Experience with Three Parallel Processing Systems

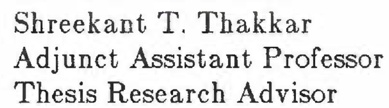
Marta Kallstrom
B.A., University of Oregon, 1972

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

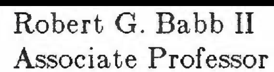
March 2, 1987

copyright© 1987 by Marta Kallstrom

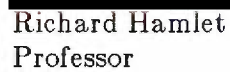
The thesis "Experience with Three Parallel Processing Systems", by Marta Kallstrom, has been examined and approved by the following Examination Committee:



Shreekant T. Thakkar
Adjunct Assistant Professor
Thesis Research Advisor



Robert G. Babb II
Associate Professor



Richard Hamlet
Professor

David Maier
Associate Professor

Acknowledgements

I would like to thank my thesis advisor Shreekant Thakkar for his attention, his guidance, and his confidence in me, and for the time he took out of his busy schedule at Sequent Computer Systems to work with me. My committee members provided valuable advice in the final stages of my thesis and made a special effort to help me meet my deadline. I am especially grateful to Wm Leler, who commented on many iterations of this thesis and who has inspired and encouraged me during my studies at OGC.

In addition, I want to thank Sequent Computer Systems, Floating Point Systems, INMOS, Tektronix, and Intel Scientific Computers for the use of their computers and their willingness to assist in this study. Dave Olien and Ron Parsons at Sequent, Stuart Hawkinson at Floating Point, Pete Wilson at INMOS, Peter Borgwardt at Tektronix Computer Research Labs, and Chris Grant at Intel answered a number of questions and helped me familiarize myself with the different systems.

Table of Contents

Abstract iv

Chapter 1: Introduction 1

1.1 Parallel Processing Concepts 2
1.2 Literature 6
1.3 Three Parallel Processing Systems 8

Chapter 2: The Languages 12

2.1 C with Parallel Library on the iPSC 12
2.2 C with Parallel Library on the Sequent Balance 16
2.3 Occam on INMOS Transputers 20

Chapter 3: The Programs 25

3.1 Dining Philosophers 26
3.2 Matrix Multiply 28
3.3 Traveling Salesman Problem using Simulated Annealing 30

Chapter 4: Comparison of Three Parallel Programming Systems 32

4.1 Division of Work: Designation of Parallel and Sequential Computation 32

4.1.1 iPSC 33
4.1.2 Occam on Transputers 36
4.1.3 Sequent Balance 38
4.1.4 Comparison of Division of Work 40

4.2 Sharing of Data 42

4.2.1 iPSC 42
4.2.2 Occam on Transputers 50
4.2.3 Sequent Balance 54

4.2.4 Comparison of Data Sharing 57

4.3 Synchronization Methods 58

4.3.1 iPSC 59
4.3.2 Occam on Transputers 60
4.3.3 Sequent Balance 61
4.3.4 Comparison of Synchronization Methods .. 62

Chapter 5: Parallel Program Development 63

5.1 iPSC 64
5.2 Occam on Transputers 69
5.3 Sequent Balance 76
5.4 Comparison of Program Development Methods 79

Chapter 6: Conclusions 82

6.1 Strengths and Weaknesses of Each System 82
6.2 Future Directions 84

Appendix A: Timing and Results of the
Traveling Salesman Program 86

A.1 iPSC Version 87
A.2 Sequent Balance Version 90
A.3 Occam Version 91
A.4 Sample Output 91

Appendix B: Dining Philosophers Program Listings 94

References 103

Abstract

Several programs were developed using three different parallel programming systems: the Intel iPSC, Sequent Balance and INMOS transputers. The objective was to examine different approaches to parallel processing from a programmer's point of view. The study compares methods for dividing work among processors, for sharing data, and for synchronizing processes. It also examines program development, including compilation, process management and debugging techniques.

The Sequent Balance provides the "friendliest" environment, but its shared memory limits parallelism in some cases. Programming in occam on transputers requires the longest learning curve, but the language's support for pervasive parallelism allows flexibility and promotes creativity in the development of parallel algorithms. The iPSC embodies only a high-level parallelism and is not particularly easy to program.

Chapter 1

Introduction

Until recently parallel processing was largely experimental, but now a number of multiprocessors have arrived in the marketplace and with them a profusion of new program development systems. An important consideration in evaluating a parallel processing system is how it is to be programmed. In some cases smart compilers can search for parallelism in existing sequential programs and free the programmer from the job of parallelizing code. Currently, however, compilers are limited in the amount of parallelism they can find and compilers that detect any parallelism are not widely available. Efficient use of multiprocessors relies on the programmer to implement parallel, or parallelizable, algorithms. On some systems, such as the Sequent Balance[TGF], Intel iPSC[Int85], Cray X-MP[CHL, CCC85], and Alliant FX/8™[AAA86, KDL86], library extensions to conventional operating systems and languages help the programmer to specify actions of multiple processes. Other systems offer new languages that embody concepts of parallelism, including the Meiko Computing Surface™[86], Floating Point Systems T Series[GHS86], and the Connection Machine[Hil85]. While information on architectures and programming models abound, little has been available on the practical aspects of parallel programming. This study is a step in that direction.

™ Alliant and FX/8 are both trademarks of the Alliant Computer Systems Corporation.

™ Computing Surface is a trademark of Meiko Ltd.

Three typical programming problems were implemented on each of three commercially available systems. They include the Sequent Balance®, the Intel iPSC™ and INMOS transputers. The objective was to compare the expressive power of the three parallel processing language implementations, to discover the degree to which these typical problems could be solved in a straightforward and efficient manner, and to understand the different viewpoints that must be taken when programming on each system.

The remainder of this chapter covers parallel processing concepts that form the basis for the comparisons made in the study, a review of background literature, and a description of the three systems used in the study.

1.1. Parallel Processing Concepts

Flynn [Fly66] classifies parallel processing systems as:

Single instruction stream - single data stream (SISD)
Single instruction stream - multiple data stream (SIMD)
Multiple instruction stream - single data stream (MISD)
Multiple instruction stream - multiple data stream (MIMD)

Most serial computers available today are SISD. MISD has no real representatives, but parallelism has been achieved by array and vector processors in the SIMD category and multiprocessors in the MIMD category. The systems in this study are all MIMD. The Sequent Balance is a shared-memory multiprocessor, and the Intel iPSC and INMOS transputers are both distributed systems that rely on message passing.

® Balance is a registered trademark of Sequent Computer Systems.
™ iPSC is a trademark of Intel Scientific Computers.

Hwang and Briggs [HwB84] define parallel processing as follows:

> Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity, and pipelining.

They describe four levels of parallel processing. First is multiprogramming and time sharing at the program or job level. Second is multiple tasks or procedures within the same program. Third is parallelism among multiple instructions, including pipelined execution. Fourth is parallelism within an instruction, including the operation of a single instruction on multiple data items. The three systems examined in this study share the characteristic of the second level, although in certain cases they allow additional levels of parallelism.

For procedures within the same program to execute concurrently, a program must be decomposed into separate tasks. Imperative languages are based on a *process* model, so the separate tasks are processes. Dividing the program can be the job of the programmer or of the system. This study is concerned with the programmer's job.

The granularity of parallelism is a factor of the algorithm itself and the judgment of the programmer. It may range from large procedures to single instructions executing in parallel. The programmer must be aware of the effect of granularity on efficiency of the program. The overhead associated with a process varies from system to system. Creating more processes than necessary can introduce overheads that slow execution, but creating too few processes, or the wrong size processes, can result in underutilized processors.

If processes are to cooperate in problem solving, they must be able to *communicate* with each other to share computational results. Parallel processing architectures may be divided into two main categories: *shared-memory multiprocessors* and *distributed-memory systems*. In the former, all processes share some global memory and communicate through shared variables. In the latter, each processor's memory is separate, and processes communicate by passing messages over communication links.

Synchronization specifies how processes order events and coordinate updates to shared data. One process must signal that it is in a certain state, and other process must receive that signal. Several mechanisms support synchronization, including message passing, shared variables, and explicit synchronization routines.

On shared-memory systems a basic synchronization mechanism is a *semaphore*, a shared data structure that has two or more states. A *lock*, the simplest form of a semaphore, may be used to protect variables in shared global memory. A lock is associated with a variable. A process that locks a variable obtains exclusive access to it. *Barriers* are a second low-level mechanism for synchronizing processes. Two or more processes may share a variable defined as a barrier. They may meet at the barrier to ensure that they are at the same point in execution. Another form of synchronization provides protection for an entire section of shared code that one process must execute indivisibly, called a *critical region* [Bri72][Hoa72]. A shared-memory parallel programming environment typically provides higher-level control mechanisms based on semaphores or barriers to accomplish these forms of synchronization.

Distributed systems achieve synchronization by sending messages. There are two types of message transmission: *blocking* and *nonblocking*. In the first type, the sender blocks its execution until the message has been received. In a nonblocking send, a process sends a message but then continues executing code without waiting for the message to be received. Nonblocking messages can be simulated using blocking messages by buffering messages through a third process. Since the purpose of synchronization is typically to insure that a process has certain information before continuing its execution, either type of message may be sufficient for synchronization.

Three error conditions can arise when executing a set of parallel processes: deadlock, starvation and race. On a shared-memory system, a process is *blocked* while it waits for a busy semaphore to become free; on a distributed system, a process may be blocked when a message transfer cannot be completed. If a cycle of processes are blocked, they are *deadlocked*. This condition can result from a set of processes each holding a shared resource while attempting to access another; or, on a distributed system, from a set of processes all trying to send messages when none is receiving or all waiting to receive while none is sending. *Starvation* occurs when a particular process cannot access a shared resource while other processes continue to execute. The cause of starvation is often a matter of priority among the processes. In a *race* condition, multiple processes vie for the same resource in an unsynchronized manner, and the results are unpredictable and incorrect. On a shared-memory system, races may result from accessing shared resources outside of a critical region or in an improper order. In a distributed-memory environment where message transfer is not tightly synchronized, races occur when messages arrive out of order.

By incorporating a library of interface routines, conventional imperative languages can be used to specify concurrent execution. For example, the utilities `fork()` and `join()` are familiar to most UNIX(TM) programmers [KeR78]. By invoking these routines a program may dynamically create and terminate a *child* process that cooperates in the execution of a section of code. Libraries to describe parallel execution must include concepts of multiple processes and processors, communication and synchronization. In addition to extending existing commercial languages, new languages with built-in support for concurrency are appearing in the marketplace, such as Occam[Inm85a, MaS84], Ada[UUU80], Modula-2[Wir83], and C++[Str80, Str86]. The choices are complex.

While the programming task grows more complicated, the programmer's interface to the physical system may also require new skills. Different systems allow the programmer varying degrees of control over the palellel processing elements. They may also provide tools to help a programmer cope with the increasingly complex environment, such as parallel compilers[CHL, PKL80], debuggers[PaL, SSS86b], and monitors [SSS85].

1.2. Literature

Commercial multiprocessors have existed for more than a two decades but have only become widely available in the past several years. To date, most papers addressing multiprocessor programming issues have focused on a particular system or have compared theoretical aspects of various models. Case studies comparing program

---

(TM) UNIX is a trademark of AT&T.

development on different multiprocessors have only recently become possible.

A paper by McGraw and Axelrod of Lawrence Livermore National Laboratory [McA] inspired this study. The authors describe their experiences porting programs from a single-processor system to two different shared-memory multiprocessors. They recount typical problems involving assignment of work to different processes, sharing of data, and races and deadlocks.

Three general references on parallel processing were most instructive. Andrews and Schneider [AnS83] provide a comprehensive introduction to basic concepts, including process interaction and synchronization. Filman and Friedman [FiF84] focus on language models for distributed systems and include a number of programming examples implemented in different languages. Wand and Wellings [WaW84] present a brief introduction to parallel processing primitives and their implementation in various languages, plus a comprehensive bibliography.

A number of papers describe the development of a particular language or operating system for a given multiprocessor. These papers reveal issues important to the designer. Jones, et al., [JJD78] describe general issues of parallel programming with respect to the Cm* system. They emphasize that a multiprocessor architecture is not necessarily *programmable*; that to be programmable, it must be possible to invent algorithms that suit the architecture and to code them expediently. They list issues that make programming multiprocessors more difficult than uniprocessors. These include methods for decomposing problems, managing failure of individual processors, I/O support software, object manipulation, and management of software development.

All of these issues are still important. A programmer's response to these issues depends on the architecture and operating system of a particular multiprocessor. On some systems they are hidden from the programmer, and on others they require detailed attention.

Another category of papers examines parallel processing primitives and process structures in different languages and their effect on the expressive power of the language. These papers include ones by Liskov, et al. [LHG86] and Kieburtz and Silberschatz [KiS79].

A final category of papers compares theoretical aspects of different languages and systems. Representative papers included ones by Stotts [Sto82], Shaw, et al. [SAN81], and Welsh, et al. [WLS79]. These papers help form criteria for comparison of the three systems discussed herein.

The main reason for the dearth of articles comparing practical experiences with parallel processing systems is that until quite recently few systems were commercially available.

1.3. Three Parallel Processing Systems

This study focuses on three different systems, one shared-memory multiprocessor and two distributed-memory systems, all programmed using imperative languages. These particular systems were chosen because they represented the spectrum of parallel processing systems on the market. Programs were also implemented on a Cray X-MP and an Alliant, and several other systems were explored, including the Floating Point Systems T Series and Meiko's Computing Surface, both based on INMOS

transputers. The final choice of systems was limited to three in order to have time to adequately explore each system.

The three program development systems are:

• **C with a parallel programming library** [SSS86a] **on the Sequent Balance 8000** [SSS85]. The Balance 8000 is a shared-memory multiprocessor consisting of up to twelve processors. Programs run under DYNIX® (a multiprocessor UNIX system) using C with multitasking and microtasking support to provide forking of processes, synchronization primitives and management of interprocess communication through shared memory.

• **C with a parallel programming library on a 32-node Intel iPSC** (Personal SuperComputer) [Int85]. The iPSC is a distributed system with 32 to 128 processors connected in a hypercube topology. Code is developed under XENIX™ (a microprocessor UNIX system) on a host processor (the Cube Manager). Processes may be mapped to a node in the hypercube network. Each node has its own processor and memory, and communicates with its neighbors by queued message passing over bidirectional communications channels and with the cube manager over a global Ethernet[MeB76] channel. An interface library provides message passing primitives to allow communication between processes. For background information on the iPSC and the Cosmic Cube, its precursor at CalTech, see articles by Rattner[Rat85] and Seitz[Sei85].

® DYNIX is a registered trademark of Sequent Computer Systems.
™ XENIX is a trademark of Microsoft.

• **occam on transputers**. The occam(TM) language [MaS84] was designed to run on INMOS transputers [Whi85] and is based on Hoare's Communicating Sequential Processes (CSP) [Hoa78]. The transputer is a microprocessor with local memory and communication links that allow easy connection to other transputers. Occam and the transputer were developed concurrently with the objective that code intended for a network of transputers could be developed and verified on a single transputer. Programs are typically developed on a mainframe or personal-computer development system, and then downloaded to transputer networks for execution. Two of the programs for this study were implemented in occam I using the Occam Programming System (OPS)[Inm85a] and have not actually run on transputers. One was implemented in a beta release of occam II on a PC-based occam evaluation board. In this case, program development actually occurs on the transputer using the Transputer Development System (TDS)[Inm86].

Within the confines of procedural languages, the three systems chosen cover the range of currently available systems. This study explores the effect of memory and processor organization, language, and development toolsets on the implementation of parallel programs. Chapter 2 provides a brief introduction to the languages and parallel libraries. An explanation of the algorithms developed on each system follows in Chapter 3. Using illustrative segments of code, Chapter 4 details how each system handles division of work onto multiple processors, communication and synchronization of the processes. Chapter 5 discusses the program development environment on each system, including program visualization tools, debugging methods and real-time

(TM) Occam is a trademark of the INMOS Group of Companies.

support. Throughout the study, the focus is on ease of program development. By implementing the same problem on each system, it was possible to compare the development process and the overhead required to handle concurrency. Admittedly, some bias results when implementing an algorithm on several systems. The process becomes easier each time. To try to erase some of the bias, a different system was used initially for each algorithm.

Chapter 2

The Languages

A rudimentary overview of the languages and parallel libraries of each system will provide a basis for concepts discussed in the following chapters. Both the Intel iPSC and Sequent Balance use the general purpose language C. Parallel interface libraries augment the language to permit C processes to run concurrently and cooperatively. On the other hand, the notion of parallelism is built into the occam language.

