A Unified System for Scheduled Communication

David Shoemaker shoe@lcs.mit.edu
Frank Honore frank@lcs.mit.edu
Pat LoPresti patl@lcs.mit.edu
Chris Metcalf metcalf@lcs.mit.edu
Steve Ward ward@lcs.mit.edu

Massachusetts Institute of Technology
Laboratory of Computer Science
545 Technology Square #631
Cambridge MA 02141

Abstract

Managing communications in parallel processing systems has proven to be one of the most critical problems facing designers. As processor speeds continue to increase, communication latency and bandwidth become more of a bottleneck. Traditional network routers receive messages that are examined and forwarded by a local router. Performance is lost both by the time spent examining a message in order to determine its destination, and by the inability of a processor's router to have a global sense of message traffic.

The NuMesh system defines a high-speed communication substrate optimized for off-line routing. The goal of the project is to explore a new approach to the construction of modular, high-performance digital systems in which the components plug together in a nearest-neighbor three-dimensional mesh. Each component module contains a specialized programmable communications controller that routes message traffic among neighboring modules according to a precompiled pattern. The NuMesh project reserves bandwidth for possible message transfers at compile time. By setting fixed periods in which processors can communicate with each other, no message data need be examined and a compile-time analysis of message traffic can minimize network congestion. This paper examines the design space for supporting scheduled communication from both the hardware and software perspective.

Keywords: scheduled communication, communication network, parallel processing

1 Introduction

In recent history, the backplane bus has been dominant as the primary mechanism for connecting multiple devices together. For small numbers of devices, this model worked very well because the designer could simply connect additional processing elements to the bus, without having to consider interactions with other devices or bus arbitration. A protocol was standardized and a connecting device merely had to follow the protocol set up by the bus designer. As the number of devices grew, the backplane bus quickly ran into trouble, since it serialized all system level communication. High performance designers developed a variety of special purpose networks, but they required the processor to be modified considerably in order to communicate with the network efficiently. One of the goals of the NuMesh project is to standardize this interface so that communication operations can be parallelized and any off-the-shelf component can be connected to the network with limited overhead. Since the network involves point-to-point communications between NuMesh nodes, the electrical difficulties of traditional busses are avoided. A designer can connect an arbitrary number of processing elements depending on application needs. The goal is for devices to be added or removed from the network with a "Tinker-Toy" modularity.

A high level view of the NuMesh shows a large number of heterogeneous processors that operate synchronously. The processors are connected to a communication network which orchestrates communications between the processors on a per cycle basis. Consecutive clock cycles involve completely different communication patterns through a single node. Some processors have a very high bandwidth reserved for their communication while others may only be allowed to communicate very infrequently. These bandwidths are determined at compile time, although changes can be made during operation with limited overhead. FIGURE 1 shows a global communication pattern for a 2-D network of nodes and how a snapshot of consecutive cycles might appear. Note that although messages A and B both use node (2,0), the communication can be time-sliced so that on consecutive clock cycles, the node is handling completely different communication threads. Message C has been assigned a hundred percent bandwidth so it will be scheduled every cycle. Effectively, this link will be reserved for that message and all other messages that would logically go through this link would be misrouted around the link.

The communication controller in each NuMesh node, which is termed the communications finite state machine (CFSM), mediates communications between the host node and its immediate neighbors. Off-line analysis of internode communication provides separate communication instructions, loaded during boot-up, for each NuMesh node. Each CFSM provides control signals for its node during each cycle of a globally synchronous clock. Individual NuMesh nodes can be connected like Legos, to form the base platform for a scalable, three-dimensional parallel processing machine. By minimizing network congestion through off-line software efforts and by minimizing clock cycle delays with a heavily pipelined, data independent architecture, the network should provide exceptional performance, especially for those cases that have a limited number of run-time decisions being made. A picture of a NuMesh node is included in FIGURE 2

