# Network-Based Multicomputers: A Practical Supercomputer Architecture

**Peter Steenkiste**

**School of Computer Science**
**Carnegie Mellon University**
**Pittsburgh, Pennsylvania 15213**

## Abstract

Multicomputers built around a general network are an attractive architecture for a wide class of applications. The architecture provides many benefits compared with special-purpose approaches, including heterogeneity, reuse of application and system code, and sharing of resources. The architecture also poses new challenges to both computer system implementors and users. First, traditional local-area networks do not have enough bandwidth and create a communication bottleneck, thus seriously limiting the set of applications that can be run effectively. Second, programmers have to deal with large bodies of code distributed over a variety of architectures, and work in an environment where both the network and nodes are shared with other users. Our experience in the Nectar project shows that it is possible to overcome these problems. We show how networks based on high-speed crossbar switches and efficient protocol implementations can support high bandwidth and low latency communication while still enjoying the flexibility of general networks, and we use three applications to demonstrate that network-based multicomputers are a practical architecture. We also show how the network traffic generated by this new class of applications poses severe requirements for networks.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract number MDA972-90-C-0035, in part by the National Science Foundation and the Defense Advanced Research Projects Agency under Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

Accepted for publication in IEEE Transactions on Parallel and Distributed Systems.

Keywords: multicomputer, workstation cluster, high-speed network, host-network interface, distributed programming, traffic characteristics.

## 1. Introduction

Current commercial parallel machines cover a wide spectrum of architectures: shared-memory parallel computers such as the Alliant, Encore, Sequent, and CRAY Y-MP; and distributed-memory computers including MIMD machines such as the Transputer [29], iWarp [9], and the Paragon [31], and SIMD machines such as the Connection Machine [62], DAP, and MasPar [7]. Like SIMD machines, distributed memory MIMD computers, or multicomputers, are inherently scalable. Multicomputers however can handle a larger set of applications than SIMD machines because they allow different programs to run on different processors. Multicomputers with over 1,000 processors have been used successfully in some application areas [27].

Multicomputers are traditionally built by using proprietary interconnections to link a set of dedicated processors. Like a proprietary internal bus in a conventional machine, the interconnect is intended to connect to a small set of specially-designed processor boards, and is optimized to do so. System-specific interconnects are used instead of general networks, such Ethernet or FDDI, mainly for performance reasons. However, since they are system-specific, these interconnects do not have the flexibility of general networks: for example, they cannot be connected to many types of existing hosts. Moreover, traditional multicomputers typically also use a customized runtime system and programming environment [30, 5].

This paper argues that it is feasible to build high-performance **network-based multicomputers** that use **general networks** instead of system-specific interconnects. Such a multicomputer is able to use **existing** hosts, including workstations and special-purpose processors, as its processors. The processors keep their standard software, thus allowing the reuse of both systems and application software and providing programmers with a familiar environment. At the same time, network-based multicomputers are able to take advantage of rapid advances in network and processor technology, since new networks and systems can be added without affecting the rest of the system. While enjoying a high degree of flexibility, the underlying network can have performance comparable to that of a dedicated interconnect. As a result, many applications that run on existing multicomputers also run efficiently on network-based multicomputers.

The **Nectar** project is one of the first attempts to build a high-performance network-based multicomputer. Nectar is composed of a high-bandwidth crosspoint network and dedicated network coprocessors. A system using 100 Mbit/second links has been operational since early 1989. The number of nodes attached to Nectar changed over time, starting with 2 nodes in 1989 and going through a peak of 26 nodes in 1991; it was decommissioned in October 1994. Most hosts attached to Nectar were workstations, although some application experiments [33] were performed involving the Warp system [2], iWarp [9], and the Cray-YMP. The Cray-YMP is accessible via a 26 kilometer single-mode fiber link to the Pittsburgh Supercomputing Center. We are currently further exploring multicomputer issues in the gigabit Nectar project [57, 59], and the Nectar testbed, one of the 5 national gigabit testbeds that is exploring the benefits of gigabit networks on applications such as high-performance computing.

The Nectar system has been used as a vehicle to study architectural issues in network-based multicomputers and as a testbed for the development of programming tools. A key element of our work is the development of applications covering a range of programming models in cooperation with several applications groups. This paper is based on insights and experiences gained from the development of the Nectar system and its use by applications programmers.

This paper is organized as follows. In Section 2 we summarize the advantages of network-based multicomputers and the challenges they create for both implementors and users. In Section 3 we give an overview of the Nectar system, and we discuss how it addresses the general network requirements while maintaining the performance required from multicomputer interconnects. In Section 4 we use a set of applications to prove the viability of network based multicomputers. Section 5 characterizes the traffic of a set of distributed applications and discusses its impact on both the network and application. We summarize our results in Section 6.

## 2. Network-based multicomputers

We summarize the advantages and new challenges offered by network-based multicomputers.

### 2.1. Network-based multicomputer advantages

Network-based multicomputers offer substantial advantages over traditional multicomputers that use dedicated interconnects. A first advantage is that a single application can **combine the resources of many computer systems**, e.g. CPU cycles and memory. Traditional multicomputers offer the same benefit, but network-based multicomputers make this possible using off-the-shelf systems such as workstations, which are more economical than special-purpose systems. The underlying high-speed network is also inherently suited to support high-speed I/O to standard devices such as displays, sensors, file systems, mass stores, and interfaces to other networks.

A second advantage is that by incorporating **existing** systems as hosts, a network-based multicomputer can take advantage of rapid improvements of commercially available computers, and can reuse existing systems software and applications. In fact, a network-based multicomputer provides an environment for gracefully moving applications to new architectures: initially the application can execute only part of the computation on the new system, and as more software and application code for the new systems is developed, the use of the new system can be increased. Finally, a network-based multicomputer can incorporate **heterogeneous architectures**, so applications can select the most suitable computing resources for each computation.

### 2.2. Multicomputer challenges

Network-based multicomputers, however, are more complex than tightly coupled systems. This creates new challenges, including developing high-performance networks and providing programming tools that help users dealing with a new set of programming problems.

### 2.2.1. Multicomputer interconnect requirements

For today's traditional multicomputers, the bandwidth of each interconnection link can be as high as several 100 MByte/second [31, 1] and the communication latency between processes on two processors can be as low as several 10s of microseconds [8] or lower [1]. Traditional local-area networks have much lower performance, in part because they have to operate in a much more complex environment. The increased complexity shows up in two areas: the interconnection network itself, and the interface to the host.

The nodes of network-based multicomputers are typically distributed over a building, campus or even continents, so homogeneous interconnects organized as a torus or hypercube are not practical. Because of the long physical distances, the underlying network of a network-based multicomputer needs to cope with data errors and network failures, and since the nodes of the multicomputer are autonomous computer systems with multiple users, security is a consideration. Moreover, the network has to handle the addition and removal of nodes while the network is operating. Finally, since the system is shared with other users, the system has to deal with congestion and fair bandwidth allocation among users, functionality that is not required in dedicated systems. In summary, as a result of the differences in the environment, networks of multicomputers have to be more flexible, reliable, and secure than the interconnects of tightly coupled systems.

How tightly a node is coupled to the network has a big impact on the communication efficiency. In tightly coupled distributed memory systems, the CPU and network are very closely coupled, either at the chip level (e.g. iWarp [10]), or at the board level with access times to the network similar to that of cache accesses (e.g. Touchstone [5] or DASH [45]), resulting in low-overhead communication. In network-based multicomputers, however, the network enters the host through an I/O bus, i.e. one step further removed from the CPU than in tightly coupled systems, resulting in slower network access.

The performance of a network determines both the class of applications can be distributed and the degree of parallelism that can be exploited effectively. While many applications have been distributed successfully over Ethernet-based workstation clusters, improving the performance of the network will make the cluster useful for a larger set of applications. In this paper we use the Nectar system as an example of how it is possible to build a high performance network and use it as the core of multicomputer. Higher network link rates, as supported by ATM and HIPPI, combined with efficient communication support on the hosts, as discussed in this paper, will make workstation clusters increasingly attractive for a wide class of applications. Specialized tightly-coupled systems will always support faster communication, but their advantage is likely to shrink over time.

### 2.2.2. Programming challenges

Developing large applications for network-based multicomputers involves additional challenges beyond the ''normal'' problems associated with programming distributed memory machines:

1. Multicomputers are an attractive architecture for large ''grand challenge'' applications. These applications often involve large bodies of code that have been developed earlier for individual nodes of the

multicomputer. A major challenge is to be able to reuse this code in the distributed application without significant changes.

2. Both the network and nodes of a network-based multicomputer are shared with other users, and the amount of available resources (computing cycles, network bandwidth, memory, ..) is unpredictable. As a result, dynamic load balancing is more important than on dedicated multicomputers, and monitoring tools that allow users to understand the behavior of their application are essential.
3. Having access to different types of computing systems is attractive for many applications, but a heterogeneous environment also creates the problem that users have to be familiar with the runtime environment, communication mechanisms and programming models of a potentially large number of systems. Programming tools are needed to give programmers more uniform access to these computing resources. For example, PVM [25] provides uniform access for applications using messages passing.

In Section 4 we discuss how these problems were addressed for several applications over Nectar.

## 3. High-performance networks as a multicomputer interconnect

We give an overview of the Nectar system, and we then discuss in more detail how its host-network interface and network interconnect address the performance, flexibility and reliability requirements described in Section 2. Since most Nectar hosts are workstations, we focus our discussion on workstation hosts.

### 3.1. Nectar system overview

Figure 1 gives an overview of the Nectar system [4]. The Nectar network consists of fiber-optic links and crossbar switches (HUBs). Hosts are attached to Nectar using Communication Acceleration Boards (CABs) and the system uses 100 Mbit/second fiber links and 16×16 HUBs. At its peak (1991), the Nectar system had 26 hosts, mostly Sun 4 workstations.