2.1. C with Parallel Library on the iPSC

C processes on the iPSC use two different parallel libraries, one for the Cube Manager and one for the nodes, but many of the functions are identical. Routines in each of these libraries are divided into three categories: message sending and receiving, informational, and process management. All library routines are not discussed here, only those required for the programs in the study.

The cube manager is connected to every node in the hypercube by a global Ethernet link. The message routines for *host* processes on the cube manager are `sendmsg()` and `recvmsg()`. These procedures allow a host process (1) to transmit a message to another host process, to a single node process, or to all node processes at once, and (2) to receive a message from a host process or an individual node. The `sendmsg()` routine requires six parameters to indicate the communication channel,

type of message, message buffer, message length, and the destination node id and process id. The `recvmsg()` routine requires seven parameters, all identical to those of `sendmsg()` plus a parameter specifying the number of bytes actually received in the message buffer.

Nodes communicate with each other over bidirectional links and with the host over the global Ethernet. The node interface library includes two types of routines for sending and receiving messages, the non-blocking `send()` / `recv()` and the blocking `sendw()` / `recvw()`. When a non-blocking function is called, it returns to the calling process as soon as the request is recorded. The return status of a `send()` or `recv()` must always be checked before reusing a message buffer to prevent overwriting by another message. Blocking sends and receives do not return until the kernel has actually finished the operation and the message buffer is ready for reuse. The programs in this study used the blocking `sendw()` and `recvw()` exclusively.

The `sendw()` routine uses six parameters, identical to those of `sendmsg()`. The `recvw()` routine has seven parameters, which are similar to those of the host's `recvmsg()` except that the `type` parameter qualifies the type of message received, whereas on the host `type` is a pointer to the type received.

Both the cube manager and node libraries provide routines `copen()` and `cclose()` to create and close a communication channel. A routine called `syslog()` permits host and node processes to log messages to a system log file on the cube manager. Time stamps and originating node and process ids are added to each message.

Informational routines on the host and nodes include `cubedim()` to determine the dimension of the hypercube in which the program is running and `mypid()` to find the process id of the calling process. In addition, the node library includes a routine called `mynode()`, which returns the node id.

Process management routines on the cube manager allow the host process to direct the node processes. They include `load()` to load a file onto a specified node and start it, `lwaitall()` to wait until a node process has completed, and `lkill()` to kill a node process. In all cases, the host may act on a single node or all nodes at once by specifying a special parameter.

Figures 2.1 and 2.2 show the skeletons of typical host and node processes. The parameters in all capital letters are constants defined in a header file common to both the host and node processes.

```
main() {

    hostpid = getpid();                  /* get my process id */
    hostcid = copen(hostpid);            /* open channel */
    load("test", ALL_NODES, PID);        /* node process */

    /* put something in the message buffer */

                                         /* send to all nodes */
    sendmsg(hostcid, MSGTYPE, send_buf, BUFLEN, ALL_NODES, PID);
    num = 1 << cubedim();                /* how many nodes? */
    for (i = 0; i < num; i++) {          /* receive from nodes */
        recvmsg(hostcid, &type, recv_buf, BUFLEN, &cnt, &node, &pid);

        /* process the information received */

        }
    lwaitall(ALL_NODES, ALL_PROCS);      /* wait til nodes finish */
    lkill(ALL_NODES, ALL_PROCS);         /* kill node processes */
    }
```

Figure 2.1 Parallel Library Calls by an iPSC Cube Manager Process

```
main() {

    cid = copen(PID);                    /* open channel */
                                         /* receive from host */
    recvw(cid, MSGTYPE, recv_buf, BUFLEN, &cnt, &hostname,
          &hostpid);

    /* process the message and do other work */

                                         /* send to host */
    sendw(cid, MSGTYPE, send_buf, BUFLEN, hostname, hostpid);
    cclose(cid);                         /* close channel */
    }
```

Figure 2.2 Parallel Library Calls by an iPSC Node Process

2.2. C with Parallel Library on the Sequent Balance

The Sequent Balance supports an extended UNIX process model, which uses a shared-data segment in addition to private address spaces. The DYNIX C compiler supports the variable declaration modifiers `shared` and `private` to specify the proper location of the data in memory. The parallel library includes functions `shmalloc()` and `shfree()` to dynamically manage shared data.

The parallel library on the Balance consists of routines to support microtasking and multitasking models of parallel programming. Microtasking is a shared-memory parallel programming model that has a *master* thread of execution and zero or more *slave* threads. The master thread runs the sequential parts of the application and at appropriate points in the algorithm causes all slaves to work with it in executing some procedures in parallel. The master and slave processes together are called *workers*. Each worker is identified by a variable id number `m_myid`. The `m_myid` of the master thread is `0`. Functions are provided to create the slaves, dispatch them to work on any procedure with arbitrary arguments, and suspend, resume or kill them (in a UNIX sense). Synchronization primitives in the multitasking model are used to manage shared memory, lock individual data structures and provide process barriers.

Low-level multitasking synchronization primitives include the data types `slock_t` to associate a lock with a variable and `sbarrier_t` declare a barrier for two or more processes. The routines `s_init_lock()`, `s_lock()` and `s_unlock()` initialize, lock and unlock an `slock_t` variable. The routines `s_init_barrier()` and `s_wait_barrier()` initialize a barrier of type `sbarrier_t` for a specified

number of processes and cause the calling process to wait at the barrier. The call `cpus_online()` returns the number of processors available for use.

All other parallel library routines are part of the microtasking model. These routines are more abstract and do not require explicit declaration or initialization. Often they do not require parameters. Microtasking supports ways to explicitly parallelize code, including routines to manage processes, access shared variables and synchronize concurrent activity. The routine `m_set_procs(nprocs)` initializes a specified number of processes that will cooperate in executing a task, and `m_fork(func[,arg ...])` creates `nprocs - 1` new slave processes if processes have not been created already. Otherwise, it reuses existing processes. The parameter to `m_fork` is the name of the function to be executed. The master process joins the slave processes to cooperatively execute the function. When the function has been completed, the master process calls `m_kill_procs()` to kill the forked processes.

Code enclosed by microtask calls `m_lock()` and `m_unlock()` is executed by one thread at a time. These microtasks are interfaces to a single `slock_t` lock. The calls are one way of specifying a critical section of code.

Two microtasks cause processes to synchronize, but in slightly different ways. On an `m_sync()` and `m_single()` call processes wait at a barrier, in the first case until all processes arrive and in the second case until the master thread executes a call to a corresponding `m_multi()`. The `m_single()/m_multi()` pair are used to implement *single-threading*, where only the master thread executes a section of code.

Figure 2.3 demonstrates some of the constructs that might be used in a typical application on the Balance.

```
/* global declarations */

shared int x;

shared struct y_struct {
    int state;
    slock_t lp;                 /* primitive lock */
    } y


/* master thread */

main(){

    s_init_lock(&y.lp);         /* initialize primitive lock */
    m_set_procs(nprocs);        /* initialize processes */
    m_fork(work);               /* create processes and start */
    m_kill_procs();             /* kill slave processes */
    }


/* process to be executed concurrently */

work() {

    m_lock();                   /* begin a critical section */
    x++;                        /* increment shared variable */
    printf("Hello, world");
    m_unlock();                 /* end critical section */

    m_single();                 /* slaves wait at barrier */
    printf("x = %d", x);          /* only master thread prints */
    m_multi();                  /* slaves commence execution */

    s_lock(&y.lp);              /* lock y's primitive lock */
    y.state = 0;                /* change shared variable */
    s_unlock(&y.lp);
    }
```

Figure 2.3 Sequent Balance Program Using Multitasking and Microtasking

2.3. Occam on INMOS Transputers

The occam language employs three primitives, := to support assignment of variables, ! to output to a channel, and ? to input from a channel. A process consists of a sequence of these primitive actions. Communication in occam is synchronous and unbuffered. When two processes are ready to communicate over the same channel, a message is output from one process and input to the other. Assignment could be simulated by the communication primitives, but a primitive operator is more convenient.

Primitive processes are combined into more complex processes using SEQ to execute statements sequentially, PAR to execute them in parallel, and ALT to accept input from a number of alternative constructs. Wherever two or more statements appear in an occam program, one of these constructs must precede them to specify how they are to be executed.

For two processes to communicate, they need to execute in parallel and share a common channel, declared using a variable of type CHAN. In the following example, one process outputs (!) the value 2, and the second process inputs (?) that value and assigns it to x. A variable is local to the construct that immediately follows it. In this case, the scope of the variables x and c is the PAR. (Examples in this section were drawn from a tutorial introduction to occam [Pou86].)

```
INT x:
CHAN OF INT c:
PAR
  c ! 2
  c ? x
```

Declaration of two channels provides two-way communication between processes. The SEQ constructs in the following example ensure that two processes exchange values sequentially, first over channel c1 and then over channel c2.

```
CHAN OF INT c1, c2:
PAR
  INT x:
  SEQ
    c1 ! 2
    c2 ? x
  INT y:
  SEQ
    c1 ? y
    c2 ! 3
```

The following example demonstrates the ALT construct. The ALT watches a number of channels for the first input and then executes the process associated with that input. For example, if input arrives first on channel c2, the second process is executed. The process does not continue waiting for input on channels c1 and c3. If this were desired, a PAR construct would suffice.

```
CHAN OF INT c1, c2, c3:
INT x:
ALT
  c1 ? x
    ... first process
  c2 ? x
    ... second process
  c3 ? x
    ... third process
```

The purpose of ALT is to allow a process to be influenced by its environment and to preserve the nondeterminism of a parallel system. If three processes are proceeding

toward output on channels c1, c2, and c3, ALT allows this process to handle the first available input. This policy avoids deadlock and makes the system more efficient.

An IF construct is followed by a number of conditions. The processes under the first true condition are executed.

```
IF
  x = 1
    c1 ! y
  x = 2
    c2 ! y
```

A WHILE causes the process under it to be executed repeatedly as long as the condition is true. In this example, the process reads input from channel input and outputs to output until x is no longer greater than 0.

```
INT x:
SEQ
  x := 0
  WHILE x >= 0
    SEQ
      input ? x
      output ! x
```

The three basic constructs SEQ, PAR, and ALT can be replicated to create multiple copies of the process. A replicated SEQ is similar to a for loop in C and causes a sequential process to be executed multiple times. When an ALT is replicated, n processes concurrently watch for input on the channels specified in the ALT. This construct is used for arrays of channels. A replicated PAR builds an array of parallel processes as shown in the following example. Assuming that a separate process,

possibly the host, begins by sending data on channel c[0], each process accepts input on channel c[i] and outputs on channel c[i + 1] forever.

```
CHAN OF BYTE c[n + 1]:
PAR i = 0 FOR n
  WHILE TRUE
    BYTE x:
    SEQ
      c[i] ? x
      c[i + 1] ! x
```

A name can be given to a process using the label PROC. The body of the procedure is then executed whenever its name is is referenced in a program. Procedures may contain formal parameters, passed by reference and prefixed by keywords indicating their data types. In fact, since transputers have local memories, occam has no concept of global variables. All variables are declared locally and passed as parameters. In this example, 4 occurrences of the procedure buffer are executed concurrently.

```
PROC buffer (CHAN of INT in, out)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x
:
VAL num IS 4:
PAR i = 0 FOR num
  buffer(c1[i], c2[i])
```

Once the logical behavior of a program is established, the program is configured onto a network of transputers by associating logical channel names with physical links and PAR processes with processors.

The samples of occam programs in this study are written in two versions of the language, occam I and occam II. Occam I is the earlier version and contains no data types. In occam II, both data and channel variables are typed, as illustrated in the examples in this section.

Chapter 3

The Programs

A wide range of problem areas are appropriate for parallel processing, including numerical analysis, image processing, graphics, programming artificial intelligence and scientific simulation. Parallel algorithms used to solve these problems may be homogeneous, where all processes are identical, or heterogeneous, where algorithms are partitioned into different sub-functions. The problems chosen for this study come from several disciplines and their solutions are primarily homogeneous algorithms.

To compare the three parallel processing environments, a number of problems were solved on each system. Three were chosen as representative: dining philosophers, matrix multiply and the traveling salesman problem. Four additional programs were written on one or more of these systems, including the Sieve of Eratosthenes, calculation of *pi* using integration, a linear system solution and two-dimensional convolution. They provided some further information but to keep comparisons clear, only three programs are discussed in depth.

The solution to the dining philosophers problem is a relatively small model of operating system contention problems. The matrix multiply problem is one example of a numerical application than can be parallelized in several different ways. The traveling salesman problem was selected for being large enough to be more representative of the complexity of problems that may be required for industry.

The following sections describe each of the problems and their abstract solutions. The next chapter discusses implementations of the solutions on different systems.

3.1. Dining Philosophers

Dining philosophers is a classic resource allocation problem where *philosophers* alternately think and eat, but in order to eat must acquire two shared *forks*. A successful solution provides an environment where every philosopher can think and eat in a continuous cycle [Dij72][Car82].

Cargill has proposed a distributed solution to the Dining Philosophers problem that is suitable to various concurrent programming styles [Car82]. In the introduction to his paper, he describes the problem:

> Dijkstra's "dining philosophers" problem is one of the standard programming exercises in resource allocation. An arbitrary number of "philosophers" are seated at a circular table. In front of each is a plate of "a very difficult kind of spaghetti". Between each pair of philosophers is a fork. The philosophers alternately "think" and "eat" in an unpredictable manner. When thinking a philosopher requires no forks. In order to eat a philosopher must acquire the two adjacent forks. The problem is to provide a deadlock-free and starvation-free fork allocation discipline.

Cargill's solution is to label the forks consecutively around the table shown in Figure 3.1, and when a philosopher wants to eat he must pick up his odd-numbered fork before his even-numbered one. Each fork is under the control of an independent "fork proprietor". Each philosopher requests of a fork proprietor that a fork be allocated or released. If a fork is in use the philosopher waits until it is free. There is no other control in the algorithm.

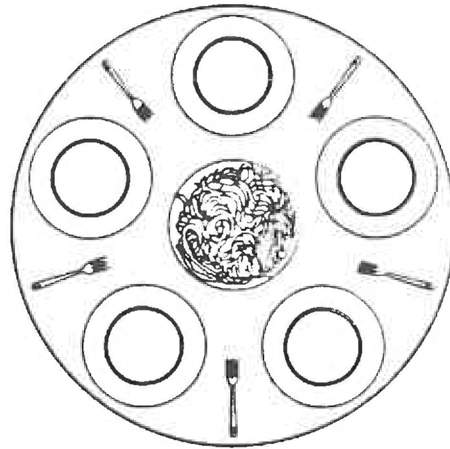


Figure 3.1 Dining Philosophers' Table

Cargill's solution is given in Ada. It includes three separate tasks: a maitre d', a fork proprietor and a philosopher. There is one maitre d' who assigns places at the tables to the philosophers, and N fork proprietors and N philosophers modeled by processes running in parallel. The programs in this study are modeled on his implementation.

The algorithm to describe the cycle of one philosopher is:

```
THINK
pick up odd fork
pick up even fork
EAT
put down both forks
```

The behavior of a fork proprietor is as follows:

```
allocate fork
release fork
```

Of course, a fork proprietor cannot allocate a fork if it is already in use. The actual

program representing the fork proprietor varies according to how a particular system handles synchronization. Throughout the simulation, each philosopher reports his progress by printing informational messages.

3.2. Matrix Multiply

In the matrix multiply program, matrices A and B are multiplied to form the product matrix C. Below is the body of a sequential version of the program written in C. Matrices were not necessarily square and could differ in size from each other.

```
for (i = 0; i < m; i++)
    for (j = 0; j < p; j++) {
        sum = 0;
        for (k = 0; k < n; k++)
            sum += A[i][k] * B[k][j];
        C[i][j] = sum;
        }
```

Gustafson and Hawkinson [GuH86] suggest that "matrix multiplication should execute at very close to peak theoretical speed on most scientific computers." This situation may be true on an array processor, but dividing the problem among general-purpose processors vastly increases the communication costs required to perform each multiplication and addition on a different functional unit. It would involve a pipelined algorithm where each process performed partial products and partial sums.

Matrix multiplication can be parallelized in many other ways with larger grain solutions. Each process can calculate one element of the product matrix using a row from the A matrix and a column from B. A larger grain solution is for each process to calculate an entire row or column of the product matrix. For this approach, each

process must know a row of the A matrix and the entire B matrix or *vice versa*. Finally, a process may calculate a submatrix of the product matrix. This solution involves little communication of data once the parameters for the computation have been established.

A fine-grain pipelined solution in occam, modeled after Hoare's iterative array solution in CSP, proved successful. Processes are arranged in a grid with communication links to each of four neighbors as shown in Figure 3.2. Each of the central processing nodes knows one element of the B matrix. Vectors of the A matrix enter the grid from the *west*. Central nodes find partial products and send their sums *south*. The sums emerging from the southern end of the grid are rows of the product matrix. Each central node sequentially receives an element of the matrix, performs a multiplication and addition and sends the element on. Details of the implementation of this algorithm are covered later.