2 Design Space

A parallel processing system attempts to take advantage of the fact that applications often have largely independent computational portions that can run concurrently and require some amount of communication between the processes in order to complete. These processes run on different nodes of the parallel processing system, periodically injecting and removing data from the communication network in order to meet the communication demands of the application. Decisions made when implementing the communication network, the processing nodes, and the interface between the two, have marked effects on the number and style of applications that can be supported

2.1 Dynamic networks

The most general approach to parallel programming is to assume no information about the frequency and patterns of communication that will occur over the course of an application. The need for communication occurs because a node

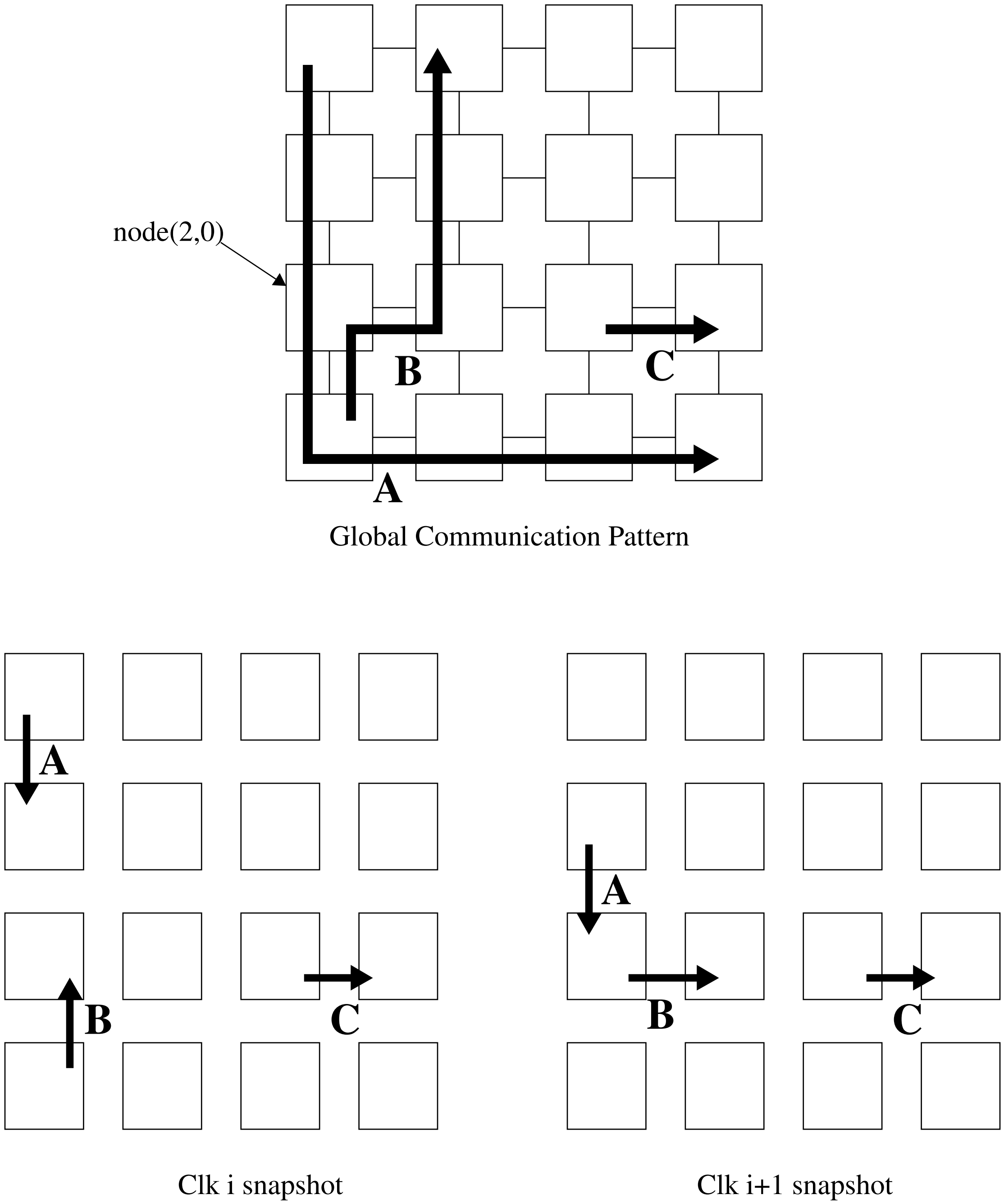


FIGURE 1 High Level View of NuMesh

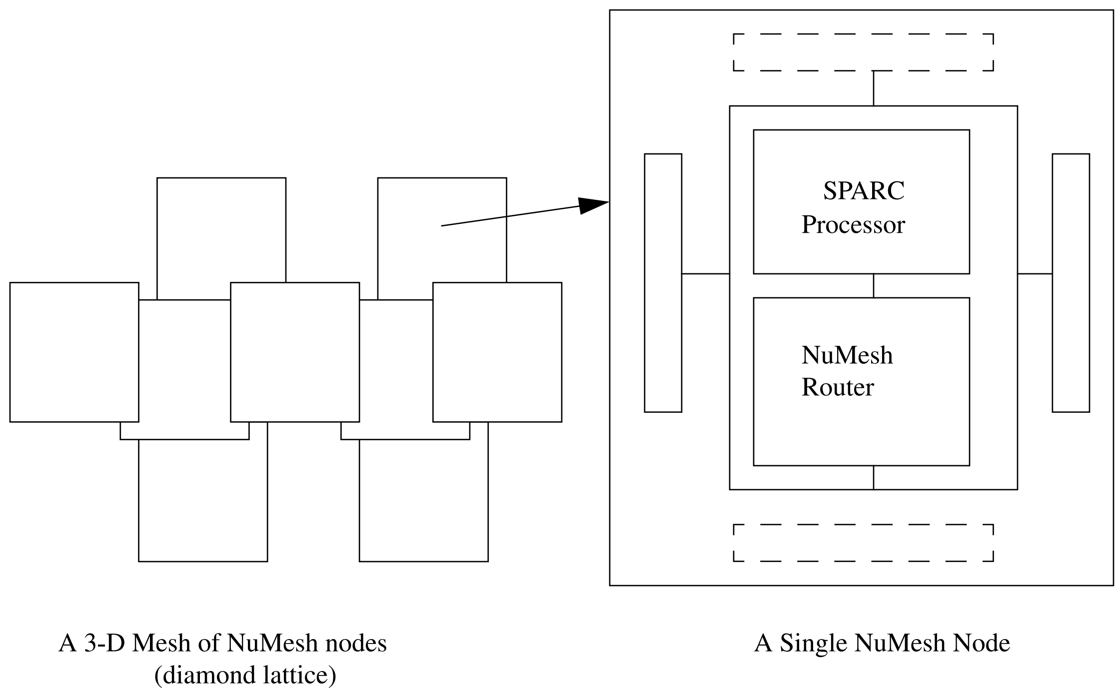


FIGURE 2 NuMesh Overview

either requests information that exists on another node or it computes some value that needs to be sent to another node. If no information about the frequency or patterns of these communications is known, the sending processor includes a destination address for the message, and the communication network must be able to interpret this data at run-time and eventually route the data to the correct node. The communication routers can keep track of previous messages and can be supplied with various control bits from neighboring nodes to help determine the least congested route for forwarding messages. Routers implemented in this fashion form the broad class of communication networks called dynamic networks. Dynamic routers have the benefits of simplified programming and compilation models. Since all routing decisions are data dependent, the application need only determine where a desired piece of data resides or where a message needs to be sent. The rest of the routing problem is handled in hardware. This dependence on limited hardware to handle the routing decisions suffers a number of disadvantages as well:

- Nodes create messages and inject them into the communication network in an unpredictable pattern causing hot-spots of congestion to be created. Since each node of the communication network has a limited amount of information about which links in the system are currently being used, messages may be stalled due to congested links even though alternative links may be free.
- Every time a message is forwarded a node in the communication network, a data-dependent decision must be made by the router to determine which direction the message should next be sent. For routers attempting to incorporate more information into this decision, the latency of a message through a node is even greater, since the decision process involves some number of gate delays.