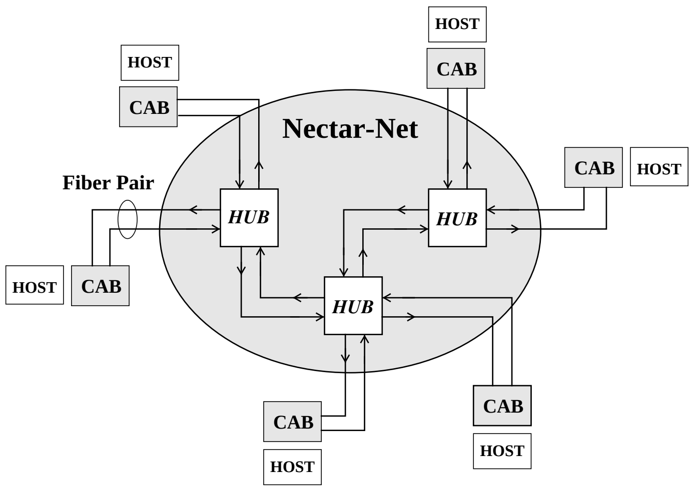


Figure 1. Nectar system at Carnegie Mellon

The advantage of switch-based networks (such as Autonet [52], HPC [24], S3.mp [50], Nectar and ATM networks [21]), compared with LANs based on a shared medium (such as Ethernet, token ring, or FDDI), is that for the same link technology, the aggregate bandwidth is much higher, since each attached computer has an exclusive link to the switch and can communicate with other systems attached to the same switch at the full bandwidth of the link.

The HUBs implement a command set that allows source CABs to open and close connections through the switch [4]. These commands can be used by the CABs to implement both packet switching and circuit switching. The command set also supports multi-hop routing and multicast. In the HUB, the latency to set up a connection and transfer the first byte of a packet through a single HUB is ten cycles (700 nanoseconds). The latency to transfer a byte over a connection is five cycles (350 nanoseconds), but transfers are pipelined to match the 100 Mbit/second peak bandwidth of each fiber link. The HUB controller can set up a new connection every 70 nanosecond cycle.

The CAB is implemented as a separate board on the host VMEbus and is connected to the network via a fiber port that supports data transmission rates up to 100 Mbit/second in each direction. Each CAB has 1 megabyte of packet memory, 512 kilobytes of program memory, a 16.5 MHz SPARC processor and hardware devices. The Nectar system software [20] consists of a CAB runtime system and libraries on the host that exchange messages with the CAB on behalf of the application. The CAB software performs protocol processing but since the system has support for multiprogramming (threads), it is possible to offload communication-oriented application tasks onto the CAB.

| Protocol | CAB-CAB Latency | Host-host Latency |
|---|---|---|
| Datagram | 97 | 169 |
| Request-Response | 154 | 225 |
| RMP | 127 | 213 |
| UDP over IP | 234 | 308 |

**Table 1:** Message latency in Nectar (microseconds)

The streamlined structure of the CAB software and the fast interconnect result in low communication latency. Table 1 shows the latency between applications running on two CABs and on two hosts using both proprietary (datagram, request-response, Reliable Message Protocol) and standard (User Datagram Protocol Internet Protocol) protocols [58]. The latency is under 100 microseconds for a message sent between threads on two CABs, and about 200 microseconds between processes residing in two workstation hosts. These performance results are similar to those of traditional multicomputers using similar technology [8, 5].

Figures 2 and 3 show the application-level CAB-CAB and host-host throughput for different packet sizes [20]. In the host-host case, the throughput is limited by the bandwidth of the VME bus. For TCP, when TCP checksums are not computed, the throughput between two CABs is over 80 Mbit/second for 8 kilobyte packets. When checksums are computed, TCP throughput drops to about 30 Mbit/second.

In the remainder of this section, we describe how we optimized the communication over Nectar, within the constraints imposed by the multicomputer systems architecture (Section 2.2.1). We first discuss the design of the network interface, which often determines the communication performance over high-speed networks, and we then discuss the impact of the multicomputer network requirements on the design.

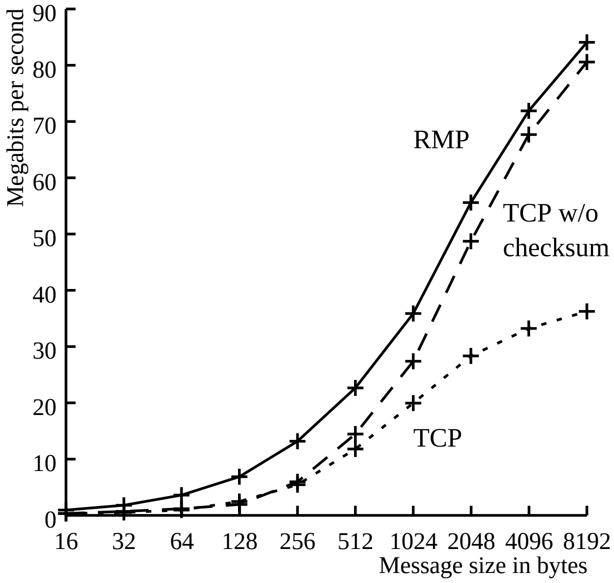


Figure 2. CAB-CAB throughput

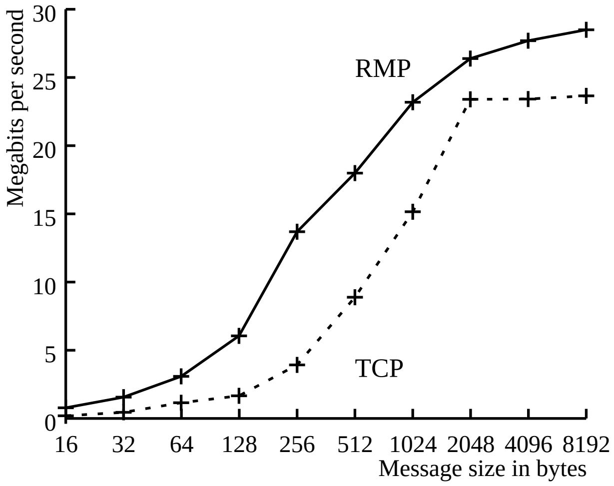


Figure 3. Host-host throughput

### 3.2. Network interface design

The most critical factor determining the efficiency of the network interface is where the network enters the memory hierarchy of the host. Figure 4 illustrates some possible designs. On workstations, the most common node found in a network-based multicomputer, the network interface is placed on the I/O bus, which is connected to the main memory-cache-CPU path through a bridge (4d). On tightly coupled systems, the network enters the node closer to the CPU: on the main memory bus (4c, e.g. Paragon), in parallel with the cache (4b, e.g. CM5), or at the register level (4a, e.g. iWarp). This close coupling of the network and the CPU on tightly-coupled multicomputers is a result of the fact that the node and the network interface are designed together, while they are designed separately and then ''integrated'' using an I/O bus for stand-alone computer systems.

Relegating the network interface to an I/O bus has a significant impact on network performance, since it fundamentally results in a path between the network and the CPU/memory that has higher overhead and lower bandwidth compared with a tightly coupled design. Since this suboptimal design (from the performance perspective) is not likely to change, we should optimize performance with this constraint. Figure 5 shows three ways in which data can flow between main memory and the network when sending messages; the data flow for receive is obtained by reversing the arrows. The grey arrows indicate the building of the message by the application; the black arrows are copy operations performed by the system.

The architecture depicted in Figure 5(a) is the network interface found in many computer systems, including most workstations. When a system call is made to write data to the network, the host operating system copies the data from user space to system buffers. Packets are sent to the network by providing a list of descriptors to the network

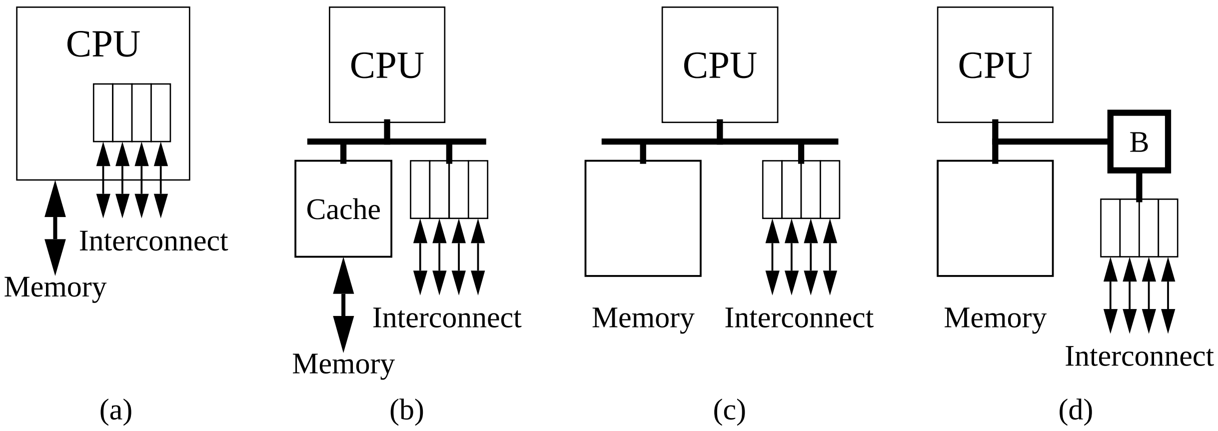


Figure 4. Alternatives for the placement of the network interface

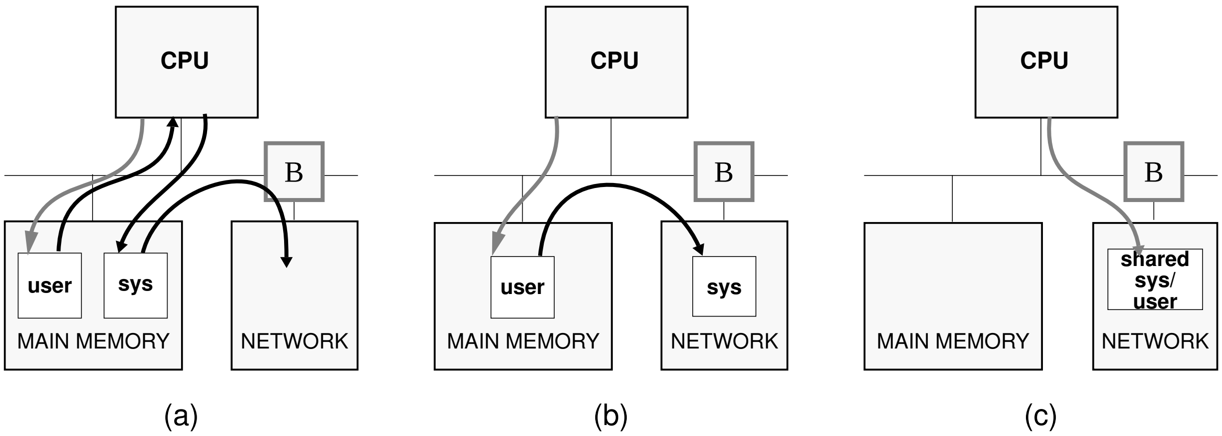


Figure 5. Design alternatives for data flow across the network interface

controller, which uses DMA to transfer the data to the network. The main disadvantage of this design is that four bus transfers are required for every word sent and received. This is not a problem if the speed of the network medium is sufficiently slow compared to the memory bandwidth, as is the case for current workstations connected to an Ethernet, but if the communication activity increases, the memory bus becomes a bottleneck.

