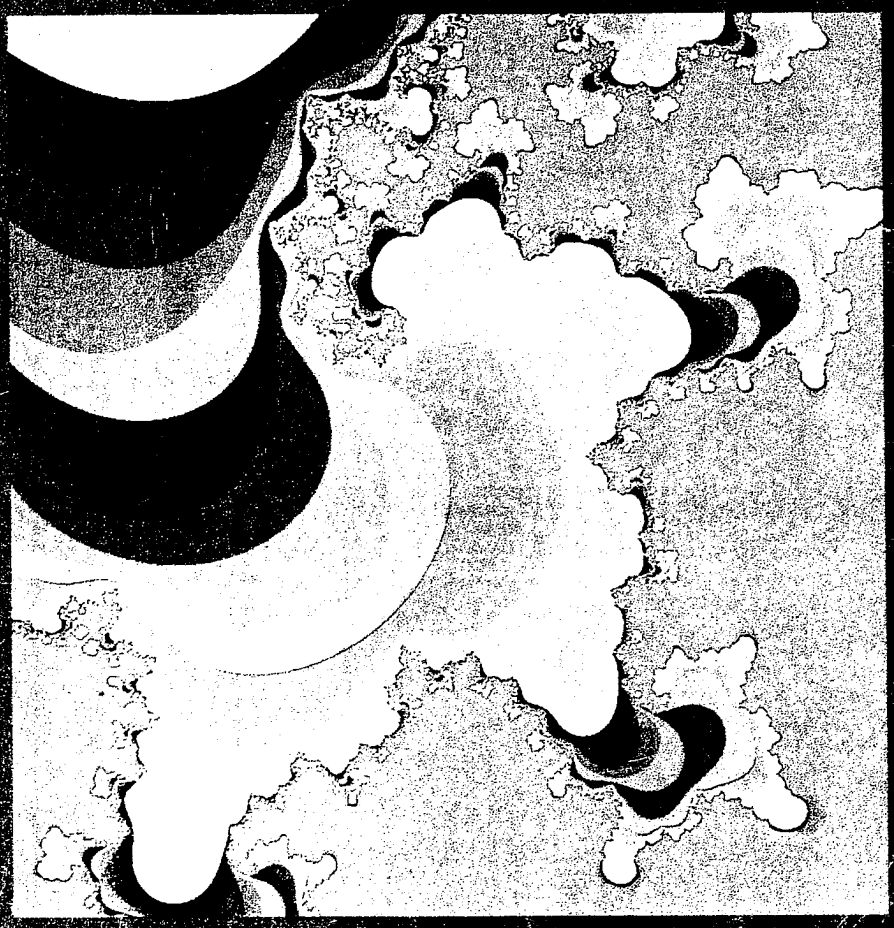
$15.00

GUIDE TO PARALLEL PROGRAMMING



ON SEQUENT COMPUTER SYSTEMS

SECOND EDITION

This book was set in Century Schoolbook and Courier 10 by the author, using an Imagen 7320 laser printer driven by a Sequent S81 running under the DYNIX ® operating system.

Printed in the United States of America.

Acknow

The author wishes to thank a
time, expertise, and encouragen
In addition, many thanks to Se
for its responsiveness and th
formatting, and typesetting duti

Cover design: Jeanne Galick

Cover photo: Dahlstrom Phot

About the cover: The cove
segment of the Mandelbrot se
can be graphed as points on
generated by a Sequent B21 co

Chapter 3

Parallel Programming Tools

3.1. Introduction

This chapter describes some of the programming tools available on Sequent systems. Some of these tools are available from Sequent and some have been developed by Sequent users. Together, they show the wide range of parallel programming approaches that are supported by Sequent systems.

The applications that can be adapted for parallel programming vary greatly in their requirements for data sharing, interprocess communication, and synchronization. To gain optimal speed-up from a parallel solution, the programmer must develop an algorithm that meets the requirements of the application while still exploiting all of its inherent parallelism. To aid in this effort, the programmer needs tools that adapt easily to the needs of a given application.

For example, a matrix multiplication on a large data set is best expressed in terms of data partitioning: the solution requires repeating the same operation on many different data items. This problem is very synchronous. The program will have a well-defined beginning and end, and the programmer can easily predict at what points the processes must synchronize or communicate shared data. Ideal tools for this application would support creation and termination of multiple identical processes and division of shared data among processes.

In contrast, a large data base application might be much better expressed in terms of function partitioning. At any time, different users may be using different utilities to access the data base. These processes may need to communicate to share data, or one process may need to ensure that another process doesn't corrupt its data. This application is asynchronous: the programmer cannot predict when users will create processes that need to communicate or access shared data. This application requires tools that allow processes to communicate on an as-needed basis.

The Sequent systems support programming tools for a wide range of applications:

- The FORTRAN parallel programming directives support parallel execution of FORTRAN DO loops. With these directives, users can execute many DO loops in parallel simply by adding a single line to the source code.
- The microtasking routines in the Parallel Programming Library support data and function partitioning applications. They allow users to quickly and easily create sets of processes, schedule tasks among processes, and synchronize processes between tasks.
- The Force is a flexible tool which adapts to both data partitioning and function partitioning applications. In addition to the process creation, scheduling, and synchronization capabilities of data partitioning tools, it supports synchronization based on availability of shared data.
- The DYNIX operating system includes a number of facilities that support communication of data and status information between loosely related processes.
- The parallel Ada tasking facility supports a similarly asynchronous programming approach.

The following sections briefly describe these tools.

3.2. FORTRAN Parallel Programming Directives

The Sequent FORTRAN compiler can restructure DO loops for parallel execution. The user prepares the program for the preprocessor by inserting a set of directives which identify the loops to be executed in parallel, the shared and private data within each loop, and any critical sections of the loops (loop sections containing dependences). The directives also allow the user to control the scheduling of loop iterations among processes and the division of data between processes. The directive are described in the *Sequent FORTRAN Compiler User's Guide*.

Once the user has identified the parallel loops and properly marked the data and critical sections, the preprocessor handles all the low-level tasks of data partitioning. The preprocessor produces a program that transparently sets up shared data structures, creates a set of identical processes, schedules tasks among processes, and handles mutual exclusion and process synchronization.

Chapter 4 explains how to use the directives and how to analyze DO l data and critical code sections.

3.3. Parallel Programming

The Sequent Parallel Programming l allow the programmer to execute C, in parallel. The library includes rou tions:

- Allocation of memory for sha
- Creation of processes to exec
- Identification of individual pr
- Suspension of processes duri
- Mutual exclusion on shared
- Synchronization of processes

Programs that use the Parallel Pro automatically balance loads betwee adjust the division of computing task of processors configured in the syst programmer to handle the communi an algorithm at a high level while parallel algorithm.

Chapter 5 explains how to use the illustrates some data analysis and sc

3.4. The Force

The Force is a set of FORTRAN m the University of Colorado at Bould data partitioning in a manner simila programming directives, but they al solutions.

For simple data partitioning, the Fo tion and termination, declaration

gramming Tools

·ogramming tools for a wide range of

›rogramming directives support parallel
DO loops. With these directives, users
ps in parallel simply by adding a single

s in the Parallel Programming Library
n partitioning applications. They allow
sily create sets of processes, schedule
and synchronize processes between

›l which adapts to both data partition-
›ning applications. In addition to the
ng, and synchronization capabilities of
it supports synchronization based on
a.

ystem includes a number of facilities
ıtion of data and status information
·ocesses.

facility supports a similarly asynchro-
ach.

-ibe these tools.

'rogramming Directives

· can restructure DO loops for parallel
program for the preprocessor by insert-
ify the loops to be executed in parallel,
n each loop, and any critical sections of
ng dependences). The directives also
scheduling of loop iterations among
ı between processes. The directive are
N Compiler User's Guide.

parallel loops and properly marked the
›reprocessor handles all the low-level
preprocessor produces a program that
a structures, creates a set of identical
· processes, and handles mutual exclu-

Chapter 4 explains how to use the FORTRAN parallel programming directives and how to analyze DO loops to identify shared and private data and critical code sections.

3.3. Parallel Programming Library

The Sequent Parallel Programming Library is a set of C routines which allow the programmer to execute C, FORTRAN, or Pascal subprograms in parallel. The library includes routines to handle the following functions:

- Allocation of memory for shared data
- Creation of processes to execute subprograms in parallel
- Identification of individual processes
- Suspension of processes during serial program sections
- Mutual exclusion on shared data
- Synchronization of processes during critical sections

Programs that use the Parallel Programming Library can be made to automatically balance loads between processors and to automatically adjust the division of computing tasks at run time based on the number of processors configured in the system. The library routines allow the programmer to handle the communication and synchronization needs of an algorithm at a high level while concentrating on the design of the parallel algorithm.

Chapter 5 explains how to use the Parallel Programming Library and illustrates some data analysis and scheduling techniques.

3.4. The Force

The Force is a set of FORTRAN macros developed by Harry Jordan of the University of Colorado at Boulder. These macros support standard data partitioning in a manner similar to the Sequent FORTRAN parallel programming directives, but they also offer support for less synchronous solutions.

For simple data partitioning, the Force provides automatic process creation and termination, declaration of shared and private data, and

synchronization of critical code sections. It will restructure loops for parallel execution using either prescheduling or self-scheduling.

The Force also includes a special data type, Async, and two special operations, Produce and Consume, that allow synchronization based on data availability. An Async variable is a shared variable that has a "full/empty" state flag associated with it. An Async variable is marked full by a Produce operation. If the variable is already full, the Produce operation waits until the variable is empty before writing a new value. When a process performs a Consume operation on an Async variable, the Force verifies that the Async variable is in the full state. If not, the Consume operation waits until the variable is full, executes, and then sets the variable state to empty.

For more information about the Force and where to obtain the Force macros for Sequent computers, contact Sequent Technical Marketing.

3.5. UNIX Function Partitioning Tools

The DYNIX operating system provides support for asynchronous parallel programming through standard UNIX 4.2bsd system calls, with special DYNIX system calls and libraries, and with system calls in the System V Applications Environment (SVAE).

UNIX system calls such as `sigpause()`, `sigvec()`, and `sigblock()` allow processes to send and receive signals among themselves. The SVAE system calls `semop()`, `semget()`, and `semctl()` allow programs to create and use counting and blocking semaphores. The UNIX Interprocess Communication (IPC) subsystem allows processes to perform direct data transfers among themselves, even across a network of systems. The SVAE message-passing system calls allow processes to send and receive data via message queues. Together, these facilities support a wide range of function partitioning applications, ranging from a single program with a set of unique parallel processes to a set of programs working on a shared data base.

All of these facilities are described in more detail in Chapter 5.

3.6. Parallel Ada

The standard Ada language suppo
parallel programming. The Ada lan
called *tasks*. Tasks resemble subrouti
can be executed in parallel. The S
(PRTS) allows Ada tasks to execute i

Ada tasks communicate and sync
ENTRY, ACCEPT, and "call" state
ENTRY declarations, each of which
The task's ENTRY declarations and
ments in the task body define all the
can perform when it is called by anc
bles a function call that specifies
ENTRY in the called task, and the a
task.