Dynamic networks do not attempt to take advantage of communication information present in the application at compile-time. While some applications may yield little such information, many applications provide at least some indication as to the communication that will occur over the course of the application. This information may include the source and destination of upcoming messages, the relative frequency at which nodes can expect to send and receive messages, the exact timing of when processes will send or receive messages, and the evolving communication patterns that might occur under depending on branches taken in the application. While applications may provide all or none of this information at compile-time, dynamic routers can not take advantage of the case even when communication is fully specified.

2.2 Static Networks

Static networks demand that all communication patterns and timing be known at compile time. Static networks allow applications to specify multiple communication paths by allowing physical channels to be shared among a number of communications required by the application. The FSMs of the communication routers can be time-multiplexed such that a separate set of communication paths can be supported on every cycle. Since the exact timing of the processors is known at compile time, the communication FSMs can know exactly where each message is to be routed on each clock cycle. The communication network can be viewed a one giant finite state machine that is orchestrating communications between the processors. When this FSM is being created, a clever compiler can utilize all the physical links in the system, thereby maximizing bandwidth for the system. Unavoidable congestion can be handled by time-multiplexing physical channels among the multiple messages requesting use of the channel. The system has the advantages that no run-time data-dependent decisions need to be made, and congestion can be minimized at compile time. An arbitrary number of messages can be used by the application since the communication FSMs can allow nodes to share the physical resources by time-multiplexing the channels. However a number of disadvantages remain:

- The application code must be written such that every communication can be extracted at compile time. Not only does the source and destination for each communication message need to be known, but the exact time at which the processing nodes create and accept messages must also be known. Any dynamic behavior in the system can be lethal since the communication network can be viewed as one big FSM that demands precise timing.
- The processing units must be designed such that the time to execute code is completely predictable. Even something as simple as a cache miss can completely throw off the timing of the system. Traditional microprocessors have a number of sources of variable timing. Traps, interrupts, cache misses, and DRAM refreshes are just a few of the sources that prevent application code timing from being easily predictable.

While static networks are more robust than systolic networks, the processor must be carefully engineered and characterized to allow for the precise timing needed in a static system. Static networks are optimized for applications that exhibit an arbitrary number of regular communications. Both the computation and communication behavior of the application must be evident at compile-time. While this model does not hold for all parallel applications, a large body of scientific and graphics code is appropriate.

2.3 NuMesh System

The NuMesh system examines a different point in the design space[4]. At compile time, the NuMesh system tries to take advantage of paths of communication in a system, but does not require exact timing information from the processing nodes. *Virtual streams* of communication are extracted from the application that identify a source node, a destination node, and a requested bandwidth that determines how often this communication is likely to occur. A directed graph of communication can be composed from the virtual streams of communication. Similar to the static network, the graph of communication gets mapped to a global FSM of communication, part of which gets stored on each communication node. Each virtual stream gets scheduled for some number of clocks, based on the bandwidth request set up in the program. When a virtual stream is scheduled, data need not be sent, but a communication path will be reserved between the source and destination of the message. Since a message can travel one hop on every clock cycle, scheduling a virtual stream implies assigning consecutive cycles on consecutive nodes of the communication path for the stream. This model of communication is termed *scheduled communication.*

While the communication network reserves bandwidth between communicating nodes, the processors are dynamic in nature, meaning that the exact time at which messages will be injected or removed from the network is variable. This requires that words be allowed to back up in the communication network, without upsetting the timing of the global FSM. Some amount of buffering is assigned to each virtual stream at each node of the communication path.

The decoupling of the communication network from the attached processors yields an interesting combination of static and dynamic behavior. The static nature of the communication network allows the system to take advantage of improved network performance due to the faster cycle times and reduced congestion. At the same time, the restrictive timing constraints of processors found in a traditional static systems are removed, preventing the dynamic behavior of caches and other unpredictable processor effects from crippling the timing of the system.