Nectar reduces the number of transfers across the bus by moving the system buffers to the network adapter (Figure 5(b) and (c)). Figure 5(b) reduces the number of data transfers while maintaining an application interface where messages are specified using a pointer-length pair (copy semantics, e.g. Unix sockets [44]). This architecture is also implemented in gigabit Nectar [57, 36]. To support the host interface of Figure 5(c), the Nectar interface implements ‘‘buffered’’ send and receive primitives [56, 60]. With buffered send and receive primitives, the application builds and receives messages in message buffers that are shared with the system. The advantage of buffered sends and receives over socket-like primitives is that data no longer has to be copied as part of the send and receive calls [60].

Our experience with the Nectar system indicates that buffered primitives are faster for large messages, but that socket-like primitives are faster for short messages. The reason is that for short messages, the extra complexity of the buffer management for buffered primitives is more expensive than the cost of simply copying the data. In the Nectar system, the buffered primitives are more efficient for messages larger than about 200 bytes.

In Nectar, the shared buffers used in buffered communication are located in the network interface. This approach was attractive because host accesses to CAB memory across the VME bus are not significantly slower than (uncached) host accesses to host memory (1.1 microseconds), so the memory on the network interface could be viewed as an extension of host memory. On more recent workstations, high-speed synchronous busses [22, 51] couple the host more tightly to the I/O bus, but they require burst transfers to get good throughput. As a result, allowing applications to write individual words to the outboard buffer is unattractive on these systems and the shared user/system buffer should be kept in host memory and transferred to the network interface using burst transfers.

## 3.3. Nectar as a network

We discuss the impact of required multicomputer network features such as robustness on performance [41].

### 3.3.1. Robustness

Although the error rates in modern fiber-optic networks are low, it is still necessary to implement end-to-end checksums and reliable protocols above the physical layer. These protocols are responsible for detecting corrupted and lost packets, for recovering from these errors by retransmission, and for flow and congestion control [14]. The network must be able to recover from errors quickly, otherwise the performance of the multicomputer will degrade rapidly. In contrast, traditional multicomputers typically rely on parity or error correcting codes to detect and recover from bit errors, since their interconnects have extremely low error rates. If an uncorrectable bit error is detected, the entire system or the application fail.

Error recovery in software adds overhead in a number of ways. First, there is protocol processing overhead (e.g. TCP processing); for optimized protocol implementations, this overhead can be as low as 200 instructions per packet [18, 58]. A second component is the checksum calculation; since this involves touching the data, the cost is relatively high (Figure 2), but hardware support can eliminate this cost. Finally, there is extra buffer management overhead since a retransmit copy of the data has to be kept; this cost is hard to evaluate and depends on the communication interface used by the application. Note that even on tightly-coupled systems there will be some protocol processing overhead, for example to implement end-to-end flow control between the sender and the receiver.

### 3.3.2. Flexibility

Since LANs typically do not have a regular topology, and nodes may be added and removed while the system is running, LANs need more flexible routing and configuration support than tightly coupled systems. Nectar uses deterministic source routing, managed in software by the CABs [41]. This has the advantage of making the switches very simple as they need no control software or hardware for routing, but it does place an extra burden on the CAB datalink software since it has to manage routing tables (complexity grows with the scale of the network) and adds a fixed routing overhead for each packet sent (1-2% for 1 kilobyte packet). Furthermore, for larger networks, deterministic routing has the problem that as packets must traverse more HUBs, it becomes harder to establish and end-to-end connection, because the probability of encountering a busy link increases rapidly with the number of hops. Neither problem was an issue in the small Nectar prototype system.

The software approach used in Nectar is in contrast with, for example, Autonet [52], which explored the problems involved in managing large, general, switch-based networks, and supports automatic reconfiguration when host or switches are added to or removed from the network. Adding these features results in additional hardware (including switch control processors) and software to manage the network, but simplifies the management of larger networks.

### 3.3.3. Safety and security

Since both the nodes and the network of a multicomputer are shared with other users, it is important to make sure that the different users cannot access each other’s messages or address spaces. This is typically done by having all communication go through the operating system, which controls access to data and to the network. This adds overhead in a number of ways. First it adds system call overhead, typically a few tens of microseconds or less. Second, the OS spends time verifying access rights, (de)multiplexing between multiple users, protocols and possibly networks; this overhead is very implementation dependent. Finally, since OS communication functions are shared by many applications, they tend to be general, and as a result, less efficient than functions that are optimized for a specific task.

In Nectar, communication bypasses the host operating system, and the functions that are normally performed in the OS are now performed by the CAB. This results in more efficient communication, since the CAB is optimized to support communication [20]. In this architecture, the CAB is responsible for providing the necessary user protection, with some help from the OS. Although this was not implemented, the CAB provides the necessary support to provide protection. The idea is that different applications use different parts of CAB memory to store messages and message descriptors (mailboxes). The virtual memory system of the host operating system can limit access of each application process running on the host to the CAB memory area that was allocated to it. For application threads executing on the CAB, the memory protection hardware on the CAB [4] can provide the protection.

Computing over networks also raises the issue of security: a malicious third party could obtain the results of the computation by intercepting messages, or could even influence the result by replacing messages that are exchanged by cooperating tasks. In many cases this is not a concern, but for some applications security is extremely important, e.g. business applications going over a public network, and end-to-end security measures are needed. No security features were implemented in the Nectar system. An overview of the issues can be found in [34].

#### 3.3.4. Multicast

High-bandwidth multicast is a valuable feature for multicomputers because multicomputer applications make significant use of multicast for distributing large amounts of data [42]. Network support for multicast allows the sender to send multicast messages once instead of many times, thus reducing both the communication overhead on the sender and message latency. While there are some exceptions, e.g. nCUBE-2, CM5 control network, and the T3D barrier hardware, most tightly-coupled multicomputers do not have hardware multicast: since they have a dedicated and regular interconnect, it is possible to implement in software efficient multicast algorithms with a predictable execution time.

The multicast used by multicomputer applications is different from the multicast or broadcast typically provided in general-purpose networks. The multicast used for network management in general-purpose networks is typically unreliable (i.e. there is no guarantee that all members will receive the message), and the members are not known to the sender. This form of multicast is typically used to retrieve information from (replicated) servers whose location is not known.

The multicast implemented on Nectar is reliable, and the multicast group is created by the application on the sending node. It is implemented by allowing a CAB to have multiple open outgoing connections within one switch. Thus a CAB can set up a multicast connection and send the same data packet to multiple destination CABs. The source routing used in Nectar makes multicast simple as each host can independently establish its own multicast groups. In networks with switch-based routing, routing information for multicast connections must be entered into the routing tables in the switches. This adds complexity to the switch, but the approach is more scalable.

## 4. Programming a network-based multicomputer

Network-based multicomputers pose new challenges for programmers. These include working in a dynamic, heterogeneous environment and dealing with applications involving large bodies of code. In this section we first discuss our strategy for moving applications onto network-based multicomputers, and we then describe three of the applications that were implemented on Nectar.

### 4.1. Methodology for parallelizing large applications

Implementing large applications on Nectar-like architectures involves considerations along many dimensions: partitioning the application, mapping and distributing global data, ensuring data consistency, performing load balancing, and possibly providing fault tolerance for applications with long execution times. Ideally, these tasks are performed using programming tools that support distributed computing and that can significantly reduce the complexity of parallel and distributed computing. Examples of tools include parallelizing compilers [61, 15, 23], object-based runtime tools [16, 6, 26, 3, 48], and languages and environments for specific application domains [39, 28, 64].

While many distributed applications benefit from the use of tools, today's tools cannot yet, in general, characterize the way irregular programs update complex data structures or the complex control flow of a substantial application. For this reason, we use a task-based methodology based on the separation of *application code* and *system code* [42], which is an intermediate solution between automatic parallelization and rewriting the entire application. When parallelizing an application starting with a serial implementation, the first step is to partition the application in units of work, called *tasks*; this collection of tasks makes up the application code and it has all the application-specific knowledge. The code for each task is a sequential program to be executed on a single node. The next step is to identify the synchronization and communication requirements between the tasks. Their implementation makes up the system code. By implementing the system code as a separate module, it is possible to reuse the system code for other applications.

Separating application and system code has several advantages. First, it naturally supports reuse of existing application code. This is very important since the cost of rewriting and maintaining different versions of large applications is prohibitive. Second, this approach is not limited to a single communication model: different modules for system code can be provided to support a range of synchronization and communication styles. Finally, by implementing the same system modules on different architectures, porting applications across these architectures becomes easier. The implementations of the system modules on the different architectures can be optimized for the architecture. Hopefully this task-based approach is also a useful step towards a more automated approach.

## 4.2. COSMOS: a logic simulation application

We describe a parallel implementation on Nectar of COSMOS [13], a high-performance logic simulator developed at Carnegie Mellon by Randy Bryant and his associates. The key feature of COSMOS is that it compiles the circuit into executable code. The COSMOS compiler partitions the circuit into a number of channel-connected subcircuits, which it then translates into C language evaluation procedures and declarations of data structures that describe the connections between the subcircuits. This circuit code is then compiled and linked with a COSMOS kernel and user interface to generate the simulator program.