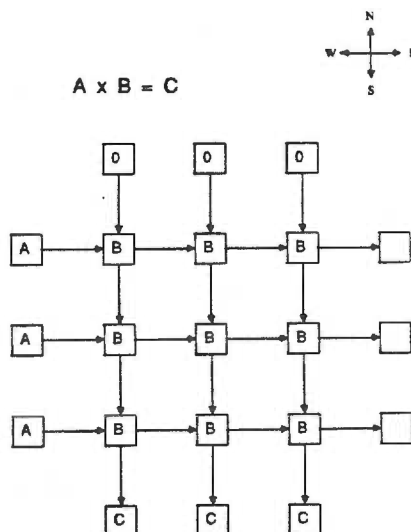


Figure 3.2 Matrix Multiply Grid for occam Solution

Such a solution was tried both on the Balance and on the iPSC, but resulted in a large number of extra calculations to partition the data. Because of the high overhead of pipelining at the instruction level, a larger grain solution was chosen. The solution chosen for the Balance was for each process to calculate a row of the product matrix. This algorithm was simple to implement on a shared-memory system. Its details are discussed later. For the iPSC, the largest grain solution was chosen to keep communication to a minimum.

3.3. Traveling Salesman Problem using Simulated Annealing

The traveling salesman problem involves finding the shortest route connecting a set of cities, visiting each city only once. The difficulty with the problem is that as the number of cities grows the number of possible paths connecting them grows exponentially (faster than any finite power of the number of cities).

The solution chosen uses simulated annealing to sample a subset of the possible routes to find an approximate solution [FKO85]. Starting with some initial configuration of paths connecting cities, at each iteration we go through the route and swap pairs of cities so that the total path length is shortened. If a swap makes the route shorter always accept it; if not, reject it. When no more swaps can be made, the algorithm stops. The problem with this method is that the path length can easily get trapped in a local minimum.

Simulated annealing uses a technique borrowed from statistical mechanics to help overcome local minima. Changes that make the route shorter are still accepted, but now a change that lengthens the route may also be accepted with a conditional

probability. The probability depends on the state of the entire system and a factor that the user controls. By occasionally accepting swaps that do not necessarily improve the length of the route, the system can escape from local minima.

By analogy with the physical process of annealing, the factor that the user controls is called *temperature*. As the temperature is lowered the system *freezes*, allowing fewer changes. Parameters can be controlled to avoid getting trapped in frozen states.

Researchers at the California Institute of Technology (CalTech) published a parallel version of this algorithm for a hypercubic MIMD computer [FKO85]. To communicate information, the nodes are mapped in a *ring* configuration. Each node knows the identity of a node on either side of itself. A node receives a section of the path containing a group of cities and swaps cities within its path and with its neighbors a number of times at each drop in the temperature of the system. The end result is usually a "good" total path length. A small example is shown in Figure 3.3

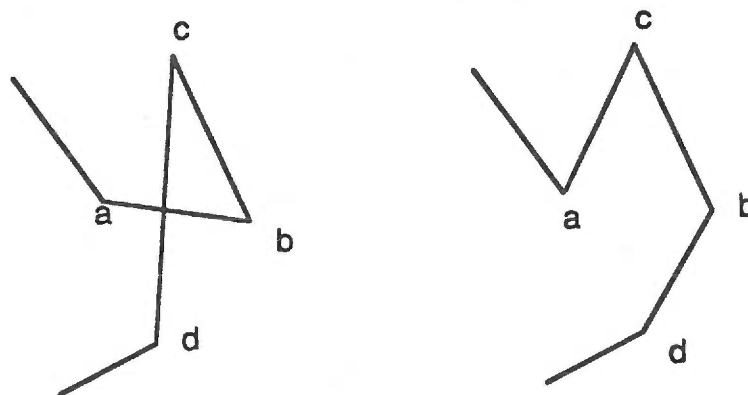


Figure 3.3 Shortening of a Traveling Salesman's Path

Chapter 4

Comparison of Three Parallel Programming Systems

Specification of concurrency by a programmer always entails some overhead. It may be simply a call to execute a routine on multiple processors, or it can mean a sizeable piece of code to determine the partitioning of the problem, including code, data and communication paths. This chapter considers these differences by studying examples of code from the three sample programs. The focus is primarily on the dining philosophers program, but examples from the other two programs are included when they illustrate important points. Comments on the code fall under three major components of parallel programming: how execution is divided among processing elements and how it is controlled, how data are shared, and how events are synchronized.

4.1. Division of Work: Designation of Parallel and Sequential Computation

The ideal parallel programming language would allow a programmer to specify an algorithm without regard to whether it will be run on a single or multiple processors and let the compiler restructure it as required. The Sequent Balance is the only one of the three environments studied that currently has such a facility, but it is limited. The Balance FORTRAN compiler provides a C$DOACROSS directive to partition DO loops. When the compiler encounters this directive, it restructures the code within the DO loop to execute concurrently on a specified number of processors. Unfortunately, technique is not general enough to handle all the potential concurrency

in algorithms.

In each of the systems examined here, a program must be divided in some way into parts that execute sequentially and parts that execute concurrently. The *grain* of the parallelism differs among the three systems. On the iPSC, large-grain parallelism must be emphasized for efficient computation. On the Sequent Balance and especially the transputer, various levels of parallelism can be achieved efficiently using parallel and sequential constructs.

4.1.1. iPSC

The programmer has to develop at least two separate programs for a homogeneous application on the iPSC: the host program on the cube manager and the node program for concurrent processes on the hypercube. At a high level, division of work on the iPSC is intuitive. The host program is responsible for communicating with the outside world and with the nodes. Typically, the host gathers data and disseminates it throughout the hypercube, then later collects data from the nodes and displays results. The nodes are used for the computation-intensive sections of the algorithm.

In the dining philosophers program, the maitre d' process runs on the host, as shown in Figure 4.1. Each node contains two processes, as in the original Ada implementation by Cargill [Car82], one representing a philosopher and the other a fork proprietor. In this particular implementation the number of philosophers is limited to the number of nodes in the hypercube because each philosopher and each fork is identified in communications only by its node number. A greater degree of parallelism could be simulated by placing more than one philosopher and fork process on each

node. Unique process identification numbers would have to supplement node numbers to insure proper routing of messages. Multiple processes on the same node are time-sliced by the node operating system or can be suspended under programmer control (via a command called `flick()`).

Aside from managing the node processes, the maitre d' process assigns places at the table to the philosophers. The simulation occurs in the node processes of the hypercube itself. After receiving a place assignment, the philosopher acquires two forks, "eats" and then replaces the forks and "thinks" for a random amount of time,

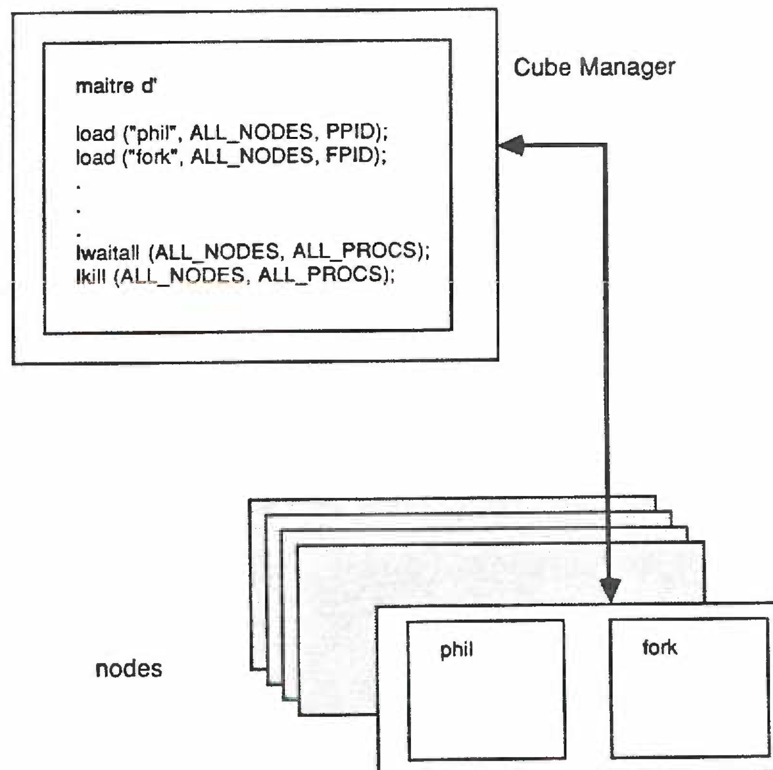


Figure 4.1 Dining Philosophers on iPSC Host

continuing these activities in a loop.

The simulation is monitored by `syslog()` messages to the host. Note that logging to a central file forces sequentiality into an otherwise parallel program. The overhead of sending messages to the host also slows processing.

Each fork proprietor manages control of one fork and has two functions: to make the fork `busy` and to make it `free`. Two philosophers share each fork. The fork proprietor waits to receive a message from one of the philosophers. If the fork is `free`, the proprietor allocates it to that philosopher by sending a message. If a philosopher's neighbor is using the fork, the proprietor remembers the node and process id of the requesting philosopher and sets a `wait_ptr` so that when the fork is `free`, a message can be sent to the waiting philosopher and the fork can be allocated.

If the philosopher is releasing the fork, it is set to `free`. The `wait_ptr` is tested to see if the other philosopher is queued for this fork. If so, `wait_ptr` is nullified, the fork made `busy` and a message sent back to the waiting philosopher to notify him that the fork has been granted.

The division of work in the dining philosophers program is straightforward since it has clear sequential and parallel components. In other applications, there are times when concurrency in the nodes must be interrupted so that a single stream of activity can take place. To achieve a single thread of execution nodes must synchronize and return control to the host process or a specified node process. Synchronization on the iPSC is described in Section 4.4.

4.1.2. Occam on Transputers

In occam, a programmer has control over parallelism at many levels. At the top level, a program may be divided into large-grain processes or procedures. At a finer level, any number of individual statements may be grouped to executed concurrently. An array of identical processes may be placed on a network of transputers to describe a homogeneous application, or a different process may be placed on each transputer for a heterogeneous application.

The dining philosophers program consists of a maitre d' process, and philosopher and fork processes that are replicated to run concurrently. Figure 4.2 shows how the maitre d' process on the host invokes the philosopher and fork procedures. On an actual transputer network, code to configure the processes on processors would be added to the code shown.

As a result of the `SEQ` construct in Figure 4.2, the maitre d' calculates the indices of each philosopher's forks before calling the philosopher and fork procedures. The parameters to `phil()` include `i`, the philosopher's identity and the particular channels that the process will need to communicate with the fork proprietors. Arrays of communications channels are declared as `CHAN`. The individual `phil()` process only needs to know four of those channels, two for his odd-numbered fork and two for his even-numbered fork. When this program is physically mapped to a transputer network, the channels must be statically mapped onto transputer links. The `PAR` immediately preceding the calls to `phil()` and `fork()` means that these two processes will execute in parallel. The replicated `PAR` at the top of the maitre d' process means

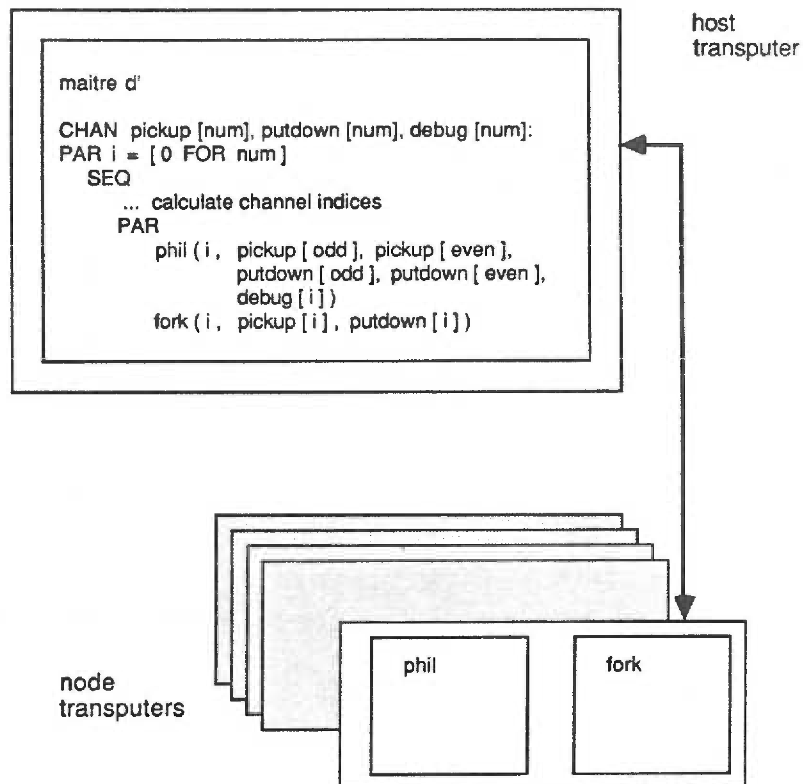


Figure 4.2 Dining Philosophers in occam I

that they will execute concurrently on num nodes.

In the code below, the ALT accepts the first input available from a process' debug channel. In the maitre d' process, the ALT is replicated to accept input from any of the philosopher processes. The purpose of the routine is to receive and print messages to the screen. In this occam I version, the channel receives only one byte of the message at a time, so it must loop to receive the entire message. After the message has been received and printed, the routine returns to the top of the WHILE loop to receive input on another debug channel.

```
WHILE TRUE
  ALT i = [0 FOR num]
    debug[i] ? msg.char
      ... code to output message to screen
```

This is another example where two high-level routines must be written at once. It is impossible to send a message from a routine before the screen output routine has been written to receive the message. Therefore, before the philosopher process could send messages, the screen output routine had to be included. It runs on the host in parallel with the maitre d'.

4.1.3. Sequent Balance

Under the Balance's microtasking model, the routine `m_fork()` can fork only one process per processor. The call locks a process onto a processor during execution. For this reason, a program must check that the number of requested processes does not exceed the number of available processors, leaving one processor for the operating system. Figure 4.3 shows part of the maitre d' code for the dining philosophers program. The master thread compares the number of philosophers to the number of available processors. If an application requires more processes than processors, the DYNIX `fork()` routine may be used to allow the operating system to allocate processes to processors.

The master thread initializes `n` slave processes and then forks them to run the function `phil`. The rest of the dining philsophers program consists solely of philosopher processes that run concurrently until the simulation is through.

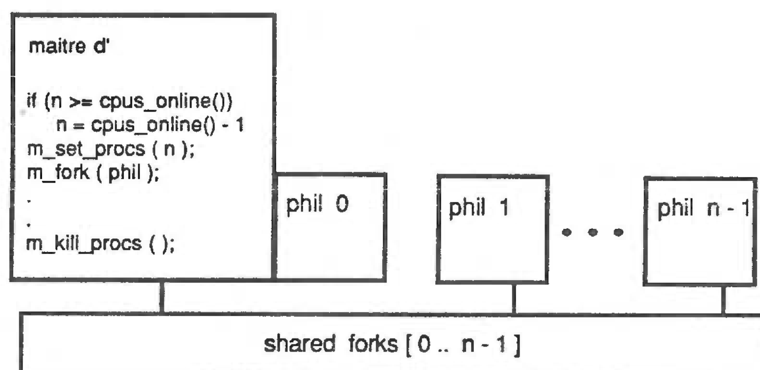


Figure 4.3 Dining Philosophers on Sequent Balance

A more interesting example of division of work on the Balance comes from a section of the traveling salesman program shown in Figure 4.4. The node processes in this program must repeatedly give control back to the host, while the host gathers more information from the user or performs calculations. Rather than dividing a program into two separate processes, host and node, it is possible to integrate single and multiple streams into one process using microtasks `m_single()` and `m_multi()`. A problem with this method is that the slave processes use processor time while they spin at the `m_single()` barrier. An alternative to this method would be to return to the host process, suspend the slave processes, execute the sequential code and then fork new processes. Using this technique, the slave processes do not waste processor time by spinning, and there is no overhead when the next `m_fork()` is issued.

