(4)

DTIC FILE COPY

AD-A205 184

RADC-TR-88-203
Final Technical Report
October 1988

DEVELOPING SOFTWARE TO USE PARALLEL PROCESSING EFFECTIVELY

Dynamics Research Corporation

Sponsored by
Strategic Defense Initiative Office

Strategic Defense Initiative Office

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

89 3 06 143

RADC-TR-88-203 has been reviewed and is approved for publication.

APPROVED: [illegible]

ALAN N. WILLIAMS, 1Lt, USAF
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: James W. Hyde III

JAMES W. HYDE, III
Directorate of Plans & Programs

A205154

RADC-TR-88-203

Title: DEVELOPING SOFTWARE TO USE PARALLEL PROCESSING EFFECTIVELY

Dated: October 1988

UNCLASSIFIED
ERRATA
March 1989

Please delete last item on cover which reads:

> The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

and add the following item:

> The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Strategic Defense Initiative Office or the U.S. Government.

Also delete the following from inside front cover:

> If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTC) Hanscom AFB MA 01731. This will assist us in maintaining a current mailing list.

and add the following:

> If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTC) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Approved for public release; distribution unlimited.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force, Base, New York 13441

DEVELOPING SOFTWARE TO USE PARALLEL PROCESSING EFFECTIVELY

Julian Center

Contractor: Dynamics Research Corporation
Contract Number: F19628-86-D-0006
Effective Date of Contract: 29 Jun 87
Contract Expiration Date: 31 Dec 87
Short Title of Work: Developing Software To Use Parallel Processing Effectively

Period of Work Covered: Jun 87 - Dec 87

Principal Investigator: Julian Center
Phone: (617) 475-9090

Project Engineer: 1Lt Alan N. Williams
Phone: (315) 330-2925

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by 1Lt Alan N. Williams, RADC (COTC), Griffiss AFB NY 13441-5700, under Contract F19628-86-D-0006.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-203
6a. NAME OF PERFORMING ORGANIZATION Dynamics Research Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTC)
6c. ADDRESS (City, State, and ZIP Code) 60 Frontage Road Andover MA 01810	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Strategic Defense Initiative Office 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-86-D-0006
8c. ADDRESS (City, State, and ZIP Code) Office of the Secretary of Defense Wash DC 20301-7100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63223C PROJECT NO. B413 TASK NO. 03 WORK UNIT ACCESSION NO. 39

11. TITLE (Include Security Classification)
DEVELOPING SOFTWARE TO USE PARALLEL PROCESSING EFFECTIVELY

12. PERSONAL AUTHOR(S)
Julian Center

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Jun 87 TO Dec 87	October 1988	184

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Multiprocessing, Concurrent Processing, Processor Evaluation Tools, Parallel Debugging, FLUIDICS, FLUERICS, DS

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report discusses the difficulties involved in writing efficient parallel programs and describes the hardware and software support currently available for generating software that utilizes parallel processing effectively. → [illegible] 1-1

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Alan N. Williams, 1Lt, USAF	22b. TELEPHONE (Include Area Code) (315) 330-2925 22c. OFFICE SYMBOL RADC (COTC)

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

SECTION 1 - INTRODUCTION AND SUMMARY

1.1 OVERVIEW

Historically, the processing rate of single-processor computers has increased by one order of magnitude every five years. However, this pace is slowing since electronic circuitry is coming up against physical barriers. Unfortunately, the complexity of engineering and research problems continues to require ever more processing power (far in excess of the maximum estimated 3 Gflops achievable by single-processor computers). For this reason, parallel processing architectures are receiving considerable interest, since they offer high performance more cheaply than a single-processor supercomputer, such as the Cray. They are even more attractive now, since their processing rates will, theoretically, be able to reach the teraflop region. This has resulted in a proliferation of vendors offering a variety of parallel processing architectures.

In the near future, highly parallel machines - machines with hundreds or even thousands of processors cooperating on a single problem - will be commercially available. It has been demonstrated that these machines can provide enormous increases in computational power, but is this power usable? The answer depends on the extent to which programmers will be able to write code that effectively exploits hardware capabilities. Programmers must contend with the complexity of managing asynchronous interacting processes, the lack of experience needed to create appropriate programming abstractions, and the tremendous variations in highly parallel architectures. As a result, it is extremely difficult to write a parallel program, and it is even more difficult to write an efficient parallel program.

In this report, we discuss the software support available for writing performance efficient parallel programs. In the remainder of this section, we provide an overview of the difficulties involved in writing such programs.

Section 2 sets the stage for the discussion by describing the range of hardware architectures currently available. In Sections 3, 4, and 5 we survey the available programming support tools in each of three areas: program specification; parallel debugging; and performance evaluation and tuning. The appendices discuss related issues of object-oriented programming and multiprocessor operating systems.

1.2 DIFFICULTIES OF WRITING HIGHLY PARALLEL PROGRAMS

Parallel programming is difficult because of the complexity of parallel systems, our lack of experience and the large variations in parallel architectures.

The complexity of highly parallel programs affects both their specification and their debugging. Mechanisms for program specification can be divided into those in which parallelism is explicit and those in which it is implicit.

If it is explicit, the programmer is faced with the task of decomposing his algorithm into its parallel components, coding each as a sequential program, orchestrating their behavior with communication and synchronization operations, and controlling their mapping onto existing hardware. Currently, programming support for these activities is minimal.

Alternatives in which parallelism is implicit - based on automatic program restructuring [Kuck83, Allen87, Koebel87] or on novel languages and architectures [Nikhil87] - are being developed but may not be appropriate for many proposed architectures and applications.

After a highly parallel program is written, it still must be debugged. An executing parallel program generates potentially overwhelming amounts of relevant information that the programmer must absorb; worse, asynchronous systems do not have the consistent global states or reproducibility that have formed the basis for standard sequential break/examine debugging strategies.

In coping with the complexity of parallel programs, programmers do not have the advantage of a ready set of abstractions built up from years of collective experience. In the sequential domain, we have developed a number of such abstractions or programming paradigms. These paradigms - high-level methodologies common to many effective algorithms, such as *divide-and-conquer*, *dynamic programming*, or the greedy algorithm - provide a "tool kit" for the programmer, giving him a place to start and the advantages of accumulated experience with a particular strategy [Nelson87]. In the parallel domain, common techniques (such as generate and solve, iterative relaxation, passive data pool, systolic, or compute-aggregate [Nelson87, Finkel87]) are only beginning to emerge as paradigms.

More problematical than either complexity or lack of experience, however, is the tremendous variation in the capabilities of highly parallel architectures. It is easier to program in the sequential domain, in large part, because there is a consensus on an "idealized" von Neumann machine (having a small set of features such as random access memory and a program counter) [Segal]. This abstract machine serves to define the features available for language translation and the features that must be implemented in hardware. Because sequential machines are relatively uniform, the programmer can code for the idealized machine, confident that his program will be efficiently implemented and that his design choices will be adequately reflected in the behavior of his program. Programmers can write efficient code without knowing the low-level details of their target architecture, and they can easily port their code to other machines.

In the parallel domain, this is not possible because of the diversity of architectures and the corresponding variations in the models of computation that they support. Performance in the parallel domain is often dependent on the match of algorithm and architecture, forcing the programmer to consider architectural characteristics of the target machine very early in the design process. This reduces productivity and limits the portability of the resulting code. It is unlikely that a single idealized parallel machine will emerge but, in the future, as parallel architectures evolve and stabilize, it may be possible to obtain some of the benefits of the virtual machine approach by considering small sets of idealized machines [Segall87].

1.3 WRITING PERFORMANCE EFFICIENT, HIGHLY PARALLEL PROGRAMS

There are many possible sources of performance degradation in parallel systems. They can be divided roughly into losses manifested by busy processors and losses manifested by idle processors [Segall87]. Busy processor losses can result from communication overhead (due to the diversion of computational resources), redundant computations (on data that has been replicated to avoid the need for interprocess communication), useless computations (as found in search programs, for example, where multiple paths are pursued until the unsuccessful ones terminate) and "reduced knowledge" (as found in "chaotic" algorithms where processors are not always using the most up-to-date information). Idle processor losses include communication delays (when processors must wait for data), lack of computation to be performed (due to uneven allocation of work across processors), and inaccessible work (due to synchronization or locking).

Reducing program inefficiencies involves determining that a loss of efficiency exists, locating the source of that loss and then making the appropriate modifications [Segall87]. None of these tasks is easy. They imply both the ability to predict and to monitor performance. Prediction is needed in determining that a loss exists and in evaluating the effects of possible modifications; monitoring is needed in validating models, in assessing the extent of losses and in isolating their causes. New models of performance prediction (algebraic, stochastic, simulation) must be developed to take into account the complex interactions between parallel algorithms and hardware. Transparent techniques for program monitoring (transparent because they do not affect the course of the computation) must also be developed. "Debugging" parallel programs for performance will be even more difficult than debugging them for correctness.

1.4 SUMMARY

Parallel computing offers new capabilities to system developers, but it also presents new challenges. Researchers and developers are evolving new ways of thinking about program design and new language constructs to express parallelism. They are also dealing with the complex problems of debugging and performance tuning of programs that involve many loosely-coupled processors.

This field is still in its infancy, and its methods and tools are far from being standardized. This report attempts to survey the current state-of-the-art by reviewing some of the most promising new techniques.

SECTION 2 - PARALLEL PROCESSING ARCHITECTURES

2.1 OVERVIEW

The parallel processing architectures offered by vendors are usually designed with a specific range of applications in mind. One major problem is that computer users often base their buying decision on the specified processing speed without realizing that the architecture may be totally unsuited to their application. Users can find themselves in one of three situations: (1) the architecture is well suited to the application and existing algorithms, (2) the architecture is totally unsuited to the application, and (3) the architecture is suited to the application but the algorithms must be reworked to take advantage of the architecture. Clearly, the first situation is ideal and the second situation should be avoided. What is less clear, is what happens when users find themselves in the third situation. A knowledge of how to implement algorithms to take advantage of parallel architectures leaves the user with more options and will likely save time and money. What is described in this section are the different parallel processing architectures and the impact these architectures have on applications programmers and users.

No one architecture has been singled out as the way to structure parallel processors. True parallel architectures (i.e., those in which individual nodes work together to speed up the execution of a single program) can be categorized as either single-instruction, multiple data (SIMD) or multiple instruction, multiple data (MIMD) machines. SIMD machines simultaneously and synchronously execute the same instructions on different data streams. SIMD processors can achieve very high levels of performance, but they are unsuitable for many applications and are said to be difficult to program. For this reason, recent develop work has focused on MIMD architectures.

2.2 MULTIPLE-INSTRUCTION, MULTIPLE-DATA (MIMD) MACHINES

There are a variety of MIMD machines available, and these can be classified by (1) the number of processors in the system, (2) the organization of the memory (shared or local), (3) the interconnection structure (bus, switch or binary N-cube based), and (4) whether vector processing can be done. Perhaps the most distinguishing feature is the interconnection structure of the system. Three main techniques exist: (1) bus, (2) switch and (3) binary n-cube based interconnections schemes.

2.2.1 Bus-Based Machines

Bus-based machines usually incorporate the global shared-memory approach where processors access the central shared memory via the communications bus (see Figure 2-1). One disadvantage of this architecture is that the communications bandwidth of the bus places an upper limit on the maximum number of processors in the system. However, this can and has been remedied through the use of multiple buses (see Figure 2-2). Some bus-based machines include the FX Series from Alliant, Elxsi 1400, and Sequent Balance 2100 (see Figure 2-1).

Alliant's FX/8 parallel computer supports a combination of pipelined vector and scalar operations and can include up to 8 parallel processors in combination with 12 sequential processors in a bus-based shared-memory configuration [Bond87]. "Disk striping" on the FX/8 provides a transparent form of parallel disk I/O that distributes individual files across multiple storage devices with simultaneous access to multiple data elements.

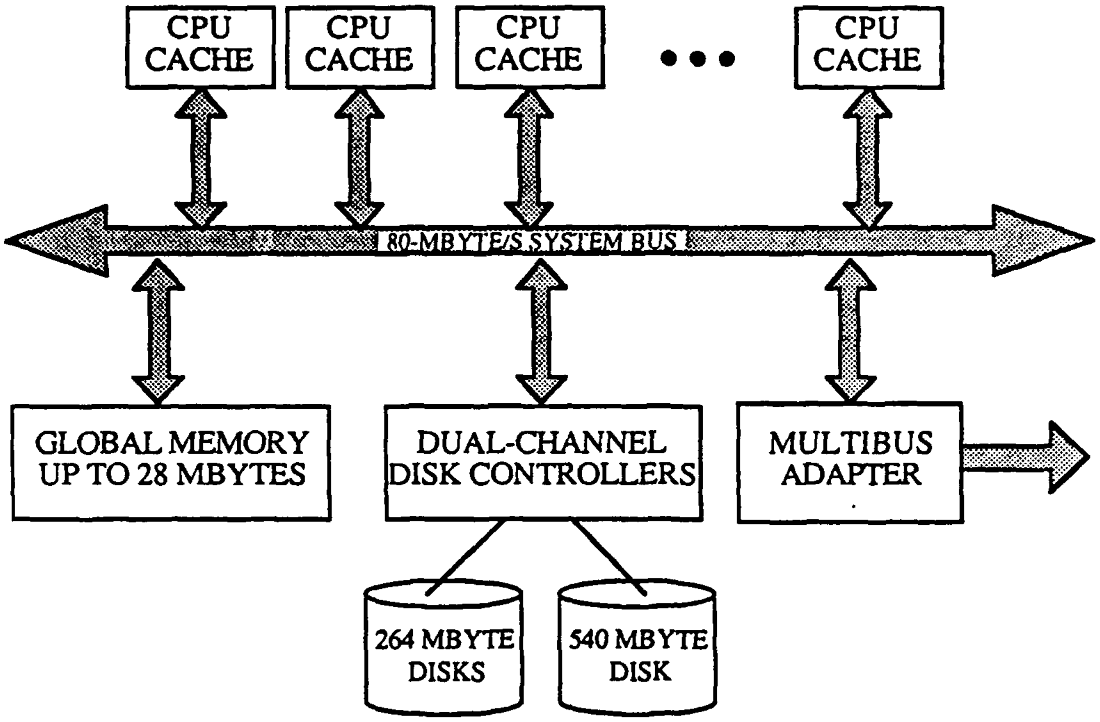


Figure 2-1. Sequent Parallel Processor

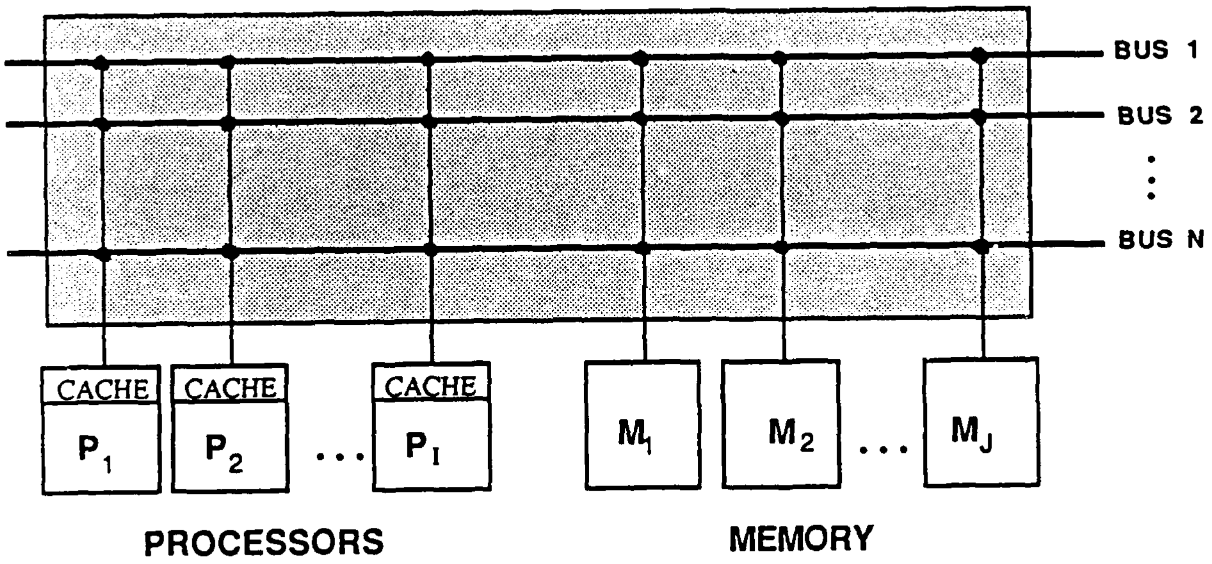


Figure 2-2. Multiple Bus Architecture

Each FX/8 parallel processor implements a 68020 instruction set in custom logic with a wider (80-bit) parser, local cache, and other hardware enhancements. In addition to basic vector and floating-point operations, each parallel processor contains a concurrency control unit (processor synchronization) and data path interface (interconnect switch between instruction processor, control section, main memory and cache). Less complex "interactive processors" offload serial processing, terminal handling, disk I/O and operating system functions.

Alliant is the only current vendor whose products can run standard Fortran programs in parallel without reprogramming. Data dependencies between loop iterations running on multiple processors are synchronized by special instructions and concurrency control hardware. However, other programming languages do not support any automatic parallelization. Parallel actions in C and Pascal are supported by libraries of vector and concurrent operations, with versions of Ada and Lisp under development.

The Alliant operating system is a version of BSD 4.2 Unix modified to support a multiprocessor environment (similar efforts have been in use at universities since 1983, cf [Barak85, Emrath85, Janssens86].

Sequent's Balance and Symmetry machines use standard microprocessors (NS32332 and 80386, respectively) in a bus-oriented symmetric architecture. Sequent's operating system supports its tightly-coupled shared-memory configuration with both BSD 4.2 and AT&T System V Unix functions. C, Fortran and Pascal compilers are supported with a parallel function library and a parallel debugger. Ada, Cobol, Lisp and Prolog are under development.

A handbook from Sequent Technical Publications [Osterhaug86] stresses that prior manual analysis is always required for effective use of parallel programming language extention and operating system function libraries. The assumed hardware configuration contains multiple processors in a tightly-coupled symmetric configuration on a common bus with shared memory, hardware support for mutual exclusion, and dynamic load balancing.

2.2.2 Switch-Based Machines

Switch-based systems consist of independent nodes (processor/ memory pairs) and are connected together through a switch. The simplest configuration is a crossbar switch (see Figure 2-3) which connects every element in the system; however, the hardware costs and the complexity of these switches increases considerably with an increase in the number of processing elements. More practical techniques are needed, such as that of the Butterfly switch from BBN Advanced Computers (see Figure 2-4), which is the only switch commercially available. Other switch-based systems are the Ultra Computer/NYU and the RP3/IBM.

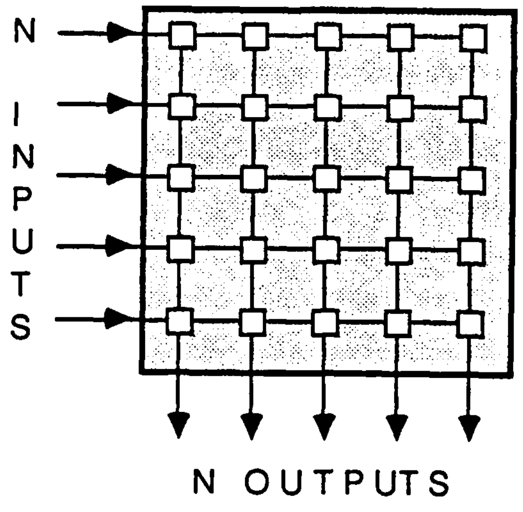


Figure 2-3. Crossbar Network

The Butterfly switch allows communications between the processor and memory elements, based on packet switching techniques. Each node in the Butterfly system consists of a 68020 microprocessor and a 6881 floating-point coprocessor with 1 megabyte of memory. Collectively, these memory elements form a shared memory which is accessible to every node. When nodes are added to the system, extra columns and rows of chips can be added to the switching system. Thus, for a linear increase in communications capacity, the switching complexity only increases slightly faster than linearly. At this time researchers are unsure whether the switch complexity will place an upper limit on the number of processors in the system.

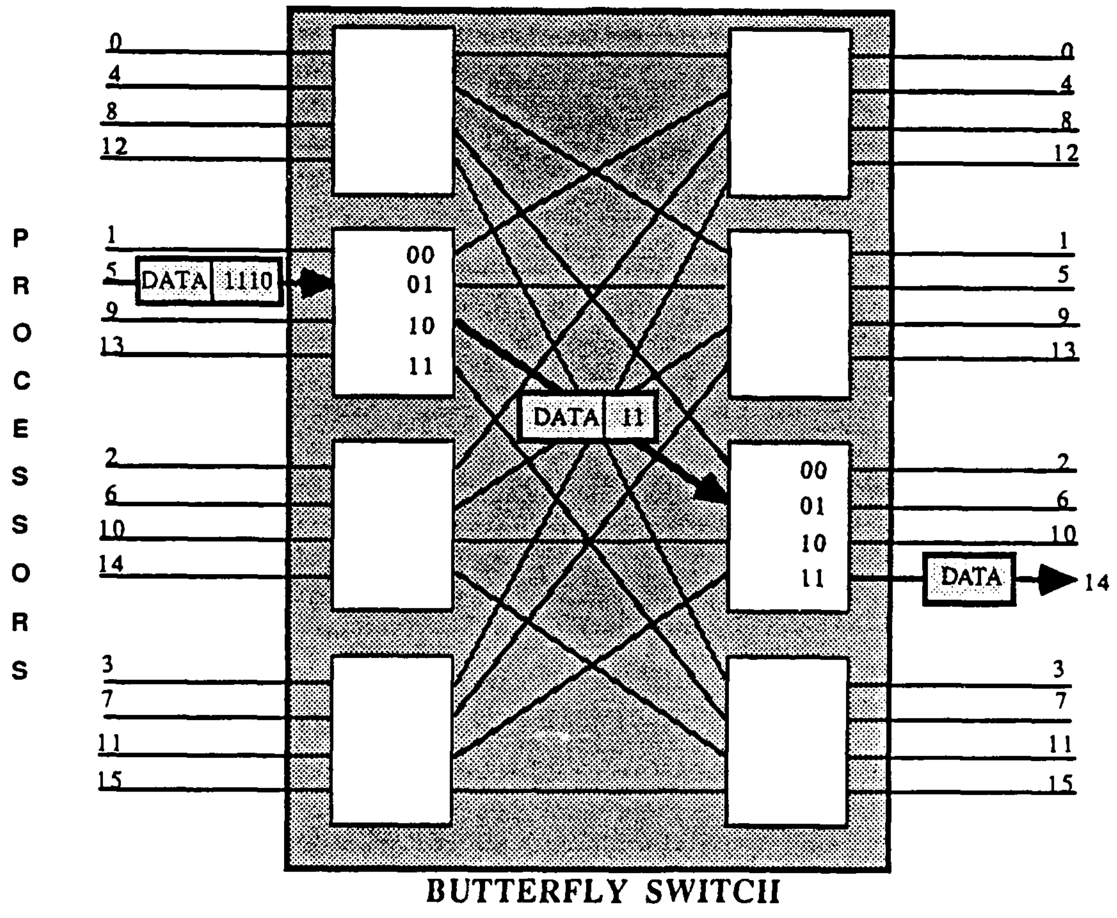


Figure 2-4. BBN Butterfly Switch

BBN's Butterfly uses techniques similar to packet switching. Switching nodes configured as a serial decision network provide redundant paths between each processor node pair. Although memory is local to individual nodes, it appears to the programmer that processors exist in a unified address space of shared memory [Bond87].

Butterfly interconnect network topology resembles dataflow in a fast Fourier transform (FFT), a recurring x-shaped pattern. As more switching modules are added, communications capacity grows nearly linearly, but growth of the number of wires being switched is only slightly faster than linear. I/O boards can be attached to any node for distributed access via the switching network.

Such an approach accommodates communications overhead without adding contraints of message passing programming style or extra latency (as occur with hypercube designs). Most programmers are accustomed to shared-memory environments and can cope easily.

BBN provides two separate software development environments for the Butterfly. Procedural languages are supported under the Unix-like Chrysalis operating system, which offers numerous C and Fortran function libraries. In addition to standard features such as tasking, memory management and interprocess communication, this environment includes functions that permit concurrent application of a single procedure to elements of a complex data structure distributed across multiple processors. Each task sees a common address space, and can be dynamically assigned to a specific processor node.

Programmers who prefer a functional or applicative language are supported with a complete multiprocessing CommonLisp environment. User interface to the shared-memory multiprocessor Lisp is provided by a front-end Symbolics 3600. Butterfly CommonLisp permits simultaneous evaluation of multiple Lisp expressions in a single, uniformly mapped shared-memory context. Portions of its design reflect concepts borrowed from Halstead's Multilisp [Halstead84].

2.2.3 Hypercubes

Another popular architecture is based on the binary n-cube, or hypercube, interconnection scheme. An n-th order cube having 2n nodes with each node connected to its n nearest neighbors. The hypercube interconnection scheme for eight nodes (an order 3 hypercube) is depicted in Figure 2-5. Hypercube architectures are the best choice for problems requiring a large number of parallel processors and are being extensively studied in academic institutions and government laboratories. Hypercubes can run multiple programs that operate on multiple sets of data. Each node in the hypercube has its own memory, floating-point hardware, communications processor, and copy of the operating system and applications program. It is the communications structure and independent nodal memory elements of the hypercube that allow designers to expand these computers far beyond most other parallel architectures.