Circuits are simulated one clock *phase* at a time, and the simulation of a phase consists of a number of *simulation steps*. During each step, subcircuits whose input signals have changed since the previous step are evaluated; for the first step of each phase, external signals such as clocks can change. The simulation of a phase is finished when all signals are stable, so the number of steps in a phase depends both on the circuit and on the input signals.

### 4.2.1. Mapping COSMOS onto Nectar

In the COSMOS implementation on Nectar, called Nectar-COSMOS, subcircuits are statically distributed over the Nectar nodes [35]. The connectivity information for the subcircuits is used to determine what signals have to be communicated between the nodes on every simulation step. Each node runs a copy of the simulator and the COSMOS code corresponding to the subcircuits assigned to the node. After each simulation step, the node sends the output signals to other nodes that need them (Figure 6(a)). A node can start on the next simulation step once it has received the necessary input signals from the other nodes. Distributed termination detection is used to determine the end of a simulation step. Because all the nodes work on one phase at a time, the simulation time for a phase is determined by the slowest node. As a result, the circuit should be distributed across the nodes in such a way that the simulation time on each node is about the same.

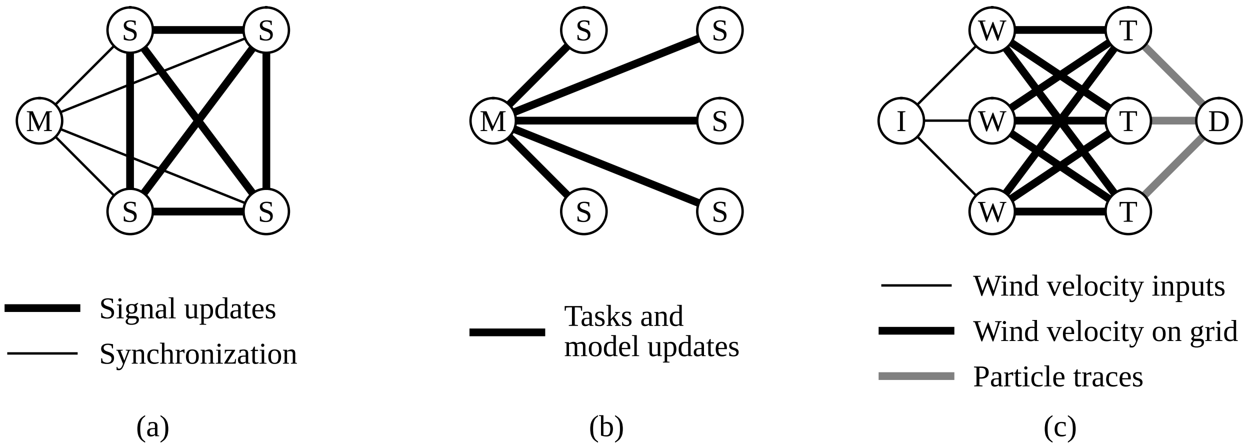


Figure 6. Structure of Nectar applications: (a) COSMOS, (b) NOODLES, (c) Air pollution

Nectar-COSMOS reduces the cost of synchronization by overlapping computation with communication. On each node, the system distinguishes two types of subcircuits − *boundary modules* and *interior modules*. Interior modules are only connected to subcircuits on the same node, while boundary modules also have connections to subcircuits on other nodes. Each Nectar node first simulates the boundary modules, since the simulation of these subcircuits will produce results needed by other nodes for the next simulation step. After this is done, the host simulates the interior modules, while the CAB sends out the results produced previously by the simulation of the boundary modules.

#### 4.2.2. Measurements

To evaluate the performance of Nectar-COSMOS, we simulated a 36×36 maze routing chip implemented using dynamic CMOS, and consisting of 245000 transistors. Table 2 shows the speedup on a network of dedicated Sun 4/330 hosts. The chip could not be be simulated on a single workstation because we were not able to generate the simulator due to memory limitations, so we used a 2 node implementation as the basis for our comparison. We observe a close to linear speedup up to 6 nodes. A more detailed analysis of the results shows that our technique for hiding communication overhead is very effective: most subcircuits are interior circuits and as a result, there is a significant overlap of communication and computation. For example, with 5 nodes, less than 1% of the modules are boundary modules, and only 15% of the of time is spend on communication.

The results for the maze router chip show that it is possible to speed up circuit simulation using a Nectar-like system, although the synchronization and communication overhead limits the speedup for larger numbers of nodes.

| Number of nodes | 2 | 3 | 4 | 5 | 6 | 9 |
|---|---|---|---|---|---|---|
| Time/cycle (seconds) | 0.254 | 0.179 | 0.144 | 0.121 | 0.104 | 0.080 |
| Speedup | 2 | 2.84 | 3.52 | 4.20 | 4.90 | 6.32 |

**Table 2:** Nectar-COSMOS speedup for 36×36 maze router

A study of more realistic circuits such as the iWarp component indicates that these circuits have more potential parallelism, since they are larger, but they have the drawback that they are not regular, thus making it harder to distribute work evenly across the network nodes. During simulation, peak activity can move around in the circuit, complicating load balancing [35].

Even though COSMOS is a significant application (about 50,000 lines of code in the COSMOS compiler chain and kernel), the porting of COSMOS to Nectar was relatively easy. The reasons are that sequential COSMOS already implemented the main data structure partitioning (circuit), and that the sequential implementation already existed on the same workstations that form the Nectar nodes. As a result, a mapping where each node runs a copy of the original program (a simulator), and operates on part of the input data (circuit) is natural and required few changes to the original program. The only change is that the simulator now gets input signals and returns output signals in a slightly different format. The main effort in Nectar-COSMOS was in implementing the system code, which communicates the signals between the nodes and supports communication between the master and the simulators for initialization and termination detection.

### 4.3. NOODLES: a geometric modeling application

NOODLES is a geometric modeling system [17] developed by Fritz Prinz' group at the NSF Engineering Design Research Center at Carnegie Mellon University. NOODLES models objects of different dimensions as a collection of basic components such as vertices, edges, and faces. Applications of NOODLES include knowledge-driven manufacturability analysis and rapid tool manufacturing. We describe a Nectar implementation of NOODLES, called *Nectar-NOODLES*, developed jointly by the EDRC and School of Computer Science at Carnegie Mellon.

The basic operation in NOODLES is the *merge* operation. Using this operation, complicated objects can be built by intersecting a pair of simpler ones. The merge operation does a pairwise geometric test on components in both input objects, and it breaks up components if they intersect. Geometric tests may yield updates to the database of the models, which influences what tests have to be done in subsequent stages. For example, if two edges intersect, they will be replaced with four non-intersecting edges, which will be used in later tests. Thus, both the number of tests and their cost are data-dependent.

#### 4.3.1. The parallelization of NOODLES

Updates of the Noodles database are intrinsically sequential, but the geometrical tests that produce these updates can be performed in parallel. When merging two models, each with $n$ components, the number of updates to the database representing the models is $O(n)$ or $O(n \log n)$ [17], while the number of geometric tests is $O(n^2)$. Thus, for large models, the speedup resulting from parallelizing geometric tests can be substantial. The goal of Nectar-NOODLES is to allow interactive use of NOODLES, even for large designs. To maintain the consistency of the models, geometric tests have to be performed in stages depending on the type of the comparison test, i.e. vertex-vertex, vertex-edge, .... The different stages have to be performed sequentially, and they are separated by barriers in Nectar-Noodles.

Because the execution time of the tests and updates in NOODLES is data dependent, the distribution of work across Nectar nodes is done dynamically at runtime. Nectar-NOODLES uses a central load balancing strategy: a master node keeps a central task queue, and slave nodes execute tasks that they receive from the master (Figure 6(b)). Each task consists of a series of basic geometrical tests, where each basic test consists of comparing one component of one model with a class of components in the other model, for example, an edge of one model with all vertices of the other model. The dynamic load balancing function executes on the CAB of the master node, allowing it to respond faster to requests than if it were to execute on the workstation. The interaction between the master and the slave is pipelined to hide communication latency: the slave asks for the next task while it is handling the previous task.

Nectar-NOODLES cannot rely on a straightforward partitioning of the input data space, as was done in Nectar-COSMOS. NOODLES uses an intricate data structure with a large number of pointers, and distributing this data structure over the Nectar nodes would require a total rewrite of NOODLES. Nectar-NOODLES avoids this by giving each Nectar node a copy of the geometric models. The copies of the models on the nodes are kept consistent by updating all the models at the same time and in the same order at the end of each stage: the master collects updates during the stage, and multicasts them to all the slaves at the end of the stage, where they are executed. To allow nodes to exchange updates, global names were added to each entry in the NOODLES database. The NOODLES code was not changed: it still operates on its original data structure using local pointers (which can be different on all the nodes). The translation between local pointers and global names is done using a table lookup at the interface between NOODLES and the system code. Note that because of the replication of the main data structure, we lose one of the benefits of using a multicomputer (more memory).

As in the case of COSMOS, NOODLES was mapped onto Nectar by running a version of the sequential program on every node. Again, few changes had to be made to the existing sequential application, which has about 12,000 lines of code. Almost all the code that is specific to the parallel implementation is in a separate module.

#### 4.3.2. Measurements

Table 3 shows the speedup for Nectar-NOODLES merging two models, each consisting of two spheres. Each of the models has about 3500 components. The hosts are dedicated Sun 4/330 workstations, and the speedup is relative to the sequential version of NOODLES. The speedup is relatively low because the single node version of Nectar-Noodles is almost 30% slower than the sequential Noodles. The reason is that some of the geometric tests are performed more than once in the parallel implementation. This duplication occurs because updates to the database are delayed until the end of each stage, and as a result, redundant operations are not deleted in time. This illustrates an important difficulty in parallelizing code that uses complex data structures.