At any time during program executio
suspends its execution until the call
ACCEPT statement. Once the ACC
tasks are said to be "in rendezvous"
suspended until the accepting task h
ENTRY and passed the results back.
lel execution until either needs to ren

For more information on the Sequent

3.7. Other Tools

Parallel researchers have implemen
gramming tools on Sequent machin
developed PPL©, a C-based parallel
process management features. (Appe
on PPL.) Several Sequent users ha
for use on their Sequent systems. D
more National Laboratories has
processes on a Sequent system.

Most applications can be solved effi
The programming tools described in
range of applications, and paral

ections. It will restructure loops for cheduling or self-scheduling.

l data type, Async, and two special , that allow synchronization based on ıble is a shared variable that has a with it. An Async variable is marked ə variable is already full, the Produce is empty before writing a new value. ne operation on an Async variable, the ble is in the full state. If not, the Con- ıriable is full, executes, and then sets

Force and where to obtain the Force tact Sequent Technical Marketing.

:ioning Tools

ides support for asynchronous parallel NIX 4.2bsd system calls, with special and with system calls in the System E).

`ıuse()`, `sigvec()`, and `sigblock()` eive signals among themselves. The `emget()`, and `semctl()` allow pro- and blocking semaphores. The UNIX) subsystem allows processes to per- themselves, even across a network of ssing system calls allow processes to ıge queues. Together, these facilities partitioning applications, ranging from ique parallel processes to a set of pro- ase.

in more detail in Chapter 5.

3.6. Parallel Ada

The standard Ada language supports an asynchronous approach to parallel programming. The Ada language includes program structures called *tasks*. Tasks resemble subroutines except that, by definition, they can be executed in parallel. The Sequent Parallel Run-Time System (PRTS) allows Ada tasks to execute in parallel.

Ada tasks communicate and synchronize with each other through ENTRY, ACCEPT, and "call" statements. A task can include several ENTRY declarations, each of which represents a subroutine declaration. The task's ENTRY declarations and the corresponding ACCEPT statements in the task body define all the operations that a task of that type can perform when it is called by another task. A call statement resembles a function call that specifies the task being called, the desired ENTRY in the called task, and the arguments to be passed to the called task.

At any time during program execution, one task can call another. It then suspends its execution until the called task executes the corresponding ACCEPT statement. Once the ACCEPT statement is present, the two tasks are said to be "in rendezvous". At this point, the calling task is suspended until the accepting task has completed the operations for that ENTRY and passed the results back. Both tasks can then resume parallel execution until either needs to rendezvous with another task.

For more information on the Sequent PRTS, contact Sequent Marketing.

3.7. Other Tools

Parallel researchers have implemented a variety of other parallel programming tools on Sequent machines. Herb Schwetman of MCC has developed PPL©, a C-based parallel programming language with built-in process management features. (Appendix D gives a reference for a paper on PPL.) Several Sequent users have developed hypercube simulators for use on their Sequent systems. Dr. Eugene Brooks of Lawrence Livermore National Laboratories has implemented gang scheduling of processes on a Sequent system.

Most applications can be solved efficiently with parallel programming. The programming tools described in this chapter can be applied to a wide range of applications, and parallel programmers are constantly

developing new tools that can be run on Sequent systems. With its symmetric architecture, shared memory, and built-in parallel programming support, the Sequent architecture can support almost any application and parallel programming model.

Cha

Data Partitioning wi

T

Illus

Chapter 5

Data Partitioning with DYNIX

5.1. Introduction

This chapter explains how to structure C, FORTRAN, and Pascal programs for data partitioning, and how to use the DYNIX Parallel Programming Library to execute loops in parallel. (Sequent FORTRAN includes special directives for data partitioning of DO loops. If you wish to data partition a FORTRAN DO loop, refer to Chapter 4.)

This chapter is organized as follows:

- Section 5.2 introduces the data partitioning method called *microtasking*.
- Section 5.3 introduces the Parallel Programming Library routines.
- Section 5.4 explains how to analyze data flow within a loop.
- Section 5.5 explains how to structure a microtasking program.
- Section 5.6 briefly explains how to compile, load, execute, and debug your program.
- Section 5.7 lists additional sources of information.

NOTES

Most examples in this chapter are in C or Pascal. The discussion and instructions apply to FORTRAN, C, and Pascal programs except where noted.

The Parallel Programming Library is compatible with Sequent Pascal, pascal*(1), not with Berkeley Pascal,* pc*(1).*

5.2. The Microtasking Method

The data-partitioning method described in this chapter is sometimes called *microtasking*. Microtasking programs create multiple independent processes to execute loop iterations in parallel. The microtasking method has the following characteristics:

- The parallel processes share some data and create their own private copies of other data.
- The division of the computing load adjusts automatically to the number of available processes.
- The program controls data flow and synchronization by using tools specially designed for data partitioning.

You determine which data is shared between parallel processes and how the program adjusts to the number of available CPUs. (Sections 5.4 and 5.5 explain how to do this.) The Parallel Programming Library contains the tools to create and control parallel processes in your microtasking program.

A microtasking program works like this:

- Each loop to be executed in parallel is contained in a subprogram.
- For each loop, the program calls a special function which forks a set of child processes and assigns an identical copy of the subprogram to each process for parallel execution. The special function creates a copy of any private data for each process.
- Each copy of the subprogram executes some of the loop iterations. You can set up the subprogram to use either static scheduling or dynamic scheduling.

- If the loop being executed in
  dent, the subprogram may c
  chronize the parallel processe
  barriers, and other semaphore
- When all the loop iterations h
  from the subprogram. At th
  minates the parallel process
  they are needed to execute a
  to spin in a *busy wait* state u

5.3. The Parallel Programmi

The DYNIX Parallel Programming L
tines: a microtasking library, a set of
partitioning programs, and a set of
data partitioning programs. Appendix
for the Parallel Programming Library

5.3.1 The Microtasking Library

The microtasking library routines
processes, assign the processes to exe
synchronize the processes as neces
between loop iterations. Table 5-1 lis
Parallel Programming Library.

ıod

ribed in this chapter is sometimes
›rograms create multiple independent
n parallel. The microtasking method

re some data and create their own

ng load adjusts automatically to the
es.

ı flow and synchronization by using
lata partitioning.

l between parallel processes and how
of available CPUs. (Sections 5.4 and
.rallel Programming Library contains
·allel processes in your microtasking

this:

in parallel is contained in a subpro-

calls a special function which forks a
assigns an identical copy of the sub-
· parallel execution. The special func-
›rivate data for each process.

·am executes some of the loop itera-
he subprogram to use either static
›duling.

- If the loop being executed in parallel is not completely independent, the subprogram may contain calls to functions that synchronize the parallel processes at critical points by using locks, barriers, and other semaphores.
- When all the loop iterations have been executed, control returns from the subprogram. At this point, the program either terminates the parallel processes, suspends their execution until they are needed to execute another subprogram, or leaves them to spin in a *busy wait* state until they are needed again.

5.3. The Parallel Programming Library

The DYNIX Parallel Programming Library includes three sets of routines: a microtasking library, a set of routines for general use with data partitioning programs, and a set of routines for memory allocation in data partitioning programs. Appendix E contains the DYNIX man pages for the Parallel Programming Library routines.

5.3.1 The Microtasking Library

The microtasking library routines allow you to fork a set of child processes, assign the processes to execute loop iterations in parallel, and synchronize the processes as necessary to provide proper data flow between loop iterations. Table 5-1 lists the microtasking routines in the Parallel Programming Library.

Table 5-1
Parallel Programming Library Microtasking Routines

Routines	Descriptions
`m_fork`	Execute a subprogram in parallel.
`m_get_myid`	Return process identification number.
`m_get_numprocs`	Return number of child processes.
`m_kill_procs`	Terminate child processes.
`m_lock`	Lock a lock.
`m_multi`	End single-process code section.
`m_next`	Increment global counter.
`m_park_procs`	Suspend child process execution.
`m_rele_procs`	Resume child process execution.
`m_set_procs`	Set number of child processes.
`m_single`	Begin single-process code section.
`m_sync`	Check in at barrier.
`m_unlock`	Unlock a lock.

NOTE

The microtasking library is designed around the `m_fork` *routine. The other microtasking routines should be used only in combination with the* `m_fork` *routine. Otherwise, they can cause unexpected side effects.*

5.3.2 Data Partitioning Library

The general-purpose data-partitioning routines include a routine to determine the number of available CPUs and several process synchronization routines that are more flexible than those available in the microtasking library. Table 5-2 lists the general-purpose data-partitioning routines in the Parallel Programming Library.

Tal
Parallel Programming Libra

Routines	De
`cpus_online`	Ret
`s_init_barrier`	Ini
`S_INIT_BARRIER`	C r
`s_init_lock`	Ini
`S_INIT_LOCK`	C r
`s_lock` or `s_clock`	Lo
`S_LOCK` or `s_CLOCK`	C r
`s_unlock`	Un
`S_UNLOCK`	C r
`s_wait_barrier`	Wa
`S_WAIT_BARRIER`	C r

5.3.3 Memory Allocation Routi

The memory allocation routines a
allocate and de-allocate shared m
shared and private memory assig
memory allocation routines in the

Ta
Parallel Programming Libra

Routines	Desc
`brk` or `sbrk`	Char
`shbrk` or `shsbrk`	Char
`shfree`	De-a
`shmalloc`	Alloc

e 5-1
·ary Microtasking Routines

ions
a subprogram in parallel.
process identification number.
number of child processes.
ate child processes.
lock.
gle-process code section.
ent global counter.
d child process execution.
e child process execution.
nber of child processes.
single-process code section.
in at barrier.
a lock.

OTE

y is designed around the ıther microtasking routines ımbination with the `m_fork` *can cause unexpected side*

ing routines include a routine to deter-
Js and several process synchronization
an those available in the microtasking
ıl-purpose data-partitioning routines in

Table 5-2
Parallel Programming Library Data-Partitioning Routines

Routines	Descriptions
`cpus_online`	Return number of CPUs on-line.
`s_init_barrier`	Initialize a barrier.
`S_INIT_BARRIER`	C macro.
`s_init_lock`	Initialize a lock.
`S_INIT_LOCK`	C macro.
`s_lock` or `s_clock`	Lock a lock.
`S_LOCK` or `s_CLOCK`	C macros.
`s_unlock`	Unlock a lock.
`S_UNLOCK`	C macro.
`s_wait_barrier`	Wait at a barrier.
`S_WAIT_BARRIER`	C macro.

5.3.3 Memory Allocation Routines

The memory allocation routines allow a data-partitioning program to allocate and de-allocate shared memory and to change the amount of shared and private memory assigned to a process. Table 5-3 lists the memory allocation routines in the Parallel Programming Library.

Table 5-3
Parallel Programming Library Memory-Allocation Routines

Routines	Descriptions
`brk` or `sbrk`	Change private data segment size.
`shbrk` or `shsbrk`	Change shared data segment size.
`shfree`	De-allocate shared data memory.
`shmalloc`	Allocate shared data memory.