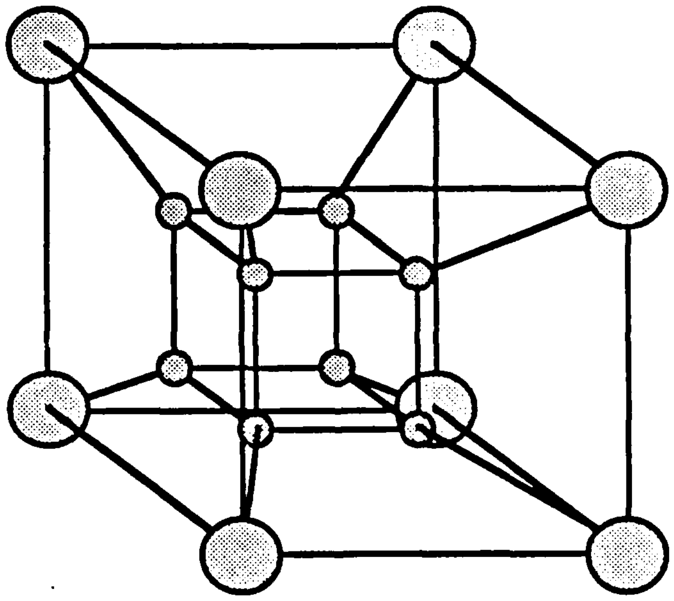


Figure 2-5. A Hypercube of order 4

Each node in a hypercube is independent and communicates directly with its nearest neighbors, and via message-passing through intermediary nodes to all other nodes. Because the communications is efficient between nearest neighbors, this architecture is well suited to scientific and engineering applications involving loosely connected localized phenomena.

There are some issues with regard to hypercubes that, if addressed, will increase their popularity even more. The issues mainly arise from the hypercube's inability to support a shared-memory architecture. For instance, non-neighbor communications rely on message-passing, which is currently supported by software. This routing software could be replaced by hardware, speeding up the communications considerably.

2.3 VERY-LONG-INSTRUCTION-WORD (VLIW) MACHINES

Explicit multiprocessing requires a close match between hardware capability and application program structure, usually obtainable only by manually reworking source code. However, the average user wants parallel speedup to apply to entire programs rather than parts of them, to be independent of application type and to be transparent to the programmer. Multi-flow computer is attempting to satisy these desires by using a combination of a Very-Long-Instruction-Word (VLIW) processor architecture and optimizing compiler technology. Although Multiflow's Trace computer [Bond87] contains only a single CPU, it permits substantial parallelism at the level of individual machine instructions (assembly language operations with from 32 to 1024 bits of parallel microcode and up to 28 scalar actions).

Each dedicated bit-field of the VLIW controls a data path or pipelined functional unit, which execute synchronously (in a way similar to a vector processor). Each instruction applies a variety of hardware functions to heterogeneous data rather than repeating a single action across a uniformly typed array. Thus each machine instruction accomplishes more work, regardless of the properties of higher-level language source code.

Earlier attempts at overlapped execution in VLIW processors were unable to optimize conditional branch or jump operations. When operations are overlapped with prior conditional jumps, the results of overlapped execution are frequently incorrect. Multiflow overcomes this difficulty through trace scheduling and compacting compiler technology. Conditional branches are optimized via trial execution and statistical analysis during compilation. Multiflow compilers (C, Fortran) profile a sample run of each application to estimate conditional branch direction and loop exit condition frequencies. Hardware resources and data precedence then constrain compaction of lengthy execution paths.

2.4 THE CONNECTION MACHINE

Thinking Machines' Connection Machine [Hillis85, Hillis87] blurs the distinction between SIMD and MIMD parallel processor architectures. Although multiple-data-element instructions are broadcast to multiple processors and are executed in parallel on an entire data set, the width of individual data elements is dynamic (between 1 and 4096 bits), and single-data-element instructions are executed on a front-end host processor.

Interconnection occurs either among four nearest neighbors via a grid network or between an arbitrary pair of processors using a hypercube router. The latter permits dynamic reconfiguration of communication paths.

Considerable attention has been given to provision of an appropriate set of data-parallel syntactic extensions for Connection Machine LISP [Steele86, Hillis86]. Nevertheless, naive software developers are likely to experience stylistic problems analogous to those encountered by users of Goodyear Aerospace's Massively Parallel Processor (a pure SIMD machine).

The Connection Machine's use of a huge number of processors, each having rather modest computational capability, necessarily encumbers the application programmer with issues of fine-grained data-level parallelism. Also, its hypercube topology places an absolute limit on interconnection capacity with fixed latency between two given processors (even when multiple paths are used).

One might argue that this is unimportant given the size of individual CM processors: no single processor can ever get enough work done to saturate its communication links. However, a pair of processors each having 2/3 capacity is not obviously equal to a single processor of unit capacity.

Although the Connection Machine can be configured to exhibit moderate asymmetry, its inherent uniformity and small grain size may not be well suited to real-time control applications.

2.5 THE APPLICATIONS PROGRAMMER'S VIEWPOINT

If an applications programmer is fortunate enough to have a say in the design of a (MIMD) parallel processing machine, there are several things to keep in mind. For instance, there is a trade-off involving architecture simplicity and system extensibility - the choice of the interconnection structure may have an effect on the maximum number of processors in the system (e.g., a single-bus architecture is usually limited to 20 or 30 processors). The organization of processor and memory elements will have an effect on the performance of the system, as will the locality of data and the ability to do vector processing.

If an applications programmer must go out and buy a ready-made machine, there are similar considerations. First, the computing power, the architecture, and the ability to accommodate a range of applications must be considered. It must be remembered that the performance specification alone is not an accurate indicator of how the particular machine will work since it is optimized for only a select range of applications. If an implementation for the algorithms has been chosen, then an architecture that suits this implementation must be found. A candidate architecture will be even more appealing if it can support a variety of problems that a programmer may need to solve in the future. If a suitable architecture cannot be found, then the algorithms must either be reworked, or suboptimal performance must be tolerated.

If a parallel processing machine has already been purchased, then a different set of issues arise. Of utmost importance is knowing how to map the problem onto the hardware to take advantage of the parallel processing. Software tools exist to help programmers describe the type of parallelism explicitly to the system (in the form of subroutine libraries and language extensions). For instance, the method of communications, which is dependent on the memory organization, requires the addition of several language extensions. For message-passing systems, "send" and "receive" primitives must be supported. For shared memory systems, synchronization support is needed.

Once a parallel machine is chosen, two approaches exist for programmers. One approach is to obtain a parallelizing compiler. The other approach is to program in a good parallel language.

There is a fair amount of research into the automatic parallelization of sequential code by a parallelizing compiler. Although this area of research is important, a compiler cannot rewrite an algorithm (e.g., to improve performance), and programmers who are encouraged to think in parallel will often find new ways to solve a problem. Although explicitly-parallel programming is complex, those who have programmed in parallel find that it is not fundamentally harder than conventional programming.

It is clear that much more research and experimentation is needed in parallel processing to resolve the issues and controversies. Multiprocessor vendors readily admit in conversation that no single partitioning method is best for all problems. To be effective, multiprocessing software design and implementation benefits enormously from human intervention, only slightly from automated analysis. Use of parallel algorithms with conscious attention to concurrency is a key element of success. Some automated speedup is possible, but it seems to work best for pure numerical problems, or when restricted to low-level operations (e.g., overlapping execution of processor instructions).

Substantial programmer training will be required for successful implementation of complex software systems on parallel architectures. Implementations designed for serial uniprocessor environments cannot be effectively transformed for multiprocessor execution without considerable algorithm change. Only moderate performance enhancement is possible without human intervention and without abandoning uniprocessor software design methods.

Unfortunately, few design tools are available. Yet even when using a parallelizing compiler (e.g., Alliant Fortran), "for the highest performance, it may be necessary to rewrite the original algorithm to make it more concurrent" [Bond87].

SECTION 3 - PROGRAM SPECIFICATION TOOLS AND TECHNIQUES

3.1 PARALLEL PROCESSING STRUCTURES

Parallel application design mostly involves decisions about multitasking of closely cooperating processes, as distinct from multiprogramming. Multiprogramming concerns simultaneous execution of multiple unrelated programs (from different users, without any inter-program synchronization, e.g., a multi-user operating system on a uniprocessor). Multitasking implies concurrent execution of several communicating functions (programs, tasks) relating to a common objective, perhaps via time-slicing rather than literal simultaneous execution. Parallel computation from this viewpoint is primarily a hardware extension of operating system features that have existed for decades.

Multitasking speedup depends on the percentage of program execution time that can be spent performing actions in parallel (rather than in sequence); the number of processors available; hardware contention for I/O channels, system memory, disk storage or other peripheral devices; and overhead for communication and synchronization of multiple application fragments. An important related issue is how evenly computational load can be balanced among parallel tasks (running on separate processors).

Nearly every application program contains inherent sequences of action that prohibit completely parallel execution. Despite this, large portions of the typical application can be structured to suit multiple processor environments. A multitasking design naturally encounters two ways that a program can be decomposed: data partitioning and task partitioning.

Data partitioning creates multiple identical processes and assigns part of a large data set to each one. This works best when the same operations are performed repeatedly on uniform collections of data (e.g., matrix multiplication, Fourier transforms, ray tracing or signal processing).

Task partitioning (or function partitioning) expects that an application performs many different operations on a small data set. The program can be separated into numerous distinct tasks. Task partitioning produces designs that are more difficult to balance among processors, and which require moderate to substantial programming effort to implement.

In either case, suitable partitioning of an existing serial application requires profiling (cf [Maples85, Gregoretti86]), which produces a subprogram-call frequency distribution and dependency graph. Task partitioning focuses immediately on the subprograms that consume most of the program's execution time. Data partitioning searches through the subprogram call hierarchy to find ancestors that contain loops. An "independent loop" passes no information between iterations other than the value of its loop index, and can be subjected to data partitioning for parallel execution.

Multitasking operation involves issues of data protection, scheduling, sequential dependency and task synchronization even on a uniprocessor (with a global clock). [Osterhaug86] presents a concise summary of relevant concepts as well as problems which arise in an application (e.g., concurrent update of a file by multiple tasks, and processor waiting or task suspension during I/O).

Published discussions of parallel processing often refer to decomposition of a highly iterative, data intensive computation as "partitioning." A large number of novel programming language constructs have been developed to allow programmers to describe how data-parallel operations should be partitioned [Karp83]. Much less attention has been given to automating the necessary prior analysis, or to defining a parsimonious set of syntactic extensions. Some authors use the term "parallel" only to mean "data-parallel." Task-parallelism is also known as "multiprocessing," a topic nearly as old as computer science.

Task-parallel decomposition has to divide a computation into semi-autonomous pieces (partitioning), schedule each task for execution by one or more processors (scheduling), and coordinate flow of control and data during execution (synchronization) [Gajski85]. Task decomposition obtains only minor benefit from data-parallel optimizing compilers and language syntax and is often constrained by operating system and hardware architecture.

3.2 METHODS FOR EXTRACTING TASK-PARALLELISM

No single decomposition technique is appropriate for every parallel algorithm; there is no "one size fits all" method of problem partitioning for parallel computation. Three major approaches exist for mapping tasks among multiple processors [Shen, Lee87]:

- Graph-theoretic: task precedence and interprocessor communication paths are represented as graphs, and some form of minimal-cut algorithm is applied.

- Mathematical programming: task assignment is viewed as an optimization problem, and a suitable objective function is formulated.

- Heuristic: suboptimal but fast "rules of thumb" suitable for real-time computation are used.

Published strategies make use of all three approaches, and can benefit from repetitive refinement (as successive analyses clarify relationships among problem components).

Quasi-optimal assignment of tasks to processors is a scheduling problem that can be reduced to mapping a "problem graph," which relates the tasks, onto a "system graph" which links processing elements [Shen85, Berger87, Lee87]. The system graph is undirected, but the problem graph is normally a digraph (computation proceeds through definite stages and working sets of resources). Graph-theoretic reference papers describe this correspondence among problem- and system- nodes and edges as a "homomorphism."

The scheduling problem is frequently constrained by how many processors are available, the connection topology and communication bandwidth available among processors, or the maximum time allowable to complete a computation. Expression of such constraints in an objective function (O.F.) leads naturally to methods of mathematical programming.

Heuristics become useful when a solution must be obtained in real-time and the O.F. so constrains the problem that an optimum cannot be found quickly enough. Use of heuristics can usually be restricted to one or a few subtasks within a larger framework.

For algorithms which partition easily onto systolic arrays, multiprocessor configurations promote "vertical" partitioning of operations along parallel control paths with a minimum of interprocessor communication. Array processors require "horizontal" partitioning into quasi-synchronous vectors with substantial data transfer between successive actions.

The major problem in multiprocessor systems is saturation of communication channels when excessive amounts of data must be transferred between tasks running on different processors. Lack of channel capacity degrades computation throughput.

A method commonly used for judging how well tasks have been mapped to processors adds an interprocessor communication cost to a processing cost to obtain weights for an O.F. Shen and Tsai [Shen85] use a minimax approach that assigns tasks so that the longest running task completes in as short a time as possible. Their assignment model is a graph oriented mapping of the kind briefly described earlier.

Shen and Tsai restrict the solution range with a reasonable, but non-optimal initial partitioning, then use the O.F. to perform state-space search for a (local) optimum via the A* algorithm [Winston]. A* is an improved version of branch-and-bound search that combines dynamic programming with an estimate of distance that remains on an optimal path. Heuristic information (e.g., a lower-bound estimate of minimum cost) enters as part of a function used to order candidate nodes for processing by the A* algorithm. The graph matching to be obtained is a weak homomorphism: if an edge exists between two task nodes in the problem graph, an edge must also connect corresponding processor nodes (to which the tasks are assigned) in the system graph. The resulting algorithm is O(N**2) estimated via the number of node expansions generated within A*, but Shen and Tsai observe that graph matching problems are exponential in the worst case.

Communication overhead is also an important consideration in partitioning of a straight numerical computation for execution on multiple processors. Adaptive mesh refinement is a technique of finite element analysis which partitions a problem domain into rectangular grid patches of varying dimensions. [Berger87] examines the communication costs of mapping a mesh obtained via binary decomposition onto three different multiprocessor connection geometries: a mesh-connected array, a binary tree, and a hypercube.

Interprocessor communication for mesh arrays and hypercubes can be viewed as occurring in two phases, which correspond to movement along axes of a cartesian grid. Communication overhead implied by mapping a problem to a particular interprocessor connection topology can then be characterized using quantities of

- Skewness: the ratio of maximum patch edge length to the dimension of a patch in a uniform partitioning (which divides a domain into grid patches having identical edge lengths).

and

- Dilation: maximum number of edges through which a datum must pass during communication,

considered relative to a partitioning's

- Depth: number of times a domain has been subdivided via bisection of a region.

Overlapping "chains" (linear arrays of processors all of which receive from or send to an adjacent processor in a given step) of data transfer give rise to "congestion" (the number of chains contending for communication within a row or column). [Berger87] assumes that a processor can send or receive only one item on one communications link at a time. Two possible strategies for managing congestion are considered:

- Permutation: data values are transferred one at a time between communicating processors, with contention at each separate transfer.

- Pipelining: each processor transmits data to all others that require its result as a continuous action, blocking communication access during the entire interval.

The total communication cost varies according to the strategy adopted, and either strategy may be preferable depending on the skewness and depth of a given problem partitioning.

[Berger87] also considers multiprocessors connected in a binary tree configuration in which only leaf nodes perform computation, while intermediate nodes perform communication. (These usually suffer from traffic bottlenecks at the root, but fit naturally with the given partitioning method.) For a binary problem decomposition, performance of a binary tree configuration approaches that of a nearest-neighbor mesh to within a linear factor (of the depth of partitioning), given a sufficient number (2**depth) of processors.

Mesh performance nearly equals that of a hypercube for a low depth of partitioning (over a wide range of skewness). Hypercube performance exceeds that of a mesh for high depth of partitioning and moderate skewness, but is not as clearly superior for very low and very high skewness. Very high skewness yields a cost of communication which dominates both the square root communication latency of a mesh and the logarithmic latency of a hypercube.

[Lee87] presents an efficient strategy for mapping problem graphs onto an arbitrary multiprocessor system graph using one of a set of objective functions and an accurate characterization of the communication overhead. An example from image processing is used to motivate discussion and verify the algorithm's operation. Two of the O.F.'s are claimed to be especially suitable for real-time parallel processing.

The assignment model used is again a graph oriented mapping that uses four square matrix data structures: a problem matrix (represents problem graph), a nominal distance matrix (the system graph), an assignment matrix (maps problem to system), and a communication overhead matrix (weights problem edges according to their volume of traffic).

The O.F.'s presented are:

OF1: sum of communication overheads of all problem edges, when no two edges need to communicate during the same time interval

OF2: largest individual overhead among all edges, when all problem edges must communicate simultaneously

OF3: sum of maximum overheads selected from a sequence of working sets, when edges in a working set are needed simultaneously and sets are encountered sequentially

OF4: largest individual edge communication overhead selected from among the maxima of working sets in a sequence (this differs from OF2 because of reduced contention assumed among edges due to working set segregration)

The authors provide detailed procedures for calculating communication overhead for either synchronous or asynchronous links. No assumption is made about the shape of a communication path, but it is expected that a path is described as a sequence of edges or nodes. Treatment of dynamically configurable topologies (where routing rules could be changed during successive steps) is left for future research. The example application is mapped to a hypercube topology, for which a shortest path can be determined without search when source and destination nodes are specified.

Optimization of the problem to system mapping is possible at two levels: either select a good initial assignment or pursue incremental improvement of mapping. The most efficient method is to do both. The optimization method presented in [Lee87] exchanges selected pairs of problem nodes.

3.3 PROGRAM SPECIFICATION WITH EXPLICIT PARALLELISM

To specify parallelism explicitly, the programmer must decompose his algorithm into its parallel components:

- code the components as sequential programs
- orchestrate their behavior with communication and synchronization operations
- control their mapping onto the target hardware.

This is a lot to expect from a programmer. At least for the near future, however, it will be the most efficient method of exploiting the capabilities of many MIMD architectures, particularly MIMD nonshared memory architectures.

Programming environments supporting explicit specification of parallelism should provide:

- Scalability. Because parallel programs are often developed "in the small" and then scaled for massive parallelism, it must be possible to specify a "family" of parallel programs parameterized by size.

- Automatic distribution and specialization of code segments. Often the processes of a highly parallel program are nearly homogeneous and therefore, an environment should provide convenient mechanisms for the specification, distribution and specialization of common code.

- Explicit description of process interconnection structures. Explicit descriptions provide a natural medium for understanding parallelism, a basis for graphical displays, structural information potentially useful in mapping and redundancy for automatic error detection.

- Graphics interface. Programmers will need sophisticated graphics to cope with the potentially overwhelming amount of information present in parallel programs.

- Automatic mapping of logical process structures onto architectures. Mapping is an extremely difficult problem that requires detailed knowledge of the target machine and, as a result, is beyond the capabilities of many programmers. Environments should provide tools to assist in mappings of arbitrary process interconnections to target architectures.

Recently, tools have been developed to provide some of the necessary support, and we describe a few representative examples. In each case, the tools were developed for nonshared memory architectures, and thus they share a model of computation in which disjoint processes communicate through asynchronous messages. The models differ, however, in their view of process structures which may be either static or dynamic (depending on whether or not they are known before runtime) and may be either single phase or multiphase (depending on whether or not they are changed between stages of the computation).

3.3.1 The Poker Parallel Programming Environment and Related Tools

Poker [Snyder84], the first comprehensive parallel programming environment, is based on a static, multiphased model of computation. It provides unified facilities for specifying parallel programs: sequential code segments, processor assignments, interconnections, and the distribution of external data streams are all described with a consistent graphics interface. Originally designed for the CHiP family of architectures [Snyder82], versions of Poker have been developed for the hypercube and for systolic architectures.

A Poker program is a relational database. Five different specification modes are provided, each corresponding to a different database view. Typically a programmer begins with a description of his interprocess communication structure, which he provides by manually drawing the necessary connections on a grid of processors. He then writes parameterized code for each of his process types, using a standard sequential language with extensions for interprocess I/O directed to logical ports. Two additional views - one for assigning processes to processors and the other for defining logical port names - are used to connect the code with the communication structure. Finally, external I/O is specified by describing the way in which file records are to be partitioned into data streams on input and the way in which data streams are to be composed into files on output.

Poker has many of the desirable features listed above - (limited) automatic distribution and specialization of code, explicit descriptions of interprocess communication structures and a graphics-oriented user interface - but, because the user must manually embed his logical interconnection structure into a processor lattice, it fails to provide full support for scalability and it does not provide any support for mapping. The Prep-P Mapping Preprocessor [Berman87] (not part of Poker) addresses these problems.

Prep-P is a preprocessor that automatically maps arbitrary, bounded-degree logical-interconnection structures onto target CHiP machines, multiplexing code for processes assigned to a single processor. Prep-P first contracts the interconnection graph to the appropriate size for the target machine, partitioning the process set so that each partition executes on a different processor. It then places the partitions of processes onto a grid of processors and routes the necessary communication channels between them (currently routing is performed for the CHiP machine).

3.3.2 <u>PISCES</u>

While the original aim of Poker was to provide access to a specific architecture, the aim of PISCES ("Parallel Implementation of Scientific Computing Environment") [Pratt87] was to provide a stable environment for scientific and engineering applications insensitive to changes in the underlying architectures. It provides the FORTRAN programmer with a virtual machine that can be efficiently implemented on a variety of MIMD architectures. (PISCES 2 has been implemented on a Flex/20 and PISCES 3 is planned for a hypercube.)

As in Poker, PISCES programs are tasks that communicate through asynchronous messages; unlike Poker, PISCES uses a dynamic, but single-phased model of computation. PISCES is less graphical than Poker but it contains a number of novel features not found in the earlier environment. PISCES permits the user to take advantage of multiple grain sizes; tasks, subprograms, code segments (for example, loop bodies) and arithmetic operations (vector operations) can all run in parallel. Code segment parallelism is implemented with "forces" [Jordan87], which are groups of tasks that run the same code and communicate through shared variables. The number of processors executing a force is determined by runtime availability. PISCES also makes use of "windows" [Mehrotra82] to represent the partitioning of an array that is to be simultaneously accessed by a number of processes. In addition, PISCES promotes efficient implementations by allowing the user to "see through" the virtual machine and control its mapping onto the hardware. To do this the programmer defines a configuration, specifying the number of process clusters to use, the primary processor for each cluster, the secondary processors for each cluster and the extent of multiprogramming.

3.3.3 MUPPET

The goal of MUPPET (a multiprocessor programming environment) [Muehlenbein86], like PISCES, is to provide support for the fast development of portable parallel programs for scientific applications. Unlike PISCES, it is viewed not simply as a programming environment but as a "problem solving environment" which includes concurrent languages (needed to drive architectures efficiently), a programming environment (consisting of language-based editors, an interpreter and a simulation system), applications environments (with applications-oriented languages and expert systems) and a graphical man-machine interface.

MUPPET is based on a dynamic, multiphased model of computation, referred to as the Local Memory Abstract Machine (LAM) model. The LAM model does not restrict communication, but it can be refined into more restricted versions, such as the Ring LAM or Tree LAM, which are then mapped onto the hardware. Three different approaches to automatic mappings have been considered: topological (in which an attempt is made to preserve processor adjacencies); mathematical (in which the problem is treated as an optimization problem); and adaptive (in which the system is allowed to dynamically adjust to its loads) [Kramer87].

3.3.4 Polylith

The difficulty of implementing a single parallel program on a single machine often precludes the possibility of experimenting with a variety of implementations across architectures. Polylith [Purtilo87] was designed to support such experimentation by providing for the fast prototyping of architecture-independent (portable) parallel programs. It uses a dynamic, single phase model of computation. Programs are separated into "specifications" and "implementations." A specification describes a process structure as a set of clusters that provide services through visible interfaces and a set of interface bindings that establish channels of communication. The implementation of a specification consists of code segments (functions, procedures, subroutines, etc.) that can be written in a variety of common languages. A message handler is provided to realize communication, making it possible to execute specifications even before their implementations are complete. Thus the Polylith programmer has a great deal of flexibility in experimenting with alternate program designs, and his programs can be ported to any target architecture with an implementation of the message handler.

3.3.5 Sequent Fortran

Analysis of a loop to determine its suitability for parallel execution requires identification of the types of variables used. A "shared" variable is either constant within the loop, or an array each element of which is accessed by a single iteration. "Local" variables are initialized in each iteration prior to use.

A "reduction" variable is an array or scalar that is used in only one associative or commutative operation within the loop (i.e., multiplication/division, addition/subtraction, AND, OR, XOR) and is a self-assignment (e.g., X = X + (C1 * C2)).

The value of a "shared ordered" variable depends on exact sequencing of loop iterations. If the iterations were executed in random order, it would not end up with the correct contents. A "shared locked" variable is a shared ordered variable which in addition is read and written by more than one loop iteration.

