Research in Parallel Computing Final Report: 1987-1990

Thomas Gross, Peter Steenkiste, Richard Rashid, Zary Segall,
C.H. Chien

Prepared by the Research Documents Group
Jutta Denton and C. Roy Taylor, editors

August 5, 1994
CMU-CS-94-180

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

Abstract

This report chronicles an integrated research program conducted at Carnegie Mellon's Computer Science Department during the contract period 1987 to 1990. This research produced the iWarp System, a homogeneous multiprocessor that provides both high-speed computation and communication; a high-performance, fiberoptic interface board and a crossbar switch for high-speed, heterogeneous computing; a parallel operating system as well as software for analyzing and debugging its performance; and an environment for building parallel, distributed, vision-processing programs.

This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through ARPA Orders 4864, 9216, and 9187, and monitored by the Space and Naval Warfare Systems Command under contract N00039-87-C-0251. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the DARPA or the United States Government.

Preface

This report chronicles an integrated research program during the contract period 1987 to 1990 in the Computer Science Department, Carnegie Mellon University. The major goals of this research were to:

- Develop heterogeneous architectures, tools, and prototype applications that combine diverse cooperating subsystems.
- Couple work on architectures with development of software tools and operating system facilities, thereby providing a full range of resources necessary to facilitate the actual use of parallel processing.
- Respond to the needs of computationally demanding AI problems.

Based on these broad goals our research produced the following results in four separate projects:

Homogeneous multiprocessors

- The Integrated Warp System (iWarp), developed jointly with a commercial contractor
- Programming support tools, such as a user interface, an optimizing compiler, and a debugger
- Program transformation and synthesis tools that automatically generate parallel programs for Warp-like systolic array computers

A high-speed, heterogeneous network architecture

- A high-performance, fiberoptic interface board and a crossbar switch board for heterogeneous computing
- Communication protocols, node operating system support, and an application interface to facilitate high-speed computing over a heterogeneous network

Parallel operating systems

- A "pure" Mach kernel on which UNIX functionality can be implemented as one or more user-level processes
- Refined kernel support for tightly and loosely coupled multiprocessors
- Emulations of two non-UNIX environments for execution within the Mach framework
- Software for monitoring, analyzing, and debugging the kernel's performance

A new-generation AI-architecture

- An environment for building parallel, distributed vision-processing programs.

Table of contents

1 Homogeneous multiprocessors

1.1 Introduction

Over the last nine years Carnegie Mellon has been engaged in a research program to build complete systems to meet the demanding needs of application areas like image processing, signal processing, and scientific computing. This research addressed a broad range of issues from hardware design and implementation to software tools that include programming environments, compilers, and debuggers.

The primary goal of the research effort reported here was to develop the "integrated Warp" (iWarp) machine. The iWarp system evolved from our experience with the "Warp machine," a systolic array multiprocessor we had designed and prototyped under the proceeding ARPA contract.

This report is organized in the following manner: The next section provides a brief description of the Warp machine and a detailed discussion of the iWarp System. Following sections describe program development tools, program generators, and application development.

1.2 Warp

The Warp machine is a linear array of 10 cells and has a peak performance of 100 MFLOPS (million floating point operations per second). The initial design and prototyping of this machine has been funded by an earlier contract, and two prototypes became operational in 1985. The original design is described in [Annaratone et al. 86]. Each cell is implemented on a large (19 inch square) wire-wrap board, using standard off-the-shelf components. The Warp machine is connected to a custom multiprocessor (three 68000-based, single-board, off-the-shelf computers), which provides a connection to a general purpose host (a SUN workstation running UNIX) as well as a large staging memory [Annaratone et al. 87a].

The Warp system provides a good platform for a variety of applications. To make this system available to other researchers, we redesigned (in collaboration with General Electric, our industrial partner) the Warp machine so that the system can be reproduced efficiently. The design changes are described in [Annaratone et al. 87d]. Each cell has been implemented on a printed circuit board, and the use of VLSI technology allowed larger local memories and other enhancements (like an improved controller that provides better support for compiler generated code, see below). The overall machine and its evaluation is described in [Annaratone et al. 87b], which appeared in a special issue on supercomputers. An in-depth evaluation of the Warp machine was published [Annaratone et al. 87b], which summarizes the contents of earlier papers, [Annaratone et al. 86] [Annaratone et al. 87a] [Annaratone et al. 87d], and also includes an evaluation for a large number of kernels and complete applications.

Although the Warp architecture is very attractive, it has some disadvantages that limit its use in important application scenarios. The most important limitations are its physical size (each cell takes up a 19-inch square board) and its fixed arrangement in a linear array. Both limitations make it difficult to scale the system to hundreds or thousands of processors. Overcoming these limitations was part of the motivation for developing, in collaboration with Intel Corporation, the integrated Warp system (iWarp) [Borkar et al. 88].

1.3 The iWarp system

The iWarp processor integrates both a high-speed computation and communication capability in a single component. The processor is a powerful computation engine that employs instruction-level parallelism to allow simultaneous operation of multiple functional units. The feature that makes iWarp unique, however, is its interprocessor communication capability. An iWarp processor can simultaneously communicate with a number of other iWarp processors at high speeds. More importantly, the iWarp processor has a highly flexible communication mechanism that can support different programming models, including the tightly coupled computing found in systolic arrays and the message passing style of computation found in many existing distributed memory machines. These communication capabilities allow the effective use of iWarp for a wide range of applications.

The first silicon of the iWarp component was fabricated in December 1989. It consists of approximately 650,000 transistors and measures about 1.4 cm (551 mil) on a side. The target frequency of the iWarp component is 20 MHz with data being transferred between processors at twice that frequency. The first system operated at 10 MHz. During 1990 and 1991, Intel revised the implementation to meet the design goal of 20 MHz, and Carnegie Mellon has now three systems operating at this speed.

1.3.1 The iWarp component

The iWarp component consists of three autonomous subsystems: the computation agent, the communication agent, and the memory agent.

The *computation agent*, which executes programs, can deliver 20 (or 10) MFLOPS for single (or double) precision calculations plus 20 MIPS for integer/logic operations. The computation agent includes three units that can be scheduled to operate in parallel in one instruction:

- Floating-point adder: Nonpipelined, 10 and 5 MFLOPS for 32- and 64-bit additions (IEEE 754 standard), respectively
- Floating-point multiplier: Nonpipelined, full divide, remainder, and square root support; 10 and 5 MFLOPS for 32- and 64-bit multiplications (IEEE 754 standard).
- Integer/logical unit: Arithmetic, logical and bit operations with 20 MIPS peak performance on 8/16/32-bit integer/ordinal data.

The design of the computation agent was derived from detailed studies with an optimizing compiler (we retargeted the optimizing Warp compiler to iWarp) [Cohn et al. 89]. The extensive compiler modelling was a key and necessary activity for achieving the attractive iWarp architecture.

- The *communication agent*, which implements the iWarp's communication system, can sustain an aggregate intercell communication bandwidth of 320 MBytes/s by using four input and four output busses. The internal data storage and interconnect system is comprised of a shared, multiported, 128-word register file plus special register file locations for local memory and communication agent access.
- The *memory agent*, which provides a high-bandwidth interface to the local memory, can transfer streams of data into or out of the communication agent at a rate of 160 MBytes/sec. The memory system provides:
  - Off-chip local memory for data and instructions with
    - separate address and data busses (24-bit word address bus, 64-bit databus)
    - 20 million memory accesses/s peak performance
    - 160 MBytes/s peak memory bandwidth
    - Read, write, and read/modify/write support
  - On-chip program store
    - 256-word cache RAM
    - 2K-word ROM (built-in functions)
    - 32- and 96-bit instructions

1.3.2 Communication

We have identified two communication models used for distributed memory parallel systems: *message passing* and *systolic*. iWarp supports both models [Borkar et al. 90]. The two models differ primarily in the granularity of communication and computation. In message-passing mode, as in computer networks, the unit of processing is a complete message. That is, a message is accumulated in the source cell memory and transmitted as a unit to the destination cell. At the destination cell, the message is not available to be operated upon until the full message is in the local memory. In contrast, the systolic mode allows a unit of communication and processing to be as fine- grained as a single word.

In both message passing and systolic communication models, there is a need to multiplex multiple communication paths onto a physical bus. Efficient support of communication paths requires dedicated hardware resources which in turn imposes a practical limitation of providing a small number of paths. This resource limitation raises the issue of resource management [Gross 89]. For some classes of programs, the communication resources can be managed at compile time [Kung 88a], while others require the runtime system to make the resource allocation decisions. Thus, the runtime system must (and does) include a message-passing system that implements a simple, safe (reliable) message protocol.

The key iWarp communication abstraction is the *pathway*: a pathway is a direct connection from a cell (called the source cell) to another cell (called the destination cell). Each segment of the pathway connects the communication agent of a cell to its own computation agent or that of another cell. Using the pathway abstraction, we can establish a "direct" connection between two cells in an array even if the two cells are not physically connected. That is, the program sees a direct connection, but the cells are not neighbors in the iWarp torus (they do not share any wires). This feature allows us to configure an iWarp torus in a variety of ways (e.g., as a tree or binary hypercube) without changing the physical connections (wires). This capability is used effectively by a number of application programs.

1.4 Program development tools

During the design (Phase I) and implementation (Phase II) of the iWarp system, we used two 10-cell Warp systems as a platform for program development and tuning of the iWarp (micro and macro) architecture. To support the use of the Warp machine for a large number of users in our local environment, we developed the Warp Programming Environment (WPE) [Bruegge 88a]. WPE has four main components [Bruegge et al. 87a] [Bruegge et al. 87]:

- the user interface (by which means the user specifies programs to run on the Warp machine and their operands)
- an optimizing high-level language compiler for the Warp cells (to generate code)
- a symbolic debugger for the Warp cells (to assist in program development)
- a loader/program executive (to control execution)

1.4.1 User interface

WPE allows transparent access to the Warp machine from any workstation connected to our local area network. The user runs programs in the "Warp shell" — a CommonLisp shell with Warp-specific extensions such as "run program on Warp machine<name>". The integration of the Warp machine into the shell of each user is a novel idea and simplified program development significantly [Bruegge 88a].

The Warp Programming Environment was adopted by General Electric, our industrial partner, has been distributed to all other Warp sites [Bruegge 88], and has become the de-facto system to use the Warp machines.

1.4.2 Optimizing compiler

The optimizing compiler translates a simple, Pascal-like language (named W2) into code for a single Warp cell [Gross and Lam 86]. The optimizing single-node compiler allows each node to be programmed in a Pascal-like language, which includes conditional branches, do-loops, while-loops, and subroutine call/return[Gross et al. 87]. Communication between adjacent nodes is specified explicitly by *send* and *receive* statements. The communication facilities directly mirror the hardware implementation of the system. The physical topology of the (linear) array determines the possible communication interactions between nodes. A node can receive data via two distinct channels from the left or via one channel from the right neighbor node. Since one link from the left node shares the input queue with the link from the right node (as depicted in Figure 1) at any point in time, a node can receive either two items from the left or one item

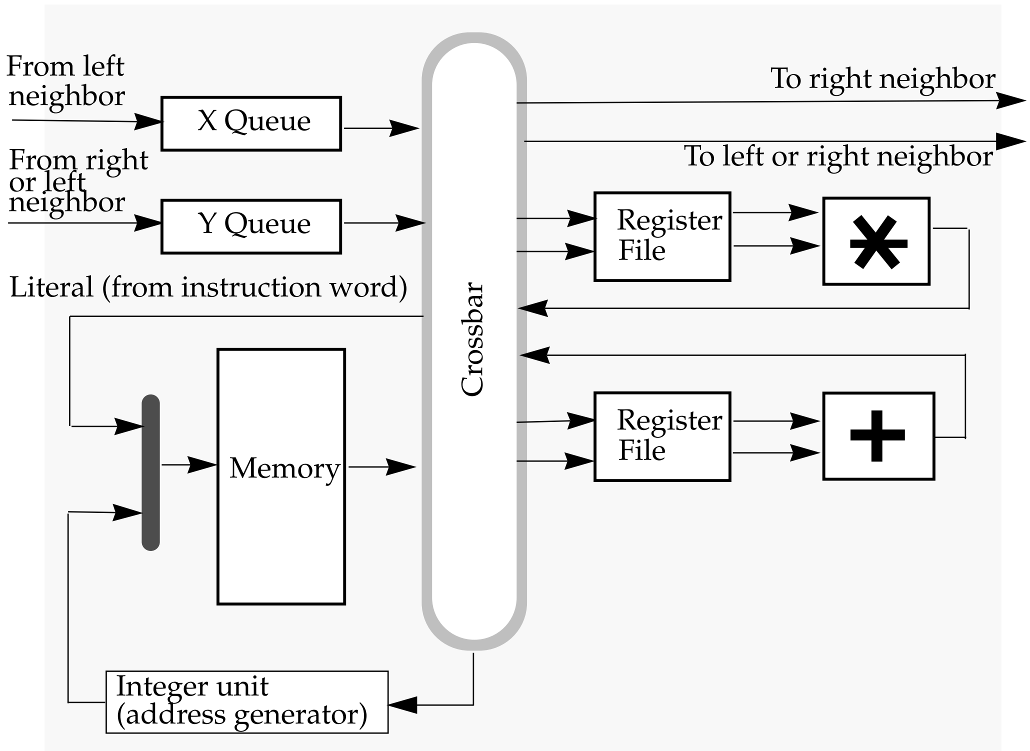

Figure 1

from the left and one from the right. Although the hardware requires that this restriction be satisfied on a per-micro-instruction basis, dealing with this constraint at that level proved to be difficult in the compiler. Therefore, the compiler imposed the more restrictive constraint that each source language function be allowed to use the shared link either as a link in the left-to-right or as a link in the right-to-left direction. That is, direction changes are only supported at the granularity of source-level procedures. The “send'' statements have corresponding functionality; each node can send data via two channels to the right and via another channel to the left. There are no provisions for communication between nodes that are not adjacent, and the number of channels is fixed by the number of physical links in the system.

