and image N-1 is being written out. If the computation goal is just to generate weather maps, this pipeline latency is not a problem. Due to the time-critical nature of applications such as those listed in section 1.2, though, long pipeline delays cannot be afforded. A driver backing up needs to know what's behind the car now, not what was behind the car several scans ago.

While general-purpose message-passing (such as offered by MPI libraries[15,27]) is a commonly used communication paradigm for parallel machines, a number of characteristics make it undesirable for the types of applications discussed. First of all, the overhead and unpredictable delays an interrupt-driven message-passing system implies can’t be afforded in a real-time control problem. Second, message-passing systems typically evaluate routing issues ("***how*** do I send a message from **A** to **B**") on a message-by-message basis at runtime. For all the applications shown, the necessary communication patterns can be worked out at compile time. The precise ***usage*** of those communication patterns may be unknown, but the patterns themselves can be known. It is far more efficient, then, to work out the communication resource and routing assignments once, when compiling or linking, rather than re-evaluating them for each and every message sent at runtime [17,22].

Instead of message-passing, a connection-like mechanism is needed for communication between processors. A connection acts as a "first-in, first-out" buffer connecting the output of one cell to the input of another. Data written into the connection (from the output of the sending cell) is available to be read out (at the input of the receiving cell) in the same order it was written in. The actual means by which connections are implemented is unimportant, provided that the implementation can offer minimal-quality-of-service bandwidth and latency guarantees, and that an adequate number of connections can be supported. These guarantees are necessary to insure that processors can forward data fast enough to keep up with input data bursts.

Point-to-point wires between communicating cells are the most direct means of supporting connections. This approach has several difficulties, the biggest being that communication paths are essentially "programmed with solder"; reconfiguring to support different communication patterns becomes impossible. Supporting multiple applications, each having different connection requirements, on a machine with finite resources, implies the ability to reconfigure the machine between application runs.

The ability to reconfigure connections while running an application (and not just between applications) is also desirable. To provide low-latency communication, any connection implementation requires some sort of direct hardware support. Because low-latency connections must rely on a finite physical resource, the total number available will have some finite limit. If an application requires more connections than the underlying implementation is able to support at one time, the application's needs could still be met if the implementation supports ***reconfigurable connections***. Reconfigurable connections allow resources to be allocated that guarantee minimal-quality-of-service for one connection, and when the connection is no longer needed, those resources can be revoked and reallocated to support another connection. Because most parallel applications exhibit a "locality of communication", only a few connections are usually needed during any particular stage of program execution[17]. Thus, a few reconfigurable connections are usually adequate to meet an application's needs.

1.4 "Tasking" - sharing the load

Once a specific parallel system is established as sufficient to meet the application's requirements, the challenge becomes mapping the application components, or tasks, to different cells within the machine. A ***task*** is a functional unit of computation; all applications consist of one or more communicating tasks. The specific cells that a task is mapped to are referred to as that task's ***allocation***. Two tasks running on different cell allocations are said to be ***parallel tasks***. Two non-communicating tasks which have at least one cell in common between their cell allocations are said to be ***sequential tasks***; they cannot both run at the same time. A task will not execute until all the cells of its allocation are ready to run that task.

Parallel tasks may either be synchronous or asynchronous. In a synchronous tasking model, all tasks start together and end together, much like a marching band. The brass, woodwinds, and percussion all start together, march together, and stop together. If an application requires multiple task sets over time, a global barrier separates the different task sets so that all tasks in a set begin together. Everything runs on a fixed schedule which must make worst-case assumptions; thus, tasks can be blocked due to conditions entirely beyond their concern. In a more flexible, asynchronous tasking environment, a task will only block until the resources it needs are available, then execute. This model more closely resembles dinner in a restaurant: arriving parties are seated and served as tables become available. Once the resources become available (a sufficiently large table becomes free), dinner proceeds independently of the other parties in the restaurant.

Actually, the restaurant analogy can be extended to illustrate some of the problems of synchronous tasking on a large parallel system. Consider a large catered dinner event, such as a wedding reception. In this case, arriving parties are seated and left to sip ice water until all other guests have been seated. Meanwhile, the servers are left standing idle. Once all guests are seated, the meal is served one course at a time. If a sufficient number of servers are available, all guests are simultaneously given their soup, then the soup bowls are cleared away. All guests are simultaneously given their salad, then the salad bowls are cleared away. No guest receives a salad until the last guest has had his soup bowl removed. Unfortunately, most catered dinners suffer from limited “busboy bandwidth”. Food service is not simultaneous, but rather occurs in a wave, as the servers shuttle food from kitchen to successive tables. Guests who have finished their soup are forced to wait until all other guests have finished their soup before they can begin their salad. The larger the group of guests (or the larger the number of processors in the machine) the worse this “wave of waiting” becomes. Globally synchronous execution in a parallel machine not only forces cells to wait for their neighbors at each stage, but also magnifies the problems of finite communication bandwidth. The asynchronous tasking model means cells spend less time waiting, but allocating communication resources becomes a more difficult problem.

The problem, in essence, is "***how*** can one combine connection-based communication (which implies static scheduling/resource allocation), with a flexible tasking model (which inherently involves dynamic resource allocation)?"

1.5 Thesis

By placing some restrictions on the tasking model (statically allocating the potential communication resources an application may need), the application goals (multiple interacting tasks, high-bandwidth I/O, multiple computations per data point, hard latency constraints) can be met while maintaining effective processor utilization. Given a parallel computer with connections having guaranteed minimal-quality-of-service and a local connection state that is directly-writable by the local computing cell, one can construct a small set of barrier-based control primitives that yield predictable performance. By exposing the communication complexity to the linker, these primitives can be used to create parallel tasks which also exhibit predictable performance, and those tasks can in turn be hierarchically assembled to create even more complex tasks while still maintaining predictability.

A prototype programming system, TCS, was created to demonstrate the validity of this hypothesis. TCS applications are composed of tasks that communicate via sets of unidirectional connections. Tasks can be hierarchically constructed by assembling simpler tasks, and complex communication patterns can be expressed as a series of local communication phases within the task. Tasks (with latency constraints in the tens to hundreds of microseconds) are built with a small set of barrier-based control primitives which offer predictable (to within a microsecond) performance. Properly constructed, tasks using these primitives also exhibit predictable execution times and can be assembled into more complex tasks that maintain their predictability. Their communication resources are statically scheduled by the linker as sets of connections within each task, but dynamically invoked by the task at run-time.

1.6 Structure of thesis

The next few chapters explore the characteristics of TCS connection-based communication and explain the hierarchical nature of the four TCS control primitives: ***barrier synchronization***, ***local communication context switch***, ***task start***, and ***task end***. Both the communication and the control primitives are implemented on a real target machine, and their performance is measured and compared with predicted performance.

Chapter 2 explains the TCS programming model in more detail and explains the functions of the ***barrier synchronization***, ***local communication context switch***, ***task start***, and ***task end*** primitives.

Chapter 3 introduces the target machine, iWarp, and outlines the three major communication mechanisms it provides: ***PCT-supported connections***, ***RTS message-passing***, and ***deposit message-passing***. These communication mechanisms are explored and characterized. PCT-supported connections are the mechanism used to implement TCS connections.

Chapter 4 deals with ***barrier synchronization***: what it is, relevant aspects, and ways to implement it. The interaction between a barrier implementation's ***physical signaling scheme*** and ***messaging protocol*** is first predicted, then illustrated by constructing and benchmarking barrier implementations built from the three communication mechanisms introduced in Chapter 3. Based on these results, a **1-D (N-1) ring** built using PCT-supported connections is chosen as the basis for the TCS barrier primitive.

Chapter 5 introduces the remaining TCS control primitives. The barrier primitive introduced in Chapter 4 underlies all dynamic resource allocation at runtime, and it is used in constructing the remaining three primitives: ***local communication context switch***, ***task start***, and ***task end***.

Chapter 6 uses the TCS control primitives and a prototype connection linker to create three single-task communication patterns representative of real application communication: scatter/gather, reduction/broadcast, and all-to-all. The TCS implementations are shown to have predictable (within a few percent) performance regardless of transfer size and number of cells. A message-passing implementation, based on deposits, was shown to have comparable performance and predictability with simple patterns on an unloaded machine, but as congestion increased, message-passing was unable to maintain predictable performance.

Chapter 7 demonstrates the hierarchical nature of TCS tasking, constructing a real-time video-rate motion-detector by assembling several simpler tasks. This composite task was predicted to meet video requirements as it was assembled, then it was benchmarked to verify predicted performance.

Chapter 8 discusses related work which is significant for using sets of connections, dynamic tasking on a parallel machine, or both.

Chapter 9 is the conclusion and summarizes the key points of the thesis.

Chapter 2 -

The TCS Programming Model

2.1 The model for addressing the problem:

TCS (for **T**asks and **C**onnection **S**ets) is a general computation model for reconfigurable connection-based parallel machines which exploits certain machine properties. In particular, special advantage is taken of hardware-supported, low-latency connections for communication within and between running tasks. Task-internal communication, and the synchronization barriers needed for connection resource management, are all concealed within the task that decouples the task's internal execution from its neighbors. Communication between tasks is self-synchronizing and is the only synchronizing operation crossing task boundaries.

Under this model, all communication occurs through unidirectional ***connections***. Connections provide communication both within tasks (internal connections) and between tasks (external connections)

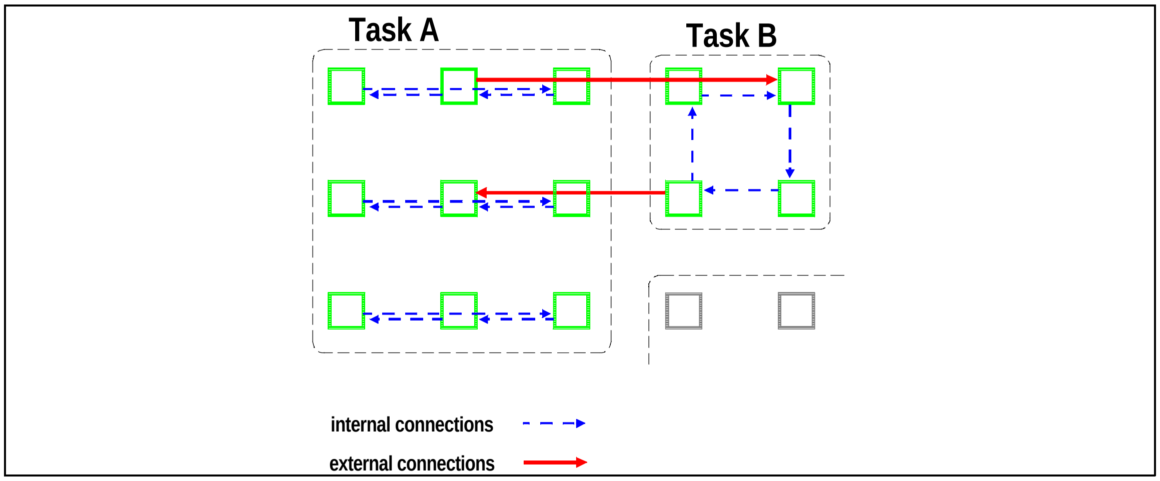


Figure 2.1 Cells within a task communicate via ***internal connections***. Inter-task communication occurs via ***external connections***.

(Figure 2.1). While the external connections persist for the lifetime of a task, internal connections within the task may be reconfigured under the task's local control. Connections are grouped into ***networks***, and networks are in turn grouped into ***netgroups***. A netgroup is just a set of local connections. A connection may only belong to one network, but a network may belong to more than one netgroup. A task may have only one netgroup active at any time. A connection is ***active*** if the network it belongs to is in the active netgroup; active connections may be used for communication. If the connection does not belong to the active netgroup, no communication resources are supporting it and it may not be used for communication. Tasks can perform ***communication context switches*** to change the active netgroup.

Good candidate applications for the model have the following characteristics:

(1) they process multiple "sets" of data;

(2) they can be expressed as a collection of communicating tasks, each task having:

   (a) a fixed set of communication patterns (but not necessarily knowledge of the order in which the patterns will be used), and

   (b) a good estimate of required execution time, though the actual run time may have data dependencies.

Having a fixed set of communication patterns allows static allocation of the communication resources, which in turn allows making some guarantees about minimum runtime communication performance. Having an accurate estimate of task execution time is important when mapping tasks to cells; using too few cells to support a task could result in a computational bottleneck, and using too many is a waste of resources.

Purely systolic applications, with a static set of connections ordered at compile-time, can be cleanly implemented using TCS, but would not see a substantial benefit over globally synchronous tasking models. TCS will allow efficient use of systolic tasks as part of a larger, non-systolic application, though.

The benefits of using the TCS model include:

(1) support for mapping problems (such as the examples shown) onto realizable parallel architectures;

(2) the ability to express loosely-coupled tasks without any artificial couplings; there is no requirement for the developer to construct artificial global phases. Eliminating artificial couplings enables faster performance by eliminating unnecessary synchronization barriers.

2.2 Tasking under the TCS model

TCS tasks rely on cell-to-cell connections and four control primitives: ***barrier synchronization***, ***connection reconfiguration*** (also known as a ***communication context switch***), ***task start***, and ***task end***. Connection (communication) performance is a function of the underlying hardware and communication resource scheduling. In the next few chapters the performance of the control primitives are characterized and (most important!) shown to be predictable (to within ten percent or better) using simple models. ***Barrier synchronization*** is the fundamental primitive upon which both ***connection reconfiguration*** and ***task start*** are both based. In fact, the TCS control primitives are hierarchical in nature, and thus a fast barrier implementation is a key implementation concern because it is repeatedly encountered at each hierarchical tasking level.

Tasks consist of program code executing on a predetermined (at link time) ***cell allocation*** as a coordinated entity, together with all communication generated by that program code, and the ***external ports*** used to communicate with other tasks. A task begins execution when it is invoked (***task start***) by a parent task; parent task operation is suspended on those cells, and the child task executes. When the child task terminates, parent task execution resumes. The ***lifetime*** of a task lasts from when all task members (the cell allocation) complete a barrier synchronization on task startup, until all members complete a barrier synchronization on task termination. Only one task may be actively executing on a single cell at a time.

A parent task can pass ***invocation parameters*** to the child tasks it starts. Each cell in the parent task's allocation passes the same *set* of parameters to the child task, but the values of the parameters can vary from cell to cell.

For example, consider Figures 2.2 and 2.3. In 2.2, an application is starting that includes the tasks **Video In**, **Tee-off**, and **Ball Detect**.

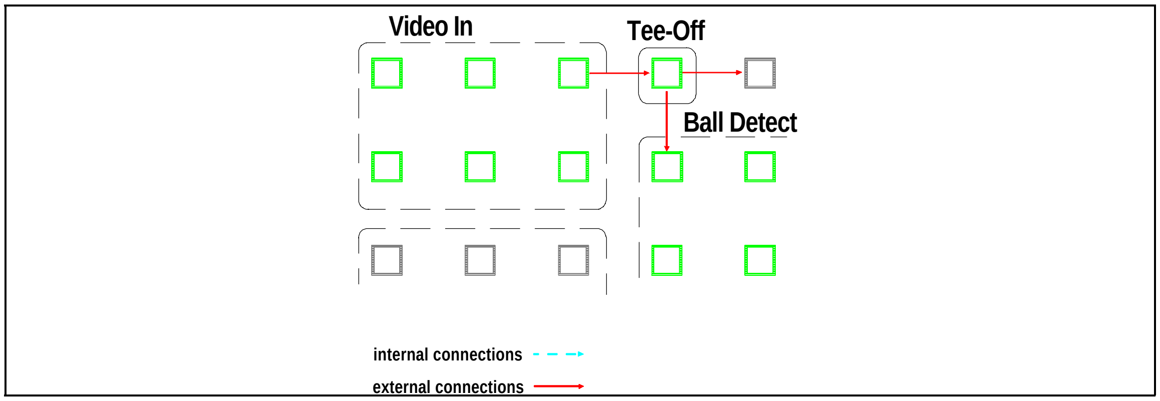


Figure 2.2 Three of the tasks used in the “predict and plot the ball’s trajectory” example.

Video In then invokes two child tasks, **Sample Camera** and **Pack Pixels** (Figure 2.3). **Sample camera** acquires data from four video cameras at once (4 bytes, 1 byte per camera, packed as one 32-bit word), and forwards the data to **Pack Pixels**, which takes 4 words, discards data from the 3 irrelevant cameras, and packs the 4 bytes

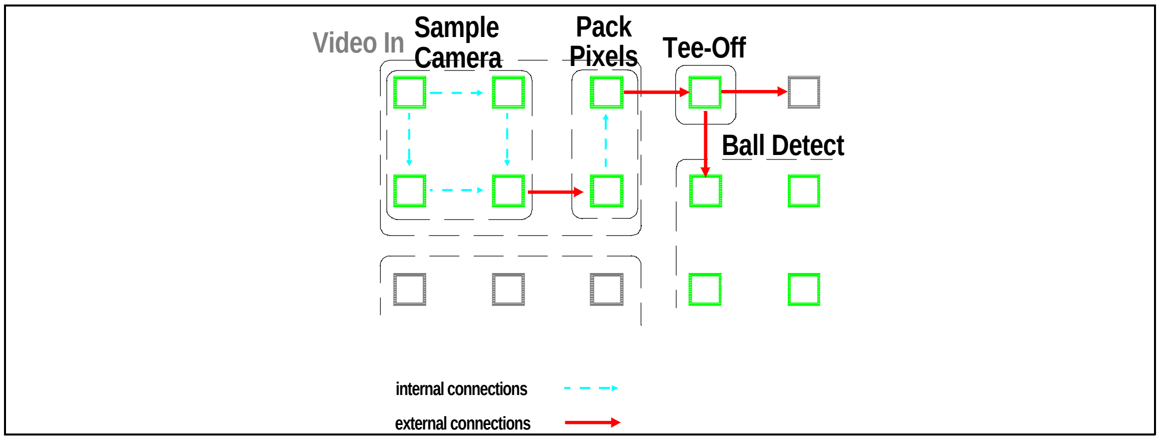


Figure 2.3 **Video In** has invoked two children, **Sample Camera** and **Pack Pixels**.

of data from the relevant camera into a new word and outputs it using an external connection inherited from the parent task (**Video In**). Thus, the complex task **Video In** has been constructed by assembling two simpler, smaller tasks.

2.3 Task relations

As a task begins execution, all members of the task’s cell allocation synchronize to verify that all cells needed to run that task are indeed ready. If the task has any "personal" external connections (as opposed to an external connection inherited from a parent), the local work needed to set up an external connection is done, and another synchronization is performed, which now includes both communicating tasks' cell allocations. This second barrier is necessary to ensure that no data is sent before the receiving end of the connection is established. External connections persist for the entire lifetime of a task, hence, an additional barrier synchronization is necessary between communicating tasks when the task terminates to ensure the connection is no longer needed before tearing it down.

Because only one task may be actively executing on a cell at a time, tasks with overlapping cell allocations may not execute concurrently. Therefore, concurrent tasks that need to communicate with each other must be mapped onto the machine such that their allocations do not overlap. Conversely, if a task wishes to invoke a child task, the child must lie entirely within the allocation of the parent task. If a task wishes to invoke two communicating child tasks, both must lie within the parent's allocation without overlapping (See Figure 2.3).

Child tasks have limited external communication options: they may have external connections between themselves and other (non-overlapping) child tasks invoked from the same parent, or they may communicate with tasks external to the parent's allocation via external connections inherited from the parent. Child modules may not create new external connections extending outside the parent’s cell allocation; this restriction is necessary to keep the encapsulation “pure”. The parent module presents a particular interface to the application. If an invoked child were to “reach out” of the parent’s allocation without the parent’s express knowledge, the parent module’s interface would no longer be sufficient: a calling task (or application) would need to know about both the parent and the child. Because knowledge of the parent’s interface alone would no longer be sufficient, the parent’s ability to encapsulate communication complexity would be lost. By allowing child tasks to inherit a parent's external connections, complicated multi-stage tasks can be assembled from a collection of simpler tasks, while concealing the internal complexity from the calling task or application. For example, in Figure 2.3, **Pack Pixels** is shown inheriting the external connection from **Video In** to **Tee-Off**.

A parent task may communicate with its child only via parameters and pointers; there is no concept of a connection between a parent and child because parent execution suspends while the child task runs. Parent tasks may invoke children to an arbitrary depth, but recursion and reentrancy are expressly forbidden. The absolute depth of task invocation must be known at link time to ensure adequate communication resources can be available at runtime. If variable depth recursion were allowed, runtime resources could not be guaranteed at link time unless some arbitrary depth limit were pre-established. The depth limit approach is unacceptable because

(1) all scheduling would have to assume the worst-case depth limit, resulting in inefficient resource utilization, and
(2) some program would inevitably try to exceed the pre-established limit at runtime and crash, violating our guaranteed predictability.

Thus, to ensure predictability and allow efficient resource allocation, the absolute depth of task invocation must be known at link time.

2.4 Utilizing reconfigurable connections

All communication within and between tasks occurs via unidirectional **connections**. A connection is a long-lived bandwidth reservation between a source port on a source cell and a destination port on a destination cell. Data put into the source port is guaranteed to be available at the destination port within a time interval determined by the connection's level of service. A **port** is a software construct belonging to the task which makes the connection (which is really just a bandwidth reservation) accessible to the program code. While it is realized by specific hardware resources belonging to the cell, it is managed as an entity belonging to the task. A connection can be thought of as a pipe connecting two cells; the ports are the openings of the pipe. Data poured into the uphill end of the pipe flows out the downhill end.

All communication between cells within a task occurs via **internal connections**, defined by a source cell, source port name (needed by the source cell code), destination cell, destination port name (needed by the destination cell code), and an optional bandwidth reservation. Connections used together are grouped by the application developer (or a higher-level compiler) into **networks**. Task-local communication phases, called **netgroups**, are defined by grouping networks together. A connection may only belong to a single network, but a network (and hence its connections) may belong to several netgroups (Figure 2.4). All aspects of internal connections (connections, networks, and netgroups) are entirely contained within the task definition. Only one netgroup may be active within a task at a given time.

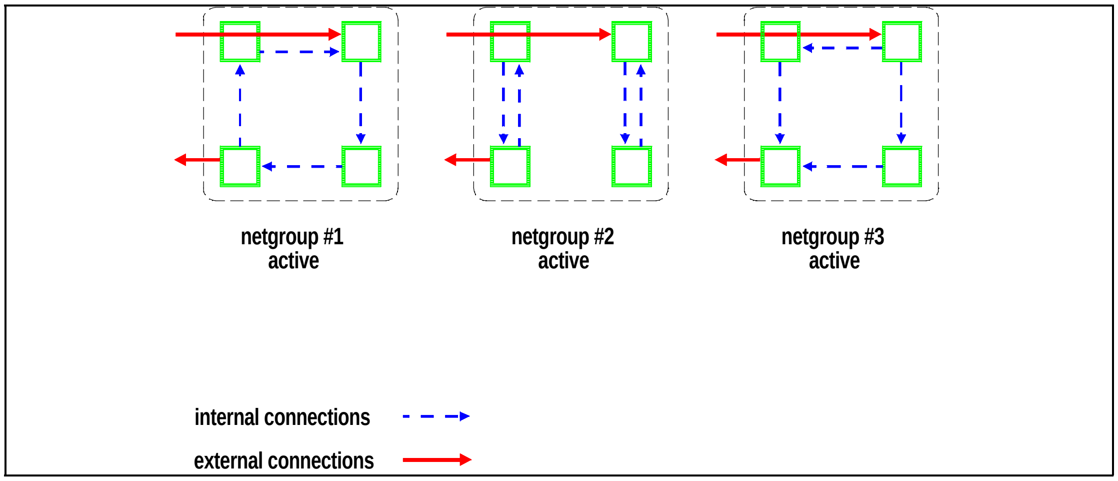


Figure 2.4 **Netgroups** allow finite physical connection resources to support multiple local communication phases.

Communication between tasks occurs through **external connections**, which join **external ports** on each task. External ports may either be defined as part of the task, or may be passed in to a child task from a parent. Because external connections are not wholly owned by the task (the task only owns one of the external ports, and cannot specify bandwidth), external connections need to be defined by a higher-level (parent) task, or at the application level.

If a task has external ports, a barrier synchronization is required at the beginning of task execution covering all cells belonging to both communicating tasks, ensuring that all cells of each task's allocations are ready. This operation is necessary to ensure no data is sent via an external port before the connection is established. Similarly, another barrier is required at task termination to ensure all communication stops have completed before reclaiming the external connection resources. Barrier synchronizations are also needed whenever a task changes the active netgroup, but requires only the participation of the task's cell allocation. No other cell, external controller, nor any other agent outside of the task's allocation is required to participate when changing the active netgroup. Connection reconfiguration within a task occurs purely under local control.

Note that all connections are defined by endpoints and bandwidth; no routing information is included as part of any connection definition. The mapping of connections to physical communication resources, including their routing on the target machine, is the linker’s concern, not the application designer’s.

2.5 Implementing applications

Applications exist as one or more communicating tasks executing on physical cells on a real machine. A TCS "program" isn't a single entity; it exists as a database containing the executable program code for each task for each cell, as well as the hardware-specific connection resource mappings for each cell. A TCS program is created by mapping the cell allocations of specific module instances to specific cells on a target machine, linking the program code of the tasks and their children for the individual target machine cells, routing the connections and assigning specific hardware resources to support those connections, evaluating what barrier synchronizations memberships are needed, and assigning the necessary resources, then creating the loadable images for code, synchronization, and communication. To run a TCS application, the program code, synchronization information, and communication information must be loaded onto all cells in the machine, then all cells can begin execution.

2.6 Chapter summary

This chapter introduced the TCS programming model. TCS applications are constructed from multicellular tasks which communicate by means of unidirectional connections. Internal task communication occurs via internal connections, which are grouped into networks, and networks are grouped into sets called netgroups. Only one netgroup may be active at a time; tasks may perform local communication context switches to change the set of active connections from one netgroup to another. External connections support communication between tasks and persist for the lifetime of the task. Task execution is controlled using a small set of primitives: task start, local communication context switch, and task end. These primitives are all built upon a fourth control primitive, barrier synchronization, which will be covered in more detail in Chapter 4.

Chapter 3 -

Target Platform Communication Mechanisms

The last chapter introduced the TCS machine model and the notion of TCS ***connections***. This chapter introduces iWarp[12], the target machine, and shows how TCS connections can be supported on this hardware. Two different message passing implementations, ***RTS message-passing*** and ***deposit message-passing***, are introduced for comparison, and the performance of TCS connections and message passing communication are characterized in isolation on an unloaded machine. While both message passing and TCS are shown to offer good performance and predictability for large transfers, TCS maintains a substantial performance advantage for small transfers.

3.1 Target machine overview

An iWarp array is the target platform used to validate the TCS model because it offers a rich set of communication hardware that allows fair comparisons of different communication models.

3.1.1 iWarp array

The target machine is composed of 64 processing cells arranged as an 8x8 torus, plus one host-interface cell (Figure 3.1). Each

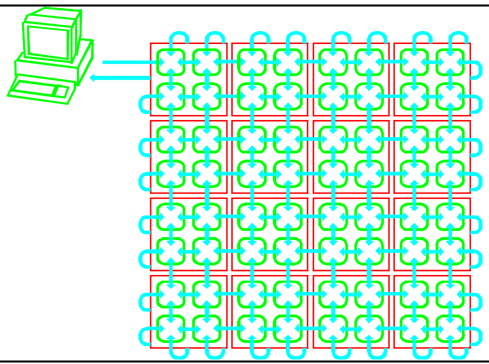


Figure 3.1 The iWarp array configuration - an 8x8 torus plus a host interface.

cell is composed of an iWarp chip (or ***iWarp component***) plus 512K static RAM. Each iWarp component contains a VLIW CPU (the ***computation agent***) and a network interface (the ***communication agent***).