Sequent's Fortran compiler supports directives which permit the user to mark loops and variables so that execution of loops which contain shared, local or reduction variables are performed in arbitrary order on multiple processors. Iterations of a loop that has shared ordered variables must be arranged to occur in a prescribed order. A shared locked variable imposes an additional constraint: access by only one loop iteration at a time (mutual exclusion among iterations with respect to the variable).

A requirement that Fortran programmers explicitly recognize and mark data dependencies seems preferable to the sort of "blind" parallelization performed by Alliant's compiler. NASA once lost a Mariner probe to Venus due to a Fortran compiler's interpretation of a programmer's punctuation error [Annals84].

Similar considerations enter into design of multiple independent tasks to execute "loop iterations" in parallel, which [Osterhaug86] refers to as "microtasking". The tasks share some data and make private copies of the rest. Data flow and synchronization are controlled using special-purpose functions provided in Sequent's compiler libraries. The kinds of actions permitted include forking of multiple parallel computations; identification, census, suspension and termination of subordinate computations; locking and unlocking of shared resources; forced serial execution of marked sections of code; and microtask synchronization.

Methods supported for functional decomposition of tasks are fork/join and pipelining. Fork-join works best where no major function requires results from any other. Pipelining is easier when major functions depend on each others' results and the data sets involved are very large.

A fork-join application first assigns each of a set of tasks to access one element of a shared data set (the fork). Data that can be simultaneously accessed by two or more tasks must be protected by a lock. After doing its work, each subtask waits until all the others have finished (the join).

Pipeline applications also assign tasks to access individual data elements, but promote overlapping rather than simultaneous execution. The first task in line does its work, then writes its results to shared memory, tells the next task in line that the results are available, and fetches fresh input. Each successive task reads the results of its predecessor(s) when it is ready to, and passes its own results along to those which follow it. When work runs out, each task terminates after writing its result.

Many applications require use of both approaches together.

3.4 PROGRAM SPECIFICATION WITH IMPLICIT PARALLELISM

Implicit parallelism is either detected automatically by a restructuring compiler or it is inherent in the programming language.

3.4.1 Automatic Programming Restructuring

Automatic restructuring of sequential programs for parallel execution has been used primarily for scientific and numerical applications on shared memory architectures. It allows the programmer to work in the familiar sequential domain, avoiding problems of parallel program specification and parallel debugging. In addition, it results in portable code. However, effective data-level parallelism in non-numeric applications does not seem likely to emerge from automatic restructuring of serial algorithms. Very good results have been obtained for numerical algorithms [Padua80], but non-numerical algorithms appear to be less tractible [Lee85].

A program can be viewed as a set of statements constrained by data and control dependencies. Data dependencies arise from access to common values, and control dependencies arise from conditional branching. Dependency analysis has been used on conventional compilers to insure that optimizing transformations (such as code motion or dead code elimination) are legitimate. The output of such an analysis is usually a graph in which the nodes represent statements and the directed edges between them represent dependencies. Program restructurers use the partial orderings imposed by these graphs to schedule statements concurrently. Most restructurers focus on the medium grain parallelism that exists between iterations of a loop because it offers the greatest source of performance improvement (due to the high degree of parallelism and the relatively even load balancing). Loop iterations that are independent can have completely parallel executions using the "forall" construct; those with dependencies can often have staggered executions using the "doacross" construct.

The Parafrase Fortran preprocessor performs static analysis of source code, yet is not capable of transforming WHILE loops into Fortran DO loops. This task was accomplished by hand in [Lee85]. For a reasonable sample of 15 non-numerical algorithms, automatic restructuring resulted in good performance for 4, and acceptable (but not good) for 2 during simulation of a medium-grain multiprocessor configuration (32 computing elements).

Several categories of loops were uncovered that could not be parallelized at compile time (Table II and Figure 3 of paper [Lee85]). Performance improvement in such cases required use of a different algorithm (designed for inherently parallel instead of sequential or linear operation).

Ominously for any workers still pursuing AI research using Fortran, the Parafrase preprocessor could not recognize the sort of parallelism typically involved in linked list manipulation. However, [Hillis86] and [Halstead85] describe several algorithms for a parallel LISP which executes in a multiprocessor environment.

Many features from the Parafrase preprocessor are built into Alliant's Fortran compiler. (Indeed, the Cedar supercomputer under construction at the University of Illinois consists of multiple clusters of Alliant FX/8 machines) [Kuck85]. The FX compiler optimizes three types of loop operations for vectorization and concurrency [Alliant87]: a) DO-loops in vector-concurrent mode, e.g., iterative array updates; b) Nested DO-loops and multidimensional array operations; and c) DO WHILE loops, in scalar-concurrent mode (i.e., elimination of redundant expressions and invariant code).

Inclusion of statement and intrinsic functions does not inhibit parallel optimization, however when an external procedure is referenced in a loop, the FX compiler cannot make the loop concurrent unless explicitly instructed to do so via a source code directive or command line option. Only restricted types of Fortran statements are allowed optimized loops.

Vectorization is possible when syntax use is restricted to data assignment, COMMENT and CONTINUE statements, forward GOTOs and arithmetic IF branches to labels within the loop, atomic IF/ENDIF blocks nested to not more than three levels, and logical IF statements. Concurrent execution is possible wherever vectorization is, and also in the presence of non-blocked ELSE IF statements, block IFs nested more than three levels deep, GOTO branches to labels outside the loop, RETURN and STOP.

Much of the work on program restructuring grew out of work on vectorizing compilers developed at the University of Illinois, where these techniques are now being generalized for MIMD machines as part of the Cedar project [Kuck83]. The Cedar restructurer is a source to source translator, converting sequential FORTRAN programs into parallel FORTRAN programs that make use of both "forall" and "doacross" parallelism. (Although FORTRAN is used, the transformations would be applicable to many high-level languages.) It is interactive, prompting the user for assertions about his program that would remove suspected dependencies. User assertions are checked at runtime. Invalid assertions may result in correct, but non-optimal executions or in program termination.

PTOOL [Allen87] is also a semi-automatic restructurer aimed at exploiting loop level parallelism; it was designed for use with the Denelcor HEP and the Cray X-MP. PTOOL users identify regions that are potential candidates for parallelism (reducing the considerable expense of considering all possibilities). These regions are then analyzed for potential parallelism. Regions without interdependencies are scheduled as "forall" loops; regions with interdependencies are reported to the programmer using diagnostic displays of potential conflicts.

The BLAZE compiler [Koebel87] provides a source-to-source translation of programs from BLAZE into E-BLAZE with semi-automatic domain decomposition. (Domain decomposition refers to the process of distributing data on a parallel architecture). BLAZE is a high-level language intended for scientific applications across a variety of machines. It contains a single parallel construct, the "forall" loop. E-BLAZE is a superset of BLAZE that provides a virtual target architecture in the form of low-level tasking operations, such as processor allocation, interprocess I/O and synchronization primitives.

The pattern of data distribution is critical to performance even in shared memory machines because of the disparity between local and global memory access time and the costs of memory contention. The BLAZE compiler performs domain decomposition based on general patterns of distribution supplied by the user. Thus, for example, the user might specify that a two-dimensional array be distributed across processors by rows or by columns or by blocks. The compiler distributes the data based on this advice and then performs subscript analysis (currently limited to subscript expressions that are linear functions of at most one loop index) to determine the locality of memory references needed for code generation.

3.4.2 Novel Parallel Languages and Architectures

Restructuring compilers represent a somewhat roundabout approach: first the user is forced to overspecify his program by giving a total ordering of statements, and then the compiler is expected to determine which of those orderings are significant. A more straightforward approach would be to allow the programmer to write in a language that does not force him to specify any sequencing, relying only on that imposed by data dependencies. Dataflow machines and their associated languages are good examples of this approach. We discuss the Id language being developed at MIT for the Tagged Token Dataflow Machine [Nikhil87].

Id is a functional language (extended to include a form of data structuring mechanism called I-structures) that was developed with three objectives [Nikhil87]: (1) to insulate the programmer from the details of his machines, including such details as the number of processors available and the topology of their interconnections; (2) to free the programmer from the need to specify parallelism explicitly; and (3) to provide for determinate computations, eliminating the need for scheduling and synchronization.

Id programs are trivially translated into dataflow graphs in which nodes represent operations (instructions) and arcs carry values, called tokens. The execution of an operation in a dataflow graph is limited only by the availability of tokens. Any operation that has at least one token on each input arc can fire. As it executes, it removes a token from each input arc and deposits a result token on each output arc. These graphs can be executed directly by dataflow machines. There are a number of such architectures under development [Nikhil7, Gurd85], and they offer promising alternatives to more conventional approaches.

Although few or no researchers have developed methods for automatic decomposition of a design into parallel components, at least one team has worked on automated support for such problem decomposition. The authors of [Kerner86] have combined concepts from SADT and Petri-Nets to produce an automated executable parallel design language, in which description of an algorithm is strictly separated from its implementation details.

EDDA is a dataflow design language with graphical notation similar to SADT and a semantic structure resembling Petri-Nets. Unlike Petri-Nets, EDDA can represent recursion. An EDDA design contains predefined functional elements and obeys strict semantic rules, and can therefore be compiled into executable code.

Programs are built in EDDA by combining graphical objects, called "blocks." A graphical editor checks consistency of data paths between constituent blocks of a diagram, and a database system takes care of storage administration for large collections of EDDA diagrams.

There are three kinds of blocks: elementary, data directing, and flow directing. They represent primitive operations and data types, compound data structures, and control structures, respectively. Stepwise decomposition of small clusters of blocks (not more than seven per diagram) produces an hierarchical system of dataflow graphs.

A type name must be assigned to every data path in an EDDA design (EDDA is strongly typed). Allowable types are either elementary (int, real, char, bool or their derivatives) or built up by the user using three classes of type structures. Each type structure specifies a (multiway) split in the data type being defined. An EDDA type specification also defines manipulation functions for separating or combining individual data elements.

One of the type structures used in data directing blocks, "sequence," is equivalent to a Pascal record; such breakdowns are marked with a "+". Another, called "iteration" ("*"), subsumes arrays and files (and assumes parallel access to individual data elements). "Selection" ("x") is the third; it differs from the iteration structure in merely choosing one of several alternative data elements, rather than splitting a data structure into items which can be processed in parallel.

EDDA puts data structure ahead of control flow, similar to the attitude exhibited by SADT, but avoids worrying about how to accommodate details of physical storage. Constructs for storing or retrieving data from individual memory cells are omitted. Data can appear only on flow lines (in buffers) and cannot be globally addressed. This prevents implicit serialization of data accesses, but also neglects issues of communication topology (and may encourage a message-passing design).

There are also three types of flow directing blocks: Case, Merge and Loop (while/until). An EDDA Case routes incoming data to one of several outputs, depending on guard conditions of the outputs. When input consists of several paths, fresh data must exist on all of the paths, or Case does not fire. If more than one guard condition is satisfied by an input, it is routed to a single output data path, selected with unfair non-determinism. Merge, the inverse function of Case, combines all its inputs into one output. Loop blocks permit arbitrary iteration.

Another type of predefined block exists to permit use of "foreign language" subroutines in EDDA procedure diagrams. The parameters of a foreign block must correspond to the names of its input and output data paths. Any communication between foreign subroutines which circumvents their explicitly declared ports is forbidden (users cannot expect EDDA to detect problems that exist outside of any design specification).

This mechanism has three motivations. "Programming in the large" exhibits visible parallelism and benefits from functional programming, whereas "programming in the small" tends to involve short sections of strictly sequential code. Secondly, foreign subroutines simplify the compiling process. Finally, EDDA users are thus able to use preexisting routines written in conventional programming languages.

Two levels of EDDA exist. The first, EDDA-C, does not enforce any constraint on flow of data elements (Petri tokens) between blocks, and hence is subject to potential deadlock or overflow due to combination of branching blocks (Selection-split, Case) and merging blocks (Selection-combine, Merge). An EDDA-C subset that omits recursion (EDDA-D) produces designs that can be easily converted into Petri-Nets for deadlock analysis.

Addition of a simple structuring rule produces EDDA-S, and eliminates deadlock and overflow from conforming designs. For every input token of a Case block (or Selection-split) exactly one token must be produced on the output datapath of the corresponding Merge block (or Selection-combine) [Raniel85].

SECTION 4 - TOOLS FOR PARALLEL DEBUGGING

The overwhelming complexity of asynchronous, highly parallel computation makes debugging support particularly important. At the same time, however, it is difficult to provide this support because parallel systems are not amenable to existing debugging techniques. Parallel systems have enormous amounts of potentially relevant information, they do not have meaningful global states and they exhibit nondeterminism (time-dependent behavior).

- Enormous Amounts of Relevant Information. It is hard to keep track of relevant values for a single sequential program. It is even harder to keep track of the values generated by a collection of cooperating processes. Thus, some form of information reduction is necessary and methods of presentation are important.

- Absence of Meaningful Global State. Since most architectures do not impose lock step execution, a system may never reach a consistent global state in which all instructions have terminated. Even if such states existed, it is unlikely that the programmer could interpret them because the state space is so large (due to the many possible interleavings of instructions).

- Presence of Nondeterminism. Because of asynchronous operation, the results of a highly parallel program may be time-dependent. This implies that errors are not necessarily reproducible and that program monitoring may not be transparent.

As a result, the standard break/examine cycle of debugging does not work. In this cycle, the user stops the execution of his program to browse through the system state. He may then choose to rerun the program to stop at an earlier break point, to get a more detailed look at specific behaviors or to add additional output statements. In parallel systems, break/examine techniques are suspect because of the amount of relevant information to be considered and because of the absence of meaningful global states; cyclic strategies are not viable because of the absence of reproducibility.

We can identify several possible solutions to these problems. In the first, the system is reduced to sequential execution for debugging purposes. Once the program has been debugged, it is run in parallel. This approach gives the programmer a chance to use a familiar environment with all of the many tools already available for sequential debugging. It has the serious disadvantage, however, of masking time-dependent errors, which will reappear when parallel execution is resumed. In the second approach, debugging is done post-morten based on information collected during execution. This approach allows the user to go back and forth over the trace as often as he likes. It is, however, "one shot" in the sense that all relevant information must be collected even before it is known that an error occurred. Thus enormous amounts of trace data are maintained during normal operation. The third approach attempts to reduce the amount of relevant information by providing behavioral abstractions - models of expected behavior that can be compared to actual system behavior.

There are few extant parallel debuggers. The few that we have chosen to discuss here were chosen not because they are representative, but because they are novel and complementary.

Together, they combine aspects of reproducibility, behavioral abstraction and graphics-oriented user interfaces that could form the basis for the development of a more sophisticated parallel debugger.

4.1 CBUG

A concurrent debugger provides a direct means of observing not only dynamic behavior of individual tasks but also interactions between them. Many existing debuggers are not useful in multiprocessing environments. They are incapable of exploring many problems that occur in systems of communicating asynchronous tasks (e.g., deadlock, synchronization points, traces of individual tasks, and other parallel actions).

More suitable tools can be tied either to the compiler or run-time system of a programming language and operating system, or to specific hardware. The former approach facilitates portability across various processor designs and interconnect configurations. A concurrent debugger must permit dynamic intervention by the user and restrict the kind and quantity of information collected and presented.

The debugger described in [Gait85] requires no modifications to the target operating system, although it is dependent on specific features of Unix. CBUG is hardware independent, even though tied to a specific language and operating system, and thus may migrate wherever C and Unix can. Its window-oriented graphics gather activity from individual tasks together on a single display.

CBUG presents a uniform user interface at a source code level, with the same perspective of concurrency as exists during program design and composition. The author's test environment uses C and Unix system calls with semaphores and shared memory on a uniprocessor, however its design admits portability to an environment having only message-based communication.

Supported debugging aids applicable to individual members of a set of tasks include dumps of all accessible variables, conditional or interactive breakpoints, and single-stepping execution. Users can also trace execution across an entire task set, monitoring task invocations, interprocess synchronization points and message traffic.

Tracing of subroutine calls and stack contents is omitted, being considered of limited use in a multiprocessing environment. The basic unit of a selective trace is a task. CBUG's author suggests use of a conventional sequential debugger in situations where such quantities are of interest. Breakpointing and single stepping are of real use only in critical sections (of sequential execution). A task set in CBUG executes transparently to breakpoints and single--stepping.

CBUG modifies source code to return control to the debugger after execution of each line, and to provide breakpoint testing, single stepping, and status displays. During execution, only the original source code is displayed. Task scheduling is performed by the normal (non-deterministic) Unix scheduler.

A supervisory routine sets up required semaphores and communication channels, starts and terminates individual tasks. Each task contains an embedded copy of CBUG, all of which communicate with each other. The result executes more code than the programs being debugged, and may thus potentially exhibit a probe effect (in which delays introduced at each line of code may mask timing or synchronization errors). However, the probe effect did not arise in normal testing performed by CBUG's author.

Trace windows make the reason for a deadlock fairly obvious. A separate viewing area exists for each task, with provision for command entry to CBUG, and display of I/O occurring on standard Unix channels, pipes, shared memory, semaphores, open files, etc. Concurrent programs may need to be run several times to expose (or confirm the absence of) synchronization errors. A command for slow tracing introduces an explicit and variable delay prior to execution of each line of code (to expose probe effects).

4.2 DISDEB

Indeterminate timing relationships among tasks executing on separate processors prevent simultaneous intervention and make it difficult to ensure debugger transparency. These relationships can be distorted by excessive debugger overhead. [Lazzerini86] describes a concurrent debugger developed for a multi-microprocessor system designed for real-time process control.

The DISDEB system uses special-purpose hardware devices to monitor concurrent processing with a minimum of overhead. No changes whatsoever are made to compiled code. Only minor modification of the operating system kernel is needed. Activity on a target system is recognized as specific configurations of signals on an (interprocessor) bus.

A high-level command interpreter resident on a host system coordinates activity of several programmable debugging aid (PDA) boards. Each PDA is connected both to a signaling bus (for communication with the host) and to a bus on its target processor (to intercept actions in real-time). (See Figures 1 and 2 of [Lazzerini86].

The target computer(s) provide protected memory environments (nodes, perhaps comprising multiple processors), and Ada-package-like protected access to programs executing therein. Standard synchronization mechanisms (locks, semaphores) are supported with message-oriented communication. A system configuration language (SCL) specifies the number of processors per node, programs to be executed thereon and their partitioning among processors, task groupings and protection domains of programs, and levels of protection/privilege associated with each function environment.

DISDEB is a widely distributed debugger for multiprocessing programs. Designing the debugging system as an optional add-on component makes it possible to detect a wide range of conditions on the target machine without slowing its execution. Generation of an interrupt request internal to the PDA's microprocessor triggers firmware that can implement complex actions required by the debugging system.

4.3 BELVEDERE

The design of Belvedere is based on the belief that the behavior of highly parallel programs is best understood in terms of the flow of data and control resulting from interprocess communication and that these behaviors are often very structured: fine grain, tightly coupled processes communicate across regular interconnection networks resulting, at least logically, in patterned data and control flows. Belvedere [Hough87] is, therefore, a "pattern-oriented" debugger that enables the user to identify, manipulate and animate communication patterns. (Belvedere comes from the Latin bellus meaning "beautiful" and videre meaning "view.") It is a trace-based, post-mortem debugger.

Belvedere treats the event trace of a system as a relational database, providing animations of user-selected events. (Displays for animation are constructed from positional information normally obtained during program specification.) Patterns of expected interactions are described using the behavioral abstraction approach of Bates [Bates83], thus reducing the amount of presented information. Abstract events are recognized in the trace and made available as the targets of queries. Selected abstract events can be animated from a number of user-defined perspectives.

Belvedere is currently running within the Simple Simon Programming Environment [Cuny87] (a rudimentary environment for prototyping parallel programming support tools).

4.4 INSTANT REPLAY

A debugger should suppress irrelevant information to improve human analysis of program errors. The debugging technique most commonly used today is cyclic: a program is executed repeatedly (using an interactive debugger) until an error manifests itself. This is an improvement over an earlier technique, examination of an exhaustive record ("dump") of states of an entire execution. Such repetition works well for deterministic sequential programs.

Debugging parallel programs is difficult because their non-determinism complicates cyclic debugging. Successive executions of the same program may not produce identical results. A group at the University of Rochester has developed a debugger for the BBN Butterfly which enforces partial orderings among initially non-deterministic multiprocessing programs (during repetitive quasi-deterministic re-execution) without an exhaustive record of intermediate data [Leblanc87].

Parallel debugging must either collect all required information for diagnosing program errors during a single execution (the "dump" strategy revisited), or provide a mechanism to guarantee reproducible program behavior. There are at least two problems with the former approach. If the debugger is to capture all potentially interesting events, the user must somehow specify them prior to execution. Even given such specification, most of the information collected is voluminously uninteresting.

The later approach, reproductible program execution, permits application of cyclic debugging techniques. The authors of [Leblanc87] mention several prior methods, each of which has difficulties. Concurrent processes which interact through semaphores exhibit a total (and reproducible) sequence of synchronizations. The disadvantage of this method is that much potential parallelism is lost by the required serialization of semaphore actions.

Another method keeps a checkpoint record of each version of every atomic object in a computation. This restricts computation to programs structured as nested atomic actions, and incurs enormous storage overhead. A detailed event log has also been used to reproduce parallel program execution behavior.

Event logging has several disadvantages. Communication acts are assumed to take place infrequently, and time needed to copy a message to the log is assumed small relative to the time needed to send the message. Again, space requirements for the event log are daunting for real computations. Finally, this technique does nothing to simplify determination of global effects of multi-task interactions. Yet a simple modification avoids these problems.

The technique of Instant Replay [Leblanc86] makes it possible to reproduce the behavior of a parallel program, based on a minimal trace of process interactions. These interactions are modeled as operations on shared objects. Each modification of such an object produces a new version which is identified by a version number unique to that object. Processes record the associated version numbers on their inputs and these logs are used to replay an execution. During replay, accesses to standard variables are delayed until version numbers match the logged values; as a result, each process consumes the same input values in the same order and produces the same output values in the same order, repeating the previous execution. The system can be cyclically debugged, adding new (external) outputs and new break points where necessary.

Only the relative order of significant events needs to be recorded, not actual data associated with each event. Identical inputs from the external environment are presented in the same relative order during debugging replay as occurred during the original program execution. This requires substantially less time and space, and produces repeatable behavior because each (sequential) task in a parallel program is deterministic. When provided with a fixed series of inputs in a given order, it will produce a fixed series of outputs in the same order each time.

The Instant Replay debugger models all interactions between tasks as operations on shared objects. Modifications on the objects are recorded as a totally ordered series of versions (a version number is maintained and updated for each object). During re-play, each process recomputes its own output values. The record of object accesses is used to ensure that the same versions of input values are used during replay as occurred originally.

Instant Replay techniques have minimal impact on program execution since they do not log the enormous amounts of information that might otherwise be needed and because the information that they do record can be maintained locally, avoiding bottlenecks. They could be used in conjunction with other trace-based debuggers such as Belvedere: during normal execution, the system would generate the minimal trace needed for replay; if an error is encountered, a replay would be used to generate the remaining trace information. Instant Replay techniques have been successfully implemented on the BBN Butterfly.

Although this approach has many virtues, it may not be well suited for real-time applications, which often receive input as the result of asynchronous interrupts. Provision to record when interrupts occur during program execution exacts a significant performance penalty, and without it, timing of such inputs cannot be accurately repeated.

The method also requires that the set of operations on shared objects have a valid serialization: the result of each individual operation must be equivalent to what would be obtained from a definite sequential execution order of all the operations. Shared objects must also be "regular": all reads not concurrent with a write must get correct values, and any read that overlaps a series of writes must obtain either the object value prior to the first write, or one of the values being written.

Concurrent-read-exclusive-write (CREW), which ensures total orderings of writers with respect to each shared object, of readers with respect to writers of each shared object, and a partial ordering of readers with respect to each shared object, is one suitable object access protocol. Another simpler one, the mutual-exclusion (ME) protocol, forces serial execution of operating system event and queuing primitives, but reduces overhead significantly for such atomic operations.

A separate "history tape" for each process records significant events at an arbitrary level of detail, and is modified only by its corresponding process. This is very useful for debugging nondeterministic selection statements (by recording which alternative was chosen), as well as system timeouts and clock accesses.

Recomputation required to reproduce an execution sequence using history tapes (rather than exhaustively detailed event logs) is both an advantage and a disadvantage. The expense of maintaining an arbitrarily detailed event log trades off against the expense of re-execution during replay. However, the repeatable execution supported via history tapes allows programmers to continue to use simple cyclic debugging techniques, e.g., addition of output statements. Absent of the history log, such alteration often changes the relative timing of operations and consequently, their execution ordering. Successive replays also allow gradual refinement of detail (i.e., avoids overwhelming a programmer with incomprehensible amounts of output).

Another popular sequential technique is breakpoint insertion. The ability to halt a distributed program in a consistent state permits a programmer to examine the global state of a computation (to the extent that such a state exists with respect to communication delays). Finally, single-step execution can be used to trace state transitions of individual processes without destroying synchronization.

SECTION 5 - TOOLS AND TECHNIQUES FOR PERFORMANCE EVALUATION AND TUNING

5.1 PROBLEMS OF PERFORMANCE ESTIMATION

The performance of highly parallel architectures is critically dependent on the match of algorithm to hardware. Thus it would be nice to be able to answer such questions as:

- Given an algorithm, which architecture would run it most efficiently?
- Given an architecture, what algorithm should be used to solve a particular problem?
- Given an architecture and an algorithm, what can be done to improve performance?