Warp is programmed in Pascal-like language, but the architecture of the nodes imposed a number of constraints on what language features could be supported: There is no “case'' or “switch” statement, since there is no path from the program counter to the data memory. For procedure calls, the turn address is stored in a stack local to the controller. There is no heap, the memory is word-addressable, support for byte operands is limited, and there is no support for double-precision floating point arithmetic. These restrictions make it impossible to efficiently implement widely accepted languages such a **C**, Pascal, or Fortran. Although this limits the overall usefulness of the system, there are fortunately many applications, mostly from the domain of image processing, for which these restrictions impose no problems [Annaratone et al. 87b].

The optimizations implemented in the W2 compiler include not only those typical of general-purpose processors, but also those tailored to pipelined architectures, specifically local optimization and global flow analysis.

- The local optimizations implemented include common subexpression elimination, constant folding, height reduction, dead code removal, and idempotent operation removal. Height reduction of the dag is especially important for heavily pipelined machines. A compiler for a conventional processor favors long tree-like dags so that they can be evaluated with a small number of registers. However, on Warp, a balanced tree is much better since it allows the parallel evaluation of multiple branches of the tree.
- The global flow analyzer collects detailed intra-block information for all variables of the program [Gross and Steenkiste 90]. For regular accessing patterns, the analysis is powerful enough to distinguish between individual array elements and different iterations of a loop. This global data dependency information is incorporated into the dag as arcs between nodes in different basic blocks. There are two types of arcs. If the global analyzer can deduce a strict dependency, we say that node *n* uses the values of node *m* (for example, iteration *i* of a loop always uses the result of iteration *i - 1* of the loop). If a strict dependency cannot be established, the global flow analyzer inserts sequencing arcs that enforce a conservative order of evaluation. This information makes it possible for the code scheduler to overlap the execution of different basic blocks which is important for heavily pipelined processors like the Warp cells.

To exploit the high degree of pipelining and parallelism in the machine, extensive global optimization is performed during code generation. We use two scheduling algorithms: a scheduling technique specialized for innermost loops, called *software pipelining*, and a new unified scheduling technique that applies both within and across basic blocks [Lam 87].

Software pipelining was proposed originally for scheduling hardware pipelines. In the W2 compiler, we have extended the scope of applicability of this technique and have shown that this code generation technique is very successful for a large class of VLIW (Very Long Instruction Word) processors [Lam 88].

1.4.3 Debugger

The optimizing W2 compiler freed the programmers of the Warp machine from writing assembly language code (which in this case was writing microcode, since the assembly language operations corresponded closely to the micro-instructions of the machine). However, as a consequence, users required more assistance when debugging their programs. Furthermore, a parallel machine like Warp (with 10 cells) presents another level of complexity to the user. Our research addressed this problem, and we implemented a debugger for the Warp machine [Bruegge and Gross 88a]. One challenge for every debugger (or programming environment with a debugger) is to ease the transition from debugging to real program execution. Nothing is more annoying than having a program that works with the debugger and does not work without the debugger. Previous research groups have often been able to ignore this problem since the primary objective was to get a working program. However, a machine like Warp is used for real-time applications, and a program that "works" with the debugger may be too slow to be considered *working*. This issue was elevated in importance as a result of our collaboration with the computer vision group at Carnegie Mellon, and we devised a software architecture that provides a smooth transition from debugged program to embedded application [Bruegge and Gross 88].

Although the primary programming language of the Warp machine is imperative, we investigated other styles as well and concluded that Warp and Warp-like machines provide a good basis for applicative programming languages like SISAL [Gross and Sussman 87]. We implemented a SISAL compiler (which maps SISAL into W2) as part of a thesis project [Sussman 91].

1.5 Program generators

With the advent of a high-level optimizing compiler, users program in a high-level language instead of writing micro code or machine instructions. But programming a multiprocessor like Warp is difficult even with a single-cell compiler. To simplify this task several *parallel program generators* based on top of the single-node compiler were introduced. A parallel program generator maps a computation specified for a single thread of control and for a single address space onto the parallel system.The parallel program generator produces a high-level language program for each node, and this program is subsequently compiled with the single-node compiler. The key point is that the program is written without explicit communication between nodes. The parallel program generator manages all the nodes of the parallel system, i.e., the distribution of data and computations handled by the parallel program generator. Typically, the parallel program generator imposes restrictions on the type of programs or computations that can be mapped onto the parallel system. For example, the user may be asked to provide hints that help in identifying the parallelism, like if it is better to partition a matrix by rows or by columns. Or only programs from a specific application domain, e.g., low-level image processing with local neighborhood operators, may be accepted. (An example of a neighborhood operator is a two-dimensional convolution: Each $y_{i,j}$ result depends only on the kernel and fixed-size window of the input image.) Such restrictions or hints ease the task of the program generator. To determine which operations can be done in parallel, the program generator can use simple dependence analysis (or none at all in the case of neighborhood operators). There can be a single mapping strategy to map computations and data onto the parallel system, or the mapping is provided as a hint by the user. Therefore, limited analysis of the program is sufficient to determine when it is necessary to move data from one node to another.

A parallel program generator is an example of a parallelizing compiler, but we prefer the term "program generator'' to emphasize that these program generators cannot deal with arbitrary (uniprocessor) programs but require some form of hints or directives. Nevertheless, compared to writing individual programs for each node, they simplify programming of parallel system dramatically. The parallel program generators developed for the Warp project are directed towards specific application domains: computer vision and image processing (Apply) [Wallace et al. 89] [Hamey et al. 87], scientific computing (AL) [Tseng 89][Tseng 90], and perfectly nested loops as found in many linear- algebra computations [Ribas 90]. By concentrating on specific application domains, the input language for a parallel program generator can include features that are specific to the application domain, such as the notion of image boundaries in Apply. Or the parallel program generator can produce high-quality code for a limited application domain, as it is done by the compiler for nested loops. Common to all parallel program generators is the question of *how* to map (partition) a program. Our research has demonstrated the importance of developing adequate models for the behavior of the partitioned programs. Such models allow the parallel program generator to select the most appropriate mapping.

Parallel program generators do not render the single-node compilers unnecessary, in the contrary, a good single-node compiler is a necessary condition for the development of a parallel program generator. A translator that maps an application directly into the (micro) instructions of a parallel system duplicates a lot of the backend of a single-node compiler. And including a system-specific backend in a parallel program generator complicates retargeting the program generator to a different parallel platform. However, it is important that the single-node compiler produces high-quality code. Otherwise, it is difficult or impossible for the parallel program generator to develop a realistic cost model (to guide mapping decisions). So it is not surprising that the parallel program generators were developed after the single-node compiler was implemented.They become operational long after the last system had been delivered by GE, and their development coincided with the design of the next generation of systolic systems (the integrated Warp system), as discussed earlier.

1.6 Application development

The prototype Warp machines at Carnegie Mellon have been used in a diverse range of applications, including robot vehicle control, signal processing, and medical image processing, and as a tool for vision research. A small number of algorithm partitioning methods have allowed efficient use of the Warp machine in all of these areas. Large applications that use Warp as part of a system are efficiently supported by Warp's flexible host [Annaratone et al. 87c] [Annaratone et al. 87]. By implementing a broad range of applications on the Warp machine, we demonstrated the feasibility of programmable, high-performance systolic array computers. The programmability of Warp has substantially extended the machine's application domain. The cost of programmability is limited to an increase in the physical size of the machine; it does not incur a loss in performance, given appropriate architectural support. This is shown by Warp, as it can be programmed to execute many well-known systolic algorithms as fast as special-purpose arrays using similar technology.

The group of applications for the Warp machine include:

- General image processing: A large library of image processing subroutines has been implemented. The Warp machine is being used for vision research, as well as a tool in more applied domains. Medical image processing, particularly the processing of nuclear magnetic resonance (NMR) data, is also being developed as an applications area.
- Image processing for robot navigation: Algorithms and systems implemented include road following, obstacle avoidance using stereo vision, and obstacle avoidance using the Environmental Research Institute of Michigan (ERIM) laser range scanner [Wallace et al. 89].
- Signal processing: Several different algorithms have been developed in this area, including singular value decomposition (SVD) for adaptive beamforming [Annaratone et al. 86a].
- Scientific computing: Algorithms include successive over-relaxation (SOR) for solution of systems of partial differential equations. We investigated methods for dense matrices [Tseng 89] as well as methods for sparse matrices (direct methods [Tseng 88] as well as iterative methods [Tseng 88a]).
- Neural networks: Several versions of backprop (a neural net learning algorithm) have been implemented, and one version of backprop has been used in the control of Carnegie Mellon's autonomous land vehicle.

In addition, Carnegie Mellon has been providing help to several DARPA contractors in their applications of Warp, including Martin Marietta Corporation in Autonomous Land Vehicles [Dunlay 88] and Hughes Aircraft Corporation image analysis.The area of signal processing (in the most general sense) provides a natural application domain for Warp and Warp-like machines. Several groups (both at Carnegie Mellon and elsewhere) have investigated this area, and a number of publications by current and former members of our research group illustrate the importance of this class of applications: [Baheti et al. 89], [OHallaron 91a], [OHallaron and Fiduccia 91], [OHallaron 91], [Baheti, OHallaron, and Itzkowitz 90]. One issue that is often overlooked when mapping a signal processing application onto a parallel computer is that a real application consists of many parts. Some parts may be easy to map, whereas others cause difficulties. Our collaboration with real application users [Kung et al. 88] provided interesting insights.We mapped a large application onto the Warp machine, and we noticed that a straight forward approach (which maps each top-level “task” of a signal-processing application onto a separate processor or cell) produces unsatisfactory results. In practice, only a small number of cells can be used that way [Printz 91], [Printz 92]. To use a larger number of cells, it is necessary to map each individual task onto *multiple* cells, and this observation is a key factor in the design of our next parallel program generator (which is being developed as part of a follow-on contract).

1.7 Bibliography

[Annaratone et al. 87]

Annaratone, M., F. Bitz, J. Deutch, H.T. Kung, L. Hamey, P. Maulik, P. Tseng, and J. Webb. Applications Experience on Warp. *Proceedings of the 1987 National Computer Conference*, pages 149-158. AFIPS, June, 1987. Also appeared in Proceedings of COMPCON Spring '87.

[Annaratone et al. 86]

Annaratone, M., E. Arnould, T. Gross, H.T. Kung, M.S. Lam, O. Menzilcioglu, K. Sarocky, and J.A. Webb. Warp Architecture and Implementation. *Conference Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 346-356. IEEE/ACM, June, 1986.

[Annaratone et al. 86a]

Annaratone, M., E. Arnould, H.T. Kung, and O. Menzilcioglu. Using Warp as a Supercomputer in Signal Processing. *Proceedings of ICASSP 86*, pages 2895-2898. IEEE, April, 1986.

[Annaratone et al. 87a]

Annaratone, M., E. Arnould, R. Cohn, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, K. Sarocky, J. Senko, and J. Webb. Architecture of Warp. *Proceedings of COMPCON Spring '87*, pages 264-267. IEEE Computer Society, February, 1987.

[Annaratone et al. 87b]

Annaratone, M., E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb. The Warp Machine: Architecture, Implementation, and Performance. *IEEE Transactions on Computers* . C-36(12):1523-1538, December, 1987. Also appeared as technical report CMU-CS-87-166.

[Annaratone et al. 87c]

Annaratone, M., F. Bitz, E. Clune, H.T. Kung, P. Maulik, H. Ribas, P. Tseng, and J. Webb. Applications and Algorithm Partitioning on Warp. *Proceedings of COMPCON Spring '87*. IEEE Computer Society, February, 1987.