On the Balance, all processors are capable of performing I/O. Since printing causes a side effect, print statements should be enclosed in a critical region using locks so that a block of statements from one process will print coherently. When the

```
m_single();                    /* master only */

for (currtemp = itemp; currtemp > ftemp; currtemp -= drop) {
      pathlen = 0;
      m_multi();               /* slaves calculate path lengths */
      mypathlen = 0.0;
      for (j = low, city = bottom; j <= top; j++,
        city = city->next) {
            mypathlen += distance(city, city->next);
            printf("myid (%d) path length = %f\n",
              m_myid, mypathlen);
            }
      m_lock();
      pathlen += mypathlen;
      m_unlock();
      m_single();             /* master prints statistics */
      printf("current temp = %6.3f ", currtemp);
      printf("current path length = %6.3f\n", pathlen);
      }

m_multi();                    /* slaves participate */
```

Figure 4.4 Division of Work in Traveling Salesman Problem on Balance

program requires that results from all of the nodes be printed by one process, an `m_single()` may precede the print statements, as shown near the end of the `for` loop in Figure 4.4.

4.1.4. Comparison of Division of Work

The way work is divided determines the overall appearance of a program. It influences the amount of communication and synchronization required and thus is the basis for determining program efficiency. Occam seems to present the clearest

representation of parallelism at many levels. Occam is based closely on CSP, and CSP is used to model parallel processes. Keywords that indicate parallel and sequential flow make occam programs easy to read. Identical syntax is used to achieve both large- and fine-grain parallelism.

At a high level, division of work on the iPSC also seems clear. A process on the Cube Manager controls node processes that run concurrently. Only the host can invoke parallel processes, so fine-grain parallelism is precluded by the significant overhead.

The two distributed systems share one problem. Control flow can be difficult to trace since it changes with the passing of messages. To read the text of a program, message sends and receives have to be matched. This is slightly easier in occam because of named channels, as will be discussed in the next section.

Explicit code for forking processes and specifying single and multiple streams make parallelism and control flow evident on the Sequent Balance. The problem is that these microtasks are not the only methods for specifying parallelism on the Balance. Processes may be forked using a UNIX fork or a microtask fork. Once multiple processes are running, a number of routines are available to enforce sequential execution; some of them are not as apparent as others. Locks and synchronization mechanisms have side effects that are described in the following two sections. There is no easy way of achieving the fine-grain parallelism available with occam. At this time, microtask forks cannot be nested. Even if they could, the overhead for forking a single-statement process would be high.

4.2. Sharing of Data

Parallel tasks would be much easier to program if there were no sharing of data between sub-tasks or processes; however, in practical applications data is required. This section examines how processing elements access data.

4.2.1. iPSC

On the iPSC, the programmer is responsible for managing message consistency. The primary requirement of message passing is that the sender must have the address of the receiver, but message formats require more than just an address.

In the dining philosophers program, after determining the number of nodes in the hypercube, the maitre d' assigns a `left` and `right` fork to each philosopher, sending the odd-numbered one first and then the even-numbered one, thus establishing his place at the table. The node processes are waiting for this information in order to begin the simulation. Figure 4.5 shows how the host process sends a message to each of the nodes. The parameters that indicate the message destination are node `i` and process id `PPID` for the philosopher process on that node.

The message buffer `send_buf` must be a character string, so it is necessary to transform the integer fork identities into characters. If there are never more than 128 nodes, there will not be more than 128 forks, so one character is sufficient to store the fork identity, (but this means that our program is limited to running on 256 or fewer processors). The size of the message buffer in this case is 2 characters.

The philosopher process will communicate with the host, so it must remember the host name and process id that came with the message containing its place

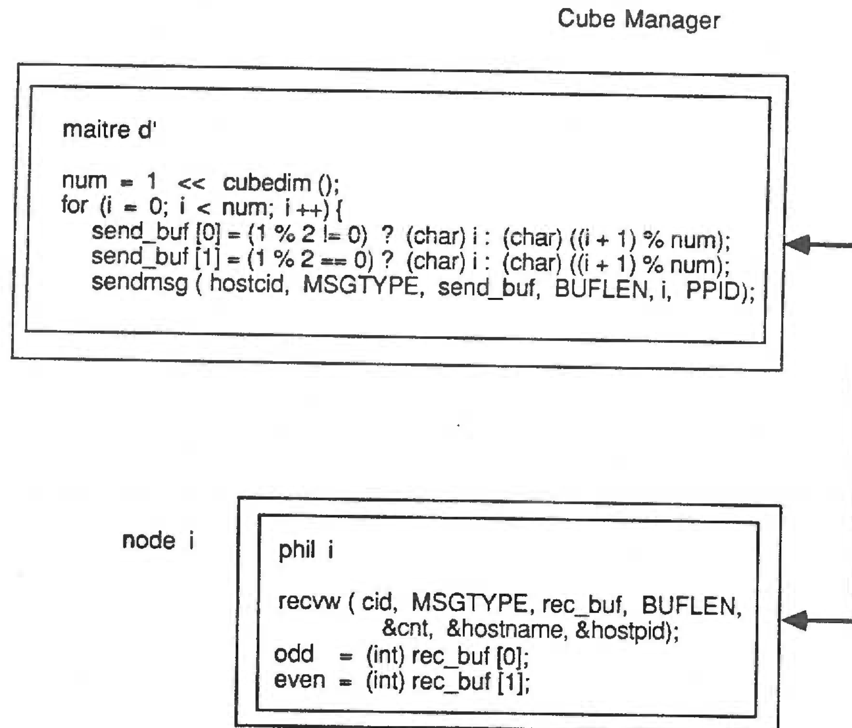


Figure 4.5 Host-Node Message Passing in Dining Philosophers on iPSC

assignment. The node numbers of the neighboring forks are unpacked from that message's buffer.

For the remainder of the simulation the philosopher and fork processes communicate in a cycle. The philosopher process requests a left and right fork by sending an `allocate` message to the fork proprietor. The requests will always be filled, but the philosopher may have to wait awhile in one or both fork processes if the forks are in use. A confirming message is used to prevent the philosopher from continuing before he has received both forks. When the philosoper is done eating, he releases his forks with a `release` message to the appropriate fork proprietors. Figure 4.6 shows two philosopher processes trying to pick up the same fork.

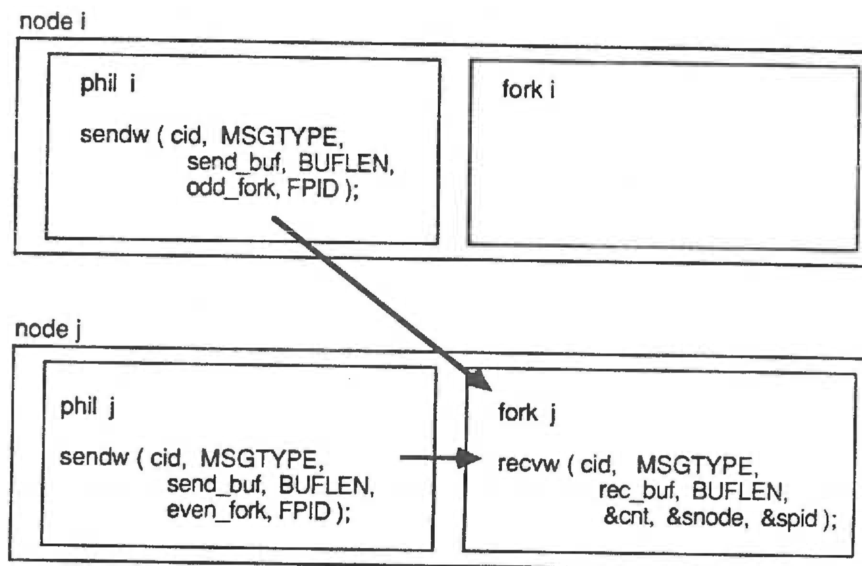


Figure 4.6 Node-Node Message Passing in Dining Philosophers on iPSC

The message-passing sections of the philosopher and fork proprietor processes are shown in Figure 4.7.

Aside from making sure that all message parameters are properly included, certain parameters can be helpful or may present complications, depending on the experience of the user.

The original CalTech code for the traveling salesman problem also required message length specification. Their solution was portable to the iPSC. Each message consisted of one packet of information with two floating point numbers. Messages were either commands to the nodes or data. When the host sent a command to the nodes, the first item in the packet was a predefined value indicating the command. The second item in the packet could be used for a numeric value if needed. For instance,

From phil on nodes

```
send_buf[0] = (char) Allocate;
sendw(cid, MSGTYPE, send_buf, 1, odd, FPID);
recvw(cid, MSGTYPE, rec_buf, BUFLEN, &cnt, &snode, &spid);
sendw(cid, MSGTYPE, send_buf, 1, even, FPID);
recvw(cid, MSGTYPE, rec_buf, BUFLEN, &cnt, &snode, &spid);

send_buf[0] = (char) Release;
sendw(cid, MSGTYPE, send_buf, 1, odd, FPID);
sendw(cid, MSGTYPE, send_buf, 1, odd, FPID);
```

From fork on nodes

```
recvw(cid, MSGTYPE, rec_buf, BUFLEN, &cnt, &snode, &spid);
action = (actionkind) rec_buf[0];

switch(action) {
case Allocate:
   if (fork == Not_Busy) {
      fork = Busy;
      sendw(cid, MSGTYPE, send_buf, strlen(send_buf),
        snode, spid); }
   else {
      waiting.snode = snode;
      waiting.spid  = spid;
      wait_ptr = Occupied; }
   break;

case Release:
   fork = Not_Busy;
   if (wait_ptr == Occupied) {
      fork = Busy;
      wait_ptr = Empty;
      sendw(cid, MSGTYPE, send_buf, strlen(send_buf),
        waiting.snode, waiting.spid); }
   break;

default:                        /* error */
   break; }
```

Figure 4.7 Communication between fork and phil Processes on iPSC

the command to *set* the temperature could also send the temperature, or the command to *iterate* could specify the number of times to iterate. All other messages consisted of a *point* with floating-point x and y coordinates. The constant message length meant one less parameter to calculate. The only data coercion necessary was character string to float, and *vice versa*, and that was not difficult.

In Figure 4.8 the host process instructs the nodes that it intends to *write* data points from an input file to the nodes. In addition to the `write` command, it also sends the number of points in the path. The message is only sent to node 0 because it will be disseminated throughout the nodes using a spanning tree. (The routine `grecvw()` [Mol--], which is a global receive, spares the expense of the host sequentially sending messages to each node. A recent version of the iPSC System Software (R3.0) allows the host to efficiently broadcast a message to the nodes by altering only one parameter in `sendmsg()`.) The host then sends the x, y coordinates, one per packet, to node 0. Each node *keeps* a section of the path and sends on the points that are not on its path.

By the nature of the algorithm, the traveling salesman messages had to be short and frequent. A more efficient use of the iPSC is to use fewer, but larger, messages since each message carries an overhead of at least one kilobyte. When the programmer tries to pack as much data as possible into each message, the nature of the message buffer can complicate coding. In one program it was helpful to write a special macro to handle data packing and unpacking.

From `wrtcon()` on host

```
packet[0] = WRITE;
packet[1] = (float) numpts;
sendmsg(hostcid, GTYPE, packet, PACKET_LEN, NODE_O, PID);

for(i = 0; i < numpts; ++i){
     fscanf(fp,"%f %f", &packet[0], &packet[1]);
     sendmsg(hostcid, DTYPE, packet, PACKET_LEN, NODE_O, PID);
     }
```

From `writecom()` on nodes

```
grecvw(cid, GTYPE, (char *) packet, PACKET_LEN, cnt, dim);
~
~
for(i = 0; i < put; ++i){
     recvw(cid, DTYPE, (char *) packet, PACKET_LEN, &cnt, &snode,
          &spid);
     sendw(cid, DTYPE, (char *) packet, PACKET_LEN, dest, PID);
     }

for(i = keep + 1; i > 1; --i)
     recvw(cid, DTYPE, (char *) ppoint[i], PACKET_LEN, &cnt, &snode,
          &spid);
```

Figure 4.8 Message Parameters in Traveling Salesman Problem on iPSC

```
temp = p;
stuff(temp, int, part->x_start);
stuff(temp, int, part->x_dim);

for (i = part->x_start; i < part->x_start + part->x_dim; i++)
    for (j = 0; j < P; j++) {
        c[i - part->x_start][j] = 0;
        for (k = 0; k < N; k++)
            c[i - part->x_start][j] += a[i][k] * b[k][j];
        stuff(temp, double, c[i - part->x_start][j]);
      }
sendw(cid, UTYPE, p, sizeof(PART) + subsize, hostname, hostpid);
```

Figure 4.9 Data Partitioning in Matrix Multiply on iPSC

The code in Figure 4.9 is from the node procedure of the matrix multiply program. It was by far the most complicated example of packing information into message strings. Rather than send a single element or row of the product matrix back to the host, each node computed a submatrix consisting of several rows of the product matrix and sent it all back to the host in a single message. For the data to be meaningful, the packet also had to include parameters to specify the starting row and number of rows of the submatrix. A data structure described the parameters and the data. In this section of code, the parameters and data are coerced into the character message buffer as soon as they are known using a macro called `stuff`. Finally, it was necessary to calculate the size of the message buffer in bytes. A true application would have also needed to handle a message that exceeded 64 kilobytes, the maximum size for the program model.

The message parameter `type` specifies the message type and is user-definable. Its purpose is to identify the message or its content. Cube manager processes and node processes use `type` differently. On the nodes, a message is received only if the `type` specified in the send matches that in the receive. On the cube manager the receipt of a message does not depend on its type. It is up to the programmer to check the type and determine how to handle the message.

From `wrtcon()` on host

```
for (i = 0; i < numpts; i++) {
     fscanf(fp, "%f %f", &packet[0], &packet[1]);
     sendmsg(hostcid, DTYPE, packet, PKT_LEN, NODE_0, PID);
     }
```

From `iterate()` on node

```
/* Send point to prev node; receive point from next node */

sendw(cid, TYPE1, (char *)ppoint[2], PKT_LEN, prev, PID);
recvw(cid, TYPE1, (char *)ppoint[keep + 2], PKT_LEN, &cnt,
  &snode, &spid);

/* Send point to next node; receive point from prev node */

sendw(cid, TYPE2, (char *)ppoint[keep + 1], PKT_LEN, next, PID);
recvw(cid, TYPE2, (char *)ppoint[1], PKT_LEN, &cnt,
  &snode, &spid);
```

Figure 4.10 Type Parameters in Traveling Salesman Problem on iPSC

One problem in the traveling salesman program involved the `type` parameter in the `sendw()` command; the code is shown in Figure 4.10. At each drop in temperature the entire system synchronizes and sends the total path length to the host. For each iteration a node first tries to shorten its own path. Then, in three steps, the node first sends a city to its previous node and decides if it should swaps cities in its own path, sends a city to its successor node and swaps, and again sends a city to its successor node and swaps. The mistake is to think that, since these send/receive steps happen sequentially in each node, they are synchronized with each other. They are not. It is easy to get a race condition where a node is accepting a message from its previous node before it receives a message from its successor node because the previous node happens to have sent a message first. The result is that cities mysteriously drop out of the path when they are swapped between nodes. There are two solutions to the problem. One is to have all nodes synchronize after each node swap. This approach would be inefficient and involve extra message transmissions. An easier solution is to change the type of each message so that when a message of `TYPE1` is sent it can only be received by a `recvw()` expecting a `TYPE1`.

The iPSC programmer must ensure consistency over a number of message parameters, often in physically separate files. The compiler may catch some errors, but many are left to create illogical results, race conditions and deadlock.

4.2.2. Occam on Transputers

On a transputer-based system, the programmer does not have to manage messages explicity. Message passing in occam is simple in some respects: (1) messages are

not buffered, hence eliminating the need for a message buffer associated with each channel and message length parameters; and (2) two processes must synchronize to pass data, thus eliminating a message queue and preventing the possible loss of data.

Figure 4.11 demonstrates simple message transfers, where a philosopher communicates with each fork proprietor in occam I. To pick up his odd-numbered fork, the philosopher sends out a message on the channel `pickup.odd`. The occam I keyword ANY may be used in place of an actual message when the content of the message is irrelevant, as in this case. The fork process knows the meaning of the message by the channel on which it arrives.