Unfortunately, in most cases, the answers to such questions cannot be determined because we lack the necessary performance prediction and performance tuning tools.

The numerous parameters which influence performance of a multiple-processor system make performance estimation exceedingly difficult without detailed knowledge of a specific application. Among parameters which may exert significant influence [Cvetanovic86] are:

a) inherent parallelism of the problem

b) decomposition method

c) processor assignment or scheduling method

d) subproblem granularity (grain size)

e) potential for interleaving processing and communication

f) memory access method (global versus local)

g) processor interconnection network topology

h) relative speeds of processors and communication links

The parameters are not entirely independent. For example, appropriate granularity for a given algorithm is highly dependent on hardware features. The smallest feasible quantum of parallelism is dictated by an algorithm's communication patterns and the network topology [Reed87a]. Excessive parallelism can produce a diminishing return of performance or even a decrease due to the increased communication overhead.

Although there is a large body of literature on performance evaluation for sequential machines, many of the techniques developed for those systems do not transfer into the parallel domain. Because parallel problems are so complex, analytical methods often fail and simulation techniques are often too expensive. It is not even clear that we know the correct measures of parallel performance (a common measure, MIPS, for example, may go down on a vector processor as computation increases) [Segall 87].

5.2 TOOLS FOR PERFORMANCE PREDICTION

5.2.1 Asymptotic Performance Limits

Static effects of interconnection topology are evident from examination of asymptotic properties of queueing networks. [Reed87a] develops a simple analysis method for topology selection, based on network diameter, average path length and visit ratios.

The maximum internode distance (diameter) of a network is the greatest number of links a message must traverse to reach any node along a shortest path (largest minimum path). Diameter is independent of how paths are distributed through a network.

Mean internode distance is determined by first specifying for an arbitrary pair of nodes their probability of exchanging a message (routing distribution), then computing a sum of path lengths weighted by cumulative exchange probabilities (from unit length up to the network diameter). More simply put, this is an average path length (number of links a typical message visits).

Closed-form solutions for the mean internode distance exist for at least three different message routing distributions in a symmetric network. (Symmetric networks are self-isomorphic, i.e., a given node of the graph maps onto any other node. Non-symmetric networks are discussed in [Reed87b]).

A uniform distribution of message routing, where the probability of node i sending a message to node j is the same for all i and j, yields a likely upper bound on mean internode distance for a given topology. A second distribution assumes that nodes in close proximity (within a "sphere of locality") exchange messages with high frequency. Destinations within a given radius of a source receive messages with probability p, whereas more distant nodes are targeted with probability 1-p. Iterative partial differential equation solvers such as those which motivate [Berger87] typically exhibit this kind of communication locality.

When the size of a locality approaches the network size, or the probability of visiting a given locality is low, a third distribution becomes useful. Decreasing probability message routing assumes that a node transmits a message with decreasing probability to destinations at increasing distances. One such distribution computes probability weights as a locality parameter d raised to the power l (path length) multiplied by a normalizing function (of network diameter and the parameter d).

Mean internode distance alone does not completely determine node and link utilization. One must also consider how often the average message crosses a given link or triggers computation at a specific node. The "visit ratio" is the mean number of times a node or link is visited by a single message. For symmetric networks which have the same message routing behavior at all nodes, the visit ratio of a node is simply the reciprocal of the total number of nodes, 1/N. For a network which contains only one kind of link, the visit ratio for a link is the ratio of the mean internode distance over the number of links.

Visit ratios provide bounds on the maximum rate of message transfer (cf figures 2, 3 and 4 of [Reed87a]). Given information about the relative speeds of processors and communication links, they can also be used to find the most appropriate size of subproblem (granularity). To prevent communication delays from limiting the maximum computation rate, the ratio of computation time to communication time for a message must be at least N times the maximum link visit ratio. This explains the performance motivation for a binary hypercube topology: the width of the network is fixed, so only the number of links incident on each node changes, and the optimal computation grain size is constant for any size network.

5.2.2 Decomposition Strategy and Bandwidth Limits

[Cvetanovic87] develops a model of a shared-memory multiple-processor architecture to determine how several iterative algorithms behave with respect to bandwidth, grain size, and decomposition method (considered as a reciprocal function of communication overhead). Slightly less attention is given to pipelining and memory access.

The author considers decomposition onto N processors, which yields processing times proportional to 1/N, together with three functional dependencies for communication overhead, proportional to 1/N, 1/SQRT(N) and 1. The 1/N case represents a system whose total communication requirement is independent of the number of processors; the unitary case yields a communication requirement which grows in proportion to the number of processors. The two intervals considered together define a "decomposition group", e.g., (N,N) is a group for which processing and communication are each proportional to 1/N.

Interconnection bandwidth is taken to represent effects of topology and memory allocation within a multistage network. Delay for accessing local data is considered part of processing time, whereas global access increases communication overhead. In either case, shared memory is visible to all processors. The author references examples of similar analysis with different assumed processor architectures in her PhD dissertation [Cvetanovic86].

Table 1 of [Cvetanovic86] provides formulas for estimating speedup that results from various combinations of decomposition group, memory access method and communication pipelining. Three quantities are required to compute an estimate: the number of processors N, the average time to complete one computation Tp, and the time needed to transfer a message through a network link (neglecting channel contention) Tc.

[Cvetanovic86]'s results indicate that:

- Where communication overhead can be distributed fully among N processors, speedup increases with N for any bandwidth except the worst case (single channel shared among all N, with a single communication request per cycle), for which speedup becomes the ratio Tp/Tc. Where communication overhead cannot be decomposed at all, speedup improves for small N to a maximum Tp/Tc then falls gradually to zero (hence at some point multi-processor performance becomes worse than that of a uniprocessor).

- Pipelining (overlapping intervals of communication and processing) improves performance only for small N, unless complete decomposition is possible (the (N,N) group). If a computation is a mixture of parallel and serial code segments, speedup becomes inversely proportional to the proportion of time spent in the serial code sections, for the best case bandwidth. For less than optimal bandwidth or communication decomposition, serial segments have diminishing importance to the time complexity of speedup.

- An interconnection bandwidth limit can reduce speedup from linear in N to a constant (independent of N), with increasingly severe effects when communication cannot be fully decomposed. If bandwidth is limited and the algorithm selected does not permit effective decomposition of communication, granularity must be reduced (computation must occur in bigger chunks) for performance improvement to occur.

- The ratio Tp/Tc determines maximum speedup and the optimal number of processors. When processors are much faster than their communication network, decomposition strategy and bandwidth limit have enormous effects. When communication occurs faster than any required processing, none of the factors considered by [Cvetanovic 87] are of major importance to performance analysis.

5.2.3 SPAN

SPAN (a Speedup Analyzer for Parallel Programs) [So87] is a tool for estimating the algorithmic speedup of parallel FORTRAN programs running on a shared memory architecture. Algorithmic speedup refers to the maximum performance that can be expected on a parallel machine and thus does not include degradation due to memory contention, necessary synchronization, etc. To avoid the expense of instruction by instruction analysis, SPAN uses the task as its unit of measurement. It is assumed that programs are deterministic and that their execution does not depend on the number of processors used. Tasks are identified from the parallel constructs of the source code and a trace of their sequential execution is obtained. A multiprocessing schedule is then constructed for that trace and estimates of speedup are based on this schedule.

5.2.4 CPPP

CPPP (the Cedar Performance Prediction Package) [Abu-sufah85] is a set of tools for predicting the performance of FORTRAN programs on the Cedar Multiprocessor Supercomputer. It has two important characteristics: it is hierarchical and it is modular.

CPPP is hierarchical in that it provides the user with a range of levels of performance prediction, varying in accuracy and cost. The first level is the least accurate and the least expensive. It consists of performance estimates provided by the Paraphrase compiler (the Illinois program restructurer). Paraphrase, like SPAN, ignores the many sources of performance degradation to provide rough estimates of algorithmic timing, speedup and efficiency. The second and third levels are based on simulation tools. In each case, the user is allowed to specify the degree of detail in the simulation and the set of statements that will be simulated. At the second, intermediate level, simulation is based on a simplified model of processors having both local and global memories. At the third level, a detailed simulation of processors, memory hierarchies and interconnection structures is provided. This is the most accurate and most expensive level. It provides numerous performance statistics such as total execution time, estimated speedup, the number of each type of operation performed, MIPS, MFLOPS, processor utilizations, etc.

CPPP is modular in the sense that all of its components - the processor model, the memory model and the global interconnection network model - are parameterized; the user can, for example, choose between a crossbar and a banyan network or they could determine the size of a switching element. This enables the user to predict on performance on alternative Cedar designs as well as on a variety of other architectures.

5.3 TOOLS FOR PERFORMANCE TUNING

5.3.1 General Requirements

Because the interaction between algorithm and architecture is so complex, it will be necessary to provide the user with support for tuning performance.

This requires provision for observing performance and status events at all levels, as well as some high-level representation for information collected during development and at run-time [Gregoretti86]. An infrastructure for observing, integrating and presenting parallel program performance information must extract both static and dynamic details.

Evaluating run-time performance involves separate policies for hardware, the operating system kernel, programming language run-time libraries, and the end-user application itself. Hardware monitoring concerns basic processor speed, communications throughput, cache performance and other memory contention, and buss and peripheral contention. Kernel policy includes context switching, communications and I/O buffering, memory management, file system, process creation and deletion, and system functions. Areas important at the language level are procedures, statement blocks, and control constructs, especially for communication and concurrency. The application level focuses on performance of a specific (parallel) algorithm.

The sensor mechanism of the monitoring system collects information at three levels of abstraction: detection, filtering and notification. Sensors can be hardware, software or a mixture of the two. A sensor must be non-intrusive (have minimum effect on operation of the object being observed), but also capable of accurately observing a desired quantity. The testbed described in [Gregoretti86] uses hybrid sensors to monitor the operating system kernel and language syntax/run-time levels, resorting to hardware sensors for monitoring data structures and the processor itself.

Hardware sensors can be attached as part of an in-circuit emulator (within the CPU), at CPU output pins, or as part of communication channels with peripherals and other processors. Instrumentation of the communication links usually provides the best cost/performance compromise in a multiprocessor system.

Instrumentation supports three primitive operations: a) counting of events and measurement of time intervals; b) data sampling; and c) detection of events which span more than one processor. The information collected is timestamped and stored into a FIFO memory for subsequent analysis.

5.3.2 Multiprocessor Profiling

Measurement of actual dynamic program behavior is typically achieved via interrupt-driven sampling (e.g., in uniprocessor Unix implementations). This type of approach is less feasible in multiprocessor environments, especially those which lack shared memory. A high-precision per-process timer is one reasonable substitute method [Carrington86].

Two time values are of interest: direct time in a routine, and cumulative time spent in a routine and all others it invokes. Collection of the second quantity also produces a graph of inter-procedure calling relationships. Dynamic analysis collects this data during execution of a target program, for subsequent reduction and presentation.

Standard System V Unix profiling counts the number of function calls and samples the program counter to develop an execution time profile. Sampling usually occurs via a line frequency clock interrupt (50 or 60 Hz). Use of faster sampling rates greatly increases measurement overhead. 4.2 BSD Unix retains additional information that permits construction of a dynamic call graph, in which each directed arc connects a caller to some subordinate routine.

Use of a separate hardware execution timer for each process permits computation of cumulative time for each routine during the measurement phase rather than during subsequent analysis. This approach requires that both entry and exit from each function be detected, to permit allocation of processor time to the most recently executed routine.

Handling for a timer at function entry closely resembles the code generated for profiling via interrupt-driven sampling. For function exit, special handling can be implemented with minimum alteration of an existing compiler by allocating an extra Unix stack frame at each function call (continuation). The counting routine for function calls invoked prior to entry allocates a supplemental stack frame, which is resumed after the target function executes and exits. Only after profiling information has been collected is the original caller resumed and the supplemental stack frame deallocated.

When profiling data are measured rather than sampled, there is no need to correlate execution address ranges with function addresses or to generate a call graph. Measurements are directly associated with function addresses. One minor problem with this approach is that time spent executing routines that are not being profiled is allocated to the calling routine. Another is that measurement can increase execution time from 20 to 40 percent (in the implementation by [Carrington]'s author on an Elxsi processor). However, overhead cost is isolated to a single process and profiling activity has no effect on behavior of the Unix kernel.

5.3.3 Monitoring Loosely Coupled Multiple Processors

A distributed program is just a group of tasks cooperating to achieve a common objective, perhaps distributed across several processors. Two extreme cases have all processes running on the same machine, or each process running on a separate machine. Components of a distributed program compute and communicate.

The performance of a distributed program is not completely described by paging activity or subroutine call frequencies of its individual tasks (or the machines on which they run). [Miller86] describes a distributed program monitoring system under 4.2 BSD Unix which collects performance data for activities that transcend machine boundaries.

Three factors contribute to increased complexity of performance metrics for a distributed program. Asynchrony results from the way components of a distributed program can execute with true simultaneity, but at non-deterministic rates. This complicates synchronization of the components. Further, distributed systems lack a universal time base across all processors, which prevents complete ordering of events in a computation. Finally, a finite and non-deterministic delay intervenes in communication between processors. Thus there is no way of obtaining an instantaneous picture of the state of a computation, and actions which have multiple preconditions cannot be accurately scheduled (and clocks cannot be fully synchronized).

Events in the measurement model include process creation, destruction, and execution; shipment, delivery and receipt of messages; and communication paths. Their observation should exhibit two features: transparency (the assumption that measurement of events has no effect on their progress) and consistency (the attempt to provide a view of computation at the same level of detail perceived by the programmer).

Performance measurement involves three stages: metering, filtering and analysis. Metering collects raw measurement data from the operating system. Filtering ignores part of the data and transforms the rest. Analysis draws simple conclusions from the data.

[Miller86]'s measurement tools trace system calls (in particular, interprocess communication) as an indication of distributed program activity. The tools are themselves structured as a distributed program with four components: metering, filtering, control and analysis.

Metering functions tend to be built into an operating system kernel, to improve accuracy of time stamping and reduce overhead (avoids some context switching). Metering detects events in the lives of tasks and forwards the measurements to a filter process.

The filter task selects events that meet certain criteria, summarizes and stores them until the control task requests them. Types of events which can be selected include initiation and acceptance of a communication connection; transmission, polling, and reception of messages; creation, reference or termination of a 4.2 BSD communication "socket"; and task forking or termination.

Analysis routines interpret the traces created by the filter to provide communication statistics, measures of the extent of parallelism and other structural features of a computation.

Coordination of the measurement tool components has to be indirect, since under 4.2 BSD process identifiers are local, i.e., meaningful only to the machine on which a task is running. (The provision for direct control of remote processes would require substantial modification of the basic 4.2 BSD operating system kernel.) A server process is created on each host in the distributed computation to carry out commands of control processes which reside on other machines. These exchanges are structured as remote procedure calls (except that, in the case of process termination, the local server unilaterally informs any remote controlling task).

As a side effect of indirect control, to meter a distributed computation a user must have an account on every machine involved in the computation.

5.3.4 PIE

PIE (the Program and Instrumentation Environment) [Ogle85] is a complete programming environment. Two of its more interesting components are a Modular Programming Metalanguage (MP) and a Relationship System.

The language, MP, is used to specify a program as a set of processes, frames, sensors and tasks. A process is a sequential code segment. A frame is a mechanism for controlling access to shared memory; it consists of an abstract data type together with constraints (synchronization). A sensor is a monitoring device inserted to detect and report the occurrence of a specific event (for example, a task termination or the change of the value of a variable). Sensors may be user-defined and may be enabled or disabled dynamically. A task is a parallel operation composed of processes and frames together with control information and the specification of monitoring devices (determining, for example, which sensors are to be enabled).

The Relational Representation System, much like that of Poker, provides different views of an integrated database containing all of the known information - static and runtime - on a program. This information is presented graphically; thus, for example a programmer can display both his static task structure graph and his runtime task structure tree which shows task creation and termination over time.

5.3.5 ISSOS

ISSOS [Ogle85] is a testbed for high performance parallel software that includes a programming environment, an operating system and monitoring mechanisms as well as parallel and distributed hardware. It is designed to provide the user with the information needed to tune his programs. He can do this statically by altering the program itself or dynamically by adjusting to runtime conditions. Runtime adaptations might, for example, include spawning an additional copy of a process whenever its input queue reaches some threshold or turning off sensors when system behavior appears to have stabilized. Our attention here is focused on the monitoring mechanisms.

ISSOS provides both sensors (small code segments within either the distributed operating system or the applications code itself) and probes (code segments within the resident monitor on each node having direct access to the address space of processes on that node). Sensors and probes may be used to produce event records either as histories (traces) or as samples (responses to individual queries). Collected information is filtered to insure that it meets the monitoring device specifications. To minimize communication throughout the system, this filtering occurs at all levels: within sensors, resident monitors and the central monitor.

APPENDIX A -- OBJECT ORIENTED PROGRAMMING

A.1 INTRODUCTION

In the object-oriented approach, a program is decomposed into objects that communicate through messages. An object encapsulates data and the operations on that data; operations are invoked in response to incoming messages and result in outgoing messages. This approach appears to be well suited for highly parallel computation [Odijk87]. The decomposition into objects provides a natural basis for parallelism and synchronization (on messages). Since the objects themselves form the unit of parallelism, the user controls granularity and can tailor different parts of his program for different architectures. Objects do not share data and, therefore, map in an obvious way onto the processors of an MIMD machine. The object-oriented approach produces well-structured and robust programs and facilitates the team designs and software engineering practices that will be needed to program significant applications [Odijk87].

Object-oriented programming is primarily a data abstraction technique, although it elaborates this technique with the notion of "inheritance." In addition, it provides a model of computation in which data and processing are contained in "objects" that communicate by sending messages. These objects are good candidates for distribution among nodes of a distributed system. They are also good candidates for internal concurrency and for special implementations that make use of specialized parallel hardware.

In terms of programming support, object-oriented programming will need many of the tools discussed previously.

- Programming Language Support. The semantics of object-oriented programming languages must be expanded to include parallelism. The obvious approach of placing each object on a separate processor is not sufficient because synchronous message transmission (where the sender waits for a response) will serialize execution. The alternatives are to (1) allow asynchronous messages or, (2) permit objects to initiate and maintain activities of their own. This second approach has been followed in POOL (Parallel Object-Oriented Language) [Odijk87] which is being developed as part of the DOOM project. (The DOOM (Decentralized Object-Oriented Machine) Project is a coordinated effort to use the object-oriented approach with massive parallelism. An architecture, a parallel programming language and a number of significant applications programs are being developed.)

- Mapping. Because architectures have limited connectivity, the placement of objects within the system will affect performance. Mapping assistance such as that provided by Prep-P is useful in determining the original layout, but dynamic mapping and load balancing techniques will be necessary. (Dynamic garbage collection to remove objects which are no longer referenced will also be essential.)

- Performance Prediction. Existing performance prediction tools could be easily adapted to object-oriented programming. In particular, Polylith (discussed above under program specification tools) would seem to have most of the necessary features; if more detailed simulations of hardware are required, a version of CPPP (without the FORTRAN input or shared memory constructs) could be developed.

- Debugging for Correctness. The available tools for debugging are quite limited but those that exist are of use in the object-oriented domain. The Instant Replay technique, for example, would work well at least for large grain objects (it would require extensive logging for small grain objects). Animation of communication, such as that found in Belvedere, is useful (although the very regular patterns that Belvedere exploits will occur only in very small grain parallelism). The modeling aspects of Behavioral Abstraction [Bates83] would be applicable.

- Debugging for Performance. The systems that do exist, such as PIE and ISSOS, would work well in an object-oriented system.

The object-oriented approach is well suited to massive parallelism and it could be used with many of the existing software tools. The existing tools, however, are themselves quite limited. The effective use of massive parallelism will require a substantial amount of development work in parallel programming environments.

The remainder of this Appendix defines object-oriented programming and discusses its value, both as a general technique for program development and as a method for incorporating concurrency within a program. We begin in Section A.2 by defining data abstraction and its role in the program development process. Then in Section A.3 we discuss inheritance and what it adds to program development. Next in Section A.4 we discuss the object-oriented model of computation and how it can be used to make use of concurrency. Finally, in Section 5 we discuss the support provided for data abstraction and inheritance in some programming languages.

A.2 DATA ABSTRACTION

Data abstraction is a means of abstracting from the way data structures are implemented to the behavior they provide. Data abstractions are particularly important because they hide complicated things (data structures) that are highly likely to change in the future. They permit the representation of data to be changed locally without affecting programs that use the data. They also simplify the structure of programs that use them because they present a higher level interface. For example, they reduce the number of procedure arguments, because abstract objects are communicated instead of their representations.

This section is devoted to data abstraction: what it is, what it buys in tne programming process, and how it exists in various programming languages. We begin by discussing abstraction in general, and then we move on to data abstraction; a more detailed discussion of this material can be found in [Liskov87]. We conclude with a discussion of the role of abstraction in program development.

A.2.1 Abstraction and Procedures

The purpose of abstraction, in programming, is to separate 'use' from 'implementation.' Procedures, the first programming abstraction mechanisms, illustrate the purpose of such mechanisms. A procedure implements some task or function that is of use within a program. To accomplish this task, another part of the program calls the procedure. In writing the program that makes the call, the programmer cares only about what the procedure does, not how it is implemented. Any implementation that provides the needed function will do, provided that it implements the function correctly and efficiently enough. Typically, the procedure has parameters that allow it to be tailored to the needs of a particular use. Some programming languages allow only simple parameters, such as integers or arrays. Others, such as Ada, allow more sophisticated parameters such as procedures or types.

To get the full benefit of procedures, we must emphasize the distinction between the abstraction and its implementation. Since the implementation is not the same as the abstraction, we need an independent description of the abstraction, called a specification. Thus we have an abstraction that is defined by a specification and implemented by a program module. Many modules can implement the same abstraction, as is shown in Figure A-1. Here abstraction A has a specification and n implementations.

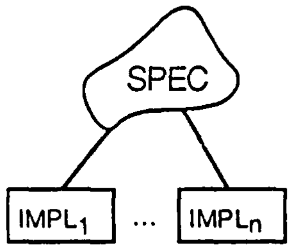


Figure A-1: Abstraction, Specification, and Implementations

An implementation is correct provided it meets the specification. Correctness can be proved mathematically if the specification is written in a language with precise semantics; otherwise we establish correctness by informal reasoning or by the somewhat unsatisfactory technique of testing. Correct implementations differ from one another in how they work, i.e., what algorithms they use, and therefore they may have different performance. Any correct implementation is acceptable to the caller provided it meets the caller's performance requirements. Note that correct implementations are not identical to one another. The whole point is to allow implementations to differ, while ensuring that they remain the same where this is important. The specification describes what is important.

A.2.2 Data Abstraction

The above subsection discussed the virtues of abstraction in general, and used procedural abstractions to illustrate the idea. Procedures go quite a long way, but in the early seventies some researchers realized that they were not enough [Parnas71, Parnas72, Liskov72]. These early papers proposed a new way of organizing programs around the "connections" between modules. The concept of data abstraction, or abstract data types, arose from these ideas [Hoare72, Liskov74].

A data abstraction provides the same benefits as procedures, but for data. Recall that the major idea is to separate what an abstraction is, from how it is implemented, so that implementations of the same abstraction can be substituted freely. An implementation of a data object is concerned with how that object is represented in the memory of a computer. This information is called the representation, or rep for short. To allow changing implementations without affecting users, we need a way of changing the representation without having to change all using programs. This is achieved by (1) encapsulating the rep with a set of operations that manipulate it, and (2) restricting using programs so that they cannot manipulate the rep directly, but instead must call the operations. Then, to implement or reimplement the data abstraction, it is necessary to define the rep and implement the operations in terms of it, but user code is not affected by a change. Thus a data abstraction is

<objects, operations>

i.e., a set of objects that can be manipulated directly only by operations in the set. An example of a data abstraction is the integers: the objects are 1, 2, 3, and so on and there are operations to add two integers, to test them for equality, and so on. Programs using integers manipulate them by their operations, and are shielded from implementation details, such as a representation using 2's complement. Another example is strings, with objects "a" and "xyz", and operations to select characters from strings and to concatenate strings. A final example is sets of integers, with objects { } (the empty set) and {3, 7}, and operations to insert an element in a set, and to test whether an integer is in a set.

Note that integers and strings are built-in data types in most programming languages, while sets and other application-oriented data abstractions, such as stacks and symbol tables, are not. To allow such user-defined abstract data types to be implemented, new linguistic mechanisms were needed as discussed further below.

Just as for procedures, the meaning of a data abstraction is defined by a specification. A data abstraction can be implemented in many different ways; an implementation is correct provided it satisfies the specification. In addition, the users of the data abstraction must be constrained so that they do not access the rep directly, but instead use the operations. Encapsulation is needed or the benefits of abstraction will be lost. Encapsulation is related to the principle of "information hiding" advocated by Parnas [Parnas]; it guarantees that information about how a data abstraction is implemented is hidden from the rest of the program.

Data abstractions are supported by linguistic mechanisms in several languages. Two major variations are discussed below.