3.1.2 iWarp Communication agent

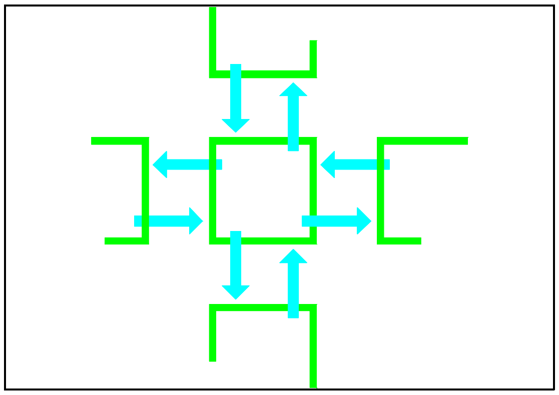


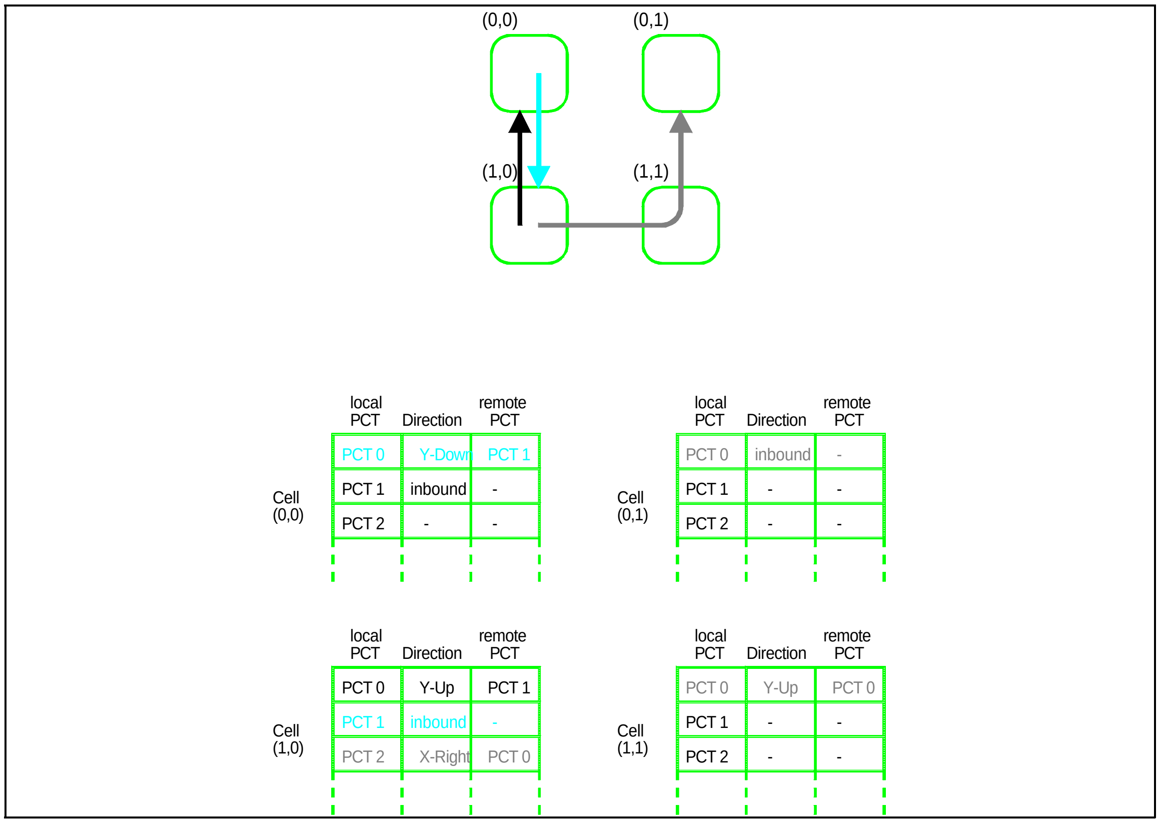
Figure 3.2 iWarp connectivity.

Each communication agent has eight external physical network connections, four in and four out. These are designated as X or Y, Up/Left or Down/Right, and In or Out. Each external network connection has a maximum bandwidth of 40 MB/sec (Figure 3.2).

Internally, the communication agent has 20 eight-word FIFOs known as ***PCT***s. Each PCT can be configured to receive data from an external physical network connection or from the computation agent, and each PCT can send data either to an external network connection or to the computation agent.

3.1.3 PCT-supported connections

Connections are built by chaining together PCTs on adjacent cells, building a contiguous path from source to destination (Figure 3.3). A connection consumes physical link bandwidth only if it is actively forwarding data. If two connections share the same physical link but only one is carrying data, the one carrying data gets full link bandwidth. If both connections are actively carrying data, each gets only half the link bandwidth, multiplexed between them on a word-by-word basis. For a given connectivity, congestion (and therefore available bandwidth) depends on both routing and connection activity. In Figure 3.4, both examples show a connection from each cell in the bottom row to the center cell in the top row. In the left-hand example, if all three



Cell (0,0)

local PCT	Direction	remote PCT
PCT 0	Y-Down	PCT 1
PCT 1	inbound	-
PCT 2	-	-

Cell (0,1)

local PCT	Direction	remote PCT
PCT 0	inbound	-
PCT 1	-	-
PCT 2	-	-

Cell (1,0)

local PCT	Direction	remote PCT
PCT 0	Y-Up	PCT 1
PCT 1	inbound	-
PCT 2	X-Right	PCT 0

Cell (1,1)

local PCT	Direction	remote PCT
PCT 0	Y-Up	PCT 0
PCT 1	-	-
PCT 2	-	-

Figure 3.3 Example PCT configuration illustrating how three connections could be supported via PCTs

connections are active at once, only one-third of the physical link bandwidth is available to each connection. In the right-hand example, the same source/destination connectivity is provided, but no congestion occurs - each connection is routed over a different physical network link.

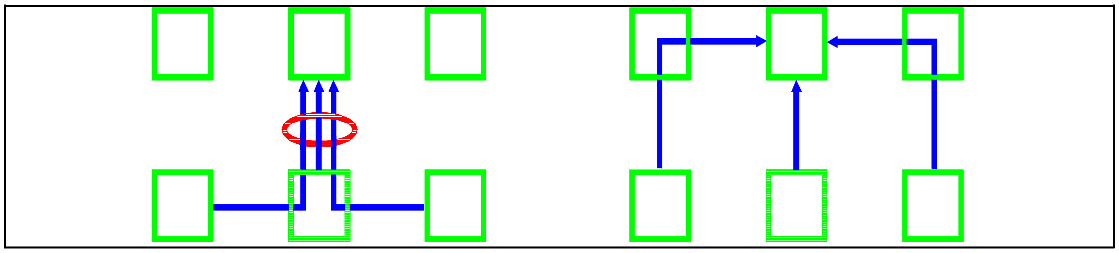


Figure 3.4 For a given connectivity, the routing affects the maximum bandwidth available.

The computation agent can read or write from connections by accessing the PCTs of the communication agent either through ***gates*** or ***spools***. A ***gate*** is a special register that can map an iWarp component’s PCT in the communication agent into the computation agent’s register file. CPU operations treat a gate like any other register, but reading a gate pulls data from the front of the mapped PCT’s FIFO. Writing to a gate appends data at the back of the PCT’s FIFO. Each iWarp component has two read-only gates and two write-only gates, which can be mapped to any of the twenty PCTs.

A ***spool*** is a hardware feature that provides DMA-like transfers between a block of memory and a PCT. Each active spool “steals” up to one-third of the computation agent’s CPU cycles, but requires no other direct CPU action once a transfer has been started.

Connections may be created (or destroyed) by one of two mechanisms:

(1) **Source routing**

Special ***tagged words*** may be launched at the connection’s source that automatically set the state of the communication agents as they pass through the array. PCT assignments dynamically occur as the communication agents forward the connection header along. The computation agent at the destination can be notified of an incoming connection either by polling or by an interrupt, depending on how it has configured its communication agent.

If a resource needed to complete a route is busy, the communication agent will block the connection until the resource becomes available.

(2) **Direct configuration**

As the name implies, with direct configuration the computation agent directly writes the state of the communication agent to set a specific PCT configuration. While source routing requires only computation agent participation at the source and destination of a connection, direct reconfiguration requires the active participation of computation engines along the entire path from source to destination. Furthermore, while the communication agent is responsible for PCT assignment/reclamation in the source routing approach, direct configuration requires all PCT assignments to be known prior to runtime. Direct configuration offers two potential performance advantages: the state of the communication agents along the route of a connection can all be configured together in parallel, and multiple connections can be configured in parallel. With source routing, as a connection header makes its way through the system, it must sequentially configure the state of the communication agent at each step of the way. With direct configuration, the state for the entire path can be configured at once. Furthermore, with source routing, a cell can only launch one connection header at a time. With direct configuration, an entire set of connections may be established simultaneously.

Because direct configuration requires the participation of cells other than just the connection source and destination, some form of ***barrier synchronization*** is needed whenever a connection state change is needed. For instance, in Figure 3.3, the connection from cell (1,0) to (0,1) passes through (1,1). Cell (1,1) needs to be certain the connection is no longer needed before it reclaims the PCT.

3.1.4 Physical communication schemes

PCT-based connections form the basis of all iWarp communication mechanisms, but how those connections are used yields three very different physical signaling mechanisms.

3.1.4.1 PCT-supported static (TCS) connections

TCS connections are implemented using the direct configuration approach but allow for PCT subsets to be configured; that is, a TCS module may only need to reconfigure PCTs 1 through 8, and will leave the remaining 12 (which may be supporting other connections or the runtime system) alone.

3.1.4.2 RTS message-passing

The iWarp runtime system, or ***RTS***, is a low-level system monitor that allows programs to be loaded and executed on array cells, provides proxy I/O service for the array cells, and allows a cell’s internal state to be examined or modified by the host. To provide these services, the RTS requires a communication system that provides connectivity to all cells with minimal use of cell resources. The RTS communication system is implemented as a general-purpose message-passing system built upon a unidirectional token-ring communication structure. Each cell forfeits two PCTs to the runtime system to build a large, single closed-loop connection that passes through all cells exactly once; this loop then supports a token-ring-like communication mechanism. At boot time, the host interface cell injects a token into this closed ring; the token circles endlessly until a cell requires RTS services. When a cell needs to send a message, it acquires the token, then injects its message into the ring. The message follows the ring until it reaches its destination, upon arrival it signals an interrupt at the destination cell, and the destination cell consumes and processes it. An acknowledgment is sent from the destination in a ringward direction until it reaches the message source. The source consumes the acknowledgment then re-injects the RTS token into the ring.

This communication mechanism, ***RTS message-passing***, is available to user programs and provides a simple means for any two arbitrary cells in the array to communicate. All communication requires circumnavigating the array at least once (the message travels partway around the ring; the acknowledgment completes the round-trip), generating interrupts at the source and destination cells (and consequently causing in program context swaps). Only one cell pair can use the ring at a time, therefore, the total bandwidth available through this mechanism is limited, especially impacting multiple short message transfers which could otherwise occur in parallel.

3.1.4.3 Deposit message-passing

Deposit message-passing is an iWarp communication library [42,43] providing message-passing services similar to RTS message-passing, but with vastly improved performance. Features include: multiple cells can send at once, cells can receive and send at the same time, and fewer copies and program context swaps are used when communicating. The sender specifies the address of the buffer to be used by the receiver. Deposit message-passing requires nine PCTs and two spools be dedicated to the message-passing system, but allows all cells to send and receive at once. Messages are implemented as source-routed dynamic connections. Only one message at a time is supported over a physical network link, but the message has the full link bandwidth available to it once it does go through. The PCTs used by a message are immediately deallocated as the message trailer passes through each of the communication agents along its route. Routing is calculated on-the-fly when a message is launched. Unlike RTS message-passing (or even Nx-based message-passing), deposit message-passing assumes a pre-allocated memory buffer at the destination so that protocol overhead is much reduced. This reduced processing overhead in turn results in a more efficient implementation.

In summation, three general communication options are available on the iWarp:

(1) static PCT-supported connections, which are routed prior to runtime and can last longer than just one message time,
(2) RTS message-passing, which provides a token-ring like communication system, and
(3) deposit message-passing, which uses source-routed dynamic connections and allows simultaneous sending and receiving by all cells at once.

TCS connections are a special form of static PCT-supported connections that allow small groups of cells to reconfigure independently of the rest of the array without disturbing existing connections passing through the cells.

3.1.5 Known system irregularities

While the iWarp is a good target platform, it has a few eccentricities that make accurate performance prediction difficult, but not impossible.

3.1.5.1 Network contention unfairness

In theory, multiple connections sharing a physical network connection share the bandwidth fairly. In reality, the on-chip pathway scheduler views PCTs as four groups of five PCTs each. Each pass through the scheduler (for each of the four outgoing physical connections) the scheduler looks at the four groups in a round-robin fashion, and chooses a PCT within that group in a round-robin manner. If a group has no PCTs with data to send, that group is skipped. Thus, scheduling is fair if all PCTs with data to be sent lie within one group, or if the same number of PCTs lies in each of the different groups. Otherwise, PCTs belonging to groups with a small number of active PCTs get a disproportionately higher percentage of bandwidth.

For example, consider three PCTs with data in group one, and one PCT with data in group two, all competing for the same outgoing pathway. The one PCT in group two would get one-half of the physical pathway bandwidth, and each of the three PCTs in group one would get one-sixth of the physical pathway bandwidth (rather than one-fourth as expected under a fair scheduling scheme).

3.1.5.2 DQ contention

Every PCT that receives a word from a network connection must return an acknowledgment word to the cell that sent the data. This acknowledgment word is called a ***DQ*** (short for “dequeued message acknowledgment”) and is carried on a special, physically separate link parallel to the data link but running in the reverse direction. In an ideal world, the DQ bandwidth would be the same as the forward link bandwidth. Unfortunately, under certain conditions, when multiple connections pass through a cell and at least one connection changes direction in the cell (for example, the message had been going up but turned left at the cell), congestion occurs within the cell’s DQ-processing hardware, and DQs are forwarded in an unfair manner. This amount of congestion can be predicted, and if the forward links are fed no faster than the congested rate (by intentionally sending data at a reduced rate), forward pathway bandwidth is shared fairly (within the constraints of Section 3.5.1). If one tries to feed the forward pathways faster than the DQ congestion-limited rate, the DQ signals are returned in an unpredictable manner, and forward data flow is choked by the lack of DQ signals showing available buffer space.

3.1.5.3 Uneven forward link bandwidth

Theoretically, the iWarp is supposed to deliver 40Mbytes/sec on each pathway. In reality, the scheduler tends to “skip” sending a word every thousand words or so, yielding a true bandwidth closer to 39.96Mbytes/sec.

3.1.6 iWarp platform summary

While the communication hardware has a few anomalies, they are known and can be accounted for in performance models that maintain detailed knowledge of the underlying PCT assignments.

Three general communication methods are available: PCT-supported connections, RTS message-passing, and deposit message-passing. While the PCT-supported connections require resource allocation prior to runtime, both message-passing schemes handle communication resource allocation on-the-fly. The RTS message-passing has the lowest resource requirements and, given its token-ring-like nature, the lowest expected performance.

Because the iWarp cells use static RAM for main memory, computation performance can be accurately predicted. Figures 3.5 and 3.6 show communication times, both predicted and measured, for short and long transfers using simple point-to-point PCT-based connections, demonstrating that communication performance (at the lowest level) is both predictable and repeatable.

3.2 Measured iWarp communication performance

The iWarp architecture provides three general communication schemes: PCT-based connections, RTS message-passing, and deposit message passing. This section quantitatively measures the performance of these communication schemes for varying quantities of data and varying distances. Times are measured in “clock cycles”. Each iWarp component has an on-chip, program-accessible clock/counter. While the iWarp runs at a 20MHz system clock rate, the counter runs at only one-eighth of the processor clock rate. Eight system clock ticks occur for every counter clock tick. Counter ticks are multiplied by 8 to yield the number of system clock cycles. Thus, while times are reported in “system clock ticks,” the actual resolution is only to every eighth clock tick.

3.2.1 PCT-based connection communication

Figures 3.5 and 3.6 show the measured times for single-word exchanges on the iWarp for distances ranging from 1 to 7 cell-widths. Times are measured by taking the round-trip exchange time (cell A sends a word to B, B receives the word then sends a word back to A, cell A receives it) and dividing by 2. Figure 3.5 shows that single-word exchanges have a repeatability well within the measurement error of the timer, and Figure 3.6 shows that runs of 8-word exchanges have a time-per-exchange that is repeatable to within a single clock.

	1 cell	2 cells	3 cells	4 cells	5 cells	6 cells	7 cells
Avg time	12	19	22	28	32	38	42
max time	12	20	24	28	36	40	44
min time	12	16	20	28	32	36	40

Figure 3.5 - PCT-supported-connection single-word communication time (in clocks), average, maximum, and minimum times vs. distance, for 1000 single-word sequential exchange runs. Max measured error is 4 clocks.

	1 cell	2 cells	3 cells	4 cells	5 cells	6 cells	7 cells
Avg time	10	16	20	26	30	36	40
max time	10	16	20	26	30	36	40
min time	10	16	20	26	30	36	40

Figure 3.6 - PCT-supported-connection single-word communication time (in clocks), average, maximum, and minimum times vs. distance, for 1000 eight-word exchanges. Max measured error less than 1 clock.

These measurements (particularly Figure 3.5) demonstrate that communication latencies within a real machine are neither uniform nor constant. Notice that in Figure 3.5 the communication time increments by 6 then 4 then 6 then 4 etc. This variation is due to the physical construction of the iWarp; cells are grouped four to a board. Cell-to-cell communication within a board incurs a latency of 4 clocks/cell, whereas communication between two cells on adjacent boards incurs a 6 clocks/cell latency. Furthermore, communication that “turns a corner” at a cell (such as transitions from left-to-right travel to up-to-down travel) incurs an additional 1 clock penalty. Assuming a 5 clocks/cell communication latency is a reasonable approximation that simplifies the modeling.

Connection communication cost can be modeled as having:

(1) a fixed set-up cost for sending,
(2) a per-word transfer cost which is a function of available network bandwidth (depends on runtime link usage, but known at link time),
(3) a distance-dependent network-latency cost, and
(4) a fixed set-up cost for receiving.

Connection_xfer_time = **(Send_Overhead + Recv_Overhead) +**
(Msg_size / Network_BW) + (Dist x cost_per_cell_hop)

This simple model allows comparisons between predicted vs. measured communication using PCT-supported connections for multiple-word exchanges. The following tables (Figures 3.7 and 3.8) show varying predicted and measured (avg, max, and min) exchange times for payloads ranging from four bytes to 16 Kbytes over distances of one to seven cells.

Both single and multi-word exchanges can be measured. Even for exchanges as large as 16 Kbytes, communication performance on the unloaded network is both extremely repeatable and predictable (to within a microsecond). Figure 3.7 shows the results of 1000 “short bursts” of communication; Figure 3.8 shows the results of longer bursts. As can be seen, even the longer bursts maintain predictability within half a microsecond, which is better than one percent. The iWarp connection hardware’s high degree of predictability is key to obtaining fast, predictable barrier performance, which enables construction of the other TCS control modules. As will be shown shortly, while certain kinds of message passing can maintain predictability on an unloaded machine, the TCS connections will maintain predictability even on a heavily loaded machine. Certain simplifying approximations at the task level of modeling will degrade the predictability somewhat from the degree shown in Figure 3.8; still, predictability within three percent or better can be expected.

message distance (cells)

bytes	1 cell	2 cells	3 cells	4 cells	5 cells	6 cells	7 cells
4	43	49	53	59	63	69	73
(predict)	(43)	(48)	(53)	(58)	(63)	(68)	(73)
max	43	60	64	68	72	80	84
min	42	48	52	56	60	68	72
16	49	55	59	65	69	75	79
(predict)	(49)	(54)	(59)	(64)	(69)	(74)	(79)
max	60	64	68	76	80	88	88
min	48	52	56	64	68	72	76
64	73	79	83	89	93	99	103
(predict)	(73)	(78)	(83)	(88)	(93)	(98)	(103)
max	84	88	92	100	104	108	116
min	72	76	80	88	92	96	100
256	169	175	179	185	189	195	199
(predict)	(169)	(174)	(179)	(184)	(189)	(194)	(199)
max	180	188	188	196	196	204	208
min	168	172	176	184	188	192	196
1024	553	559	563	569	573	579	583
(predict)	(553)	(558)	(563)	(568)	(573)	(578)	(583)
max	564	572	576	580	584	588	596
min	552	556	560	568	572	576	580
4096	2089	2095	2099	2105	2109	2115	2119
(predict)	(2091)	(2096)	(2101)	(2106)	(2111)	(2116)	(2121)
max	2100	2104	2112	2116	2120	2124	2132
min	2088	2092	2096	2104	2108	2112	2116
16384	8233	8239	8243	8249	8253	8259	8263
(predict)	(8241)	(8246)	(8251)	(8256)	(8261)	(8266)	(8271)
max	8240	8252	8252	8260	8264	8268	8272
min	8232	8236	8240	8248	8252	8256	8260

Figure 3.7 PCT-supported connection communication time (in clocks), average, predicted, maximum, minimum, vs. size and distance, 1000 **single**-ping runs (error +/- 4 clocks/measurement)

message distance (cells)

bytes	1 cell	2 cells	3 cells	4 cells	5 cells	6 cells	7 cells
4	42	48	52	58	62	68	72
(predict)	(43)	(48)	(53)	(58)	(63)	(68)	(73)
max	42	48	52	58	62	68	72
min	42	48	52	58	62	68	72
16	48	54	58	64	68	74	78
(predict)	(49)	(54)	(59)	(64)	(69)	(74)	(79)
max	48	54	58	64	68	74	78
min	48	54	58	64	68	74	78
64	72	78	82	88	92	98	102
(predict)	(73)	(78)	(83)	(88)	(93)	(98)	(103)
max	72	78	82	88	92	98	102
min	72	78	82	88	92	98	102
256	168	174	178	184	188	194	198
(predict)	(169)	(174)	(179)	(184)	(189)	(194)	(199)
max	168	174	178	184	188	194	198
min	168	174	178	184	188	194	198
1024	552	558	562	568	572	578	582
(predict)	(553)	(558)	(563)	(568)	(573)	(578)	(583)
max	552	558	562	568	572	578	582
min	552	558	562	568	572	578	582
4096	2088	2094	2098	2104	2108	2114	2118
(predict)	(2091)	(2096)	(2101)	(2106)	(2111)	(2116)	(2121)
max	2088	2094	2098	2104	2108	2114	2118
min	2088	2094	2098	2104	2108	2114	2118
16384	8232	8238	8242	8248	8252	8258	8262
(predict)	(8241)	(8246)	(8251)	(8256)	(8261)	(8266)	(8271)
max	8232	8238	8242	8248	8252	8258	8262
min	8232	8238	8242	8248	8252	8258	8262

Figure 3.8 PCT-supported connection commuication time (in clocks), average, predicted, maximum, minimum, vs. size and distance, 1000 **eight**-ping runs, (error < 1 clock)

3.2.2 RTS Message-passing communication

RTS message passing is the low-resource-overhead communication mechanism provided by the runtime system. It enables any two arbitrary cells to communicate over a token-ring-like network constructed from just two PCTs per cell. While its token-ring nature serializes all communication and hence makes it undesirable for high-bandwidth parallel communication, it is worth examining as a model for a collision-free network with serialized access. The following table (Figure 3.9) shows measured performance for messages of varying size.

	message distance (cells)						
bytes	1 cell	2 cells	3 cells	4 cells	5 cells	6 cells	7 cells
32	27964	28200	27966	28200	27966	28198	28839
max	28180	28204	28180	28204	28180	28200	28840
min	27884	27600	27884	27600	27884	27600	28540
64	27980	28215	27982	28215	27982	28217	28855
max	28196	28216	28196	28216	28196	28220	28856
min	27664	27904	27904	27904	27904	27900	28844
256	28076	28311	28078	28311	28078	28313	28951
max	28292	28312	28292	28312	28292	28316	28952
min	27996	28016	28000	27992	28000	28000	28940
1024	28460	28695	28462	28695	28462	28695	29337
max	28676	28696	28676	28696	28676	28696	29340
min	28380	28384	28384	28384	28384	28380	29040
4096	29996	30231	29998	30233	29998	30231	30873
max	30212	30232	30212	30236	30212	30232	30876
min	29680	29920	29920	29920	29920	29916	30576
16384	36140	36375	36142	36375	36142	36375	37017
max	36356	36376	36356	36376	36356	36376	37020
min	36060	36064	36064	36060	36064	36064	36752

Figure 3.9 - measured RTS message-passing time (in clocks), average, maximum, and minimum vs. payload size and distance for 1000 single-ping runs

Given that RTS message-passing requires multiple context switches and utilizes code not accessible to the programmer, one cannot easily predict communication performance for this communication mechanism. Thus, Figure 3.9 merely reflects the measured performance. Note that on an unloaded machine, RTS message passing also shows repeatable performance to within one-half to one-percent, but its absolute average communication time is roughly 400 times slower than PCT-supported connections for small messages (64 bytes) and 4 times slower for large messages (16 Kilobytes). As multiple cells exchange messages, the token-ring-like nature of this communication mechanism will lead to greater performance variance as cells vie for the limited ring bandwidth.

3.2.3 Deposit message-passing

Deposit message-passing[22] uses a foreground-send/background-receive communication model; thus all communication incurs the cost of a program context switch on every receive. As with PCT-based connections, communication costs can be modeled rather simply. Costs include:

(1) a fixed set-up cost for sending;

(2) a per-word transfer cost which is a function of available network bandwidth (depends on physical link usage at runtime);

(3) a distance-dependent network-latency cost;

(4) a fixed set-up cost for receiving. Depending on the communication pattern, this set-up at the receiving cell may be done in parallel with the sending cell's set-up, and the cost may be "hidden" in the overlap. Patterns which require simultaneous sending and receiving cannot hide the receive cost.

Putting these together, the message passing transfer time is modeled as follows:

Mp_xfer_time = (Send_overhead + Recv_overhead) + (Msg_Size/Network_BW) + (Dist x cost_per_cell_hop)

While this looks similar to the formula expressed in Section 3.2.1, the difference is that the Network_BW is <u>not</u> known at link time, and in fact is resolved on-the-fly at runtime. One group of messages may block another, temporarily reducing available bandwidth to zero. The tables shown in Figures 3.10 and 3.11 list the transfer times predicted by the above equation, as well as the measured average, maximum, and minimum times for 1000 individual runs. On an unloaded machine (as used for this set of measurements) 100% of the bandwidth is available for each measurement as only a single message is flowing at a time. Thus, one expects (and sees) good repeatability, to the extent that the 4/6 cell latency differences (due to physical board crossings) is visible in the measurements.

message distance (cells)

bytes	1 cell	2 cells	3 cells	4 cells	5 cells	6 cells	7 cells
4	258	264	269	275	278	283	289
(predict)	(260)	(265)	(270)	(275)	(280)	(285)	(290)
max	260	268	272	280	280	288	292
min	256	260	268	272	276	280	288
16	324	328	336	340	344	348	356
(predict)	(372)	(377)	(382)	(387)	(392)	(397)	(402)
max	324	328	336	340	344	352	356
min	324	328	336	340	344	348	356
64	350	355	360	364	369	375	378
(predict)	(396)	(401)	(406)	(411)	(416)	(421)	(426)
max	352	360	364	384	372	380	400
min	348	352	360	360	364	368	372
256	448	451	456	465	465	468	479
(predict)	(492)	(497)	(502)	(507)	(512)	(517)	(522)
max	448	452	456	468	476	476	484
min	448	448	456	460	464	468	472
1024	833	835	843	848	849	856	860
(predict)	(876)	(881)	(886)	(891)	(896)	(901)	(906)
max	836	840	844	880	880	896	876
min	832	832	840	848	848	856	860
4096	2364	2371	2376	2385	2385	2390	2400
(predict)	(2414)	(2419)	(2424)	(2429)	(2434)	(2439)	(2444)
max	2364	2372	2376	2388	2392	2400	2420
min	2364	2368	2376	2380	2384	2388	2400
16384	8510	8515	8520	8524	8529	8536	8540
(predict)	(8564)	(8569)	(8574)	(8579)	(8584)	(8589)	(8594)
max	8516	8516	8520	8540	8536	8544	8548
min	8508	8512	8520	8524	8528	8536	8540

Figure 3.10 Deposit message-passing time (in clocks), average, predicted, maximum, and minimum, vs. payload size and distance for 1000 **single**-ping runs

Despite requiring a program context-switch for receiving, performance is still predictable to the same percentage (one percent or better) as PCT-supported connections. While performance is worse than connections for small messages (roughly a factor of six on 64-byte transfers), for large messages (16 Kbytes) the performance difference between deposit message-

message distance (cells)

bytes	1 cell	2 cells	3 cells	4 cells	5 cells	6 cells	7 cells
4	255	262	265	271	276	282	285
(predict)	(260)	(265)	(270)	(275)	(280)	(285)	(290)
max	256	262	266	271	276	282	285
min	255	262	265	271	276	282	285
16	371	376	379	385	390	395	399
(predict)	(372)	(377)	(382)	(387)	(392)	(397)	(402)
max	371	376	379	386	391	395	399
min	371	376	379	385	390	395	399
64	399	402	409	412	416	422	426
(predict)	(396)	(401)	(406)	(411)	(416)	(421)	(426)
max	399	402	409	412	417	422	426
min	399	402	409	412	416	422	426
256	492	500	503	508	514	519	522
(predict)	(492)	(497)	(502)	(507)	(512)	(517)	(522)
max	492	500	503	508	516	519	522
min	492	500	503	508	514	519	522
1024	880	885	894	896	899	907	910
(predict)	(876)	(881)	(886)	(891)	(896)	(901)	(906)
max	881	885	894	897	900	909	911
min	880	885	894	896	899	907	910
4096	2412	2420	2423	2428	2434	2439	2442
(predict)	(2414)	(2419)	(2424)	(2429)	(2434)	(2439)	(2444)
max	2412	2420	2423	2428	2434	2439	2444
min	2412	2420	2423	2428	2434	2439	2442
16384	8560	8565	8574	8576	8579	8586	8590
(predict)	(8564)	(8569)	(8574)	(8579)	(8584)	(8589)	(8594)
max	8560	8565	8574	8577	8580	8587	8592
min	8660	8565	8573	8576	8579	8586	8590

Figure 3.11 Deposit message-passing time (in clocks), average, predicted, maximum, and minimum, vs. payload size and distance for 1000 **sixteen**-ping runs

passing and connections is under five percent. On an unloaded system, deposit message-passing is between four and one hundred times faster than RTS message-passing.

3.3 Chapter summary

For simple cell-to-cell transfers on an unloaded system, both deposit message-passing and PCT-supported connections offer fast, predictable performance. Relative to message passing, connections offer better performance with small transfers (only a few words per exchange). For large transfers, the small amount of additional overhead incurred by deposit message passing is swamped by the (**Msg_size / Network_BW)** factor, and the two offer comparable performance. Because RTS message-passing requires multiple context swaps and always requires passage through all cells in the system (both of which increase latency regardless of message size), it consistently offers the worst performance, and offers repeatable performance only on an unloaded system. As network load increases, RTS ring access is granted on a “first-come, first-serve” policy regardless of cell bandwidth usage.