The disadvantages of the NuMesh approach are that bandwidth in the communication network may go wasted if the processor is unable to fill its reserved communication slots and the processor may be unable to send a message if a virtual stream is not scheduled but the processor is ready to send data. Also, the application must be written such that the sources and destinations of messages can be extracted at compile time. Although the NuMesh system does support limited dynamic routing, there is a cost in efficiency.

3 Scheduled Communication

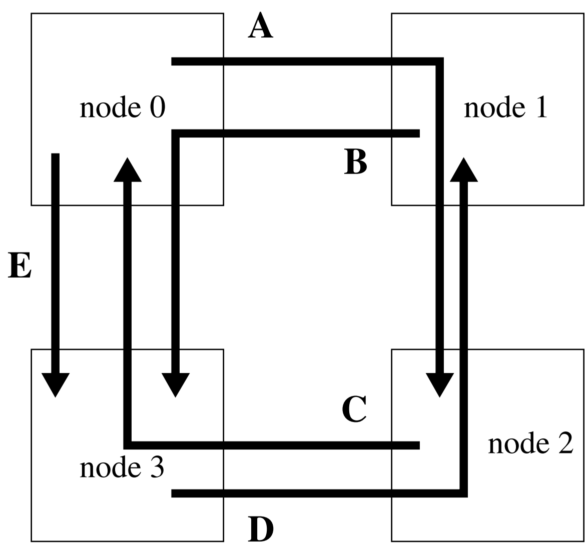
This NuMesh system assumes that communication patterns can be extracted from the parallel language in which the applications are written. Scheduled communication indicates that some determination has been made as to the communication of the processors during compile time. Ideally, all sources and destinations will be specified, and an idea of how often each communication will occur will be provided. Once these virtual streams are specified, a scheduler can load state into each of the NuMesh communication routers that executes a run-time schedule matching the requests of the virtual streams.

However, the determination may be that there are a variety of run-time decisions that will be made. Even for these cases, the NuMesh will load instructions during bootup that allow the nodes to handle more dynamic communication. Off-line analysis of application code may even determine that some routing decisions will be data dependent. In these cases, some mechanism must be incorporated in the scheduled communication to allow for these checks of data. The more regular and predictable the communication patterns, the more benefit can be extracted from scheduled communication.

Scheduled communication does not suffer from the rigidity of static or systolic routing. While a purely static system demands that communication be fixed and that data be sent in a regular pattern, a scheduled communication system reserves bandwidth between processors, although this bandwidth need not be used if the source processor has no data to be sent. Dynamic communication has even more flexibility, since all decisions are made during run-time, although this flexibility naturally leads to network congestion and possibly pathological network performance such as livelock or deadlock.

By assuming that the applications can have their scheduled communication extracted at compile-time, the NuMesh system can take advantage of the minimized network congestion, the lack of data-dependent routing decisions, and the ability for the hardware to know about possible upcoming communications well ahead of the actual transfer of data.

FIGURE 3 presents a simple example of the kind of information that a scheduled communication analysis might present. In the example, the communication needs of a four node system are examined. In the figure, five communication messages are identified. Each message has a bandwidth assigned to it that represents the percentage of the time that the message is expected to appear in the network. A bandwidth of twenty-five percent would mean that the communication links needed for that message would be utilized around one out of every four clock cycles. These bandwidth numbers need not be exact, but they will be used to reserve communication links for a particular message. If a message gets allocated twenty-five percent bandwidth, but then the sending processor only sends out a message every hundred cycles, then there will be a lot of cycles reserved for that particular communication that are not used. Likewise, if the sending processor has data available to be sent every other cycle, the network will be unable to accommodate all the processor's attempts to inject messages into the network, and data will back up in the sending processor's memory causing the application to stall. While neither of these situations is catastrophic, the closer the assigned bandwidth is to the processor's performance, the better the application will perform.