The first of these is CLU [Liskov77, Liskov84]. (The data abstraction mechanism in Ada, discussed in Section A.5.4 borrows heavily from CLU's mechanism.) CLU provides a mechanism called a cluster for implementing an abstract type. A template for a cluster is shown in Figure A-2. The header identifies the data type being implemented and also lists the operations of the type; it serves to identify what procedure definitions inside the cluster can be called from the outside. The "rep =" line defines how objects of the type are represented; in the example, we are implementing sets as linked lists. The rest of the cluster consists of procedures. There must be a procedure for each operation, and in addition, there can be some procedures that can be used only inside the cluster.

```
int_set = cluster is create, insert, is_in, size, ...
     rep = int_list
     create = proc ...
     insert = proc ...
     ...
end int_set
```

Figure A-2: Template of CLU cluster

In SmallTalk [Goldberg83], data abstractions are implemented by classes. A class implements a data abstraction similarly to a cluster. Instead of the "rep =" line, the rep is described by a sequence of variable declarations; these are the instance variables. The remainder of the class consists of methods, which are procedure definitions. There is a method for each operation of the data type implemented by the class. (There cannot be any internal methods in SmallTalk classes because there is no way of identifying what methods are available for outside use.) Methods are called by "sending messages," which has essentially the same effect as calling operations in CLU.

One important difference between CLU and SmallTalk in how they enforce encapsulation. CLU enforces encapsulation with compile-time type checking, while SmallTalk uses runtime type checking. E.g., suppose s is an int_set, and consider the illegal call

```
     x: int := car(s)
```

(Here car is an operation on lists that returns the first element of the list.) In CLU, the call would be found to be illegal by the compiler, since car expects a list argument, not a set. In SmallTalk, the error would be found when the statement is executed at runtime.

Compile-time checking is superior for real-time projects for two reasons: it allows a class of errors to be automatically eliminated from a program, and it permits more efficient code to be generated by a compiler. Thus SmallTalk is not a good language for use in real-time projects. Ada is a reasonable choice; it has a data abstraction mechanism similar to CLU's and uses compile-time checking.

Other Object-oriented languages are for the most part even less satisfactory than SmallTalk, e.g., [Bobrow86, Moon86], because they do not enforce encapsulation at all. It is true that in the absence of language support, encapsulation can be guaranteed by manual procedures such as code reading, but these techniques are error prone. In essence, languages that enforce encapsulation by either compile-time or runtime checking guarantee an important property of programs, which can be relied on with confidence and without the need to read any code at all. Runtime checking is less satisfactory than compile-time checking because violations of encapsulation are found later. Languages without checking provide no guarantees, which means that programmers must read lots of code to be sure that encapsulation holds. Note that although things may be somewhat manageable for a newly-implemented program, they will degrade rapidly as modifications are made.

A.2.3 Benefits of Abstraction

Abstraction provides locality within a large program. Locality allows a program to be implemented, understood, or modified one module at a time:

1. The implementer of a module knows what is needed because this is described in the specification. Therefore, he or she need not interact with programmers of modules that use this abstraction, or at least the interactions can be limited.

2. Similarly, the implementer of a using module knows what to expect, namely, the behavior described by the specification.

3. Only local reasoning is needed to determine what a program does and whether it does the right thing. The program is studied one module at a time. In each case we are concerned with whether the module does what it is supposed to do; that is, does its implementation satisfy its specification. We can limit our attention to just that module and ignore both modules that use it ("consumers") and modules that it uses ("suppliers"). Consumers can be ignored because they depend only on the specification of this module, not on its code. Suppliers are ignored by reasoning about what they do using their specifications instead of their code. There is a tremendous saving of effort in this way because specifications are much smaller than programs. For example, if we had to look at the code of a called module, we would be concerned not only with its code, but also with the code of any modules it used, and so on.

4. Finally, program modification can be done module by module. If a particular abstraction needs to be reimplemented (to provide better performance or correct an error or provide extended facilities), the old implementing module can be replaced by the new without affecting using modules.

Note that both specifications and encapsulation are critical to locality. Specifications allow separation between modules by serving as a contract between consumers and supplier, allowing each to work independently. Encapsulation ensures that modules are really independent, so that it is safe to concentrate on just the module at hand. For example, encapsulation ensures that a module can be reimplemented without looking at the code of other modules.

Locality provides a firm basis for fast prototyping. Typically there is a tradeoff between the performance of an algorithm and the speed with which it is designed and implemented. The initial implementation can be a simple one that performs poorly. Later it can be replaced by another implementation with better performance. Provided both implementations are correct, the calling program's correctness will be unaffected by the change.

Another use of locality is to encapsulate potential modifications. For example, suppose we want a program to run on different machines. We can accomplish this by inventing abstractions that hide the differences between machines, so that to move the program to a different machine, only those abstractions need be reimplemented. A good design principle is to think about expected modifications and organize the design by using abstractions that encapsulate the changes.

The benefits of locality are particularly evident in data abstractions. Data structures are often complicated, and therefore the simpler abstract view provided by the specification allows the rest of the program to be simpler. Also, changes to storage structures are highly likely as programs evolve. The effects of such changes can be minimized by encapsulating them inside data abstractions.

A.3 INHERITANCE

The previous section discussed data abstraction and its benefits in program development. Data abstraction is a powerful tool in its own right. Certain uses of inheritance can enhance this tool. Therefore, we now go on to discuss inheritance and what these uses are. We begin by talking about what it means to construct a program using inheritance. Next we discuss two major uses of inheritance, only one of which (subtyping) is of interest in programming methodology. Finally, we discuss how the subtype hierarchy can help in program development.

A.3.1 Class Hierarchy

In a language with inheritance, a data abstraction can be implemented in several pieces that are related to one another. Although various languages provide different mechanisms for putting the pieces together, they are all fairly similar. Thus we can illustrate them by examining a single mechanism, the subclass inheritance mechanism in SmallTalk.

In SmallTalk, a class can be declared to be a subclass of another class (or classes); such a class is its superclass. To understand what a subclass does, we need to understand what code results from such a definition. For example, if we were to reason about the correctness of a subclass, we would need to look at this (resulting) code.

From the point of view of the resulting code, saying that one class is a subclass of another is simply a shorthand notation for building programs. The exact program that is constructed depends on the rules of the language, e.g., such things as what to do if two immediate superclasses have instance variables or operations of the same name, and when methods of the subclass override methods of the superclass. The exact details of these rules are not important for our discussion (although they clearly are important if the language is to be sensible and useful). The point is that the result is equivalent to directly implementing a class containing the instance variables and methods that result from applying the rules.

An example is given in Figure A-3. Figure A-3a shows two classes, S and T, where S is a subclass of T. Here T has operations O_1 and O_2, and S, which is declared to be a subclass of T, provides a method for O_1 and another method, O_3. Also T has instance variable V_1 and S has a further instance variable V_2. Then the result in SmallTalk, as shown in Figure A-3b, is effectively a class with two instance variables, V_1 and V_2 and three operations, $O_1, \ldots, O_3$, where the code of O_2 is supplied by T, and the code of the other two operations is supplied by S. It is this combined code that must be understood, or modified if S is reimplemented, unless S is restricted as discussed further below.

One problem with almost all inheritance mechanisms is that they compromise data abstraction to an extent. In Object-oriented languages, a data abstraction implementation has two classes of users. First there are the "outsiders" who simply use the objects. But in addition there are the "insiders". These are the subclasses. Typically the rep is fully accessible to the subclasses, and therefore the benefits of data abstraction can disappear for them. In particular, if a subclass violates encapsulation, then to understand it we must examine the combined reps of sub- and superclasses and the code of any superclass operations that have not been overridden by subclass operations. When encapsulation is not violated, the code is simpler. We need only look at the rep of the subclass; reps of superclasses can be ignored. Furthermore, we need not look at implementations of operations inherited from the superclasses. Instead, we can reason about them using their specifications.

If encapsulation is violated, we have the following disadvantages:

1. Reasoning about the code is more complicated, since we must consider the combined code. Thus we lose the benefits of locality.

2. If the superclass needs to be reimplemented, we must consider all of its subclasses; if they violate encapsulation, we will need to reimplement them.

```
        Class T

             v1   an instance var
             o1   a method
             o2   another method

        Class S subclass of T

             v2   an instance variable
             o1   a reimplementation of T's o1
             o3   a new method

   (a)  An example of inheritance.

        Class S

             v1
             v2

             o1   this definition comes from S
             o2   this definition comes from T
             o3   this definition comes from S

   (b)  The resulting code.
```

Figure A-3. How Inheritance Works

3. If a subclass refers to any superclass of one of its superclasses, this is another violation of encapsulation. It means that we cannot change the hierarchy above the subclass without possibly needing to change the subclass too. For example, suppose class S refers directly to R, where R is a superclass of T, and T is a superclass of S. Now suppose that we reimplement T so that R is no longer its superclass. In this case, S will no longer be meaningful.

Finally, we note that if there is no enforcement of encapsulation for superclasses, then even if a subclass does not violate the superclass's encapsulation, we cannot rely on this when modifying the superclass. Instead, it is necessary to at least examine all subclasses. Thus we have the same situation here that we had for data abstraction in languages that provided no mechanism for enforcing encapsulation.

A.3.2 Implementation Hierarchy

The first way that inheritance is used is simply as a technique for implementing data types that are similar to other existing types. For example, suppose we want to implement integer sets, with operations (among others) to tell whether an element is a member of the set, and to determine the current size of the set. Suppose further that a list data type has already been implemented, and that it provides a member operation and a size operation, as well as a convenient way of representing the set. Then we could implement set as a subclass of list. We would not need to provide implementations for member and size. We would need to implement other operations such as one that inserts a new element into the set. We probably would like to suppress certain other operations, such as car, to make them unavailable since they are not meaningful for sets. (This can be done in SmallTalk by providing implementations in the subcall for the suppressed operations; such an implementation would signal an exception if called.)

Another simple way to implement sets is to use list as the rep. In this case, we would need to implement the size and member operations; each of these would simply call the corresponding operation on lists. Writing down implementations for these two operations, even though the code is very simple, is more work than not writing anything for them. On the other hand, we need not do anything to take away undesirable operations such as car.

Of course, using inheritance we can violate encapsulation. This ability can be useful, but is best viewed as being a fast way to write programs by rewriting the code of existing programs. The result is a completely separate piece of code, and changes to the implementation of the superclass cannot be reflected easily into the new code. For example, suppose we implement set by starting from list and modifying its implementation using inheritance, and then list is reimplemented and we want to use its new implementation in the set implementation. To do this we must reimplement set.

Thus, inheritance is used for implementations in two different ways. In one case, encapsulation is violated and its benefits are lost. This ability to construct programs fast is probably useful, but it does not require any special programming language mechanisms. Instead, an interactive editing system that helps a programmer produce a new program from an old one would be quite satisfactory. (It would be useful to keep track of the relationship between the two classes, so that if the superclass changes, the programmer can be informed in case he or she wants to reimplement the subclass.) The other case preserves program structure, but has little if anything to offer over the alternative of using one (abstract) type as the rep of another. Therefore, we will not consider further using inheritance as an implementation technique.

A.3.3 Subtyping

The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: Consider any program P that is defined in terms of type T. If type S can be uniformly substituted for type T in P without changing the behavior of P, then S is a subtype of T. Of course this definition is vague, since researchers in this area have not yet succeeded in capturing precisely what is wanted. Research [Bruce86, Leavens88] is underway to determine what the appropriate definition should be.

We are using the words "subtype" and "supertype" here to emphasize that now we are talking about a semantic distinction. By contrast, "subclass" and "superclass" are simply linguistic concepts in programming languages that allow programs to be built in a particular way. They can be used to implement subtypes, but mentioned above, in other ways.

We begin with some examples of types that are not subtypes of one another. First, a set is not a subtype of a list nor is the reverse true. If the same element is added to a set twice, the result is the same as if it had been added only once; for example, in computing the size of the set, the element is counted only once. However, if the same element is added twice to a list, it occurs in the list twice. Thus a program expecting a list might not work if passed a set. Similarly a program expecting a set might not work if passed a list. Another example of non-subtypes are stacks and queues. Stacks are LIFO; when an element is removed from a stack, the last item added (pushed) is removed. By contrast, queues are FIFO. A using program is likely to notice the difference between these two types.

Notice that the above examples ignored a simple difference between the pairs of types, namely related operations. A subtype must have all the operations of its supertype since otherwise the using program could not use an operation it depends on. However, simply having operations of the right names and signatures is not enough. (An operation's signature defines the types of its input and output arguments.) The operations must also do the same things. For example, stacks and queues might have operations of the same names, e.g., "add_el" to push or enqueue, and "rem_el" to pop or dequeue, but they still are not subtypes of one another because the meanings of the operations are different for the two types.

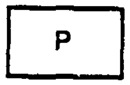
Now we give some examples of subtype hierarchies. The first is indexed collections, which have operations to access elements by index. All subtypes have these operations too, but in addition, each would provide extra operations. Examples of subtypes are arrays, sequences, and indexed sets. The second example is abstract devices, which unify a number of different kinds of input and output devices. Particular devices might provide extra operations. In this case, abstract device operations would be device independent, while subtype operations would be device specific. For example, a printer would have modification operations such as put_char but not reading operations such as get_char. Another possibility is that abstract devices have all possible operations, and thus all subtypes have the same set of operations. In this case, operations that not all real devices can do must be defined in a general way that allows exceptions to be signaled. For example,

```
get_char = proc (d: dev, c: char) signals (not_possible)
```

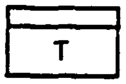
When this operations is called on a printer, it signals the exception.

A.3.4 Benefits of Subtypes

Subtypes are a useful adjunct to data abstraction in two ways. First, data abstractions are usually developed incrementally as a design progresses. In early stages of the design, we only know a few operations and a part of the behavior. Such a stage of design is shown in Figure A-4. The design is shown by means of a "module dependency diagram," or "mdd" for short, which illustrates how a program is subdivided into modules. An mdd is a graph with two kinds of nodes;



represents a procedure abstraction, and



represents a data abstraction. Thus the figure shows two procedures, P and Q, and one data abstraction, T. The nodes are connected by arcs. An arc consisting of a single line means that the abstraction whose node is at the head of the arc is implemented in terms of the abstraction whose node is at the tail of the arc. Thus P is implemented using Q (i.e., its code calls Q) and T (i.e., its code uses objects of type T). (Recursion is indicated by cycles in the graph. Thus if we expected the implementation of P to call P, there would be an arc from P to P.)

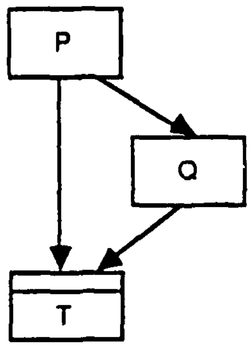


Figure A-4. The Start of a Design

This figure represents an early stage of design, in which the designer has thought about how to implement P and has invented Q and T. At this point, some operations of T have been identified, and the designer has decided that an object of type T will be used to communicate between P and Q.

The next stage of design is to investigate how to implement Q. (It would not make sense to look at T's implementation at this point, because we do not know all its operations yet.) In studying Q we are likely to define additional operations for T. This can be viewed as refining T to a subtype S as is shown in Figure A-5. Here, an arc drawn as a double line means that the data abstraction at the head is a supertype of the data abstraction at the tail; double line arcs can only connect data abstractions.

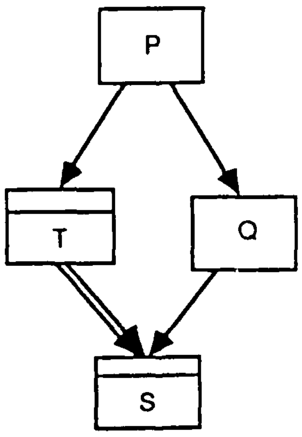


Figure A-5. Later in the Design

The kind of refinement illustrated in the figures may happen several times; e.g., S in turn may have a subtype R, and so on. Also, a single type may have several subtypes, representing the needs of different subparts of the program.

It is better, for several reasons, to keep track of these distinctions as subtypes, rather than treat the group of types as a single type. First, it can limit the effect of design errors. For example, suppose further investigation indicates a problem with S's interface. When a problem of this sort occurs, it is necessary to look at every abstraction that uses the changed abstraction. For the figure, this means we must look at Q. However, provided T's interface is unaffected, we need not look at P. If S and T had been treated as one type, then P would have had to be examined too.

Another advantage of distinguishing the types is that it may help in organizing the design rationale. The design rationale describes the decisions made at particular points in the design, and discusses why they were made and what alternatives exist. By maintaining the hierarchy to represent the decisions as they are made over time, we can avoid confusion and be more precise. If a specification error is discovered later, we can identify precisely at what point in the design it occurred and what other modules may be affected by it.

Finally, the distinctions may help during implementation. However, it may be that the hierarchy is not maintained in the implementation. Frequently, the end of the design is really just a single type, the last subtype invented. In other words, there is no desire to actually implement the program by having a separate module for each supertype. Even so, however, the distinctions remain useful after implementation, because the effects of specification changes can still be localized, even if implementation changes cannot.

The second use of subtypes is for "generics." The designer may recognize that, in the system, there will be several data abstractions that will be similar, but different. The differences represent different variants of the same general idea, where the subtypes may all have the same set of operations, or some of them may extend the supertype. An example is the generalized abstract device mentioned earlier. To accommodate generics in design, the designer introduces the supertype at the time the whole set of types is conceived, and then introduces the subtypes as they are needed later in design.

With generics, the intention is that objects belonging to the different subtypes will actually be used by the program when it runs. There may also be parts of the program that use the supertype. For example, some parts of the program use the printer, while others use the terminal; still other parts simply use abstract devices. There are two reasons why such a structure is worthwhile. First, each subtype, and the parts of the program that use it, is independent of all other subtypes, and thus is unaffected by errors in the other subtypes, or by errors in additional new subtypes. Second, the part of the program that uses the supertype is unaffected by (1) the addition and removal of subtypes, (2) the differences between the subtypes and, (3) the errors in subtype interfaces. For example, if a new subtype of the abstract device supertype is added, then the part of the program that uses either the other subtypes, or the supertype, will be unaffected.

A subtype hierarchy arises in two different ways, as is illustrated in Figure A-6. In the first, T was introduced early in the design and is used by P; P also uses Q and R, and they in turn use subtypes S_1 and S_2. P only uses operations belonging to T; Q and R use the additional operations of S_1 and S_2, respectively. In the second example, T was introduced later in the design, after types S_1 and S_2 had already been defined. The need for T was recognized because of the desire to define a procedure G that works on both S_1 and S_2 but depends only on some small common part of these two types. For example, G might be a sort routine that relies on its argument "collection" to allow it to fetch elements, and relies on the element type itself to provide a "<" operation. This second use of hierarchy is more problematical than the first, since it is unlikely that S_1 and S_2 have exactly the operations required by G. If they do not, then at the time T is introduced, it is likely that the interfaces of the two subtypes will need to be changed, which may in turn cause changes in other parts of the design.

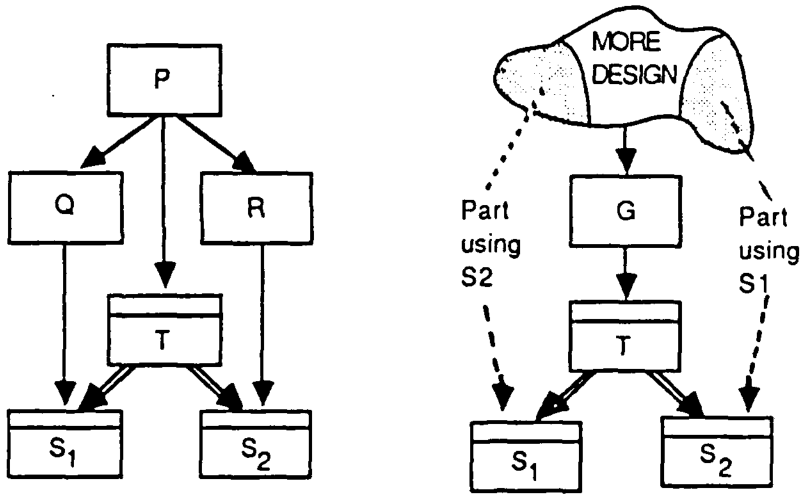


Figure A-6. Generics

In programs with these structures, we probably do want to use subclasses as an implementation mechanism. One reason is that this permits us to implement, just once, whatever can be done in a subtype-independent way. The other reason is that, for languages that have inheritance, this will permit us to implement the procedure that uses the supertype, e.g., P or G, in terms of the supertype, and later to use it at runtime with an object of one of the subtypes.

A.3.5 Implementing Hierarchy with Inheritance

An inheritance mechanism can be used to implement a subtype hierarchy in a fairly direct way. For example, there would be a class to implement the supertype and another class to implement each subtype. The class implementing a subtype would declare that the supertype's class was its superclass.

However, there are some problems with this technique, so it must be used with restraint. First, there will be trouble if the subclass accesses the rep of the superclass. As discussed earlier, this violates encapsulation, and means that we cannot reimplement the supertype without having to reimplement the subtype too.

The second problem concerns multiple implementations. It is often useful to have multiple implementations of the same type, for use in different programs or even in the same program. For example, for some matrices we use a sparse representation, and for others a nonsparse representation.

Languages with inheritance don't support the concept of multiple implementations properly because they don't distinguish between an abstraction and its implementation. Instead a class that implements a type, such as matrix, is considered to be that abstraction. Furthermore, if a second class that implements the type were provided (e.g., providing the nonsparse implementation), it would eliminate the first one. A language like Ada is in much better shape here because of the distinction it makes between private and public parts of modules. Many private parts can be associated with a single public part, i.e., there can be many implementations of the same abstraction.

Object-oriented languages allow users to simulate multiple implementations with inheritance. Each implementation is implemented as a subclass of another class that implements the type. This latter class would probably be "virtual", i.e., it would not really contain any code, but would stand as a place holder in the hierarchy. For example, there would be a "virtual" class implementing matrices, and subclasses implementing sparse and nonsparse matrices.

Using inheritance in this way allows us to have several implementations of the same type in use within the same program, but it does interfere with type hierarchy. For example, suppose we invent a real subtype of matrices called "extended matrices." We would like to implement extended matrices with a class that inherits from matrices rather than from a particular implementation of matrices, but this is not possible. Instead, the extended matrices class must explicitly state in its program text that it is a subclass of sparse or nonsparse matrices.

The problem arises directly from the lack of distinction between a type and its implementations. What we really want is two types (matrix and extended matrix), one of which is a subtype of the other. Furthermore, each type can have several implementations, and the implementations of the subtype should be combinable with those of the supertype in various ways.

A.3.6 Remarks

We have shown that subtype hierarchy enhances data abstraction in two main ways: by allowing the capture of a finer grain of design information, and by supporting generics. There is one other way that hierarchy is sometimes useful, and this is to aid in the organization of a type library.

It has long been recognized that programming is more effective if it can be done in a context that encourages the reuse of program modules implemented by others. However, for such a context to be usable, it must be possible to navigate it easily to determine whether the desired modules exist. Hierarchy is useful as a way of organizing a program library to make searching easier, especially when combined with the kind of browsing tools present, e.g., in the SmallTalk environment.

Hierarchy allows "related" types to be grouped together. Thus, if a user wants a particular kind of "collection" abstraction, there is a good chance that the desired one, if it exists at all, can be found with the other collections. The hierarchy in use is either a subtype hierarchy, or almost a subtype hierarchy (i.e., a subtype differs from an extension of the supertype in a fairly minor way). The point is that types are grouped based on their behavior rather than how they are used to implement one another.

The search for collections in general, or numeric types, or whatever, is aided by two things. The first, is considering the entire library as growing from a single root or roots, and providing a browsing tool that allows the user to move around in the hierarchy. The second, is a wise choice of names for the major categories, so that a user can recognize that "collection" is the part of the hierarchy of interest.

Using hierarchy as a way of organizing a library is a good idea, but need not be coupled with a subclass mechanism in a programming language. Instead, the notion of related types could be supported by an interactive system that supports construction and browsing of the library.

A.4 OBJECTS AND PARALLELISM

In the preceding sections we have talked about how objects can be used to make program development more effective. Now we go on to consider how they are useful as a model of computation that supports distributed and parallel programming. We consider distribution and concurrency together because they are closely related. For example, we may want to position a part of a program at a particular computer node in a network so it can truly run in parallel with other parts, or so that it can take advantage of special hardware at that node.

An object is an entity that consists of both data and code. The data is the rep of the object; the code implements the object's operations. With encapsulation we know furthermore that it is not possible for the code of one object to access the rep of some other object directly.

Such an object is a good candidate to locate, as a unit, to some node in a distributed network. Since its code accesses its data, it is efficient to locate both code and data at the same place. Furthermore, the object's node can be different from the nodes where its users reside, since they cannot access its data. Instead they must call its operations, and such calls can be implemented by sending call messages over a network, and later sending back reply messages. Provided the programmer is careful in how he or she designs the system, such an organization can work efficiently. Most likely the objects are relatively large and communicate with one another relatively infrequently.

Objects are also a good candidate for concurrency. Both specifications and encapsulation are critical here. Specifications give freedom to implementors, while shielding users from details.

Thus, an implementor can provide a highly concurrent implementation; as long as it meets the specification, the user will be satisfied. Furthermore, to understand the using program, we need only consider the specifications and need not be concerned with what is probably a very complex implementation. The effect of this complexity is localized.