For building fast, efficient parallel applications on iWarp, the choices are thus limited to either deposit message-passing or PCT-supported connections. TCS uses PCT-supported connections for communication. Not only were connections shown to be the fastest of the three available communication methods, they were also shown to offer predictable performance (better than one percent, which is less than a microsecond) on an unloaded machine. If the network bandwidth can be known/controlled, PCT-supported connections should be predictable even on a heavily loaded system.

Chapter 4 -
Barrier Synchronization

4.1 Introduction

Fast barrier synchronization is vital for good TCS performance. TCS tasks rely on ***local communication context switches*** to swap between sets of connections; each communication context switch requires three barriers. Barriers are also needed when child tasks are started or ended. Because barriers play this vital role in resource allocation, they must have predictable performance (otherwise tasks using them would not be predictable), and they need to be fast (since they are used so frequently). This chapter examines some of the major issues regarding barrier synchronization and explores the trade-offs made when implementing a barrier synchronization scheme.

4.1.1 What *is* a barrier, and what does it do?

A ***synchronization barrier***, or ***barrier***, is a named rendezvous point in parallel program code which is shared across multiple cells. When a barrier is encountered in a cell's program, the cell's foreground execution is suspended until all other cells sharing that barrier (***barrier members***) reach the same named barrier in their code as well. ***Barrier execution time*** is defined as the time elapsed from when the last cell reaches the barrier, until the last cell exits the barrier, assuming that all members of the barrier arrive at the same time and that no background processing occurs. While barrier synchronization is a necessary tool for sharing resources in a parallel environment, it doesn't accomplish any productive work. Good application performance requires minimizing the impact of barrier synchronization on total application execution time.

Until recently, most parallel applications only needed a barrier or two at startup and termination; since the time spent synchronizing was small relative to the total application execution time, barrier execution speed wasn't relevant to overall application performance. More recently, though, applications have been developed that require thousands of barriers for connection reconfigurations, or that rely on barrier synchronization to control congestion in a message-passing system[25]. As barrier use increases, barrier execution speed plays a larger role in overall application performance.

4.1.2 Barrier properties

Regardless of the underlying implementation, a barrier has a name and a cell membership. The barrier name uniquely identifies the group of cells (barrier members) who participate in the barrier. Cells may be members of more than one barrier, and a parallel computer may support multiple barriers executing at once. Two barriers with non-overlapping memberships may execute simultaneously; barriers which share members (***overlapping memberships***) will either be forced into sequential execution or deadlock, depending on the ordering of the barriers in the overlap cells' programs. A cell may only participate in a single barrier at one time. (If a cell supports multitasking, each job on the physical cell will be treated as a single task running on a separate "logical cell." While the physical cell may be executing multiple barriers at once, each logical cell is still limited to executing a single barrier at a time.)

4.2 Issues affecting barrier synchronization implementations

This section will first present the canonical barrier implementation, then look at some of the issues that must be dealt with when creating a real implementation.

4.2.1 The canonical barrier implementation

As cells that are members of barrier ***foo*** reach the barrier in their code, they suspend their foreground execution until all other members reach the barrier point as well. Members resume foreground execution as they become aware that all other members have arrived. In essence, barrier execution is an all-to-all information exchange between the members ("I am here at ***foo***") that must occur before execution resumes. ***How*** that information exchange occurs is a function of the ***physical signaling scheme*** and the ***messaging protocol***, and will be addressed later. What's important now is recognizing that barrier execution time is essentially the time needed for the barrier members to complete an all-to-all information exchange.

The all-to-all communication occurs via an exchange of messages. A barrier implementation's performance, the impact of how messages are encoded, signaled, and distributed, can be modeled with just two parameters: the ***message time***, and the ***number of parallel message times***. The ***message time*** is the time needed for one cell to compose and send a message to another, and for the recipient to receive and decode it. Message time encompasses both the software overhead of creating/launching the message and receiving/decoding it, as well as the network latency in delivery. The ***number of parallel message times*** reflects the number of messages which need to be sent in the barrier and the degree to which multiple messages may be sent simultaneously. Assuming no other load on the synchronization network, barrier execution time is just the product of the effective message time and the effective number of parallel message times.

4.2.2 Scalability of a barrier implementation

Scalability characterizes how quickly an implementation's performance changes as the number of cells increases (in other words, how quickly does it slow down as N becomes large?). Depending on one's definitions of "performance" and "number of cells", scalability can have (at least) four different meanings:

(1) barrier execution time vs. number of barrier members

(2) barrier execution time vs. number of cells in the machine

(3) barrier execution time vs. number of simultaneous barriers supported

(4) number of simultaneous barriers supported vs. number of cells in the machine

4.2.2.1 Barrier execution time vs. number of barrier members

As shown earlier, barrier execution time equals the product of the implementation's ***message time*** and the ***number of parallel message***

times needed per barrier. Barrier execution time depends on the scalability of both message time **and** the number of parallel message times needed per barrier as the number of barrier members increases. Just looking at the number of messages needed vs. number of barrier members is a useless (though oft cited) means of predicting performance because it ignores the possibility of message times getting longer with more barrier members, as well as the speedup allowed by multiple cells sending in parallel.

4.2.2.2 Barrier execution time vs. number of cells in the machine

Again, the real concern is the scalability of ***message time*** and the ***number of parallel message times*** needed vs. the number of cells in the machine. Depending on the implementation, message time may scale with the number of cells in the machine, the number of barrier members, or both.

4.2.2.3 Barrier execution time vs. number of simultaneous barriers

The ***number of parallel message times*** needed to execute a barrier is independent of the number of other barriers executing; it is a measure of the communication overlap **allowed** by the implementation, not a guarantee. Since each cell (or logical cell) can only be executing one barrier at a time, it is unaware of the existence of other barriers which may be executing on the machine at the same time. The cell's pattern of message exchange (who sends what to whom in what order) will thus be the same for a given barrier regardless of whether or not other barriers are (or could be) executing, although individual messages may be blocked or delayed. Thus, only the ***message time*** can be affected as the number of simultaneous barriers on the machine increases. The scalability of barrier execution time vs. number of simultaneous barriers thus solely depends on the scalability of the message time vs. the number of simultaneous barriers. Depending on the implementation, the message time may scale based on the number of simultaneous barriers ***allowed for*** by the implementation (static sync network bandwidth allocation), or it may depend on the number of simultaneous barriers ***executing*** (dynamic sync network bandwidth allocation).

4.2.2.4 Number of simultaneous barriers vs. number of cells

This "performance measure" is more of a philosophical question than a stand-alone metric. Before answering "How hard is it to allow more simultaneous barriers as the number of cells increases?", one needs to ask "How many barrier channels are ***desirable*** for a multicomputer of N cells?". A related question is: "How many distinct simultaneous barriers can be supported at once by a pool of K barrier resource sets?" In other words, can two distinct barriers with non-overlapping memberships share the same underlying resources? The answer to this third question colors the answer to the first, and is highly implementation dependent, reflecting both ***how*** multiple barriers are implemented and the allowable barrier memberships supported.

4.2.3 Barrier message memory

When a barrier executes, the member cells perform an all-to-all information exchange; the details of how information is actually signaled and received is the responsibility of the implementation's physical signaling scheme and messaging protocol. "Barrier message memory" is an important characteristic of all implementations which ensures barrier synchronization information is not lost during the all-to-all information exchange. Two major cases need to be handled:

(1) If a cell arrives at a barrier in advance of the other barrier members, its information must not be lost, even though other barrier members aren't executing the barrier code yet. If the cell is re-entering a named barrier that it had previously completed, other members must not confuse the cell's second execution of the barrier with a continuation of the first execution of the barrier.

(2) The programming model states that when a barrier is encountered in a cell's program, the cell's foreground execution is suspended until all other barrier members reach the same barrier in their code as well. Real implementations have an additional constraint: a cell not only needs to know that all other members have arrived; it must also be certain that everyone else **will know** that **it** has arrived. Note that

the cell **does not** need to wait until everyone else **knows** that it has arrived; the cell merely has to know:

(1) that everyone else has arrived, and

(2) that everyone ***will know*** that ***it*** has arrived.

This is an important distinction, because forcing all other cells to ***know*** the cell has arrived requires in effect, executing an unnecessary second barrier.

All barrier synchronization implementations must include some form of barrier message memory that can satisfy these two cases. Lacking either one will cause unexplained program behaviour as barriers randomly fail to complete.

4.2.4 Barrier skew

Real cells running a real application seldom reach a barrier at the same time; rather than all reaching the barrier together, one cell will be first and another will be last. Consequently, the cells reaching the barrier first can start the computations necessary to process the barrier earlier than the one arriving last. ***Barrier skew*** reflects the difference in time between when the first cell exits the barrier and when the last cell exits the barrier. Barrier skew is generally, though not always, influenced by the timing skew between cells entering the barrier.

Since barrier performance will be no worse than the case of all cells entering the barrier together, barrier skew will be ignored in this paper. Still, it remains an interesting area for research on possible future optimizations.

4.3 Design Space for Barrier Implementations

Synchronization schemes can be roughly categorized by the services provided and how those services are implemented. Users only see the functionality offered by the synchronization services: flexibility for defining (and redefining) barrier memberships, the number of barriers supported on their target machine, and barrier execution time. Similar user services could be provided with very different implementations. Precise classification is complicated by the fact that sophisticated messaging protocols can emulate capabilities lacking in hardware (usually at the expense of speed or number of simultaneous barriers supported). Still, a particular implementation can be placed within a four-dimensional design space, and certain performance generalizations can be made from this categorization. The four not-quite-orthogonal axes describing the design space are:

(1) physical signaling scheme,
(2) messaging protocol,
(3) allowable barrier memberships,
(4) barrier capacity of the implementation.

Knowing where an existing implementation sits in this space offers clues to expected barrier performance and scalability. Conversely, given target platform hardware and membership/capacity requirements, these axes can help guide choice of an appropriate signaling scheme and messaging protocol. Initially, each axis will be looked at in isolation. Later, various combinations of physical signaling scheme and messaging protocol will be assembled to show how they interact in real barrier implementations.

4.3.1 Physical signaling scheme

Barrier synchronization requires an all-to-all information exchange between the barrier members. The physical signaling scheme determines ***how*** information is carried, and ***who*** a cell may directly communicate with. While the messaging protocol determines which cell talks to whom and when, the physical signaling scheme puts hard constraints on the messaging protocol by defining the allowed communication. The message time for a barrier is strongly dependent on the physical signaling scheme, ranging from a few clocks for a hardware broadcast/combining network to thousands of clocks for an implementation based on the machine's general message-passing facilities. Physical signaling methods generally fall into one of the following five classes:

Class S1 - low-level hardware broadcast/combining network (1's of clocks)

Class S2 - multiplexed hardware broadcast/combining network (10's of clocks)

Class S3 - special messages on a private network (100's of clocks)

Class S4 - special messages on the regular communication network (100s to 1000s of clocks)

Class S5- general messages on the regular communication network (1000s of clocks)

Lower-class signaling methods generally execute faster but can support only a limited number of simultaneous barriers. Capacity of lower-class signaling implementations can be increased at the cost of greater hardware complexity. Higher-class physical signaling methods more closely resemble a general-purpose network; increased capacity occurs at the expense of slowing signal times.

4.3.1.1 Class S1 physical signaling - Low-level hardware broadcast with network combining

Class S1 signaling is the simplest and most direct method of signaling, whereby a cell indicates its readiness by setting an absolute voltage level on a wire to indicate a logical “ready” or “not ready”. A simple logical combining of all the cells’ signals is done and the results continually broadcast back to all participants. Both the combining network and the broadcast network may either be buffered or unbuffered. Buffering allows greater scalability, but requires extra hardware and adds latency with each stage. The unbuffered implementation is faster for small implementations, but scales poorly with large numbers of cells.

4.3.1.1.1 Unbuffered broadcast

This scheme provides a true broadcast capability, and allows use of efficient messaging protocols that can complete in just one message time because all communication can be simultaneously overlapped. The simplest hardware broadcast network is the distributed-NOR circuit shown in Figure 4.1, similar to that discussed by Hwang and Shang in [29]. This is a NOR circuit, wired to all cells, such that any cell can pull the signal line low or let it float high. Only if all cells let the line float high will the test bit be read as high. The barrier is initialized with all participating cells pulling their barrier lines low, and either disconnecting all non-participating cells by

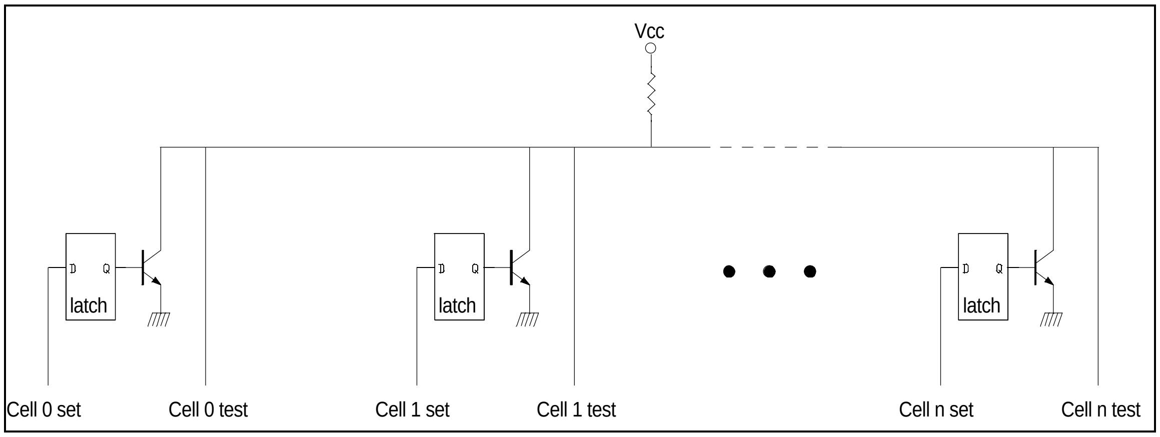


Figure 4.1 A single hardware barrier channel

partitioning the network (Figure 4.15), or having the non-participating cells let their lines float high (and subsequently ignore them). When participating cells enter the barrier region, they release their barrier lines and let them float high. When the last cell enters the barrier, all cells will then read the line as high. This barrier only requires one message time to complete; if all cells enter the barrier together, they will all release their barrier lines together, and could all probe their test bits together.

While conceptually simple, this barrier implementation has several drawbacks. While it only requires one message time to complete, that "one message time" scales linearly with the number of cells. Cells need to allow the voltage on the wire to stabilize after releasing the line before reading it; how fast the line stabilizes is a function of the line's capacitance and inductance, which is in turn a function of the line length, with is proportional to the number of cells wired together. For twice as many cells, one needs a wire twice as long to hook them up, and the settling time required also doubles. This implementation’s execution time is independent of the number of barriers supported; each barrier has its own copy of the circuit; adding more channels means replicating hardware but doesn't affect any single barrier's speed.

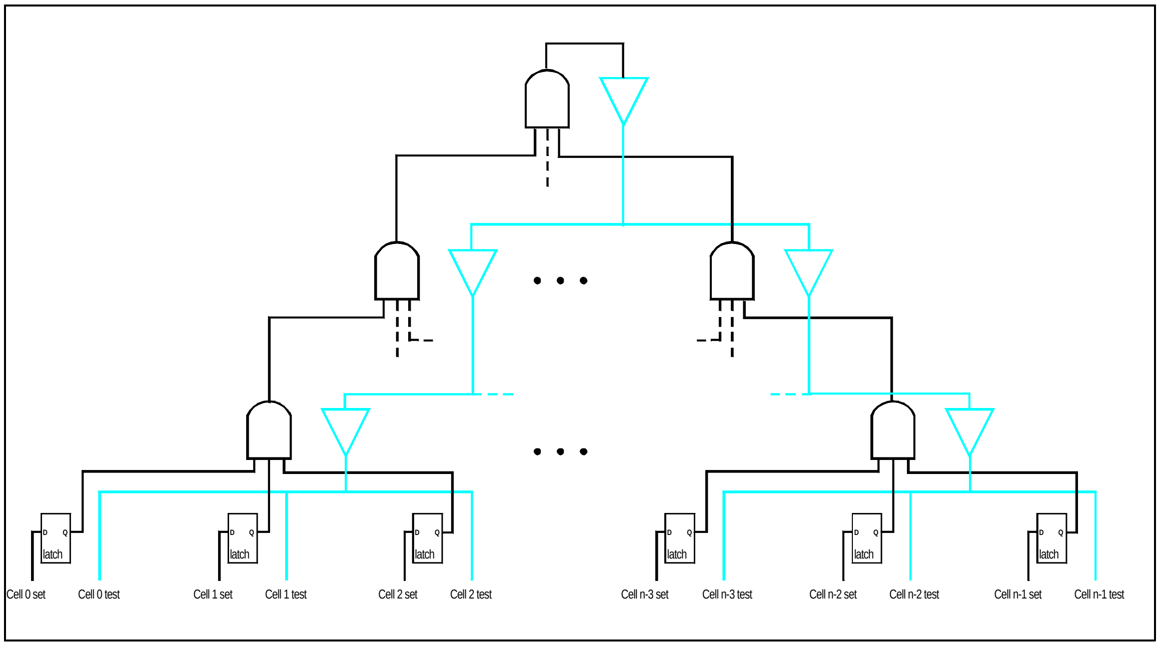


Figure 4.2 Single channel buffered hardware barrier circuit.

4.3.1.1.2 Buffered broadcast

Message time scaling can be reduced from linear to logarithmic by adding hardware "buffering" to the line. (Figure 4.2) This adds a constant delay per buffering stage, but by keeping the hookup wires short, the settling time remains constant. With a buffered system, message time only scales O(logN) because buffering latencies are a function of the number of layers (which are proportional to logN), but the settling time per layer remains constant. An unbuffered system scales as O(N) because the settling time needed is proportional to wire length. For a small multicomputer housed in a single cabinet, the unbuffered system offers simpler implementation, and the small size ensures a fast message time. For a system distributed across multiple cabinets, the buffered system offers a clear advantage. This is similar to an approach suggested by O’Keefe and Dietz in [37,38] and Beckmann et al in [7].

An insidious problem with this type of low-level hardware is supporting barrier message memory. Both the buffered and unbuffered networks shown can allow a cell to know that everyone else has entered the barrier, but, as shown, they cannot ensure that everyone else will know that the last cell has arrived. If the last cell arrives, lets the barrier signal line float high, sees it as high, and then resets itself, other cells doing background processing while waiting at the barrier may never see the line go high. Thus, some form of message memory must be added either at the message protocol level (in other words, use a second, uninterruptible barrier to ensure everyone's reached the first; because the second barrier is uninterruptible, everyone will complete within a known amount of time), or each channel's cell interface must be augmented with hardware to provide a message memory similar to that shown in Figure 4.3.

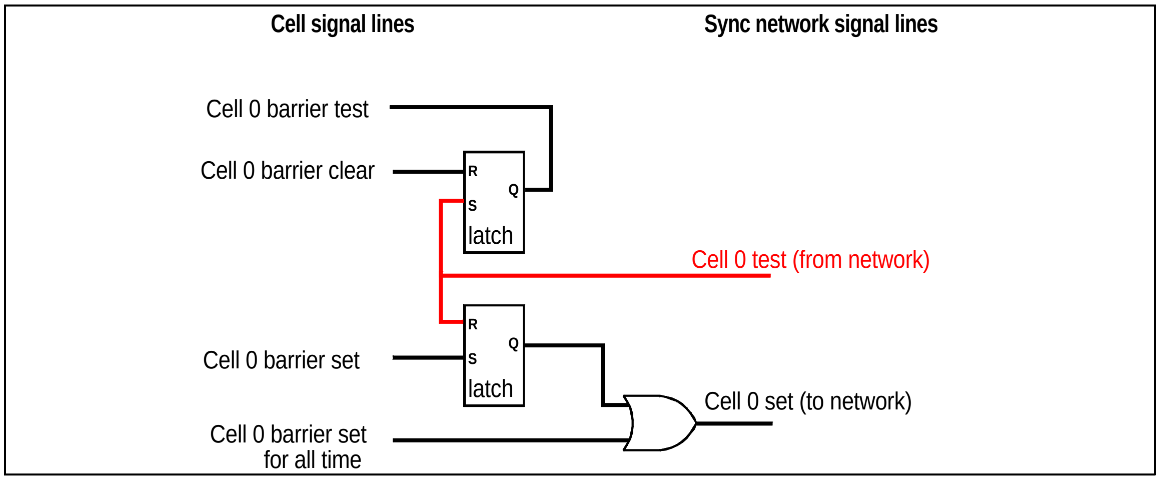


Figure 4.3 Modifying the cell’s interface to the synchronization network allows a more reliable barrier message memory.

4.3.1.2 Class S2 physical signaling - Multiplexed low-level hardware broadcast/combining

Both the buffered and unbuffered broadcast implementations allow fast information exchange, but the amount of hardware required scales with the number of simultaneous barriers needed. An alternate approach to this problem is to **time division multiplex** several barrier signals ("barrier channels") onto a single broadcast network. With this technique, message time becomes the sum of two components: a propagation/settling time (same as Class

S1) that scales with the number of processors, and a serialization latency that scales linearly with the number of barriers supported (each channel has to "wait its turn" to be broadcast and received). The settling time required between each "barrier channel" sets the reasonable limit on how many channels may be reasonably multiplexed over a single network. If more channels are required, additional copies of the synchronization network will be needed.

4.3.1.3 Class S3 physical signaling - special messages on a private network

Multiplexing several barriers on a single wire is little more than static bandwidth allocation to a fixed number of "barrier channels". A drawback of this approach is that barriers consume bandwidth whether they are executing or not, limiting the total number of barriers a machine may support. An application's barrier synchronization needs must be mapped to the fixed set of barrier channels, becoming a "graph coloring" problem similar to register allocation. An alternative is to make the synchronization network more general-purpose and use more sophisticated messages (as opposed to single bit ready/not ready) for barrier communication. While this approach will run about an order of magnitude slower than simple multiplexed bits on a wire (due to the greater overhead of processing/decoding the messages), it allows demand-driven allocation of sync network bandwidth to barriers as they execute, and, coupled with a more sophisticated messaging protocol, can allow use of simpler network hardware. Intelligent choice of a messaging protocol can relax the network connectivity requirements, resulting in simpler wiring; for example, explicit hardware broadcasting capabilities may not be necessary. Furthermore, by using a more general-purpose messaging scheme, greater numbers of simultaneous barriers may be supported merely by allocating additional cell memory to buffer barrier data.

By using a private synchronization network (as opposed to using the machine's general communication network), communication latency between cells is controllable. The designer has firm control over the message types and sizes which need to be supported, allowing for a more restrictive (and hence simpler) hardware/firmware network design. At the same time, the designer gains flexibility in terms of protocols allowed (as opposed to a simple ready-bit-on-a-wire), as well as the number of simultaneous barriers supported, at the expense of greater processing overhead and potentially longer communication latencies. The Alliant FX-8 is one example of a shared memory multicomputer that uses a separate, dedicated bus for barrier synchronization messages. Rather than tieing up shared memory bus bandwidth with spin-wait locks on shared variables, the designers elected to give barrier traffic its own private network[2].

Barrier message memory, particularly handling the early sync message, is cumbersome to address with hardware alone in a Class S3 or higher signaling scheme and needs greater involvement of the messaging protocol. An early sync message may:

(1) generate an interrupt at the recipient cell and be buffered for future use,

(2) block the network until the message can be serviced,

(3) continue circulating through the network until the message is actively received, or

(4) be discarded, relying on a higher software level to guarantee message arrival (via retransmission until successful).

Each of these options offers the developer a tradeoff between communication network bandwidth, application bandwidth needs, and any time-critical CPU demands the application may require.

Applications unfettered with real-time requirements do well with interrupt-driven buffering, since at most N-1 buffers are needed (that is, if every other cell in the array tries to execute a different barrier with the cell in question). On the other hand, applications with real-time requirements may not be able to afford servicing interrupts while in their critical loop. If one is doing only global synchronizations, blocking the synchronization network until the message can be serviced may yield the fastest and simplest implementation. Conversely, if multiple subset barriers are possible, blocking the network will almost certainly cause deadlock. Allowing the message to endlessly circulate consumes network bandwidth, perhaps slowing down "real" data transfers in progress (if synchronization shares the general purpose communication network). Discarding early messages complicates the necessary handshake protocol and can more than double the time needed to complete the barrier (since some cells will always be early).

4.3.1.4 Class S4 physical signaling - Special messages on a general network

If a private synchronization network is not available (such as when one is "retro-fitting" an existing machine with barrier capabilities), one can often "make do" utilizing special-purpose messages over the general network. Using "virtual channels" or similar capabilities, one can send barrier data over the general communication network but utilize private data handlers to launch and receive barrier information, resulting in lower barrier communication overhead than obtainable with the regular communication facilities. This approach still requires having low-level program access to the machine's communication hardware, but avoids the need for "programming in solder". Allowable messaging protocols are restricted by the connectivity allowed by the private data handlers. Depending on the machine, the private data handlers may sacrifice some of the connectivity normally offered by the regular communication facilities in exchange for lower communication latencies. This would force the use of a messaging protocol that requires a greater number of parallel messages, but may guarantee significantly shorter message times. Barrier message memory issues are the same as for Class S3. This general signaling approach is discussed in [10] but, as far as we know, was not implemented.

4.3.1.5 Class S5 physical signaling - General messages on a general network

This approach offers the most flexibility in choice of messaging protocols but permits the least control over message times or connectivity. Due to copying and buffering overhead, this approach usually has the slowest execution time but the greatest portability to other platforms. Furthermore, if the appropriate messaging protocol is used, the problems of barrier message memory can be transparently handled by the underlying communication services. One cannot avoid the use of interrupts, so one might as well let the regular communication buffers hold the early messages.

4.3.1.5 Physical signaling summary

Physical signaling implementations require a tradeoff between messaging speed and barrier capacity. At one end of the spectrum, the simple “bit-on-a-wire” scheme of Figure 4.1 provides fast signaling times but only supports one barrier per hardware instance. Time-division multiplexing multiple barrier signals on the wire will increase capacity at the expense of increasing message latency. At the other end of the spectrum is using the general purpose communication facilities to send and receive messages. This allows much greater flexibility for arbitrary barrier subsets and multiple simultaneous barriers on the machine at the expense of a much higher latency. Sharing the general-purpose network can be speeded up by using special, optimized private messages over the same physical network, at the expense of the type/shape of barrier subsets (memberships) supported (completely arbitrary vs. contiguous blocks only). While TCS uses only one physical signaling method, other real-world barrier implementations may use a hierarchy of signaling methods. Barriers that are used infrequently can utilize a slower signaling implementation that doesn’t require any specialized resources, whereas a few frequently-executed barriers within an inner program loop could benefit from rare, specialized hardware. As will be explored later in Section 4.5, choosing a physical signaling scheme requires a tradeoff between speed, capacity, and resource cost.

4.3.2 Messaging protocol

Given a group of N cells entering a barrier, none are allowed to exit until all have arrived. In essence, this boils down to each cell asking its N-1 neighbors "Are you ready yet?", and answering "Yes" when queried by its neighbors. Consider the two-cell

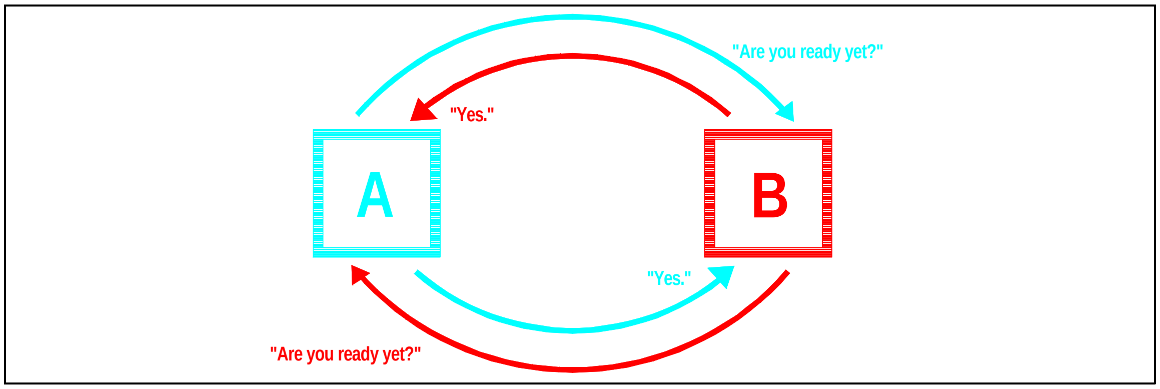


Figure 4.4 The simplest two-cell synchronization

synchronization shown in Figure 4.4. Once a cell has 1) Answered "Yes, I'm ready" to its neighbor's query, and 2) received a "Yes" to its own query, it is free to leave the barrier.