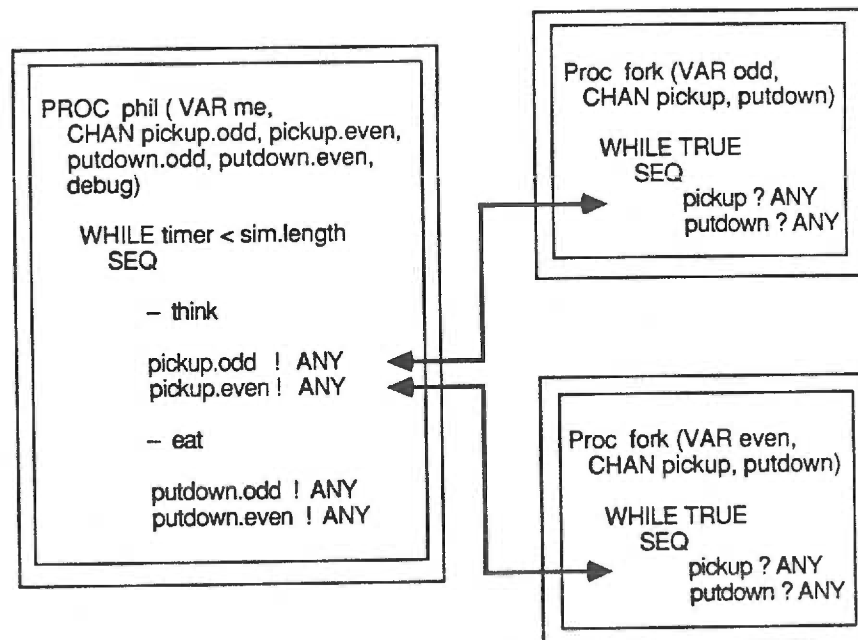


Figure 4.11 Message Passing in Dining Philosophers in occam I

Each fork proprietor executes a loop to allow a philosopher to pick up his fork and put it down. Control is achieved by the order in which messages are sent. If a fork is busy, a request to pick it up will not be received until communication has occurred on the putdown channel.

The occam matrix multiply program provides an example of data sharing in a pipelined program. Figure 4.12 shows the process matnode. These processes reside on the central nodes of the grid that was shown in Figure 3.2. They receive data from east and north nodes and send data to west and south nodes. As in the earlier example, a node's communication channels are sent to it as parameters. The programmer's task is simply a matter of directing the data to the appropriate channel and determining which activities may take place in parallel. Here, matnode is receiving a data value x on channel east from its eastern neighbor and then sending that value out channel west to its western neighbor. The declaration of those channels was CHAN WE[(m * n) + m], an array of channels crossing a grid from west to east, dimensioned by the size of the matrix (plus an extra column). Two elements of that array were passed to each matnode process as its west and east channels when it was invoked. When one matnode process sends the value x out its east channel, its neighbor process will receive the value on its west channel.

A benefit of the ALT construct is demonstrated in Figure 4.13. It can eliminate the need for an explicit message. An action can be taken depending upon which channel is activated first. It does require the declaration, and eventually the physical mapping, of additional channels. In this example, the host sends an arbitrary (ANY)

```
PROC matnode(VALUE y, CHAN east, west, north, south) =
  VAR k,                                    -- loop index
      x,                                    -- element of A
      sum:                                  -- partial sum
  SEQ k = [0 FOR l]                         -- for each row in A
    SEQ
      PAR
        east ? x                            -- receive element
        north ? sum                         -- receive partial sum
      PAR
        west ! x                            -- to next column
        south ! (x * y) + sum               -- send partial sum
      SKIP:
```

Figure 4.12 Pipelined Matrix Multiply Processes in occam I

message to the nodes on a particular channel indicating the command. A more familiar method of achieving the same result would have been to send the command as a message on a generic channel and have the nodes interpret the command.

Although occam does not buffer messages, the programmer may wish to add buffering routines. A buffering routine running as a separate process can capture data without forcing the original two processes to synchronize. In addition, occam I programs need routines to buffer their single-word messages and convert them into typed data. A program with buffering and data-conversion routines sacrifices some readability since these routines may obscure the presence of message traffic. Occam II has data types and eliminates the need for extensive buffering. In exchange, channel declarations must specify the type and amount of data that may travel over them.

From `run.nodes()` in Traveling Salesman Problem

```
-- global declarations
BYTE ANY:
[numnodes] CHAN OF BYTE anneal.cmd, exit.cmd, read.cmd,
   write.cmd:

PROC run.nodes(VAL INT me, CHAN anneal.cmd, exit.cmd, read.cmd,
   write.cmd)
  -- local declarations
  SEQ
    WHILE NOT done
      ALT
        write.cmd ? ANY
          -- receive points from host
        anneal.cmd ? ANY
          -- shorten path using simulated annealing
        read.cmd ? ANY
          -- send points back to host
        exit.cmd ? ANY
          -- stop this loop and send stop message to screen
:
```

Figure 4.13 ALT in Traveling Salesman Problem in occam II (beta)

4.2.3. Sequent Balance

In the dining philosophers program, the function of the `fork` proprietor is really that of a binary semaphore, ensuring that a fork is either `busy` or `free`. On the Balance version of the program, the forks are represented simply as a shared array of binary semaphores, or locks. The declaration is shown in Figure 4.14.

A second shared variable `place` allows the philosophers to assign their own places at the table by updating a shared counter. This variable needs no associated

lock since it will be locked by `m_lock()`.

In order to find his place at the table, each philosopher must look at the shared variable `place` and increment it. A microtask lock was chosen for this part of the algorithm because it is simpler to code than a primitive lock, and the resulting execution would have been similar in either case. Each slave process must access the `place` variable once at the beginning of the program, so either type of lock would have required the slaves to execute sequentially.

Microtask locks were used exclusively in the traveling salesman program. Since `m_lock()` creates a critical section, the entire program runs sequentially whenever data is locked. The work-intensive loop of the program is in the `iterate()` routine. Over multiple iterations, processes try to shorten their section of the path, which is stored in a shared array. A routine called `cswap()` determines if two points in a path should be swapped. In making this decision, four points are used to calculate distances. Originally this entire routine was locked. If it was not locked, other processes could change the same points that one process was considering.

Timing the program for different numbers of processes revealed that no matter how many processors were applied, the time remained the same. This behavior resulted because all of the calculations in the routine were locked. The program was behaving as a sequential program. The solution was to lock only the section of code where data points were actually swapped. Such a solution would normally be unsafe. It could change the actual value of a point that a process was considering. In this particular case, it would only affect the border points and so only be likely with a

```
shared slock_t forks[n];                    /* global declarations */
shared int place;

void                                        /* philosopher */
phil() {
int odd, even;

m_lock();                              /* get my place assignment */
odd  = (place % 2) != 0 ? place : (place + 1) % n;
even = (place % 2) == 0 ? place : (place + 1) % n;
place++;
m_unlock();

while (1) {

    /* think */

    s_lock(&forks[odd]);                    /* pick up both forks */
    s_lock(&forks[even]);

    /* eat */

    s_unlock(&forks[odd]);                  /* put down both forks */
    s_unlock(&forks[even]);
    }
```

Figure 4.14 Data Sharing in Dining Philosophers on Sequent Balance

small data set. In addition, the decision to swap data points is partially random, so a little more randomness should not influence the outcome to a great degree. With this change, speedups were linear according to the number of processes added. This experience points out one of the hazards of a shared-memory system. The microtask `m_lock()` was the only easy way of accessing such a large global data structure.

Locking individual sections would have complicated the program.

4.2.4. Comparison of Data Sharing

The mechanisms of sharing data may influence how an algorithm is implemented. A message on the iPSC carries the overhead of a kilobyte-long message buffer. Consequently, an application must have a high computation-to-communication ratio to be efficient. A second deterrant to sending many messages is coding of their complicated syntax. A possible solution is for the programmer to write higher-level routines to call the message sends and receives, but this still involves added work. On transputers, messages are efficient. Traveling on named channels, messages require no parameters and so are simple conceptually. Messages proliferate in occam programs. In applications that require many types of massages, however, occam messages can be as complicated as those on the iPSC because of the need for explicit data typing.

A problem that distributed systems share is global transfer of messages. The iPSC's global Ethernet link between the host and nodes automatically routes messages from the host to any node, whereas a host transputer can only be connected to at most four other transputers, so each transputer must run a process to forward messages for global host-node transmissions.

Access to shared memory requires locks instead of messages. The important decisions for the programmer are what type of lock to use, a primitive multitasking lock or a microtask lock. The choice can have a large impact on program efficiency depending upon the number of processes that are blocked.

For a programmer accustomed to sequential programming, it may be easier to conceptualize shared memory than distributed memory because the data resides in one location. Large data structures in shared memory still have to be partitioned. Locking and unlocking data is conceptually easy, but the result may be cluttered code with a barely visible algorithm, or a supposedly parallel program that actually runs sequentially.

4.3. Synchronization Methods

A process in a distributed system runs autonomously until it needs to share information or wait for a event. Since data is not globally available to all processes, as it is on a shared-memory system, there is no need for synchronization to ensure mutual exclusion or to protect shared variables. Synchronization is needed to ensure proper ordering of events. Data must be explicitly sent and received, so messages enforce synchronization. They insure that a process only receives data at the proper time and in a proper state. When messages are used only for synchronization, they need not contain data. Synchronization on a distributed system does not require that all processes be exactly the same point in execution at a given time. It simply means that they have all passed through the same state.

On a shared memory system, all processes have access to global data. A process can access a variable before the variable is in a proper state or after it has changed to a new state. Therefore, mechanisms are required to force processes to access the variable only when it is in a suitable state. Proper access to a variable can be assured by repeatedly testing its state. This method can also be used to control the order of

events. Another technique is to force all processes to wait at some point in execution until a specified number of other processes have also arrived. Processes also need a mechanism to ensure mutual exclusion so that only one process at a time may update a variable or execute a section of code. This is often accomplished by locking out other processes.

4.3.1. iPSC

The dining philosophers program requires synchronization of events in two places. First, the philosophers cannot commence their cycle of eating and thinking until they have their place assignments. The host sequentially sends messages containing this information to each of the nodes. Second, the philosophers cannot eat until they have acquired both their left and right forks. Here they must synchronize with the two other philosopher processes that share their two forks. The fork proprietors act as binary semaphores that allow the forks to be in one of two states, `busy` or `free`. Only one philosopher can use a fork at a time.

In the traveling salesman program, at each drop in temperature, nodes must finish processing before the host process can proceed. Such a transfer of control requires some form of global synchronization. A simple method might be one similar to that used at the conclusion of the dining philosophers program, where the host waits for a message from each of the nodes. However, in the traveling salesman program time is a factor, and numerous node-to-host messages cost time. A method that is presumably more expedient is a global synchronization message, where a message visits each of the nodes before the final node passes it on to the host. The CalTech

traveling salesman program used a routine where each node contributed a byte of information to a packet that was transmitted to the host. A similar routine for the iPSC by Intel is gop() [Mol--], in which a *global operation* is performed on data from all nodes. For this implementation, the routine was modified slightly to perform a *no-op*. In this way, the lowest nodes in a spanning tree initialize a global synchronization of the nodes, and node 0 sends the message to the host when all of the nodes have been synchronized. The difference between gop() and grecvw() is that the latter routine begins with one node and directs a message down the tree; the former starts at the leaves of a tree and completes when the message reaches node 0.

As mentioned in a previous section, the newest release of the iPSC operating system has a broadcast message capability from the host. This capability slightly eases the job of synchronizing processes from one direction. Synchronization from the nodes, however, must be performed as described above.

4.3.2. Occam on Transputers

Synchronization is built into the occam language. Two processes must synchronize in order to communicate. In fact, communication over channels is the only form of synchronization provided. To order events among parallel processes, nodes can send and receive an arbitrary (possibly empty) message. The word ANY, which was a keyword in occam I and is a declared variable in occam II, is typically used.

A host process can synchronize a number of similar node processes by performing a message send inside a replicated PAR construct. In the code below, the host process of the traveling salesman problem sends an arbitrary message to the nodes on a

channel that indicates a read-data command. (This construct is not necessarily supported in a physical implementation of an occam program, a problem that is discussed in Chapter 5.) The channel that is used for synchronization can convey implicit information. The content of the synchronization message can provide explicit information.

```
PAR i = 0 FOR numnodes
  read.cmd[i] ! ANY
```

4.3.3. Sequent Balance

A shared-memory system requires two types of synchronization: *condition synchronization*, a method of ordering the execution of events, and *mutual exclusion*, where a section of code must be indivisibly executed by one process [AnS83]. The parallel programming library on the Balance offers several mechanisms for both forms of synchronization.

Conditional synchronization can be accomplished by having processes spin at a barrier or by having them wait for a shared variable to change state. At the primitive level, the multitasking library allows initialization of a rendevous point for exactly `n` processes using `s_init_barrier()`. Processes issue a `s_wait_barrier()` command and wait until `n` processes have arrived at the same point in execution before continuing.

The microtasking library provides a more general form of conditional synchronization that involves all processes forked by an `m_fork()`. Each worker executing an `m_sync()` spins at a barrier until all workers have arrived. The microtask

m_single() also synchronizes slave processes and should be used when a single stream of execution is required. Slave processes wait until a corresponding m_multi() is executed by the master process.

A second method of conditional synchronization is to have processes wait for the state of a shared variable to change. This strategy is simply a matter of data sharing and requires the same locking mechanisms used for data sharing.

For mutual exclusion, a process executes a sequence of statements surrounded by the microtasking calls m_lock()/ m_unlock(). For instance, the dining philosophers program uses microtask locks whenever the philsopher processes prints progress messages during the simulation.

4.3.4. Comparison of Synchronization Methods

The concept of synchronization encompasses flow of control, so many of the earlier comments on division of work apply here. Since messages are the synchronization mechanism on distributed systems, the programmer only needs to learn one concept. Implementation of synchronization messages may require a great deal of work if messages have to be distributed throughout a network of processes, but the same methods would apply to global dissemination of data. The Balance's explicit library calls for synchronization are simple to use, but again the two programming models, multitasking and microtasking, provide numerous ways to synchronize processes.

Chapter 5

Parallel Program Development

One striking difference among the three programming environments was the length of time it took to develop the programs on each. As mentioned earlier, the traveling salesman problem was ported from a CalTech implementation written for a hypercube MIMD computer similar to the iPSC. Although the only actual changes required for the iPSC were substitution of parallel library calls, process mapping routines and `makefile`'s, the port took a month of full-time work. It was a large program, and part of this time was spent understanding the algorithm, but the majority of it was spent debugging message traffic, handling system software failure and constructing the `makefile`. When the program was finally finished, message traffic was so slow that only very small data sets (16 cities) produced reasonably straight paths during interactive runs.

Development of the program in occam was slow because of the low level of the language and by the need to incorporate I/O and data-handling utilities, which were borrowed from other programs or written as required. A port from VMS occam I to the beta release of occam II on a Transputer Development System also influenced the development time. In all, this work spanned two months.

By the time the program was ported to the Sequent Balance, the algorithm was familiar, but surprisingly, it took only three days to write and debug.

The large differences in program development time were due in part to the parallel programming toolsets available on each system. This chapter explores some of the possible reasons for the discrepancies in development times, including how a program is conceptualized, coded and debugged. Another factor is how familiar the programmer must be with the physical machine in order to implement a logical algorithm.

5.1. iPSC

The XENIX operating system on the iPSC cube manager resembles UNIX but in addition supports system calls to manage nodes on the hypercube. Since programs are developed on the cube manager, the environment is familiar to a UNIX programmer. Programs are developed as separate modules, usually consisting of a host process and one or more node processes. These processes may be compiled using a `makefile` just as on a conventional uniprocessor, with the exception that host and node processes require different flags and files for linkage.

A programmer often approaches the iPSC with a sequential program that is to be converted to a parallel one. An algorithm may be parallelized in a number of ways, including replicating a section of code and dividing the data equally among multiple processes or pipelining data through the processes, or partitioning the algorithm into separate functions that run concurrently. Perhaps the easiest type of parallel program to describe on the iPSC is the homogeneous one, where the nodes execute identical processes.

In addition to the functional division of the program, the data may have to be partitioned. The programmer must determine whether the division is to be

accomplished by the host process before data is sent to the nodes, or in the nodes themselves after they receive data.

The next consideration is how the nodes are to communicate. Each node in an n-dimension iPSC hypercube is physically linked to n neighbors via bidirectional communication paths and is identified by an n-bit binary address, as shown in the physical representation of the hypercube in Figure 5.1. A node's nearest neighbors are nodes whose addresses differ by a single bit. When a hypercube topology is not the most efficient in terms of the distances messages must travel, software *mappings* can change the topology used by an application.

The traveling salesman program uses a form of mapping that minimizes communication distances. Each node in this program swaps cities with two neighbors.

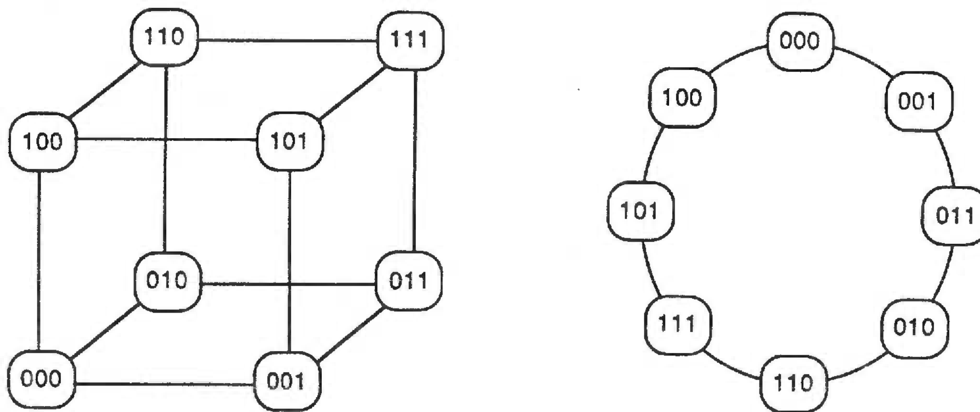


Figure 5.1 Physical and Gray Code Representations of Hypercube

This communication implies a ring configuration. To find a mapping where each node communicates with only two of its closest n neighbors, the node's binary address is mapped to a Gray code. A node in the traveling salesman program learns its own address by calling `mynode()` and then uses routines to determine its *predecessor* and *successor* nodes whose addresses differ from its own by a single bit.