[Annaratone et al. 87d]

Annaratone, M., E. Arnould, R. Cohn, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, K. Sarocky, J. Senko, and J. Webb. Warp Architecture: From Prototype to Production. *Proceedings of the 1987 National Computer Conference*, pages 133-140. AFIPS, June, 1987.

[Baheti et al. 89]

Baheti, R.S., V.J. Karkhanis, D.R. O'Hallaron, and M. Wilson. Fast Mapping of Gravity Equations on Warp. *Proceedings of SPIE Symposium, Real-Time Signal Processing XII*. Society of Photo-Optical Instrumentation Engineers, San Diego, CA, August, 1989.

[Baheti, OHallaron, and Itzkowitz 90]

Baheti, R.S., D.R. O'Hallaron, and H.R. Itzkowitz. Mapping Extended Kalman Filters onto Linear Arrays. *IEEE Transactions on Automatic Control.* 35(12):1310-1319, December, 1990.

[Baxter et al. 90]

Baxter, B., G. Cox, T. Gross, H.T. Kung, D. O'Hallaron, O. Peterson. Building Blocks for a New Generation of Application-Specific Computing Systems. *Proceedings of IEEE Application Specific Array Processor Conference*, pages 190-201. IEEE, 1990.

[Bitz and Kung 87]

Bitz, F. and H.T. Kung. Path planning on the Warp computer; using a systolic array in dynamic programming. *Proceedings of SPIE Symposium, Vol. 826, Advanced Algorithms and Architectures for Signal ProcessingII*. Society of Photo-Optical Instrumentation Engineers, 1987.

[Borkar et al. 90]

Borkar, S., R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, pages 70-81. ACM/IEEE, May, 1990.

[Borkar et al. 88]

Borkar, S., R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, and J. Webb. iWarp: An Integrated Solution to High-Speed Parallel Computing. *Proceedings of Supercomputing 1988*, pages 653-660. IEEE Computer Society and ACM SIGARCH, 1988. Also available as technical report CMU-CS-89-104.

[Borkar et al. 89]

Borkar, Cox, Gleason, Hofsheier, Levine, Meece, More, Peterson, Rankin, Sutton, Urbanski, Hammerstrom, Annaratone, Bono, Cohn, Gross, Kung, Lam, Maulik, Pieper, Tseng, and Webb. *iWarp macro-architecture specification.* Carnegie Mellon University and Intel Corporation, 1989.

[Bruegge 88]

Bruegge, B. *Warp Programming Environment: User Manual.* Technical Report CMU-CS-88-105, Computer Science Department, Carnegie Mellon University, January, 1988.

[Bruegge 88a]

Bruegge, B. Program Development for a Systolic Array. *Proceedings ACM/SIGPLAN 1988 (Parallel programming: experience with applications, languages, and systems)*, pages 31-41. ACM, 1988.

[Bruegge and Gross 88]

Bruegge, B., and T. Gross. An Integrated Environment for Development and Execution of Realtime Programs. *Proceedings of International Conference on Supercomputing*, pages 153-162. ACM, 1988.

[Bruegge and Gross 88a]

Bruegge, B., and T. Gross. A Program Debugger for a Systolic Array: Design and Implementation. *Proceedings of the Second Workshop on Parallel and Distributed Debugging*, pages 174-182. SIGPLAN, ACM, Madison, WI, May, 1988. SIGPLAN Notices Vol. 24, Nr. 1.

[Bruegge et al. 87]

Bruegge, B., C. Chang, R. Cohn, T. Gross, M. Lam, P. Lieu, A. Noaman, and D. Yam. The Warp Programming Environment. *Proceedings of the 1987 National Computer Conference*. AFIPS, June, 1987.

[Bruegge et al. 87a]

Bruegge, B., C. Chang, R. Cohn, T. Gross, M. Lam, P. Lieu, A. Noaman, and D. Yam. Programming Warp. *Proceedings of COMPCON Spring '87*. IEEE Computer Society, February, 1987.

[Cate and Gross 91]

Cate, V. and T. Gross. Combining the Concepts of Compression and Caching for a two-level Filesystem. *Proceedings of Fourth International Conference on Architectural Support for Programming*. ACM/IEEE, April, 1991.

[Clune et al. 87]

Clune, E., J.D. Crisman, G.J. Klinker, and J.A. Webb. *Implementation and performance of a complex vision system on a systolic array machine.* Technical Report CMU-RI-TR-87-16, The Robotics Institute, Carnegie Mellon University, June, 1987.

[Cohn 91]

Cohn, Robert. Source-Level Debugging of Automatically Parallelized Code. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 132-143. Santa Cruz, CA, May, 1991.

[Cohn et al. 89]

Cohn, R., T. Gross, M. Lam, and P.S. Tseng. Architecture and Compiler Tradeoffs for a Long-Instruction-Word Microprocessor. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, April, 1989.

[Cohn et al. 89a]

Cohn, R., H.T. Kung, O. Menzilcioglu, and S.W. Song. Highly reconfigurable array of powerful processors. *Proceedings of SPIE Symposium, Vol. 975, Advanced Algorithms and Architectures for Signal Processing III*. Society of Photo-Optical Instrumentation Engineers, August, 1989.

[Deutch et. al. 87]

Deutch, J., P.C. Maulik, R. Mosur, H. Printz, H. Ribas, J. Senko, P.S. Tseng, J.A. Webb, and I.C. Wu. *Performance of Warp on the DARPA Architecture Benchmarks.* Technical Report CMU-CS-87-148, Computer Science Department, Carnegie Mellon University, September, 1987.

[Dunlay 88]

Dunlay, T.R. Obstacle Avoidance Perception Processing for the Autonomous Land Vehicle. *IEEE International Conference on Robotics and Automation, Vol. 2*, pages 912-918. IEEE, April, 1988.

[Gross 89]

Gross, T. Communication in iWarp systems. *Proceedings of Supercomputing '89*, pages 436-445. November, 1989.

[Gross and Lam 86]

Gross, T., and M. Lam. Compilation for a High-performance Systolic Array. *Proc. of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 27-38. ACM SIGPLAN, Palo Alto, June, 1986.

[Gross and Steenkiste 90]

Gross, T. and P. Steenkiste. Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler. *Software: Practice & Experience.* 20(2):133-155, February, 1990.

[Gross and Sussman 87]

Gross, T., and A. Sussman. Mapping a single-assignment language onto the Warp systolic array. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM, September, 1987.

[Gross et al. 87]

Gross, T., M. Lam, and J. Reinders. Programming Warp in W2. B. Bruegge (editor), *Warp Programming Environment*Carnegie Mellon Department of Computer Science, 1987.

[Hamey et al 89]

Hamey, L.G.C., J.A. Webb, and I. Wu. An architecture-independent programming language for low-level vision. *Computer Vision, Graphics, and Image Processing 48.* ,246-264, 1989.

[Hamey et al. 87]

Hamey, L.G., J.A. Webb, and I.C. Wu. Low-level vision on Warp and the Apply programming model. *Parallel Computation and Computers for Artificial Intelligence.* In Kowalik, J., Kluwer Academic Publishers, 1987, pages 185-199. Also available as technical report CMU-RI-TR-87-17.

[Kanade and Webb 87]

Kanade, T., and J.A. Webb. *End of year report for parallel vision algorithm design and implementation.* Technical Report CMU-RI-TR-87-15, The Robotics Institute, Carnegie Mellon University, June, 1987.

[Kung 88]

Kung, H.T. Systolic Communication. *Proceedings of the International Conference on Systolic Arrays*. IEEE Computer Society, May, 1988.

[Kung 88a]

Kung, H.T. Deadlock Avoidance for Systolic Communication. *Journal of Complexity* . 4(2):87-105, 1988. A revised version appeared in Conference Proceedings of the 15th Annual International Symposium on Computer Architecture, June 1988.

[Kung 88b]

Kung, H.T. Computational Models for Parallel Computers. *Philosophical Transactions of the Royal Society.* 32(1591):357-371, 1988. Also published in Scientific Applications of Multiprocessors, edited by R. Elliot and C.A.R. Hoare, Prentice Hall, New York, 1988, and available as technical report CMU-CS-88-164.

[Kung and Menzilcioglu 87]

Kung, H.T. and O. Menzilcioglu. A general switch architecture for fault-tolerant VLSI processor arrays. *Proceedings of SPIE Symposium, Vol. 827, Real Time Signal Processing X*. Society of Photo-Optical Instrumentation Engineers, 1987. Also available as technical report CMU-CS-87-171.

[Kung et al. 88]

Kung, H.T., H. Printz, T, Mummert, S. Kassam, S. Bulack, K. Maguire, P. Scherer, B. Mullins, M. Woods, E. Danganan, N. Tavani, and J. Gambale. Parallel Array (ASW) Acoustic Processor. *Fourteenth DARPA Strategic Systems Symposium*. Naval Postgraduate School, Monterrey, CA, 1988.

[Kung 90]

Kung, H. T. iWarp Multicomputer with an Embedded Switching Network. *Microprocessors and Microsystems.* 14(1):59-60, January/February, 1990.

[Lam 88]

Lam, M. Software pipelining: An effective scheduling technique for VLIW machines. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318-328. ACM, June, 1988.

[Lam 87]

Lam, M. *An optimizing systolic array compiler.* PhD thesis, Computer Science Department, Carnegie Mellon University, May, 1987.

[Mayer and Baxter 91]

Mayer, H., and B. Baxter. Software and Hardware Parallelism on the iWarp Multicomputer. *Proceedings of the International Conference on Supercomputing*. ACM, Cologne, Germany, 1991.

[Menzilcioglu et al. 89]

Menzilcioglu, O., H.T. Kung, and S.W. Song. Comprehensive evaluation of a two-dimensional configurable array. *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, pages 93-100. June, 1989. Also available as technical report CMU-CS-89-164.

[OHallaron 91]

O'Hallaron, D.R. Uniform Approach for Solving Some Classical Problems on a Linear Array. *IEEE Transactions on Parallel and Distributed Systems.* 2(2):236-241, April, 1991.

[OHallaron 91a]

O'Hallaron, D.R. The Assign Parallel Program Generator. *Proceedings of the 6th Annual Distributed Memory Computing Conference*. May, 1991. A more detailed version of this article is available as technical report, CMU-CS-91-141.

[OHallaron and Fiduccia 91]

O'Hallaron, D.R., and C.M. Fiduccia. Efficient Method for Computing Double Matrix Products. *Circuits, Systems, and Signal Processing.* 10(2):221-232, February, 1991.

[Pieper 89]

Pieper, J.S. *Parallel I/O systems for multicomputers.* Technical Report CMU-CS-89-143, School of Computer Science, Carnegie Mellon University, June, 1989.

[Printz 92]

Printz, Harry. Compilation of Narrowband Spectral Detection Systems for Linear MIMD Machines. *Proceedings of the International Conference on Application-Specific Array Processors*. IEEE Computer Society, 1992.

[Printz 91]

Printz, H. *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine.* PhD thesis, School of Computer Science, Carnegie Mellon University, May, 1991. Also available as technical report, CMU-CS-91-101.

[Printz et.al. 89]

Printz, H., Kung, H., Mummert, T., Schere, P. Automatic Mapping of Large Signal Processing Systems to a Parallel Machine. Letellier, J. (editor), *Realtime Signal Processing XII*, pages 1-15. SPIE, Bellingham, WA, August, 1989.

[Ribas 90]

Ribas, H. B. *Automatic Generation of Systolic Programs from Nested Loops.* PhD thesis, School of Computer Science, Carnegie Mellon University, June, 1990.

[Siegell and Gross 87]

Siegell, B. and T. Gross. Program-specific and architecture-specific simulators. *Proceedings of the 8th International Symposium on Computer Hardware Description Languages and their Applications*. April, 1987.

[Kung 89]

Kung, H.T. Network-based multicomputers: redefining high performance computing in the 1990s. *Proceedings of the Decennial Caltech Conference on VLSI*, pages 49-66. MIT Press, March, 1989. Also available as technical report CMU-CS-89-138.

[Stricker 91]

Stricker, T.M. *Message Routing on Irregular 2D Meshes and Tori.* Technical Report CMU-CS-91-109, School of Computer Science, Carnegie Mellon University, January, 1991.

[Sussman 91]

Sussman, A. *Model-driven Mapping of Computation onto Distributed-memory Parallel Computers.* PhD thesis, School of Computer Science, Carnegie Mellon University, September, 1991. Also available as technical report, CMU-CS-91-187.

[Sussman 92]

Sussman, A. *Execution Models for Mapping Programs onto Distributed Memory Parallel Computers.* Report 92-8, ICASE, March, 1992.