Obviously, this scheme carries some redundant communication. Assume ***A*** enters the barrier first and sends the "Are you ready yet?" message to ***B***. The message will sit at ***B*** until ***B*** enters the barrier as well (barrier message memory). ***B*** shouldn't need to ask ***A*** "Are you ready?" because ***A***'s message in ***B***'s buffer implicitly indicates readiness. Suppose that instead of a question/answer exchange, each processor simply sends the message "I'm ready, tell me when you are" to its neighbor. In effect, the message sent is "overloaded" so that a single message sent from one cell to another really carries the weight of two messages sent by two different cells. Modifying the handshake protocol in this manner reduces the synchronization to what is shown in Figure 4.5. How many message times does this barrier require to complete? If the synchronization network supports multiple simultaneous

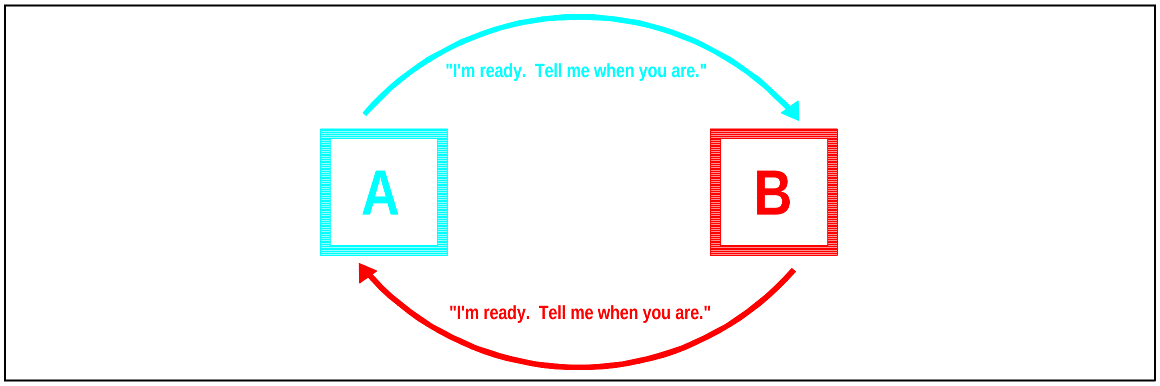

"I'm ready. Tell me when you are."
A
B
"I'm ready. Tell me when you are."

Figure 4.5 Reduced protocol two-cell sync

communications, the cells can overlap their message transmissions, and the barrier will complete in just one message time. On the other hand, if the network only supports a single message at a time (say, the two cells share an ethernet connection), the messages become serialized due to the network access restriction, and the barrier requires two message times to complete. In both cases, two messages were required to complete the barrier. The ***number of parallel message times***, though, varied based on the underlying physical signaling scheme. Choosing a protocol that minimizes redundant communication (whether directly, or because of serialization at the physical signaling layer) is key to good barrier performance.

The "ideal messaging protocol" for a given system is strongly influenced by the underlying signaling scheme being used. Major issues affecting the choice of protocol include degree of support for broadcast capabilities, and the number of barrier messages that can be sent simultaneously (both by the system as a whole, and by individual cells). Regardless of the protocol chosen, all cells need to transmit at least one message per barrier. By using a protocol that takes advantage of the underlying signaling mechanism, one can execute some of those communications in parallel, reducing the number of message times needed per barrier.

Messaging protocols fall into two general classes of communication: those that rely on some form of broadcast or multicast (Class P1), and those that use only private, point-to-point messaging (Class P2).

4.3.2.1 Class P1 messaging protocol- Broadcast communication

Synchronization protocols based on some form of broadcast messaging can be further subdivided into 4 subclasses:
(a) shared global channel, all-to-all, with network combining;
(b) shared global channel, one-to-all;
(c) private channel, all-to-all, with network combining;
(d) private channel, one-to-all.
Each of these will be examined in more detail.

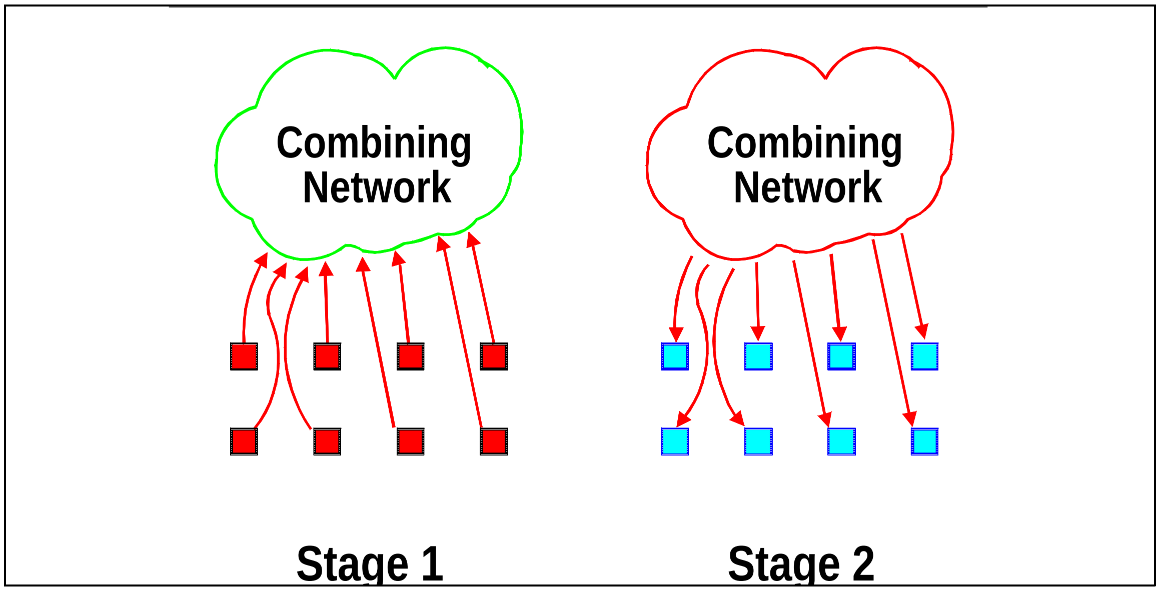


Figure 4.6 Class P1a protocol. Synchronization completes in one message time when the protocol and underlying hardware support a broadcast-combining network.

4.3.2.1.1 Class P1a messaging protocol -

Shared global channel, all-to-all, with network combining

This protocol assumes that the cells have a Class S1 or Class S2 physical signaling scheme, similar to what is shown in Figures 4.1 and 4.2. When executing a barrier, each cell sends a single message (usually just setting a single bit) to the combining network to indicate readiness, and waits until the network combining function indicates that the barrier is complete. Once the cell is assured that all other cells will know that it reached the barrier as well, the cell is free to leave. With the correct underlying hardware, this protocol can complete in one message time (if all cells enter the barrier together), because all cells can send and receive at once (Figure 4.6).

4.3.2.1.2 Class P1b -

Shared global channel, one-to-all, no network combining

This assumes an ethernet-like communication network, where one cell can broadcast and all other cells can listen (Figure 4.7). Unfortunately, this scheme offers the worst performance of the

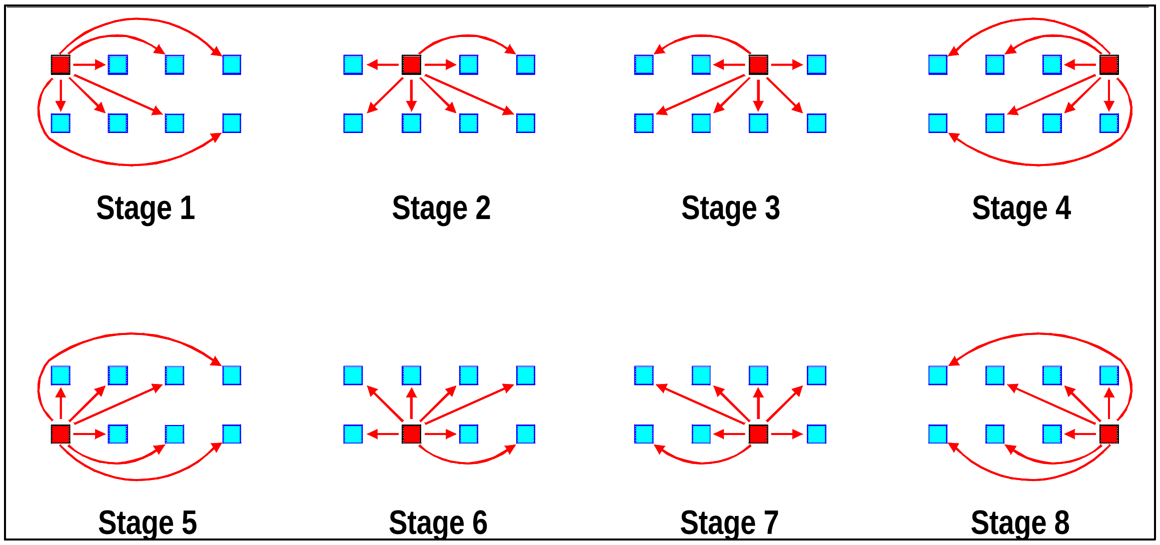


Figure 4.7 Class P1b protocol. Example of an eight-cell barrier executing using one-to-all broadcasts.

broadcast protocols, with best-case execution requiring N sequential message times to synchronize a barrier with N members. Because every cell participating in the barrier needs to send information at least once (otherwise no one can know it has reached the barrier), because there is only a single, shared global channel, and because there is no inherent combining in the network, all N broadcasts occur sequentially. There is no way to get parallel barrier transmission; the best you can hope for is parallel receives. The only way to obtain good performance with this protocol is to have an underlying physical signaling implementation that offers a very fast message time.

4.3.2.1.3 Class P1c - Private channel, all-to-all, with network combining

This is similar to Class P1a (shown in Figure 4.6), except that multiple "barrier channels" are visible at the protocol level. This means that systems lacking the additional message memory hardware can still get the functionality of barrier message memory by utilizing a second "barrier channel" in the network. Cells which are not members of a barrier simply indicate they are "ready for all time" to the network, and thereafter ignore that channel.

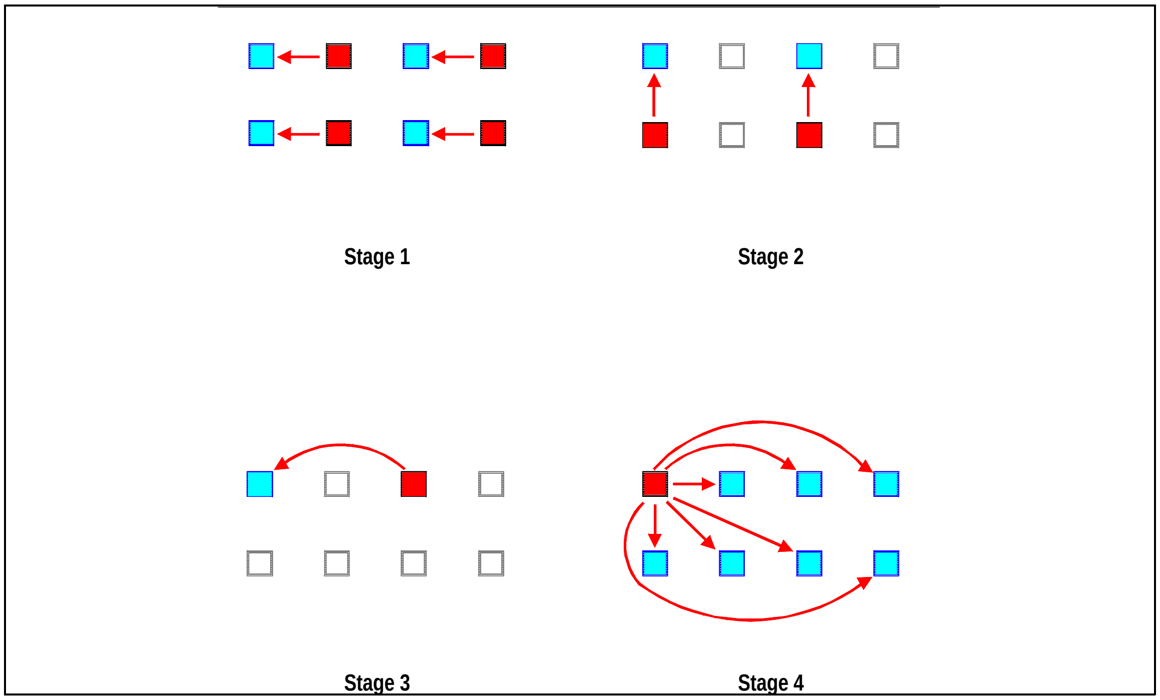


Figure 4.8 Class P1d protocol. Optimal for a network that allows both private communication and broadcasts.

4.3.2.1.4 Class P1d - Private channel, one-to-all, no network combining

With appropriate hardware, this messaging protocol offers similar performance and scalability to Class P2 protocols, except that the effective message time is somewhat shorter because all communications are unidirectional. By utilizing "private channels", multiple cells can communicate at once. The synchronization is composed of N-1 unidirectional messages (occurring in $\log_2 N$ time with the appropriate underlying signaling mechanism), plus one broadcast at the end to complete the barrier. In this scheme, a cell either transmits or receives, but never does both at once (Figure 4.8). Note that this is in contrast to the handshaking done in Class P2 protocols where a cell can send and receive at the same time. By reducing the overhead per message (since each communication is unidirectional), this protocol *could* have a shorter message time than a Class P2 protocol, which uses only private messages (no broadcasts), but requires bidirectional communication at each stage.

Note that there are still N transmissions being made; if this protocol is used with a physical signaling scheme that only supports global broadcasts (such as ethernet), the communications will become serialized and occur in N message times, not ($\log_2 N$ + 1).

4.3.2.2 Class P2 - Point-to-point messaging protocols (no broadcasts)

All of the Class P1 protocols relied on a final broadcast to inform all member cells that the barrier had been completed. If the underlying physical signaling mechanism does not allow a final broadcast, some other means of distributing the information is required. A further complication is that arbitrary connectivity may not be supported; cells may be restricted in whom they can communicate with. If the synchronization services share network bandwidth with the application, it may also be important to restrict the total number of messages used to synchronize.

4.3.2.2.1 Optimizing point-to-point messaging protocols

The underlying problem remains: an all-to-all information exchange has to occur. Since communication is restricted to point-to-point messages, there are several principles to make a protocol fast:

(1) Reduce the per-cell, per-barrier communication;
(2) Control the synchronization network loading;
(3) Reduce the total number of messages carried by the synchronization network;
(4) Overload messages where possible so that a single communication carries the information of multiple messages.

Depending on the underlying physical signaling scheme, different principles carry different weight. While they apply to broadcast-based protocols as well, they're particularly relevant to point-to-point messaging protocols.

4.3.2.2.1.1 Reduce the per-cell per-barrier communication

The simplest example of reducing communication is the evolution of the handshake from a two-message split handshake (Figure 4.4) to the reduced protocol single-message handshake shown in Figure 4.5. Pure hardware schemes (Figures 4.1 and 4.2) take this to an extreme with the communication consisting of a single voltage transition.

4.3.2.2.1.2 Control the synchronization network loading

Consider the four-cell synchronization problem shown in Figure 4.9. Each cell sends a single "I'm ready, tell me when you are" message to each of its three neighbors, then waits until it receives all three replies.

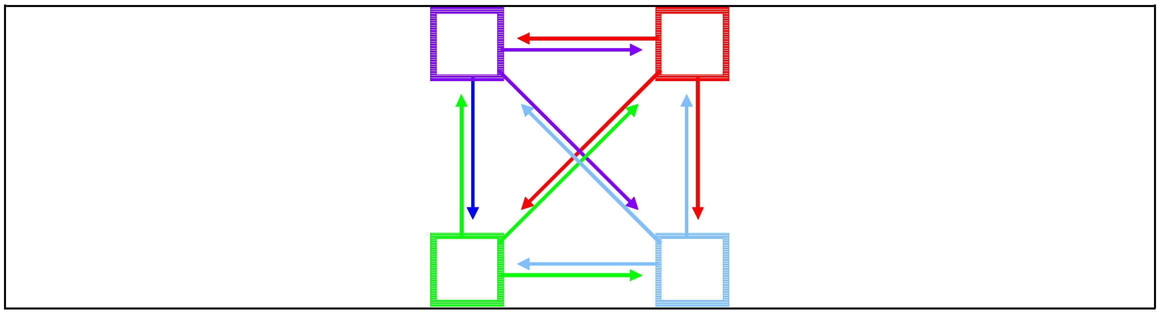


Figure 4.9 Four-processor all-to-all handshaking

Allowing one cell to "proxy" for another reduces the all-to-all communication pattern shown in Figure 4.9 to the three-stage communication shown in Figure 4.10.

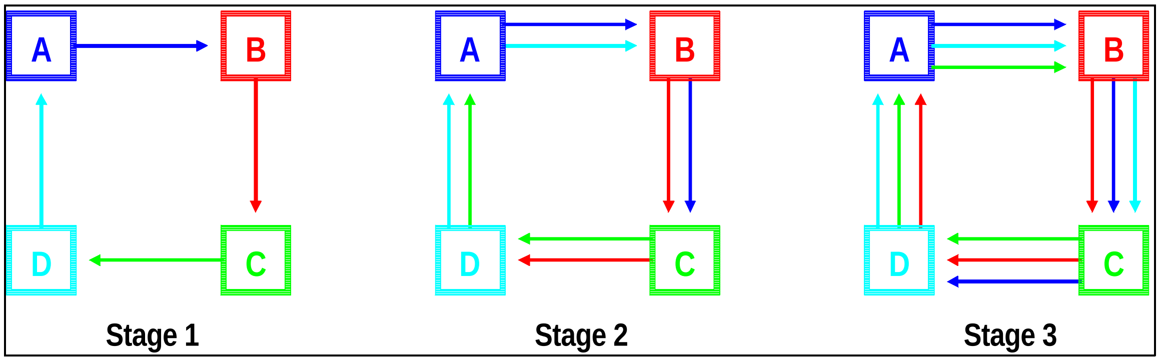


Figure 4.10 During each communication stage a cell proxies to its downstream neighbor for all the upstream neighbors it has heard from. At the final stage, each cell receives a proxy message for all other cells.

Each cell runs the same algorithm. For a barrier of N cells communicating in a unidirectional ring:

```
for I=0 to (N-1) {
    send a message to my downstream neighbor;
    wait for a message from my upstream neighbor;
}
```

Consider cell **A**. At the beginning of the barrier, stage 1, **A** sends a single "I'm ready, tell me when you are" message to **B**, then waits until it receives a similar message from **D**. Once it has received the message from **D**, it begins stage 2, and again sends a single "I'm ready, tell me when you are" message to **B**. This second message carries more information that the first, though, because sending it means that **A** has received a message from **D**. Thus, the stage 2 message **B** receives from **A** ***really*** means "**A** and **D** are both ready, tell us when you are". (The two parallel arrows in stage 2 of Figure 4.10 represent a single message carrying information about two cells; the three parallel arrows in stage 3 represent a message carrying information about three cells.) **A** meanwhile waits to receive the second message from **D**, which in turn means "**D** and **C** are both ready, tell us when you are." After receiving this second message, **A** then sends the third and final message to **B** which carries the meaning "**A**, **D**, and **C** are ready, tell us when you are." Once **A** receives its final message from **D**, it knows that **D**, **C**, and **B** are all ready, hence all cells have entered the barrier, and that **A** may now safely exit the barrier.

While the number of messages sent using the scheme shown in Figure 4.10 is the same as shown in Figure 4.9 (N **x**(N-1)), several things are being accomplished. First, only limited network connectivity is needed; each cell sends messages in a single direction to a single recipient. This makes it easier to build a separate, dedicated network for synchronization messages, or to add a small synchronization facility to an existing messaging system. Furthermore, the rate at which messages enter the network is controlled. Each cell can only send a single message at a time; the cell must then wait until it receives a message before sending out the next. Finally, the message transmission is fully parallelized. To the extent the underlying physical signaling scheme allows, each cell can send and receive a message at the same time, so the barrier completes in only (N-1) message times.

4.3.2.2.1.3 Reduce the total number of messages carried by the synchronization network

If the synchronization network does not support multiple simultaneous sends and receives, then message transmission becomes serialized. The protocol shown in Figure 4.10, which runs in just 3 message times (on a four-cell membership) if multiple sends and receives are supported by the underlying physical signaling mechanism, would require 12 message times to complete if message transmission were serialized due to network access restrictions. In other words, if an ethernet-like signaling scheme is all that's available, a Class P1b messaging protocol using broadcasts (which could complete in 4 message times) would be more efficient. Alternatively, the total number of messages sent for synchronization may need to be limited to conserve network bandwidth. In both cases, the aim is to use a messaging protocol that reduces the total number of messages, rather than trying to achieve parallelism. Such an approach is shown in Figure 4.11.

At stage 1, cell **A** sends an "I'm ready, tell me when you are" message to cell **B**. When **B** receives **A**'s message, it then sends a message "**A** and **B** are ready, tell me when you are" message to cell **C** (stage 2). At the end of stage 3, cell **D** knows that all cells have entered the barrier. While ***D*** now has knowledge that all cells have entered the barrier, no one else knows **D**'s status. **D** is obliged to send at least one message so that the other cells can know that **D** reached the barrier too. At stage 4, cell **A** receives the message from **D** that all cells have entered the barrier, then forwards that message to **B**. By stage 6, all cells are aware that all other cells have entered the barrier and hence can exit the barrier.

This approach, hereafter referred to as "token twice around" or **token 2x**, is fairly easy to program on a message-passing system. One cell is the "master" (**A** in Figure 4.11), and runs the following code:

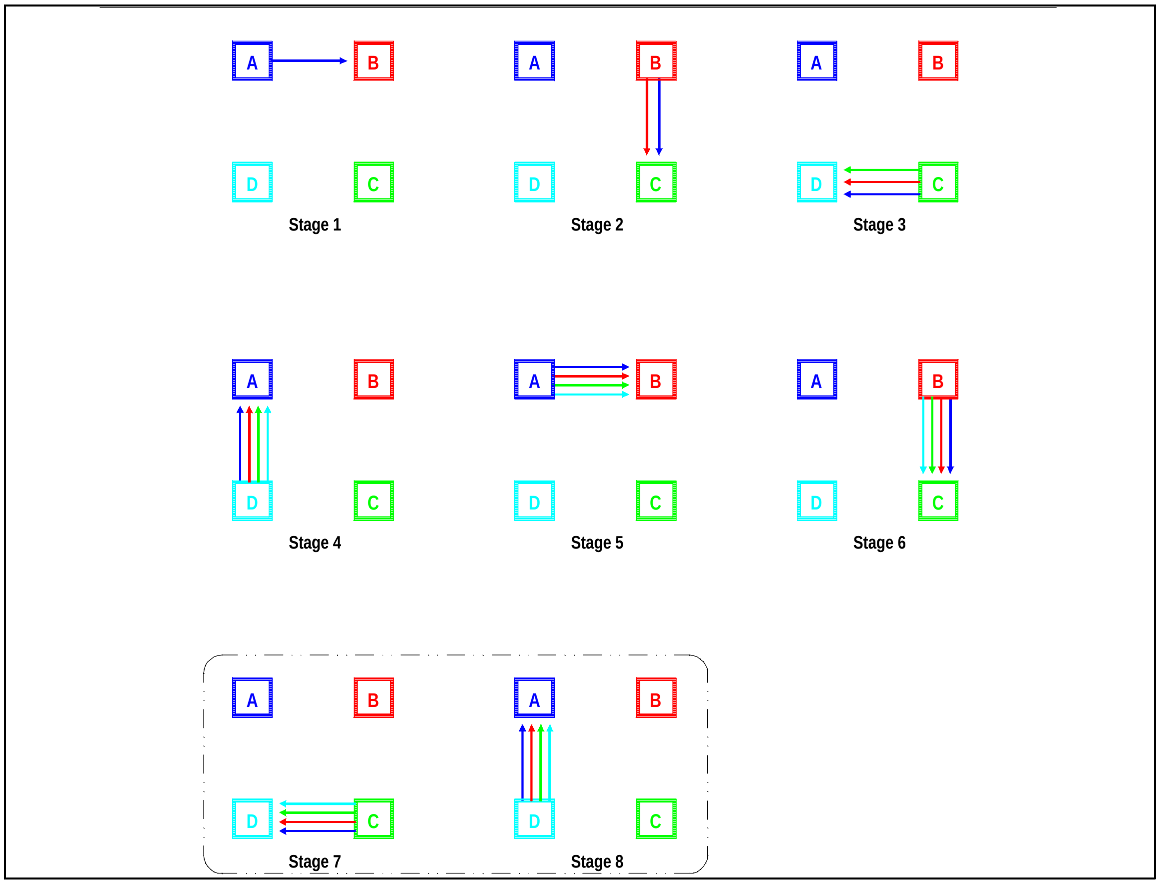


Figure 4.11 The "token twice around" scheme requires fewer messages than Figure 4.9, but all messages are now serialized. The last two stages carry no new information.

```
(1) send a token to downstream neighbor

(2) wait to receive a token from upstream neighbor
(3) send a token again to downstream neighbor
(4) wait to receive a token from upstream neighbor, and discard it
```

The other cells are "slaves" and run the "mirror image" of the above code.

```
(1) wait to receive a token from upstream neighbor
(2) send a token to downstream neighbor
(3) wait to receive a token again from upstream neighbor
(4) send a token again to downstream neighbor
```

While this simple scheme possesses a rather nice symmetry, it lacks efficiency. In the four-cell example shown above (Figure 4.11), the last two communication stages do not carry any new information. Keeping those two redundant stages simplifies implementation, but eliminating them reduces the number of message times from eight down to six, in this case yielding a 25% speedup.

4.3.2.2.1.4 Overload messages where possible so that a single communication carries the information of multiple messages

Each of the previous two examples shows some degree of overloading. As a message is relayed around the ring, it signifies readiness of greater numbers of cells at each stage. A transmission indicates not only the sender's readiness, but also the readiness of everyone the sender has heard from (whether directly or by proxy). Taken to the extreme, proxying can be extended across multiple dimensions as well as just for upstream cells.

Consider the nine cell group shown in the left half of Figure 4.12. Assuming that the synchronization network allows multiple simultaneous message transmissions, one can see that a synchronization scheme like that of Figure 4.10 would complete in N-1 message times (8, in this case). If the one large ring is split into two orthogonal sets of three rings each (shown in the right half of Figure 4.12), synchronization occurs in only 4

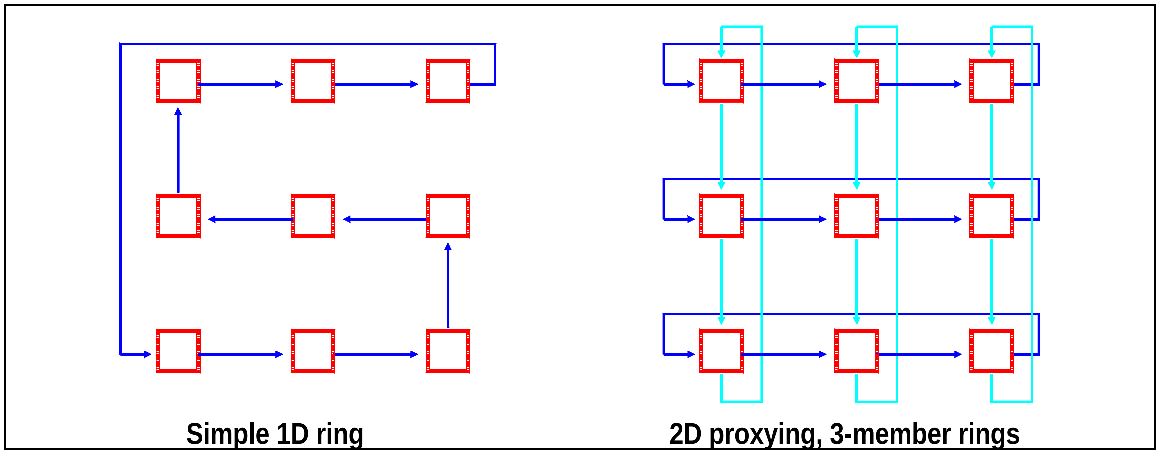


Figure 4.12 One large ring can be broken up into sets of smaller rings

message times, twice as fast as using a single ring. Figure 4.13 shows how the synchronization proceeds. Stages 1 and 2 synchronize all three processors in each of the three rows. Stages 3 and 4 not only synchronize the three cells within the three columns, but also carry information about the rows those cells are in. The synchronization messages within the columns are overloaded to carry the information that the rows are already synchronized. At the end of stage 2, each cell knows that all other cells in its row have entered the barrier. At the end of stage 4, each cell knows that ***all*** cells have entered the barrier. By proxying in two dimensions rather than just one, the problem of

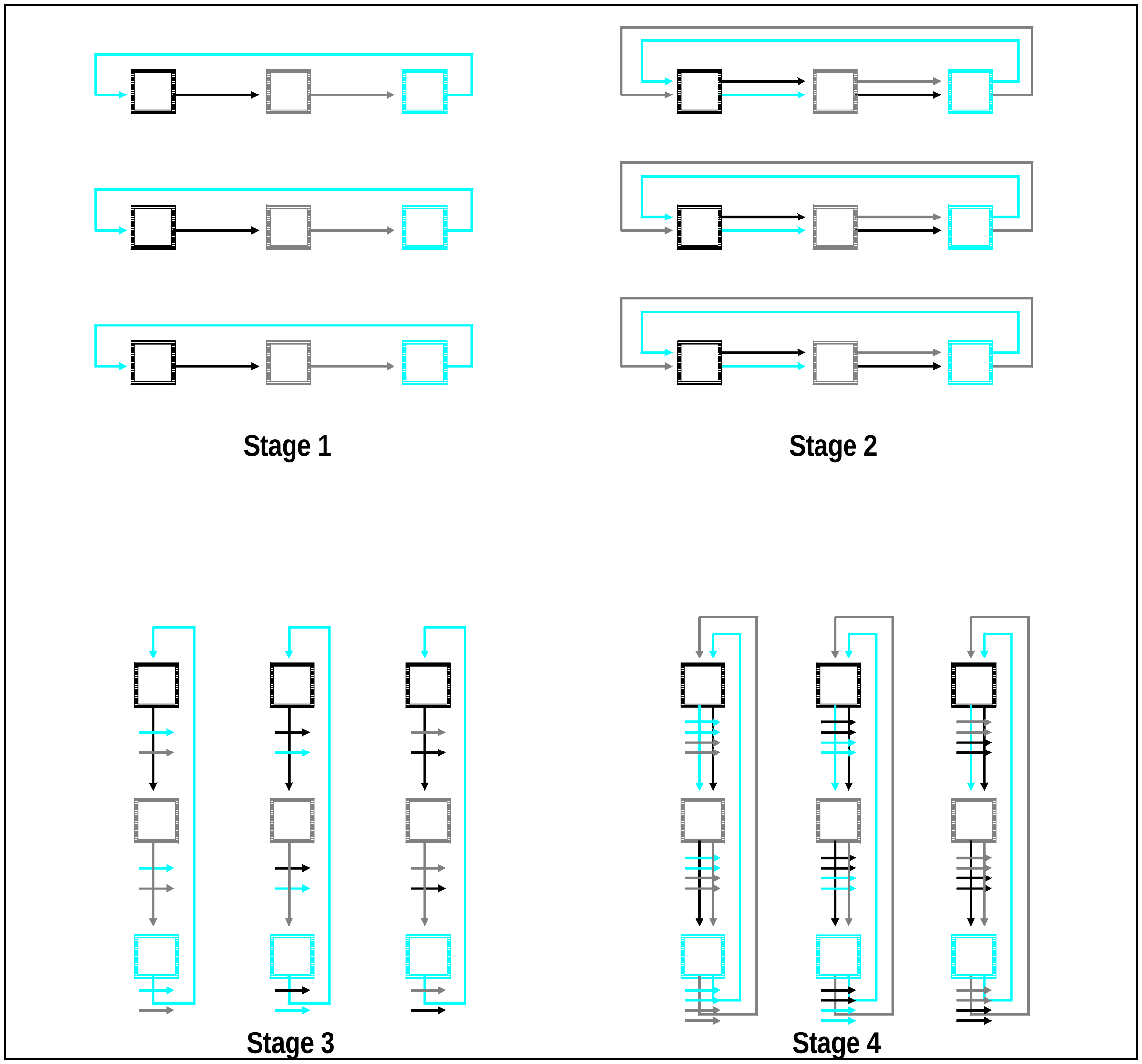


Figure 4.13 Synchronization occurs in four stages: two horizontal stages, and two vertical stages.

synchronizing nine cells is reduced into two sequential synchronizations of three parallel three-cell groups. Each three-cell synchronization requires 2 message times to complete. Thus, the total synchronization requires just 2*(3-1) = 4 message times. This is only half the time required for synchronization using a single ring and only a quarter of the time needed for a "token twice around" barrier.