The different columns show the results for various task sizes, starting with 1 test per task to 30 tests per task. The size of a test ranges from 3 milliseconds in the early stages to as high as 50-150 milliseconds in the later stages (6 milliseconds average). We observe a similar speedups for all task sizes, even for the smallest task size. As the task size increases, the speedup drops slightly, and this effect becomes stronger as the number of nodes increases. The reason is that the load balancing becomes less effective in the later stages, each of which has a smaller number of larger tasks.

| Number of nodes | 1 | 5 | 10 | 15 | 30 |
|---|---|---|---|---|---|
| 2 | 1.52 | 1.52 | 1.51 | 1.50 | 1.49 |
| 3 | 2.23 | 2.21 | 2.21 | 2.20 | 2.17 |
| 4 | 2.84 | 2.85 | 2.83 | 2.79 | 2.73 |
| 5 | 3.46 | 3.45 | 3.41 | 3.38 | 3.28 |
| 6 | 4.06 | 4.02 | 3.97 | 3.89 | 3.74 |
| 7 | 4.60 | 4.55 | 4.50 | 4.36 | 4.21 |
| 8 | 5.04 | 5.02 | 4.94 | 4.78 | 4.54 |

**Table 3:** Nectar-NOODLES speedup for different task sizes (expressed in tests per task)

The load balancing code used in Nectar-NOODLES was developed as a separate package. This allowed a second application, ray tracing, to be ported very quickly to Nectar. The task size in this application is about 400 microseconds and the speedup curve flattens at about 5 nodes with a speedup of 4. This shows the limitations of a central load balancing scheme: the master node can handle a new request about every 100 microseconds and becomes a bottleneck with a large number of nodes or small task sizes.

### 4.4. Simulation of air pollution in Los Angeles

Flow field problems, such as weather forecasting and tracking of spills, are computationally intensive and can benefit from parallel processing. We ported a program that tracks pollutant particles in the atmosphere of the Los Angeles area written by Ted Russell and Greg McRae to Nectar. The input to the program are the wind velocities recorded at 67 weather stations around the Los Angeles (LA) area once every hour. The program calculates the traces of pollutant particles that are released in the area. The LA simulation program is an example of a medium size static application (2500 lines of FORTRAN).

Computing the particle traces given the wind conditions is a two phase process. The first phase consists of computing the wind velocity at each point of a 80×30 grid on the geographic area concerned, for every hour, given the measurements from the weather stations and precomputed weights. This problem involves interpolating from the measurements, as well as solving the conservation of mass equations across the grid. In the second phase, each particle is tracked as it moves about the grid; this requires an interpolation in both space and time. The time step used in this phase is 30 seconds.

#### 4.4.1. Parallel implementation over Nectar

When partitioning this program over Nectar, we maintained the structure and code of the original sequential program as much as possible. In the first phase, a task consists of calculating the wind velocities at each point in the grid for a given hour (Figure 6(c)). The second phase is parallelized by partitioning the particles among the processors: each processor tracks the motion of a set of particles for the duration of the simulation. The two phases are pipelined: while some processors are tracing particles at time T, other processors are calculating the wind velocities for the following few hours. In the initial implementation, load balancing was done statically: the hours and particles were divided among the processors before computation begins. Table 4 shows the speedup for the parallelized particle tracking code on Nectar, relatively to the sequential code, using dedicated Sun 4/330 workstations as hosts.

| Number of nodes | Time (seconds) | Speedup |
|---|---|---|
| 1 | 125 | 1.0 |
| 2 | 65 | 1.9 |
| 3 | 44 | 2.9 |
| 4 | 34 | 3.7 |
| 6 | 23 | 5.4 |
| 8 | 19 | 6.6 |

**Table 4:** Speedup for LA pollution simulation

The amount of data to be communicated by this application increases linearly as more processors are added to the system, since every processor in the second phase must have all the information computed by the processors in the first phase. With the use of the Nectar (hardware) multicast facility between phases one and two, the communication overhead per simulated hour remains constant for each node. This makes it possible to achieve a speedup, although, as the number of processors increases, the constant communication overhead will eventually limit the number of nodes that can be used effectively for a fixed problem size.

The total number of bytes sent can be reduced by using a mapping in which the grid is partitioned across the processors. For first phase, each processor calculates the wind velocities for its part of the grid, for all hours, and for the second phase, each processor traces the particles in its area at any given time. This mapping significantly reduces the communication bandwidth requirements, but it has several disadvantages: 1) it requires more fine-grained interactions between the processors working on the same phase, 2) it is more complicated to implement, because the structure of the original sequential program is changed dramatically, and 3) load balancing becomes much more difficult. Our original mapping preserves the program structure and is more appropriate for a network environment: there is a good match between the resulting coarse-grained parallelism and an architecture with a small number of powerful nodes. However, it needs a high speed network.

#### 4.4.2. Adding dynamic load balancing

Unfortunately, since the distribution of work is done statically, performance degrades quickly if the amount of work assigned to each node does not match the processor speed, or if the load by competing processes on the nodes changes during the execution [11]. To avoid this degradation in performance, we added dynamic load balancing to phase two of the application. Each slave processor normally spends equal amounts of time on phases one and two, so providing load balancing for only one phase of the application is sufficient if differences in competing load between nodes are limited to 50%.

| Number of nodes | Time (seconds) | Speedup |
|---|---|---|
| 1 | 129 | 1.0 |
| 2 | 66 | 1.9 |
| 3 | 47 | 2.9 |
| 4 | 35 | 3.7 |
| 6 | 25 | 5.2 |

**Table 5:** Speedup with dynamic load balancing in dedicated environment

Load balancing is implemented using a central load balancer that monitors the progress of each of the slave nodes; if the difference in simulated time on the nodes becomes too large, it moves particles from slow nodes to fast nodes. If the network environment does not change, each processor traces its particles with minimal disruption: the only overhead consists of occasionally reporting its progress to the load balancer. Table 5 shows the performance of the application with dynamic load balancing in a dedicated environment (i.e. when dynamic load balancing is not needed): a comparison with Table 4 shows that the overhead introduced by the dynamic load balancer is small, at least up to 6 nodes.

Showing that the dynamic load balancer is effective in redistributing the load requires information on the dynamic behavior of the application [11]. Figure 7 shows the effect of dynamic load balancing using the BEE monitoring tool [47]. The three windows (left to right) show for each node, the simulation time reached, the load (number of particles assigned to the node), and accumulated CPU time. Even though all nodes are Sun4/330 workstations, the application is not well balanced if a static partitioning is used (top screen dump) because **zinfandel** and **gamay** are also running other jobs, as is shown by the CPU time view. As a result, they run behind. The lower part of Figure 7

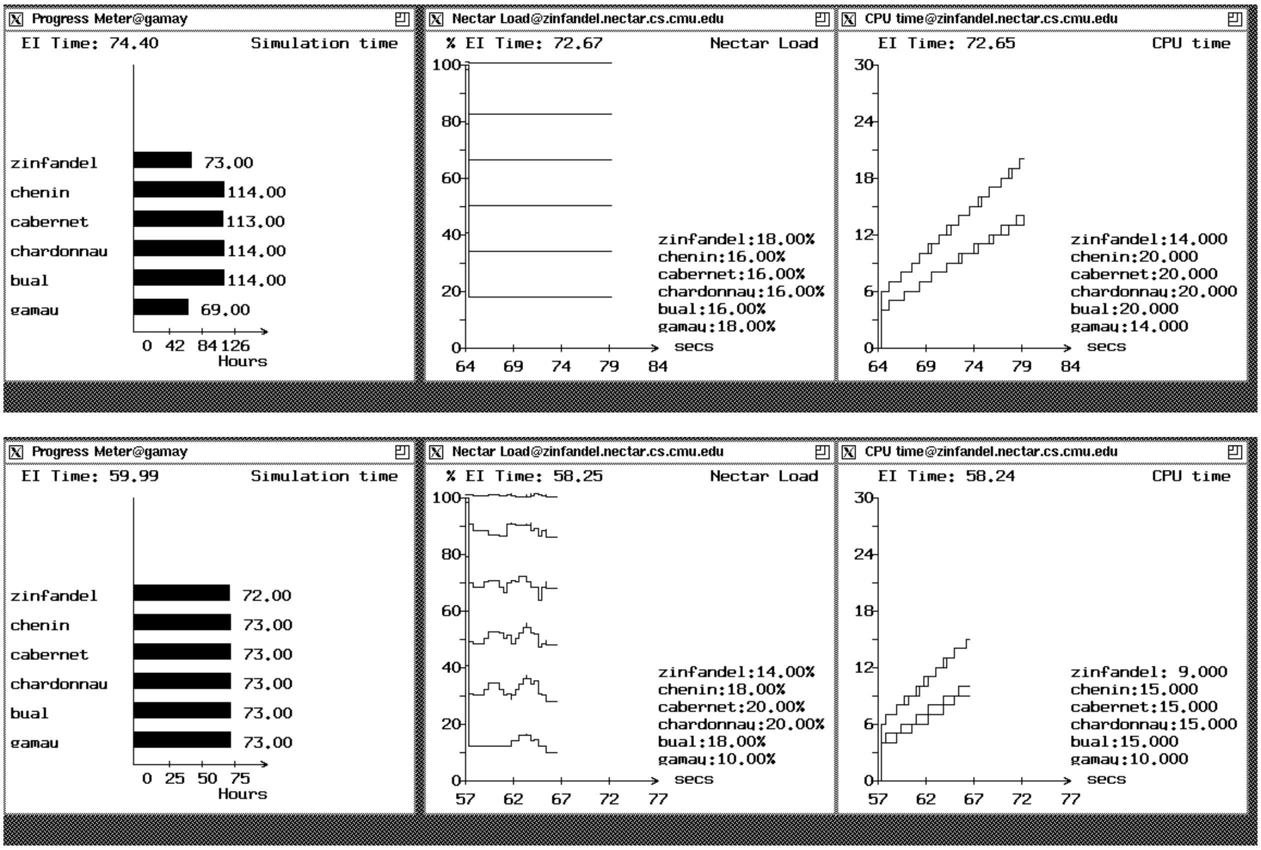


Figure 7. Nodes with Different Computational Loads without (top) and with (bottom) load balancing

shows how the load balancer takes care of the problem by moving particles from **zinfandel** and **gamay** to less loaded nodes: the left window shows that all nodes are simulating the same time step, while the middle window shows how tasks move between nodes.