Section 5.5 explains how to use the Parallel Programming Library routines in a program and presents some sample programs. For a detailed reference to the Parallel Programming Library, refer to Section 3P in Volume 1 of the *DYNIX Programmer's Manual*.

5.4. Analyzing Variable Usage

Before you can convert a loop into a subprogram for data partitioning, you must analyze all the variables in the loop and determine two things:

- Which data can be shared between parallel processes and which must be local to each parallel process.
- Which variables cause dependences or *critical regions*, code sections which can yield incorrect results when executed in parallel.

(If you have already read Chapter 4, you are familiar with the information presented in this section. You may wish to turn directly to Section 5.5.)

5.4.1 Shared Variables and Private Variables

A variable must be private if it is initialized in each loop iteration before it is used. All other variables are shared. Private variables are usually scalar (single-element) variables, although other data structures may be private.

The following sample matrix multiply loop contains both shared and private variables. (Assume that the outermost loop is the one to be executed in parallel.)

```
for (i=0; i<n; i++)
    for (k=0; k<n; k++)
        for (j=0; j<n; j++)
            r[i][j] = r[i][j] + s[i][k] * t[k][j];
```

In this loop, the variables `i`, `k`, and `j` are local: they are initialized at the beginning of each loop iteration before they are used. (Remember that we are referring to the outermost loop.)

Once you have identified the private variables, you can declare the shared and private variables in your program. In C, you do this by using the keywords `shared` and `private` in declaration statements. In FORTRAN, you do this by placing all the shared variables in one or more COMMON blocks and then using the **-F** compiler option to declare

those COMMON blocks to be shar
piler option to make all global va
private.

In C, you need to define only stati
private. Automatic variables are
variables cannot be shared. To de
simply add the keyword `shared`
tion statement. For more inform
keywords, refer to the *Sequent C C*

In FORTRAN programs, all variat
are explicitly declared to be share
compiled with the **-mp** option.)
variables in shared COMMON blo
the **-F** option to declare which CO

Section 5.6.1 also explains how to

5.4.2 Identifying Dependent Va

Dependent variables are shared v
by more than one loop iteration.
incorrect information between loo
cuted out of order or if two loop ite
taneously. This section explains h
tion 5.5 presents some special toc
dent variables to ensure correct re

You can use the following simple
variable is dependent:

- Is it a *read-only* variable
written within the loop?
- Is it an array in which
loop iteration? (This occu
with the loop index.)

If the answer to either of these
independent and you simply decla
then the variable is dependent an
dependence.

Dependent variables fall into the f

g with DYNIX

Parallel Programming Library rou-
ıe sample programs. For a detailed
ing Library, refer to Section 3P in
's Manual.

ıge

a subprogram for data partitioning,
ı the loop and determine two things:

ıetween parallel processes and which
el process.

ndences or *critical regions*, code sec-
ect results when executed in parallel.

:, you are familiar with the informa-
may wish to turn directly to Section

ate Variables

ıitialized in each loop iteration before
ıhared. Private variables are usually
though other data structures may be

tiply loop contains both shared and
ıe outermost loop is the one to be exe-

```
r)
; j++)
= r[i][j] + s[i][k] * t[k][j];
```

nd `j` are local: they are initialized at
on before they are used. (Remember
ıost loop.)

.vate variables, you can declare the
your program. In C, you do this by
`private` in declaration statements.
ing all the shared variables in one or
ing the **-F** compiler option to declare

those COMMON blocks to be shared. In Pascal, you use the **-mp** compiler option to make all global variables shared and all local variables private.

In C, you need to define only static or external variables to be shared or private. Automatic variables are handled correctly for you, and register variables cannot be shared. To declare a variable as shared or private, simply add the keyword `shared` or `private` to the variable's declaration statement. For more information on the `shared` and `private` keywords, refer to the *Sequent C Compiler User's Manual.*

In FORTRAN programs, all variables are treated as private unless they are explicitly declared to be shared. (This assumes the program is not compiled with the **-mp** option.) Therefore you must place all shared variables in shared COMMON blocks. Section 5.6.1 explains how to use the **-F** option to declare which COMMON blocks are shared.

Section 5.6.1 also explains how to use the **-mp** Pascal compiler option.

5.4.2 Identifying Dependent Variables

Dependent variables are shared variables that can be read and written by more than one loop iteration. These variables can sometimes pass incorrect information between loop iterations if the iterations are executed out of order or if two loop iterations try to write the variable simultaneously. This section explains how to identify these variables and Section 5.5 presents some special tools and techniques for handling dependent variables to ensure correct results.

You can use the following simple tests to determine whether a shared variable is dependent:

- Is it a *read-only* variable; in other words, is it read but never written within the loop?
- Is it an array in which each element is referenced by only one loop iteration? (This occurs when the array index varies directly with the loop index.)

If the answer to either of these questions is "yes," then the variable is independent and you simply declare it as shared. If the answer is "no," then the variable is dependent and you need to determine the type of its dependence.

Dependent variables fall into the following three categories:

- Reduction variables
- Ordered variables
- Locked variables

The remainder of this section explains how to identify these types of dependent variables. Section 5.5.2 describes techniques for handling each type of dependence in your program.

Reduction Variables

A reduction variable is an array or scalar variable that has the following properties:

- It is used in only one associative, commutative operation within the loop. These operations include addition, multiplication, logical AND, logical OR, and exclusive OR.
- In C or FORTRAN programs, the operation is of the form:

  var = var op expr

  In C programs it may also be of the form:

  var op= expr

  In Pascal programs, the operation is of the form:

  var := var op expr

  where *var* is the reduction variable, *op* is an associative, commutative operation, and *expr* is an expression that does not include the variable *var*. The variable may occur in more that one such statement, as long as the operation is consistent.

The following example loop contains a reduction variable:

```
for (k=0; k<i-1; k++)
    q = q + b[i][k] * w[i-k];
```

In this loop, the variables `b`, `w`, and `i` are independent, because they are read-only within the loop. The variable `q` is a reduction variable. It is used in a single associative, commutative operation (addition) and the operation has the correct form. (The loop index, `k`, is local.)

Locked Variables

A locked variable is an array or sca properties:

- The variable can be read a iteration.
- If the loop iterations were order, the operations invo correct results.

Because a locked variable can be rea iteration and because we intend tc ously, we have to ensure that only able at a time. The mechanism to name locked variable.

The following example computes th number of other cities, then compar distance, and selects the array inde tains one locked variable.

```
x = 0
y = 1
least = 999999;
for (i=1; i<n; i++) {
     xsqdis = sq(bvrtn[
     ysqdis = sq(bvrtn[
     dist   = sqrt(xsqd
     if (dist < least)
          closest = i;
          least   = dist
     }
}
```

In this loop, the variables `bvrtn` ables: they are read-only within `ysqdis`, and `dist` are local: they they are read. The variables `cl` They are read and written by each the iterations are executed does nc involving them. As long as the loo `dist` will be compared with `lea` value of `closest` or `least` bet assignment statement, the loop wil

ns how to identify these types of
describes techniques for handling
am.

alar variable that has the following

ıtive, commutative operation within
ıclude addition, multiplication, logi-
:lusive OR.

ɜ, the operation is of the form:

e of the form:

ration is of the form:

variable, *op* is an associative, com-
expr is an expression that does not
he variable may occur in more that
as the operation is consistent.

a reduction variable:

```
w[i-k];
```

ıd i are independent, because they
ariable q is a reduction variable. It
nutative operation (addition) and the
ɜ loop index, k, is local.)

Locked Variables

A locked variable is an array or scalar variable that has the following properties:

- The variable can be read and written by more than one loop iteration.
- If the loop iterations were executed one at a time in random order, the operations involving the variable would produce correct results.

Because a locked variable can be read and written by more than one loop iteration and because we intend to execute loop iterations simultaneously, we have to ensure that only one loop iteration is using the variable at a time. The mechanism to do this is called a lock, hence the name locked variable.

The following example computes the distance between one city and a number of other cities, then compares each distance with the minimum distance, and selects the array index of the nearest city. This loop contains one locked variable.

```
x = 0
y = 1
least = 999999;
for (i=1; i<n; i++) {
     xsqdis = sq(bvrtn[x]-a[i][x]);
     ysqdis = sq(bvrtn[y]-a[i][y]);
     dist   = sqrt(xsqdis + ysqdis);
     if (dist < least) {
          closest = i;
          least   = dist;
     }
}
```

In this loop, the variables `bvrtn` and `a` are independent shared variables: they are read-only within the loop. The variables `xsqdis`, `ysqdis`, and `dist` are local: they are written in each iteration before they are read. The variables `closest` and `least` must be locked. They are read and written by each loop iteration, but the order in which the iterations are executed does not affect the results of the operations involving them. As long as the loop is executed `n` times, each value of `dist` will be compared with `least`. As long as nothing changes the value of `closest` or `least` between the `if` statement and either assignment statement, the loop will return the correct answers.

Ordered Variables

An ordered variable is an array or scalar variable that has the following property:

- The loop consistently yields correct results only if the operations involving the variable are executed one iteration at a time, in serial order.

The following example loop contains two ordered variables.

```
for (i=0; i < n; i++ ) {
    x(i) = xa(i) + xb(i);
    dx = x(i) - x(i-1);
    y(i) = ya(i) + yb(i);
    dy = y(i) - y(i-1);
    rho(i) = sqrt(dx * dx + dy * dy);
}
```

In this loop, the variables `xa`, `xb`, `ya`, and `yb` are shared, because they are all read-only. The variables `dx` and `dy` are local because they are initialized in each loop iteration before their values are used. The variables `x` and `y` are ordered. If the loop iterations were executed in random order, the operations involving `x` and `y` would produce different values than when the loop is executed in sequential order.

5.4.3 Variable Analysis Worksheet

As you analyze the variables in your loop, you may find it helpful to use the worksheet shown in in Figure 5-1.

	SHARED	LOCAL
VARIABLE NAME	*Is the variable read-only within the loop OR is it an array where each element is read and written by only one loop iteration?*	*Could the variable be renamed in each iteration without affecting the program result?*

Fig. 5-1. Variable

To use this worksheet, simply list first column. For each variable, n tions until you either answer "yes tions. When you mark a "yes" in type in the label at the top of the c

VARIABLE NAME	SHARED *Is the variable read-only within the loop OR is it an array where each element is read and written by only one loop iteration?*	LOCAL *Could the variable be renamed in each iteration without affecting the program result?*	REDUCTION *Is the variable used in only one associative, commutative operation within the loop AND is it always read, then written?*	SHARED ORDERED *If the loop iterations were exectued in random order, would the operations involving this variable produce different results?*	SHARED LOCKED *Have you answered "no" to all the other questions?*

1003-45453A

Fig. 5-1. Variable analysis worksheet.

To use this worksheet, simply list all the variables in your loop in the first column. For each variable, mark your answers to the listed questions until you either answer "yes" to one question or run out of questions. When you mark a "yes" in any column, you'll find the variable type in the label at the top of the column.