The general method of overloading messages by splitting the rings into more groups of smaller loops can be continued until only loops of two or three cells remain. In essence, one embeds the largest complete hypercube into a group of processors, then "buddies up" the remaining processors with members of the hypercube. The synchronization algorithm then becomes:

(1) **handshake with your buddy if you have one,**
(2) **if you are a hypercube member, complete the hypercube synchronization,**
(3) **handshake with your buddy.**

Assuming the network can handle multiple messages, and that all messages have the same latency, this is the fastest way to synchronize a group of processors with N>9.

Consider the group of thirteen processors shown in Figure 4.14. Logically, they can be thought of as a 3-ary 2-cube, with five of the eight nodes having "buddies". Stage 1 is the hypercube and non-hypercube buddies handshaking. Stages 2 through 4 are the hypercube members handshaking along each dimension. Stage 5 is the hypercube buddies and non-hypercube buddies handshaking again. Obviously, powers-of-two cell groups will run one or two message-times faster than non-powers of two because the hypercube/non-hypercube buddy handshakes at the beginning and end are eliminated.

While the underlying communication network has a certain dimensionality, the messaging protocol can "embed" and use a higher order dimensionality. For instance, in a sixty-four cell system, a global 1D ring-based synchronization would take N-1, or 63, message times, and require N*(N-1) messages. One could embed

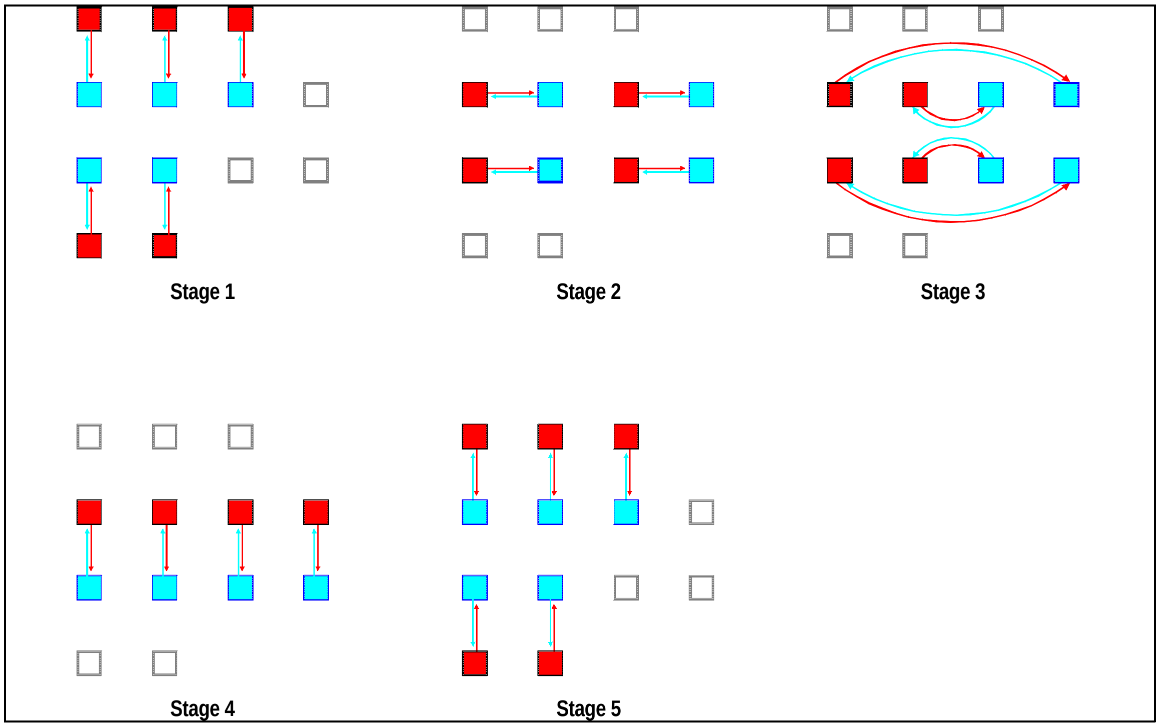


Figure 4.14 Thirteen cell synchronization with hypercube buddies

a 6-ary 2-cube into the ring, and reduce the communication requirements to N*6 messages, occurring in 6 message times. In a machine where message time is dominated by network latency rather than processing overhead, though, ***message times for the two may be different and are not directly comparable***. In the global 1D (N-1) ring, the message time is merely the time needed to send to one's adjacent neighbor. With a hypercube embedded in a ring, though, message times are longer by a factor of N/2 to N, because one of the handshake partners is always somewhere between half-way around the ring to all the way around the ring. Even though the embedded hypercube uses fewer total messages and completes in “fewer message times", its message times are N/2 to N times longer due to the higher network latency. When embedding higher-order communication into a lower-order network, message time isn't a constant; rather, it scales with N. The N-1 ring, a "worse" technique (because it runs in N rather than $\log_2 N$ time) will always run faster than a "more efficient" higher-order embedded network in a network latency-dominated system (as opposed to a message-processing overhead-dominated system) because the N-1 ring needs only 1 message-trip-time around the ring to complete. Embedded higher-order schemes will require multiple trips around the ring.

If multiple network dimensions are available, using them will speed things up by reducing message latency. For instance, splitting an 8x8 2D torus into a set of horizontal eight-member rings and vertical eight-member rings reduces the total message distance (and hence time) per ring from 63 hops (N-1 where N=64) to only 7 hops (N=8). Thus, by using proxying in 2 dimensions, total synchronization time drops from 63 next-neighbor message times to 2*(8-1) = 14 next-neighbor message times, a better than 4-fold improvement.

4.3.2.3 Messaging Protocol Conclusions

Barrier messaging protocols fall into two general classes: those that rely on some form of broadcast, and those that rely on private messages. All private-message protocols reduce to some combination of two basic schemes:

(1) pass a token twice around a ring of cells (referred to hereafter as the ***token 2x*** method), or
(2) for each cell in a ring of N cells to perform (N-1) sends and receives (referred to hereafter as ***(N-1) send/recv***).

At its most parallel, the token 2x method becomes a reduction/broadcast tree embedded into the physical network, and the (N-1) send/receive becomes handshakes on an embedded hypercube (Figure 4.14). For best performance, when using a non-broadcast physical signaling scheme that serializes message transmission (such as RTS-based message-passing), one should use the token 2x method (good) or the embedded reduction/broadcast (better) to minimize the number of messages sent. If the underlying physical signaling scheme supports parallel messaging, the 1D (N-1) method (or some higher embedding) is a better messaging choice (by roughly a factor of two) over the token 2x method. A number of people have observed that reduction/broadcast maps nicely (that is, no congestion) to hypercubes[46], but the reduction/broadcast requires two separate send and receive stages. Handshaking allows simultaneous sending and receiving, thus completing in roughly half the time of a reduction/broadcast.

4.3.3 Allowable barrier memberships

Connectivity between participating barrier members affects both proxying efficiency as well as the amount of information needed to identify the participating members. For example, a compact 3x3 block can be represented more compactly (and with less information) than 9 random cells which are scattered throughout the array. Consider a group of nine cells with a connecting message ring. Assume that any cells not participating in the barrier would simply let messages pass through. This scheme easily lets one synchronize an arbitrary subset of N cells out of the nine in (N-1) message times, assuming all cells in the subset knew how many were participating.

If two dimensional connectivity were available, synchronizing all nine could be done more efficiently using 2D rings as shown in Figure 4.13. This requires only four message times, but requires more information at each cell than the single 1D loop. All cells now need to know the number of participating cells ***in each dimension of communication***, and some subgroups are completely impossible to synchronize with this scheme.

Figure 4.15 shows a subset of five cells out of the nine. While the simple 1D ring is able to synchronize the shown subset of

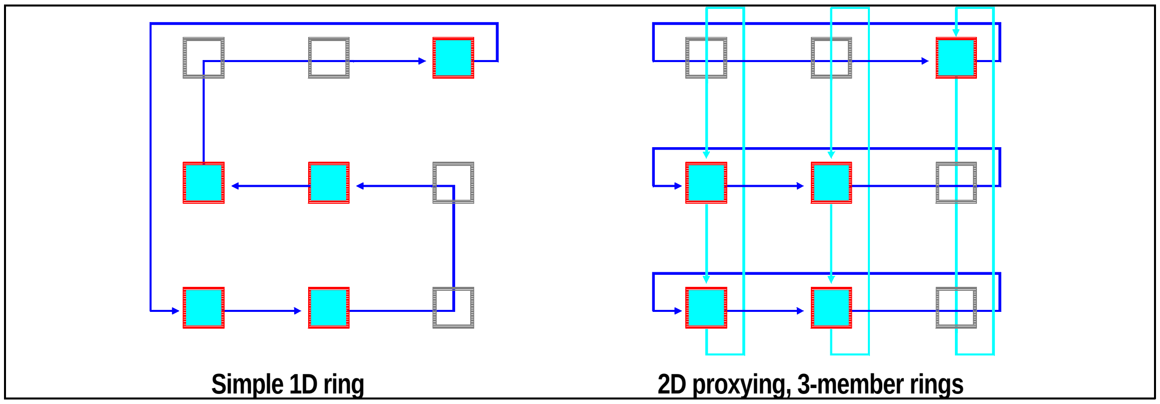


Figure 4.15 Faster barrier schemes can't handle arbitrary subsets and still deliver high performance. Some tradeoffs are necessary.

five, the 2D proxying scheme (twice as fast when synchronizing the whole group of nine) is unable to work with this barrier membership. The upper-right-hand cell is simply unable to communicate with the other members either directly or via proxying through another member cell. Thus, by restricting the allowable subsets, faster barrier implementations may be used. It is generally faster, and less complex, to implement a synchronization scheme that allows restricted partitioning than to implement one that allows arbitrary subset partitions.

Four general classes of barrier membership exist; listed in order of increasing implementation complexity, they are:

(1) a single global, system-wide barrier;
(2) multiple barriers with fixed, non-overlapping memberships;
(3) multiple barriers with fixed, overlapping memberships;
(4) multiple barriers with arbitrarily overlapping memberships.

The following sections describes each in more detail.

4.3.3.1 Class M1 - A single global, system-wide barrier

The simplest barrier is the single global barrier which synchronizes the entire array. Only one sync may be pending at any cell; anything else is a deadlock condition. For any given implementation style (dedicated hardware vs. general message-passing), this barrier can be made faster than any comparable subset-capable design that synchronizes the whole array because only a minimum of information is needed in the synchronization message. The underlying physical signaling scheme doesn’t even need to carry a barrier ID because it can have only one value. Only a single "barrier message memory" per cell is required, since at most one message can be outstanding.

4.3.3.2 Class M2 - Fixed, non-overlapping memberships

Subset barriers are allowed, but only certain subsets are permissible, and the subsets may not overlap. If one builds a dedicated synchronization network in hardware, one can easily allow for partitioning within the network by isolating a particular subset (Figure 4.16). A system using Class S3 or S4 physical signaling may want to use a partitioning scheme like this in the interest of performance, such as restricting subsets to rectangles on a 2D mesh or blocks on a 3D mesh, so that more efficient handshake protocols may be used. Because barriers are non-overlapping, only one sync may be pending at a single node at any one time. This restriction means only a single wire is needed for a hardware implementation, or a single buffer for a message-based or hybrid implementation.

4.3.3.3 Class M3 - Fixed, overlapping memberships

Subset barriers are allowed, and may overlap. This is more difficult to implement with a low level (Class S1 or S2) physical signaling scheme because multiple barriers may be pending at each cell. The sync and acknowledgment signals must be kept separate, so multiple hardware instances are necessary. If a cell has three pending syncs, three copies of the synchronization hardware at each cell are needed, along with three copies of the fan-in/fan-out network. This resource issue isn’t as significant in a higher-class physical signaling implementation, but additional buffering (one buffer per pending sync) is still needed.

4.3.3.4 Class 4 - Arbitrarily overlapping memberships

This implementation is the most difficult to directly build in hardware, as this scheme requires either building multiple copies of a reconfigurable synchronization network, or else suffering a configuration dependent performance. Essentially, one copy of network interface hardware is necessary per possible pending sync allowed per cell.

4.3.4. Barrier capacity

Barrier capacity is a measure of "how many different barriers can be supported simultaneously"? Since hardware and bandwidth are required to support each pending sync, this is an important design decision. Even a pure message-passing-based system needs hardware memory buffers to store the incoming messages until they can be processed. Barrier IDs can be virtualized (so that over the course of an application, a cell may participate in hundreds or thousands of different barriers by re-using barrier resources), but the total number that may be pending at a single cell, or on the machine as a whole, is a constraint of the physical implementation.

Consider the simple hardware barrier scheme shown back in Figure 4.1. Now imagine adding a second pull-up resistor, and a switch in the middle of the network capable of splitting it into two parts (Figure 4.16). Within the restriction of the barrier memberships being non-overlapping, one barrier channel implementation is serving the needs for two barriers. If two barriers with an overlapping membership (one cell can be a member of both) were needed, then two distinct channels must be implemented.

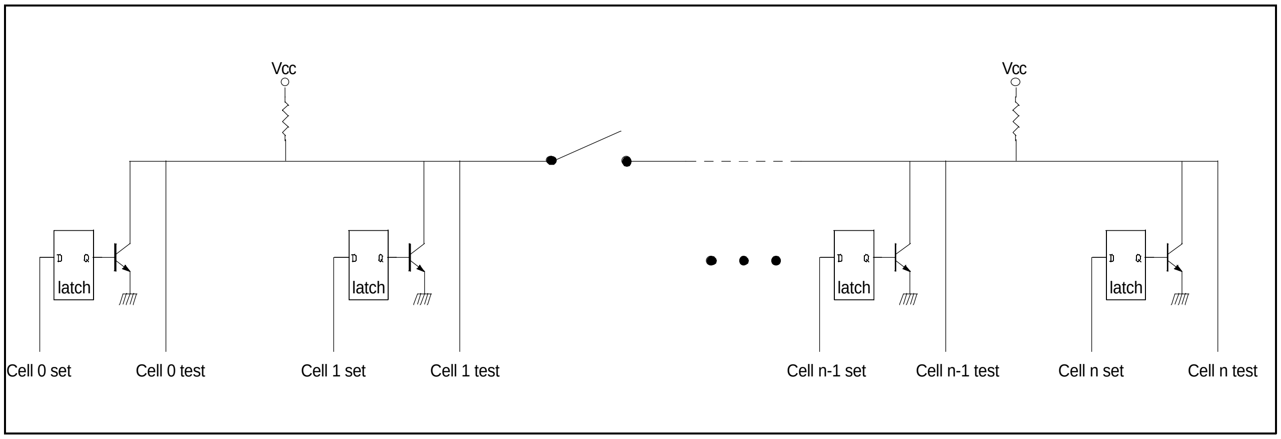


Figure 4.16 A simple implementation of a single hardware barrier that can be partitioned to provide two distinct, non-overlapping barrier channels.

While only a hardware implementation example is shown here, the same sort of techniques apply to message-based systems. For instance, suppose one used a 1D ring snake passing through a group of cells to carry the barrier messages. One could either use two distinct barrier IDs on the barrier messages, or one could re-route the 1D ring snakes so that they do not overlap (Figure 4.17)[16]. Using two disjoint ring snakes for communication allows reuse of the barrier IDs, meaning fewer barrier IDs have to

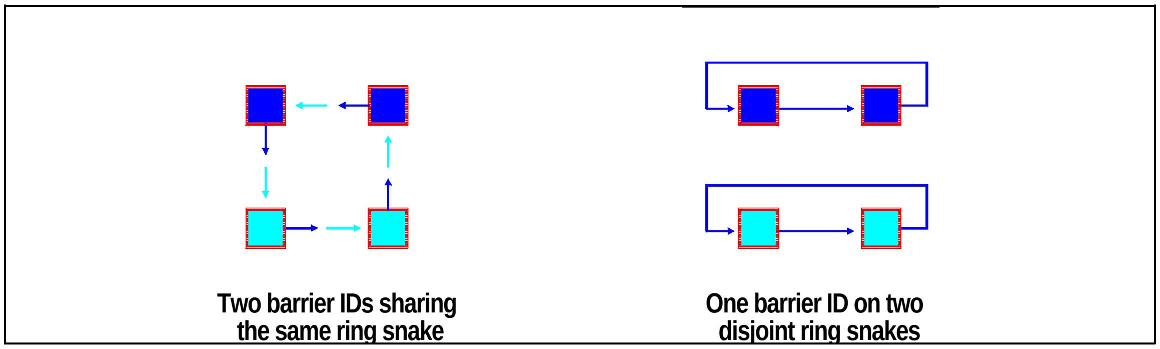

Two barrier IDs sharing the same ring snake
One barrier ID on two disjoint ring snakes

Figure 4.17 By splitting the communication network, a message-passing based system can use fewer barrier IDs while still providing the same number of disjoint subsets.

be supported overall, which implies fewer bits are needed to communicate readiness, resulting in shorter barrier messages and smaller storage requirements for buffering early sync messages. If fewer "barrier channels" are implemented, less buffer space must be reserved.

Because "number of simultaneous barriers supported" can be a vague number, "barrier capacity" is a more accurate description of the implementation's capabilities. Barrier capacity has two components:

(1) How many barriers can be pending simultaneously at a single cell with a particular implementation, and

(2) Does the implementation allow partitioning (as shown in Figures 4.16 and 4.17)?

The answer to (1) tells us about the intrinsic "capacity" of the synchronization implementation, and the answer to (2) tells us how effectively that capacity can be allocated to support application barrier needs.

4.4 Design methodology

Now that the design space (physical signaling scheme, messaging protocol, allowable barrier memberships, barrier capacity) has been defined, this knowledge can be used to create an appropriate barrier implementation with a given a set of finite resources.

4.4.1 The questions

The relevant design questions cover three general areas:

(1) Resources availablity (physical signaling and messaging schemes) on the target machine.

Physical signaling

What physical signaling schemes are available? What connectivity do they offer, and to what degree do they allow sending messages in parallel?

Messaging

For the physical signaling schemes available, what message schemes make sense? How fast is a message time for each, and will it allow messaging in parallel?

(2) What demands will be placed on the implementation (memberships (global, non-overlapping subsets, arbitrary subsets) and when they are defined (compile time, link time, run time))?

Barrier memberships

What memberships are required, and when are they set (compile time, link time, run time)?

Barrier capacity

What kind of capacity is needed? Is a single global barrier sufficient, or are multiple (potentially overlapping) barriers required?

(3) What are the performance criteria for declaring a “better” implementation (raw execution speed, minimal resource consumption, or speed within some resource constraint)?

How frequently is this barrier going to be used (thus how important is barrier speed)? Is a barrier’s overhead execution allowed to interrupt a foreground program, or must the barrier avoid delaying foreground execution? How much importance is attached to speed vs. minimizing communication resource consumption?

4.5 Crafting a barrier implementation

The first part of the chapter outlined a barrier synchronization design space (physical signaling scheme, messaging protocol, barrier memberships, and barrier capacity), followed by a general methodology for creating a barrier. This section reviews the non-broadcast physical signaling options on iWarp, then combines them with various prototypical messaging protocols to show how barrier performance is affected by the combination of signaling scheme ***and*** messaging protocol. Design assumptions include:

(1) barrier memberships will be contiguous blocks with dimensions that are multiples of two, and

(2) multiple simultaneous barriers need to be supported, with up to four possible barriers pending at a given cell.

4.5.1 Physical signaling on iWarp

Figure 4.18 summarizes the results of Chapter 3: PCT-based connections offer about an order of magnitude better performance than deposit message passing, and several orders magnitude better performance than RTS-based message passing.

Physical signaling	1 cell	2 cells	3 cells	4 cells	5 cells	6 cells	7 cells
RTS-based message-passing (32 bytes)	27964	28200	27966	28200	27966	28198	28839
Deposit message-passing (4 bytes)	258	264	269	275	278	283	289
PCT-based connections (4 bytes)	10	16	20	26	30	36	40

Figure 4.18 Measured average message times (clocks) on iWarp for the three signaling schemes vs. distance.

These results strongly support using PCT-based connections for the TCS barrier, but it is worth studying the interaction of all three signaling schemes with the different messaging protocols. RTS-based message passing is a good model for any real-world single-access collision-free communication method (such as FDDI). Deposit message-passing is a reasonable model for general-purpose message-passing in a machine that supports parallel communication (such as an MPI library on a group of machines connected through a crossbar switch), and PCT-based connections serve as a general model for connection-based communication.

4.5.2 Non-broadcast messaging protocols on iWarp

Two general non-broadcast messaging models exist for barrier synchronization: one is passing a token twice around a ring of cells (***token 2x***), and the other is performing (N-1) send and receive operations at each of the N cells in the ring (***(N-1) send/recv***). At their most parallel, the token 2x method evolves to a ring-embedded reduction/broadcast tree, and (N-1) send/recv becomes handshakes on a hypercube.

Figure 4.19 shows the models for barrier execution time for the two prototypical barrier messaging schemes, and their two fully-parallel derivatives, for a group of N cells. The ***token 2x*** method generates a total of 2N messages and has an expected execution time of [startup + (2N x message_ time)]. "Startup" is the time required to enter the barrier subroutine and obtain control of the particular communication resource, and was measured at 40 clocks in the test code. The ***(N-1) send/recv*** method generates a total of Nx(N-1) messages; execution time depends on the degree to which parallel messaging can occur. If the underlying physical signaling scheme serializes all messages, execution requires [startup + (N x (N-1) x message_time)] clocks. If the physical signaling scheme supports full parallel messaging, execution requires only [(N-1) x message_time]. The number of messages sent is (N x (N-1)) in both cases, but the non-parallel case takes N-times longer to send them. The parallel derivatives (embedded reduction/broadcast tree, handshakes on a hypercube) are listed as well, but note that their values for **message_time** may be larger.

The following constants were measured on iWarp, and should be used when evaluating the model equations of Figure 4.19:

startup time is 40 clocks (measured);
RTS-based message_time is 28340 clocks (measured as an average of all cell-to-cell communication times);
Deposit-MP message_time is 260 clocks (derived from Figure 4.17);
PCT-based message_time is 11.5 clocks (derived from Figure 4.17).

Messaging Scheme	Number of Messages	Execution Time with Fully-Parallelized Messaging	Execution Time with Fully-Sequentialized Messaging
1D ring, (N-1) send/receives	N x (N-1)	startup + ((N-1) x message_time)	startup + (N x (N-1) x message_time)
fully-embedded hypercube, handshakes on each dimension	N x ($\log_2$N)	startup + ($\log_2$N x message_time)	startup + (N x ($\log_2$N) x message_time)
1D ring, token 2x around	2N*	startup + (2 x N x message_time)	startup + (2 x N x message_time)
fully-embedded reduction / broadcast tree	2 x (N-1)	startup + (2 x $\log_2$N x message_time)	startup + (2 x N x message_time)

* The 1D ring, **token 2x** method is generally implemented using 2N message times, but the last two messages, as shown earlier in this chapter, carry no new information and could be omitted, yielding 2N-2 messages.

Figure 4.19 Relevant iWarp barrier messaging schemes

This table implies that a physical signaling scheme that supports parallel messaging (such as PCT connections or deposit message-passing) runs faster using the **(N-1) send/recv** method (or a more-parallel derivative of it) than the **token 2x** method. Conversely, **(N-1) send/recv** and its derivatives do worse on a fully-sequentialized messaging system (such as a token-ring or ethernet).

4.5.3 Putting it together

Now that the major components (the physical iWarp signaling schemes and the messaging schemes) have been introduced, they are put together in various combinations to show how their interaction affects real-world barrier performance.

4.5.3.1 RTS message-passing and various messaging schemes

Figure 4.18 shows the measured message time for communication across distances of 1 to 7 cells, using RTS-based message passing, deposit message-passing, and low-level hardware-supported (PCT) connections. These data indicate that PCT-based next-neighbor message time is 11.5 clocks and RTS-based message time is around 28,340 clocks. Using these derived message times, performance can be predicted for synchronization barriers using the different messaging schemes.

Figure 4.20 shows measured barrier performance on iWarp using RTS message-passing as the underlying signaling scheme for a variety of messaging schemes. The **token 2x** messaging scheme inherently serializes all messages (Figure 4.11), and performance is as predicted. The r**eduction/broadcast** messaging scheme attempts to send multiple messages at once. Given the token-ring-like nature of the underlying RTS message-passing system, one expects (and indeed sees) execution time scaling as O(N) for both the **token 2x** and **reduction/broadcast** barriers using RTS message-passing, whereas the **(N-1) send/recv** messaging scheme's execution times explode exponentially with increasing numbers of cells due to the N^2 messages being serialized. The absolute performance of **reduction/broadcast** is better than the **token 2x** method (3.6 million clocks vs. 2.1 million clocks at 64 cells), despite sending the same number of messages on a serial network, because the on-cell computation needed for messaging can be done in parallel even though the actual sending of messages on the token ring remain serialized. The **(N-1)send/recv** also showed that some messaging overlap occurred (performance was not as bad as predicted for "no overlap" during sending), but not much. For RTS message-passing, the **(N-1) send/recv** method offered the worst-overall performance, and the embedded **reduction/broadcast** (the most-parallel form of **token 2x**) offered the best performance.

Messaging scheme	4 cells	8 cells	16 cells	32 cells	64 cells
RTS/Token2x (predicted)	223,564 (226,760)	448,371 (453,480)	910,282 (906,920)	1,815,121 (1,813,800)	3,648,581 (3,627,560)
RTS/Reduction-Broadcast (pred overlap)	136,592 (113,400)	264,622 (170,080)	511,648 (226,760)	1,043,711 (283,440)	2,119,398 (340,120)
RTS / (N-1) send/recv -100runs (pred overlap) (no overlap)	220,961 (85,060) (340,120)	950,134 (198,420) (1,587,080)	4,021,443 (425,140) (6,801,640)	16,485,541 (878,580) (28,113,320)	23,861,298 (1,785,460) (114,266,920)

* the (N-1) send/recv implementation was measured over just 100 runs

Figure 4.20 - Predicted and measured barrier execution times in clocks (1000 runs) using RTS message passing with various messaging schemes for varying barrier membership sizes.