[Tseng 88]

Tseng, P.S. Sparse Matrix Computations on Warp. *Proceedings of the Third International Conference on Supercomputing, Volume 2*, pages 402-410. May, 1988.

[Tseng 90]

Tseng, P.S. A Systolic Array Parallelizing Compiler. *Journal of Parallel and Distributed Computing.* 9117-127, 1990. Based on his Ph.D. thesis, Carnegie Mellon University.

[Tseng 89]

Tseng, P.S. *A parallelizing compiler for distributed memory parallel computers.* PhD thesis, School of Computer Science, Carnegie Mellon University, May, 1989. Also available as technical report CMU-CS-89-148.

[Tseng 88a]

Tseng, P.S. Iterative Sparse Linear System Solvers on Warp. *Proceedings of the International Conference on Parallel Processing*. 1988.

[Tseng et al. 88]

Tseng, P.S., M. Lam, and H.T. Kung. The Domain-Parallel Computation Model on Warp. *Proceedings of SPIE Symposium, Vol. 977, Real-Time Signal Processing XI*. Society of Photo-Optical Instrumentation Engineers, 1988.

[Wallace et al. 89]

Wallace, R.S., J.A. Webb, and I.C. Wu. Machine-independent Image Processing: Performance of Apply on Diverse Architectures. *Computer Vision, Graphics, and Image Processing*. 48265-267, 1989. Also in Proceedings of the Third International Conference on Supercomputing, May 1988.

[Wu 91]

Wu, I-Chen. *Communication complexity for parallel divide-and-conquer*. Technical Report CMU-CS-91-165, School of Computer Science, Carnegie Mellon University, July, 1991.

2 A high-speed, heterogeneous network architecture

Parallel processing is widely accepted as the most promising way to reach the next level of computer system performance. Currently, most parallel processing is done on a single, large parallel machine that provides efficient support only for homogeneous, fine-grained parallel applications. Although these machines have proven to be effective for a large class of applications, they possess three inherent limitations:

- There is a practical limit to how far the performance of these machines can be scaled. When that limit is reached, it becomes desirable to exploit coarse-grained, or task-level, parallelism by connecting together several such machines as nodes in a multicomputer.
- Homogeneous systems are not always the best solution for heterogeneous applications. Many applications process information at multiple, qualitatively different levels. For example, a computer vision system may require image processing on its raw input at the lowest level, and scene recognition using a knowledge base at the highest level. Such application requirements benefit from heterogeneity at both the hardware and software levels.
- Using just large parallel machines is not always the most cost-effective approach for achieving the required performance. Such machines are typically built from custom-made processor boards, although they sometimes use standard microprocessor components. These machines cannot readily take advantage of rapid advances (short product life-cycles) in commercially-available sequential processors.

A multicomputer architecture has the potential to alleviate the above limitations. However, current local area networks (LANs) do not provide the high-bandwidth and low-latency communication that is often required by applications.

The Nectar project addresses the limitations of heterogeneous, coarse-grained parallelism on several fronts: the underlying hardware, communication protocols, node operating system support, and an application interface to the communication network. We have designed and built a prototype Nectar system. A four-node system was operational in late 1988 and was later expanded to 26 nodes.

This report discusses the Nectar architecture and the prototype system. We also present our experience in porting applications.

2.1 The Nectar system architecture

The Nectar architecture (Figure 1) allows exploiting fine-grain parallelism within a task

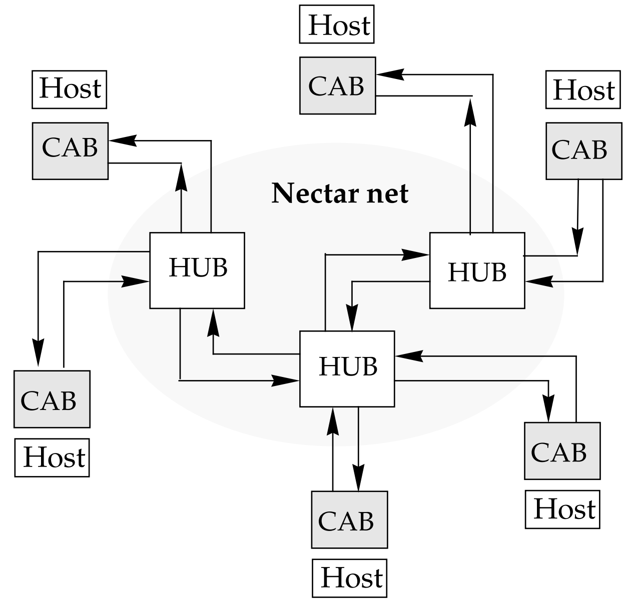

Figure 1: Architecture of a three-HUB Nectar system

executing at a particular node and coarse-grain parallelism among tasks on different nodes. In addition to supporting heterogeneous parallelism, the Nectar system provides low-latency, high-bandwidth communication and is scalable: Distributed heterogeneous machines suitable for specific tasks can be added or replaced without affecting the high bandwidth and low latency between the nodes or requiring modification of the communication software.

The Nectar network is built of fiber-optic lines and consists of one or more “HUBs” that provide low-latency, message-passing communication among the nodes. Hosts are connected to the network through powerful communication accelerator boards, or “CABs” The following subsections describe the special features of the Nectar HUB and CAB.

2.1.1 The Nectar HUB

The core of the Nectar HUB is a high-speed crossbar switch with a flexible datalink protocol implemented in hardware. Crossbar switches substantially reduce the amount of contention found in broadcast-based networks (Ethernet, FDDI), increase bandwidth as nodes are added, and make the system scalable. By combining crossbar switches with fiber-optic lines we increased the network's bandwidth and lowered its latency by at least one order of magnitude over traditional LANs.

In our prototype systems we used 16x16 crossbars. Moderately-sized, 8-bit- wide 32x32 crossbars can be built with off-the-shelf parts, and 128x128 crossbars are feasible with custom VLSI chips. In a single-HUB Nectar system, the HUB's central controller can establish a new connection through the switch every 70-nanosecond cycle. Total latency for setting up a connection and transferring the first byte of a packet through the HUB is ten cycles. In systems with multiple HUBs, connections are made one HUB at a time. This serial connection scheme adds little latency to interprocess communication due the low transfer latency of each individual HUB.

The operation of the HUB is controlled by the CABs through commands that are sent over the fiber and interpreted by the HUB controller. HUBs are connected to nodes or to other HUBs in a multi-HUB system through I/O ports. These ports contain circuitry for optical-to-electrical and electrical-to-optical conversion, input and output queues that connect to the incoming and outgoing fiber, and logic to extract commands from the incoming data steam and insert responses into the outgoing stream. Connections are established by linking the input queue of one I/O port through the crossbar switch with the output queue of another I/O port. Multicasting is achieved by connecting an input queue to multiple output registers.

Simple commands for the most frequently used HUB operations, such as opening and closing connections, are implemented in hardware, while the CAB kernel handles the more complicated datalink protocols such as multicasting. This hardware implementation of low-level inter-HUB communication allows the central HUB controller to execute simple operations in one cycle. In addition, it makes flow control efficient and facilitates building systems with hundreds of nodes.

2.1.2 The Nectar CAB

The CAB is a powerful communication processor. It is plugged into the backplane of each node and interfaces the node with the Nectar network by handling the transmission and reception of data via the network's optical fiber lines [Menzilcioglu and Schlick 91].

The CAB's high-speed processing capability allows it to keep up with the fibers' transmission rate of 100 megabits/s in each direction and ensures the Nectar system's high performance. This capability is provided by a high-performance CPU (built with the latest RISC chips) and fast local memory. Its CPU distinguishes the CAB from other I/O controllers and makes it compatible in speed with custom microsequencers. The CPU also provides flexibility to experiment with different protocols.

Furthermore, high-speed communication is supported by various devices in the CAB, such as a hardware DMA controller, the removal of checksum computation, and special timers that allow time-outs to be set by software. The DMA controller meets local processing needs and, in addition, manages simultaneously the transfer of data between incoming and outgoing fibers and between the VME and the CAB memory. This approach leaves the CAB's CPU free to process protocols and applications.

2.2 Nectar's software architecture

Inefficiency in networking implementations is largely due to excessive overhead caused by

- Application interfaces that require context switching and data copying between a user process and the node' s operating system

- Higher-level protocols that the node must process to ensure reliable communication among applications
- Network interfaces that burden the node with interrupt handling and header processing for each packet

The Nectar software architecture requires substantially less communication overhead on the nodes and as a result, provides low latency and increased bandwidth. We achieved this efficiency by restructuring the way applications communicate: We mapped the high-level CAB/network interface into the application's address space, thereby allowing user processes to read and write messages directly into "mailboxes," special buffers in the CAB's memory. Since applications have direct access to the network and data does not have to be copied, the excessive cost of system calls between applications and the node's operating system is avoided.

We also designed the CAB to process higher-level protocols that control the transmission and reception of data over the network [Cooper et al. 90]. This scheme relieves the host of having to handle packet interrupts, process packet headers, retransmit packets, fragment large messages, and calculate checksums. Interrupts are required only for those high-level events the application is interested in (e.g. delivery of complete messages), but not for such low-level events as the arrival of control packets or timer expiration.

The networking software we developed for our prototype system consists of the CAB kernel and special communication protocols. Both are described below, followed by a description of "Nectarine," a programming interface to Nectar's hardware and low-level software.

2.2.1 The CAB kernel

The CAB kernel provides support for simple, time-critical operations, such as memory management and timers, while the node's operating system handles the more complex operations, such as file I/O. This communication software is built around lightweight processes, which are similar to Mach threads and permit executing multiple activities concurrently, in time-shared fashion. The little state associated with lightweight processes reduces the cost of context switching. This approach transforms what would be a bare "protocol engine" into a flexible, customizable environment for implementing protocols and selected communication-intensive applications, such as load balancing.

2.2.2 Nectar's communication software

The three major communication software components that make use of the CAB kernel are: the datalink protocol and a transport layer, both on the CAB, and the CAB/node interface software.

The *datalink protocol* uses HUB commands for data-packet transfers between CABs, manages HUB connections, and recovers from framing errors and lost HUB commands. Frequently used, simple operations (such as sending a packet to a node in the same HUB cluster) are implemented in hardware as a single HUB command, while more complicated and less common operations (e.g. multicasting or error recovery) are implemented in software.

The *transport layer* regulates the transfer of messages between the CABs' mailboxes. This task involves breaking messages into packets, reassembling them, controlling the message flow, and retransmitting lost and damaged packets. This layer supports three Nectar-specific protocols (a datagram protocol, a reliable message protocol, and a request-response protocol) and the standard Internet protocols (IP, TCP, and UDP).

The datalink and transport protocols cooperate closely, passing packets by reference and thus eliminating data copying.

The efficiency of the *CAB/node interface* derives from the shared-memory approach its software uses. Since the CAB's memory is mapped into the address space of the node process, messages are built and consumed "in place." In other words, node processes invoke services by placing commands directly into CAB's mailbox, and messages are received by polling the CAB's memory. This scheme eliminates copying messages between node and CAB and involving the node's operating system.

2.2.3 A Nectar programming interface

To give the programmer easy access to the Nectar hardware and its low-level software, we developed a programming interface that we called "Nectarine" [Steenkiste 91]. This environment offers features similar to those of communication interfaces available on other distributed-memory machines (such as hypercubes) and, in addition, can accommodate heterogeneous nodes, operating systems, and memories as well as attached devices.

This interface is implemented as a library and represents an application as a series of "tasks" — processes that can execute on any CAB or node and communicate by transferring messages between user-specified buffers. Based on this simple abstraction, the programmer can create tasks, manage buffers, and send or receive messages. In contrast to traditional interfaces, such as sockets, Nectarine allows building messages directly in the mailbox buffers of the CAB's memory, and, by using direct memory access whenever possible, minimizes the number of copy operations.

2.3 System evaluation

We measured latency and throughput in a prototype Nectar system that consisted of two HUBs (with 16x16 crossbars) and 26 hosts (Sun4 workstations).

Latency

The following table shows roundtrip latency in microseconds for UDP and Nectar-specific protocols.

Protocol	host-to-host	CAB-to-CAB
datagram	325	179
reliable message	414	241

Table 1: Roundtrip latency

Protocol	host-to-host	CAB-to-CAB
request-response	438	287
UDP	842	536

Table 1: Roundtrip latency

Host-to-host roundtrip latency for a single-word message using the UDP protocol was about one fourth that in traditional, Ethernet-based networks. About 40% of the time taken to send a message between two host processes using the Nectar-specific datagram protocol is spent in the CAB/node interface, mainly because each read or write over the VME bus takes about 1 millisecond.