	BW	Size
Message A	25%	1
Message B	50%	2
Message C	25%	1
Message D	25%	1
Message E	50%	1

FIGURE 3 Scheduled Communication Example

The size metric obtained during off-line analysis serves only to indicate how the bandwidth should be scheduled. When a processor is ready to inject a message into the network, it is usually ready to inject the entire message, regardless of size. If a node has a bandwidth assignment of fifty percent, that means that the data links for that communication will be used every other cycle on average. If the messages are of size two, the processor ideally will have consecutive cycles available to inject both words of the message into the network. The communication thread will not be scheduled again for another two cycles while the processor gets the next message ready. If the message size were one, then the processor would have every other cycle reserved to send its one word message. While, for this example, the difference between size one and size two messages being sent with a fifty percent bandwidth is not that significant, the difference is important for messages of a larger size that might be scheduled infrequently. A message of size thirty communication words might get assigned a bandwidth of one percent. If no size limits are taken into account, the schedule would provide this communication thread one cycle of reserved bandwidth for every hundred cycles. Taking the size metric into account, the thread gets thirty consecutive cycles of reserved bandwidth followed by two hundred and seventy cycles of not getting scheduled. The application performance might be drastically different for the two cases.

Once the scheduled communication information has been extracted, graph analysis can yield a viable network schedule that attempts to meet the required bandwidth and size constraints. The example in FIGURE 3 assumes a fixed path between the nodes, but in practice all that may be specified are the source node and the destination node of the message, allowing more flexibility in the scheduling process. While some constraints may be impossible to satisfy, the communication streams compiler can decide how to come as close as possible to satisfying them. For the simple example in FIGURE 3, one possible schedule is illustrated in FIGURE 4. The schedule that was generated has four cycles in length. For this example, it should be noted that each node can handle two distinct communications on each clock cycle. Each message appears in the schedule based on the bandwidth that was assigned to the node. For example, message A requested a bandwidth of twenty-five percent. Looking at the schedule, we see that node zero (the source node for message A) has message A scheduled in slot zero. This demands that node one, the next node in the communication path, has message A scheduled in clock cycle one. Consequently, node two has message A scheduled in clock cycle three. For messages B and E, which both have a bandwidth of fifty percent, the messages appear in two out of the four cycles for each of the nodes in those paths. Since message B has a size of two, it gets scheduled for two consecutive clock cycles while message E with a size of one gets scheduled every other cycle. This will be the communication pattern for these nodes for all time since after the fourth clock cycle, the schedule starts over again at clock cycle 0. While this simple example is trivial to schedule, more complicated applications provide more complicated communication schedules that often require multiple phases of communication in order to complete.

clock cycle \ **nodes**	0	1	2	3
0	msg A & idle	msg B & idle	msg C & idle	msg D & msg E
1	msg B & msg E	msg A & msg B	msg D & idle	msg C & idle
2	msg B & msg C	msg D & idle	msg A & idle	msg B & msg E
3	idle & msg E	msg 0 & idle	idle & idle	msg B & idle

FIGURE 4 Four Cycle Schedule for Simple Example

4 Software Environment

While the hardware architecture to support the NuMesh system was described in [5], a strong software environment has been created to allow for the creation of parallel applications. The Communications OPerator language (COP) is a high level parallel language that provides complex application semantics that automatically map down to the NuMesh hardware. The Tadpole compiler handles the scheduling of communication once a directed graph of communication is extracted from the COP language.

4.1 Communications Operator Language

The models discussed so far have presupposed a mesh of static, compile-time streams that are constant for the duration of the application. Such models are sufficient for a variety of signal processing or simple scientific applications, but become a liability for more complex applications. However, NuMesh is not limited to this simple style, and a higher-level model is available for programming as well: this model is known as COP, the Communications OPerator language[3]. Two particular issues differentiate COP from the previous models.

The first issue is that many applications do not have a consistent communication requirement throughout the duration of an application, but rather communicate in successive *phases*. NuMesh handles this model by allowing for high-speed reprogramming of the router unit. This reprogramming can be done in one of two ways. Firstly, we can intersperse *update* threads into the current phase that transfer RAM data/address pairs from some processor register to the router memory. We can use this functionality to create a new set of threads and schedules in a part of router memory not involved in the current phase, then synchronously transfer control to the new phase's schedule with a final data/address pair write. Alternately, for higher speed router updates, we can transfer the router's schedule pointer to a fixed *update phase* consisting entirely of update threads, transfer one or more phases to the router, then switch to the new phase.