### 4.5. Conclusion

The examples discussed in this section show that it is possible to successfully distribute applications over a network-based multicomputer. These applications differ both in the application domain and in the programming model, and they illustrate the versatility of multicomputers. Even though the number of nodes in the experiments is limited, either because of constraints in the application or in the system available at the time, the speedups indicate that multicomputers are a viable architecture. While these measurements were collected using Nectar, we expect to see similar results over (commodity or proprietary) networks with similar performance characteristics. Several other large applications, not reported in this paper, have been successfully ported to Nectar. These include a parallel solid modeler from University of Leeds (called Mistral-3), distributed algorithms of finding exact solutions of traveling salesman problems [38], image processing using Adapt [65] and a chemical flowsheeting simulation [33].

Implementing applications on a network-based multicomputer is a non-trivial effort, and programming tools that simplify that task are needed. As described earlier in this section, programming tools for both tightly-coupled and loosely-coupled distributed-memory systems are an active area of research. In the context of the Nectar system, we have demonstrated tools in three critical areas: monitoring tools that help the programmer understand the behavior of the application [47, 12], support for data sharing across the network [48, 49], and load balancing tools that help in distributing work to make efficient use of the cycles on the nodes [67, 66, 67, 53, 54, 55].

## 5. Traffic characterization

An important part of any system design is understanding how the system will be used. In the case of networks this corresponds to characterizing the traffic it will carry. Based on our application experience in Nectar, we can make a number of observations on how multicomputing applications use networks. This sheds some light both on the impact of this class of application on network behavior and on the performance multicomputer applications can expect from networks compared with dedicated interconnects.

### 5.1. Traffic considerations for multicomputer applications

Traditional network applications involve mainly point to point communication, i.e. RPC, telnet, ... More complex patterns that involve several nodes, such as multi-RPC used in file servers, do occur, but they are not very common and they account for only a small percentage of the transmitted data. Multicomputer applications on the other hand use many communication operations that involve several or all nodes used by the application. In the applications we ran over Nectar, we identified a small set of common collective communication operations. The patterns include:

**individual** a single message between two nodes, unrelated to other messages

**request** single request-response interaction, unrelated to others

**server** all nodes have request-response interactions with a single node (server)

**multicast** one node sends the same message to all other nodes

**distribute** one node sends a different message to all other nodes

**collect** one node collects a message from all other nodes

**exchange** all nodes exchange data - the two cases we encountered are that every node performs a multicast (m-exchange) or that every node does a distribute (d-exchange)

Applications expect all communication operations to be reliable. The first three operations are similar to traditional LAN communication, while the last four are collective communication operations that are specific to multicomputing. Some message passing libraries, e.g. MPI [46], support collective communication as primitive operations.

Table 6 lists the features of the main communication operations used in a number of Nectar application. For each operations, Column 1 gives a short description of the function of the operation in the application and the frequency (Rare or Frequent) of the operation. While the exact numbers will differ from run to run, depending on the application (e.g. data set) and system (e.g. dynamic load balancing) parameters, we estimate that in general the communication operations listed in the table account for more than 99% of the traffic in the applications, while the Frequent patterns are responsible for more than 90% of the data. Column 2 describes the communication pattern, as described above, while columns 3 and 4 describe the status of the sender and the receiver after and before the communication operation (discussed in Section 5.3). The first three applications in the table have been discussed in the previous section. We briefly describe the programming model used by the other applications:

- *Simplex* is a parallel simplex algorithm. Searching and updating rows is done in parallel, but the selection of the pivot row is done on a central processor.
- *TSP* [38] is a traveling salesman problem that uses dynamic load balancing to balance the load during the space search. The current best solution is broadcasted periodically to prune the search.
- *Flow sheeting* [33] simulates a simple model of a chemical plant. It uses simple-master model.
- *BEE* [47] is a monitoring tool that collects monitoring information from all the nodes in the multicomputer.
- *Adapt* [64] is a mid-level image processing language. It also uses a simple master-slave programming model where slaves ask for tasks. Results are sent back to the master at the request of the master to avoid congestion.
- *Aroma* [48] is a distributed object system for mostly data parallel computations. Most messages are updates or requests for data assigned to other processors.
- *Initialization* is a typical initialization phase that most applications go through. It consist of a master node distributing or multicasting initial data to all the nodes.

Table 6 shows that most of the operations are collective communication operations. This high percentage of collective communication operations has a significant impact on the characteristics of the traffic seen by the network. Specifically:

- **Communication volume**: a single application can use a substantial fraction of the network bandwidth during a collective communication operation, since several nodes exchange usually large blocks of data at the same time as quickly as possible. This is different from the traditional view that network traffic is the sum of a large number of independent traffic streams.
- **Locality in destinations**: there is a lot of spatial locality in the network traffic while collective communication takes place. There is also some locality with traditional applications, e.g. file servers will typically be responsible for the bulk of the network traffic, but this is a static traffic pattern that administrators can plan for. In the case of multicomputing, the set of high-traffic nodes is potentially much larger and the hot spots will change over time. Note that applications often can (and should) address the issue of avoiding hot spots within the application. For example, in an exchange, not all nodes should start sending messages to the same node. However, this does not eliminate the high spatial locality in the network traffic.
- **Locality in time**: communication by different nodes is synchronized because the way the application is structured: applications often have compute cycles separated by periods of intensive collective communication. This results in bursty traffic. Applications often try to distribute communication over time, but this is often difficult due to data dependencies.

These observations clearly have an impact on the design and evaluation of networks. Performance benchmarks that use point-to-point throughput and roundtrip latency are not very representative of distributed computing applications. Similarly, network throughput and latency studies that assume random, uncorrelated traffic are not at all representative of multicomputing traffic. Note also that the main requirement is that applications can **burst** data at high rates; the average bandwidth requirements of the application are much less relevant.

| Purpose of Communication | Traffic Pattern | Sender (after) | Receiver (before) |
|---|---|---|---|
| Cosmos: | - | - | - |
| R master-to-slave | multicast | busy | waiting |
| R slaves-to-master | collect | waiting | waiting |
| - | individual | waiting | waiting |
| F slave-slave updates | d-exchange | limited busy | limited busy |
| Noodles: | - | - | - |
| R slave-to-master updates | individual | busy | server |
| R master-to-slave updates | multicast | busy | waiting |
| F tasks-to-master | server | limited busy | server |
| - | - | busy | limited busy |
| LA simulation: | - | - | - |
| F wind velocity updates | multicast | generator | limited busy |
| F grid updates | m-exchange | busy | limited busy |
| F particle trace info | collect | busy | limited busy |
| Simplex: | - | - | - |
| F master-to-slave command | multicast | limited busy | waiting |
| F slave-to-master | collect | waiting | limited busy |
| F master-to-slave pivot | multicast | waiting | waiting |
| F slave-to-slave | multicast | busy | waiting |
| TSP: | - | - | - |
| F task requests | server | limited busy | server |
| - | - | busy | limited busy |
| F pruning information | multicast | busy | limited busy |
| Flow sheeting: | - | - | - |
| F master-to-slave | multicast | waiting | waiting |
| F slave-to-master | collect | waiting | waiting |
| BEE: | - | - | - |
| F client-to-server | individual | busy | server |
| R server-to-clients | requests | waiting | busy |
| Adapt: | - | - | - |
| F slaves-to-master | server | limited busy | server |
| - | - | busy | limited busy |
| F master-slaves | request | waiting | busy |
| Aroma: | - | - | - |
| F updates | individual | busy | limited busy |
| F updates replicated data | multicast | busy | limited busy |
| F requests | request | limited busy | busy |
| Initialization (typical): | - | - | - |
| R master-slave | distribute | busy | waiting |
| R master-slave | multicast | busy | waiting |

**Table 6:** Traffic characterization of Nectar applications

Finally, given the nature of collective communication, switch-based networks are very attractive and an effective way of increasing the network bandwidth, since they exploit the parallelism in the communication operation. Inversely, shared media will form a bottleneck since the high locality in time results in extremely poor peak aggregate performance. For example, over an Ethernet, the high locality in time will result a high collision rate and in poor communication in the phase of the application when communication is most critical. The availability of switch-based networks is one of the reasons why network-based multicomputers have become attractive for a wide class of applications.

## 5.2. Impact on applications

Applications programmers should be aware of some of the differences between networks and dedicated interconnects since these differences have an impact on communication behavior experienced by the application. First, programmers should expect that network latencies are somewhat larger, as discussed in Section 3.1. Furthermore:

- since the network and end systems are shared with other users, the latency is less predictable and more variable than on a dedicated interconnect;
- the bursty nature of the multicomputing network traffic also increases latency, since packets are likely to enter a busy network; and finally
- the nature of several of the collective communication operations (combine and exchange) is such that the duration of the operation corresponds to the maximum delay encountered by any of the packets involved in the operation. Many performance evaluation studies focus on the optimal or mean delay, which is clearly better than the maximum delay, especially in networks where communication delays are very variable.

To illustrate the impact of variable latency on collective communication we implemented and evaluated a simple collective communication operation consisting of a *multicast* of a one word message followed by a *collect* of a one word message. We repeatedly ran the test using 1 master and 10 slaves over the departmental network consisting of bridged Ethernets at CMU. In total, about 5000 transaction were measured during a 24 hour period. Figure 8 summarizes the result. 90% of the transactions took 10 milliseconds or less and the median transaction time is 5 milliseconds, which is about 10 times the average cost of a point-point remote procedure call. However, the spread is very high. For example, 1% of the transactions took over 1443 milliseconds and 6 transactions took 3 seconds. As a result, the average time is 53 milliseconds, which is 10 times higher than the median (5 milliseconds) and minimum (4 milliseconds) time.