Throughput

Figure 2 shows the throughput rates for two CAB threads using the on-CAB implementation of TCP/IP and the Nectar-specific reliable message protocol (RMP). For small packets of

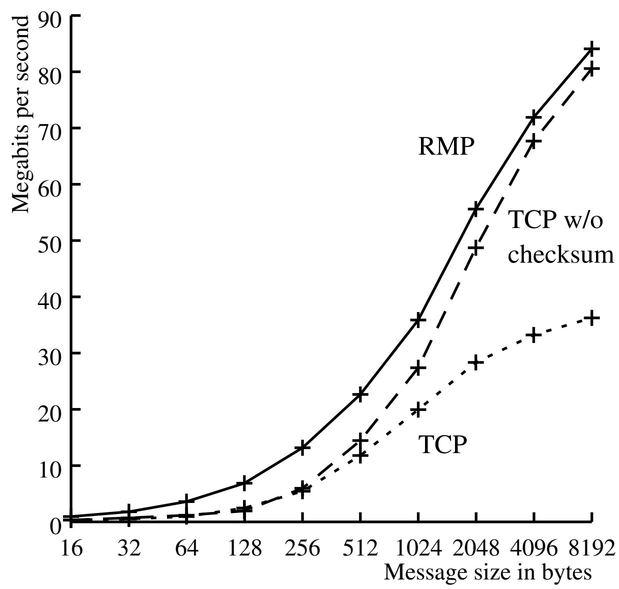

Figure 2: CAB-to-CAB throughput

up to 256 bytes, the per-packet overhead is made up completely of the time required in TCP/IP processing, and throughput doubles when the packet size doubles. For packets larger than 256 bytes, the transmission time becomes significant and throughput increases more slowly. The TCP/IP protocol's lesser performance compared to RMP is mostly due to the cost of checksums implemented in software. RMP doesn't have this cost since it relies on the CRC provided by the CAB hardware. Without software checksums, TCP is almost as fast as RMP.

Based on the same protocols, the curves in Figure 3 show the throughput rates between two host processes. These curves have the same shape as the CAB-to- CAB throughput curves but flatten earlier due to the slow VME bus, whose bandwidth (about 30 Mbit/s)

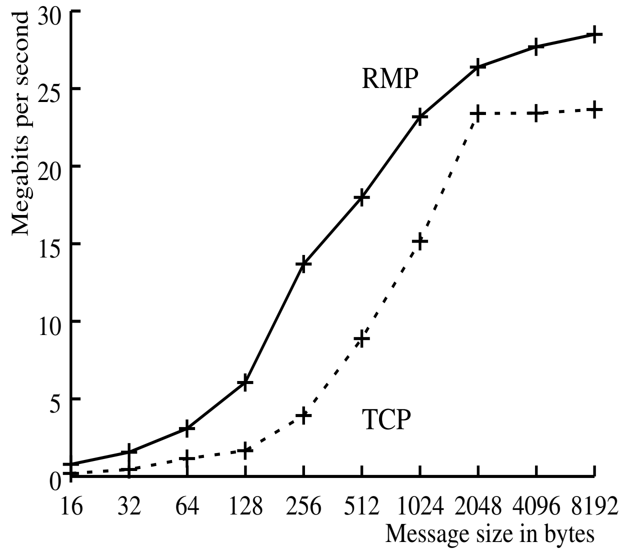

Figure 3: Host-to-host throughput

limits the throughput of both protocols. Maximum TCP/IP bandwidth is approximately 24 Mbit/s, a considerable improvement over the maximum host-to-host throughput of 6.4 Mbit/s when the Sun4 CPU handles TCP/IP processing and the CAB acts simply as network interface. In fact, the performance with the CAB functioning as a simple network interface is worse than the 7.2 Mbit/s throughput achieved with on-board host/Ethernet interfaces that bypass the VME bus.

2.4 Distributed event monitoring

Program and performance debugging as well as load balancing require keeping track of various runtime-system activities — a difficult task in a dynamic and distributed computer network. Rather than providing a multiplicity of separate tools, we created BEE, a basis that facilitates building tools for monitoring, debugging, and evaluating events in heterogeneous, distributed systems, such as Nectar [Bruegge 90][Bruegge and Steenkiste 91]. A test version of this environment was operational in March 1990.

Our monitoring system views an executing, distributed program as a generator of different events that can be used to characterize the network's behavior. While event-based systems are simple, handling and storing events increases the application's execution time. BEE reduces the overhead of runtime monitoring by allowing users to take the interpretation of events out of the application program and run it on another node. To provide an overview of the activities in the entire system, as required for dynamic load balancing, our event interpreter can tap into client applications executing on different nodes. A client process can also be monitored by several event interpreters tapping into the same event stream to provide different views of the client's behavior. Another feature of the BEE environment is that it allows users to filter out unneeded events, thereby reducing the number of events to be processed and, as a result, the overhead.

2.5 Applications on Nectar

We demonstrated our system's effectiveness and versatility by implementing several large and complex applications on the Nectar prototype [Kung et al. 91]. Three applications with widely different domains and programming models are briefly discussed below: a high-performance switch-level circuit simulator (COSMOS), a geometric modeling package (NOODLES), and a simulation of air pollution in Los Angeles. We also ported successfully a parallel, solid modeling program (Mistral-3), distributed algorithms providing exact solutions to several traveling-salesman problems, an image-processing application (see Chapter 4), and a chemical flowsheeting simulation.

Porting the COSMOS application to the Nectar prototype was relatively easy since the main data structure (the circuit) was already partitioned and a sequential implementation existed on the workstations that constitute the system's nodes. The COSMOS circuit was parallelized by distributing it over the Nectar nodes. It benefits from the low overhead, low latency communication on Nectar. Total execution time of the COSMOS simulator was considerably shortened due to the CAB's effectiveness in overlapping computation with communication.

The NOODLES application is a geometric modeling package that was developed at the Engineering Design and Research NSF Center at Carnegie Mellon. Geometric models are represented by complex data structures, which, in the Nectar implementation, had to be replicated on all nodes and are kept consistent through an application-specific consistency algorithm. Parallelizing NOODLES facilitates working with larger models interactively. Since computation in NOODLES is very data dependent, dynamic load balancing was used to achieve good node utilization. Running the load balancer on the CAB allowed load balancing at very fine granularity.

In the simulation of air pollution in Los Angeles, pollutant particles are tracked as they are carried by the moving air. This application combines very finely grained tasks (i.e., tracking the particles) with coarse-grained ones (i.e., calculating the wind-velocity vectors based on measured input data). Distributing the wind-velocity vectors requires considerable bandwidth and benefits from the availability of hardware-based multicasting. As in the NOODLES application, dynamic load balancing increased the performance speed.

In spite of the complexity of these applications, it took relatively little effort to implement them on the Nectar system, mainly because the network's nodes are general-purpose computers: The parallelized applications can use the host's software and available application code, and users can work in a familiar environment. Since Nectar's bandwidth and latency characteristics are similar to those of state-of-the-art custom-made multicomputers, the required heavy communication among Nectar nodes does not create a bottleneck. The circuit and the air-pollution simulations in particular demonstrated the effectiveness of the Nectar system for parallelizing large, heterogeneous applications over a network and providing a relatively easy means for speeding performance.

2.6 Bibliography

[Arnould et al. 89]

Arnould, E.A., F.J. Bitz, E.C. Cooper, H.T. Kung, R.D. Sansom, and P. 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, April, 1989. Also available as technical report CMU-CS-89-101.

[Bruegge 90]

Bruegge, Bernd. *BEE: A Basis for Distributed Event Environments (Reference Manual).* Technical Report CMU-CS-90-180, School of Computer Science, Carnegie Mellon University, November, 1990.

[Bruegge and Steenkiste 91]

Bruegge, B., P. Steenkiste. Supporting the Development of Network Programs. *Proceedings of the Eleventh International Conference on Distributed Systems*. IEEE, May, 1991.

[Bruegge and Walzer 90]

Bruegge, B., F. Walzer. Runtime Monitoring in a Network Environment. *ICPP*, pages 278-279. IEEE, August, 1990.

[Bruegge et al. 91]

Bruegge, B., H. Nishikawa, P. Steenkiste. Computing over Networks: An Illustrated Example. *Proceedings of the Sixth Distributed Memory Computing Conference*. IEEE, April, 1991.

[Cooper 90]

Cooper, E.C. Programming Language Support for Multicast Communication in Distributed Systems. *Proceedings of the Tenth International Conference on Distributed Computing Systems*. IEEE, May, 1990. Also available as technical report CMU-CS-90-121.

[Cooper et al. 90]

Cooper, E.C., P. Steenkiste, R.D. Sansom, B.D. Zill. Protocol Implementation on the Nectar Communication Processor. *Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols*. ACM, September, 1990.

[Kung et al. 91]

Kung, H.T., P. Steenkiste, M. Gubitoso, M. 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.

[Menzilcioglu and Schlick 91]

Menzilcioglu, O., S. Schlick. Nectar CAB: A High-speed Network Processor. *Proceedings of the Eleventh International Conference on Distributed Systems*. IEEE, May, 1991.

[Siegel and Cooper 89]

Siegel, E., E.C. Cooper. Implementing OSI layer functionality. *Proceedings of the Third Workshop on Large-Grain Parallelism*. Software Engineering Institute, October, 1989.

[Steenkiste 89]

Peter Steenkiste. Nectar - A Testbed for Research in Distributed Systems. *Proceedings of the Third Workshop on Large-Grain Parallelism*. Software Engineering Institute, October, 1989.

[Steenkiste 91]

Peter Steenkiste. A Symmetrical Communication Interface for Distributed-Memory Computers. *Proceedings of the Eleventh International Conference on Distributed Systems* . IEEE, April, 1991.

3 Parallel operating systems

3.1 The Mach approach

We began developing the Mach operating system in late 1984 with the goal to provide support for the newly emerging general-purpose, tightly and loosely coupled multiprocessor architectures and compatibility with UNIX BSD 4.3 and other operating-system environments. Our overall plan was to restructure UNIX BSD 4.3 code to take advantage of some UNIX capabilities, replace others with our own facilities (such as scheduling and memory management), and, eventually, separate the operating system's control of basic hardware resources from unique UNIX characteristics.

The resulting operating system [Rashid et al. 89][Rashid et al. 89] consists of a kernel that provides a set of key facilities and allows layering and running simultaneously various existing environments in "native mode." The advantage of this scheme is that machine-dependent software for different hardware bases needs to be written only once. Furthermore, separating from the kernel those functions that provide binary compatibility with existing operating system environments makes the system modular. The advantage of modularity is that it simplifies maintenance and increases security.

Status of the Mach operating system at the beginning of this reporting period

Mach was fully compatible with UNIX BSD 4.3 and provided the following key features:

- Simplified, user-level construction and management of concurrent programs. This feature is provided by Mach's separation of the traditional "process" into a *task* and one or more *threads*, which run within the task. Each Mach task represents a system resource, foremost its address space. A thread is a Mach control unit and consists of a program counter and a set of data registers. Threads share the capabilities of the task in which they operate.
- Extensible and secure interprocess communication (IPC), the kind of I/O channel between threads that exists in a multiprocessor with a message-passing bus or between workstations on a network
- Architecture-independent virtual-memory management, which provides flexible facilities for sharing memory between tasks and manipulating large, sparse address spaces.
- Tight integration of Mach's IPC and virtual-memory management. This scheme allows virtual-memory remapping with *copy-on-write* and *copy-on-reference* message passing to avoid unnecessary data copying.

We demonstrated the practical applicability of these facilities by porting Mach to the four-processor VAX11/784, the IBM RT PC, the Sun3 workstation, and the 16-processor Encore Multimax, all machines whose processors have uniform access to the system's shared memory. In January 1987, we released the first official version of the Mach operating system (Mach 0) to sites outside Carnegie Mellon.

3.2 Research effort 1987-1990

Our main effort under this contract consisted in separating Mach's control of basic hardware resources from UNIX functionality and moving all UNIX code into one or more user-level processes to be implemented on the resulting "pure Mach kernel," which is totally free of BSD-derived code and can be distributed without license restrictions. In addition to refining the kernel's functionality, particularly its support for tightly and loosely coupled multiprocessors, we

- Further integrated Mach IPC and virtual-memory management to allow external, user-level management of memory. In this scheme, the system's memory is represented as an extensible, single-construct, virtual-memory object that can be created and managed by applications acting as external pagers. A primary application is the "network memory server" for no-remote-memory-access (NORMA) multiprocessor architectures [Young 89], [Forin et al. 89a], [Forin et al. 89]
- Improved user control of multiprocessor allocation and scheduling, such as "gang" scheduling [Black 90a]
- Improved memory management for nonuniform memory-access (NUMA) [Black et al. 89a].

We demonstrated the effectiveness of these features by developing various operating-system emulations on top of Mach: An "in-kernel" UNIX 4.3 BSD emulation; a single-server and a multiserver "out-of-kernel" UNIX 4.3 BSD emulation; and two non-UNIX emulations (DOS and Macintosh). All emulations will be discussed in the following sections.