5.5. The Microtasking Program

This section explains how to structure a microtasking program. In such a program, each loop to be executed in parallel is contained in a subprogram which we will call the *looping subprogram*. Section 5.5.1 describes the calling program, Section 5.5.2 describes the looping subprogram, Section 5.5.3 discusses shared memory allocation, and Section 5.5.4 presents some complete program examples.

5.5.1 The Calling Program

The calling program handles the following tasks:

- Including any header files required by the Parallel Programming Library routines (C programs only).
- Determining how many parallel processes are created to execute the loop. This determination is based on the number of CPUs in the system. The program can either call the Parallel Programming library routine `m_set_procs` or it can use the default number computed by the Parallel Programming Library.
- Calling the Parallel Programming Library routine `m_fork` to execute each looping subprogram in parallel.
- Suspending or terminating parallel processes between calls to looping subprograms, and terminating all parallel processes after the last looping subprogram has been executed.

Parallel Programming Library Header File

DYNIX includes two C header files which contain declaration statements for the Parallel Proramming Library routines. One file contains declarations for the microtasking routines and the other contains declarations for the other routines. Both of these header files reside in the directory */usr/include/parallel*. The header files are named *microtask.h* and *parallel.h*. Refer to Section 3P in the *DYNIX Programmer's Manual* for information on which file to include for a specific routine.

Determining How Many Parallel Processes to Use

To determine how many parallel processes your program will use to execute the loop subprogram, you can either call the Parallel Programming Library routine `m_set_procs` or you can use a default number

computed by the Parallel Prorgam
function sets the number of proces
calls to the routine `m_fork`. (This
If your program uses `m_set_procs`
tine `cpus_online` to find out how

By default, the number of processes
number of CPUs on-line divided by
function, you can set this number
number of CPUs on-line minus 1.

In C, the calls to the **cpus_online**
this:

var = `cpus_online()`

val = `m_set_procs(`*n*

In Pascal, the calls to these function

var := `cpus_online`

val := `m_set_procs(`

In FORTRAN, the calls to these fun

var = `cpus_online(`

val = `m_set_procs(`*n*

The variables *var*, *val*, and *nprocs*
grams, type `longint` in Pascal
FORTRAN programs.

Calling the Looping Subprogram

The Parallel Programming Library
subprogram in parallel. `M_fork`
existing processes and assigns them
subprogram. It can also pass an ar

In C, the `m_fork` function call look

`m_fork(`*func*`[,`*arg*`,..`

In Pascal, the `m_fork` function call

`m_pfork(`*func*`[,`*arg*`,.`

·am

ɜ a microtasking program. In such
n parallel is contained in a subpro-
ubprogram. Section 5.5.1 describes
:cribes the looping subprogram, Sec-
ry allocation, and Section 5.5.4
nples.

wing tasks:

ɪuired by the Parallel Programming
ɜ only).

llel processes are created to execute
is based on the number of CPUs in
ɪn either call the Parallel Program-
:_procs or it can use the default
·allel Programming Library.

nming Library routine `m_fork` to
ram in parallel.

parallel processes between calls to
terminating all parallel processes
ɡram has been executed.

:ader File

·hich contain declaration statements
routines. One file contains declara-
and the other contains declarations
ɜ header files reside in the directory
files are named *microtask.h* and
e *DYNIX Programmer's Manual* for
ɔr a specific routine.

Processes to Use

cesses your program will use to exe-
ither call the Parallel Programming
you can use a default number

computed by the Parallel Prorgamming Library. The `m_set_procs` function sets the number of processes that will exist after subsequent calls to the routine `m_fork`. (This number includes the parent process.) If your program uses `m_set_procs`, you may want to also use the routine `cpus_online` to find out how many CPUs are currently on line.

By default, the number of processes created by `m_fork` is equal to the number of CPUs on-line divided by two. By using the `m_set_procs` function, you can set this number as low as one or as high as the number of CPUs on-line minus 1.

In C, the calls to the **cpus_online** and *m_set_procs* functions look like this:

```
var = cpus_online();

val = m_set_procs(nprocs);
```

In Pascal, the calls to these functions look like this:

```
var := cpus_online();

val := m_set_procs(nprocs);
```

In FORTRAN, the calls to these functions look like this:

```
var = cpus_online()

val = m_set_procs(nprocs)
```

The variables *var*, *val*, and *nprocs* must all be of type `int` in C programs, type `longint` in Pascal programs, and type `INTEGER*4` in FORTRAN programs.

Calling the Looping Subprogram: The m_fork Routine

The Parallel Programming Library function `m_fork` executes the looping subprogram in parallel. `M_fork` creates processes or reuses a set of existing processes and assigns them to execute copies of the specified loop subprogram. It can also pass an argument list to each copy.

In C, the `m_fork` function call looks like this:

```
m_fork(func[,arg,...]);
```

In Pascal, the `m_fork` function call looks like this:

```
m_pfork(func[,arg,...]);
```

In FORTRAN, the `m_fork` function call looks like this:

```
external func
call m_fork(func[,arg,...])
```

The *func* argument is the name of the looping subprogram and the arguments *arg* are its parameters. These parameters can be of any type. In a C program, you must declare the `m_fork` function to be of type `void`.

When the `m_fork` function is called, it determines whether there are existing child processes, processes created by a previous `m_fork` call. If there are existing child processes, it reuses them to execute the loop subprogram. If not, it creates a new set of child processes to execute the subprogram.

The `m_fork` routine creates enough child processes to bring the total number of processes (including the parent process) to either the default (number of CPUs on-line/2) or the number you set with a previous call to the `m_set_procs` function. As `m_fork` creates child processes, it assigns each process a private integer variable called *m_myid*, which uniquely indentifies that child process within the set of processes belonging to that program. The main program (the parent process) has the *m_myid* value 0, the first child process created has the *m_myid* value 1, and so on. You can find the identification number of any process by calling the Parallel Programming Library function `m_get_myid`.

Once child processes are available, `m_fork` passes them copies of their parameters and starts them executing the looping subprogram *func*. When all the child processes are started, the parent process gives itself a copy of the loop subprogram and parameters, and all the processes execute the loop subprogram until they all return from it. At this point, the child processes spin, waiting for more work. The parent process can either kill the child processes, suspend them, or let them spin until they are reused by another `m_fork` call.

Re-using and Terminating Parallel Processes

As explained in Section 5.2, a program typically forks as many child processes as it needs at the beginning and does not terminate them until all parallel computation is complete. The Parallel Programming Library includes three routines to manage child processes after `m_fork` calls: `m_park_procs`, `m_rele_procs`, and `m_kill_procs`. By default, after the program returns from an `m_fork` call, the child processes spin, using CPU time. If your program requires a lot of computation before the next `m_fork` call, it can suspend the child processes and relinquish their CPUs for use by other processes by calling the `m_park_procs` routine.

The program then resumes child
`m_rele_procs` routine. After the la
call the routine `m_kill_procs` to te

5.5.2 The Looping Subprogram

This section explains how to construc
to executing a loop in parallel, the
lowing tasks:

- *Scheduling*, determining wl
iterations .
- Protecting code sections th
that they yield correct resul
- Synchronizing processes as
- Handling I/O, if required.

Static and Dynamic Scheduling

In data-partitioning programs, you
scheduling. Static scheduling re
processes. Dynamic scheduling req
even out an unbalanced computing l

Static Scheduling. If you know
mately the same for each iteratio
scheduling. The static scheduling
iterations evenly among the processe

The static scheduling algorithm f
steps:

1. Call the Parallel
`m_get_numprocs` to dete
created by the `m_fork` call
2. Call the Parallel Programm
find out my process ID num
3. Start by executing the *Nth*

all looks like this:

```
;,...])
```

looping subprogram and the argu-
parameters can be of any type. In
_fork function to be of type void.

:, it determines whether there are
ated by a previous m_fork call. If
euses them to execute the loop sub-
et of child processes to execute the

child processes to bring the total
arent process) to either the default
mber you set with a previous call to
1_fork creates child processes, it
ger variable called *m_myid*, which
s within the set of processes belong-
gram (the parent process) has the
ss created has the *m_myid* value 1,
ation number of any process by cal-
y function m_get_myid.

m_fork passes them copies of their
ting the looping subprogram *func*.
ted, the parent process gives itself a
rameters, and all the processes exe-
all return from it. At this point, the
ore work. The parent process can
ad them, or let them spin until they

lel Processes

gram typically forks as many child
ng and does not terminate them until
. The Parallel Programming Library
child processes after m_fork calls:
id m_kill_procs. By default, after
fork call, the child processes spin,
quires a lot of computation before the
e child processes and relinquish their
calling the m_park_procs routine.

The program then resumes child process execution by calling the m_rele_procs routine. After the last m_fork call, the program should call the routine m_kill_procs to terminate the child processes.

5.5.2 The Looping Subprogram

This section explains how to construct a looping subprogram. In addition to executing a loop in parallel, the looping subprogram handles the following tasks:

- *Scheduling*, determining which process will execute which loop iterations .
- Protecting code sections that contain dependent variables so that they yield correct results.
- Synchronizing processes as necessary.
- Handling I/O, if required.

Static and Dynamic Scheduling

In data-partitioning programs, you can use either static or dynamic scheduling. Static scheduling requires no communication between processes. Dynamic scheduling requires more communication, but can even out an unbalanced computing load.

Static Scheduling. If you know that the computing time is approximately the same for each iteration of your loop, you can use static scheduling. The static scheduling algorithm simply divides the loop iterations evenly among the processes.

The static scheduling algorithm for a process involves the following steps:

1. Call the Parallel Programming Library routine m_get_numprocs to determine how many processes were created by the m_fork call. (We'll call this number M.)
2. Call the Parallel Programming library routine m_get_myid to find out my process ID number. (We'll call this number N.)
3. Start by executing the *Nth* loop iteration.

4. Execute every *Mth* iteration until I reach the end of the loop.

Refer to Section 5.5.4 for an example program that uses static scheduling.

Dynamic Scheduling. If you know that the computing time varies for each iteration of your loop, you can use dynamic scheduling. With dynamic scheduling, the loop iterations are treated as a task queue, and each process removes one or more iterations from the queue, executes those iterations, and returns for more work. This method is sometimes called "hungry puppies" because the processes "nibble" away at the work until it is all done.

Dynamic scheduling creates more communication overhead than static scheduling because all the processes must access a single shared task queue, but the computing load can be very evenly distributed because no process is idle while there is still work to be done. For data partitioning, the task queue can be implemented by using the `m_next` routine.

A typical dynamic scheduling algorithm includes the following steps:

1. Lock a lock.
2. Check shared loop index and verify that there is still work to be done.
3. Increment or decrement the shared loop index by *N*. (The `m_next` routine is useful for this if your shared loop index can start at zero and increment.)
4. Unlock the lock.
5. Execute *N* iterations.
6. Repeat steps 1 through 5 until all the work is finished.