Encapsulation is critical because it means that the implementor need not worry about interference from other parts of the program. Concurrent programs can be very difficult to understand, but this difficulty is greatly reduced if the reasoning is local. Then, only the code written for that object need be considered. Therefore, the invariants that are the basis for why a particular implementation works can be relied on with confidence.

Finally, objects are often a rich source of concurrency. Specifications typically say as little as possible, thus allowing freedom for the implementor. This freedom translates into an opportunity for applying concurrency to the implementation.

The desire for distribution and concurrency impose a number of requirements on a programming language. These are discussed below.

1. In implementing a concurrent object, it is necessary to write a concurrent program. This implies that the programming language provide a facility for having many processes running inside the object at the same time.

   For example, it might provide a construct that allows several sub-processes to be started, such as:

```
co-enter
S1
S2
...
end
```

Here each statement S_i runs as a separate process; the co-enter terminates when all are finished, or there may be a way of terminating it earlier.

One possibility is simply an asynchronous process fork. In this case, the process doing the fork and the new process run concurrently, and the forking process can determine when the new process is done by doing a join. A particularly elegant form of forking is the "future" mechanism of Multilisp [Halstead85] or the "promise" mechanism of Argus [Liskov87]. Here, a procedure call creates a process to run the call and then returns immediately, returning an object whose value is not yet filled in. When the forked procedure returns, the value is filled in. When the forking process needs the value, it "claims" it, waiting if necessary.

2. If one object makes a remote call to another object, it should not be blocked until the call returns. This requirement can be satisfied by the concurrency mechanisms mentioned above; the promise mechanism is particularly useful here. It can also be satisfied by using a lower level form of remote call in which the caller explicitly sends a message, and later explicitly waits for the reply.

3. An object needs the ability to run more than one remote call at once. At the least, it needs to be able to accept a new call if it is unable to complete the current one (e.g., because a needed resource is locked) or if the current call is delayed while some nested remote call runs. In the former case, accepting a new call may be necessary to avoid deadlock (e.g., the new call unlocks the resource needed by the blocked call.) In the latter case, accepting a new call can prevent the object from becoming a bottleneck. The best way to satisfy this requirement is to allow each call to start up a process when it arrives. An alternative technique is explicit task swapping: the called object queues the current call for later processing and proceeds to a new call. This technique works but is less satisfactory because the code of the called object is much more complicated. Also, it assumes the called object runs on a single processor, which may not be the case.

4. The programmer needs a way to control the scheduling of processes. In the case of a program that must support hard real-time constraints, some form of deadline scheduling is needed. In addition, it may be necessary to have some priority mechanism to distinguish between processes without hard deadlines, since there may be more processes than available processors.

5. The programmer needs a way to control what a remote object consists of. Typically, the rep of an object consists of other objects, e.g., an array is used to represent a set. For the object to run well, however, it is generally necessary for most of the representing objects to reside at its node. Otherwise, every access

to the rep will require a remote call, which can lead to a major performance degradation, since remote calls today are ten to one hundred times slower than local calls. On the other hand, some representing objects may need to be remote, for the reasons mentioned above. Therefore, the programmer needs a way of controlling which representing objects are local and which are remote.

6. The programmer needs to control where an object (and its representing components) resides in a network. The object may be positioned at a location to take advantage of special hardware, or it may be positioned to be near its users.

7. There must be a way of moving objects without requiring changes to user code. An object might be moved because performance analysis indicates a better place for it to reside, or to balance load at various locations, or to take advantage of new hardware in the network. When such a move happens, the code of using programs should not need to change. Satisfying this requirement implies that the way users make remote calls must be location-independent. It also implies that there is a way to map object names to locations at runtime.

8. The runtime system of the programming language needs to be extensible, so that programs can take advantage of and adapt to changes in hardware configurations and applications requirements. In particular, it must be possible to create new objects, and also to remove objects.

9. It is desirable to be able to replace a group of objects (possibly consisting of a single object) by a new group that provides the same behavior in a different way, i.e., the new group is a reimplementation of the old. The replacement should be done in a way that results in no loss of information so that users of the original group can continue using the new one without problems. In addition, the replacement should cause very little delay and certainly should not require bringing down the entire system.

Many of the requirements above are satisfied by at least some languages, e.g., Argus [Liskov83] and, to a large extent, Ada. However, some require research to determine the right solutions. This is especially true for the scheduling and replacement requirements. A partial solution to the latter is described in [Bloom83].

A.5 IMPLEMENTATION LANGUAGES

This section addresses the issue of what programming language would be suitable for use in the kinds of projects of interest to us. We assume here that type hierarchy will be desirable at least occasionally. In addition, however, the language must satisfy our concurrency and distribution goals. We conclude that object-oriented languages such as SmallTalk are not suitable for our use. We then discuss the suitability of Ada. Ada meets many of our concurrency goals, but has no inheritance mechanism, so we go on to discuss how to overcome this weakness when subtypes are needed.

A.5.1 Object-Oriented Languages

Most, if not all, object-oriented languages do not support our concurrency and distribution requirements. For example, concurrent SmallTalk [Yokote86] fails every one of our requirements. Other object-oriented languages may do better here, e.g., Actors [Hewitt79], but they are research vehicles unsuitable for use in our projects.

An object-oriented language such as SmallTalk does provide direct support for inheritance. However, SmallTalk's mechanism is deficient in two important ways. First, it enforces data abstraction only with runtime type-checking. This is unsatisfactory for production programs for reasons discussed earlier in the paper. Second, SmallTalk does not enforce encapsulation completely, since subclasses can violate the encapsulation of their superclasses.

The inheritance mechanisms of most other object-oriented languages are even less satisfactory than SmallTalk because they do not enforce encapsulation at all. One exception is Trellis/Owl [Shaffert86]. This is a strongly typed, Object-oriented language developed by Digital. However, it is still an experimental language and is probably not appropriate for use outside of Digital.

Another problem with SmallTalk is that it has only classes and no procedure mechanism. (Other object-oriented languages do not necessarily have this problem.) There is no way to implement directly a procedural abstraction that accomplishes some task. Instead such a procedure must be turned into a psuedo-object. Another related problem is that methods are not "first class" objects, and in particular, they cannot be passed as arguments themselves, an ability that is sometimes needed. Procedures are an important kind of abstraction and a programming language should support them directly.

A.5.2 Ada

Ada supports most of our requirements for concurrency and distribution. It also supports data abstraction. Furthermore, it is defined for use in the development of production programs. Therefore, it seems an appropriate choice for us. It is true that Ada does not support type hierarchy. However, it is possible to simulate type hierarchy as explained further below.

The unit of distribution in Ada is the task. This is an object that provides a group of operations for others to use through the "entry" mechanism. Tasks are somewhat limited in their support for concurrent execution of entry calls, as discussed in [Liskov86b], but are generally satisfactory, and meet many of the other goals.

A task contains data that correspond to its representation. Some of these data may be other tasks, but most are objects local to the task. The representation of a task is not encapsulated, so encapsulation here must be achieved through programming conventions. The local data of the task, however, can be abstract objects whose reps are encapsulated. These abstract types are defined by using the Ada's private type mechanism.

We do not believe that the absence of support for hierarchy in Ada is an overwhelming disadvantage. Language support for type hierarchy is needed only for programs in which objects of different subtypes exist at runtime, and furthermore where there are parts of the program that use objects of various subtypes as if they belong to the supertype. We believe such cases are relatively rare. Nevertheless, they do occur occasionally. Sometimes the part of the program that uses the supertype can be implemented using Ada's generic mechanism to implement polymorphic abstractions. Otherwise, type hierarchy must be simulated.

In the remainder of this section, we discuss when polymorphic abstractions are adequate, how to implement types with multiple representations, and how to implement a subtype hierarchy. Finally, we discuss some requirements for the program development environment.

A.5.3 Polymorphic Abstractions

In this section we discuss an alternative to subclasses as a way of achieving some of the benefits of hierarchy. This mechanism is the polymorphic abstraction. A polymorphic abstraction is one that can be parameterized by types.

For example, consider a procedure that does sorting. In many languages, such a procedure would be implemented to work on an array of integers; later, if we needed to sort an array of strings, another procedure would be needed. This is unfortunate. The idea of sorting is independent of the particular type of element in the array, provided that it is possible to compare the elements to determine which ones are smaller than which other ones. We ought to be able to implement one sort procedure that works for all such types.

Some languages allow such procedures to be implemented by allowing procedures to be parameterized by types. Here is the header of a sort procedure in CLU:

```
sort = proc [T: type] (a: array[T])
         where T has lt: proctype (T, T) returns (bool)
```

This header constrains parameter T to be a type with an operation named "lt" with the indicated signature; a specification of sort would explain how lt must behave. We could even define a more polymorphic sort routine in CLU that would work for all collections that are "array-like", i.e., whose elements can be fetched

and stored by index.
The sort procedure is similar to the generics discussed earlier. There we defined a hierarchy in which the supertype T was "sortable" and the sort routine took as an argument an array of T. Thus polymorphic procedures allow us to group types that have operations and behavior in common, and in this way we approximate the effect of hierarchy. In effect, the polymorphic procedure operates on the supertype, but without a need to define the supertype by hierarchy.

There are cases in which using polymorphic procedures is superior to using hierarchy. In particular, polymorphism allows us to relate types without having to maintain a hierarchy. If hierarchy is used for every such procedure, the type universe can become very complicated, and it will be troublesome to consider, for each new type, what all the supertypes are. Secondly, if all parts of the hierarchy are not done properly in the first place, it can be difficult to fix things up after the fact. For example, suppose that a new generic G is invented after many types already exist, and that G requires an operation that was not thought of before. Then in order to use G with some existing type S, it is necessary to add hierarchy above S so it will have the operation. (It is undesirable to modify S itself, since this increases the code that must be changed when S is implemented or reimplemented.) For example, suppose G required two operations O_1 and O_2, and that we wanted to use G with two types S_1 and S_2, and that neither of these types had these operations. Then we would need to invent the hierarchy shown in Figure A-7. The idea here is that the operations are implemented differently for the two types, so they cannot both be implemented in T. Therefore, T implements them for S_1 and R implements them for S_2. (We avoid implementing the operations inside the S_i's because the modularity is better if they are outside. Of course, implementing them outside presupposes that they can be implemented using the subtype's operations.)

There are even situations in which G cannot be made to work with some type S without changing S's implementation. This happens whenever S already has an operation of the required name, e.g., O_1, but with different behavior. Polymorphic abstractions, as defined in Ada, can avoid such problems.

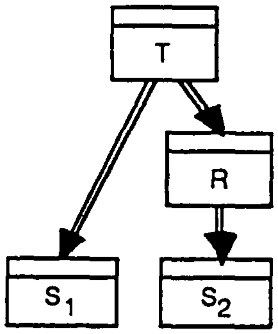


Figure A-7: Adding Hierarchy

A.5.4 Implementing Type Hierarchy in Ada

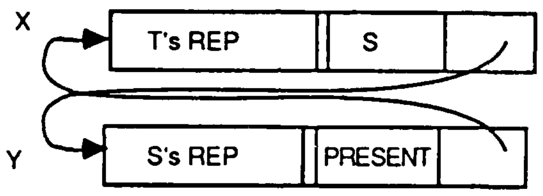
There are uses of type hierarchy that cannot be achieved through polymorphic abstractions. The most important example is a data structure that contains, as elements, objects of various subtypes. For example, imagine a directory that maps strings to abstract devices; the string "printer" might map to an abstract printer, while the string "display" might map to a terminal. Ideally, the directory would be of type map[T] where T is the abstract display type. Such a structure is not possible in Ada without simulation of type hierarchy.

There are two main ways to proceed. One is to store variant objects in the directory, where the tag of the variant indicates which subtype the particular element belongs to. Thus, the type of the directory is map[V] where V is the variant type. This solution is simple, and is probably the best one in some cases. However, it has a major flaw: If a new subtype is added to the system, it is necessary to find every program that uses the data structure and change it. For example, we would need to modify the implementation of the directory, and also every program that uses the directory, since the type of the directory has changed. Thus, a change in the type hierarchy requires modification of the using code that really ought to be unaffected by the change.

The alternative solution has some implementation complexity but avoids the problem. The idea is to implement the supertype as well as the subtypes, and maintain the relationship between the supertype and its subtypes at runtime within their code. In this solution we really have a directory of type map[T]. If a new subtype is added, the supertype code will need to be changed, but code of programs that use the supertype will not. It is much easier to remember to change the supertype in this case than to find all places in user code.

A method for implementing the hierarchy works as follows. The technique provides encapsulation for the super- and subtype implementations; for example, each might be implemented in its own package. In addition, the subtypes are independent of one another and are not affected by the addition of a new subtype; only the supertype must change.

1. Recall that the rep of a subtype object contains both its own rep and also the rep of a supertype object that is created as part of creating it. We can simulate this by having each subtype object refer to the supertype object in its rep, and also having the supertype object refer to the subtype object, as illustrated:



Here x is an object of the supertype T, and y is an object of subtype S. The rep of supertype object x is a variant record; the tag of the variant indicates what kind of subtype object is referenced. Furthermore, the subtype object y's rep must be a variant that is either "null" (indicating the reference to its supertype object has not yet been stored) or "present," indicating that the reference has been stored.

2. The supertype provides a creation operation per subtype for creating the supertype object. For example, if supertype T has subtypes S1 and S2, then it has two creation operations:

```
make_S1 (S1, ...) returns (T)
make_S2 (S2, ...) returns (T)
```

   Here "..." stands for other arguments that may be needed to initialize the new supertype object. The returned object refers to the subtype object passed as an argument to the "make" operation and has the tag for that subtype.

3. A new subtype object is created (by a creation operation of the subtype) as follows. First, the variant record is created in the null state, and the subtype's fields are filled in. Then the make_S operation (where S is the subtype's name) of the supertype is called to obtain the supertype object. Finally, the subtype object is modified to have the "present" tag and to refer to the returned supertype object.

4. The subtype provides an operation to return its supertype object:

```
get_super (S1) returns (T)
```

   This operation would be used, e.g., before storing the abstract printer into the directory (since the directory contains objects of the supertype abstract devices, rather than objects of the subtypes). The supertype provides a similar ability, with one such operation per subtype, e.g., get_S1 (T) returns (S1) signals (wrong_type)

This operation would be used by a program that fetched an object from a directory and wanted to use it according to its subtype. Typically, such a program would know what subtype the object was (e.g., it looked up "printer"); the exception is signaled if it is mistaken about this.

5. Each supertype operation that is redefined by subtypes must check the subtype of the object passed in as an argument, and call the operation provided by that subtype. Furthermore, if the supertype operation op that really does something (i.e., it is not virtual), we must provide another supertype operation, op_hidden, that can be called by subtypes when they need to run the supertype operation.

It should be noted that these rules are simply an implementation of what an inheritance mechanism does. For example, whenever a subtype really calls on a supertype operation, a private way is created for it to do this, similar to the op_hidden approach mentioned above.

Implementing types like this is not very attractive. However, the transformations are all straightforward, so that it should be possible to define a preprocessor that hides the details and in essence appears to add hierarchy to Ada.

A.5.5 Multiple Implementations of a Type

Ada already supports multiple implementations of a type in the program development system, but it allows only a single implementation to be used within a single program. Sometimes it is useful to have simultaneous implementations of the same type in use within the same program. For example, we might want to use both sparse and nonsparse implementations for matrices in this way. This ability can also be provided by a mechanism similar to that above. However, the problem is simpler because the various implementations are not visible to users. Instead, all user variables are of the type, e.g., of type matrix; thus there is no need for operations that convert from the type to an implementation or vice versa. The one possible exception to this is when objects are created; at this point we need to know what implementation to use. This ability can be created by binding when the part of the program that uses a particular implementation is compiled or linked.

As was the case above, there would be one implementation of the supertype and one for each subtype. The difference is that all calls by users would go to the supertype, which would then dispatch to the appropriate subtype operation. To permit this dispatching, the rep of the supertype would have a variant part identifying which subtype a particular object belonged to. The rep would also have additional fields to store the rep of the supertype.

Although the subtype implementations are independent, they may need to be written with the knowledge that there are many of them. This is true if the type has any "binary" operations, i.e., operations taking in two (or more) arguments of the type. An example of such an operation is adding two matrices. The problem is, if one matrix has the sparse representation and the other has the nonsparse representation, what should be done? (This is not a new problem; it arises in object-oriented languages all the time). One possibility is for the type to call the operation belonging to implementation of the first argument. This operation must access the second argument indirectly by calling its operations. For example, suppose we add two matrices m1 and m2, and m1 is sparse while m2 is not. Then the matrix add operation would call the sparse add operation, which would access its second argument m2 by fetching each <i, j> element using the fetch operation of the nonsparse implementation, rather than looking directly at the rep of m2.

A.5.6 The Program Development Environment

In this section we discuss briefly some requirements on the program development environment. These requirements are concerned with providing the proper tradeoff between efficiency and performance.

Using data abstraction can be expensive because of the extra calls involved. For example, if set is implemented using a list, then the size operation for set simply calls the size operation for list. Keeping the call in there aids flexibility: if list is reimplemented we can simply bind set to the new implementation without necessarily recompiling it.

The performance of a data abstraction can be improved at the cost of flexibility by straightforward optimization techniques. The key technique is that of "inline substitution," which means that the code of a called procedure is placed inline at the point of the call. For example, the set size operation would contain within it the code of the list size operation. In this way the extra procedure call is avoided. Once inline substitution has been done, other optimizations that cross abstraction boundaries, such as dead or redundant code elimination, are also possible, which can further improve performance. Of course, when the inline optimization is done, we lose the ability to change the implementation of the used type without changing the implementation of the using code.

Similar kinds of optimizations can be done in object-oriented languages. For example, if S inherits from T and in particular operation o comes from T, then there are two possible implementation techniques. The first is to look for O at runtime. When a call to O is made, the runtime system first checks the operations explicitly provided for S to see if O is there; if not it looks in S's superclass T. This technique is inefficient but flexible, since it means we can reimplement T (assuming S did not violate encapsulation) and later use it as part of S without having to change S's code at all. The alternative technique is to provide code for S that includes T's operations such as O. This avoids the search at runtime, but means S must be recompiled to make use of the new implementation of T.

At certain points in the lifetime of a program, flexibility is the most important goal, e.g., during debugging. Later, when the program enters production, good performance may become the overriding factor. What is needed from a program development system is the ability to support both needs, but under programmer control. In particular, it should be possible to compile a call using either inline substitution or not, depending on what the programmer wants.

If the program development system does not provide inline substitution, then programmers in need of improved performance will start to do it by hand. This is not a good way to proceed, however, because later when modifications need to be made, the abstraction boundaries that would have made modification easy have disappeared. By contrast, when the system does the work, the modifications are made to just the affected abstractions. Later, inline substitution and other optimizations can be done again to achieve needed performance.

The above discussion has focused on how modifications in code can be accommodated during stages of program development. However, the ease of producing the new code is only part of what is needed. We also need to install the new code in the running program. This is easy to do if we stop the running program, replace it with the new implementation, and start it again, provided that the program does not have any objects, e.g., in files, that make use of older representations. Dealing with such objects requires more sophisticated techniques.

APPENDIX B -- MULTIPROCESSOR OPERATING SYSTEM ISSUES

This appendix discusses a variety of issues related to multiprocessor operating systems. These issues all have some bearing on the operating environment for parallel programs.

B.1 VARIATIONS ON MULTIPROCESSOR UNIX

B.1.1 General Considerations

Two choices of strategy are possible for construction of a multiprocessor operating system: a) design from scratch, or b) adapt existing code for new hardware [Janssens86]. The later approach takes advantage of software available for the existing Operating System (O/S), often a larger concern than optimal use of special hardware features.

Users want to preserve applications already running under Unix, or to migrate to Unix to avoid the need to rewrite again in the future. Over the past five years, multiprocessing versions of Unix have been developed by numerous universities and computer vendors [Hurst87]. The hardware configuration most often selected is a tightly coupled multiprocessor system with shared memory.

Unix is constructed as nested layers of services, which are thought of as concentric shells. Conceptually, each layer calls only the services offered by the next lower level (in practice, minor violations of this convention occur for efficiency). The layer responsible for multitasking, i.e., interleaved scheduling of user programs is called the kernel. A kernel directly controls the hardware of its host, managing I/O, interprocess communication and the file system in addition to scheduling tasks.

An operating system is perceived by an end user as a command interpreter and a collection of system functions. Portability of existing software is enhanced if system service calling conventions are unchanged between "different" operating systems. Unix (and its derivatives) have a few features which cause problems when adapting it to suit a multiprocessor environment [Janssens86].

The essence of these problems is the protection mechanism required to control access to shared resources during execution of "critical sections" of instructions. Execution of critical sections (in two or more competing processes) must be mutually exclusive in time, to maintain consistency of shared variables, in-core tables, disk files and the like.

In uniprocessor Unix implementations, resource competition exists only among user processes. A process is defined to be an execution of a program (which may spawn more than a single task). Distinction is made between "user mode," in which processes are restricted from potentially system-crashing actions, and "kernel mode" in which a process can do whatever the hardware permits.

Several processes can exist simultaneously in either user or kernel mode, however, transfer of control between user processes is random (beyond control of a user process), whereas a kernel process can be replaced by another kernel process only at its own request. Not even an interrupt causes a kernel process switch. If a hardware interrupt occurs during execution of a kernel process, only interrupt processing can take place; calls to the scheduler and transitions out of the interrupt are not allowed. This complicates matters only slightly on a uniprocessor.

Two user processes can synchronize their access to a shared variable by use of "lock" and "unlock" functions which ensure that only one process at a time enters its critical section. In the absence of interrupts, kernel processes require no explicit protection of their critical sections, since a kernel process can be switched out only when it explicitly calls the scheduler.

A problem does arise on a uniprocessor when a variable is shared between a kernel process and an interrupt handler. Interrupts must be disabled before entering the critical section, and re-enabled upon exiting from it. Despite this, mutual exclusion remains only a minor annoyance so long as a kernel process does not attempt to call the scheduler within a critical section.

Protection of a critical section that calls the scheduler in a kernel process requires a complicated arrangement of flags and busy-wait guard conditions, but is again possible. However, existing Unix protection mechanisms are not sufficient in a configuration containing two or more independent processors.

With shared memory, concurrency at a hardware level permits two kernel processes to test a flag roughly simultaneously, in which case both might enter their critical sections at the same time. Indivisibility of a kernel process makes detection of the beginning of a critical section fairly easy (indeed, no special provisions are required up to the point at which a scheduler call occurs). Determining where a critical section ends can be much harder, if not impossible, without explicit line-by-line review of the operating system kernel source code.

Adaptation of Unix to a multiprocessor environment requires active detection of critical sections, rather than modification of existing protection mechanisms. Review of kernel source code is encumbered by a) its lack of structure, b) subtle differences in superficially similar functions arising from optimizations, c) nesting of critical sections (producing deadlock), and d) multiple execution paths that return repeatedly to a given section of kernel code.

Thus, implementation of a multiprocessing Unix requires one of three approaches. The most direct simply examines every line of code in the complete kernel, adding locks where needed. This tends to focus attention on details rather than issues of global design. A second solution requires that only a single processor be allowed to execute kernel processes at any given time. [Meyer75] is an early example of this approach. However, a uniprocessor Unix spends about half its time executing kernel code, so this imposes a significant serialization constraint, with corresponding loss of parallel performance.

A third approach is to throw away the existing kernel code and start over. This requires significant development effort, but is the approach taken in [Barak85] and [Emrath85]. A much less drastic modification is possible with the asymmetric processor configuration of [Tuyenman86].

B.1.2 MOS

[Barak85] describes a multiprocessor operating system that mimics Unix, operational since 1983. A true distributed implementation, MOS requires a symmetrical configuration of similar processors (all PDP-11) connected via a fast local area network. The operating system integrates the independent processors into a single virtual machine, which permits process migration across hosts. MOS replaces the entire Unix kernel with an equivalent one that supports all (Version 7) system calls. Users need not be aware of the existence of multiple processors or the LAN; the Unix process interface is preserved. Dynamic configuration and replication of storage exploit hardware redundancy to increase system availability. However, few provisions are made for fault tolerance apart from tracing a fault to its physical machine (and subsequent testing in stand-alone mode).

Each machine in a MOS configuration runs an identical kernel that provides the standard Unix environment. Control is decentralized, with each machine making independent decisions and performing its own local housekeeping. Unnecessary communication is avoided when accessing strictly local objects, but machines validate incoming requests before servicing them. Granularity of remote operations is coarse, so that objects cannot be left in a locked or inconsistent state.

Tools exist to evacuate all processes from a given machine, when it must be disconnected from the configuration. When a machine crashes, minimal disruption of other processors ensues. Dynamic load balancing is also supported via process migration.

Each individual object (e.g., file or directory) resides on exactly one computer. The file system is implemented as several disjoint trees, each of which constitutes a Unix file system complete with root directory. These stubs are connected by a "super-root" directory that assigns symbolic names to each of the roots. Each machine keeps its own copy of the super-root (which must be kept identical with the others, but not by the kernel).

The kernel is structured as an object manager, where an object is any system entity having an independent existence (i.e., processes, files, pipes, terminals, memory segments, disks, etc). Objects reside near their physical manifestations, and are mostly passive. The sole exceptions to this in MOS are processes, the only objects that can change the state of other objects. All objects visible to a user can be operated on by remote processes.