Carrying out such phase switches requires synchronization within the application. The easiest technique to achieve this is simply to allow for each processor to transfer its router to the next phase when it completes all the I/O for the current phase. This may leave the router out of synch with its neighbors, but as long as it is careful not to try to transfer data initially using the same timeslots as the previous phase, no conflict will occur. To limit the number of such protected timeslots, an additional barrier operation can be added to the previous phase to synchronize the phase change among a set of nodes.

A difficult question for the compiler is knowing when to insert phase changes into an application's execution. An application may define communications that occur sequentially and/or in parallel, and that may vary for different parts of the mesh at the same time. The COP compiler attempts to place particular communications in separate phases when doing so improves the predicted runtime, while grouping communications in the same phase when required by the application semantics. A large number of phases can then be stored in the processor's memory, and loaded into the router at runtime in a cache-like manner as the application runs.

The second issue that is not adequately addressed by a simple stream model is runtime-variable communications. A typical application may perform a broadcast from a different node each iteration, or do point-to-point communications between runtime-varying nodes. The COP model handles this by allowing the user to specify as much detail as possible about the communications, and then using that to generate a suitably constrained runtime solution. For example, the COP operator "(broadcast (runtime 0 1 2 3))" tells the compiler to emit code to support a broadcast with a runtime source, where the source is constrained to be one of nodes zero through three. The compiler is thus provided with information about the type of communication and its sources and destinations; the volume of data to be transferred is normally also specified for each operator. Higher-level constructs provide information on operators' parallel or sequential groupings, their loop context, and so forth.

The easiest way to handle data-dependency if there is a small number of possible destinations is to schedule all of the paths and use the appropriate one at compile time. This is particularly true if the amount of data to be transferred is small, since allocating multiple paths results in no extra message startup overhead time, unlike most of the other schemes described below. This scheme rapidly becomes impractical as the number of destinations grow and the amount of data to be transferred increases.

A different approach, when the number of run-time possibilities is relatively small, is to simply select an appropriate schedule based on the run-time arguments, load that version of the schedule on all the nodes, and then run at full bandwidth. This approach requires that all the nodes involved in the communication know which schedule to load. For example, in a broadcast, if all the nodes know who is broadcasting they can choose the appropriate piece of broadcast schedule to run. If all the variant schedules can fit simultaneously in the router, furthermore, the runtime decision may simply consist of a comparison or two and a single write to the router's schedule pointer.

A slightly more complex alternative is to generate a fairly generic schedule which can be customized at run time. It may also be possible to run such a schedule in parallel with other operators, since the customization can be applied to a specific thread after thread scheduling has been performed. For example, for NuMesh, we could implement a linear broadcast from a runtime value by allocating two paths going through the mesh, then on the node that is doing the broadcast we rewrite the router code to read from the processor rather than the neighbors, and write data into the two channels from that node.

If the operator needs to transfer a lot of data, we can generate a schedule using operator-specific code, tuned to produce an efficient schedule using the run-time data that is available. In the simplest case, each node computes the same problem (e.g., the scheduling problem for the whole mesh for a given permutation) and then puts the relevant part of the solution into its own router. Since the startup time for this technique is high, however, we must ascertain that the amount of data to be transferred is large enough to amortize the startup schedule computation.

If we have no other alternatives, we can use dynamic routing to move all the data, by creating specialized *dynamic routing* channels. We have complete information on how much bandwidth will be required along link, and can thus specialize the dynamic routing channels to match the demands of the application. The processors on each node provide the packet-forwarding intelligence by using an interrupt-driven dispatch function and in-memory queuing. Note that when doing general-purpose dynamic routing, we can still run offline channels within the same schedule; e.g., we could have a low-bandwidth broadcast channel set up offline, and run a general dynamic router with the remaining bandwidth, during a phase with unknown communications characteristics. A future generation of NuMesh (or similar projects, such as RAW[6]) can add a hardware assist on the timeslots used by dynamic routing, rather than relying on interrupt-driven processor handling.