The commercial distribution of the Mach operating system was carried out by a subcontractor, MTXINU, as described in section 3.2, page 37.

Furthermore, we developed software for monitoring the Mach kernel and employed it to improve Mach's scheduler (see section 3.3, page 37).

3.2.1 In-kernel UNIX emulation

In our first UNIX emulation, we layered UNIX BSD 4.3 functionality above the kernel primitives, but packaged both as a monolithic unit running in privileged state. This emulation, distributed as Mach 2.5, was the first step toward developing a kernel interface for other operating-system environments.

Compared to traditional UNIX implementations, Mach 2.5 on a Sun 3/60 workstation executed simple compilation benchmarks nearly 40% faster than SunOS 4.0, Sun Microsystems' own UNIX version. UNIX `fork` and `exec` operations were nearly two times faster with this Mach version than with the SunOS.

3.2.2 Out-of-kernel UNIX emulations

Our next goal was to separate Mach's layers further, so that only a "pure kernel" or "microkernel" runs in privileged mode, while the other components of the environment execute as one or more client/server processes on top of it [Golub et al. 90]. Instead of being handled directly by an in-kernel operating system, system call traps issued by application programs are redirected through the Mach microkernel to user-level servers or modules emulating the operation system. By insulating the software from the hardware, the kernel becomes a type of universal socket into which more than one operating-system environment can be plugged.

The technology that provides binary compatibility with various operating-system environments and enables out-of-kernel "non-native" implementations is Mach's **emulation library**. It is transparently loaded into the address space of the application program — to optimize data transfer — and contains routines equivalent to the UNIX system-call handler. These library routines intercept the program's system calls, transform them into remote procedure calls, and forward them through the kernel's communication and memory facilities to the appropriate server or servers. The emulation library functions both as translator for system service requests and as cache for the results returned from these requests.

A single-server UNIX emulation

Emulating an operating system as a single-server application program has the following advantages:

- The server has a similar structure as the UNIX in-kernel implementation and is alone responsible for providing all OS environment semantics. It has global knowledge of all information required for the implementation and allows extremely fast context switching between threads.
- The server is completely pageable and, by sharing data structures and stack space, it can make more efficient use of memory than a multiple-server implementation.
- It was easy to transform our in-kernel emulation into a single-server program, which preserves both existing code and semantics.

While ours was not the first single-server implementation, the others are based on a different relationship between the system kernel and the supported OS environment. IBM's CP/67, for example, uses a virtual-machine approach; in AT&T's MERT system the kernel is layered on a simple message engine; and CHORUS loads into the kernel emulation-assist code specific to the emulated operating-system environment.

Our approach, on the other hand, is to provide all UNIX facilities through a client/server program running on top of a kernel that contains no UNIX-specific functionality. The UNIX server provides all system services and resources commonly associated with a BSD 4.3 environment, while the kernel handles IPC, scheduling, and virtual-memory services. Other functions of a traditional UNIX system, such as support for higher-level, application-specific resource abstractions (files and sockets), are handled by Mach's transparent system-call-emulation library.

The server program is contained in a single, multithreaded Mach task and typically invoked by a Mach message for each system call issued by the application process. Mach IPC is the primary communication tool between the UNIX server and a UNIX application, but shared memory may also be employed. Both the UNIX server and the emulation library can chose the communication interface they prefer: Message passing is more natural for network communication while, in a uniprocessor or tightly coupled multiprocessor, large amounts of data are more efficiently transferred using the virtual-shared-memory scheme. In shared-memory communication, the UNIX server can act as memory-object manager, or "external inode pager," for 4.3BSD files mapped into the application's address space.

Performance

We ported the pure Mach kernel (version 3.0) with the single-server UNIX to uniprocessor and multiprocessor VAX systems, DECstations 3100 and 5000, the Sun SPARCstation, and i386-PC clones.

To evaluate the microkernel's performance we ran the Andrew Benchmark as modified by Ousterhout. The table below shows the results on a Sun 3/60 (with 8 Mbytes of memory

Operating system	Directory creation	File copy	Recursive file stats	Find	Compile	Elapsed
Mach 2.5	4	20	13	26	336	399
3.0 plus UNIX server	1	20	24	34	332	411

Table 3-1 : Modified Andrew Benchmark times in seconds

and a Priam 300 Mbyte disk drive) in comparison to the performance achieved with the Mach 2.5 Release.

A multiserver UNIX emulation

Implementing an operating-system environment as multiple, small-grained server tasks provides the advantage of greater portability, flexibility, and security compared to a single-server emulation: Servers can easily be added, removed, and individually secured and verified. In addition, this approach simplifies developing, upgrading, debugging, and maintaining the system.

The prototype multiserver UNIX emulation we developed is only a step toward our greater goal of providing a common, object-oriented framework that allows implementing a wide range of existing operating system environments on a single host and executing them concurrently.

In designing our prototype, we were guided by the following principles:

- Developing many independent, modular servers to maximize flexibility and extensibility
- Identifying common functions that different services and emulations require—such as authentication—and implementing them separately
- Concentrating on the details of the interface that is emulated in the emulation library and keeping the services as generic as possible to maximize the reusability of entire components

- Specifying system interfaces and implement low-level components by using object-oriented techniques, in order to maximize adaptability and low-level reusability
- Including client-side processing in the emulation library to increase system performance.

System architecture

Our prototype system is organized as three independent software layers:

- The Mach microkernel providing the basic facilities for the execution of the various system components
- A collection of independent servers specializing in particular functions
- An emulation library similar to that used in the single-server system.

To maximize portability and reusability, we standardized the basic supporting facilities and interfaces between the various system components and made them (to different degrees) independent of the target environment.

Each server executes in a separate Mach task and uses a combination of IPC and shared memory to communicate with the library and, in some cases, with other servers. There are "application servers" and "system servers." The former are considered high-level servers, since they interact directly with the applications. They provide basic functions of an operating system and include:

- One or more file servers
- A terminal (TTY) server
- A task server (process manager)
- A local IPC server supporting basic communication, such as UNIX pipes, between applications
- One or more network servers that provide access to the network by implementing various protocol families
- A device server that controls user access to the physical devices managed by the kernel.

System servers are indirectly invoked by the application and support its operation by performing authentication, name-space management, diagnostics, etc.

To simplify handling and combining the individual servers, we developed an object-oriented programming facility, called "MachObjects." This facility is integrated with Mach's network communication and transparently extends traditional kernel-based functions to user-level servers.

In early 1990 our prototype system provided a basic level of functionality. It supported a command interpreter (Bourne shell or C shell) and many simple UNIX commands, such as `ls`, `cp`, `rm`, `ln`, and `vi`, including commands required for compiling small **C** programs, such as `make`, `cc`, and `ld`.

3.1.1 Other environments running on Mach

The two other operating systems we implemented as emulations executing within the Mach framework are the single-user MS-DOS [Malan et al 91] and the Apple Macintosh environments. Our approach was to provide all functionality required by DOS and Macintosh applications through the emulation library.

The emulation library has three functions: It serves as interface to the original code of the emulated operating systems; it gives applications access to the hardware; and it provides external services.

Significant portions of the native operating system are allowed to run in "native mode," since few requests involve operations that require the Mach kernel's intervention. For example, to access I/O devices the native DOS filesystem code invokes low-level BIOS functions and similarly, once the Macintosh system's display has been appropriately mapped, graphics primitives writing to the display can execute directly from the Mac ROMs. Since the system-call invocation mechanisms in the DOS and Macintosh systems differ from those in Mach and UNIX on the same hardware, the Mach kernel treats invocations from the emulated systems as exceptions and redirects them to the emulation library. In its role as interface, the emulation library in turn directs these system calls to the native operating system, which provides the services.

In addition to invoking the code of the emulated operating systems, the emulation library creates virtual access to hardware devices, such as displays, disks, and keyboards. Providing multiuser support is necessary since many applications for single-user systems expect direct access to some devices without having to invoke a system service.

Finally, the emulation library provides most system services by translating requests for services from shared resources, such as UNIX file-system disks, into requests to the specific servers managing them and to the hardware devices managed by the Mach kernel.

Both emulation systems are loaded from the UNIX file system (which is transparently available to the applications running on them) and execute, as Mach 3.0 operating-system servers, in parallel with a stable UNIX BSD 4.3 server.

Almost all applications supported by the native DOS and Macintosh operating systems can be implemented without change on the respective emulation system. By enhancing Mach's RPC with "continuations," first-class objects for managing the state of a computation, we were able to show that an applications' performance under the emulation is comparable to that on the native system [Draves 91].

The DOS emulation on Intel's 80386 or 80486 virtual-memory processor supports business applications, such as Lotus 1-2-3, WordPerfect, and Windows 3.0, as well as games, such as Wing Commander and Space Quest IV. The MacMach system on the Macintosh II, IIx, IIcs, IIci, IIfx, and SE/30 architectures emulates Apple's Macintosh System 7.0 and allows running Multifinder, X11r4 Windows, business applications (such as MacDraw 2.0, Excel, Powerpoint), and several games. Both emulations support most display types, such as EGA and VGA, as well as various sound boards.

3.2 Mach distribution

Carnegie Mellon subcontracted MTXINU to coordinate and supervise the distribution of Mach 2.5. MTXINU's effort involved evaluating Carnegie Mellon's Mach kernel and its BSD-compatible software on four platforms — the IBM RT PC, DEC VAX, the Sun3 families, and the i386 platform. MTXINU then integrated the kernel with BSD code, verified the operation of the integrated software, and shipped it to various test sites.

In addition, MTXINU negotiated and integrated licensing terms and conditions, compiled an annotated bibliography of publications on the included software packages, and wrote a 12-volume Mach2.5/4.3BSD manual as well as installation instructions for the Release-1 Distribution.

3.3 Support for parallel and distributed programming

To assist programmers in quickly and efficiently evaluating performance trade-offs between different Mach kernel versions, we developed monitoring software and integrated it with the architecture-independent programming and instrumentation environment (PIE) we had developed under a previous contract [Lehr et al. 89].

The PIE system provides a toolset for instrumenting parallel and concurrent systems at the hardware, operating system, and application levels. These tools and PIE's own graphic display facility allow programmers to observe, debug, and analyze the behavior and performance of their system versions to identify, for example, the hardware approach that most effectively utilizes the available resources (CPUs, buses, caches and memories) or the software scheme that produces the best scheduling or virtual-memory management performance.

Mach kernel monitor (MKM)

In contrast to other monitoring schemes that trace all measurable processes and thus merely gather statistics, our approach permits users to selectively monitor specific computations — each independently or several simultaneously [Lehr et al. 89a]. (The term "computation" is meant to be more general than "program" or "process" and to include system and application software.) We provide this capability by modeling the kernel monitor on Mach's task and thread abstractions. Furthermore, our monitor has low overhead and a high bandwidth.

The kernel monitor comprises:

- A list of the particular threads to be monitored
- Predefined sensors, embedded within the kernel code, for detecting specific events, such as instances of context switching
- A data structure that provides for each process a pointer to a separate buffer and specifies*:*
    - The particular event to be monitored
    - Creation/termination of monitored threads

- The processor on which a particular thread is running
- A time stamp

For monitoring Mach's scheduler activities, MKM operates in the following manner: When a thread switches, a software sensor detects which, if any, monitor is tracking it and writes the event to that particular monitor's protected buffer. This storage area is organized as a relational database: Events are ordered chronologically and according to the threads and processors they are associated with.

Monitoring context switching

We implemented the kernel monitor and used our PIE environment to observe and analyze the context-switching activity of Mach kernel versions with different scheduling algorithms. Based on these analyses we were able to determine how each scheme affects computation performance and modify the scheduler accordingly.

First, we studied the effects of several uniprocessor scheduling algorithms on sequential applications.The MicroVAXII, on which we executed the applications, was booted in single-user mode to reduce the number of threads competing for the machine's processor. We observed that the first scheduling algorithm, designed to switch threads every 100 ms, produced "context-switch flutter," indicating that threads switch frequently to themselves. To avoid this problem and reduce the number of context switches, we modified the scheduler to increase dynamically the time slices it allocates to individual threads. Since this change did not provide satisfactory performance either, we further improved the scheduler by introducing a thread-priority-evaluation policy.

Next, we observed two matrix-multiplier computations executing in parallel on a 16-processor, shared-bus machine. The Mach operating-system version employed our simple, nondiscriminating scheduler that assigns equal time slices to competing threads. The PIE environment's special "cpu-view" provided insight into how the computations utilized the available processors during their execution and revealed ways to optimize the scheduler [Black 90].