Messaging scheme	4 cells	8 cells	16 cells	32 cells	64 cells
DMP/ Token 2x (predicted)	2,141 (2,200)	4,339 (4,400)	8,706 (8,800)	17,403 (17,600)	34,798 (35,200)
DMP/Reduction/ Broadcast (predicted)	1,488 (1,692)	2,079 (2,322)	2,623 (2,952)	3,142 (3,582)	3,687 (4,212)
DMP/1D (N- 1) send/recv (predicted)	1,264 (1,515)	2,982 (3,535)	6,669 (7,575)	13,569 (15,655)	27,361 (31,815)
DMP/hypercube (predicted)	822 (1,090)	1,244 (1,635)	1,673 (2,180)	2,074 (2,725)	2,532 (3,270)

Figure 4.21 - Predicted and measured barrier execution times (in clocks) using deposit message-passing (1000 runs) with various messaging schemes for varying barrier sizes.

4.5.3.2 Deposit message-passing and various messaging schemes

Figure 4.21 shows the results of similar performance predictions and measurements for deposit message-passing. Unlike the RTS message-passing implementation, the deposit message-passing implementation of **(N-1) send/recv** shows better performance than the **token 2x** method, and the logical extension of **(N-1) send/recv**, handshakes on a hypercube, offers between 1.5 and 2 times better performance than the reduction/broadcast.

4.5.3.3 PCT-supported connection and various messaging schemes

Figure 4.22 shows the performance of the **token 2x** and **(N-1) send/recv** methods using connections supported by two PCTs-per-cell. The error in the predictions is primarily due to the use of a simplified latency model. Note that PCT-supported connections allow overlap of sending and receiving, as well as allowing all cells to send at once. As a result, the **(N-1) send/recv** method outperforms the **token 2x** method by almost a constant factor of 2. Because PCTs are scarce, higher order embeddings (such as a full hypercube for hypercube handshakes) are too expensive to implement, especially since one of the design criteria for this exercise is to be able to support four pending barriers per cell. Thus, a 2D **(N-1) send/recv** implementation, with proxying in 2 dimensions, was only implemented for the 64-cell case because, at that size, it could map cleanly onto the underlying network using the array “backloops” (because the underlying network is a torus rather than a flat array) requiring only a single additional PCT. To implement the 2D **(N-1) send/recv** for any other size would require 3 additional PCTs; more resources than can be afforded. In any event, even the simple 1D **(N-1) send/receive** implemented with two PCTs is roughly three times faster than the fastest deposit message-passing implementation, and for special, whole-array cases, the 2D implementation is achievable for even greater speed. Furthermore, the worst-case difference between predicted and measured performance is 1 to 2 microseconds.

Figure 4.23 demonstrates how barrier skew (the time between when the first cell exits a barrier and the last cell exits) is affected by the messaging scheme. While not extensively covered

Messaging scheme	4 cells	8 cells	16 cells	32 cells	64 cells
PCT/Token 2x (predicted) max min	152 (160) 160 152	247 (264) 248 240	426 (448) 432 424	795 (816) 808 792	1513 (1552) 1520 1512
PCT/1D (N-1) send/recv (predicted) max min	97 (110) 104 88	146 (156) 152 144	242 (248) 248 232	423 (432) 432 416	800 (800) 808 792
PCT 2D (N-1) send/recv (predicted) max min	N/A	N/A	N/A	N/A	448 (456) 448 448

Figure 4.22- Predicted and measured barrier execution times PCT-supported connection-based barriers (1000 runs) for varying messaging schemes and barrier membership sizes.

by this thesis, it is worth noting that the “token 2x” method has a constant skew whether any stragglers (cells which enter the barrier much later than all the other participants) are present or not because cells always exit the barrier sequentially in a ringward order. Also note that in the presence of stragglers (real-world conditions) the higher-order embedded ring (2D in this case) has a lower skew than the lower-order embedded ring (1D) (64 cells is the only example shown). This result will hold for higher-order embeddings because the maximum skew is a function of the total number of parallel message times needed to complete: high-order embeddings will have fewer parallel message times total, and hence cannot “get as far behind”.

4.5.4 Conclusions

PCT-supported connections let us build fast barriers with predictable and repeatable performance. Furthermore, while the physical signaling scheme has a great effect on barrier performance, getting the best performance from a particular signaling scheme implies choosing a messaging scheme appropriate

Messaging	4 cells	8 cells	16 cells	32 cells	64 cells
PCT/Token 2x No stragglers avg skew max skew min skew	 32 32 32	 79 80 64	 170 176 168	 336 360 336	 711 712 704
PCT/Token 2x 1 straggler avg skew max skew min skew	 32 40 32	 76 80 72	 168 168 168	 351 352 344	 712 712 712
PCT/1D (N-1) No stragglers avg skew max skew min skew	 8 16 8	 8 16 8	 10 16 8	 11 16 8	 8 16 8
PCT 1D (N-1) 1 straggler avg skew max skew min skew	 22 24 16	 63 64 56	 131 136 128	 267 272 264	 547 552 544
PCT 2D (N-1) No stragglers avg skew max skew min skew	 N/A	 N/A	 N/A	 N/A	 96 96 96
PCT 2D (N-1) 1 straggler avg skew max skew min skew	 N/A	 N/A	 N/A	 N/A	 122 128 128

Figure 4.23- Measured barrier skews (in clocks) for the three PCT-supported connection-based barriers (1000 runs) with all cells entering barrier together (no stragglers), and with one cell entering 2000 clocks after all other cells (1 straggler).

to the physical scheme. Because PCT-supported connections allow full overlap of sending and receiving, Figure 4.22 shows that the (N-1) send/recv method (which generates Nx(N-1) messages) runs nearly twice as fast as the token 2x method (which only generates 2N messages). As a compromise between resource economy and speed, the TCS barrier will use 1D rings of PCT-supported connections for barrier memberships smaller than the whole array, and 2D rings for whole-array barriers, with the (N-1) send/receive messaging protocol. Resource cost will be 1 outgoing PCT, plus one PCT per possible pending barrier per cell (five total if the goal is to support up to four pending barriers per cell).

4.6 Chapter summary

This chapter explained the function of barrier synchronization, examined means of efficiently implementing barrier schemes, and explored some of the advantages/drawbacks of various implementations. Those principles were applied to implement barrier synchronization on a real target machine, iWarp. Finally, those implementations were benchmarked to demonstrate that their measured performance agrees well with the p redictions. In the end, a barrier mechanism was developed for TCS that can synchronize a group of 4 cells in under 160 clocks (8 microseconds), and a group of 32 cells in less than 816 clocks (41 microseconds). As a special case, the entire array (64 cells) can be synchronized is just 456 clocks (23 microseconds). Not only fast, this barrier mechanism is predictable to within 1 to 2 microseconds. This fast barrier mechanism allows tasks to coordinate at a very fine granularity level (tens of microseconds, in contrast to application latency requirements in the milliseconds).

Chapter 5 - TCS Control Primitives

5.1 Introduction

Chapter 4 provides a fast (8 to 40 microseconds), predictable (within one microsecond) barrier synchronization primitive. This chapter uses this primitive to construct the three remaining TCS control primitives: connection set reconfiguration, task start, and task end. Given the fast, predictable barrier synchronization primitive, the remaining primitives can also be implemented to yield fast, predictable performance.

5.2 Connection set reconfiguration

Connection sets can be reconfigured on the iWarp in one of two ways: source-routed connection setup/tear down, and switch-based reconfiguration. Because each iWarp cell has direct access to its network switch state, the cell can simply do a direct, brute-force reconfiguration of the relevant PCT entries. The general procedure is:

```
barrier_sync(task_members);
Turn off global events;
barrier_sync(task_members);
reconfigure switch(new_set);
barrier_sync(task_members);
Turn global events back on;
```

This direct approach offers several advantages, including

(1) exact knowledge of the communication resources each connection will be using,

(2) switch reconfiguration time is a constant, regardless of the connection set being switched, and

(3) simple implementation.

The operation to "reconfigure the switch" amounts to a few memory-to-register transfers (which have a predictable execution time), and a barrier_sync() operation is just the primitive discussed in the previous chapter. Thus, because each component has predictable performance, the total operation should have predictable performance as well, and we discuss this aspect (hierarchical predictability) in Chapter 7.

5.2.1 Reconfiguration model

As shown in the pseudocode in Section 5.2, switch-based connection set reconfiguration requires three barrier synchronization operations plus the time for the actual reconfiguration. The first barrier ensures all cells are executing foreground code, preventing any possible deadlocks with a cell waiting for RTS services (such as file I/O). Once the first barrier completes, interrupts (such as event handlers) are turned off and a second barrier executes. This barrier is necessary to safely put the RTS to sleep. At this point the cells can reconfigure their switch state. A final barrier is needed to let all cells know the reconfiguration is complete, and then interrupts can be turned back on. The reconfigure execution time model is therefore:

reconfig_time = switch_reconfiguration_time +
(3x(barrier_lookup_time + barrier_time(N)))

where **barrier_lookup_time** is the time needed to look up the resource configuration needed for the barrier (that is, what dimensionality of embedded rings does this barrier use, how many cells are in each dimension, and which PCTs are used for each dimension), and **barrier_time(N)** is just the barrier execution time from Figure 4.21 (the **PCT 1D ring (N-1) send/recv** row).

5.2.2 Measured performance and predictions on iWarp

Barrier_lookup_time was measured as 12 clocks, and switch_reconfiguration_time was measured as 162 clocks; no variations are expected in these as they are just local memory operations.

Figure 5.1 uses the barrier times from Figure 4.21 for the 1D PCT ring, and the equation expressed in Section 5.2.1, and combines them to predict reconfiguration time vs. numbers of cells. Those

	4 cells	8 cells	16 cells	32 cells	64 cells
reconfigure time	444	598	872	1425	2551
(predicted)	(465)	(612)	(900)	(1443)	(2574)
max	448	616	888	1456	2584
min	440	576	848	1392	2520

Figure 5.1 Reconfiguration times (in clocks) for various-sized groups of cells, using the simple 1D (N-1) ring barriers as shown in Figure 4.9, 1000 runs.

predictions are then compared to the measured reconfiguration times.

Figures 5.1 (for 1-D barrier rings) and 5.2 (a 64-cell 2-D set of barrier rings) shows good repeatability for the reconfigurations, although the predictions tend to diverge from the measured values by roughly 5%. These measured reconfiguration times are used in future pattern predictions rather than predicted reconfiguration times.

	64 cells
reconfigure time	680
(predicted)	(692)
max	712
min	680

Figure 5.2 Reconfiguration times (in clocks) for whole array, using a more complex 2D (N-1) ring barrier, 1000 runs.

5.2.3 Connection-set reconfiguration conclusions

Given a predictable barrier synchronization primitive, one can construct a predictable reconfiguration primitive. Because the reconfiguration primitive requires multiple barriers, barrier performance has a significant impact on reconfiguration performance. Using a barrier synchronization method based on PCT-supported connections yields a reconfiguration primitive capable of reconfiguring the entire array over 7,800 times per second, and by using a faster barrier synchronization scheme, that number can be driven even higher.

5.3 Task creation

A parent task can create child tasks within its cell allocation. When a child is started, parent execution on the child task's cells is suspended until the child task ends. Child tasks may communicate with each other via "external connections". Children located within the same parent may create their own “external connections”; children located within different parent tasks must “inherit” the appropriate external connections from their parents.

Consider a parent task's child tasks to be vertices of a graph. Let external connections between these child tasks be represented as edges joining the vertices representing those tasks. Thus, two vertices are joined by an edge if and only if there is an external connection between the tasks they represent. These vertices and edges form connected components, whose "size" is equal to the number of all the cells in each of the connected child tasks.

When created, these connected child tasks require a barrier with a membership of all the cells belonging to the connected component; separate barriers for each task are not adequate. Because connected child tasks communicate, one child task could conceivably start before another. The connected component barrier prevents this unwanted early data arrival.

Reconfiguring a single task requires only local barrier synchronization once the task is running, but starting the task requires the participation of all cells belonging to the larger connected component.

As long as the child task cell allocations are disjoint, deadlock cannot occur on task creation.

5.3.1 Task creation model

Creating a task requires the following operations on each cell of the child task:

```
 Do the overhead operations necessary for switching task contexts:
      Push the parent task info onto a stack;
      Look up the barrier information for the task being created;
```

```
     Look up the external connection information for the task
     being created;
     Set the task's internal connection pointers to the correct
       structure representing the "Connection-to-PCT" mapping they
       should be using.

  Perform a reconfiguration as shown in Section 5.2:
     barrier_sync(Connected_component);
     Turn off global events;
     barrier_sync(Connected_component);
     reconfigure switch(Connected_component's external
                        connections);
     barrier_sync(Connected_component);
     Turn global events back on;
```

Task create (or **task start**) is essentially the reconfiguration primitive (applied to a larger connected component rather than an individual task), prefixed by some additional information table look-up and pointer-reassignment.

The execution model is nearly the same as the reconfiguration primitive with only the addition of the task-change overhead. Each part of the task-changing operation occurs in a fixed amount of time: pushing the parent task info, looking up the barrier and connection information, and setting the pointer values for the child task. All the connection information and barrier information is pre-computed at link time; at runtime the information is retrieved via a predictable-time indexed table look-up. The execution model is then:

Task_start_time = task_change_overhead_time + reconfiguration(N)

Note that in the above equation, N is the number of cells in the total connected component, not just the number of cells in the task.

5.3.2 Measured performance and predictions on iWarp

Task change overhead was measured as 130 clocks.

Figure 5.3 shows the predicted task creation times, based on a measured task change overhead of 130 clocks and the measured reconfiguration times from Figure 5.1, and then compares them to the actual, measured creation times. As expected, performance is similar to that of Figure 5.1, plus some additional overhead.

	4 cells	8 cells	16 cells	32 cells	64 cells
creation time	575	725	1000	1550	2681
(predicted)	(574)	(728)	(978)	(1555)	(2681)
max	576	736	1008	1560	2688
min	568	704	992	1544	2680

Figure 5.3 Child task start times (in clocks) vs. connected-component size (in cells) (using 1D ring barrier), 1000 runs. Measurement error 8 clocks.

5.3.3 Task creation conclusion

The iWarp RTS is minimal, supporting only a single foreground process at a time. Only a small amount of state needs to be stored when creating a child task, and this can be done in a fixed amount of time. The task creation primitive is constructed by combining this process context switch with the reconfiguration primitive from Section 5.2. Because the process context switch can be quickly done in predictable time, and Section 5.2 established that a communication context switch can be quickly done in predictable time, one can build a task creation primitive that is both fast and predictable (to within 1.5 microseconds).

5.4 Task end

Once a child task completes, it cedes control back to the parent. If the child task has external connections to other child tasks, it must ensure that those other tasks have completed as well before it terminates. This is necessary to prevent the external connections from being destroyed until both ends of the connection agree that the connection is no longer needed. Thus, the same connected component that synchronized on task creation will again synchronize on the task end.

No actual reconfiguration needs to be done at the end of a task; control is merely returned to the parent task. Only one barrier is required at the end of a task, and no reconfiguration is needed, so the execution time is therefore both predictable and repeatable.

5.4.1 Task end model

The task end is functionally very simple: a barrier ensures all connected component cells are done, parent task information is restored from a stack, and control is returned to the parent. No additional reconfiguration is needed at the task end because a child is not allowed to disturb an existing parent’s connection when the child was invoked. Because the child couldn’t disturb the parent’s connections, there is nothing of the parent’s state that must be restored when the child terminates.
Appropriate pseudocode for **task end** is simply:

```
   barrier_sync(connected_component);
   restore parent task info from stack;
```

and the execution model is simply:

task_end_time = barrier_time(N) + parent_task_restoration_time

where N is the size of the connected component, and barrier_time(N) is the barrier execution time from Figure 4.21.

5.4.2 Measured performance and predictions on iWarp

Parent_task_restoration_time was measured as 70 clocks.

Figure 5.4 shows the predicted and measured "task end" times using the model of 5.4.1, and the barrier prediction information from Figure 4.21.

	4 cells	8 cells	16 cells	32 cells	64 cells
task ending time	165	216	310	489	870
(predicted)	(167)	(218)	(312)	(493)	(870)
max	168	216	312	496	872
min	152	216	304	488	864

Figure 5.4- Child task ending times (in clocks) vs. connected-component size (in cells) for 100 runs.

5.4.3 Task end conclusions

Because ending a task simply implies returning control to the parent, little work beyond a simple barrier is needed. Predictable task termination simply requires a predictable barrier.

5.5 Chapter summary

Several key points emerge from this chapter. First of all, the four TCS primitives can be built so that they run in a predictable manner. Locally reconfigurable connections, with predictable reconfiguration time, combined with fast, predictable barrier synchronization, are sufficient for implementing the four TCS primitives. The four TCS primitives have a simple, hierarchical construction, and imply a particular construction order. Barrier synchronization is needed for connection reconfiguration and task end, and connection reconfiguration is needed for task creation.

Perhaps not as obvious is the fact that connections with local, directly writable state make predictable connection reconfiguration very easy to implement, as opposed to source routed connections. Source-routed connections require a set-up dependent upon the both the number of connections being set-up/torn-down, as well as the runtime bandwidth utilization (since connection set-up/take-down latency is increased if the link is heavily loaded). Because of this uncertainty with source-routed connections, connections with source routing alone are not sufficient to enable predictable reconfiguration times, as reconfiguration time becomes dependent on runtime network load.

This chapter demonstrated that the TCS primitives have predictable performance in isolation. The next chapter will show how they can be assembled to create tasks that maintain predictable communication performance for a variety of communication patterns.

Chapter 6 - TCS Validation - Communication Patterns

6.1 Introduction

Chapter 3 showed that both message-passing and TCS connections can be accurately characterized in isolation; however, real-world applications involve multiple, potentially interacting communications. In this section, three representative communication patterns are modeled and implemented using both TCS connections and deposit message-passing. We use deposit message-passing because it offers the best performance of all message-passing systems available on iWarp. These patterns are scatter/gather, reduction/broadcast, and all-to-all.

Scatter/gather serializes all communication and thus provides predictable performance with both models. Reduction/broadcast involves a small amount of parallel communication; as a consequence, message-passing loses some of its predictability. This loss of predictability is shown to be the result of spurious runtime link congestions, and by carefully re-mapping the processors so runtime link congestion is avoided, the reduction/broadcast using message-passing again becomes predictable. All-to-all is a massively parallel communication pattern. While deposit message-passing loses its predictability, TCS maintains both a high degree of predictability (within a few percent) while offering substantially better performance than message-passing for large transfers.

6.2 Scatter/gather

Scatter/gather consists of:

(1) one cell sending data to all other cells;
(2) a barrier;
(3) one cell receiving data from all other cells.

This simple pattern is easily modeled because all sends are serialized (because only one cell is sending), and all incoming data goes to a single cell (so reception becomes serialized as well). If one cell is sending to N-1 other cells, the scatter/gather time is just the time to do N-1 sends, a barrier, and N-1 receives. Although all cells are trying to send during the gather phase, only one incoming message can be processed at a time by the gathering cell, and hence no parallel messaging occurs in this pattern.

6.2.1 Scatter/gather - message-passing

A simple model is sufficient to predict pattern execution time for deposit message-passing. The predicted pattern time is :

```
mp_predict = Constant_Overhead +
     ((Number_of_cells - 1) x (Msg_Send_Time + Avg_Net_Latency +
                         (Msg_size / Network_BW))) +
       Barrier_time +
     ((Number_of_cells - 1) x (Msg_Recv_Time + Avg_Net_Latency +
                         (Msg_size / Network_BW)))
```

where
Constant_Overhead was measured at 100 clocks;
Msg_Send_Time was measured at 359 clocks;
Msg_Recv_time was measured at 230 clocks;
Avg_Net_Latency is 20 clocks (avg 4 hops on a torus x 5 clocks/hop);
Network_BW = 1.998 bytes/clock.

Figure 6.1 shows the predicted and measured message-passing scatter/gather times. Because the communication is serialized, congestion is avoided, and predictability (and good performance) are maintained. Because 4-byte transfers have a substantially different communication implementation (no spools used), all patterns will be evaluated using 16 bytes or more. This approach keeps the comparisons between TCS and message-passing fair.

bytes	4 cells	8 cells	16 cells	32 cells	64 cells
16	2433	4845	9643	19299	38445
(predict)	(2407)	(4907)	(9907)	(19907)	(39907)
max	2436	4848	9648	19300	38448
min	2432	4844	9640	19296	38444
64	2551	5136	10266	20594	41093
(predict)	(2551)	(5243)	(10627)	(21395)	(42931)
max	2552	5140	10268	20596	41096
min	2548	5136	10264	20592	41092
256	3082	6379	12942	26061	52206
(predict)	(3127)	(6587)	(13507)	(27347)	(55027)
max	3092	6380	12948	26064	52208
min	3080	6376	12940	26060	52204
1024	5375	11735	24408	49791	100475
(predict)	(5431)	(11963)	(25027)	(51155)	(103411)
max	5376	11748	24416	49796	100504
min	5372	11724	24404	49760	100444
4096	14602	33265	70542	145101	294126
(predict)	(14659)	(33495)	(71167)	(146511)	(297199)
max	14608	33268	70548	145104	294128
min	14600	33260	70540	145100	294124

Figure 6.1 Scatter/gather message-passing time (in clocks), vs. size and number of cells for 1000 runs.

6.2.2 Scatter/gather - TCS Connections

A simple model is adequate for modeling connection-based scatter/gather communication time as well, and gives us a prediction within one or two percent. The major error source is that the added latency per phase (due to the connection distance) really isn't an average, but a worst case. Furthermore, at the start of each gather phase, all N-1 cells returning data try to send at once; this eagerness causes a momentary link congestion that retards the first returned data. This brief effect can quintuple the

bytes	2x2 cells	4x4 cells	8x8 cells
16	1113	4204	35150
(predict)	(1115)	(4146)	(35222)
max	1120	4296	35152
min	1112	4200	35144
64	1257	5032	38184
(predict)	(1259)	(4866)	(38246)
max	1264	5120	38184
min	1256	5032	38168
256	1848	7984	51519
(predict)	(1835)	(7746)	(50342)
max	1848	8072	51520
min	1848	7984	51496
1024	4144	19504	99636
(predict)	(4142)	(19266)	(98726)
max	4144	19592	99640
min	4144	19504	99616
4096	13360	65584	293171
(predict)	(13367)	(65406)	(292514)
max	13360	65672	293176
min	13360	65584	293152
16,384	50232	249904	1067315
(predict)	(50268)	(249906)	(1067414)
max	50232	249992	1067320
min	50232	249904	1067296

Figure 6.2 Scatter/gather TCS connection times (in clocks), vs. payload size and number of participating cells for 1000 runs.

expected latency for the first gather. The effects of this momentary "latency explosion" are limited to moderate size transfers: for small transfers there isn't enough data to cause the congestion, and for large transfers the network transfer time swamps all other factors.

With this caveat, the predicted completion time can be modeled as:

```
predict = Constant_Overhead +
          ((Number_of_cells-1) x 2 x (Avg_Net_Latency +
                            (Msg_size / Network_BW))) +
        (Reconfiguration_time x Number_of_phases)
```

where
Constant_Overhead was measured at 398 clocks;
Network_BW = 1.998 bytes/clock;

```
Avg_Net_Latency = 38 clocks for 2x2,
                  59 clocks for 4x4,
                  66 clocks for 8x8;

Reconfiguration time = 444 clocks for 2x2,
                       872 clocks for 4x4,
                      2551 clocks for 8x8.
```

(these are the measured average reconfiguration times from Section 5.2)

Figure 6.2 shows the measured vs. predicted pattern communication times for connections, with the predicted times based on the measured reconfiguration times in Chapter 5. As with messages alone, the TCS connection-based implementation outperforms message-passing best for small transfers on small arrays. For large transfers (in this mostly congestion-free pattern) the transfer time is dominated by the link bandwidth, and both message-passing and TCS connections yield similar performance results.

6.2.3 Scatter/gather conclusions

Because the pattern is inherently serial, both message-passing and TCS connections show predictable performance with this communication pattern, with TCS offering significantly better performance with the smaller-sized transfers.

6.3 Reduction/broadcast

Reduction occurs in $\log_2(N)$ stages, with half as many cells sending in each stage as in the previous stage. Broadcast is the reverse of reduction, requiring another $\log_2(N)$ stages. Parallel messaging occurs in all but the last stage of reduction, and all but the first stage of the broadcast. This parallel messaging creates link congestion for message-passing that causes it to lose predictability. By carefully remapping the processors to eliminate congestion the predictability can be regained; this shows that the loss of predictability is due to the unknown link congestion. TCS maintains its predictability (and performance) throughout.

6.3.1 Reduction/broadcast using message-passing

Because the scatter/gather communication pattern focused all communication through a single cell, all sends and receives were serialized, and performance can be predicted with a fairly simple model. In constrast, the reduction/broadcast communication pattern involves different numbers of cells sending and receiving during each stage, resulting in differing available network bandwidths. Message-passing loses some of its predictability because link congestion becomes unpredictable, yielding uncertain link bandwidths for the different stages. Unless one has special knowledge of the underlying network and cell mapping, only in the final reduction stage and the first broadcast stage can the link bandwidth be known with absolute confidence (100%). None of the other stages can be absolutely known.

The predicted performance model is as follows:

mp_predict = Constant_Overhead +
(2 x Log$_2$(Number_of_cells) x (Per_Iteration_Overhead +
Msg_Send_Time +
Avg_Net_Latency +
(Msg_size/Network_BW)) +
Barrier_time;

where
Constant_Overhead measured at 100 clocks;
Msg_Send_Time measured at 359 clocks;
Per_Iteration_Overhead measured at 40 clocks;
Avg_Net_Latency (for torus routing) is 20 clocks;
Network_BW = 1.998 bytes/clock (assume full network bandwidth even

though this probably isn’t true for any stage except the last reduction stage and the first broadcast stage);
Barrier_time = 448 clocks regardless of the number of cells.

Note that a different barrier is being used here between the stages. For ease of implementation, the 2D 64 cell (global) barrier from Figure 4.22 (last line in the table) is used which runs in 448 clocks.

Figure 6.3 shows the predicted vs. measured performance number for the message-passing-based reduction/broadcast. As can be seen, with large numbers of cells and large transfer sizes, the predictive power is lost; with 1Kbyte transfers and 64 cells, the predictions are off by almost a factor of 2. This loss of

bytes	4 cells	8 cells	16 cells	32 cells	64 cells
16	2321	4071	4901	6125	7567
(predict)	(2140)	(2994)	(3848)	(4702)	(5556)
max	2328	4072	4952	6200	7624
min	2040	3608	4472	5416	6584
64	2463	4276	5246	6583	8087
(predict)	(2236)	(3138)	(4040)	(4942)	(5844)
max	2464	4448	5432	6680	8088
min	2128	3928	5072	5904	7656
256	3031	5644	6824	8473	10908
(predict)	(2620)	(3714)	(4808)	(5902)	(6996)
max	3040	5728	7048	8488	11064
min	2848	5616	6488	8232	10896
1024	5327	10604	12555	15756	20891
(predict)	(4156)	(6018)	(7880)	(9742)	(11604)
max	5328	10608	12560	16480	22448
min	5200	10600	12424	15048	19352
4096	14549	30681	35590	45015	60230
(predict)	(10308)	(15246)	(20184)	(25122)	(30060)
max	14560	32960	37864	47296	61112
min	14416	28408	33344	42544	58616

Figure 6.3- Reduction/broadcast message-passing time (in clocks), vs. payload size and number of cells for 1000 runs, link bandwidth unknown.

predictability is due to unknown link congestions; this can be demonstrated by utilizing special knowledge of the message-passing routing engine to map the problem to cells so that no pathway contention occurs within any stage. This contention-free mapping guarantees 100% net bandwidth during each stage. Using the same program but this alternate processor mapping, Figure 6.4 shows that the predictions regain their accuracy.

bytes	4 cells	8 cells	16 cells	32 cells	64 cells
16	1958	2803	3682	4507	5383
(predict)	(2140)	(2994)	(3848)	(4702)	(5556)
max	1976	2808	3696	4512	5384
min	1952	2744	3664	4464	5336
64	2085	2967	3865	4747	5629
(predict)	(2236)	(3138)	(4040)	(4942)	(5844)
max	2088	2968	3872	4752	5632
min	2048	2944	3856	4704	5576
256	2465	3553	4649	5703	6789
(predict)	(2620)	(3714)	(4808)	(5902)	(6996)
max	2472	3560	4656	5704	6800
min	2424	3456	4568	5616	6696
1024	3999	5839	7701	9512	11368
(predict)	(4156)	(6018)	(7880)	(9742)	(11604)
max	4000	5840	7712	9528	11384
min	3968	5760	7624	9464	11280
4096	10145	15073	20003	24913	29830
(predict)	(10308)	(15246)	(20184)	(25122)	(30060)
max	10152	15080	20008	24920	29832
min	10136	14976	19920	24824	29744

Figure 6.4- Reduction/broadcast message-passing time (in clocks), vs. payload size and number of cells for 1000 runs, remapped to avoid link congestion

6.3.2 Reduction/broadcast using TCS connections

As with message-passing, the connection model for the reduction/broadcast communication pattern is a bit more complex than the scatter/gather. 2 x $\log_2(N)$ stages are required: $\log_2(N)$

reduction stages, and $\log_2(N)$ broadcast stages. Each reduction stage has half as many cells sending/receiving as the previous stage, and each broadcast stage has twice as many sending/receiving cells as the previous stage. The execution time for the whole exchange can be modeled as follows:

```
  Connection_predict = Constant_Overhead +
            (Log2(Number_of_cells) x (Per_iteration_overhead +
                                (2 x (Avg_Net_Latency +
                                   (Msg_Size/Network_BW)))))  +
          (Reconfiguration_time x Number_of_phases)
```

```
Where Constant_Overhead = 302 clocks;
Avg_Net_Latency = 38 clocks for 2x2,
                  59 clocks for 4x4,
                  66 clocks for 8x8;
Reconfiguration_time = 441 clocks for 2x2,
                       869 clocks for 4x4,
                      2550 clocks for 8x8;
Network_BW = 1.998 bytes/clock.
```

Figure 6.5 compares the measured to the predicted communication times. Again, a fairly simple communication model is being used (assuming average latencies, no link congestion, etc); still, even at this level TCS maintains good predictability. Using the measured reconfiguration times from Chapter 5 allows accurate prediction (within two percent) of execution time with this more complex communication pattern. In essense, TCS has traded message-passing's unknown link congestion for a known (programmer-visible and linker-visible) node congestion while maintaining better performance.