After a mapping has been determined, communication and synchronization calls may be added to the code. A simple application might localize all concurrent activity in one routine so that the entire program consists of nodes computing and returning a result to the host. In this type of algorithm synchronizing messages are not required, but most algorithms are not this simple, so the programmer must determine how to synchronize node processes to transfer control between the host and nodes.

The conceptual barrier that makes programming on the iPSC difficult is that separately compiled routines must communicate with each other as a cohesive system. Message sends and receives between routines in physically different files must match. On the positive side, because of the separate files, it is easy to distinguish the code that is executed sequentially on the host from that that is executed in parallel in the nodes.

When a parallel program has been completely coded, it may not be easy to follow the logic of the algorithm because of the addition of sends, receives and system logs, and often substantial code for partitioning data and mapping. Since the flow of control can go from host to nodes and back again, and since synchronization looks like any other message, it is often difficult to trace the execution of a program.

To run a program on the hypercube, the programmer executes instructions from the cube manager. The programmer controls the number of nodes used in an application by *initializing* the iPSC to the desired dimension. The value of the cube dimension is available to the running process. Reinitialization is required to change the cube dimension; it is not possible to alter it by command-line arguments to a program. Other commands allow the programmer to log process messages to a local file and to kill processes on the nodes after unsuccessful program runs.

Debugging an iPSC program usually means tracing communication. The sorts of things that tended to go wrong during program development were improper mapping functions, resulting in messages going where they were not intended, improper message parameters, resulting in incorrect message contents or destinations, and improper data partitioning. A combination of approaches may be used to debug a program on the iPSC. The first step is often to run a program on a cube of dimension 0 (1 node). Such low dimension cubes may be used to debug message passing; however, passing this test does not ensure that the program will run on a higher dimension cube without problems. The nodes have no I/O capability other than messages, so messages must be sent to a system log file using the `syslog()` command. This is a costly procedure that can interfere with program timing and detection of deadlock, and typically results in a large log file. It also requires recompilation of the program each time messages are changed. Using this method with larger cube dimensions is especially impractical.

Deadlock is one of the most common symptoms of a malfunctioning program on the iPSC. The causes of deadlock are either that all nodes are waiting to receive a message but none has been sent, or that the message queues have been filled but no nodes are waiting to receive the messages. The easiest way to spot deadlock is by the indicator lights on the front panel of the iPSC. The lights exhibit different patterns depending upon whether a node is sending or waiting to receive a message. This panel allows the programmer to *observe* the operation of the program, and certain light patterns indicate abnormalities. This feature makes it almost imperative that the programmer be working near the hypercube.

The debugging procedure that is possibly most helpful is to visualize node interactions. Unfortunately, at the time of this study the only method for visualizing iPSC processes was by pencil and paper. Concurrent with this study, however, another student was developing a tool to visualize iPSC program behavior called the Parallel Programming Event Monitor [BrT87]. It shows the distribution of work among the nodes, communication paths and the amount of time spent communicating versus computing. It can be helpful in tracing events leading up to deadlock or starvation. Two of the completed programs in this study were used to test the monitor. It appeared to be a vast improvement over hand-tracing methods.

A final comment concerns timing on the iPSC. Each node has an independent clock, which is not synchronized with the clocks of other nodes. The library includes a routine `clock()`, callable from a node process, which returns the number of milliseconds interval that have elapsed since the node was initialized. To use this rou-

tine, all nodes must calculate individual times and then send them to the host process. To find the total time of a process performed on multiple nodes, it is easier to calculate elapsed time in the host process using the C Library `times()` routine.

Programming the iPSC is a process that appears familiar at first but becomes complicated in practice. Visualization tools will be a valuable aid to program development. One of the greatest problems is to find algorithms suitable for the hypercube that keep communication and synchronization calls to a minimum and use appropriate topologies and simple mapping routines.

5.2. Occam on Transputers

An occam program is developed in two steps, first as a *logical* program that simulates a network of transputers on a single processor, and then as a *physical* program that is configured to run on a network. Logical programs are written and debugged on a host computer using the Transputer Development System (TDS) [Inm85b]. TDS includes a special *folding* editor and utilities for compiling, filing, and running programs. Later the logical program is mapped to a physical network of transputers using TDS configuration utilities.

The folding editor is designed to help visualize a program as a group of modules. Cohesive sections of code may be hidden from view on the screen by *folding* them in a manner similar to folding a piece of paper, and then labeling the fold. Figure 5.2 illustrates the concept of folds, which are represented by the notation "...", followed by a label. A directory holds a single program or a number of related programs. Using the editor, the programmer organizes the program as a tree structure, starting with a

generic top-level file and adding a substructure of program modules in folds. The programmer may enter code or include existing files; folds may be nested and filed.

All support definitions and functions such as I/O routines must be present in a program. The programmer must ensure that functions are within the scope of the calling process. By filing folded functions, functions may be referenced in any number of places without increasing the size of the program.

```
PROC host(CHAN keyboard, screen)
  ... tdshdr                              -- header values
  ... procs                               -- user routines
  ... numeric io routines
  ... tsm declarations

  ... node.tsr       node controller      -- filed fold
  PROC run.nodes(...)
    ...                                   -- folded code
  :

  ... host.tsr       host controller      -- filed fold
  ... write.point                         -- fold containing PROC
  PROC write.point(...)
    ... write point to screen             -- folded code
  :

  PAR
    ... screen output                     -- folded code
    ... run node processes
    ... user input loop

:                                         -- delimiter of PROC host
```

Figure 5.2 High-Level View of occam II (beta) TSM Program

Utilities in TDS are not to be confused with I/O routines and other user-written procedures. Utilities are the system routines that either check, compile and run programs or configure the processes on the physical transputer network. To invoke a system utility, it is necessary to first load one of two sets of utilities, those for compilation or those for configuration. When the utilities are loaded, a different function key is associated with each utility. Pressing a function key may bring down a menu to request further parameters. For instance, the `compile` utility will ask for the name of the executable file.

The compilation utilities include `make` routines that turn sections of code into `COMMENT`, `SC`, `PROGRAM`, and `EXE` code. The programmer must label major sections of code so that the compiler knows how to treat them. The `COMMENT` feature can aid program development by allowing the programmer to fold large sections of unneeded code and comment them. `SC` turns procedures into *separately compiled* procedures. Since occam programs require inclusion of utilities, this feature allows procedures that are known to function correctly to be compiled once and then ignored during program development.

Organizing the entire program under these code labels requires some clarification at first. The top level of the traveling salesman program, running on TDS, is a `PROC` with two `CHAN` parameters for the `screen` and `keyboard`. The `PROC` is encased in a fold, labeled `EXE`, to indicate that it is to be an indivisible piece of executable code. This fold is compiled by pressing the key for the `compile` utility. (The executable code is also stored in folds in the structure.) The next step is to press a key to `get`

code and then a key to run code. All of this processing is done without leaving TDS. If a compile error is found, a locate error utility places the cursor at the offending statement. The locater can only find one error at a time, so the programmer must exit the folds and recompile the entire program to find the next error. Occam I was also able to locate run-time errors, but run-time errors in the beta relase of occam II result in an error flag from the transputer, which causes TDS to abort.

Using either the check or compile utility, TDS performs semantic checking on channel assignments to ensure that message sends have matching receives and that component processes in a PAR do not share variables. Many bugs can be programmed out of a system before it is actually tested.

The parallel environment of a transputer network is simulated in the development system, and a program can be thoroughly tested before it actually runs on a network. When the program functions properly on one processor, the programmer switches to the configuration utilities and modifies the program code to *place* processes on processors and channels on physical links. Procedures and their parameters are loaded onto transputers in the network. A transputer network was not available at the time this study was completed, so it was not possible to experiment with the configuration process. Several facts about the logical *versus* physical development of a program are of interest, however.

The logical development process is not totally independent of the physical system. Two problems are apparent. First, while the host process can load processes onto all transputers, it cannot communicate with them. The code below, used to

describe synchronization in Chapter 4, works on a single processor but can function on an actual transputer network only with the help of extra hardware or software.

```
PAR i = 0 FOR numnodes
  read.cmd[i] ! any
```

Current transputers have four bidirectional communication links. The *host* transputer dedicates one link to input/output with the terminal. That leaves at most three channels that can connect to the transputer network. To physically execute the ! in the statement above it would have to be connected to the entire network. This topology is possible if a crossbar switch is added to the network or if each transputer has a dedicated I/O port. A software solution is to place a process on each node that propagates messages throughout the network.

Because of the physical nature of a transputer, a grid configuration is the most easily conceptualized. A grid is ideal for certain applications such as Hoare's pipelined matrix multiply, but it is not as general as a hypercube. A certain amount of mapping is required even in the logical program. The host procedure of the matrix multiply program is shown in Figure 5.3. Channels are declared as arrays NS and EW. Channel indices for each process are calculated when specific channels are sent as parameters to PROC's `west`, `north`, `center`, `east` and `south`.

```
-- host
DEF  l = 200,                   -- rows in A
     m = 300,                   -- cols in A, rows in B
     n = 200:                   -- cols in B
CHAN NS[(m * n) + n],           -- north to south channels
     WE[(m * n) + m]:           -- east to west channels

PAR
  PAR j = [0 FOR m]
    west(j, WE[j * (n + 1)])            -- rows of A to col 1 of B

  PAR j = [0 FOR n]
    north(NS[j * (m + 1)])              -- north sends zeroes

  PAR i = [0 FOR m]
    PAR j = [0 FOR n]
      center(B[(i * n) + j],
             WE[(i * n) + i + j],
             WE[(i * n) + i + j + 1],
             NS[(j * m) + i + j],
             NS[(j * m) + i + j + 1]) -- perform partial mult.

  PAR j = [0 FOR m]
    east(WE[(n * (j + 1)) + j])         -- sink for values from A

  PAR
    screen.out                          -- output data to screen
    PAR j = [0 FOR n]
      south(j, NS[((m + 1) * j) + m]) -- product matrix columns
```

Figure 5.3 Channel Communication in Matrix Multiply in occam I

The greatest hurdle in developing a program for a transputer network may be the unfamiliarity of the occam language. While it supports pervasive concurrency, other details of this relatively young language limit its acceptablility. Occam I is a simple language to learn, with its three primitive processes and three constructors, but

because of its simplicity it was often necessary to write assembly-level routines to support operations that are built into familiar higher-level languages such as reading and writing characters, numbers and strings. Collections of bytes always had to be translated into the appropriate data type.

Occam II eliminates the need for explicit data-typing routines, but it does not perform automatic type conversions. Not only must variables be typed, but so must real numbers. The method for doing this differs depending upon whether the number is being assigned to a variable or being used in an expression. Furthermore, to convert a 32-bit integer to a REAL32, the programmer must specify whether it should be truncated or rounded.

```
x := 0.5 (REAL32) * (REAL32 TRUNC ((p * p) + (q * q)))
```

A second problem apparent from the example is that arithmetic operators in occam have no precedence. All of the subexpressions must be fully parenthesized.

A problem that remains in occam II is the static nature of the language. Channels must be statically defined. Replicators allow more than one instance of the same channel, but if the programmer does not know in advance how many channels are required, the only solution is to dimension the channels to some maximum number. Similarly, there is no way to dynamically allocate new instances of variables or processes, and thus no recursion.

Debug messages and all other I/O to the screen or to files must be managed by the programmer using a host process running in parallel with the node processes. This

process captures messages on various channels and outputs them. This interplay was described in some detail in Chapter 4. Log files were not as necessary in occam as on the iPSC. It seemed to be more helpful to examine program structure than to trace activity.

A common problem encountered at runtime is deadlock. Although the checker detects errors in channel protocol that might produce mismatched inputs and outputs, deadlock can still occur. It frequently occurs at program startup or termination. If input data is missing or if a process has nowhere to send output, the result is deadlock. The usual case is deadlock upon termination of processes under a PAR. Processes must determine when to terminate, either algorithmically or by receipt of a message. If the termination message comes from outside of the PAR, from a user, for instance, one of the parallel processes must be monitoring the source of the message. The best way to find the cause of deadlock was to reduce the size of the program to a skeleton, which is where the COMMENT utility was handy, and to test the flow of data through communication channels.

For timing occam programs, the language includes a special data type called `clock`. It measures the number of clock ticks in an execution. The time per tick depends upon the transputer model being used.

5.3. Sequent Balance

Program development on the Balance is based on an entirely different model. The features that distinguish it from distributed systems are shared memory and shared resources such as access to standard input and output. It is possible, although

not necessarily recommended, to develop a parallel program by gradually adding microtasks to a sequential program. Multiple high-level routines do not have to be developed at the same time as on the distributed systems. One implication of this difference is that programs can be developed in a top-down fashion using program stubs.

The first step in program implementation is to determine what data must be used by multiple processes and to assign it to shared memory by inserting the keyword `shared` in front of the declaration. For dynamic memory allocation to shared memory `shmalloc()` calls are substituted for `malloc()`.

Shared memory implies several changes to a program. First, every time a process writes to shared memory, it may have to block the execution of one or more processes that are attempting to access the same variable. Blocking processes forces sequentiality into the program. If the process executes code in a critical region using the microtask `m_lock()`, the greater the number of processes and the larger the critical region, the longer the other processes are blocked from execution. Second, shared memory implies that data does not have to be disseminated and collected. Results are stored in shared memory and can be printed by the master process.

Child processes are forked to run concurrently with the master using the microtask `m_fork()`. The number of processors may be determined at run time by user input to the program. Using the microtasking library exclusively, the number of processes is limited by the number of processors (minus one, which is reserved for the operating system). The ability to dynamically allocate processes speeds the develop-

ment and testing of a program.

The child processes contain code that will be executed in parallel, but it is not absolutely necessary that they contain only multi-stream code. Changing the flow of control between single and multiple streams is simple. Within a forked process, nodes can synchronize themselves using an `m_single()` microtask and spin at a barrier while the master process executes a single stream. Because of this feature, a Balance program may look much like a sequential program with `m_single()` and `m_multi()` commands interspersed to change the flow control.

In a homogeneous model, some method must be determined for partitioning work and data. Development of the matrix multiply program demonstrated how this task can be deceptively simple at first glance. Rather than a complicated algorithmic division of the matrices, the Balance implementation used a shared variable to represent the row index of the product matrix. Processes simply picked up a new row index and began work. Attempts to divide the data more evenly or at a finer grain resulted only in increased calculations, which slowed the entire process.

Since processes also share resources, any process may print messages to the screen or to a file. This capability eliminates the need to pass messages to a host process. However, the process printing the message should execute in a critical region so that it has control of the resource and the message prints cohesively. By executing in a critical region, a process may print a block of messages at once, something not always possible on a parallel system. A disadvantage is that the program becomes more sequential.

easier to grasp with its central memory and cohesive program structure. Also, shared memory eliminated the need for routines to disseminate and collect data, and the ability to print messages from any process eased debugging. Finally, the Balance was designed to support a general multi-user environment in addition to parallel applications. By dividing user jobs over all available processors, throughput is faster and reduces the time it takes to perform routine tasks such as editing and compiling.

The major obstacle to coding programs quickly in occam, aside from learning the language and development system, is the low level of coding required. Routines were written for monitoring and buffering I/O and for typing data. Utilities developed for occam I had to be rewritten for the beta release of occam II.

Message parameters provide the greatest hindrance on the iPSC. While it is easy to begin development of a program, tracing errant message communication requires a major effort. Since some of the problems in this study were due to topology mappings, one question is whether it would be easier to write a logical version of the program on the iPSC and later include mapping routines. This method would certainly be possible. The problem is that, unlike in the Transputer Development System where parallel processes are developed on one transputer and then mapped onto a network, on the iPSC, the program must be developed on the hypercube itself to be certain that it will work correctly. Some sort of mapping would be necessary for message passing, so it may as well be the proper mapping.