Thus, by providing a complete solution to an application's communications needs, COP allows for essentially arbitrary applications to run under NuMesh. Applications with significant run-time communications requirements and varying communication phases can take advantage of NuMesh's inherent speed, suffering only incremental slowdowns as their degree of runtime unpredictability increases.

4.2 Tadpole Compiler

Tadpole[2] provides a programming model for the NuMesh based on "streams". A stream is a logical connection, with some specified bandwidth, from some source processor to one or more destination processors. Given a collection of stream specifications (i.e., each stream's source, destination(s), and desired bandwidth), Tadpole will route the streams through the physical Numesh and schedule the exact cycle-by-cycle behavior to provide the desired bandwidth.

Tadpole appears to have two distinct tasks at first glance: First, to route streams through space in a manner which avoids excessive congestion; and second, to schedule the cycle-by-cycle usage of the interprocessor communication links such that each link is only used by a single stream on any cycle. In fact, Tadpole accomplishes both tasks simultaneously by viewing the problem in space-time.

The dual job of routing in space and scheduling in time may be viewed as the single job of routing circuits in space-time. Of course, any three-dimensional system going through some sequence of states can be viewed as a single static four-dimensional system. This point of view is usually impractical, since most computations take many cycles, resulting in a large space-time representation.

However, the NuMesh has the property that its communication schedules are cyclic with a fairly small period. Consequently, it is feasible to explicitly construct the entire space-time representation of the communication substrate, and perform static resource allocations in that representation to perform dynamic resource allocations in the real world. Tadpole does precisely this.

The task of routing in Tadpole's space-time graph may be considered as one of circuit routing, or (for single-destination streams) of discrete multicommodity flow[1], or (for multicast streams) of finding Steiner trees and packing them together. All of these are well-known NP-complete problems, so Tadpole uses a heuristic approach which works reasonably well in practice. Tadpole simply routes and re-routes streams by finding minimum-cost paths through a graph whose edge costs are determined primarily by the congestion in the network. This approach was chosen primarily for its simplicity, but also because it provides a good way to incorporate knowledge about the various hardware restrictions which the NuMesh imposes: Such restrictions are coded into the edge cost function, so that illegal uses of the hardware are assigned infinite cost.

5 Conclusion

The NuMesh system attacks the problem of communication networks at a unique point in the design space. Allowing processors to behave dynamically allows for the creation of a heterogeneous network of computation. The requirement of a scheduled communication for message routing allows messages to avoid congestion in the network and allows for the architecture of the communication routers to be simple and fast. This combination of ideas does comes with the requirement of a more structured application design methodology. The NuMesh project has created a hardware system capable of supporting scheduled communication as well as a software environment capable of supporting a large number of application domains.

6 Bibliography

[1] R. Bianchini and J.P. Shen. "Interprocessor Traffic Scheduling Algorithm for Multiple-processor Networks", *IEEE Transactions on Computers*, C-36 (4): 396-409, April 1987.

[2] P. LoPresti. "Tadpole: An Off-line Router for the NuMesh System", *Master's Thesis.* MIT, Jan 1997.

[3] C Metcalf. "A Comparative Evaluation of Online and Offline Routing", *PhD Thesis in Progress*. MIT, 1997.

[4] D. Shoemaker et al. "NuMesh: An Architecture Optimized for Scheduled Communication", *Journal of Supercomputing.* Vol. 10, pages 285-302, August 1996.

[5] D. Shoemaker. "An Optimized Hardware Architecture and Communication Protocol for Scheduled Communication". *PhD Thesis*. MIT, June 1997.

[6] E. Waingold et al. "Baring it all to Software: The RAW Machine", *MIT/LCS Technical Report TR-709*. March, 1997.