6.3.3 Reduction/broadcast conclusions

While TCS connections still provide predictable performance, message passing loses its predictability when the link congestion becomes unknown. If extraordinary steps are taken (with regards to placement and routing) to make the link congestions known, message passing can regain its predictability at this level. TCS, on the other hand, maintains its predictability with no extra effort.

bytes	2x2 cells	4x4 cells	8x8 cells
16	908	1687	3801
(predict)	(927)	(1715)	(3752)
max	912	1688	3808
min	904	1664	3712
64	1003	1879	4089
(predict)	(1023)	(1907)	(4040)
max	1008	1880	4096
min	992	1856	3992
256	1388	2647	5241
(predict)	(1407)	(2676)	(5193)
max	1392	2648	5248
min	1384	2632	5152
1024	2924	5719	9848
(predict)	(2945)	(5751)	(9806)
max	2928	5720	9856
min	2920	5696	9760
4096	9067	18007	28281
(predict)	(9095)	(18051)	(28256)
max	9072	18008	28288
min	9064	17984	28192
16,384	33643	67159	102009
(predict)	(33695)	(67252)	(102058)
max	33648	67160	102016
min	33640	67136	101912

Figure 6.5 Reduction/broadcast times (in clocks) using TCS connections vs. payload size and number of cells for 1000 runs.

6.4 All-to-all communication

Personalized all-to-all communication requires each cell to send a distinct message to every other cell, and each cell to receive a message from every other cell. Thus, N cells generate Nx(N-1) sends and receives. Unlike the reduction/broadcast pattern, no simple remapping exists that can eliminate this pattern’s congestion.

6.4.1 Message-passing implementation

The predictive model for all-to-all communication using message-passing is similar to the previous two communication patterns. To prevent obvious congestions (such as every cell trying to send to cell 0 first, leading to massive congestion at cell 0 and little traffic elsewhere, followed by every cell trying to send to cell 1, etc.), each cell instead sends data using a randomly-ordered schedule. Because message passing allows (in theory) everyone to send and receive at once, one expects to be able to model the cost of an all-to-all exchange among N cells as (N-1) sends, (N-1) receives, program loop overhead, and a barrier at the end.
The predictive model is therefore just:

```
mp_predict = Constant_Loop_Overhead +
  (Number_of_cells-1) x (Per_Iteration_Overhead + Msg_Send_Time +
                        Avg_Net_Latency +(Msg_Size/Network_BW)) +
  (Number_of_cells-1) x (Msg_Recv_Time + Avg_Net_Latency +
                            (Msg_Size/Network_BW)) +
  Barrier_time(Number_of_cells);
```

```
where
Constant_Loop_Overhead measured at 100 clocks;
Msg_Send_Time measured at 359 clocks;
Msg_Recv_Time measured at 230 clocks;
Per_Iteration_Overhead measured at 40 clocks;
Avg_Net_Latency (for torus routing) is 20 clocks;
Network_BW = 1.998 bytes/clock;
Barrier_time = 77 clocks for 2x2,
              127 clocks for 2x4,
              219 clocks for 4x4,
              404 clocks for 4x8,
              780 clocks for 8x8.
```

As with the reduction/broadcast pattern, unknown link congestion makes reliable message-passing predictions difficult. Figure 6.6 shows the predicted vs. measured execution times for varying data sizes and numbers of cells. For small numbers of cells (4 or 8 cells) and small exchanges (16 to 64 bytes), predictions remain within a factor of two. As the problem size scales up, by the

bytes	4 cells	8 cells	16 cells	32 cells	64 cells
16	2837	6365	13365	28373	60269
(predict)	(2072)	(4782)	(10194)	(21019)	(42675)
max	3424	7752	14984	30632	63000
min	2528	5768	12432	26592	57680
64	3088	7082	14956	34957	75197
(predict)	(2216)	(5118)	(10914)	(22507)	(45699)
max	3832	8144	16792	38664	82760
min	2760	6344	13992	31416	71224
256	4044	9408	20403	54229	121200
(predict)	(2792)	(6462)	(13794)	(28459)	(57795)
max	4984	11008	23040	59760	133120
min	3608	8184	18464	48816	114976
1024	7612	18441	41221	119557	279247
(predict)	(5096)	(11838)	(25314)	(52267)	(106179)
max	9096	23112	46408	129328	296944
min	6336	15712	36872	110184	264488
4096	22112	54369	125301	384586	922385
(predict)	(14324)	(33370)	(71454)	(147623)	(299967)
max	26976	69112	146600	428480	974512
min	16848	45472	112152	359008	874536
16,384	80240	198183	460277	1457334	3491392
(predict)	(51224)	(119470)	(255954)	(528923)	(1074867)
max	101424	256624	518296	1595544	3477144
min	66024	162664	409808	1353016	3274592
32,768	157033	392011	907588	2880807	6932091
(predict)	(100424)	(234270)	(501954)	(1037323)	(2108067)
max	195816	477376	1044800	3274680	7353728
min	121392	333128	796152	2640976	6507808

Figure 6.6- All-to-all (randomized schedule) message-passing time (in clocks), vs. payload size and number of cells for 250 runs.

time one is doing 16 Kbyte or 32 Kbyte transfers with 64 cells, the predictions are off by more than a factor of three.

6.4.2 All-to-all communication using TCS connections

Supporting the all-to-all pattern using TCS connections proved a bit more difficult than the other two patterns because its complexity exposes some of the eccentricities of the iWarp link scheduler (unfair forward bandwidth and DQ congestion - introduced in Chapter 3). Because these eccentricities can be modeled, they can still be accounted for in performance predictions at the expense of a more complicated model.

Two types of congestion are present: forward link congestion, and "DQ congestion". While the hardware can be coaxed to provide fair forward link bandwidth by careful choice of PCT assignments, DQ congestion greater than 2 is not handled fairly. Feeding data into a link suffering from DQ congestion may result in unpredictable bandwidth. Fortunately, if one "throttles" the rate at which data is fed into a congested link so that the rate does not exceed the DQ congestion, then bandwidth remains predictable. For example, if a link has a DQ congestion of "3" and a forward congestion of "2", things will work reliably provided no connection through that link is fed at a rate greater than 1/3 link bandwidth. For links with a forward congestion of "1" and a DQ congestion of "2", no throttling is necessary; the bandwidth can be modeled as either full bandwidth (where DQ congestion=1) or half-bandwidth (where DQ congestion=2).

For each array size (2x2, 4x4, 8x8) a set of communication patterns was created using the "dragon router" AAPC code as explained in [25]. The TCS communication linker analyzed those routes to determine forward link congestion and DQ congestion in each phase (something that is impossible to do with any message-passing system as the message-passing model precludes a global view of the machine’s communication state). Based on the linker's analysis, DQ congestion was detected in half of the 4x4 phases, and in all of the 8x8 phases. In the 4x4 case, the DQ congestion was only 2; no throttling was needed to maintain fairness. The model was updated to reflect a forward link bandwidth of either full bandwidth or half-bandwidth, depending on whether DQ congestion is present or not. For the 8x8 case, it is necessary to throttle all links down to 1/3 bandwidth to maintain scheduling fairness (and thus predictability).

Additionally, a more complicated communication scheme is used in this pattern to allow simultaneous sending and receiving: all cells do foreground sends (throttled where necessary), and background receives (spooling - Section 3.1.3). With the previous communication patterns (scatter/gather and reduction/broadcast), cells would only send **or** receive at each stage of the communication pattern so message passing’s ability to send and receive at once didn’t offer an unfair advantage. With all-to-all communication, cells can ideally be sending and receiving simultaneously. To prevent an unfair 2x memory bandwidth advantage to message-passing (which would render our comparisons meaningless), “spooling receives” were added to the TCS implementation. While all communication still uses connections, data is put into the connection via a foreground send, and received from the connection via a background receive. This allows a more fair performance comparison between the two models.

As would be expected, the performance model for all-to-all communication using foreground and background connections (with throttling) is a bit more complicated than earlier models. The current model involves the sum over all communication phases of quantities that can vary in each phase, but all are known or can be calculated by the communication linker. The model is

```
predict = 10; /* constant overhead */
for each communication phase {
 predict += maximum_connection_path_latency_in_phase /*determined by
linker*/
 predict += Msg_size / Network_BW_in_phase ; /* determined by linker
                                           and throttling needed*/
 predict += Barrier_time + Reconfiguration_time + Spool_setup +
             other_overhead;
}
```

Barrier time was measured as 432 clocks;
Reconfiguration time was measured as 162 (using a faster reconfiguration routine than presented in Section 5.2);
Spool_setup + Other_overhead was measured at 459.

bytes	2x2 cells	4x4 cells	8x8 cells
16	3384	17054	68951
(predict)	(3762)	(18426)	(75130)
max	3392	17088	68984
min	3384	17016	68920
64	3507	17603	73422
(predict)	(3870)	(19098)	(79738)
max	3512	17640	73864
min	3504	17536	73384
256	3938	20074	91860
(predict)	(4302)	(21786)	(98170)
max	3944	20128	92248
min	3936	20016	91808
1024	5666	30823	165587
(predict)	(6030)	(32538)	(171898)
max	5672	30824	166016
min	5664	30816	165536
4096	12578	73851	460496
(predict)	(12942)	(75546)	(466810)
max	12584	73856	460912
min	12576	73840	460448
16,384	40226	245810	1640150
(predict)	(40590)	(247586)	(1646394)
max	40232	245816	1640536
min	40224	245808	1640088
32,768	77090	475204	3212189
(predict)	(77454)	(476978)	(3219194)
max	77096	475232	3212584
min	77088	475200	3212152

Figure 6.7 All-to-all communication time for TCS connections (in clocks), vs. payload size and number of cells for 1000 runs.

Figure 6.7 shows the predicted and measured performances for a wide variety of transfer sizes. In many cases the predicted values are within one percent of the measured values and, performance-wise, usually a factor of 2 better than deposit message-passing. For example, the 64 cell 32 Kbyte transfer takes more than 6.9 million clocks under message-passing, but runs in just a little over 3.2 million clocks with TCS.

6.4.3 All-to-all conclusions

While TCS requires a more sophisticated performance model (involving different latency and bandwidth values for each phase), the linker can easily generate the required analysis and still make useful predictions of runtime performance, usually to within a tenth of a percent. The message passing model has no good way to account for link congestion, and its performance suffers accordingly. Unlike the scatter/gather, no simple remapping exists for the all-to-all pattern to completely avoid link congestion.

6.5 Chapter summary

This chapter demonstrates that the TCS tasking primitives can be assembled to construct tasks with complex communication patterns on real hardware while maintaining predictable performance. On unloaded hardware, both message-passing and TCS connections offer predictable performance; scatter/gather was predictable when implemented with both deposit message-passing and with TCS. As communication patterns become more complex, though, message passing hides the increased complexity as an unknown link congestion, making reliable performance predictions difficult, if not impossible. The reduction/broadcast implementations showed deposit message-passing losing its predictability as link congestion increased; it could only regain predictability by an explicit re-mapping that avoided link congestion. The all-to-all message-passing pattern was beyond simple modeling, whereas the TCS implementation remains predictable, worst-case, to within 10% (overestimate) of execution time. TCS exposes both link congestion and node congestion to the programmer and to the linker, allowing accurate performance predictions to be made.

Furthermore, by exposing the communication to the linker, a TCS implementation can account for known system eccentricities. For instance, the prototype TCS linker for iWarp identifies potential DQ bandwidth problems and recommends the appropriate throttling strategy where needed to guarantee scheduling fairness, maintaining its predictive capabilities.

Finally, regardless of any hardware idiosyncracies, deposit message-passing is restricted to a local, run-time view of array congestion, while TCS maintains a more global view of the communication state at link time. TCS can thus choose routes based on information that message-passing won’t discover until runtime. This information allows TCS to use globally optimal routing; message-passing only has a local view of run-time congestion at runtime and hence cannot perform global optimizations. Thus, even for complex communication patterns, TCS offers both better performance and better predictability than deposit message-passing.

Chapter 7 -
TCS Validation - Hierarchical Tasking

7.1 Introduction

Chapter 3 showed that connection-based communication can be done in a fast, predictable manner. Chapter 4 used predictable connections to create fast barriers with execution times predictable to within a microsecond. Chapter 5 used the predictable barriers to create predictable task control primitives: connection reconfiguration, task create, and task end. Chapter 6 showed how those primitives can be combined with connections to create tasks having complex communication patterns that still maintain predictable execution time. This chapter shows how to compose more complex TCS tasks by assembling simpler TCS tasks while still maintaining predictability.

TCS's dynamic tasking allows modular, continuous-flow, predictable low-latency computations within a parallel application. Fast barrier synchronization enables predictable task control at a level of microseconds. The ability to construct complex tasks from simpler tasks allows low-level complexity to be hidden from higher level tasks. This encapsulation allows a modular approach to application development as well as enabling reuse of functional task blocks from one application in another.

This chapter demonstrates hierarchical tasking by giving an example: constructing the real-time video motion-detector introduced in Chapter 1 as part of the “predict a thrown ball’s trajectory” application. Incoming pixels are compared against a weighted moving average, then those new values are used to update the average. The output is a video-rate binary image with an output pixel value of 1 if the new value exceeds the moving average by 20 or more, and an output pixel value of 0 otherwise. This task will be implemented as a collection of smaller, predictable tasks; the performance of the larger composite task then is shown to be predictable based upon the known performances of the smaller component tasks.

7.2 Implementing the motion-detector

This section discusses the performance requirements for the motion-detector, the TCS module design of the motion detector, and the testing setup for the module.

7.2.1 Requirements

When coded in a mixture of C and assembler and compiled for iWarp, the motion-detection algorithm requires 28 clocks-per-pixel if no motion is detected at the pixel, and 33 clocks-per-pixel if motion is detected. Each incoming video frame arrives as 240 lines of 256 pixels per line. Because the data is sampled at video rates, incoming data arrives in bursts of 256 pixels in 51.2 microseconds during each 66.7 microsecond scanline, with a pause of roughly 600 microseconds occurring between frames. To maintain video processing rates the motion detector needs to handle (240 x 256 = 61,440) pixels within no more than 16,666 microseconds.

Given a worst-case processing time of 33 clocks-per-pixel, 61,440 pixels would require 2,027,520 clock cycles worst case. Given that the CPU executes 20 clocks per microsecond, a single processor only has 333,333 clock cycles per video frame time. To meet the processing requirements, at least 2,027,520/333,333 = 6.1 processors (which rounds up to 7) are required.

The amount of memory needed per processor also needs to be considered. At a minimum, storage is required for the fresh pixel data, the moving average, and the output image. For 61,440 pixels, this is 3 x 61,440 = 184,320 bytes of storage, which easily fits within a single iWarp cell.

7.2.2 Utilizing multiple processors

At first glance, the simplest multiprocessor implementation would seem to be to just distribute each incoming video image to a separate cell. Not only does this introduce a rather high latency (since no computation occurs until the whole frame is first read in), but this approach also makes it difficult to maintain an accurate moving average for the comparison. Instead, the data needs to be distributed among the processors so that the same portion of each successive video frame arrives at the same processor each time.

By distributing the data in a block-cyclic fashion, information locality from one frame to the next can be maintained across the different processors. The module has two special cells which act as "gatekeepers": one for the fresh video data flowing into the module, and the other for detector output video leaving the module (See Figure 7.1). The remaining module cells are "workers", which are given data by the incoming gatekeeper, and send their output to the outgoing gatekeeper. Data is distributed and gathered in fixed blocksizes, with the number of distribution/gather cycles necessary per-frame determined by the blocksize.

7.2.3 The TCS implementation

While the “back-of-the-envelope” calculations in 7.2.1 mandated a minimum of seven cells to meet performance goals, additional time is needed for data distribution and gathering. If the work were partitioned such that seven cells were working at any one time, with one cell loading new data and another cell writing old results, a total of nine worker cells would be needed. To simplify the implementation by maintaining worker symmetry, the number of worker cells was rounded up to ten. The **detect motion** task is implemented roughly as follows:

```
detect_motion(NUMBER_OF_FRAMES, BLOCKS_PER_FRAME) {
 for (i=0; i<NUMBER_OF_FRAMES; i++) {
  switch(cell type) :
   case WORKER:
    id=WORKER cell ID
    for (j=0; j<BLOCKS_PER_FRAME; j++) {
      taskstart(FILL(id));
      taskstart(COMPUTE);
      taskstart(DRAIN(id));
    }break;
   case GATEWAY_IN:
    for (j=0; j<BLOCKS_PER_FRAME; j++) {
      taskstart(FILL(j mod NUMBER_OF_WORKERS))
    }break;
   case GATEWAY_OUT:
    for (j=0; j<BLOCKS_PER_FRAME; j++) {
      taskstart(DRAIN(j mod NUMBER_OF_WORKERS))
    }break;
  } /* switch */
 } /* for NUMBER_OF_FRAMES */
taskend();
```

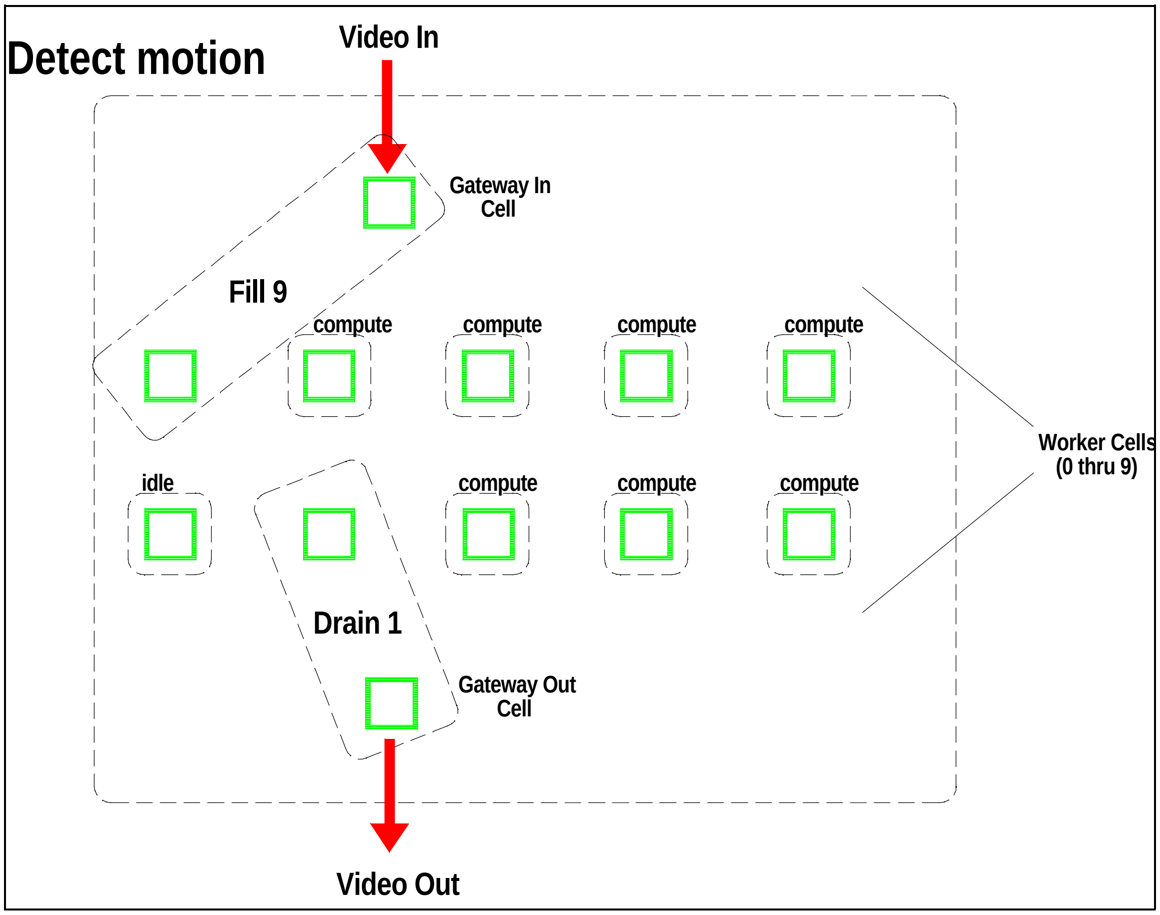


Figure 7.1 The "detect motion" task is implemented using multiple small, dynamic tasks.

Because ten worker cells are sharing the load, and there are 240 lines of video per frame, each cell is responsible for processing 24 lines of video per frame. 24 lines of 256 pixels implies between 172,032 and 202,752 clocks of processing, plus the drain and fill times, per 333,333 clocks (frame time). Thus, the worker efficiency will be between 51% and 62%. If a means existed to somehow ensure there would be <u>no</u> phase variance between drain rates and fill rates, the filling and draining could be bundled into a single module, yielding a reduction of the number of workers to eight cells. This would imply 30 lines of video per cell, for 236,544 to 253,440 clocks, boosting efficiencies to between 70% and 76%.

For the ten worker cells implementation, latency as a function of blocksize is predicted as
latency = network_latency + data_loading_latency + compute_time.

The network latency for 10 cells plus two “gatekeeper” cells, using the 5 clocks-per-hop estimate, is 12x5 = 60 clocks. An additional 3 hops are needed to travel from the cell doing the timing to the module and back again, so this adds an additional 15 clocks for a total expected path latency of 75 clocks.

Because the data is coming in as live video samples, it can’t arrive any faster than video rates. A standard NTSC video signal is 30 interlaced frames/second, or 60 non-interlaced frames/second. In either case, a scanline (including sync, leading and trailing blanking (overscan), and pixel data) occurs every 66.8 microseconds, of which only about 51.2 microseconds is active pixel data. If we sample each line at 256 pixels per line, the data arrives as 64 4-byte words over a 51.2 microsecond interval, every 66.8 microseconds. At 20 clocks/microsecond, this works out to 1,336 clocks per scanline; the rate is fixed by the camera. If the module were unable to accept data at this rate, incoming data would be lost.

Computation time is either 28 or 33 clocks per pixel, depending on whether motion is actually detected or not. At 256 pixels per scanline, this means a worst-case compute time of (33*256) = 8,448 clocks per scanline, and a best-case compute time of 7,168.

TCS task start and task end overheads are completely hidden within the “idle” time between scanlines. A scan line lasts 66.8 microseconds, but the active video portion is only 51.2 microseconds, so there are (66.8 - 51.2 microseconds) = 15.6 microseconds => 312 clocks available per scanline.

7.3 Predictions

Because the video detect task has hard realtime input and output, and the output needs low latency, we need to verify both that the task can maintain throughput and meet specified latency constraints.

7.3.1 Throughput

As designed, provided there is sufficient time during the horizontal retrace for task start and task end to execute, the application should be able to support and maintain video-rate throughput regardless of the block size of the data distributed among the cells. Because of the nature of the pipeline, the critical points are at the gateway cells executing the task start(FILL) and task start(DRAIN), and the FILL/DRAIN task ends. As implemented for this module, task start for both FILL and DRAIN occurs in 225 clocks, and task end occurs in 60 clocks. The total time to start and terminate a fill or drain is just (225 + 60)=285 clocks, which is less than the 312 clock limit. Thus, throughput is expected to be sustainable.

7.3.1 Latency

The next issue is latency: the time from when a pixel first enters the ***gateway in*** cell to it leaving the ***gateway out*** cell (See Figure 7.1). Because the fill task start and the drain task end operations are interleaved within the horizontal refresh interval, their execution times don't affect pixel latency. Only fill task end and drain task start, plus computation, plus network latency, affects pixel latency. Best case latency occurs with the smallest possible block size; in this case, one scanline of data (256 pixels).
From previous measurements, computation time is 28 clocks/pixel for no motion, and 33 clocks/pixel for motion.

For a data block size of 1 video line, then, worst-case latency can be predicted by summing the various component worst-case latencies:

network latency - 12 cells x (approximately) 5 clocks/cell = 60 clocks (Chapter 3);
data loading latency (512 pixels @ 40 pixels/clock)= 1280 clocks;
fill task end - 60 clocks (measured);
compute task start - 20 clocks (measured);
compute latency - 7168 (no motion) or 8448 (motion) (computed from single-pixel measurements);
compute task end - 20 clocks (measured);
drain task start - 225 clocks (measured).

Total latency is computed as 60 + 1280 + 60 + 20 + 7168 + 20 + 225 = 8,833 for a line with no motion, and
60 + 1280 + 60 + 20 + 8448 + 20 + 225 = 10,113 clocks for a line with motion detected at every pixel. In reality, latency should be slightly better than this because the drain task start should complete faster due to a slight skew between the compute and the previous worker’s drain. Measured drain task starting times assume (worst case) that both cells entered the start barrier together. In reality, one cell gets there first and waits; when the second cell enters, due to the way the barrier is implemented, the latecomer writes a value to its output PCT, reads a value from its input PCT (which is already waiting from the early arriver), and proceeds. The overhead of the network latency is hidden because the network latency overlaps with the late-arriving cell’s previous operations. Due to barrier skew, the task start completes at the late-arriving cell in less than worst-case time.

For data blocks larger than one scan line, the data loading latency is calculated by multiplying the additional number of scan lines by the time for a complete video scan line (66.8 microseconds = 1336 clocks). Thus, while a single line of video has a data-loading latency of 1280 clocks, additional lines include the overhead of the horizontal blanking interval and hence cost 1336 clocks. Unless one adds additional upstream buffering (bad, because this increases total application latency), one can’t get the per-line data loading latency lower than 1336 clocks.

7.4 Results

Given a 10-cell implementation and a fixed data size, the only variable parameter is the block size of the cyclicly distributed data. The TCS module is implemented to accept the "number of video frames" and the "block size" (in number of lines) as parameters, and is invoked as part of a larger module with varying block sizes as shown in 4.23. As expected, the module latency gets smaller with the smaller block sizes. Note that we can predict (and achieve) latencies of milliseconds with accuracies in the microseconds. Figure 7.2 represents a “best-case” video input, with no motion detectable, and 7.3 is a “worst-case”, with motion detected at every pixel. Note that the “maximum time” for

Scanlines per block	Avg, predicted, max, and min latency to first pixel	Avg, max, and min completion time	Worker Utilization
24	203,040 (204,425) 203,040 203,040	33,533,136 33,533,136 33,533,136	51.5%
12	101,618 (102,377) 101,624 101,616	33,433,898 33,433,904 33,433,896	51.9%
8	67,824 (68,361) 67,824 67,824	33,400,834 33,400,904 33,400,832	52.3%
6	50,912 (51,353) 50,912 50,912	33,384,284 33,384,288 33,384,280	52.5%
4	34.019 (34,345) 34,024 34,008	33,367,755 33,367,760 33,367,744	52.9%
3	25,560 (25,841) 25,568 25,560	33,359,476 33,359,480 33,359,472	53.3%
2	17,119 (17,337) 17,120 17,112	33,351,223 33,351,224 33,351,208	53.9%
1	8,659 (8,833) 10,008 8,640	33,342,940 33,342,944 33,342,936	54.8%

Figure 7.2 Best-case (no motion) latency and completion times (in clocks) for the **detect motion** task vs. block size (in scanlines) for 100 frames of video.

the 1 scanline/block entry in Figure 7.2 (10,008 clocks) is a measurement artifact; the first run of the system was the 1 scanline/block, and that first frame through the system registered

Scanlines per block	Avg, predicted, max, and min latency to first pixel	Avg, max, and min completion time	Worker Utilization
24	235,903 (235,145) 236,296 235,896	33,566,763 33,566,768 33,566,760	61.5%
12	117,953 (117,737) 118,112 117,952	33,450,728 33,450,736 33,450,728	62.1%
8	78,751 (78,601) 78,872 78,744	33,412,054 33,412,056 33,412,048	62.5%
6	59,148 (59,033) 59,232 59,144	33,392,694 33,392,696 33,392,688	62.7%
4	39,544 (39,465) 39,576 39,544	33,373,352 33,373,360 33,373,352	63.1%
3	29,696 (29,681) 29,704 29,696	33,363,680 33,363,680 33,363,680	63.7%
2	19,866 (19,897) 19,872 19,744	33,354,028 33,354,032 33,354,024	64.4%
1	10,005 (10,113) 10,008 10,000	33,344,342 33,344,344 33,344,336	65.5%

Figure 7.3 Worst-case (motion at every pixel) latency and completion times (in clocks) for the **detect motion** task vs. block size (in scanlines) for 100 frames of video noise (simulates extreme motion).

motion at all pixels. In fact, this agrees well with the predicted value for the 1 scanline/block entry in Figure 7.3.