The algorithms changed slightly from system to system. The most extreme case was the matrix multiply program. The iPSC required complex data partitioning to

maximize computation on the nodes and minimize message traffic, whereas the occam version was written to take advantage of fast communications links by pipelining data. On the Balance it was possible to write a parallel version of a matrix multiply that differed only slightly from the sequential version. Some differences were encouraged because of machine design and performance. Others were based on simplicity, gaining some performance for little work. The Balance perhaps encourages the latter. Occam requires careful consideration of the algorithm before implementation, something that is time-consuming but that may result in a more elegant implementation. The iPSC seems to have neither benefit.

Chapter 6

Conclusions

This study has contrasted programming experiences on three parallel processing systems: two systems that modify an existing language to provide primitives for process management and sharing of data, and a system that introduces a new language to handle parallel computation. No matter how novel the environment, a key to success for each of these systems will be the how the programmer perceives its ease of use.

All three systems force the programmer to partition both program function and program data according to the constraints of the architecture. Once a parallel algorithm is chosen, the programmer's attention must be diverted to certain systems programming tasks and other nuances of the particular system.

6.1. Strengths and Weaknesses of Each System

The Sequent Balance requires little low-level intervention on the part of the programmer and with its familiar UNIX environment was the easiest system to learn. Often only minor changes to a program can allow it to take advantage of multiple processors, and mechanisms for altering the flow of control, sharing data and synchronizing processes are simple. However, large global data structures can create bottlenecks when multiple processes try to access them. The number of processes that can participate on shared-memory systems is limited for this reason. Even with a

small number of processes, the programmer may have to simulate distributed methods of data sharing to achieve adequate parallelism, and this simulation may be unduly complicated.

Occam encourages a programmer to think creatively about concurrent processes because a uniform approach to concurrency is built into the language, not bolted on through a procedure call interface. Separating logical design from physical design should free the programmer to consider the algorithm apart from some of the system details. Unfortunately, occam is still a low-level language that requires attention to details, such as data-type conversions and explicitly typed channels. In addition, the physical implementation can impose restrictions on the logical program, for instance, the limitations on the number of nodes with which a host process can directly communicate, and eventually the programmer must be familiar with the physical system in order to configure the program on the network.

The Transputer Development System introduces new concepts in program development. The folding editor is designed to help the programmer visualize the program structure, but it can be tedious to use. Similarly, many of the TDS utilities show symptoms of a language that has not fully matured. On the other hand, certain utilitites such as the compiler and checker are designed for parallel programming and can catch many common errors before programs are run.

What makes occam worthwhile is its support for pervasive parallelism. Concurrency in an algorithm may be exploited at many levels.

The iPSC has few of the advantages of either system. Its global Ethernet link between the Cube Manager and hypercube does remove the physical limitations of message transfer that complicates programming on the transputer network, but in practice this type of message transfer is too slow. Most applications route messages from the host to the nodes using one host-to-node message and a spanning tree in the hypercube. The complexity of the iPSC's parallel library interfaces requires a programmer to think at a systems level and to lose sight of the purpose of the program. Programs become bulky and tend to run slowly unless they have been optimized for a hypercube environment.

6.2. Future Directions

Parallel programs have their own set of problems and are typically difficult to write, especially when the programmer must delve into system-specific details. The programming systems studied here could benefit from tools for visualizing program behavior. Watching a dynamic, graphical representation of a parallel program would provide much more information than tracing a file of debug messages.

To allow the programmer to concentrate on the algorithm rather than system details, certain improvements must be made to these parallel processing systems. Primarily, they could benefit from more abstraction within their languages and system interfaces. Occam is certainly more suited to expressing parallel concepts than C; however, the transputer is designed to execute many levels of parallelism, while the other two architectures are limited by overhead to a larger-grain parallelism. Such a uniform approach to specifying concurrency may not be appropriate in those cases.

Still, a simple directive to the compiler to execute a section of code in parallel or to synchronize a number of distributed processes would ease the job of coding. The Balance approaches this simplicity with its microtasking library.

Another abstraction would be the ability to express certain data structures and tasks as *objects*. A candidate is message passing on the iPSC. Topologies of processes could be abstracted so that processes did not have to include mapping calculations. Messages themselves could be abstracted to eliminate the complicated parameterization of send and receive calls. Occam demonstrates this concept in its division of logical and physical development, but a programmer still has to map topologies and specify data types and sizes for channels. On an ideal parallel processing system, the programmer would only have to write the logical program.

The basis for this study was the diversity of parallel processing systems. Thus, a final recommendation is for more consistency among software interfaces to the various architectures. Certainly, abstraction of system and physical details is a starting point. Further studies are needed to determine what is an appropriate software model for parallel processing.

The notion that a programmer should only have to write sequential code and let the system find concurrency is limiting. Concurrency can be conceptualized. Learning to write concurrent code requires a clear model and tools that help rather than hinder. Managing parallel processes can become an integral part of a programmer's skills.

Appendix A

Timing and Results of the Traveling Salesman Program

Although the purpose of this study was to examine the process of parallel code development, the question of value-for-effort will certainly arise. Did the performance of the systems in this study merit the extra programming effort? To answer this question timing experiments were performed on the traveling salesman program. Timing of this program should produce the most meaningful results of the three programs. Since the code was ported from a working version produced at CalTech, it is reasonably certain that the algorithm is sound. In addition, Felten, *et al.* [FKO85] report timing statistics to which the three systems in this study can be compared.

The other two programs were not timed. Timing a dining philosophers program has little meaning. The program goes through the time-consuming process of logging messages to show the process of the simulation and does not have a completion point. The matrix multiply algorithms differed significantly depending upon the architecture of the system. In particular, the occam version was pipelined, while the other two versions used a larger grain of parallelism. Also, parallel versions of the LINPACK library matrix multiply have been developed by Intel [Mol86] and Sequent [SSS85] and presumably run much faster than the algorithms developed in this study. A final hindrence to timing both of these programs is that neither was run on a transputer, much less a network of transputers. They were programmed in occam on a mainframe computer that simulated a transputer environment.

One problem that does arise in comparison of traveling salesman program timings is that the occam version was run only on a single transputer and may only be compared to the single-node timings of the other two implementations.

A.1. iPSC Version

The program was developed on the R2.0 version of the iPSC System Software. The newest software release R3.0 occurred in January 1987. Timings were performed with both versions of the software. Since the new system was intended to speed message traffic, significant speedups were expected. The speed increased some but not as much as hoped for this application.

Three factors influenced times on the iPSC. First was the number of iterations performed by the program. With few iterations, times per iteration were high. At 50 iterations and higher, times became consistent. The number of iterations used for timing the CalTech program was not specified. Second, the subtitle of the timing table in the paper was "256 Cities Arranged in a Circle." The significance of the circle and its possible impact on timing was not clear. Third, the original CalTech code did not contain any code that indicated how timing was performed. Exact placement of timing calls in the iPSC and CalTech versions of the program may differ. On the iPSC, timings were done in the host process before the host directed the nodes to begin processing. One `sendmsg()` from the host to node 0, the global dissemination of the message and one `recvmsg()` were included in timings. The total time for iterations was divided in the host process by the number of iterations. When comparing the iPSC timings with those of the other two systems, note that even with only

one node, a process passes messages to itself to swap cities. Times were obtained by calls to the iPSC's C library `times()` routine.

On a full 5-dimension cube, the new version of the iPSC software only resulted in a speedup of 1.66 over the older version. Using the R3.0 software, however, the application of more nodes to the problem resulted in greater speedup. In neither case did times per iteration nor speedup match those of the CalTech hypercube. Comparing the R3.0 times to those of CalTech, the iPSC peforms nearly two (1.97) times as slowly on a 0-d cube and nearly 3 (2.77) times as slowly on a 5-d cube. The processors on the CalTech nodes are 8086/8087's and should run slower than the 80286/80287 processors on the iPSC. The algorithm is communication-intense, so message traffic may be the slowing factor.

Comparison of iPSC and CalTech Traveling Salesman Programs
256 Cities

	iPSC				CalTech	
	R2.0		R3.0			
d	time/iter. (msec)	speedup	time/iter. (msec)	speedup	time/iter. (msec)	speedup
0	1668	1.00	1260	1.00	639	1.00
1	730	2.28	638	1.97	332	1.92
2	271	6.15	326	3.87	167	3.82
3	216	7.72	173	7.28	85	7.54
4	139	12.00	95	13.26	43	14.82
5	101	16.51	61	20.66	22	28.80

Figure A.1 iPSC and CalTech Timings

Results of the program were not dramatic. That is, it was difficult to optimize a path that contained more than 16 cities. Four reasons were possible. First was the algorithm itself. The published paper presented two versions of the program, an *adjacency swap* version where nodes exchange cities with two nearest neighbors and a *hyperswap* version where nodes also exchange cities with distant nodes. The code provided by CalTech was for the adjacency swap version. Adding hyperswap code to the program did not yield better results.

The second possible reason for less-than-expected results may be that a successful user should be familiar with simulated annealing techniques. Felten, *et al.*, provided one example, which involved a path of 64 cities and 60,000 iterations of city-swaps per temperature drop. This brings us to a third possible reason for poor performance. At 10,000 iterations, the time for a simulation on the iPSC (running R2.0) became prohibitive. A smaller number of iterations was required. Without fully understanding the mechanics of simulated annealing, it was possible to approximate initial and final temperatures and temperature drops and gain some improvement in path lengths. With this caveat, a 16-city path was the largest sample for which a good approximation of a minimum path was achieved.

A fourth, and most likely, reason is the original configuration of the cities. The CalTech paper notes that its 64 cities were organized in 4 groups of 16 cities each. It is unclear how this differed from a completely random group of 64 points, which our study used.

A.2. Sequent Balance Version

The master thread of the traveling salesman program called the DYNIX routine `getrusage()` to time the multiple-process section of the `iterate()` routine. The Balance 8000 that was used for development had only 7 available processors, so to test higher dimensions the program was moved to a Balance 21000, which supports up to 30 processors. In this case, 27 was the maximum number available for testing. The speedup is linear up to 16 processes and appears to drop off at 27 processes.

Traveling Salesman Programs 256 Cities						
	iPSC		Balance		Transputer	
processes	time/iter. (msec)	speedup	time/iter. (msec)	speedup	time/iter. (msec)	speedup
1	1260	1	363	1	115	1
2	638	2	181	2	-	-
4	326	4	93	4	-	-
8	173	7	45	8	-	-
16	95	13	22	16	-	-
27	-	-	14	26	-	-
32	61	21	-	-	-	-

Figure A.2 iPSC, Balance, and Transputer Timings

The tests used to acquire the speedups could be run quickly, but they did not result in much of an improvement in the salesman's path. On the Balance it was possible to run the 64-city example test from the CalTech paper that required 60,000 iterations per temperature drop (an overnight run). The result of the test was a noted improvement in path length, but it was still far from optimal. Again, this result might be due to the original configuration of the cities, which was random in all of the

tests.

A.3. Occam Version

As noted earlier, the final implementation of the program in occam was on a Transputer Development System with a single T414 transputer. The time per iteration for a 256-city test, shown in Figure A.2, is faster than that of the Balance and iPSC, but with caveats. Since the program ran on one transputer, the concept of a host transputer and single node transputer were simulated by two processes on one chip. To accurately compare transputer times against the other two systems, it is be important to know the comparative times of an on-chip and a off-chip message transfer. Most of the time involved in a message pass is in set-up. Since set-up can be done in parallel when a message passes between two transputers, the on-chip and off-chip transfer times are almost identical. A factor that may have reduced the timings was that a square root function, used frequently in each iteration of the algorithm, never worked, so it is estimated by multiplication, which should be faster.

A.4. Sample Output

Each line of the sample log file in Figure A.3 lists the current temperature and the path length after the nodes have iterated at that temperature. In simulated annealing at high temperatures, there is a higher amount of randomness that allows points to travel from one node to another with higher frequency, even if the resulting path is not necessarily shorter. At lower temperatures the randomness decreases and the path may shorten substantially.

```
Number of processes = 4
Number of points on path = 64
Number of iterations per drop in temperature = 60,000

current temperature = 1.000   path length = 31.911
current temperature = 0.975   path length = 30.121
current temperature = 0.950   path length = 30.568
~
~
current temperature = 0.350   path length = 28.052
current temperature = 0.325   path length = 23.345
current temperature = 0.300   path length = 25.163


Number of iterations per drop = 40,000

current temperature = 0.300   path length = 23.474
current temperature = 0.290   path length = 23.064
current temperature = 0.280   path length = 22.276
~
~
current temperature = 0.070   path length = 12.426
current temperature = 0.060   path length = 12.902
current temperature = 0.050   path length = 11.076
```

Figure A.3 A Typical Traveling Salesman Log File

Figures A.4 and A.5 show test results for a path of 16 cities, the largest sample for which a relatively straight path was achieved.