The MOS kernel is composed of three subsystems: lower kernel, linker and upper kernel. The lower kernel implements objects that reside in a given machine, and is the only part of the kernel with full knowledge of local objects. It multiplexes processor time and memory among processes and provides synchronization and other services to the upper kernel.

The linker allows the upper kernel to call procedures in the lower kernel of any machine in a MOS configuration. The linker serves as a messenger. The upper kernel knows where objects reside, but calls the linker to dispatch requests. The linker is the only subsystem with current knowledge of the network configuration, and is the only part sensitive to machine ID.

The upper kernel is a logical extension of a user's program. Its duty is to service system calls and maintain the process environment. Its address space is an extension of the user's, and services only the user's program. A user program can freely change its own contents, but must call the upper kernel to act on external system-wide objects. A process running at this level can be transferred to another physical MOS processor and resumed without any side-effect (hence migration becomes trivial).

MOS makes a cluster of loosely coupled independent computers behave as though it were a uniprocessor Unix system. Internal structure of the kernel is modular, with a high degree of information hiding and hierarchical organization.

B.1.3 Asymmetric Power for Real-Time Data Reduction

Tuyenman and Hertzberger [Tuyenman86] discuss a distributed real-time operating system that runs on an embedded multiprocessor. Fados is based on an asymmetric host-target configuration with communication between arbitrary processes on host and target machines. The FAMP system is used to partially reduce data in real time for high-energy physics experiments on a time scale from a few tenths of a millisecond to about ten msec. The partial analysis determines only whether data should be preserved for more thorough examination.

Both host and target processors are of the Motorola 68000 family. This permits object-code compatibility between host and target, even though program development is supported only on the host. Time-sharing is not provided on target processors, to avoid overhead of scheduling and provision of a user interface. The authors consider a direct interface with the user of a real-time system neither desirable nor necessary.

Features that are needed are a) easy portability of the Operating System (O/S) to different target machines; b) monitoring and operation of the target from a remote time-sharing host; c) access to file systems on the host from programs running on target machines; d) support for high-level languages; e) remote symbolic debugging; and f) fast communication between host and target.

High-level language support is provided via calls to a ROM-resident nucleus on target processors, which supports interprocess communication and interrupt handling by (remote) procedure calls. Each target nucleus, together with a collection of processes resident on the host machine, performs functions usually assigned to the kernel level of a uniprocessor Unix.

Interaction between host and target occurs in three areas: (1) code generated on the host is transferred to the target; (2) programs on each target have access to the file system of the host; (3) for symbolic debugging, interpretation of memory images and symbol tables of target processors is performed on the host. Usefulness of memory allocation in an environment that lacks hardware memory management is negligible, so Fados omits memory allocation.

Each target processor executes a nucleus for process communication and interrupt transfer, together with processes to load user processes into memory and handle break-points. The host provides a command interpreter, a debugger and a file server. Communication occurs via message passing only, since on the FAMP system memory is shared only between an individual target and the host processor (not among multiple targets).

Interrupts are regarded as a remote procedure call emanating from an external device, similar to the philosophy in Ada. A process which handles interrupts must have a priority equal to or greater than the priority of interrupts it handles. Fados's scheduling algorithm is trivial; each process executes until it suspends itself or another process of higher priority wakes up.

Process loading occurs in two phases, to avoid deadlock that could result from communication with non-existent destinations. First, all processes are made known to each of the target nuclei; only then are process addresses passed and execution commenced. This avoids consideration of how long each process takes to load. The Unix host obtains a list of executable files, and for each process to be loaded, sends a 'load process' message to the appropriate target. After all processes have been started and are awaiting a message, the host command interpreter forwards the addresses of other processes and descriptors of open files.

The Fados target debug monitor can start or stop a process, notify the host when a breakpoint is reached, return status data, and read or write data in the target local memory (similar to ptrace in conventional Unix). The message oriented communication assumes messages need be sent only once every 5 to 10 msec (the overhead involved with more frequent reports is excessive).

Fados is an easily extended real-time operating system that operates on simple hardware. In combination with a Unix host the system is user-friendly and easily portable to different target hardware. Source code to be transported is modest (800 lines) and written in a high-level language.

B.1.4 Parallel 4.2 BSD

Several universities and computer vendors have developed multiprocessing operating systems based on the 4.2 BSD version of Unix. Required modifications include extensions to C language compilers for support of interprocessor synchronization, and to Unix system calls for multiprocessing within individual programs.

Xylem [Emrath85], an operating system developed for the Cedar computer at the University of Illinois (Alliant FX/8 clusters), and Concentrix [Test87], Alliant's proprietary Operating System (O/S), are typical of shared-memory symmetric multiprocessing BSD 4.2 Unix designs.

Each processor runs a separate copy of the Xylem kernel and coordinates its activities with others through shared memory. Memory management is supported via the page mapping hardware of individual processors. Virtual address space is shared among the processors, and any single processor that decides to replace one of the global memory pages must notify every other processor that is using the page to mark it as invalid. Tasks that share a local page must execute in the same processor cluster (i.e., on the same bus/backplane).

Xylem task scheduling occurs at two levels. Within a single processor a conventional scheduling algorithm manages a queue of ready tasks kept in global memory. If a processor becomes idle, a multi-processor scheduler is invoked to compare a queue of ready processors with the process ready queue. (Recall that a "Unix" process may consist of multiple tasks.) To assign all ready tasks of a process simultaneously, a multi-processor scheduler allocates several idle processors at the same time.

Although Alliant distinguishes interactive processors (IPs) from computational elements (CEs), the Concentrix Operating System (O/S) also runs "symmetrically" on all processors in a system [Test87]. Execution in kernel mode differs only with respect to device interrupt code, which is run exclusively on the IPs. Each IP is scheduled independently; whereas, all CEs are scheduled as a single unit. A process that requires vectorization or concurrency can be scheduled only on the CEs.

Ordinary Unix (on a uniprocessor) synchronizes execution of "system-streams" (user and non-interrupt kernel processes) with "interrupt-streams" (kernel interrupt handling) by prohibiting preemption in kernel mode. System-streams protect their critical sections from interruption by raising their processor priority.

Concentrix has to deal with more complicated synchronization problems that result from multi-processor operation. Among these are conflicts between: a) pairs of system-streams on two separate processors; b) system- and interrupt- streams across processors; and c) pairs of interrupt-streams on two separate processors.

To resolve these, the implementors of Concentrix resorted to a combination of priority level locking and a hierarchy of global test-and-set based synchronization locking mechanisms. Source code was carefully adjusted to minimize lock holding time in critical sections.

Inter-processor communication is supported through a "cross-processor interrupt" (CPI), which can trigger remote procedure call (RPC) handlers on other processors. Hardware I/O device drivers manage a distributed shared Multibus address space via careful use of global locks and RPCs. Device drivers arrange to proceed transparently with I/O activity (buffer management, etc.) until actual work is required from a remote hardware device, at which point an asynchronous RPC is initiated.

B.2 MESSAGE-BASED PARALLEL OPERATING SYSTEMS

B.2.1 Embos

Embos, a proprietary operating system for the ELXSI 6400, was originally designed using message-passing, and subsequently modified to support shared-memory processing (in response to a customer request). Embos relies almost exclusively on messages for synchronization and passing data between processes [Olson85].

ELXSI processes communicate via explicit virtual circuits (logical point-to-point connection rather than datagram service). An outgoing connection from a process is called a link; incoming ones are collected into a "funnel". Messages on a given funnel are delivered to the consuming process in FIFO order.

Routing of messages is handled entirely in microcode; tables which define links and funnels are inaccessible to application software (insuring their integrity). The message system is fast relative to traditional minicomputer interprocess communications, but still slow compared with individual machine instructions.

Embos processes usually exchange short messages mostly concerned with control and synchronization. Bulk quantities of data tend to be transferred via I/O controllers which perform direct memory access (DMA) into virtual address space. Messages merely synchronize this data access.

Data structures in the message system cannot be stored in cache memory because they are shared among the microcoded kernels on each processor. Such main memory access is relatively expensive, but acceptable given the level of service in the message system.

Embos and application programs request service via explicit messages. Multiple copies of a resource server program can exist on one or more processors when there is more than one instance of an underlying resource. Each server handles requests at its own rate using private data structures. (Data sharing is explicit.)

Embos processes migrate freely among available processors in exactly the same way as application processes. Absence of implicitly shared data eliminates many potential problems that arise with shared-memory concurrency.

An Embos process has no dependance on a particular physical processor. Waiting for lower-level services is minimal. When an operation is blocked, most processes mark the fact on a work-in progress list and turn to the next available item in their work queue. Servers wait only when work queues are empty or when preempted by a higher priority process. Further, no assumptions are made about execution order; relative priority is not used as an implicit concurrency control method.

The inherent asynchrony of messages and avoidance of shared memory among operating system processes simplified Embos implementation and limits performance degradation as more processors are added to an existing configuration. Moreover, although little attention was paid to procedure-level parallel processing during Embos design, ELXSI was able to support the majority of shared-memory techniques with addition of a small subroutine library and a single new processor instruction (for flushing a write-back data cache local to each processor).

B.2.2 Meglos

Communications requirements for a system containing hundreds of processors differ from those of smaller distributed systems. Meglos, an operating system under development at Bell Labs, provides communications based on symmetric, independently flow-controlled channels with low latency and high throughput.

Many distributed applications can be structured as sets of processes whose communications topology is fixed, or changes only slowly over time [Gaglianello85]. Meglos processes communicate in pairs that establish communications channels between themselves before exchanging messages. This arrangement allows the Operating System (O/S) to provide flow control, so that a process can read from multiple incoming channels in the most convenient order. Without such provision, a process can be flooded with messages that it cannot digest.

Some algorithms send data repeatedly from one process to a large number of others. With hundreds of processors, overhead of opening separate channels and sending many identical messages can degrade performance of any application. A more efficient method used in Meglos provides "multicast channels" to transmit messages to numerous other processes with a single operation.

A related problem arises when a process communicates regularly with many others, but sends different data to each of its co-workers. In this case Meglos provides a "unicast channel," which resembles a multicast channel, but allows the sender to specify a separate recipient for each message.

Meglos channels can be unbuffered (desirable for some real-time applications), but the majority of programs require implicit buffering on a channel to improve performance and avoid deadlock.

Buffering at the sender's end avoids blocking of the sender while writing, and buffering at the receiver's end reduces latency when data is not read immediately after being transmitted. Throughput of a multicast channel is determined by its slowest reader.

Meglos supports buffering at either end of a channel, and allocates buffer space from a program's own address space, rather than a shared pool. This approach avoids contention for buffers and helps guarantee that applications remain deadlock-free.

Buffering is mandatory for multicast channels; without it, deadlock will always occur when a message is sent. An arbitrary number of processes can connect to a multicast channel. A message sent by any one of them is forwarded to each of the others. However, Meglos does not guarantee that all processes will read a series of messages in the same sequence from a multicast channel.

Communication in Meglos is unlayered. Application-level write requests are translated directly into link-level communication protocols by the Meglos kernel, eliminating several layers of overhead and improving throughput. Error recovery and flow control are handled by the link-level protocol.

Meglos resends a message only in response to a transmission failure. Messages with bad checksums, for unopened channels, or to non-existant processes are ignored. Message retransmission occurs when no acknowledgement is received within a time-out interval. The problem of endless retransmission following a lost acknowledgement is avoided by inclusion of a one-bit sequence number, inverted for each retransmission. When two successive messages arrive with the same sequence bit, the second message is discarded and the acknowledgement is resent.

System initialization establishes channels between kernels on individual processes and the resource manager. Subsequently, the resource manager is invoked when a channel or process is created or destroyed.

Communications rates sustainable between programs are substantially greater than are supported across multiple VAX 11/750 processors running BSD 4.2 Unix, or within a single Unix processor. For 1024-byte messages using bidrectional pairs of read and write servers, Meglos provides combined throughput of 512 kbyte/sec, compared with 160 kbyte/sec across Unix pipes in a single processor, or 33 kbyte/sec across an Ethernet link.

Meglos latency for transfer of short messages (two bytes long) between processors increases from 2 to 5 msec as the number of processors varies from 2 to 1000. Latency between a pair of Unix 11/750s is 11 msec; a comparable figure for a pair of Sun workstations is 3 msec. Note that the foregoing figures refer to actual message transfer rates rather than operating speeds of the underlying physical transfer mechanisms.

B.3 CHRYSALIS/UNIFORM SYSTEM

The Chrysalis Operating System supports an "object-oriented" model which explicitly connects data structures with specific operations that alter them [BBN86a, BBN86b]. Chrysalis understands about several kinds of objects, including processes, memory blocks and I/O devices. All objects have a unique 32-bit Object Identifier (sufficient to locate any object in the entire machine), an Object Attribute Block (descriptor), and optionally, a name.

Each processor runs a separate copy of the kernel, which schedules only its local processes. A process is the smallest entity that can be independently scheduled. A "deadline" method is used, which guarantees each process a minimum amount of run time before a specified deadline time. Design of the algorithm draws on prior experience with TOPS-20 and BBN's Pluribus system.

Other types of Chrysalis objects include events and the Dual Queue. An event is used for interprocess communications, or to control the flow of process execution. A process calls Chrysalis to create events that can be delivered to the process. An event can subsequently be posted to the process by any other process, the operating system, or I/O hardware. The process timer is one application of this mechanism. Chrysalis limits data storage for each process to a single instance of an event. Multiple postings are remembered, but data associated with all but the first is discarded.

An application that accepts data from several asynchronous processes must use an alternate mechanism, the Dual Queue. The Dual Queue is a general mechanism developed to manage the allocation of tasks to processors (but also works well for problems other than task allocation). A multiprocessor keeps a running record of tasks awaiting processors and processors that are looking for tasks to execute; yet one of these is always an empty set at a given moment of time. Consequently only a single queue is required, with a marker to indicate the type of object it contains. Use of a Dual Queue resembles mailbox services of other operating systems (e.g., DEC's VMS).

Beyond its repertoire of Unix-like system calls, Chrysalis also provides application libraries that manage system resources. The most important of these is the Uniform System library, which supports procedural language programming of BBN's Butterfly parallel processor.

Chrysalis provides all the usual mechanisms for interprocess communication, synchronization and control familiar from uniprocessor multitasking applications. The Uniform System provides functions to assist storage and processor management, the goal of keeping all memories and processors in a system equally busy.

Processes can share memory in two distinct ways. Restrictive designs isolate processes from each other by mapping memory such that only a small subset of each process address space can be accessed by other processes. The Uniform System uses a different approach, in which processes share a single large block of memory that is mapped into the address space of every process.

Applications programmers need not concern themselves with manipulation of memory maps. In addition, the Uniform System memory allocator scatters application data uniformly across all memories of the machine. This reduces memory contention, but costs 4% to 8% in execution time (and can also complicate low-level debugging).

Processor management falls naturally into two steps. First, the programmer must identify parallel structure in an algorithm, perhaps rearranging the number and kind of steps to improve its potential for distributed operation. Next, the most desirable number of concurrent operations must be determined for each activity. Explicit construction is one way of achieving this; a programmer arranges work so that all processors finish together.

The Uniform System instead encourages a dynamic approach to task assignment, which makes it unnecessary to know how long each individual step will run. Programmers are required to structure their applications into two parts: a) subroutines that actually perform work, and b) "generators" that identify the next task to be executed. A serial program embeds the generator function in its control flow; parallel programs that use the Uniform System must make explicit decisions about task sequencing.

A typical generator function consists of three procedures and a data structure. An "activator" procedure manages "worker" and "task generation" procedures with reference to a description of the data to be acted upon. The activator builds a task descriptor to connect the worker, task generator and data descriptor and then makes the task descriptor available to other processors.

The processor which called the generator function (along with other available processors) then uses the task descriptor to generate instances of the worker procedure to process subsets of the data. When the last worker task completes, the generator resumes execution of the program that invoked it.

Generator functions conceptually resemble Lisp functions such as mapcar (which separately applies the same action to each element of a list). The Uniform System library provides several commonly used generators to expedite application programming.

Implementation of the generator mechanism requires only a single process on each processor, and no unnecessary context swaps are incurred. Once a generator gets control, the processor kernel need not intervene until the generator runs out of work. This mechanism is insensitive to the number of processors, and allows dynamic balancing of processor workloads.

B.4 ADA'S MODEL OF PARALLEL PROCESSING

Ada's mechanisms for multiprocessing are a product of its historical and technical context [Mundie86]. Since publication of the Ada language specification, theoretical advances have suggested that a more general synchronization mechanism would simplify programming of many applications.

Ada provides process-level concurrency, rather than trying to distribute evaluation of individual expressions onto multiple processors, or execute complete statements in parallel. Concurrent processing is introduced at a subroutine level on the assumption that each given process will execute sequentially on a single processor.

Synchronization in Ada occurs via an "old maid" model, which fails to specify the occurrence time of a rendezvous event, versus merely achieving the rendezvous. This is like stipulating merely that two people should meet each other, without worrying about precisely where or when. In an environment containing multiple processors, neither is certain.

Moreover, Ada rendezvous is also a "blind date." Tasks are asymmetric in the way they refer to each other during rendezvous. The task making the entry call must name the task with which it seeks to rendezvous, but the accepting task need not. Yet these abstract difficulties are not the only problems with rendezvous.

Ada's rendezvous mechanism uses an implicit queueing scheme that can be implemented using semaphores. This is motivated by a desire for efficient code in real-time applications. Each point at which rendezvous can take place in a target task is indicated by an "entry," which resembles a procedure call heading. If its target is not available, Ada suspends the task requesting rendezvous until the target becomes available.

Although Ada eschews direct use of low-level primitives such as semaphores, its rendezvous model stipulates a restriction that encumbers implementation of otherwise desirable abstract tasking designs. Section 9.5.15 of the Language Reference Manual states that suspension of a task requesting rendezvous is accomplished by providing a separate queue for each entry in the target task. This approach uses the same mechanism for both mutual exclusion and task synchronization, just as is the case in systems that use explicit semaphores.

Efficient implementation of a queue-per-entry model results in the "blind date" reference convention, and also requires that entries be declared in the task that accepts them. For the same reason, the Ada select statement provides multiple simultaneous rendezvous requests, but does not allow mixed entry calls and accepts, nor multiple calls.

A common application of rendezvous is to control access to a shared resource. Entries can be declared only in the visible part of a task which accepts them, hence only one task can accept a given entry. The usual solution to this problem introduces a turnaround task as an intermediary between the resource and its users. Rendezvous must therefore occur between the user and the intermediary, rather than directly with the resource. Blocking a user, while the resource services its current request, blocks all other customers and instances of the resource as well. To avoid this difficulty within the Ada model, an application program must provide an additional layer of synchronization mechanisms.

A more general form of tasking in which entry queues are not directly associated with tasks would allow a programmer to block only the calling task, so that a given service can be provided by many different accepting tasks. A rendezvous between a user and a device then has no effect on similar users and devices.

Queue-per-entry rendezvous also does not guarantee fairness, i.e., it is possible for a task to become permanently blocked while waiting for a rendezvous. The only way to avoid this is to scan the queue of every entry declared in a task prior to entering rendezvous, to determine the oldest pending rendezvous request. This solution is unfortunately very expensive (and cumbersome).

Finally, tasking in Ada is meant to be implemented in a run-time package that conceals details of the underlying hardware or operating system. A highly parallel computer system should have available a language which allows the user to fully exploit the increased capabilities of the machine [Cline85]. Yet in Ada, even hardware-level interrupts are attached to parameterless library entries. Although such libraries are not portable, the programmer is prohibited from taking advantage of novel features of contemporary multicomputer hardware and operating systems.

B.5 LOAD SHARING

Multi-processor scheduling is often considered a monolithic problem, to be solved as a single action using global knowledge. Multiple tasks are assigned to multiple servers, but there is only one decision maker. Yet a centralized scheduling algorithm can itself become a bottleneck (even when executed as a parallel computation). Distributed scheduling assigns tasks to processors without use of a central dispatch. Tasks are introduced into the system via "source" nodes are executed by "server" nodes. Whether sources or servers take the initiative in matching work to resources is a necessary design decision.

A global scheduling strategy is a collection of algorithms performed by sources, servers and their communication links. (Local allocation of processing resources when multiple tasks exist on a single server is a well-understood problem in queueing theory.) Equitable allocation of processing resources to tasks, or "load sharing," is a key function of global scheduling.

[Wang85] systematically describes several load sharing algorithms and classifies them into a taxonomy. The first major distinction made is source- versus server-initiative design; the other axis of comparison concerns "information dependancy" of a strategy, i.e., how much a source node must know about the status of servers, or a server about its potential sources. Traditional monolithic scheduling can be viewed as level 1 source-initiative (static partitioning) with a single source (the central control).

Source-initiative algorithms form queues at servers and make scheduling decisions as tasks arrive at sources, whereas server-initiative ones form queues at sources and schedule assignments when a server completes a task. The performance analysis of

[Wang85] reveals that when communication costs are not a dominating effect, server-initiative tends to outperform source-initiative for a given level of information dependency.

Most server-initiative algorithms do not allow a server to become idle when there are tasks waiting in the system (a task is never left waiting in line at a busy server while another stands unoccupied; an idle server spends its time searching for work). Also, some types of source-initiative algorithms degrade rapidly as variability of service times increases or the number of servers become large. (Additional detail appears in Figures 2 through 14 of [Wang85]).

[Wang85]'s comparison uses a quality factor which measures how closely an algorithm resembles a multiserver first-come-first-serve (FCFS) system for EVERY task. Their model assumes a communications medium which provides only simple message passing (without broadcast, virtual addressing, or conditional reception). The performance analysis involves unsolved problems in queueing theory, which the authors circumvented via simulation.

B.6 REAL-TIME OPERATION

B.6.1 General Considerations

Real-time operation requires availability of results within a given time span after the arrival of input data [Halang86a]. "Deadline driven" scheduling (task execution ordered by increasing deadline) is a natural method to use for this, but features of commercially available hardware and operating system software have hampered its widespread adoption.

Deadline-driven scheduling yields a minimum number of task preemptions, and consequently reduces context switching overhead and resource synchronization problems on a uniprocessor. It also generates an optimal (preemptive) schedule even when additional runnable tasks are added to an already executing task set.

Task preemption on a symmetric configuration of M processors increases communications overhead (as well as context switching overhead), especially for designs which lack shared memory (e.g., hypercube architectures). Determination of whether a specific set of tasks can be processed within a given interval requires a rather complex analysis (for $M > 1$), irrespective of preemption. Moreover, structural constraints can extend running time on symmetric multiprocessor configurations beyond that of a single processor of M times greater power, irrespective of overhead.

Unfortunately, deadline-driven assignment is infeasible for symmetric multiprocessor configurations. A "single-processor" system supplemented with "peripheral devices" specialized for carrying through functions of an operating system nucleus (i.e., an asymmetric multiprocessor) avoids this difficulty, as does a collection of uniprocessors each of which is dedicated to a specific part of a longer sequential process.

Thus, to minimize context-switching and potential deadlock due to resource contention, it seems prudent to maintain strictly sequential handling of task sets imposed by a deadline-driven scheduling algorithm, insofar as possible. This goal fits neatly into overlapping execution in an asymmetric parallel processor, with alignment via interval deadlines.

When an application presents absolute deadlines which cannot be relaxed, asymmetric parallelism is preferable to a collection of identical multiple processors. Deviations from normal program flow on the main processor(s) should be handled by dedicated "peripheral" units to provide reaction to external events under predefined, guaranteed time limits.

When predictable real-time response is guaranteed, one may consider precisely when (rather than merely whether) a synchronization event will occur. Exact simultaneous manipulation of external parallel activity is beyond the capability of existing commercial "real-time" computers. Military systems are no better; Ada lacks any facility for specifying the occurence time of a rendezvous event, versus merely achieving the rendezvous.

Rendezvous itself can be a far from trivial problem. Despite a quintuply-redundant computer system, the first US Space Shuttle mission was delayed by 24 hours only minutes before launch due to an implementation error affecting synchronization initialization among the five computers. [Garman81] The problem arose from a 1-in-67 probability involving a queue that wasn't empty when it should have been and modeling of past and future time.

What matters in real-time control is not only "how fast" but also "exactly when." A computer control system has to provide timely and simultaneous response to multiple events, even under extreme workloads. However, simultaneous manipulation of several parallel control activities can only be approximated on a single processor. Response times depend on the computer's workload, hence the interval between arrival of a given signal and output of control information is an unpredictable function of operating system overhead.

Control theory often incorrectly assumes that a negligible time delay intervenes between measurement of a control variable and the resulting control action. However, even a small delay can cause the control loop to become unstable with respect to parameter variations or disturbance inputs. To avoid this, both acquisition of multiple state variables and output of multiple control signals should involve strictly simultaneous events.

The objective of external simultaneity can be achieved by specifying the exact times at which data exchange with a control activity should occur, and assuring that peripheral units contain hardware necessary to carry through time-related functions [Halang86b].

Language syntax and operating system services to support the foregoing objective are natural extensions of the additional hardware components (clocks, counters, and latches) needed to provide the required timestamping and triggering functions.

Such services also permit resource synchronization based on methods used in everyday life for scheduling meetings, making airline reservations, and similar activities. Individual tasks (travelers) request points in future time at which they will use specific peripherals (airline seats). The Operating System (O/S) (reservation desk) assigns time slots to the requests, and maintains them in a queue (but does NOT remove an "airline seat" from service for the whole interval between reservation and use). Synchronization conflicts which arise are resolved IN ADVANCE on a priority basis, with exception handling ("sorry sir, that flight is booked up ...").