7.5 Chapter summary

While TCS primitives allow construction of single tasks with predictable execution time, the real power of the TCS model is that complex tasks can be created by assembling simpler tasks while still maintaining predictable performance. Because the synchronization barriers can be performed so quickly (on the order of a few microseconds), complex applications can be composed at a very fine level of granularity, which allows very low latency while maintaining high bandwidth. For instance, with the video motion detector task developed in this chapter, motion can be continuously detected at video rates with latencies as low as 7.5 scanlines (10,008 clocks / 1335 clocks/scanline).

By expressing the application (or task) using the TCS primitives, the application's potential runtime communication complexity is exposed to the linker, which can then make globally-optimized communication resource allocations. Because these primitives are built upon a very fast, predictable barrier synchronization implementation, they enable task implementations to achieve a fine level of granularity. Because the control primitives are constructed to give predictable performance, the tasks created using those primitives have predictable performance. Finally, complex tasks can be constructed by assembling simpler tasks into larger structures while still maintaining predictable performance.

Chapter 8 - Related Work

8.1 Chip/Poker (1982)

In the early 1980's Lawrence Snyder introduced the CHiP architecture, a reconfigurable connection-based machine positioned as a SPMD system[39]. Connections were established through off-cell switching elements configured by a separate, global controller. The individual cells executed locally-stored programs, but network reconfiguration was an inherently global operation due to the external, global switch control.
The physical network supported only a single channel per physical link, but fanout configurations were supported.

Poker, the programming language/environment for ChiP, introduced the concept of the "XYZ levels" necessary for effectively programming this sort of machine. When using connections for communication, a distinction needed to made between *program code* (which executed the computations) and *network code* (which defined the communication). Individual algorithms were expressed as a combination of program and network code (the X and Y levels, accordingly), and the full applications was created by combining sets of algorithm implementations together (the Z level) [41].
The Poker programming environment was developed to support program development at all levels. In his words,

> ...what does a whole Poker program look like? Answer: It cannot be seen *in toto.* Unlike "regular" programs, Poker programs are not monolithic pieces of program text. Instead, they are databases[40].

The major drawbacks of the CHiP system were 1) lack of local processor control over network configuration, and 2) lack of virtual channel support. The former enforced a global synchronous tasking model, and the latter made global reconfigurations more expensive, since only a few connections can be supported per configuration.

8.2 GF-11 (1987)

While the IBM GF-11 was actually a SIMD machine rather than a MIMD machine, it is worth mentioning here because it also carried the idea that network connections constituted a separate program from the actual computation program. A special switch controller stored a program of 1024 different network configurations (each requiring 8640 bits) that the main controller could summon at runtime by sending a 10-bit configuration request. Since the machine was SIMD, it only supported a single global task, but it could rapidly reconfigure its connection setup, changing from one configuration to another in only 200 nanoseconds[6].

8.3 Polymorphic Torus (1989)

Li and Maresca introduced a SIMD polymorphic torus machine that combined the network switch with the individual processor. The network switches are dynamically configured under local processor control[32]. In one sense this was a step backward, in that connection configuration must again be explicitly built into the computation program code, but it was also an advance in that it gave the local processors the power to self-configure on demand, freeing the system of the control bottleneck an external configuration agent would imply.

8.4 Transputer-based systems: C_NET and MARC

The INMOS T800 Transputer provides direct, on-chip hardware for connections, but no routing support. A number of different systems have been built combining the T800 chips with an external routing/communication network.

C_NET (1992) supported reconfigurable connections between processors, but relied upon an external agent for reconfigurations. The C_NET programming model is similar to the XYZ levels of Poker; program (computation) code is written for the cells (similar to Poker X-level), a topology definition program is written to define the communication that occurs (similar to Poker Y-level), and phase programs map the program code and the connection definitions to particular cells on the physical machine (Poker Z-level)[1].

MARC (1991) is another programming tool for a networked T800s, but focusses on automatically placing and routing a set of communicating processes than providing programmer support for communication[11].

The T9000 promised support for virtual channels, so that multiple channels could be time-division-multiplexed over a single physical connection, but by the time the chip materialized with the promised capabilities and clock rate, the computational performance was no longer very competitive with conventional CPUs.

8.5 iWarp, PCS, ConSet, and PCS+

iWarp is a reconfigurable connection-based MIMD machine that offers register-mapped network ports to allow low-latency communication. It provides local network configuration control at each cell, but the network can also operate in a semi-autonomous manner. The first application development tools for it either treated the machine as a SPMD device and used static communication channels for data distribution and collection (Adapt, Apply), or else set up a collection of static tasks (Assign) that utilized static communication channels.

PCS was the first tool enabling explicit task-level programming on the iWarp, but like Assign, PCS supported only a single configuration per application. PCS generated only program code; network configuration information was embedded by the toolchain within the cell program code.

ConSet was the first iWarp tool to enable multiple sets of different connections over the course of a single application. While it used a SPMD view of the array (only a single global task), it partitioned communication into a series of phases, allowing more connections to be realized over a period of time than was explicitly allowed by the available logical channels. ConSet produced both cell program code and a separate network program for the needed connection states[16].

PCS+ was an augmented version of PCS which kept the tasking model of PCS, but allowed tasks to be partitioned into "global communication phases". Each global phase was allowed to have a different network configuration, but changing from one phase to the next required global participation; the need for global participation precluded PCS+ from supporting asynchronous tasking. PCS+ kept the distinction between program code and network code introduced in ConSet, and generated both a program code executable and a network code executable as output[27,28].

8.6 HeNCE(1991), CODE(1992), and Paralex(1992)

A number of tools have been created for developing parallel tasking applications running on top of a PVM or PVM-like message-passing environment (high-latency, low-bandwidth network). The basic "task unit" in each of these models is the graph. Mapping of tasks to machines is based on a user-defined "cost matrix"; heterogeneous processors can be used in all of these systems. HeNCE uses a static mapping of tasks to processors, determined at compile time[8], whereas Paralex supports dynamic load balancing[3]. CODE restricted mapping tasks to nodes within a single multicomputer (a Sequent Symmetry), but also allows runtime mapping of tasks to cells[34]. For all ow these tools, communication and task ordering is specified by drawing "arcs" in the graphs, which represent unbounded FIFO buffers connecting "computation units". While these models and environments provide a connection-like communication model, the implementations rely upon a message-passing underpinning to provide the desired functionality and cannot offer any sort of guaranteed bandwidth or latency to the application designer.

One of CODE's major strengths is that the basic "task unit" in CODE is the graph, and graphs may be hierarchically defined; that is, one graph may invoke or "Call" another graph. CODE graph instances do not exist until called at runtime; task creation is dynamic. Like TCS, this design allows the hierarchical composition of complex tasks by assembling and encapsulating a collection of smaller, simpler tasks.

A characteristic of all of these models is that the programming model is defined in terms of the capabilities of the toolchain provided. The programming model, in effect, *is* the toolchain.

8.7 Orca-C and ZPL (1992)

While Poker introduced the idea of dividing parallel code into three levels: X-level (single-cell program code), Y-level (communication phases), and Z-level (problem level), it was restricted to a single Z-level routine per application. Orca-C and ZPL ("Z-level Programming Language") were an attempt to

1. Allow more complex Z-level routines to be created from simpler Z-level routines in a hierarchical fashion, and
2. Create scalable, reusable Z-level code.

The intent was that both Y and Z levels would be parameterized in terms of size and connectivity, and that X-level code could have easy access to block-partitioned data structures. This does allow for easier scaling, but prohibits use of communication patterns that cannot be easily expressed (short of a wirelist)[33].

8.8 Fortran M (1994)

Fortran M is essentially Fortran 77 plus a set of extension to support tasking and communication between tasks.[13,18] While HPF (High-Performance Fortran) provides a convenient means of expressing data-parallel code, it does not really allow task-parallel programming; Fortran-M allows modular, task-parallel program creation. Programs create processes that communicate by sending formatted messages over point-to-point channels. The mapping of processes to physical cells is handled by explicit directives in the program code. Communication and code placement are bundled into a Fortran M program itself; process creation/placement and communication channel creation/destruction are dynamic. This allows greater program flexibility, but makes any sort of static communication scheduling impossible. Communication is “connection-like” in that data ordering is preserved, but no guarantees are made regarding <u>when</u> data gets delivered. Channel communication occurs via formatted packets only, and is carried out by a lower-level message-passing subsystem. In one of the creators words:

> the primary difficulty that arises in compiling [Fortran-M] is to achieve efficient implementations of communication, synchronization, and process-management mechanisms.[13]

8.9 Fx (1994)

Fx is an experimental Fortran compiler that “incorporates task parallelism as directives in a data-parallel language based on HPF.” Unlike Fortran M, which depends on explicitly written **send**s and **receive**s for communication, the Fx compiler is responsible for generating all program communication.

> An Fx program begins execution as a single data-parallel task running on all nodes. When the flow of control reaches a parallel section, the tasks specified by calls to task subroutines are executed subject to data-dependence constraints; that is, each task waits for its input, executes, sends its output, and terminates. Parallelism is obtained by executing different tasks on different sets of nodes.[23]

Fx has several (rather severe) limitations. First of all, tasking only goes one level deep; a task may not invoke child tasks. Further, only one task-parallel region may exist per Fx program. Finally, because communication is implied (and compiler generated) rather than explicitly programmed, implementing systolic tasks or algorithms requires a bit of “smoke and mirrors” work. While Fx has been used for such tasks as real-time computer vision [44], routing the communication to obtain sufficient bandwidth to support continuous video was a user-controlled “trial-and-error” operation [23].

8.10 Mentat (1987)

Mentat is a C++ -based “macro data flow” system intended to provide a more-or-less transparent means of achieving parallelism. Mentat computations are called “actors”, and “arcs” represent the data dependencies between actors. Tokens, which represent data and control information, flow between the actors along the arcs. When an actor has tokens present on all of its incoming arcs, the actor is “enabled” and then executes, sending the appropriate tokens out on its outgoing arcs when complete. “Persistent actors” can maintain state information between firings. Mentat objects map one-to-one as processes running on a virtual machine, which in turn is mapped onto a parallel computer. [20,21]

Because Mentat applications are written for a virtual machine, programs can be expected to be portable across different parallel platforms. The downside is that by writing for a virtual machine, programs cannot easily take advantage of machine-specific “special features” that may offer better application performance. Object creation (actors and arcs) is dynamic, allowing maximum programming flexibility, but precluding static scheduling of communication or placement.

8.11 Static communication scheduling

Bianchini and Shen developed a “communication compiler” in 1986 that completely scheduled deterministic communication at compile time, offering repeatable (and guaranteed) runtime communication performance. In this system, “switching nodes” handled communication switching and routing, and physically separate “computation modules” handled computation. Each computation module would communicate with other processors through its corresponding switching node; the switching node would route the message through other switching nodes to the appropriate destination, according to a routing scheme evaluated at compile time.[9] This approach works well for synchronous, deterministic single-task systolic applications, but does not easily support multiple tasks which may communicate non-systolicly, or a group of tasks which, over time, execute on the same group of processors and switching nodes, switching between tasks in a data-dependent fashion.

The “Virtual Wires” work done more recently at MIT is a variant of this approach, geared towards scheduling communication within and between functional units of FPGAs. “Virtual wirelists” specifying connectivity and bandwidth between logical functional components are programmed and compiled, then mapped to available physical connections and scheduled in a time-division-multiplexed manner. In essence, “virtual wires” gave “logical channels” to the pins and interconnects of FPGAs. The increased utilization of pin and interconnect bandwidth allowed greater explotation of each gate array’s capabilities, resulting in a physically smaller system [4,14].

8.12 Chapter summary

A number of machines were built with explicit support for reconfigurable connection-like communication, each having a particular programming paradigm for utilizing connections. The CHiP only allowed a single connection per “phase” per cell with an external agent controlling network configuration; the GF-11 and polymorphic torus were SIMD machines. All of these enforced a globablly synchronous view of communication. More flexible, dynamic programming paradigms were introduced that allowed asynchronous tasking (and communication reconfiguration) (HeNCE, CODE, Mentat), but these require a general-purpose message-passing communication base that forfeits runtime communication guarantees. They also differ in how they can encapsulate the complexity of multiple, distributed communicating tasks as reusable program elements. For instance, Fortran M allows nested task definitions, while Fx does not.

The TCS model is unique in that it allows runtime communication guarantees to be requested at compile time (and confirmed at link time) while still allowing non-communicating tasks to execute asynchronously. The enabling technologies for these features are
(1) cells have local control over their network configuration,
(2) places where barrier synchronization is needed can be detected (and inserted) at link time, and
(3) fast barrier synchronization services are available.

Chapter 9 - Thesis Summary

9.1 Conclusions

The results of this thesis can be summarized as follows:

(1) Connections that provide minimal quality-of-service guarantees on latency and bandwidth are sufficient to build fast, predictable barrier synchronization.

Chapter 4 showed that barrier synchronization is a special case of all-to-all information exchange; the design space encompasses **physical signaling scheme**, **messaging protocol**, **allowable barrier memberships**, and **barrier capacity**. Connections provide a fast physical signaling mechanism; by choosing a messaging protocol appropriate to the barrier memberships and capacity required, one can achieve fast yet predictable worst-case barrier performance.

The TCS barrier implementation on iWarp can synchronize a group of 4 cells in under 160 clocks (8 microseconds), and a group of 32 cells in less than 816 clocks (41 microseconds). As a special case, an entire iWarp torus (64 cells) can be synchronized is just 456 clocks (23 microseconds). Not only fast, this barrier mechanism has a worst-case performance predictable to within 1 microsecond. This fast barrier mechanism allows tasks to coordinate at a very fine granularity level (tens of microseconds, in contrast to application latency requirements in the milliseconds).

(2) Given a connection implementation with connectivity state directly writable by the local cell, and fast, predictable barrier synchronization, a group of cells can perform communication context switches in predictable time.

A communication context switch allows cells within a task to swap between sets of active connections, hence allowing greater effective connectivity than the hardware could statically provide. Locally-writable connection state is important to prevent the unknown delays and artificial serialization that would be inflicted by queuing requests at an external reconfiguration agent, and fast barrier synchronization is necessary to keep the cost of doing the communication context swap affordable. By keeping the reconfiguration mechanism local to the cells affected, their reconfigurations are unaffected by activity in the rest of the array. By combining this local reconfiguration with a barrier implementation that is predictable as well as fast, reconfiguration itself can occur in predictable (to within microseconds) worst-case time.

(3) Given connections offering minimal quality-of-service guarantees and communication context switches, tasks can perform complex communications in predictable time. If the total connectivity required exceeds the intrinsic capabilities of the target platform, communication can be split into a series of local phases, each with its own set of connections.

A number of communication patterns representative of real-world application communications were implemented on the iWarp and benchmarked. Scatter/gather, reduction/broadcast, and all-to-all were implemented using both the TCS programming model and a fast deposit message-passing implementation for a variety of data sizes and varying numbers of cells. While message-passing offers arbitrary connectivity, its predictability falters as the communication patterns become more complex due to unknown runtime network congestion. TCS, using sets of local connections, maintained its predictability even for the all-to-all pattern.

(4) Given fast, predictable barrier synchronization, a simple set of task control primitives can be constructed with predictable performance. Tasks can then be constructed with those primitives to have predictable execution time.

Chapter 5 outlines the three basic control primitives that are built upon barrier synchronization: **task start**, **task end**, and **local communication context switch**. Chapter 6 showed how those control primitives are used to create tasks with communication patterns of varying complexity, and demonstrated on the iWarp that they did indeed maintain predictable (to within a few microseconds) execution time.

(5) Complex tasks can be hierarchically constructed from simpler tasks; if the component tasks have predictable performance, the resultant task can have predictable performance as well.

Chapter 7 showed how a complex task, a pipelined scatter/gather, could be constructed from a collection of simpler tasks. This complex composite task was predicted to meet, then demonstrated as meeting, the throughput and latency constraints of the larger application. Herein lies the power of the programming model: complex tasks can be built to offer predictable runtime performance by hierarchically assembling smaller, simpler tasks. The task granularity achievable is limited by the performance of the underlying barrier synchronization implementation; the faster the barrier performance, the finer the granularity achievable.

(6) By expressing an application in this manner, its potential runtime communication patterns are exposed at link time, allowing the toolchain to make global communication and barrier optimizations.

Link time routing allows the router to minimize runtime network congestion, in turn providing better performance. For instance, in Chapter 6, the TCS implementation of all-to-all data exchange outperformed the message-passing implementation by a factor of nearly three for 64-cell 32 Kbyte transfers. This performance was due to congestion avoidance allowed by link time routing.

9.2 Future work

9.2.1 Barrier hierarchies

Just as a choice of routing can be done at link time, the choice of barrier implementation can be made as well. A target platform may have more than one barrier implementation: a fast but small capacity barrier, and a slower, higher-capacity barrier. By doing the barrier allocation at link time, the fast barrier can be used in those “inner loop” situations where it provides greatest benefit, and the slower barrier used where execution time is less critical. This hierarchy of barrier implementations with a small fast-barrier capacity and larger capacities with slower performance is similar to a memory hierarchy, where one has only a few bytes of on-chip register space but Megabytes of slower DRAM.

Programs can be written in terms of generic barriers and memberships, with the linker deciding which implementation to use. Much as an optimizing compiler might promote a variable from a memory variable to a register variable, so too could a linker promote a barrier from one based on general-purpose messages based on the general machine network to one which runs on scarcer, special-purpose barrier hardware.

9.2.2 Other platforms

While most of this thesis’s work was done on iWarp, it should be possible to carry it to other machines that offer similar communication capabilities. Prospective target machines should offer connection-based communication with minimal quality-of-service bandwidth and latency guarantees, and a locally-accessible connection state. Mesh-based architectures (which put the switches close to the cells) are thus more likely candidates than tree-based architectures (which put the cells at the base of a hierarchy of switches). Furthermore, latency guarantees are as critical as bandwidth guarantees, especially for connection-based barrier implementations. Barrier messages are usually just a few bytes of information; barrier execution time is thus dominated by the physical signaling latency rather than signaling bandwidth.

9.3 Chapter summary

Connections with minimal quality-of-service bandwidth and latency guarantees are sufficient to implement fast, predictable barrier synchronization. Predictable barrier synchronization combined with locally-writable connection state enables local communication context switches, extending the connectivity within a task beyond what the target platform intrinsically supports. Given fast, predictable barrier synchronization and communication context switches, a small set of task control primitives can be built that allow task creation, execution, and destruction in predictable time. Tasks that are built from those primitives can be constructed to run in predictable time, and can be assembled into more complex tasks while maintaining predictable performance. Finally, by exposing an application’s potential runtime communication to the linker, global communication optimization can be performed.

Bibliography

1. Adamo, J.; Bonello, C.; and Trejo, T. “The C_NET Programming Environment: An Overview,” ***Parallel Processing: CONPAR’92-VAPP V The Second Joint International Conference on Vector and Parallel Processing***, Lyon, France, September 1992; pp. 115-120.

2. Alliant Computer Systems Corporation, ***FX/SERIES Architecture Manual***, Part Number 300-00001-B, January 1986.

3. Babaoğlu, Ö.; Alvisi, L.; Amoroso, A.; Davoli, R.; and Giachini, L.; "Paralex: An Environment for Parallel Programming in Distributed Systems," ***ICS-92***; Washington, DC. USA, July 1992; pp. 178-187.

4. Babb, J.; Tessier, R.; and Agarwal, A.; “Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators,” ***Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines***; Napa, California, USA, April, 1993; pp. 142-151.

5. Bailey, D.A.; Cuny, J.E.; and MacLeod, B.B.; "Reducing Communication Overhead: A Parallel Code Optimization," ***Journal of Parallel and Distributed Computing***, vol. 4, 1987; pp. 505-520.

6. Beetem, J.; Denneau, M.; and Weingarten, D.; "The GF11 Parallel Computer," in Dongarra, J., ed., ***Experimental Computer Architectures***; Elsevier Science Publishers, B.V., 1987; pp. 255-298.

7. Beckmann, C.; and Polychronopoulos, C.; “Fast Barrier Synchronization Hardware,” ***Proceedings of Supercomputing ‘90***, IEEE Computer Society and ACM SIGARCH; New York, New York, USA, November 1990; pp. 180-189.

8. Beguelin, A.; Dongarra, J.; Geist, G.; Manchek, R.; and Sunderam, V.; "Graphical Development Tools for Network-Based Concurrent Supercomputing", ***Proceedings of Supercomputing ‘91***. IEEE Computer Society and ACM SIGARCH; Albuquerque, NM, USA, November 1991; pp 435-444.

9. Bianchini, R.; and Shen, J.; “Interprocessor Traffic Scheduling Algorithm for Multiple-Processor Networks,” ***IEEE Transactions on***

Computers; C-36, no. 4 April 1987; pp. 396-409.

10. Birk, Y; Gibbons, P; Sanz, J; and Soroker,D.; “A Simple Mechanism for Efficient Barrier Synchronization in MIMD Machines,” ***Proceedings of 1990 International Conference on Parallel Processing, vol II;*** Penn State Press, University Park, PA, 1990; pp. 195-198.

11. Boillat, J.; Iselin, N.; and Kropf, P.; “MARC: A Tool for Automatic Configuration of Parallel Programs,” ***Transputing ‘91***; Sunnyvale, CA, USA, April 1991; pp. 311-329.

12. Borkar, S., et. al; "iWarp: An integrated solution to high-speed parallel computing," ***Proceedings of Supercomputing '88***, IEEE Computer Society and ACM SIGARCH; Orlando, FL, USA, November 1988; pp. 330-339.

13. Chandy, M.; Foster, I.; Kennedy, K.; Koebel, C.; and Tseng, C.; “Integrated support for task and data parallelism,” ***International Journal of Supercomputer Applications***; vol.8, no.2; Summer 1994; pp. 80-98.

14. Dahl, M.; Babb, J.; Tessier, R.; Hanono, S.; Hoki, D.; and Agarwal, A.; “Emulation of the Sparcle Microprocessor with the MIT Virtual Wires Emulation System,” ***Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines***; Napa, California, USA, April, 1994; pp. 14-22.

15. Dongarra, J., ed; “MPI: A Message Passing Interface Standard,” ***International Journal of Supercomputer Applications and High Performance Computing***, vol. 8, no. 3-4, Fall-Winter 1994; pp. 169-416.

16. Feldmann, A.; Gross, T.; O'Hallaron, D.; and Stricker, T.; "Subset Barrier Synchronization on a Private-Memory Parallel System," ***Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA92***; San Diego, CA, USA, June 1992; pp. 209-218.

17. Feldman, A.; Stricker, T.; and Warfel, T.; "Supporting Sets of Arbitrary Connections on the iWarp through Communication Context Switches", ***Proceedings of the 5th Annual ACM Symposium on Parallel***

Algorithms and Architectures, SPAA93; Velen, Germany, June 1993; pp. 203-212.

18. Foster, I.; “Task parallelism and high performance languages,” ***IEEE Parallel & Distributed Techology: Systems & Applications***; vol. 2, no.3; Fall 1994; pp. 27-36.

19. Giap, H.; and Macey, D.; “Validation of a Dose-Point Kernel Convolution Technique for Internal Dosimetry,” ***Physics in Medicine and Biology***; vol. 40, no. 3, March 1995; pp. 365-381.

20. Grimshaw, A.; and Liu, J.; “MENTAT: An object-ordiented macro data flow system,” ***OOPSLA ‘87: Conference on Object-Oriented Programming Systems, Languages, and Applications;*** Orlando, FL, USA; October 1987; pp. 35-47.

21. Grimshaw, A.; Strayer, W.; and Narayan, P.; “Dynamic, object-oriented parallel processing,” ***IEEE Parallel & Distributed Technology: Systems & Applications***; vol. 1, no. 2, May 1993; pp. 33-47.

22. Gross, T.; Hasegawa, A.; Hinrichs S.; O'Hallaron, D.; and Stricker, T; "Communication Styles for Parallel Systems," ***IEEE Computer***; vol. 27, no. 12, Dec 1994; pp. 34-44.

23. Gross, T.; O’Hallaron, D.; and Subhlok, J.; “Task Parallelism in a High Performance Fortran Framework,” ***IEEE Parallel & Distributed Techology: Systems & Applications;*** vol. 2, no.3; Fall 1994; pp. 16-26.

24. Hempel, R. “The MPI Standard for Message Passing,” ***Proceedings of the High-Performance Computing and Networking International Conference 1994, vol II: Networking and Tools***; Springer-Verlang, Berlin, Germany, 1994; pp. 247-252.

25. Hinrichs, S.; Kosak, C.; O'Hallaron, D.; Stricker, T.; and Take, R.; "An Architecture for Optimal All-to-All Personalized Communication," ***Proceedings of the 6th Annual ACM Symposium on Parallel Architectures and Algorithms, SPAA 1994***; Cape May, NJ, USA, June 1994; pp. 310-319.

26. Hinrichs, S.; “Compiler-Directed Architecture-Independent Communication Optimization”,Thesis CMU-CS-95-155, Carnegie Mellon University, School of Computer Science, 1995.

27. Hinrichs, S.; “PCS+ Tool Chain User’s Guide, Version 3.0+”, November 1993.

28. Hinrichs, S.; “Simplifying Connection-Based Communication,” ***IEEE Parallel and Distributed Technology***; vol. 3, no. 1; April, 1994; pp. 25-36.

29. Hwang, K.; and Shang, S.; “Wired-NOR Barrier Synchronization for Designing Large Shared-Memory Multiprocessors,” ***Proceedings of the 1991 International Conference on Parallel Processing vol I***; Penn State Press, University Park, PA, 1991; pp. 171-175.

30. Le Boudec, J.; Przygienda, B.; and Sultan, R.; “Routing Metric for Connections with Reserved Bandwidth,” ***IBM Research Report RZ 2560 (#83867)***, 2/7/94; pp. 1-7.

31. Lee Grahm, M.; Cheng, A.; Geer, L.; Binns, W.; Vannier, M.; and Wong, J.; “A Method to Analyze Two-Dimensional Daily Radiotherapy Portal Images from an On-Line Fiber-Optic Imaging System,” ***International Journal of Radiation Oncology, Biology, Physics***; vol. 20, no. 3, March 1991; pp. 613-619.

32. Li, H.; and Massimo, M.; "Polymorphic-Torus Network," ***IEEE Transactions on Computers***; vol. 38, no. 9, September 1989; pp. 1345-1351.

33. Lin, C.; and Synder, L.; "Data ensembles in ORCA C", ***Languages and Compilers for Parallel Computing, 5th International Workshop Proceedings***; New Haven, CT, USA, August 1992; pp. 112-123.

34. Newton, P.; and Browne, J.; "The CODE 2.0 Graphical Parallel Programming Language", ***ICS-92;*** Washington, D.C., USA, July 1992; pp. 167-177.

35. O’Hallaron, D. “Real-time airborne sonar processing on iWarp”. ***iWarp Forum***, Washington D.C., August, 1991.

36. O’Hallaron, D.; Subhlok, J.; and Webb, J.; “Performance Issues in HPF Implementations of Sensor-Based Applications.” ***Scientific Computing***, 1996. to appear.

37. O’Keefe, M.; and Dietz, H.; “Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM),” ***Proceedings of the 1990 International Conference on Parallel Processing, vol I***; Penn State Press, University Park, PA, 1990; pp 43-46.

38. O’Keefe, M.; and Dietz, H.; “Hardware Barrier Synchronization: Static Barrier MIMD (SBM),” ***Proceedings of the 1990 International Conference on Parallel Processing, vol I;*** Penn State Press, University Park, PA, 1990; pp.35-42.

39. Snyder, L.; "Introduction to the Configurable, Highly Parallel Computer," ***IEEE Computer***; January, 1982; pp. 47-56.

40. Snyder, L.; "Parallel Programming and the Poker Programming Environment", ***IEEE Computer***; July 1984; pp. 27-36.

41. Snyder, L.; "The XYZ Abstraction Levels of Poker-Like Languages", in Gelernter, D.; Nicolau, A.; Padua, D.; editors, ***Languages and Compilers for Parallel Computing***; MIT Press, 1990; pp. 470-489.

42. Stricker, T.; Stichnoth, J.; O'Hallaron, D.; Hinrichs, S.; and Gross, T.; ***Decoupling communication services for compiled parallel programs***. Technical Report CMU-CS-94-139, Carnegie Mellon University, School of Computer Science, 1994.

43. Subhlok,J.; Stichnoth, J.; O'Hallaron, D.; and Gross,T.; “Programming task and data parallelism on a multicomputer,” ***Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PpoPP)***, May 1993; pp. 13-22.

44. Webb, J.; Warfel, T.; and Kang S.B.; "A Scalable Video Rate Camera Interface", Technical Report CMU-CS-94-192, Carnegie Mellon University, School of Computer Science, 1994.

45. West, J.; Stephens, M.; and Turcotte, L.; “Adaptation of Volume Visualization Techniques to MIMD Architectures Using MPI,” ***Proceedings of the 1994 Scalable Parallel Libraries Conference***; IEEE Computer Society Press, Los Alamitos, CA, 1995; pp. 147-156.

46. Xu, H.; McKinley, P.; and Ni, L.; “Efficient Implementation of Barrier Synchronization in Wormhole-Routed Hypercube Multicomputers,” ***Journal of Parallel and Distributed Computing***; vol 16, no. 2. Oct 1992; pp 172-184.

47. Zhuang, X; Zhu, J.; “Parallelizing a Reservoir Simulator using MPI,” ***Proceedings of the 1994 Scalable Parallel Libraries Conference***; IEEE Computer Society Press, Los Alamitos, CA, 1995; pp. 165-174.