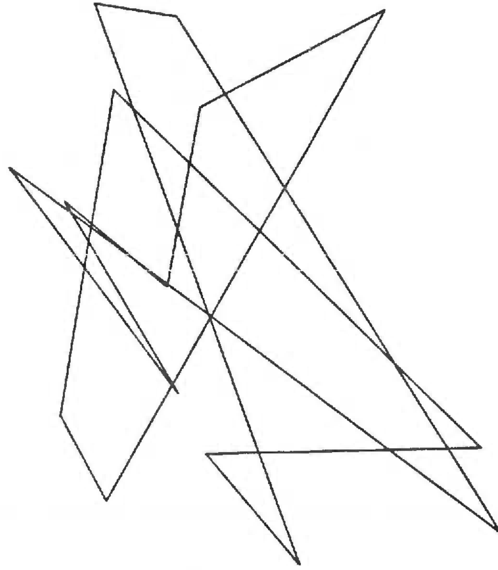


Figure A.4 16 Cities in Random Configuration

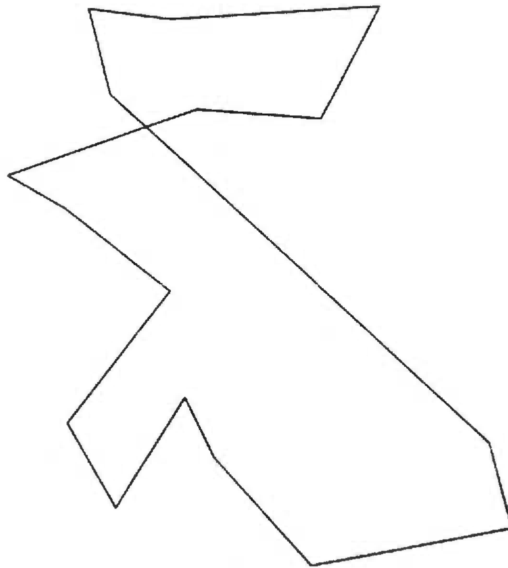


Figure A.5 16 Cities - Path Straightened by Simulated Annealing

Appendix B

Dining Philosophers Program Listings

Figure B.1 Dining Philosophers in C on iPSC

```
/*
 * dining.h
 * Macros for dining philosophers programs
 */
#define ALL_NODES -1              /* for dynamic loading */
#define ALL_PROCESSES -1          /* for dynamic loading */
#define BUFLEN 100                /* message buffer */
#define SIM_LENGTH 100            /* number of loops */
#define MSGTYPE 0                 /* message type */
#define PPID 10                   /* phils' process id */
#define FPID 11                   /* forks' process id */

typedef enum {Allocate, Release, Stop} actionkind;
```

```
/*
 * maitre_d.c
 * Host process for dining philosophers problem
 */
#include <stdio.h>
#include "/usr/ipsc/lib/chost.def"
#include "dining.h"

main()
{
int  cnt;                       /* number of bytes in message */
char debug_buf[BUFLEN];         /* debug message buffer */
int  hostcid;                   /* global channel id */
int  hostpid;                   /* host process id */
int  i;                         /* loop index */
int  num;                       /* number of philosophers */
char rec_buf[BUFLEN];           /* receive message buffer */
char send_buf[BUFLEN];          /* send message buffer */
int  snode;                     /* node of sender */
int  spid;                      /* pid of sender */
int  type;                      /* message type */

/* Dynamically load the node processes. */

load("phil", ALL_NODES, PPID);
load("fork", ALL_NODES, FPID);

/* Get host id and set up communications channel to the nodes. */

hostpid = getpid();
hostcid = copen(hostpid);

/* Assign a left and right fork to the requesting philosopher,
   but send the odd-numbered one followed by the even-numbered
   one. The maitre d' fills places at the table sequentially as
   messages arrive from the philosophers.  Assume the value of
   a fork will not be greater than 128, so it can be stored in
   a char. */

num = 1 << cubedim();
for (i = 0; i < num; i++) {
    send_buf[0] = (i % 2 != 0) ? (char) i : (char)((i + 1) % num);
    send_buf[1] = (i % 2 == 0) ? (char) i : (char)((i + 1) % num);
    sendmsg(hostcid, MSGTYPE, send_buf, 2, snode, spid);
    }

lwaitall(ALL_NODES, ALL_PROCESSES);
lkill(ALL_NODES, ALL_PROCESSES);
cclose(hostcid);
}
```

```
/*
 * phil.c
 * Philosopher node process for dining philosophers problem
 */
#include "/usr/ipsc/lib/cnode.def"
#include "dining.h"

main()
{
int  cid;                        /* channel id */
int  cnt;                        /* number of bytes in message */
int  even;                       /* index of even fork */
int  hostname;                   /* host node id */
int  hostpid;                    /* host process id */
int  odd;                        /* index of odd fork */
char rec_buf[BUFLEN];            /* receive message buffer */
int  snode;                      /* node of process sending message */
int  spid;                       /* pid of sender */
char send_buf[BUFLEN];           /* send message buffer */
int  count = 0;                  /* loop counter */

cid = copen(PPID);               /* open node channels */

/* Receive place assignment from maitre d' */

recvw(cid, MSGTYPE, rec_buf, BUFLEN, &cnt, &hostname, &hostpid);
odd  = (int) rec_buf[0];
even = (int) rec_buf[1];

while (count <= SIM_LENGTH) {

    syslog(PPID, "Phil is thinking");

    send_buf[0] = (char) Allocate;           /* Pick up forks */
    sendw(cid, MSGTYPE, send_buf, 1, odd, FPID);
    recvw(cid, MSGTYPE, rec_buf, BUFLEN, &cnt, &snode, &spid);
    sendw(cid, MSGTYPE, send_buf, 1, even, FPID);
    recvw(cid, MSGTYPE, rec_buf, BUFLEN, &cnt, &snode, &spid);

    syslog(PPID, "Phil is eating");

    send_buf[0] = (char) Release;            /* Put down forks */
    sendw(cid, MSGTYPE, send_buf, 1, odd, FPID);
    sendw(cid, MSGTYPE, send_buf, 1, even, FPID);
    ++count;
    }

cclose(cid);
}
```

```
/*
 * fork.c
 * Fork node process for dining philosophers problem
 */
#include "/usr/ipsc/lib/cnode.def"
#include "dining.h"

main()
{
typedef struct pnode {            /* represents a waiting phil */
    int  snode;                   /* phil's node id */
    int  spid;                    /* phil's pid */
    } WNODE;

actionkind  action;               /* allocate or release */
int  cid;                         /* channel id */
int  cnt;                         /* number of bytes in message */
char debug_buf[BUFLEN];           /* debug message buffer */
enum {Busy, Not_Busy} fork;       /* fork in use or not */
char rec_buf[BUFLEN];             /* receive message buffer */
char send_buf[BUFLEN];            /* send message buffer */
int  snode;                       /* node id of sender */
int  spid;                        /* pid of sender */
WNODE waiting;                    /* waiting philosopher */
enum {Empty, Occupied} wait_ptr;    /* occupied if phil waiting */

cid = copen(FPID);
wait_ptr = Empty;
fork = Not_Busy;

/* Loop until the maitre_d kills me */

while (1) {

    /* Receive message from a philosopher */

    recvw(cid, MSGTYPE, rec_buf, BUFLEN, &cnt, &snode, &spid);
    action = (actionkind) rec_buf[0];

    switch(action) {
    case Allocate:

        /* If fork is free, make it busy and send message back
           to philosopher saying request filled. */

        if (fork == Not_Busy) {
          fork = Busy;
          sprintf(send_buf, "You have picked up a fork\n");
          sendw(cid, MSGTYPE, send_buf, strlen(send_buf),
              snode, spid);
```

```
        }

      else {

        /* Make the philosopher wait for the fork.
             Store his node and process id and continue. */

        waiting.snode = snode;
        waiting.spid  = spid;
        wait_ptr = Occupied;
        }
    break;

  case Release:
    fork = Not_Busy;

    /* Check wait ptr to see if the other philosopher is
         waiting for this fork.  If so, nullify the wait
         pointer, make the fork busy and send a message back
         to this philosopher. */

    if (wait_ptr == Occupied) {
        fork = Busy;
        wait_ptr = Empty;
        sprintf(send_buf, "You have picked up a fork\n");
        sendw(cid, MSGTYPE, send_buf, strlen(send_buf),
          waiting.snode, waiting.spid);
        }
    break;

  default:                          /* error */
    sprintf(debug_buf, "Unknown action in fork: %d\n",
        (int) action);
    syslog(FPID, debug_buf);
    break;
    }
  }
}
```

Figure B.2 Dining Philosophers in occam I

```
-- output string                    -- load one on every node
PROC out.string (VALUE s[], CHAN out) =
  VAR length:
  SEQ
    length := s [BYTE 0]
    SEQ i = [1 FOR length]
      out! s [BYTE i]:

-- philosophers
PROC phil(VAR me, sim.length, CHAN pickup.odd, pickup.even,
 putdown.odd, putdown.even, debug) =
  VAR timer:                        -- to time simulation
  SEQ.
    timer := 1
    WHILE timer <= sim.length
      SEQ
        --think
        out.string("philosopher ", debug)
        debug ! me + 'O'
        out.string(" thinking.", debug)
        pickup.odd ! ANY
        pickup.even ! ANY
        --eat
        out.string("philosopher ", debug)
        debug ! me + 'O'
        out.string(" eating.", debug)
        putdown.odd ! ANY
        putdown.even ! ANY
        timer := timer + 1
    SKIP:

-- forks
PROC fork(VAR me, CHAN pickup, putdown) =
  WHILE TRUE
    SEQ
      pickup ? ANY
      putdown ? ANY
      SKIP:
```

```
-- initialization and control
PAR
-- screen output buffering
  DEF  EndBuffer = -3:              -- end of message marker
  CHAN screen AT 1:
  VAR  msg.char:
  VAR  msg.not.complete:
  VAR  i:
  WHILE TRUE
    ALT i = [0 FOR num]
      debug[i] ? msg.char
        SEQ
          msg.not.complete := TRUE
          WHILE msg.not.complete
            SEQ
              screen ! msg.char
              debug[i] ? msg.char
              IF
                msg.char = EndBuffer
                  msg.not.complete := FALSE
                TRUE
                  SKIP

-- maitre_d
  DEF  num = 4:                     -- number of philosophers and forks
  DEF  sim.length = 100:            -- number of cycles for simulation
  CHAN debug[num]:                  -- for message buffering
  CHAN pickup[num]:                 -- for picking up fork
  CHAN putdown[num]:                -- for putting down fork
  VAR i:
  VAR odd, even:                    -- fork ids
  PAR i = [0 FOR num]
    SEQ
      IF
        (i  2) = 1                 -- left fork is odd
          PAR
            odd = i
            even = i + 1
        (i  2) = 0                 -- left fork is even
          PAR
            odd = i + 1
            even = i
      PAR
        phil(i, sim.length, pickup[odd], pickup[even],
         putdown[odd], putdown[even], debug[i])
        fork(i, pickup[i], putdown[i], debug[i])
```

Figure B.3 Dining Philosophers in C on Sequent Balance

```
#include <stdio.h>
#include <parallel/parallel.h>
#include <parallel/microtask.h>

#define n 4                  /* number of philosophers and forks */
#define sim_length 100       /* loops for simulation */

shared slock_t forks[n]; /* forks are just locks */
shared int place;            /* for philosopher place assignments */

void
phil()
{
int count;                   /* to count loops in simulation */
int left, right;             /* fork numbers */
int odd, even;               /* indices of odd and even forks */

m_multi();

/* Get my place assignment. */

m_lock();
left = place++;
right = place % n;
m_unlock();

odd  = (left % 2) != 0 ? left : right;  /* which is my odd fork? */
even = (left % 2) != 0 ? right: left;

for (count = 0; i < sim_length; count++) {
    s_lock(&forks[odd]);                /* pick up forks */
    s_lock(&forks[even]);

    m_lock();
    printf("Philosopher %d is eating\n", left);
    m_unlock();

    s_unlock(&forks[odd]);              /* put down forks */
    s_unlock(&forks[even]);

    m_lock();
    printf("Philosopher %d is thinking\n", left);
    m_unlock();
    }
}
```

```
main()
{
int i;

place = 0;                          /* initialize counter */

for (i = 0; i < n; i++)             /* initialize locks */
    s_init_lock(&forks[i]);

if (m_set_procs(n) != 0) {          /* initialize n processes */
    perror("m_set_procs didn't work0);
    exit(-1);
    }
m_fork(phil);                       /* fork them to help execute phil */
m_kill_procs();
}
```

References

[AAA86] *FX/FORTRAN Programmer's Handbook*, Alliant Computer Systems Corporation, March 1986.

[AnS83] Andrews, G. R. and Schneider, F. B., "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, 1 (March 1983), pp. 3-43.

[BrT87] Brandis, C. and Thakkar, S. S., "A Parallel Program Event Monitor," *Twentieth Hawaii International Conference on System Sciences*, January 1987.

[Bri72] Brinch-Hansen, P., "Structured Multiprogramming," *Comm. ACM*, vol. 15, 7 (July 1972), pp. 574-578.

[Car82] Cargill, T. A., "A Robust Distributed Solution to the Dining Philosophers Problem," *Software - Practice and Experience*, vol. 12, 10 (October 1982), pp. 965-969.

[CHL] Chen, S. S., Hsiung, C. C., Larson, J. L. and Somdahl, E. R., "Cray X-MP: A Multiprocessor Supercomputer," in *Vector and Parallel Processors: Architecture, Applications, and Performance Evaluation*, M. Ginsberg (ed.), North Holland, . (to be published in 1987).

[CCC85] "Multitasking User Guide," Cray Computer System Technical Note SN-0222, Cray Research, Inc., January 1985.

[Dij72] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes," in *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott (ed.), Academic Press, New York, 1972.

[FKO85] Felten, E., Karlin, S. and Otto, S. W., "The Traveling Salesman Problem on a Hypercubic, MIMD Computer," *IEEE Proceedings of the 1985 Conference on Parallel Processing*, 1985.

[FiF84] Filman, R. E. and Friedman, D. P., *Coordiated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, 1984.

[Fly66] Flynn, M. J., "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54(1966), pp. 1901-1909.

[GuH86] Gustafson, J. L. and Hawkinson, S., "A Language-Independent Set of Benchmarks for Parallel Processors," Pre-print, Floating Point Systems, Inc., April 1986.

[GHS86] Gustafson, J. L., Hawkinson, S. and Scott, K., *The Architecture of a Homogeneous Vector Supercomputer*, Floating Point Systems, Inc., March 27, 1986.

[Hil85] Hillis, W. D., *The Connection Machine* , MIT Press, Cambridge, MA, 1985.

[Hoa72] Hoare, C. A. R., "Towards a Theory of Parallel Programming," in *Operating Systems Techniques*, Academic Press, 1972, pp. 61-71.

[Hoa78] Hoare, C. A. R., "Communicating Sequential Processes," *Comm. ACM*, vol. 21, 8 (August 1978), pp. 666-677.

[HwB84] Hwang, K. and Briggs, F., *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[Inm85a] "Occam Programming System ," System Manual, INMOS Ltd., July 1985.

[Inm85b] "Transputer Development System," System Manual, INMOS Ltd., November 1985.

[Inm86] *Transputer Development System 2.0 User Manual*, INMOS Ltd., June 2, 1986.

[Int85] "iPSC User's Guide," System Manual, Intel Corporation, October 1985.

[JJD78] Jones, A. K., Jr., R. J. C., Durham, I., Feiler, P. H., Scelza, D. A., Schwan, K. and Vegdahl, S. R., "Programming Issues Raised by a Multiprocessor," *Proceedings of the IEEE*, vol. 66, 2 (February 1978), pp. 229 - 237.

[KeR78] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, 1978.

[KiS79] Kieburtz, R. B. and Silberschatz, A., "Comments on Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, vol. 1, 2 (October 1979), pp. 218-225.

[KDL86] Kuck, D. J., Davidson, E. S., Lawrie, D. H. and Sameh, A. H., "Parallel Supercomputing Today and the Cedar Approach," *Science*, vol. 231, 4741 (February 28, 1986), pp. 967-978.

[LHG86] Liskov, B., Herlihy, M. and Gilbert, L., "Limitations of Synchronous Communication with Status Process Structure in Lanaguages for Distributed Computing," *Principles of Programming Languages*, 1986, pp. 150-159.

[MaS84] May, D. and Shepherd, R., "Occam and the Transputer," in *Proceedings of the IFIP WG10.3 Workshop on Hardware-Supported Implementation of Concurrent Languages in Distributed Systems*, North Holland Publishing Company, October 1984.

[McA] McGraw, J. and Axelrod, T. S., "Exploiting Multiprocessors: Issues and Options," in *Programming Parallel Processors*, R. G. B. II (ed.), Addison-Wesley, . (to be published in 1987).

[MeB76] Metcalfe, R. and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM*, vol. 19, 7 (July 1976), pp. 395-404.

[Mol86] Moler, C., "Matrix Computation on Distributed Memory Multiprocessors," in *Hypercube Multiprocessors*, M. Heath (ed.), SIAM, 1986, pp. 181-195.

[Mol--] Moler, C., (private communication), 1986.

[PKL80] Padua, D. A., Kuck, D. J. and Lawrie, D. H., "High-Speed Multiprocessors and Compilation Technology," *Transactions on Computers*, September 1980, pp. 763-776.

[PaL] Pase, D. M. and Larrabee, A. R., "Intel iPSC Concurrent Computer," in *Programming Parallel Processors,* R. G. B. II (ed.), Addison-Wesley, . (to be published in 1987).

[Pou86] Pountain, D., *A Tutorial Introduction to OCCAM Programming,* INMOS, Ltd., July 1986. (preliminary version).

[Rat85] Rattner, J., "Concurrent Processing: A New Direction in Scientific Computing," in *Proceedings of the National Computer Conference,* 1985, pp. 158-166.

[Sei85] Seitz, C. L., "The Cosmic Cube," *Communications of the ACM,* vol. 28, 1 (January 1985), pp. 22-33.

[SSS85] "Balance 8000 Technical Summary," System Manual, Sequent Computer Systems, Inc., November 1985.

[SSS86a] "Parallel Product Specification," System Manual, Sequent Computer Systems, Inc., September 1986.

[SSS86b] *DYNIX Pdbx Debugger User's Manual,* Sequent Computer Systems, Inc., May 2, 1986.

[SAN81] Shaw, M., Almes, G. T., Newcomer, J. M., Reid, B. K. and Wulf, W. A., "A Comparison of Programming Languages for Software Engineering," *Software - Practice and Experience,* vol. 11(1981), pp. 1-52.

[Sto82] Stotts, P. D., "A Comparative Survey of Concurrent Programming Languages," *ACM SIGPLAN Notices* , vol. 17, 10 (October 1982), pp. 50-61.

[Str80] Stroustrup, B., *A Set of C Classes for Co-routine Style Programming,* Bell Laboratories Computing Science Technical Report, November 18, 1980.

[Str86] Stroustrup, B., *The C++ Programming Language,* Addison-Wesley, 1986.

[TGF] Thakkar, S., Gifford, P. and Fielland, G., "Balance: A Shared Memory Multiprocessor System," *Second International Conference on Supercomputing,* Santa Clara, CA, . (to be published in 1987).

[UUU80] *Ada Programming Language Reference Manual,* U.S. Department of Defense, July 1980.

[WaW84]
Wand, I. C. and Wellings, A. J., Distributed Computing, 1984, pp. 201-215.

[WLS79] Welsh, J., Lister, A. and Salzman, E. J., "A Comparison of Two Notations for Process Communication," *Proceedings of the Symposium on Language Design and Programming Methodology,* Sydney, September 1979, pp. 225-254.

[Whi85] Whitby-Strevens, C., "The Transputer," *Twelfth Annual Symposium on Computer Architecture,* Boston, June 1985, pp. 292-300.

[Wir83] Wirth, N., *Programming in Modula-2,* Springer-Verlag, 1983.

[86] "How Meiko is Getting an Instant Supercomputer," *Electronics,* vol. 59, 36 (November 27, 1986), .

Biographical Note

The author is a Portland native who graduated from Lincoln High School and the University of Oregon Honors College. Before beginning studies in computer science, she coordinated medical conferences and published related books for Medical Computer Services Association, Seattle, and set up the Department of Continuing Medical Education at the Dartmouth-Hitchcock Medical Center, Hanover, New Hampshire. She worked as a programmer/analyst at Tektronix for five years before accepting a Tektronix fellowship for the Reentry Program in Computer Science at the University of California, Berkeley in 1985. She received an OGC fellowship for study towards a Master's degree, also in 1985. She is leaving the Graduate Center to take a position as Research Associate on the Parsifal Project at the University of Manchester in England.