Controlling monitor perturbation

While providing important information about an executing computation, the kernel monitor competes for hardware and software resources and thus perturbs the very phenomena it seeks to observe [Lehr 90]. Regular delays are caused by the sensors as they check whether a monitor is tracking a thread. However, this perturbation creates less overhead than the sensor's intermittent information retrieval or storage in the kernel buffers, both procedures requiring a system call.

Since monitoring delays cannot be completely eliminated, we must compensate for them, above all for the perturbation caused by event-information retrieval and storage. We developed a compensation algorithm for monitored computations running on a uniprocessor. This scheme adjusts each event's time stamp according to the number of preceding events that were detected (subtracting from the time stamp the sum of all previous sensor firing times plus the time consumed by monitor threads). This algorithm brings the monitored computation's execution time within 1% of the estimated time the computation would take if it were unmonitored. With this accuracy, our software monitoring technique is a practical alternative to hardware-based instrumenting, which would be nonportable and relatively expensive.

3.4 Bibliography

[Barrera 91]
Barrera, Joseph S. A fast Mach network IPC implementation. *Proceedings of the Second USENIX Mach Symposium*. USENIX, November, 1991.

[Bershad 91]
Bershad, B.N., and M.J. Zekauskas. *Midway: Shared-memory parallel programming with entry consistency for distributed memory multiprocessors.* Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, September, 1991.

[Bershad 91a]
Bershad, B. N. *Mutual exclusion for uniprocessors.* Technical Report CMU-CS-91-116, School of Computer Science, Carnegie Mellon University, April, 1991.

[Bershad 91b]
Bershad, B.N. *Practical considerations for lock-free concurrent objects.* Technical Report CMU-CS-91-183, School of Computer Science, Carnegie Mellon University, September, 1991.

[Black 90]
Black, David L. *Scheduling and resource-management techniques for multiprocessors.* PhD thesis, Computer Science Department, Carnegie Mellon University, July, 1990. Also available as technical report CMU-CS-90-152.

[Black 90a]
Black, D.L. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer.* 23(5):35-43, May, 1990. Also available as technical report CMU-CS-90-125.

[Black and Sleator 89]
Black, D.L., and D.D. Sleator. *Competitive algorithms for replication and migration problems.* Technical Report CMU-CS-89-201, School of Computer Science, Carnegie Mellon University, November, 1989.

[Black et al. 89]
Black, D.L., R.F. Rashid, D.B. Golub, C.R. Hill, and R.V. Baron. Translation lookaside buffer consistency: a software approach. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III).* ACM, April, 1989. Also available as technical report CMU-CS-88-201.

[Black et al. 88]

Black, D.L., D.B. Golub, K. Hauth, A. Tevanian, and R. Sanzi. The Mach Exception Handling Facility. *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*. ACM, May, 1988. Also available as technical report CMU-CS-88-129.

[Black et al. 89a]

Black, D., A. Gupta, and W.D. Weber. Competitive management of distributed shared memory. *Proceedings of Spring COMPCON '89*. IEEE Computer Society, February, 1989.

[Caswell and Black 90]

Caswell, D. and D. Black. Implementing a Mach debugger for multithreaded applications. *Proceedings of the Winter 1990 USENIX Technical Conference and Exhibition*. USENIX, January, 1990. Also available as technical report CMU-CS-89-154.

[Cate 90]

Cate, V. *Two levels of filesystem hierarchy on one disk.* Technical Report CMU-CS-90-129, School of Computer Science, Carnegie Mellon University, May, 1990.

[Cooper and Draves 88]

Cooper, E.C. and R.P. Draves. *C Threads.* Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June, 1988.

[Draves 91]

Draves, R.P., B. Bershad, R.F. Rashid, and R. W. Dean. *Continuations: Unifying thread management and communication in operating systems.* Technical Report CMU-CS-91-115, School of Computer Science, Carnegie Mellon University, March, 1991.

[Draves 91a]

Draves, R.P. Page replacement and reference bit emulation in Mach. *Proceedings of the Second USENIX Mach Symposium*. USENIX, November, 1991.

[Forin 91]

Forin, A., D. Golub, and B.N. Bershad. *An I/O system for Mach 3.0.* Technical Report CMU-CS-91-191, School of Computer Science, Carnegie Mellon University, October, 1991.

[Forin 89]

Forin, A. Debugging of Heterogeneous Parallel Systems. *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 130-140. ACM, January, 1989.

[Forin et al. 89]

Forin, A., R. Rashid, and J. Barrera. A Distributed Memory Server. *Proceedings of the Winter USENIX Technical Conference and Exhibition*. USENIX, 1989.

[Forin et al. 89a]

Forin, A., J. Barrera, M. Young, and R. Rashid. Design, implementation, and performance evaluation of a distributed shared-memory server for Mach. *Proceedings of the Winter USENIX Technical Conference and Exhibition*, pages 229-244. USENIX, February, 1989. Also available as technical report CMU-CS-88-165.

[Forin et al. 89b]

Forin, A., J. Barrera, and R. Sanzi. The shared memory server. *Proceedings of the Winter USENIX Technical Conference and Exhibition*. USENIX, January, 1989.

[Golub et al. 90]

Golub, D., R. Dean, A. Forin, and R. Rashid. UNIX as an application program. *Proceedings of the Summer USENIX Technical Conference and Exhibition*. USENIX, June, 1990.

[Gupta et al. 87]

Gupta, A., C.L. Forgy, D. Kalp, A. Newell, and M. Tambe. *Results of parallel implementation of OPS5 on the Encore multiprocessor.* Technical Report CMU-CS-87-146, Computer Science Department, Carnegie Mellon University, August, 1987.

[Lehr 90]

Lehr, T. *Compensating for perturbation by software performance monitors in asynchronous computations.* PhD thesis, Department of Electrical and Computer Engineering, April, 1990.

[Lehr et al. 89]

Lehr, T., Z. Segall, D. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman. *Visualizing performance debugging.* Technical Report CMU-CS-89-140, School of Computer Science, Carnegie Mellon University, April, 1989.

[Lehr et al. 89a]

Lehr, T., D. Black, Z. Segall, and D. Vrsalovic. *MKM: Mach kernel monitor description, examples, and measurements.* Technical Report CMU-CS-89-131, School of Computer Science, Carnegie Mellon University, March, 1989.

[Malan et al 91]

Malan, G., R. Rashid, D. Golub, and R. Baron. DOS as a Mach 3.0 application. *Proceedings of the Second USENIX Mach Symposium.* USENIX, November, 1991.

[Printz 90]

Printz, H. and D. Servan-Schreiber. *Foundations of a computational theory of catecholamine effects.* Technical Report CMU-CS-90-105, School of Computer Science, Carnegie Mellon University, May, 1990.

[Rashid 87]

Rashid, R.F. Mach: Layered protocols vs. distributed systems: A position paper. *IEEE.* January 1987.

[Rashid 87a]

Rashid, R.F. Mach: A new foundation for multiprocessor systems development. *Proceedings of COMPCON '87*. IEEE, February, 1987.

[Rashid 88]

Rashid, R. From RIG to Accent to Mach: the evolution of a network operating system. *The Ecology of Computation.* In Huberman, B.A., North-Holland, 1988.

[Rashid et al 88]

Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W.J. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers.* 37(8):896 - 908, 1988. Also in Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, October 1987; and available as technical report CMU-CSD-87-140.

[Rashid et al. 89]

Rashid, R., D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: a system software kernel. *Proceedings of COMPCON '89*. IEEE Computer Society, February, 1989.

[Rashid 89]

Rashid, R. A catalyst for open systems. *Datamation.* 35(10):32-33, 1989.

[Reilly 89]

Reilly, Matthew H. *Implementation and evaluation of a hardware/software performan ce monitor for parallel programs.* PhD thesis, Department of Electrical and Computer Engineering, May, 1989.

[Sansom 88]

Sansom, R.D. *Building a Secure Distributed Computer System.* Technical Report CMU-CS-88-141, Computer Science Department, Carnegie Mellon University, May, 1988.

[Subramanian 91]

Subramanian, I. Managing discardable pages with an external pager. *Proceedings of the Second USENIX Mach Symposium*. USENIX, November, 1991.

[Tevanian 87]

Tevanian, A. *Architecture-independent virtual memory management for parallel and distributed environments: the Mach approach.* PhD thesis, December, 1987. Available as technical report CMU-CS-88-106.

[Tevanian and Rashid 87]

Tevanian Jr., A. and R.F. Rashid. *Mach: A basis for future UNIX development.* Technical Report CMU-CS-87-139, Computer Science Department, Carnegie Mellon University, June, 1987.

[Tevanian et al. 87]

Tevanian Jr., A., R. Rashid, M.W. Young, D.B. Golub, M.R. Thompson, W. Bolosky, and R. Sanzi. A UNIX interface for shared memory and memory mapped files under Mach. *Proceedings of the Summer USENIX Technical Conference and Exhibition*. USENIX, June, 1987.

[Tevanian et al. 87a]

Tevanian Jr., A., R.F. Rashid, D.B. Golub, D.L. Black, E. Cooper, and M.W. Young. Mach threads and the UNIX Kernel: the battle for control. *Proceedings of the Summer USENIX Technical Conference and Exhibition*. USENIX, June, 1987.

[Tokuda 90]

Tokuda, H., T. Nakajima, and P. Rao. Realtime Mach: Towards a predictable realtime system. *Proceedings of the 1990 USENIX Mach Workshop*. USENIX, October, 1990.

[Tokuda 91]

Tokuda, H., and T. Nakajima. Evaluation of realtime synchronization in realtime Mach. *Proceedings of the Second USENIX Mach Symposium*. USENIX, November, 1991.

[Vrsalovic et al. 88]

Vrsalovic, D., Z. Segall, D. Siewiorek, R. Gregoretti, E. Caplan, C. Fineman, S. Kravitz, T. Lehr, and M. Russinovich. *Performance efficient parallel programming in MPC.* Technical Report CMU-CS-88-167, Computer Science Department, Carnegie Mellon University, July, 1988.

[Vrsalovic et al. 88a]

Vrsalovic, D., D. Siewiorek, Z. Segall, and E. Gehringer. Performance prediction and calibration for a class of multiprocessor systems. *IEEE Transactions on Computers.* 37(11):1353 - 1366, 1988.

[Vrsalovic et al. 89]

Vrsalovic, D., Z. Segall, D. Siewiorek, F. Gregoretti, E. Caplan, C. Fineman, S. Kravitz, T. Lehr, and M. Russinovich. MPC - Multiprocessor C language for consistent abstract shared data type paradigms. *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 1989.

[Young 89]

Young, M.W. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* PhD thesis, School of Computer Science, Carnegie Mellon University, November, 1989. Available as Technical Report CMU-CS-89-202.

[Young et al. 87]

Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. *Proceedings of the 11th Symposium on Operating System Principles*. ACM, November, 1987. Also available as technical report CMU-CS-87-155.

4 A new-generation AI application

Many large applications — aerial photo interpretation, autonomous vehicle control, factory automation — require computer vision, which is computationally extremely demanding. To assure that such applications perform at appropriate speeds, the parallelism of the vision-processing tasks must be exploited at all stages.

The main stages in vision processing are:

- Low-level processing of the image (or pixel-level computation) for noise reduction, image enhancement, edge detection, pixel characterization (e.g., color), determining areas that contain pixels with similar characteristics
- Low- to intermediate-level processing to group pixels into primitive features (such as line segments, curves) and regions
- Intermediate-level processing of features, which includes
  - Computing geometric properties of primitive and complex features, such as a line's length, the curvature at each pixel on a curve, a region's area
  - Determining spatial relationships, which involves analyzing whether features are adjacent, parallel, perpendicular, or symmetric to or contained in other features
  - Aggregating (bottom-up) primitive features into more complex ones, for example, grouping lines into sets of parallels and pairs of parallel lines into squares, rectangles, or parallelograms
  - Merging adjacent regions into larger regions
- The highest processing level, called image understanding, consists of two major operations on objects within an image:
  - Matching (top-down) certain object features against a model stored in a database while imposing (in geometric searches) global constraints on the matched features. This operation is also referred to as "object recognition."
  - Determining an object's location and orientation, also known as "position estimation."

Low-level processing is computationally very demanding due to the large number of pixel data. However, since this data is uniformly distributed in the image space, the operations required for processing this data are very regular and can easily be parallelized. Under a preceding DARPA contract, we demonstrated that low-level processing can be substantially speeded by mapping the required operations to the Warp machine, an array multiprocessor developed at Carnegie Mellon.

Our effort under this contract focused on parallelizing feature processing (including geometric search), the most time-consuming of all vision processing procedures. Feature data, while much smaller in number than pixels, is not uniformly distributed in the image space, and numerous operations are required to determine the geometric properties of and relationships among features. For example, to determine the binary relationship of n

features requires evaluating O(n^2) feature pairs. Since a significant amount of feature processing is involved in high-level object recognition, parallelizing feature processing also speeds image understanding.