If you use the `m_next` routine, you do not need to explicitly lock and unlock a lock. These steps are built into `m_next`. Refer to Section 5.5.4 for an example program that uses `m_next` in dynamic scheduling.

Handling Dependent Variables

This section describes techniques for handling order, reduction, and lock-type data dependencies.

Handling Locked Sections. If you need to use the Parallel Programm `m_unlock` to ensure that the code s executed by only one loop iteration a appear on the line immediately prec variable, and the `m_unlock` call sh following the last reference to a locke

Refer to Section 5.5.4 for an exampl to protect the shared loop index in a gram.

The `m_lock` and `m_unlock` routine subprogram. If your program requir can use the `s_init_lock`, `s_lock` tines. Refer to the *s_lock*(5P) man *Manual* for more information on thes

Handling Reduction Variables. locked variables, except that you ne part of the time. You can create a within the parallel loop routine, and for the reduction variable name thro loop subprogram, you can call the tion operation to combine the local reduction variable, and call the efficient than an ordinary locked var the locked section only once.

For example, consider the following e

```
for (k=mystart; k<end;
    q = q + b[i][k];
```

The reduction variable `q` is shared. in any order, but the loop can produ try to read or write `q` simultaneousl this way, it cannot be executed in p cal variable, `lq`, each process can affecting any other process. Once e it can lock the shared variable `q`, ad

il I reach the end of the loop.

ple program that uses static

at the computing time varies for use dynamic scheduling. With are treated as a task queue, and ations from the queue, executes work. This method is sometimes cesses "nibble" away at the work

nunication overhead than static ust access a single shared task ery evenly distributed because no o be done. For data partitioning, using the `m_next` routine.

includes the following steps:

erify that there is still work to be

shared loop index by N. (The his if your shared loop index can

all the work is finished.

not need to explicitly lock and un- n_next. Refer to Section 5.5.4 for : in dynamic scheduling.

andling order, reduction, and lock-

Handling Locked Sections. If your loop contains locked variables, you need to use the Parallel Programming Library routines `m_lock` and `m_unlock` to ensure that the code section containing those variables is executed by only one loop iteration at a time. The `m_lock` call should appear on the line immediately preceding the first reference to a locked variable, and the `m_unlock` call should appear on the line immediately following the last reference to a locked variable.

Refer to Section 5.5.4 for an example program that uses these routines to protect the shared loop index in a dynamically scheduled loop subprogram.

The `m_lock` and `m_unlock` routines support only one lock per looping subprogram. If your program requires more than one lock at a time, you can use the `s_init_lock`, `s_lock` or `s_clock`, and `s_unlock` routines. Refer to the *s_lock*(5P) man page in the *DYNIX Programmer's Manual* for more information on these routines.

Handling Reduction Variables. Reduction variables are similar to locked variables, except that you need to protect them with locks only part of the time. You can create a local reduction variable, initialize it within the parallel loop routine, and substitute the local variable name for the reduction variable name throughout the loop. At the end of the loop subprogram, you can call the `m_lock` function, perform the reduction operation to combine the local reduction variable with the shared reduction variable, and call the `m_unlock` function. This is more efficient than an ordinary locked variable because each process executes the locked section only once.

For example, consider the following example loop from Section 5.4.2:

```
for (k=mystart; k<end; k+=incr)
    q = q + b[i][k];
```

The reduction variable `q` is shared. The loop iterations can be executed in any order, but the loop can produce incorrect results if two processes try to read or write `q` simultaneously. As long as the loop is structured this way, it cannot be executed in parallel. However, if we declare a local variable, `lq`, each process can add its values of `b` to `lq` without affecting any other process. Once each process finishes its calculations, it can lock the shared variable `q`, add its `lq` value, and unlock `q`.

```
lq = 0;
for (k=mystart; k<end; k+=incr)
    lq = lq + b[i][k];
m_lock();
q = q + lq;
m_unlock();
```

Handling Ordered Sections. If your loop contains an ordered variable, you need to ensure that the code sections containing that variable are executed in loop iteration order. To ensure this, repeat the following procedure for each ordered variable in the loop.

1. In the main program, declare a shared integer variable to hold the current loop iteration number. (If the shared ordered variable is named `i`, you might name the new variable something like `iguard`.) Initialize the new variable to the starting value of the loop index.
2. In the looping subprogram, on the line before the first reference to the shared ordered variable, insert a conditional statement that loops on itself until the loop index value is equal to the value of the iteration count variable.
3. On the line after the last reference to the shared ordered variable, insert a statement to increment the shared iteration counter variable.

NOTE

At some optimization levels, the C optimizer can remove conditional tests in spin loops. If your codes uses any spin loops on shared variables, always compile with the **-i** *compiler option to ensure that the conditional tests are preserved. For more information on the* **-i** *option, refer to* cc(1).

If the ordered variable is written and then read more than once within the loop, you can speed up execution by treating each write/read sequence as a different variable. This allows execution to proceed in parallel between ordered sections.

The following example loop from Section 5.4.2 illustrates these modifications. The shared variables `x` and `y` are ordered. Assume that we have declared two shared variables named `xguard` and `yguard` in the main program and initialized them to zero.


```
for (i=0; i < n; i++ ) {
    while (xguard != i)
        continue;
    x(i) = xa(i) + xb(i)
    dx = x(i) - x(i-1);
    xguard = xguard + 1
    while (yguard != i)
        continue;
    y(i) = ya(i) + yb(i
    dy = y(i) - y(i-1);
    yguard = yguard + 1
    rho(i) = sqrt(dx * (
}
```

Synchronizing Processes

A looping subprogram sometimes con
on all the processes having complet
For example, a looping subprogram m
the same set of data, and the alg
processes finish executing the first
second loop. In such situations, you
the processes.

The Parallel Programming Library ir
of barriers. The routine `m_sync` syn
gle, pre-initialized barrier. To set mo
ize a subset of the processes, t
`s_init_barrier` to initialize a barr
to synchronize processes at the barrie

Handling I/O

Section 2.9 mentioned the complicat
portion of a program. The Parallel
avoid these complications by setting
looping subprogram. The looping su
gramming Library routine `m_singl`
while the parent process performs I
routine to start child process executi
while the parent is doing I/O.

with DYNIX

k+=incr)

loop contains an ordered variable,
ns containing that variable are ex-
ire this, repeat the following pro-
loop.

a shared integer variable to hold
nber. (If the shared ordered vari-
name the new variable something
iew variable to the starting value

i the line before the first reference
le, insert a conditional statement
loop index value is equal to the
riable.

erence to the shared ordered vari-
increment the shared iteration

E

the C optimizer can re-
loops. If your codes uses
ariables, always compile
ensure that the condition-
ore information on the **-i**

then read more than once within
i by treating each write/read se-
llows execution to proceed in paral-

Section 5.4.2 illustrates these
t and y are ordered. Assume that
s named xguard and yguard in
i to zero.


```
for (i=0; i < n; i++ ) {
    while (xguard != i)
        continue;
    x(i) = xa(i) + xb(i);
    dx = x(i) - x(i-1);
    xguard = xguard + 1;
    while (yguard != i)
        continue;
    y(i) = ya(i) + yb(i);
    dy = y(i) - y(i-1);
    yguard = yguard + 1;
    rho(i) = sqrt(dx * dx + dy * dy);
}
```

Synchronizing Processes

A looping subprogram sometimes contains a code section which depends on all the processes having completed execution of the preceding code. For example, a looping subprogram might execute more than one loop on the same set of data, and the algorithm might require that all the processes finish executing the first loop before starting to execute the second loop. In such situations, you can set up barriers to synchronize the processes.

The Parallel Programming Library includes routines to set up two kinds of barriers. The routine `m_sync` synchronizes all the processes at a single, pre-initialized barrier. To set more than one barrier, or to synchronize a subset of the processes, the looping subprogram can call `s_init_barrier` to initialize a barrier and then call `s_wait_barrier` to synchronize processes at the barrier.

Handling I/O

Section 2.9 mentioned the complications of doing I/O from the parallel portion of a program. The Parallel Programming Library allows you to avoid these complications by setting up single-process sections within a looping subprogram. The looping subprogram can call the Parallel Programming Library routine `m_single` to halt execution of child processes while the parent process performs I/O. It can then call the `m_multi` routine to start child process execution again. The child processes *spin* while the parent is doing I/O.

5.5.3 Shared Memory Allocation

The Parallel Programming Library contains a set of routines for dynamic allocation and management of shared memory. For C programs, the `shmalloc` and `shfree` routines allocate and release shared memory for data structures whose size is determined at run time. The `shmalloc` routine returns a shared pointer to the newly allocated shared memory. (In Pascal, dynamic shared memory allocation is handled by the `NEW` routine, and FORTRAN does not allow dynamic memory allocation.)

The `shbrk` and `shsbrk` routines increase the size of a process's shared data segment and verify that the increase does not cause the shared data segment to overlap the process's shared stack. The Parallel Programming Library `brk` and `sbrk` routines are used like the standard DYNIX `brk` and `sbrk` to increase a process's private data segment size, but they also verify that the increase does not cause the private data segment to overlap the process's shared data segment.

The **-Z** linker option also allows you to control the size and base address of the shared data segment. For more information on this option, refer to the *ld*(1) man page in the *DYNIX Programmer's Manual.*

5.5.4 Example Programs

Static Scheduling - C Example

```
/* multiply two matrices, store results in third matrix,
   and print results */

#include <stdio.h>
#include <parallel/microtask.h> /* microtasking header */
#include <parallel/parallel.h>  /* parallel lib header */
#define SIZE 10                 /* size of matrices */

      /* Global shared memory data */

       shared float a[SIZE][SIZE]; /* first array */
       shared float b[SIZE][SIZE]; /* second array */
       shared float c[SIZE][SIZE]; /* result array */

main ()
{
     void init_matrix(), m_fork(), m_kill_procs(),
          matmul(), print_mats();
     int nprocs;  /* number of parallel processes */
```



```
     printf("Enter number of
     scanf("%d",&nprocs);

     init_matrix(a, b);
     m_set_procs(nprocs);
     m_fork(matmul, a, b, c);
     m_kill_procs();
     print_mats(a, b, c);
}

/* initialize matrix function

void
init_matrix(a, b)
float a[][SIZE], b[][SIZE];
{
    int i, j;

    for (i = 0; i < SIZE; i +
        for (j = 0; j < SIZE;
            a[i][j] = (float
            b[i][j] = (float
        }
    }
}

/* matrix multiply function *

void
matmul(a, b, c)
float a[][SIZE], b[][SIZE], c
{
    int i, j, k, nprocs;

    nprocs = m_get_numprocs()
    for (i = m_get_myid(); i
        for (j = 0; j < SIZE;
            for (k = 0; k <
                c[i][k] += a
        }
    }
}

/* print results function */

void
print_mats(a, b, c)
float a[][SIZE], b[][SIZE], c
```

ontains a set of routines for dynamic
ed memory. For C programs, the
cate and release shared memory for
mined at run time. The shmalloc
the newly allocated shared memory.
y allocation is handled by the NEW
w dynamic memory allocation.)