The naive conclusion is that applications should avoid collective communication, but this is difficult since it is often the result of using a natural mapping of an application to a multicomputer. The next best thing is to try to hide as much of the communication latency as possible by overlapping communication and computation. All three applications described in the previous section do this, and this is one reason for their good speedups. Specifically:

- In COSMOS, the exchange of new signal values is overlapped with the computation of signal values that do not affect external signals, i.e. with computation that is completely local to the node.
- In Noodles, requests for new tasks are overlapped with the execution of the previous task, i.e. the interaction between the master and slaves is pipelined.
- In the pollution simulation, pipelining of the computational stages allows overlap between communication and computation.

Note that it would be very beneficial to be able to build networks with more predictable latency. This is however a very difficult task and the very nature of the multicomputing traffic (very bursty) makes this even harder. Some

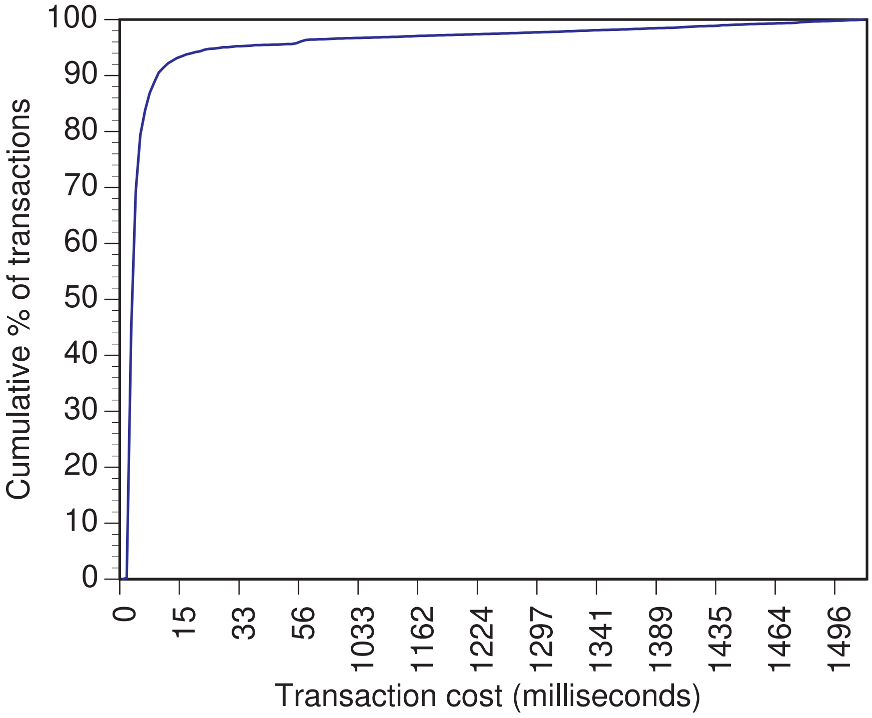


Figure 8. Cost of collective communication

emerging network technologies such as ATM [21] have the potential to make this possible, both because the small cell size makes the network more predictable, and because it allows guaranteed services using resource reservation in the network for critical data streams.

### 5.3. Connecting to hosts

The two last columns in Table 6 indicate what the senders and receivers are doing while the communication operation takes place. This is useful both to find out the level of overlap of communication and computation, and to learn what type of communication primitives are useful. Senders can be in one of four modes after a message has been sent:

**waiting** waiting for a response to that specific message.

**limited busy** can continue doing work that is unrelated to the message for limited time, but after a while, the response to the message is needed to make progress.

**busy** can continue doing work that is unrelated to the message forever, so no critical response is expected (the message is a one way message).

**generator** sender has nothing to do; an example is a data source.

In the first two cases, the sender expects a response to the message, and latency is very critical. In the latter two cases, no response is expected, and the message is an unsolicited or solicited piece of information that the sender provides to the receiver; latency is not critical to the sender, but it might be to the receiver. In the busy cases, the application is successfully overlapping communication and computation.

The receiver can be in the following state when the message arrives:

**waiting** waiting for that specific message.

**limited busy** expecting that specific message, but it can do unrelated work for limited time, after which it will be waiting.

**busy** is doing unrelated work, which has to be preempted to handle the incoming message.

**server** the receiver is in general waiting for messages, i.e. it is a server handling requests or collecting information. It is idle when not handling incoming messages.

These categories are similar to the ones for senders, and they have the same characteristics: the first two modes correspond to request-response interactions and latency is critical; the busy cases correspond to overlap between communication and computation.

The results in Table 6 show that applications are often successful at overlapping communication and computation, so communication primitives should support this in an efficient way. Note that sockets, which require (at least a logical) copy as part of the send and receive do not meet this criterion since they place a lot of the communication overhead in the critical path of the application. Communication primitives that allow a looser coupling between the application and the communication operation are more attractive. Examples are asynchronous communication and buffered communication primitives [56, 60]. Asynchronous primitives allow an application to post a send or receive (for example in the form of a pointer-length pair describing a message) and then to continue processing while the communication operation is executed. With buffered primitives the application and the system share message buffers, thus avoiding data copying (Section 3.2). These operations make it possible to exploit parallelism between the application and the communication operation more effectively. This type of operations is already widely accepted on tightly coupled systems (for example, the deposit model used on iWarp [32], and active messages [63]), but their use on network-based multicomputers is less common.

## 6. Concluding remarks

Our experience using the Nectar systems shows that network-based multicomputers are an attractive architecture for many applications. They allow applications to combine the computing power and memory (e.g. COSMOS) of multiple nodes on the network, thus speeding program execution. Moreover, since network-based multicomputers use existing computers as nodes, parallelized applications can make immediate use of system software and application code that already exists for these computers, thus simplifying the parallelization and porting effort. Special-purpose devices, e.g. the framebuffer used by the LA pollution application, can also be used easily. Finally, new architectures (e.g. more powerful workstations) can be included rapidly in the system, so network-based multicomputers can capitalize very rapidly on new technologies.

An important requirement for network-based multicomputers is good **application-level** network performance. We described the Nectar system and showed how it addresses the communication requirements of network-based multicomputers. The most serious performance problem is the loose coupling between the network interface and the memory system on most computer systems. This increases communication overhead compared with dedicated systems. There are also a number of other tasks that have to be performed on networks that are not needed on dedicated systems, but their impact on throughput is small.

However, much work needs to be done in support of the new opportunities created by Nectar-like systems. First, programming tools have to be developed that will help programmers in distributing their application. Second, new network technologies such as ATM should make it possible to not only improve network performance, but also to make the network more predictable, which should simplify application development. Finally, multicomputers make it possible to use heterogeneous architectures, which can potentially result in substantial speedups by matching application requirements with machine characteristics [40, 19]. We are exploring these issues in the context of the Gigabit Nectar [57, 36] and Credit Net [43, 37] projects at Carnegie Mellon University.

## Acknowledgements

A large number of people at Carnegie Mellon University contributed to the Nectar system. H.T. Kung, Eric Cooper, Robert Sansom, Brian Zill and Francois Bitz played a key role in designing and building the system. Fritz Prinz, Marco Gubitoso, Thomas Fahringer, Manpreet Khaira, Randy Bryant, Carl Seger, Greg McRae, Hiroshi Nishikawa and Bernd Bruegge contributed to the porting of the applications to Nectar and to tuning their performance. Thomas Gross and the anonymous referees provided valuable feedback on drafts of this paper.

## References

**1.** D. Adams. *Cray T3D system architecture overview.* Cray Research Inc., 1993. Revision 1.C.

**2.** Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica Lam, Onat Menzilcioglu, and Jon Webb. "The Warp Computer: Architecture, Implementation, and Performance". *IEEE Transactions on Computers 36*, 12 (December 1987), 1523-1538.

**3.** Jose Arabe, Adam Beguelin, Bruce Lowekamp, Eric Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Tech. Rept. CMU-CS-95-137, Computer Science Department, Carnegie Mellon University, April, 1995.

**4.** Emmanuel Arnould, Francois Bitz, Eric Cooper, H. T. Kung, Robert Sansom and Peter Steenkiste. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ACM/IEEE, Boston, April, 1989, pp. 205-216.

**5.** D. H. Bailey, E. Barszcz, R. A. Fatoohi, H. D. Simon, and S. Weeratunga. Performance Results on the Intel Touchstone Gamma Prototype. Proceedings of the Fifth Distributed Memory Computing Conference, IEEE, April, 1990, pp. 1236-1245.

**6.** H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. "Orca: A Language For Parallel Programming of Distributed Systems". *TRANSSE 18*, 3 (March 1992), 190-205.

**7.** Tom Blank. The MasPar MP-1 Architecture. IEEE Compcon Spring 1990, IEEE, San Francisco, February/March, 1990, pp. 20-24.

**8.** S. Bokhari. Communication Overhead on the Intel iPSC-860 Hypercube. ICASE Interim Report 10, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, May, 1990.

**9.** Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. Proceedings of the 1988 International Conference on Supercomputing, IEEE Computer Society and ACM SIGARCH, Orlando, Florida, November, 1988, pp. 330-339.

**10.** Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H.T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. Supporting Systolic and Memory Communication in iWarp. Proceedings of the 17th Annual International Symposium on Computer Architecture, ACM/IEEE, Seattle, May, 1990, pp. 70-81.

**11.** Bernd Bruegge, Hiroshi Nishikawa, and Peter Steenkiste. Computing over Networks: An Illustrated Example. Proceedings of the Sixth Distributed Memory Computing Conference, IEEE, April, 1991, pp. 254-257.

**12.** Bernd Bruegge. A Portable Platform for Distributed Event Environments. PDDEB91, ACM, Santa Cruz, California, December, 1991, pp. 184-193. .

**13.** Randy E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. Proceedings of the Design Automation Conference, ACM/IEEE, June, 1987, pp. 9-16.

**14.** Vinton G. Cerf and Robert E. Kahn. "A Protocol for Packet Network Communication". *IEEE Transactions on Communications 22*, 5 (May 1974), 637-648.

**15.** M. Chen. A Parallel Language and Its Compilation to Multiprocessors Machines or VLSI. Conference Record of the 13th Annual ACM Conference on Principles of Programming Languages, ACM, January, 1986, pp. 131-139.

**16.** Roger S. Chin and Samuel T. Chanson. "Distributed Object-Based Programming Systems". *ACM Computing Surveys 23*, 1 (March 1991), 91-124.