This is very far from being possible in existing commercial real-time computer technology. For example, Ada's so-called rendez-vous mechanism is comparable to specifying merely that two people should meet each other, without stipulating precisely where or when. For a uniprocessor, at least the former is implicitly known, but in an environment containing multiple processors, neither is certain.

Hardware support for precisely timed fully parallel process I/O requires:

a) REAL-TIME CLOCK INPUT be available to every I/O device;

b) CLOCK COUNTERS be integrated into each I/O controller, driven by the controlling computer's clock signal and started and reset with the same signal used to reset the controlling computer;

c) ARRIVAL TIMES of externally triggered data inputs be latched upon assertion of a handshake signal;

d) hold registers to record TRIGGER TIMES for command outputs, with command execution controlled by comparators fed by the real-time clock input.

Reading a system clock immediately after an input statement in an application task cannot account for the unpredictable delay which arises from details of interaction between low-level device handlers and the remainder of the operating system (e.g., as between assertion of a data ready interrupt signal by the controller and execution of an input statement in the task).

The mechanisms necessary to provide exact knowledge of when an input arrives and precise control of when a command is executed by a peripheral controller are straightforward; but they cannot be correctly implemented in software as an afterthought.

B.6.2 A Real-Time Multiprocessing System

Research in support of an Adaptive Suspension Vehicle (ASV) at Ohio State University (OSU) has produced a Generalized Executive for real-time Multiprocessor applications (GEM) [Schwan87]. GEM supports a large variety of operating system primitives to suit various constraints of subproblems in robotics applications.

The STILE design prototype tool under development as part of OSU GEM generates real-time multiprocessing programs from high-level graphical descriptions. Component blocks selected from one or more user-prepared libraries can be connected together using a graphical editor.

GEM itself supports medium-grain parallelism of control tasks against real-time and reliability constraints, and offers two different grain sizes: large loosely interacting tasks that execute infrequently ("processes") versus small closely coupled tasks that execute at high rates ("microprocesses"). The GEM executive schedules, synchronizes and executes tasks within strict time constraints.

GEM recognizes the importance of exact timing (relative to an external context); its current target hardware architecture uses a single global clock. Communication under GEM can occur via either shared memory or message passing. The message passing method is not desirable for low-latency process interactions or service of high-rate actuators.

Each microprocess consists of a sequence of instructions and associated data structures. It shares its address space with other microprocesses in the same process. Together a set of these provide multiple streams of execution defined and activated in a single GEM process, controlled by a single microscheduler defined statically as part of the parent process. A microprocess lacks many of the data structures of a lightweight process and never executes independently from its parent process. It resembles an event handler in that it always runs to completion.

Processes are structured into hierarchical sets with synchronous or asynchronous interaction, and can interact using either of two distinct models of communication: 1) inputs are current values of hardware sensors, with outputs also a set of current values; and 2) inputs and outputs are both discrete messages that contain commands or data values. Processes are always created statically, during initialization of a complex system.

GEM implements two methods of scheduling: Round-robin (conventional time-sliced multitasking); and Deadline (in which a time interval for completion is specifed, measured from the time a task is made ready to completion of every microprocesses therein that is ready to run). Priority scheduling is supported under both methods (deadline runs processes <u>within each priority level</u> to completion in "shortest deadline first" order).

Deadline scheduling is not currently being used in GEM's target application, the Adaptive Suspension Vehicle (ASV). Use of microprocesses accomplishes many of the same objectives, however. For prompt real-time response to requests for services that are implemented by microprocesses, a simple high-performance mechanism is used, which resembles blocking mechanisms in other operating systems.

Service is requested by writing into a control input, as an event separate from communication of data values. Thus, data can be set up well in advance of an event, and triggered with minimum latency delay. In contrast to the hardware-assisted mechanisms described in [HALANG86b], multiple writes to the same control port are lost if they appear more frequently than a microprocess can service them.

Microprocess scheduling time includes the potential cost of waking up the parent process of the microprocess. Analysis of process and microprocess latency indicates that if there is at least a 50 percent chance of finding the parent process and its microscheduler running, use of a microprocess is faster on average than use of an equivalent process.

GEM provides three different models of task interaction:

1) asynchronous execution with potential loss of data (reader always receives most recent data written);

2) synchronous execution without possible data loss (an error code returned when a reader encounters an empty input queue, or a writer finds a full one);

3) synchronous or asynchronous operation with possible loss of aged data (e.g., overflow in data logging}; readers get an error code from an empty input queue, but writers may overwrite the oldest information therein.

The authors of [Schwan87] claim model 1 is the most useful for low-level control, as an approximation of an "analog" computation.

The STILE graphic prototyping facility supports application development within GEM. A basic object in STILE contains user application code and generic communication code. Each describes a set of microprocesses in a GEM process, is separately compiled and collected in a component (object code) library.

The graphical editor supports interconnection of finished components selected from a library, with automatic generation of command files and initialization code to link, load and startup a prototype application. Graphic elements are boxes (active agents in a program); ports (connection points on the boxes); and links, (connections between ports).

STILE's port-to-port connections assume that a reader always receives the most recent data written to it (asynchronous with potential data loss, model 1 above). Port IDs are local to the object, and interconnection is implemented independently of the internal details of objects.

REFERENCES

Abu-Sufah85	Walid Abu-Sufah and Alex Y. Kwok, "Performance Prediction Tools for Cedar," Proceedings of the 12th International Symposium on Computer Architecture, pp. 406-413 (June 1985).
Allen87	Randy Allen, Donn Baumgartner, Ken Kennedy and Allan Porterfield, "PTOOL: A Semi-automatic Parallel Programming Assistant," Proceedings of the 1987 International Conference on Parallel Programming, pp. 164-169 (1987).
Alliant87	Alliant FX Series Product Summary, Alliant Computer Systems Corporation, Littleton, MA, June 1987.
Annals84	Annals of the History of Computing, Vol 6, No 1, 1984, p61.
Barak85	Amnon Barak and Ami Litman, "MOS: A Multicomputer Distributed Operating System", Software: Practice and Experience, Vol. 15(8), August 1985, p725.
Bates83	Peter C. Bates and Jack C. Wileden, "High-level debugging of distributed systems: the behavioral approach," JSS 3 pp. 255-264 (1983).

BBN86a	"Butterfly Parallel Processor Overview", BBN Report No. 6148, Bolt Beranek and Newman, Cambridge, MA, Version 1, March 6, 1986.
BBN86b	"The Uniform System Approach to Programming the Butterfly Parallel Processor", BBN Report No. 6149, Bolt Beranek and Newman, Cambridge, MA, Version 2, October 10, 1986.
Berger87	Berger, Marsha J. and Shahid H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors", IEEE Transactions on Computers, May 1987, p570.
Berman87	Francine D. Berman, "Experience with an Automatic Solution to the Mapping Problem," in The Characteristics of Parallel Algorithms, Leah H. Jamieson, Dennis B. Gannon and Robert J. Douglass (eds.), The MIT Press (1987).
Bloom83	Bloom, T. "Dynamic Module Replacement in a Distributed Programming System," Technical Report MIT/LCS/TR-303, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1983.
Bobrow86	Bobrow, D., et al. "CommonLoops: Merging Lisp and Object-Oriented Programming," Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages and Applicactions, 1986. Published as SIGPLAN Notices 21, 11 (November 1986), 17-29.

Bond87	J. Bond, "Parallel-processing concepts finally come together in real systems", Computer Design, June 1, 1987, p51.
Bruce86	Bruce, K., and Wegner, P. "Algebraic and Lambda Calculus Models of Subtype and Inheritance (Extended Abstract)," unpublished 1987. This is a revised and improved version of K. Bruce and P. Wegner, "An Algebraic Model of Subtypes in Object-Oriented Languages (Draft)," SIGPLAN Notices 21, 10 (October 1986).
Carrington86	D.A. Carrington, "Profiling under Elxsi Unix", Software: Practice and Experience, Vol 16(9), September 1986, p865.
Cline85	C.L. Cline and H.J. Siegel, "Augmenting Ada for SIMD Parallel Processing", IEEE Transactions on Software Engineering, Vol SE-11, No. 9, September 1985.
Cuny87	Janice E. Cuny, Duane A. Bailey, John W. Hagerman, and Alfred A. Hough, "The Simple Simon Programming Environment: A Preliminary Report," to appear Proceedings of the 25th Annual Allerton Conference on Communication, Control and Communication (1987).
Cvetanovic86	Cvetanovic, Zarka, Performance Analysis of Multiple-Processor Systems, PhD Disseration, University of Massachusetts, Amherst, MA, May 1986.

Cvetanovic87	Cvetanovic, Zarka, "The Effects of Problem Partitioning, Allocation, and Granularity on Performance of Multiple-Processor Systems", IEEE Transactions on Computers, April 1987, p421.
Emrath85	P. Emrath, "Xylem: An Operating System for the Cedar Multiprocessor", IEEE Software, Vol 2, July 1985, p30.
Finkel87	Rapheal A. Finkel, "Large-Grain Parallelism - Three Case Studies," in The Characteristics of Parallel Algorithms, Leah H. Jamieson, Dennis B. Gannon and Robert J. Douglass (eds.), The MIT Press (1987).
Gaglianello86	R.D. Gaglianello and H.P. Katseff, "Communications in Meglos", Software: Practice and Experience, Vol 16(10), October 1986, p945.
Gait85	J. Gait, "A Debugger for Concurrent Programs", Software: Practice and Experience, Vol 15(6), June 1985, p53
Gajski85	Gajski, Daniel D. and Jih-Kwon Peir, "Essential Issues in Multiprocessor Systems", IEEE Computer, June 1985, p9.
Garey79	M.P. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Co., San Francisco, 1979.

Garman81	J.R. Garman, "The Bug Heard 'Round The World", Software Engineering Notes, Vol 6, No 5, October 1981, p3.
Gregoretti86	F. Gregoretti, F. Maddaleno, M. Zamboni, "Monitoring Tools for Multiprocessors", Microprocessing and Microprogramming, Vol 18, December 1986, p409.
Gurd85	J.R. Gurd, C.C. Kirkham and I. Watson, "The Manchester Prototype Computer," CACM 28(1), PP. 34-52 (1985).
Goldberg83	Goldberg, A., and Robson, D. "Smalltalk-80: The Lanbuage and its Implementation," Reading, MA: Addison-Wesley, 1983.
Halang86a	W.A. Halang, "Implications on Suitable Multiprocessor Structures and Virtual Storage Management when Applying a Feasible Scheduling Algorithm in Hard Real-Time Environments", Software: Practice and Experience, Vol 16(8), August 1986, p761.
Halang86b	W.A. Halang, "A Precisely Timed Fully Parallel Process Input/Output Facility for Control Applications", Microprocessing and Microprogramming, Vol 18, December 1986, p657.
Halstead84	R.H. Halstead, "Implementation of Multilisp: Lisp on a Multiprocessor", Proceedings of the 1984 ACM symposium on LISP and Functional Programming, p9.

Halstead85	R.H. Halstead, "Multilisp: a language for concurrent symbolic computation", ACM Transactions on Programming Languages and Systems, Vol 7, No 4, October 1985, p501.
Halstead85	Halstead, R. "Multilisp: A Language for Concurent Symbolic Computation," ACM Trans. on Programming Languages and Systems 7, 4, October 1985.
Hewitt79	Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages," in Artificial Intelligence, An MIT Perspective, P. Winston and R. Brown, Eds., Cambridge, MA: MIT Press, 1979.
Hillis85	W.D. Hillis, The Connection Machine, MIT Press, Cambridge, MA, 1985.
Hillis86	D. Hillis and L. Steele, "Data Parallel Algorithms", Communications of the ACM, Vol 29, No 12, December 1986, p1170.
Hillis87	W.D. Hillis, "The Connection Machine", Scientific America, Vol 256, No 6, June 1987, p108.
Hoare72	Hoare, C.A.R. "Proof of Correctness of Data Representations," Acta Informatica 4, 1972, 271-281.

Hough87 Alfred A. Hough and Janice E. Cuny, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation," Proceedings of the 1987 International Conference on Parallel Processing, pp. 735-738 (1987).

Hust87 R. Hurst, "Multiprocessing Unix Meets Users' Needs", Computerworld Focus, 7 January 1987, p29.

Janssens86 M.D. Janssens, J.K. Annot and A.J. Van de Goor, "Adapting Unix for a Multiprocessor Environment", Communications of the ACM, Vol 29, No. 9, September 1986, p895.

Jordan87 Harry Jordan, The Force. Technical Report, Department of Electrical and Computer Engineering, University of Colorado (January 1987).

Karp87 Karp, Alan H., "Programming for Parallelism", IEEE Computer, May 1987, p43.

Kerner86 H. Kerner and H. Raniel, "EDDA, a language based on petrinents and the dataflow principle for the development of parallel programs", Microprocessing and Microprogramming, Vol 18, December 1986, p299.

Koebel87 Charles Koebel, Piyush Mehrotra and John Van Rosendale, "Semi-automatic Doman Decomposition in BLAZE," Proceedings of the 1987 International Conference on Parallel Programming, pp 521-524 (1987).

Kramer87	O. Kramer and H. Muehlenbein, "Mapping Based Strategies in Message-Based Multiprocessor Systems," PARLE: Parallel Architectures and Languages Europe, Lecture Notes in Computer Science 259, J.W. de Bakker, A.J. Nijman and P.C. Treleaven (eds.), Springer-Verlag, pp. 213-225 (June 1987).
Kuck83	David Kuck, Duncan Lawrie, Ron Cytron, Ahmed Sameh and Daniel Gajski, The Architecture and Programming of the Cedar System. Technical Report, Laboratory for Advanced Supercomputers, University of Illinois (1983).
Kuck86	D.J. Kuck, E.S. Davidson, D.H. Lawrie and A.H. Sameh, "Paralle Supercomputing Today and the Cedar Approach", Science, Vol 231, 28 February 1986, p967.
Lazzerini86	B. Lazzerini and C.A. Prete, "DISDEB: An Interactive High-level Debugging System for a Multi-microprocessor System", Microprocessing and Microprogramming, December 1986, p401.
Leavens88	Leavens, G. "Subtyping and Generic Invocation: Semantics and Language Design," Ph.D. thesis, expected 1988.
LeBlanc86	Thomas J. LeBlanc and John M. Mellor-Crummey, Debugging Parallel Programs with Instant Replay, Butterfly Project Report 12, Computer Science Department, University of Rochester (September 1986).

Leblanc87 T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", IEEE Transactions on Computers, vol C-36, No 4, April 1987, p471.

Lee85 G. Lee, C.P. Kruskal and D.J. Kuck, "An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors", IEEE Transactions on Computers, Vol C-34, No 10, October 1985, p927.

Lee87 Lee, Soo-Young and J.K. Aggarwal, "A Mapping Strategy for Parallel Processing", IEEE Transactions on Computers, April 1987, p433

Liskov72 Liskov, B. "A Design Methodology for Reliable Software Systems," Tutorial on Software Design Techniques, P. Freeman and A. Wasserman, Eds.), IEEE, 1977, 53-61. Also published in the Proc. of the Fall Joint Computer Conference, 1972.

Liskov74 Liskov, B., and Zilles, S. "Programming with Abstract Data Types," Proc. of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9, 1974, 50-59.

Liskov77 Liskov, B., Snyder, A., Atkinson, R.R., and Schaffert, J.C. "Abstraction mechanisms in CLU," Comm. of the ACM 20, 8, August 1977, 564-576.

Liskov83	Liskov, B., and Scheifler, R.W. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," ACM Trans. on Programming Languages and Systems 5, 3 (July 1983), 381-404.
Liskov84	Liskov, B., et al. "CLU Reference Manual," Springer-Verlag, 1984.
Liskov86	Liskov, Barbara and Guttag, John, "Abstraction and Specification in Program Development," MIT Press, 1986.
Liskov86b	Liskov, B., Herlihy, M. and Gilbert, L., "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing." Proc. of the 13th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, January 1986.
Liskov87	Liskov, B., and Shrira, L. "Promises: An Efficient Procedure Call Mechanism for Distributed Systems, (Extended Abstract)," Programming Methodology Group Memo 60, M.I.T. Laboratory for Computer Science, Cambridge, MA, November 1987.
Maples85	C. Maples, "Analyzing Software Performance in a Multiprocessor Environment", IEEE Software, Vol 2, July 1985, p50.

Mehrotra82	Piyush Mehrotra and Terrence Pratt, "Language Concepts for Distributed Processing of Large Arrays," Proceedings of ACM Symposium on Principles of Distributed Computing, pp. 19-28 (1982).
Meyer75	W. de Brito Meyer and J.A. Hawley III, Munix, a multiprocessor version of UNIX, Master's thesis, Naval Postgraduate School, Monterrey, CA, 1985.
Miller86	B.P. Miller, C. Macrander and S. Sechrest, "A Distributed Programs Monitor for Berkeley Unix", Software: Practice and Experience, Vol 16(2), February 1986, p183.
Moon86	Moon, D. "Object-Oriented Programming with Flavors," Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 1986. Published as SIGPLAN Notices 21, 11 (November 1986), 1-8.
Muehlenbein86	H. Muehlenbein, F. Limburger, S. Streitz, and S. Warhaut, "MUPPET - A Programming Environment for Message-Based Multiprocessors," FJCC (1986).
Mundie86	D.A. Mundie and D.A. Fisher, "Parallel Processing in Ada", IEEE Computer, August 1986, p20.

Nelson87	Philip A. Nelson and Lawrence Snyder, "Programming Methodologies for Nonshrared Memory Parallel Computers," in The Characteristics of Parallel Alorithms, Leah H. Jamieson, Dennis B. Gannon and Robert J. Douglass (eds.), the MIT Press (1987).
Nikhil87	Arvind and Rishiyur S. Nikhil, "Executing a program on the MIT Tagged-Token Dataflow Architecture," PARLE: Parallel Architectures and Languages Europe, Lecture Notes in Computer Science 259, J.W. de Bakker, A.J. Nijman and P.C. Treleaven (eds.), Springer-Verlag, pp. 1-29 (June 1987).
Odijk87	Eddy A. M. Odijk, "The DOOM System and its Applications: A Survey of the ESPRIT 415 SUbproject A, Phillips Research Laboratories," PARLE Parallel Architectures and Languages Europe, Springer-Verlag Lecture Notes in Computer Science 258 pp. 461-479 (1987).
Ogle85	Dave Ogle, Karsten Schwan, and Richard Snodgrass, "The Real-Time Collection and Analysis of Dynamic Information in a Distributed System," Technical Report OSU-CISRC-TR-85-12, The Ohio State University, (1985).
Olson85	R. Olson, "Parallel Processing in a Message-Based Operating System", IEE Software, July 1985, p39.
Osterhaug86	A. Osterhaug, Guide to Parallel Programming,

Sequent Computer Systems, Inc., Beaverton, Oregon, 1986.

Padua80 Depot Padua, D.J. Kuck and D.H. Lawrie, "High Speed Multiprocessors and Compilation Techniques", IEEE Transactions on Computers, Vol C-29, No 9, September 1980, p763.

Parnas71 Parnas, D. "Information Distribution Aspects of Design Methodology," Proceedings of the 1971 IFIP Congress, North Holland Publishing Co.

Parnas72 Parnas, D. "On the Criteria to be Used in Decomposing Systems into Modules," Comm. of the ACM 15, 12 (December 1972), 1053-1058.

Pratt87 Terrence W. Pratt, "The Pisces 2 Parallel Programming Environment," Proceedings of the 1987 International Conference on Parallel Processing, pp. 439-445 (August 1987).

Purtilo87 James Purtilo, Daniel A. Reed and Dirk C. Grunwald, "Environments for Prototyping Parallel Algorithms," Proceedings of the 1987 International Conference on Parallel Processing, pp. 431-438 (August 1987).

Raniel85 H. Raniel, "Strukturierung des Dataflusses in EDDA", TR 85/03/01, Institute fuer Angew, Informatik und Systemanalse, TU Wein, Vienna, Austria, 1985

Reed87a Reed, Daniel A. and Dirk C. Grunwald, "The Performance of Multicomputer Interconnection Networks", IEEE Computer, June 1987, p69.

Reed87b	Reed, Daniel A. and R.M. Futjimoto, Multicomputer Networks: Message-Based Parallel Processing, MIT Press, Cambridge, MA, September 1987.
Schaffert86	Schaffert, C., et al. "An Introduction to Trellis/Owl," Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 1986. Published as SIGPLAN Notices 21, 11 (November 1986), 9-16.
Schwan87	K. Schwan, T. Bihari, B.W. Weide and G. Taulbee, "High-Performance Operating System Primitives for Robotlcs and Real-Time Control Systems", ACM Transactions on Computer Systems, Vol 5, No 3, August 1987, 1989.
Segall85	Zary segall and Larry Rudolph, PIE - A Programming and Instrumentation Environment for Parallel Processing. Technical Reprot CMU-CS-85-128, Carnegie-Mellon University (April 1985).
Segall87	Zary Segall and Lawrence Snyder (eds.), Workshop on Performance Efficient Parallel Programming, Technical Report, Department of Computer Science, Carnegie-Mellon University (September 1987).
Shen85	Shen, Chien-Chung and Wen-Hsiang Tasi, "A graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion", IEEE Transactions on Computers,

March 1985, p197.

Snyder84	Lawrence Snyder, "Parallel programming and the Poker Programming Environment," Computer 12(1), pp. 47-56 (1982).
So87	K. So, A.S. Bolmarcich, F. Darema and V.A. Norton, "A Speedup Analyzer for Parallel Programs," Proceedings of the 1987 International Conference on Parallel Programming, pp. 653-651 (1987).
Steele86	G.L. Steele and W.D. Hillis, "Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing", Proceedings of the 1986 ACM Conference on LISP and Functional Programming, Cambridge, MA, p279.
Test87	J.A. Test, "Multiprocessor Management in the Concentrix Operating System", Alliant Computer Systems, Acton, MA.
Tuyenman86	F. Tuyenman and L.O. Hertzberger, "A Distributed Real-Time Operating System", Software: Practice and Experience, Vol 16(5), May 1986, p425.
Wang85	Wang, Yung-Terng and Robert J.T. Morris, "Load Sharing in Distributed Systems", IEEE Transactions on Computers, March 1985, p204.
Winston	Winston, Patrick H., Artificial Intelligence, 2nd ed, Addison-Wesley, Reading, MA, p115.

Yokote86 Yokote, Y., and Tokoro, M. "The Design and Implementation of Concurrent Smalltalk," Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 1986. Published as SIGPLAN Notices 21, 11 (November 1986), 331-340.

DISTRIBUTION LIST

addresses	number of copies
Williams, Alan N RADC/COTC	7
RADC/DOVL GRIFFISS AFB NY 13441	1
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22304-6145	5
Dynamics Research Corporation 60 Frontage Rd Andover, Massachusetts 01810	2
SDIO/S-BM (Lt Col Sowa) The Pentagon Washington DC 20301-7100	1
SDIO/S-BM (Lt Col Rindt) The Pentagon Washington DC 20301-7100	1

IDA (SDIO Library) (Albert Perrella) 1801 N Beauregard Street Alexandria VA 22311	2
SAF/AQSD (Lt Col Ben Greenway) The Pentagon Washington DC 20330	1
AFSC/CV-D (Lt Col Flynn) Andrews AFB MD 20334-5000	1
HQ SD/XR (Col Heimach) PO Box 92960 Worldway Postal Center Los Angeles CA 90009-2960	1
HQ SD/CNI (Col Hohmar) PO Box 92960 Worldway Postal Center LA CA 90009-2960	1
HQ SD/CNW PO Box 92960 Worldway Postal Center Los Angeles CA 90009-2960	1
ESD/AT (Col Paul) Hanscom AFB MA 01731-5000	1
AFSTC/XLX (Lt Col Detucci) Kirtland AFB, NM 87117	1

USA SDC/DASD-H-SE (Larry Tubbs) 1
PO Box 1500
Huntsville, AL 35807

ANSER Corp 1
Suite 800
Crystal Gateway 3
1215 Jefferson Davis Highway
Arlington VA 22202

AFOTEC/XPP (Capt Wrobel) 1
Kirtland AFB NM 87117

AF Space Command/XPXIS 1
Peterson AFB CO 80914-5001

Director NSA (V43 George Hoover) 1
9800 Savage Road
Ft George G Meade MD 20755-6000

SDIO/S-BM (Capt Hart) 1
The Pentagon
Washington DC 20301-7100

SDIO/S-BM (CDR Newton) 1
The Pentagon
Washington DC 20301-7100

HQ SD/CN (COL WILKENSON) 1
PO BOX 92960
WORLDWAY POSTAL CENTER
LA CA 90009-2960

HQ SD/CNIS 1
(LT COL FENNELL)
PO BOX 92960
WORLDWAY POSTAL CENTER
LA CA 90009-2960

HQ SD/CWX PO BOX 92960 WORLDWAY POSTAL CENTER LA CA 90009-2960	1
HQ SD/CNE PO BOX 92960 WORLDWAY POSTAL CENTER LA CA 90009-2960	1
ESD/ATS (LT COL OLDENBERG) HANSCOM AFB MA 01731-5000	1
ESD/ATN (LT COL LEIB) HANSCOM AFB MA 01731-5000	1