crease the size of a process's shared
increase does not cause the shared
s's shared stack. The Parallel Pro-
routines are used like the standard
process's private data segment size,
ase does not cause the private data
red data segment.

to control the size and base address
re information on this option, refer to
rogrammer's Manual.

```
ore results in third matrix,


.h> /* microtasking header */
.h>  /* parallel lib header */
     /* size of matrices */

cy data */

[SIZE]; /* first array */
[SIZE]; /* second array */
[SIZE]; /* result array */


fork(), m_kill_procs(),
ts();
 of parallel processes */
```



```
      printf("Enter number of processes:");
      scanf("%d",&nprocs);

      init_matrix(a, b);           /* initialize data */
      m_set_procs(nprocs);         /* set # of processes */
      m_fork(matmul, a, b, c);/* execute parallel loop */
      m_kill_procs();           /* kill child processes */
      print_mats(a, b, c);         /* print results */
}

/* initialize matrix function */

void
init_matrix(a, b)
float a[][SIZE], b[][SIZE];
{
    int i, j;

    for (i = 0; i < SIZE; i ++) {
        for (j = 0; j < SIZE; j ++) {
            a[i][j] = (float)i + j;
            b[i][j] = (float)i - j;
        }
    }
}

/* matrix multiply function */

void
matmul(a, b, c)
float a[][SIZE], b[][SIZE], c[][SIZE];
{
    int i, j, k, nprocs;

    nprocs = m_get_numprocs();    /* no. of processes */
    for (i = m_get_myid(); i < SIZE; i += nprocs) {
        for (j = 0; j < SIZE; j ++) {
            for (k = 0; k < SIZE; k ++)
                c[i][k] += a[i][j] * b[j][k];
        }
    }
}

/* print results function */

void
print_mats(a, b, c)
float a[][SIZE], b[][SIZE], c[][SIZE];
```

```
{
    int i, j;

    for (i = 0; i < SIZE; i ++) {
        for (j = 0; j < SIZE; j ++) {
            printf("a[%d][%d] = %3.2fb[%d][%d] = %3.2f",
               i, j, a[i][j], i, j, b[i][j]);
            printf("c[%d][%d] = %3.2f\n", i, j,
               c[i][j]);
        }
    }
}
```

Static Scheduling - Pascal Example

```
{ multiply two matrices, store results in third
   matrix, and print results }

program matrix_mul ;

const

SIZE = 9 ;     [ (size of matrices)-1 ]

type

matrix = array[0..SIZE, 0..SIZE] of real;
integer = longint;

var

a : matrix ;        { first array }
b : matrix ;        { second array }
c : matrix ;        { result array }
nprocs: longint; { number of processes }
ret_val: longint; { return value for m_set_procs }

procedure m_lock;
    cexternal;
procedure m_unlock;
    cexternal;
function m_set_procs(var i : longint) : longint;
    cexternal;
procedure m_pfork(procedure a);
    cexternal;
function m_get_numprocs : longint;
    cexternal;
```



```
function m_get_myid : longint;
    cexternal;
procedure m_kill_procs;
    cexternal;

{ initialize matrix function }

procedure init_matrix ;
var
i, j : integer ;
begin
    for i := 0 to SIZE do
    begin
        for j := 0 to SIZE do
        begin
            a[i, j] := (i + j
            b[i, j] := (i - j
        end;
    end;
end; { init_matrix }

{ matrix multiply function }

procedure matmul ;

var

i, j, k : integer; { local loo
nprocs  : integer; { number of

begin
    nprocs := m_get_numprocs;
    i := m_get_myid;      { sta
    while (i <= SIZE) do
    begin
        for j := 0 to SIZE do
        begin
            for k := 0 to SIZ
                c[i, k] := c[
        end;
        i := i + nprocs;
    end;
end; { matmul}

{ print results procedure }

procedure print_mats ;
var
```

```
++) {
:; j ++) {
i] = %3.2fb[%d][%d] = %3.2f",
, i, j, b[i][j]);
i] = %3.2f\n", i, j,
```

ple

```
re results in third
 }
```

```
rices)-1 }
```

```
IZE] of real;
```

```
rray }
 array }
 array }
E processes }
value for m_set_procs }
```

```
: longint) : longint;

 a);

ongint;
```



```
function m_get_myid : longint;
    cexternal;
procedure m_kill_procs;
    cexternal;

{ initialize matrix function }

procedure init_matrix ;
var
i, j : integer ;
begin
    for i := 0 to SIZE do
    begin
        for j := 0 to SIZE do
        begin
            a[i, j] := (i + j) ;
            b[i, j] := (i - j) ;
        end;
    end;
end; { init_matrix }

{ matrix multiply function }

procedure matmul ;

var

i, j, k : integer; { local loop indices }
nprocs  : integer; { number of processes }

begin
    nprocs := m_get_numprocs;     { number of processes }
    i := m_get_myid;     { start at Nth iteration }
    while (i <= SIZE) do
    begin
        for j := 0 to SIZE do
        begin
            for k := 0 to SIZE do
                c[i, k] := c[i, k] + a[i, j] * b[j, k];
        end;
        i := i + nprocs;
    end;
end; { matmul}

{ print results procedure }

procedure print_mats ;
var
```

```
i, j : integer; { local loop indices }
begin
    for i := 0 to SIZE do
    begin
        for j := 0 to SIZE do
        begin
            writeln('a[',i,',',j,'] = ',a[i,j],
               'b[',i,',',j,'] = ',b[i,j],'  c[',i,',',
               j,'] = ',c[i, j]);
        end;
    end;
end; {print_mats}

begin { main program starts here}

    writeln('Enter number of processes:');
    readln(nprocs);

    init_matrix;          { initialize data arrays }
    ret_val := m_set_procs(nprocs);   { set # of processes }
    m_pfork(matmul);        { do matrix multiply }
    m_kill_procs;         { terminate child processes }
    print_mats;           { print results }

end. { main program }
```

Dynamic Scheduling - C Example

```
/* use Cartesian coordinates to find the city closest to
   Beaverton, Oregon, and print the name and distance
   from Beaverton */

#include <stdio.h>
#include <math.h>
#include <parallel/microtask.h> /* microtasking header */
#include <parallel/parallel.h>  /* parallel library
                                   header */

#define NCITIES 10      /* number of cities */
#define BITE 1          /* bite of work for hungry puppy */

      /* Global shared memory data */

    shared float shortest;    /* distance to
                                 nearest city */
    shared int closest;       /* index of
                                 nearest city */
```



```
    struct location {
        char *name;
        float x, y;
    };
    shared struct location c
        { "CHICAGO", 2000.,
        { "DENVER", 500., -5
        { "NEW YORK", 150.,
        { "SEATTLE", 0., 200
        { "MIAMI", 3500., -2
        { "SAN FRANCISCO", -
        { "RENO", 200., -600
        { "PORTLAND", -17.,
        { "WASHINGTON D.C.",
        { "TILLAMOOK", -70.,
    };
    shared struct location b
        0., 0. };

main ()
{
  void get_cities(), find_di

  shortest = 999999999.;
  m_fork(find_dis, cities);
  printf("%s is closest to B
      cities[closest].name);
  printf("%s is %3.2f miles
      cities[closest].name,

}
/* find distance to nearest

void
find_dis(cities)
struct location cities[];
{
int i, base, top;  /* local
float xsqdis, ysqdis, dist;

while ((base = BITE*(m_next(
    top = base + BITE;   /*
    if (top >= NCITIES)
        top = NCITIES-1;

        /* execute all itera

        for (i = base; i <=
            xsqdis = pow(fab
```

```
    struct location {
        char *name;
        float x, y;
    };
    shared struct location cities[NCITIES] = {
        { "CHICAGO", 2000., 100. },
        { "DENVER", 500., -550. },
        { "NEW YORK", 150., 100. },
        { "SEATTLE", 0., 200. },
        { "MIAMI", 3500., -2000. },
        { "SAN FRANCISCO", -100., -1000. },
        { "RENO", 200., -600. },
        { "PORTLAND", -17., 0.  },
        { "WASHINGTON D.C.", 3000., -400. },
        { "TILLAMOOK", -70., -50. },
    };
    shared struct location beaverton = { "BEAVERTON",
        0., 0. };

main ()
{
  void get_cities(), find_dis(), m_fork();

  shortest = 999999999.;
  m_fork(find_dis, cities);
  printf("%s is closest to Beaverton.0,
      cities[closest].name);
  printf("%s is %3.2f miles from Beaverton.\n",
      cities[closest].name, shortest);
}
/* find distance to nearest city */

void
find_dis(cities)
struct location cities[];
{
int i, base, top;  /* local loop index, start & end value */
float xsqdis, ysqdis, dist;

while ((base = BITE*(m_next( )-1)) < NCITIES) {
    top = base + BITE;   /* take a bite of work */
    if (top >= NCITIES)
        top = NCITIES-1;

        /* execute all iterations in bite of work */

        for (i = base; i <= top; i++) {
            xsqdis = pow(fabs(beaverton.x - cities[i].x),2.);
```

```
            ysqdis = pow(fabs(beaverton.y - cities[i].y),2.
            dist   = sqrt(xsqdis + ysqdis);
            m_lock();
            if (dist < shortest) {
                closest = i;
                shortest = dist;
                }
            m_unlock();
        }
    }
}
```

Dynamic Scheduling - Pascal Example

```
{ use Cartesian coordinates to find the city closest
   to Beaverton, Oregon, and print the name and
   distance from Beaverton }

program find_distance ;

const

NCITIES = 10;      { number of cities }
BITE = 1;          { bite of work for a hungry puppy }

type

cityrecord =
    record
     name : string [15]; { names of cities }
     x : real;          { x coordinates }
     y : real           { y coordinates }
    end;

var

closest : integer ; { index of nearest city }
shortest : real ;   { distance to nearest city }
cities : array[1..NCITIES] of cityrecord ; { city info }
beaverton : cityrecord ; { coordinates of Beaverton }

procedure m_lock;
    cexternal;
procedure m_unlock;
    cexternal;
procedure m_pfork(procedure a);
    cexternal;
```



```
function m_next : longint;
    cexternal;

{ initialize array of city dat

procedure init_cities ;

begin

    cities[1].name := 'CHICAG
    cities[1].x := 2000.0;
    cities[1].y := 100.0;
    cities[2].name := 'DENVER
    cities[2].x := 500.0;
    cities[2].y := -550.0;
    cities[3].name := 'NEW YO
    cities[3].x := 1500.0;
    cities[3].y := 100.0;
    cities[4].name := 'SEATTL
    cities[4].x := 0.0;
    cities[4].y := 200.0;
    cities[5].name := 'MIAMI'
    cities[5].x := 3500.0;
    cities[5].y := 2000.0;
    cities[6].name := 'SAN FR
    cities[6].x := -100.0;
    cities[6].y := -1000.0;
    cities[7].name := 'RENO';
    cities[7].x := 200.0;
    cities[7].y := -600.0;
    cities[8].name := 'PORTLA
    cities[8].x := -17.0;
    cities[8].y := 0.0;
    cities[9].name := 'WASHIN
    cities[9].x := 3000.0;
    cities[9].y := -400.0;
    cities[10].name := 'TILLA
    cities[10].x := -70.0;
    cities[10].y := -50.0;

    beaverton.name := 'BEAVER
    beaverton.x := 0.0;
    beaverton.y := 0.0;

end; { of init_cities }

{ find distance to nearest ci
procedure find_dis;
```

```
s(beaverton.y - cities[i].y),2.);
qdis + ysqdis);

est) {
;
dist;
```