**17.** Young Choi. *Vertex-based Boundary Representation of Non-Manifold Geometric Models*. Ph.D. Th., Carnegie Mellon University, 1989.

**18.** David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. "An Analysis of TCP Processing Overhead". *IEEE Communications Magazine 27*, 6 (June 1989), 23-29.

**19.** Robert Clay and Peter Steenkiste. Distributing a Chemical Process Optimization Application over a Gigabit Network. Proceedings of Supercomputing ’95, ACM/IEEE, December, 1995. Appeared on CD-ROM only.

**20.** Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol Implementation on the Nectar Communication Processor. Proceedings of the SIGCOMM ’90 Symposium on Communications Architectures and Protocols, ACM, Philadelphia, September, 1990, pp. 135-143.

**21.** M. de Prycker. *Asynchronous Transfer Mode.* Ellis Harwood, 1991.

**22.** *TURBOchannel Overview.* Digital Equipment Corporation, 1990.

**23.** Thomas Gross, Dave O’Hallaron and Jaspal Subhlok. "Task Parallelism in a High Performance Fortran Framework". *IEEE Parallel & Distributed Technology 2*, 3 (Fall 1994), 16-26.

**24.** R. D. Gaglianello, B. S. Robinson, T. L. Lindstrom, E. E. Sampieri. HPC/VORX: A Local Area Multicomputer System. Proceedings of the Ninth International Conference on Distributed Computing Systems, IEEE, June, 1989, pp. 246-253.

**25.** G.A. Geist and V. S. Sunderam. The PVM System: Supercomputer Level Concurrent Computation on a Heterogeneous Network of Workstations. Proceedings of the Sixth Distributed Memory Computing Conference, April, 1991, pp. 258-261.

**26.** Andrew Grimshaw. The Mentat Run-Time System: Support for Medium Grain Parallel Computation. Proceedings of the Fifth Distributed Memory Computing Conference, IEEE, April, 1990, pp. 1064-1073.

**27.** John L. Gustafson, Gary R. Montry, and Robert E. Benner. "Development of Parallel Methods for a 1024-processor Hypercube". *SIAM Journal on Scientific and Statistical Computing 9*, 4 (July 1988), 609-638.

**28.** Leonard Hamey, John Webb, and I-Chen Wu. Low-Level Vision on Warp and the Apply Programming Model. In *Parallel Computation and Computers for Artificial Intelligence*, J. Kowalik, Ed., Kluwer Academic Publishers, 1987, pp. 185-199.

**29.** M. Homewood, D. May, D. Shepherd, and R. Shepherd. "The IMS T800 Transputer". *IEEE Micro 7*, 5 (October 1987), 10-26.

**30.** *iPSC/2 C Programmer's Reference Manual.* Intel, 1988.

**31.** *Paragon X/PS Product Overview.* Intel Corporation, 1991.

**32.** Tom Stricker, Jim Stichnoth, Dave O'Hallaron, Susan Hinrichs and Thomas Gross. Decoupling Communication Services for Compiled Parallel Programs. Tech. Rept. CMU-CS-94-139, Carnegie Mellon University, School of Computer Science, 1994.

**33.** R. Iyer and G. McRae. Parallel Strategies for Flowsheet Simulation Using Heterogeneous Distributed-Memory Computers. Submitted for publication.

**34.** P. Janson and R. Molva. "Security in Open Networks and Distributed Systems". *Computer Networks and ISDN Systems 22*, 5 (October 1991), 323-346.

**35.** Manpreet Khaira. Enabling Large Scale Switch-Level Simulation. Internal report.

**36.** Karl Kleinpaste, Peter Steenkiste, and Brian Zill. Software Support for Outboard Buffering and Checksumming. Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols, ACM, August/September, 1995, pp. 87-98.

**37.** Corey Kosak, David Eckhardt, Todd Mummert, Peter Steenkiste, and Allan Fisher. Buffer Management and Flow Control in the Credit Net ATM Host Interface. LAN20, IEEE, Minneapolis, October, 1995, pp. 370-378.

**38.** Gautham K. Kudva and Joseph Pekny. "A Distributed Exact Algorithm for the Multiple Resource Constrained Sequencing Problem". *Annals of Operations Research* , ( 1992), .

**39.** G. Kudva and J. F. Pekny. "DCABB: A Distributed Control Architecture for Branch and Bound Calculations". *Computers and Chemical Engineering* , ( 1994), . In Press.

**40.** H.T. Kung. Heterogeneous Multicomputers. Carnegie Mellon Computer Science: A 25-Year Commemorative, Reading, Massachusetts, 1990, pp. 235-251.

**41.** H.T. Kung, Robert Sansom, Steven Schlick, Peter Steenkiste, Matthieu Arnould, Francois J. Bitz, Fred Christianson, Eric C. Cooper, Onat Menzilcioglu, Denise Ombres, and Brian Zill. Network-Based Multicomputers: An Emerging Parallel Architecture. Proceedings of Supercomputing '91, IEEE, Albequerque, November, 1991, pp. 664-673.

**42.** H.T. Kung, Peter Steenkiste, Marco Gubitoso, and Manpreet Khaira. Parallelizing a New Class of Large Applications over High-Speed Networks. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, April, 1991, pp. 167-177.

**43.** H. T. Kung. "Gigabit Local Area Networks: A Systems Perspective". *IEEE Communications Magazine 30*, 4 (April 1992), .

**44.** Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

**45.** Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta and John Hennessy. The DASH Prototype: Implementation and Performance. 19th Annual International Symposium in Computer Architecture, IEEE, May, 1992, pp. 92-102.

**46.** The MPI Forum. MPI: A Message Passing Interface. Proceedings of Supercomputing '93, Oregon, November, 1993, pp. 878-883.

**47.** Bernd Bruegge and Peter Steenkiste. Supporting the Development of Network Programs. Proceedings of the Eleventh International Conference on Distributed Computing Systems, IEEE, May, 1991, pp. 641-648.

**48.** Hiroshi Nishikawa and Peter Steenkiste. Aroma: Language Support for Distributed Objects. International Parallel Processing Symposium, IEEE, Los Angeles, April, 1992, pp. 686-690.

**49.** Hiroshi Nishikawa and Peter Steenkiste. A General Architecture for Load Balancing in a Distributed-Memory Environment. Proceedings of the Thirteenth International Conference on Distributed Computing Systems, IEEE, Pittsburgh, May, 1993, pp. 47-54.

**50.** Andreas Nowatzyk, Gunes Aybay, Michael Browne, Edmund Kelly, David Lee, and Michael Parkin. The S3.mp Scalable Shared Memory Multiprocessor. Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences - Vol. I: Architecture, January, 1994, pp. 144-153.

**51.** *Sbus Specification A.1.* Sun Microsystems, Inc., 1990.

**52.** Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A High-speed, Self-configuring Local Area Network Using Point-to-point Links. Research Report 59, DEC Systems Research Center, April, 1990.

**53.** Bruce Siegell and Peter Steenkiste. Automatic Generation of Parallel Programs with Dynamic Load Balancing. Proceedings of the Third International Symposium on High-Performance Distributed Computing, IEEE, San Fransisco, August, 1994, pp. 166-175.

**54.** Bruce Siegell and Peter Steenkiste. Controlling Application Grain Size on a Network of Workstations. Proceedings of Supercomputing '95, ACM/IEEE, December, 1995. Appeared on CD-ROM only.

**55.** Bruce Siegell. *Automatic Generation of Parallel Programs with Dynamic Load Balancing for a Network of Workstations*. Ph.D. Th., Department of Computer and Electrical Enginerring, Carnegie Mellon University, 1995. Also appeared as technical report CMU-CS-95-168.

**56.** Peter Steenkiste. A Symmetrical Communication Interface for Distributed-Memory Computers. Proceedings of the Sixth Distributed Memory Computing Conference, IEEE, Portland, April, 1991, pp. 262-265.

**57.** Peter A. Steenkiste, Brian D. Zill, H.T. Kung, Steven J. Schlick, Jim Hughes, Bob Kowalski, and John Mullaney. A Host Interface Architecture for High-Speed Networks. Proceedings of the 4th IFIP Conference on High Performance Networks, IFIP, Liege, Belgium, December, 1992, pp. A3 1-16.

**58.** Peter Steenkiste. Analyzing Communication Latency using the Nectar Communication Processor. Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols, ACM, Baltimore, August, 1992, pp. 199-209.

**59.** Peter A. Steenkiste, Michael Hemy, Todd Mummert, Brian Zill. Architecture and Evaluation of a High-Speed Networking Subsystem for Distributed-Memory Systems. Proceedings of the 21th Annual International Symposium on Computer Architecture, IEEE, , May, 1994, pp. 154-163.

**60.** Peter A. Steenkiste. "A Systematic Approach to Host Interface Design for High-Speed Networks". *IEEE Computer 26*, 3 (March 1994), 47-57.

**61.** C. Tseng and S. Hiranandani and K. Kennedy. Preliminary Experiences with the Fortran D Compiler. Proceedings of Supercomputing '93, Portland, OR, Nov., 1993, pp. 338--350.

**62.** Lewis W. Tucker and George G. Robertson. "Architecture and Applications of the Connection Machine". *IEEE Computer 21*, 8 (August 1988), 26-38.

**63.** Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism or Integrated Communication and Computation. Proceedings of the 19th Annual International Symposium on Computer Architecture, May, 1992, pp. 256-266.

**64.** Jon A. Webb. Architecture-Independent Global Image Processing. 10th International Conference on Pattern Recognition, IEEE, Atlantic City, NJ, June, 1990, pp. 623-628.

**65.** Jon Webb. "Steps Towards Architecture-Independent Image Processing". *IEEE Computer 25*, 2 (February 1992), 21-31.

**66.** I-Chen Wu and H.T. Kung. Communication Complexity for Parallel Divide-and-Conquer. 1991 Symposium on Foundations of Computer Science, San Juan, October, 1991, pp. 151-162.

**67.** I-Chen Wu. *Multilist scheduling: A New Parallel Programming Model*. Ph.D. Th., School of Computer Science, Carnegie Mellon University, 1993. Also appeared as technical report CMU-CS-93-184.