4.1 Initial work with simple feature data

We began by designing algorithms for grouping primitive features into more complex ones, such as simple polygons with parallel and orthogonal lines. This effort involved exploring structures for efficiently representing the feature data and speeding the geometric search on it. We also investigated extending the Apply language to handle intermediate-level operations. This language, which we had developed under a previous contract, had proven effective for parallelizing low-level operations in an architecture-independent manner.

We determined that primitive features are best represented by grid structures and that Hough spaces most suitably describe the binary relationships among this data. However, the Apply language could not be modified to parallelize global, intermediate-level operations, such as image warping, histograms, and Hough transforms. We therefore designed a new language that we called “Map.” In addition to hiding from the user the details of implementing feature-grouping operations on parallel computer architectures, this language provides facilities for

- Mapping operations over other data structures as well as images
- Describing a data structure’s region that is required in executing an operation
- Specifying storage requirements
- Producing intermediate results on different processors and combining these results later

While the Map language lacks certain capabilities and the algorithms we wrote in it were never implemented, its design was used for developing the “Adapt” language, a project under another contract.

Our search algorithms for recognizing simple feature data used only two geometric properties (orientation and length of line) as constraints for successively eliminating unlikely matches. Since they are highly specialized, these algorithms were easily mapped in parallel onto a systolic array. A simulated implementation with varying sizes of datasets distributed over a 72-processor Warp array indicated a 100- to 250-fold speedup over a sequential version executing on a VAX 8650.

4.2 Processing complex features

Next we investigated data structures for efficiently representing two-dimensional, “spatial,” objects with such geometric properties as area, perimeter, centroid, moments, and minimum bounding rectangle. Since objects are usually not uniformly distributed in the image space, we also analyzed access methods, relationship descriptions, and timing data, as well as various data partitioning and task allocation methods for manipulating them.

Our study revealed that feature sets are best represented in a hierarchical structure that is compact and regular, suitable for partitioning and reintegrating, provides fast spatial indexing, and preserves locality. The “quadtree” and its extension, the “octree,” fulfills these requirements and is widely used in computer vision, spatial databases, and related areas. However, little work had been done to justify the claim that algorithms based on these structures are suited for parallel and distributed implementations. Our research revealed that quadtree/octree nodes are indeed easily distributed over multiprocessors and integrated without significant communication overhead [Chien and Kanade 89], [Chien 89].

However, data/task partitioning and allocation procedures for grouping and recognizing complex features are usually irregular. They are difficult to parallelize and require the facilities of general-purpose processors. In 1987 neither tightly coupled multiprocessors nor loosely coupled multicomputers were capable of effectively handling dynamic data/task migration schemes: Tightly coupled parallel systems didn’t provide the appropriate communication mechanisms, and loosely coupled multiprocessors were limited by the Ethernet’s low bandwidth and high latency. However, Carnegie Mellon was developing the fiberoptic-based Nectar network system (described in Chapter 1) that promised to address the above-mentioned communication problems. This prospect encouraged us to develop intermediate-level vision processing algorithms for implementation on the new Nectar system.

As we began porting our new algorithms to the Nectar architecture, we realized a need for mechanisms and primitives that would make parallel vision programming easier. This insight led us to design the “Paradigm” environment.

4.3 An environment for parallel, distributed vision processing

The Paradigm environment (Figure 4-4) maps operations to the underlying architecture so that concurrency at both the task and subtask levels is maximized. The application programmer simply needs to concentrate on designing data/task partitioning strategies, while Paradigm handles transparently the complexities of communication and data/task allocation and scheduling.

The user’s task-partitioning scheme is executed by Paradigm’s central “controller” whose task-queue server allocates the partitioned tasks/subtasks to the distributed “workers” (one on each node) for execution. Each worker has its own communication server that uses the message-passing model.

The main purpose of data/task partitioning is to improve load balancing and thereby speed execution. Load balancing is achieved by distributing subtasks to all processors so that they are kept equally busy. However, if tasks are partitioned without considering their spatial relationships, the complexity of other operations, such as integrating partial results and establishing adjacency among nodes, will increase. Trying to avoid the overhead of

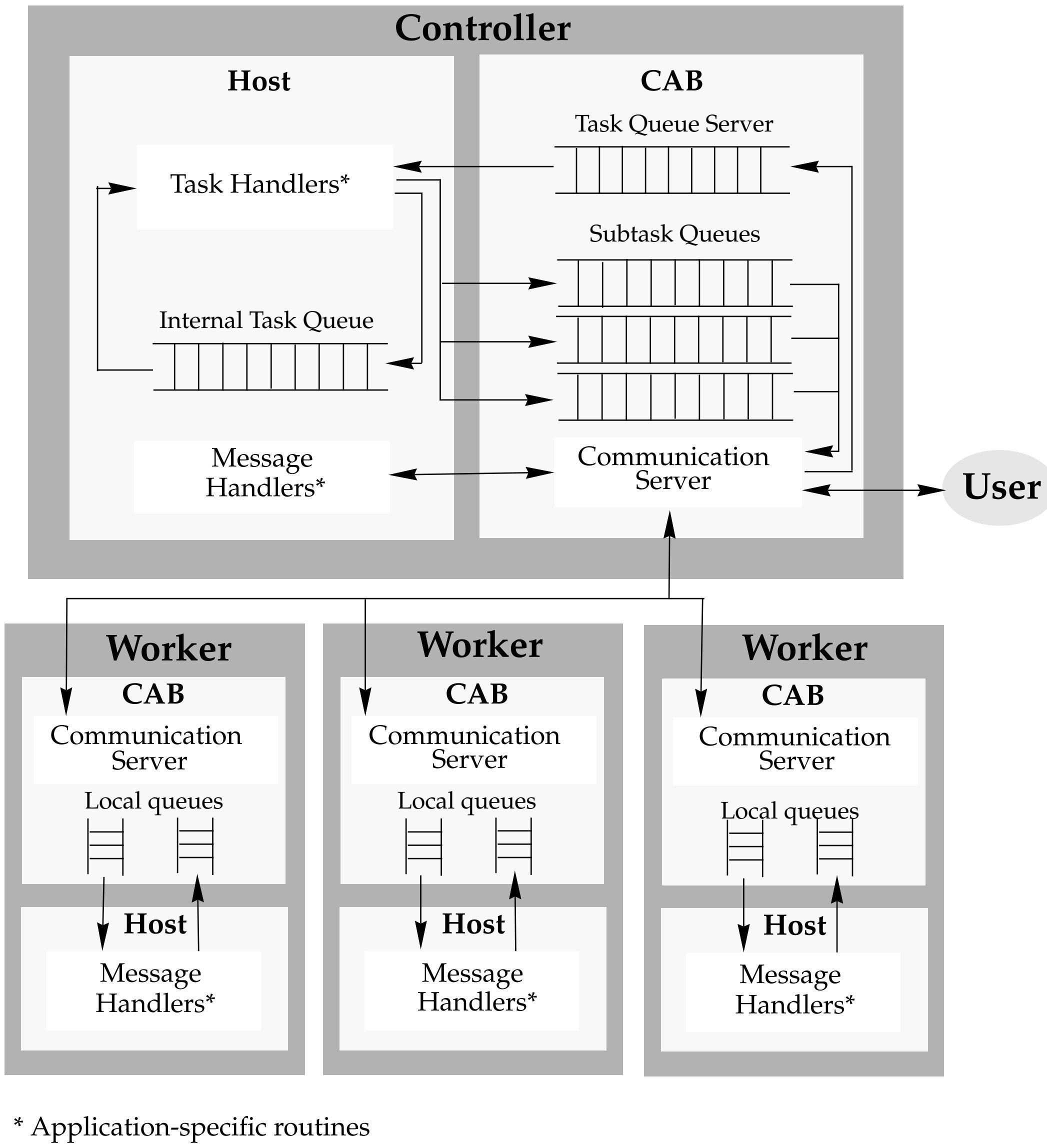

Figure 4-4: The Paradigm environment

future integration by simply combining spatially adjacent data items will cause further load imbalance when one group of adjacent subsets requires long processing times and the other extremely short ones.

We solved this dilemma by developing a multiple-task-queue mechanism, a variation of the single task queue. With multiple task queues the locality of spatial data can be preserved and interprocessor communication minimized. The task queues are based in the controller, and each worker is assigned one queue. Tasks and subtasks with spatially adjacent data are placed in the same task queue. Each worker fetches a task for execution from its assigned task queue, and only when its own queue is empty can the worker fetch tasks from other queues. This scheme maintains spatial adjacency while smoothing out variations in processing time.

4.4.1 A spatial database implementation

To test the effectiveness of our parallel processing environment we built a spatial database and implemented it on the Nectar system.Taking a set of spatial entities (polygons) as input, this subsystem can answer queries about their geometric properties, such as area, convex hulls, and overlapping regions. In addition, it can perform spatial operations, for example, checking for containment and adjacency. Some of the queries are simple, others time consuming, some relate to a single data item, others involve the entire data set. We also designed an algorithm that efficiently indexes the input polygons by representing them in quadtree structures. This quadtree-generating scheme is a preprocessing phase for all subsequent spatial operations.

When we implemented our spatial database the Nectar system consisted of only two nodes. These resources were too limited to allow analyzing the performance of the Paradigm environment. However, we were able to observe Paradigm's ability to handle several tasks (queries) concurrently and to dynamically partition, schedule, and allocate tasks and subtasks.

4.4.2 Processing 3-D object representations

Our next goal was to demonstrate Paradigm's ability to parallelize existing algorithms and map them to the Nectar architecture. We selected algorithms that rebuild the three-dimensional (3-D) structure of objects or environments from their two-dimensional (2-D) images.

Most 3-D representation and processing schemes require considerable amounts of memory and computation. The technique we adopted, referred to as *volume intersection*, facilitates describing the 3-D structure of an object from three or more different noncoplanar views (silhouettes). This approach provides tremendous savings in storage requirements since only the silhouettes of each object need to be stored in the model database. Storage is further reduced by using quadtrees for the silhouettes and an octree for the 3-D structure of the object. An additional benefit of this technique is the low processing time required for rebuilding the object's 3-D structure.

To provide information about the object's surface, we combined this volume-intersection algorithm with a *multilevel boundary search*. This procedure extracts and encodes information describing the interfaces between the object and its surrounding space.

We implemented these sequential algorithms in parallel by mapping Paradigm's controller and its three workers to the then available Nectar prototype that connected three nodes. As input images we used three orthogonal views of an aircraft, from which our parallel algorithms generated a quadtree for each view and a volume/surface octree as follows:

The image-processing task is divided into the following subtasks: data conversion, volume intersection, surface detection, and graphic display. Through data partitioning (input and output partitioning) each subtask is further divided into smaller subtasks. Thus, a quadtree from each view/image is generated by partitioning the "input" image into n×n sub-images (where n stands for the number of workers). Each sub-image is represented as a subquadtree. The resulting sub-quadtrees are then broadcast to the workers for the octree generation.

An octree of the object is generated by partitioning into subquadtrees each of the three “output” quadtrees representing a view of the object. Three corresponding subquadtrees are then “intersected” to generate n×n×n suboctrees that can be merged into an octree for a 3-D representation of the object. (This output-partitioning strategy is also employed by the graphic display task.)

Paradigm and the underlying Nectar architecture provide support for efficiently broadcasting the data to all processing nodes. No communication among workers is required for generating subquadtrees and suboctrees. However, throughout the multilevel boundary-search process workers need to exchange information. That this data exchange is carried out in real time is ensured by Nectar’s high-bandwidth and low-latency communication channel and by Paradigm’s communication servers.

Each worker creates at each node a partial view of the object. The controller combines these partial views into a complete, three-dimensional image of the object that can be observed in a graphic display.

This implementation demonstrated that the Paradigm environment is capable of parallelizing existing vision algorithms. Since all operations are simply seen as tasks, this environment handles any image-processing level equally well. In addition, Paradigm allows users to execute particular tasks on specific machines. For example, low-level image processing and limited types of feature processing can be implemented on a homogeneous multiprocessor, such as iWarp, while the fiber-optic Nectar network is best used for more general tasks in intermediate-level feature processing and high-level image understanding.

4.5 Bibliography

[Chien 90]

C.H. Chien. PARADIGM: An architecture for distributed vision processing. *Proceedings of the 10th International Conference on Pattern Recognition*, pages 648-653. IEEE, June, 1990.

[Chien 89]

C.H. Chien. Constructing Octrees from Multiple Views: A Parallel Implementation. *Proceedings of SPIE Conference on Sensor Fusion II: Human and Machine Strategies*, pages 274-285. International Society for Optical Engineering, November, 1989.

[Chien and Kanade 89]

C.H. Chien, T. Kanade. Distributed Quadtree Processing. *Symposium on the Design and Implementation of Large, Spatial Databases*. July, 1989.