ample

```
to find the city closest
 print the name and
```

```
of cities }
 work for a hungry puppy }
```

```
names of cities }
ordinates }
ordinates }
```

```
 of nearest city }
nce to nearest city }
of cityrecord ; { city info }
coordinates of Beaverton }
```

```
a);
```



```
function m_next : longint;
    cexternal;

{ initialize array of city data }

procedure init_cities ;

begin

    cities[1].name := 'CHICAGO';
    cities[1].x := 2000.0;
    cities[1].y := 100.0;
    cities[2].name := 'DENVER';
    cities[2].x := 500.0;
    cities[2].y := -550.0;
    cities[3].name := 'NEW YORK';
    cities[3].x := 1500.0;
    cities[3].y := 100.0;
    cities[4].name := 'SEATTLE';
    cities[4].x := 0.0;
    cities[4].y := 200.0;
    cities[5].name := 'MIAMI';
    cities[5].x := 3500.0;
    cities[5].y := 2000.0;
    cities[6].name := 'SAN FRANCISCO';
    cities[6].x := -100.0;
    cities[6].y := -1000.0;
    cities[7].name := 'RENO';
    cities[7].x := 200.0;
    cities[7].y := -600.0;
    cities[8].name := 'PORTLAND';
    cities[8].x := -17.0;
    cities[8].y := 0.0;
    cities[9].name := 'WASHINGTON D.C';
    cities[9].x := 3000.0;
    cities[9].y := -400.0;
    cities[10].name := 'TILLAMOOK';
    cities[10].x := -70.0;
    cities[10].y := -50.0;

    beaverton.name := 'BEAVERTON';
    beaverton.x := 0.0;
    beaverton.y := 0.0;

end; { of init_cities }

{ find distance to nearest city }
procedure find_dis;
```

```
var

i, base, top : longint ;  { local index, start value,
                  end value }
xsqdis, ysqdis, dist : real ;

begin
    base := BITE * m_next;
    while (base < NCITIES) do
    begin
        top := base + BITE;
        i := base;
        while (i < top) do
        begin
            xsqdis := sqr(beaverton.x -
                cities[i].x);
            ysqdis := sqr(beaverton.y -
                cities[i].y);
            dist   := sqrt(xsqdis + ysqdis);

            m_lock;
            if (dist < shortest) then
            begin
                closest := i;
                shortest := dist;
            end;
            m_unlock;

            i := i + 1 ;
        end;
    base := BITE * m_next;
    end;
end;

begin { main program starts here }

    shortest := 999999999.0;

    init_cities;
    m_pfork(find_dis);
    writeln(cities[closest].name,
        ' is closest to Beaverton.');
    writeln(cities[closest].name, ' is ', shortest,
        ' miles from Beaverton.');

end.
```


Dynamic Shared Memory Alloc

```
/* multiply two matrices, s
   matrix, and print result

#include <stdio.h>
#include <parallel/microtas
#include <parallel/parallel

      /* Global shared memo

       shared float **a; /*
       shared float **b; /*
       shared float **c; /*

main ()
{
  char *shmalloc();
  float ** setup_matrix();
  void init_matrix(), m_for
    matmul(), print_mats();
  int size ; /* loop end va

  printf("Enter array size:
  scanf("%d",&size);

  a = setup_matrix (size, s
  b = setup_matrix (size, s
  c = setup_matrix (size, s
  init_matrix(a, b, size, s
  m_set_procs(3);
  m_fork(matmul, a, b, c, s
  m_kill_procs();
  print_mats(a, b, c, size,
}

/* initialize matrix functi

float **
setup_matrix(nrows, ncols)
int nrows, ncols;
{
int i, j;
float **new_matrix;

  /*  allocate pointer arra
      address of newly allo
```

```
Local index, start value,
;


o



:averton.x -
);
:averton.y -
);
:sqdis + ysqdis);


:est) then

i;
 dist;









 here }



.name,
rerton.');
.name, ' is ', shortest,
:on.');
```


Dynamic Shared Memory Allocation - C Example

```
/* multiply two matrices, store results in third
   matrix, and print results */

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

      /* Global shared memory data */

       shared float **a; /* first array */
       shared float **b; /* second array */
       shared float **c; /* result array */

main ()
{
  char *shmalloc();
  float ** setup_matrix();
  void init_matrix(), m_fork(), m_kill_procs(),
    matmul(), print_mats();
  int size ; /* loop end value and loop increment */

  printf("Enter array size:");
  scanf("%d",&size);

  a = setup_matrix (size, size);  /* allocate shared */
  b = setup_matrix (size, size);  /* memory */
  c = setup_matrix (size, size);
  init_matrix(a, b, size, size);  /* initialize data */
  m_set_procs(3);              /* set # of processes */
  m_fork(matmul, a, b, c, size, size); /* execute matmul */
  m_kill_procs();              /* kill childprocesses */
  print_mats(a, b, c, size, size); /* print results */
}

/* initialize matrix function */

float **
setup_matrix(nrows, ncols)
int nrows, ncols;
{
int i, j;
float **new_matrix;

  /*  allocate pointer arrays : set new_matrix to
      address of newly allocated shared matrix */
```

```
new_matrix = (float**)shmalloc((unsigned)nrows*
    (sizeof(float*)));

  /*  allocate data arrays : set first element of
      new_matrix to address of first element of
      newly allocated data array */

new_matrix[0] = (float*)shmalloc((unsigned)nrows *
                  ncols * (sizeof(float)));

  /*  initialize pointer arrays : set each element of
      new_matrix to address of corresponding element
      of data array */

for (i = 1; i < nrows; i++) {
    new_matrix[i] = new_matrix[0] + (ncols * i);
  }
return (new_matrix);
}
/* initialize matrix function */

void
init_matrix(a, b, nrows, ncols)
float **a, **b, **c;
int nrows, ncols;
{
int i, j;

    for (i = 0; i < nrows; i ++) {
        for (j = 0; j < ncols; j ++) {
            a[i][j] = (float)i + j;
            b[i][j] = (float)i - j;
        }
    }
}
void
matmul(a, b, c, nrows, ncols)
float **a, **b, **c;
int nrows, ncols;
{
int i, j, k, nprocs;

nprocs = m_get_numprocs();
    for (i = m_get_myid(); i < nrows; i += nprocs) {
        for (k = 0; k < ncols; k ++) {
            c[i][k] = 0.0;
            for (j = 0; j < ncols; j ++) {
                c[i][k] += a[i][j] * b[j][k];
```



```
            }
        }
    }
}
void
print_mats(a, b, c, nrows, r
float **a, **b, **c;
int nrows, ncols;
{
int i, j;

    for (i = 0; i < nrows; i
        for (j = 0; j < nco
            printf("a[%d][%
            i, j, a[i][j],
            printf("c[%d][%
        }
    }
}
```

5.6. Compiling, Executing

To complete development of your c steps:

1. Invoke the appropriate co your program with the Pa
2. Execute the program and
3. If necessary, use the DY to debug the program.

5.6.1 Compiling the Program

To compile and link a C program,

cc *program.c* **-lpps**

This command compiles a C source Parallel Programming Library, pr You can also include the **-g** compi information. (For more informati piler options, refer to the *Sequent*

```
)c((unsigned)nrows*


 set first element of
)f first element of
rray */

lloc((unsigned)nrows *
zeof(float)));

ays : set each element of
of corresponding element


{
ix[0] + (ncols * i);


n */


ls)



i ++) {
ls; j ++) {
t)i + j;
t)i - j;



s)



i < nrows; i += nprocs) {
ls; k ++) {

< ncols; j ++) {
 a[i][j] * b[j][k];
```



```
            }
        }
    }
}
void
print_mats(a, b, c, nrows, ncols)
float **a, **b, **c;
int nrows, ncols;
{
int i, j;

    for (i = 0; i < nrows; i ++) {
        for (j = 0; j < ncols; j ++) {
            printf("a[%d][%d] = %3.2fb[%d][%d] = %3.2f",
            i, j, a[i][j], i, j, b[i][j]);
            printf("c[%d][%d] = %3.2f\n", i, j, c[i][j]);
        }
    }
}
```

5.6. Compiling, Executing, and Debugging

To complete development of your data-partitioned program, follow these steps:

1. Invoke the appropriate compiler with the proper options to link your program with the Parallel Programming Library.
2. Execute the program and check the results.
3. If necessary, use the DYNIX parallel symbolic debugger, Pdbx, to debug the program.

5.6.1 Compiling the Program

To compile and link a C program, enter the following command:

cc *program.c* **-lpps**

This command compiles a C source file and links the object code with the Parallel Programming Library, producing an executable file named *a.out*. You can also include the **-g** compiler option to create a file of debugging information. (For more information on these options and other C compiler options, refer to the *Sequent C Compiler User's Manual*.)

NOTE

At some optimization levels, the C optimizer can remove conditional tests in spin loops. If your codes uses any spin loop on shared variables, always compile with the **-i** *compiler option to ensure that the conditional tests are preserved. For more information on the* **-i** *option, refer to* cc(1).

To compile and link a Pascal program, enter the following command:

pascal -mp *program.p*

This command compiles a Pascal source file and links the object code with the Parallel Programming Library, producing an executable file named *a.out*. It also places all global variables into shared memory. You can also include the **-g**, compiler option to create a file of debugging information. To use the Pdbx debugger on Pascal programs, you will also need to use the **-o** compiler option to give the executable file the same base name as the source file. (For more information on these options and other Pascal compiler options, refer to the *Sequent Pascal Compiler User's Manual.*)

To compile and link a FORTRAN program, enter the following command:

fortran -F/_*shcom*_**/** *program.name* **-lpps**

This command compiles a FORTRAN source file and links the object code with the Parallel Programming Library, producing an executable file named *a.out*. It also places all COMMON blocks declared with the **-F** option into shared memory. (The COMMON block names must start and end with underbars and be enclosed in slashes (/).) You can also include the **-g** or **-gv** compiler option to create a file of debugging information. To use the Pdbx debugger on FORTRAN programs, you will also need to use the **-o** compiler option to give the executable file the same base name as the source file. (For more information on these options and other FORTRAN compiler options, refer to the *Sequent FORTRAN Compiler User's Manual.*)

For more information on the DYNIX linker, refer to the *ld(1)* man page in the *DYNIX Programmer's Manual.*

5.6.2 Executing the Program

To execute the program, simply ent
a DYNIX command. The default fil

5.6.3 Debugging the Program

If your program produces incorrect r
debugger to isolate any problems. I
ic debugger. It is based on dbx, a
tems.

When using Pdbx to debug program
library, remember that by default t
exit from child processes. When tl
points, you must enter a Ctrl-Z to
execution. To disable the automati
ignore exit.

The Parallel Programming library
mine when to allocate more space f
bugger automatically stops whenev
able these automatic breakpoints,
For more information on Pdbx, refer

5.7. Additional Sources of

The following sources provide inform

- The *Sequent C Compiler U*
  Sequent C language, the co
- The *Sequent Pascal Compi*
  the Sequent Pascal langua
- The *Sequent FORTRAN C*
  detail the Sequent FORTI
  options.
- The *DYNIX Programmer*
  descriptions of the DYNIX
  tines and the DYNIX linke

TE

ls, the C optimizer can spin loops. If your codes l variables, always compile tion to ensure that the ed. For more information).

ı, enter the following command:

ource file and links the object code brary, producing an executable file bal variables into shared memory. r option to create a file of debugging er on Pascal programs, you will also to give the executable file the same more information on these options efer to the *Sequent Pascal Compiler*

program, enter the following com-

name **-lpps**

AN source file and links the object Library, producing an executable file MMON blocks declared with the **-F** COMMON block names must start osed in slashes (/).) You can also in- to create a file of debugging informa- FORTRAN programs, you will also to give the executable file the same r more information on these options ons, refer to the *Sequent FORTRAN*

X linker, refer to the *ld(1)* man page *ıl*.

5.6.2 Executing the Program

To execute the program, simply enter the name of the executable file as a DYNIX command. The default file name is *a.out*.

5.6.3 Debugging the Program

If your program produces incorrect results, you can use the DYNIX Pdbx debugger to isolate any problems. Pdbx is a high-level language symbolic debugger. It is based on dbx, a debugger widely used in UNIX systems.

When using Pdbx to debug programs that use the Parallel Programming library, remember that by default the debugger takes a breakpoint upon exit from child processes. When the debugger encounters these breakpoints, you must enter a Ctrl-Z to return control to Pdbx and continue execution. To disable the automatic breakpoint, use the Pdbx command **ignore exit**.

The Parallel Programming library uses the signal **SIGSEGV** to determine when to allocate more space for a process's shared stack. The debugger automatically stops whenever this signal is encountered. To disable these automatic breakpoints, use the command **ignore sigsegv**. For more information on Pdbx, refer to the *Sequent Pdbx User's Manual*.

5.7. Additional Sources of Information

The following sources provide information that may be helpful to you:

- The *Sequent C Compiler User's Manual* describes in detail the Sequent C language, the compiler, and its options.
- The *Sequent Pascal Compiler User's Manual* describes in detail the Sequent Pascal language, the compiler, and its options.
- The *Sequent FORTRAN Compiler User's Manual* describes in detail the Sequent FORTRAN language, the compiler, and its options.
- The *DYNIX Programmer's Manual* provides more detailed descriptions of the DYNIX Parallel Programming Library routines and the DYNIX linker, ld.

- The *Sequent Pdbx User's Manual* provides instructions for using the Pdbx debugger and reference information on the debugger command set.
- Appendices A and B discuss factors that may affect the execution speed of your program.
- Appendix D contains the DYNIX man pages for the Parallel Programming Library.
- Appendix E lists other literature on parallel programming.

Cha

Function Partitionin

Illust

Appendix C

Locking Mechanisms and Shared Memory

C.1. Introduction

This appendix provides more detail on shared memory and locking mechanisms for readers who are interested in designing their own parallel programming support packages. For more information on Sequent architecture, refer to the *Balance Technical Summary* or the *Symmetry Technical Summary*.

The DYNIX operating system allows two or more processes to share a common region of system memory. Any process with access to a shared memory region can read or write in that region in the same way that it reads or writes in ordinary memory. (The DYNIX support for shared memory is based on the interface proposed in the article "4.2bsd System Manual," a copy of which is found in Volume 2 of the *DYNIX Programmer's Manual*.)

To help ensure that one process does not modify a shared data structure while another process is using it, Sequent systems provide hardware locking mechanisms. On Sequent systems, single-byte load and store operations are always atomic (indivisible), as are 16 and 32-bit loads and stores that are aligned on natural boundaries. To ensure that any other operation is executed atomically, you must protect it with a locking routine using the Balance or Symmetry locking mechanisms.

Balance systems include a set of hardware locks (called Atomic Lock Memory) on each MULTIBUS adapter board. For Symmetry systems, locking is handled by special System Bus and cache protocol. Access to both shared memory and ALM is controlled by the `mmap()` system call. (See *mmap* (2) for a detailed specification of the `mmap()` system call.) The locking mechanism in the Symmetry system is invoked with a special prefix to certain Symmetry assembly language instructions.

C.1.1 Balance Systems: Atomic Lock Memory

Mapping Atomic Lock Memory

By default, the only Multibus physical addresses directly accessible to user programs are those associated with ALM. (The superuser can make additional regions of the physical address space, such as those associated with special hardware devices, available using the pmap utility; see *pmap* (4) and *pmap* (8).)

Each MULTIBUS adapter board is assigned 1 Mbyte near the top of the System Bus (physical) address space. Each MULTIBUS adapter's address range is subdivided into several regions, including a 64-Kbyte region for ALM. The 32 2-Kbyte regions of ALM on the first MULTIBUS adapter board are accessed through the special files *alm00* through *alm31* in the */dev/alm* directory. To gain access to an ALM region, a process opens the corresponding file to connect to the `pmap` device driver, then maps it into its virtual address space by using the `mmap()` system call. Then the process can simply read or write the ALM address space.

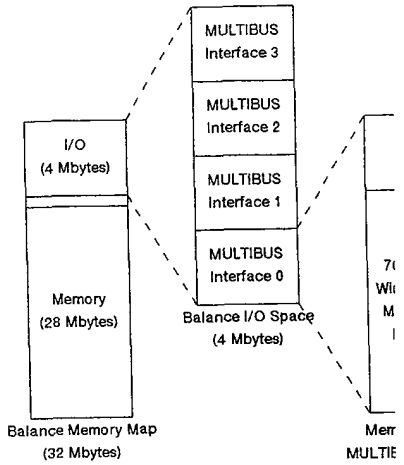


Fig. C-1. ALM in the S

Each 32-bit double-word in the ALI
of 16K locks per MULTIBUS ada
any lock contains useful informatio
through byte operations on doubl
operation causes the system to sen

Lock Operations: Test-and-Set

A lock's least significant bit deter
(0). Reading a lock returns the st
automatically to 1, thereby locking
or *atomic*. Writing a 0 to a lock lo

*On reads from the ALM
undefined; they must be r
from bytes other than th
lock but don't necessarily
Similarly, writes to b*

nd Shared Memory

ck Memory

:al addresses directly accessible to with ALM. (The superuser can ical address space, such as those ces, available using the pmap util-

signed 1 Mbyte near the top of the ice. Each MULTIBUS adapter's eral regions, including a 64-Kbyte gions of ALM on the first MUL- through the special files *alm00* ctory. To gain access to an ALM ing file to connect to the `pmap` dev- rtual address space by using the ess can simply read or write the

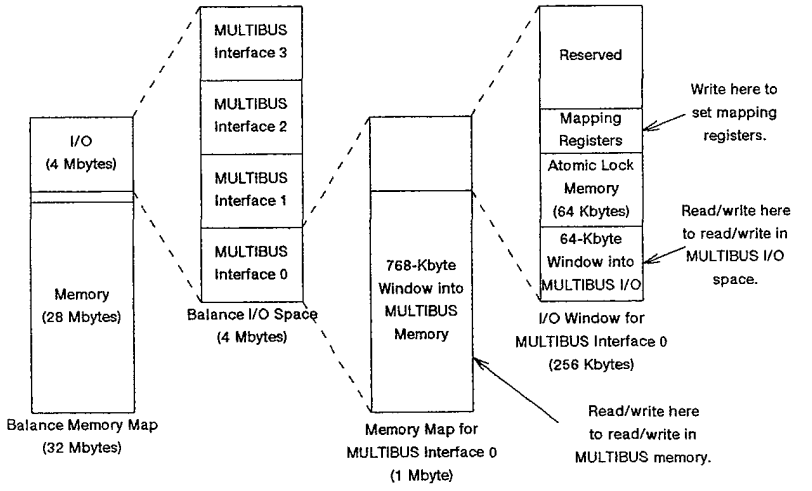


Fig. C-1. ALM in the System Bus address space.

Each 32-bit double-word in the ALM represents one lock, yielding a total of 16K locks per MULTIBUS adapter. Only the least-significant bit of any lock contains useful information. Software must access this bit only through byte operations on double-word boundaries. Any other type of operation causes the system to send a SIGBUS signal to the process.

Lock Operations: Test-and-Set and Clear

A lock's least significant bit determines its state: locked (1) or unlocked (0). Reading a lock returns the state of this bit (0 or 1) and then sets it automatically to 1, thereby locking the lock. This operation is indivisible, or *atomic*. Writing a 0 to a lock location unlocks the lock.

NOTE

On reads from the ALM, bits other than bit 0 are undefined; they must be masked off in software. Reads from bytes other than the least-significant byte set the lock but don't necessarily return the correct lock state. Similarly, writes to bytes other than the least-

significant byte may randomly affect the lock state. Accesses that cross a 32-bit boundary affect two locks simultaneously.

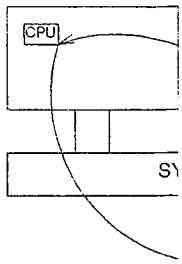
Simple Lock and Unlock Routines

The following code sample illustrates simple routines for locking and unlocking a lock in ALM. The `lock()` routine simply loops until another process clears the lock to 0. The routine can return at this point, because the hardware relocks the lock (sets it to 1) after reading the 0.

```
/*
 * Lock the ALM lock whose address is lockp.
 */
lock (lockp)
        char    *lockp;
{
        while (*lockp & 1)
                continue;
}

/*
 * Unlock the ALM lock whose address is lockp.
 */
unlock (lockp)
        char    *lockp;
{
        *lockp = 0;
}
```

This implementation works correctly, except that it may place an unnecessary burden on the System Bus. If the ALM lock is locked when the `lock()` routine is called, `lock()` repeatedly attempts to read the ALM lock, using System Bus cycles in the process, until the lock is unlocked by another process. (See Figure C-2.) Since accesses to the ALM consume bus bandwidth and compete with accesses to MULTIBUS peripherals, heavy use of this `lock()` routine may degrade system performance.



MULTIBUS ADAPTER B(

Fig. C-2. Spinning on

Eliminating Unnecessary Bus

An alternative approach is to sp copy of the lock in shared me cached by the dual-processor bo from the shadow variable, the bl variable is stored in the pro satisfied by the cache until the the shadow variable (i.e., unloc sees the write occur, it invalidat dow variable, and the next r memory. (See Figure C-3.)

ıly affect the lock state.
ıoundary affect two locks

: simple routines for locking and
routine simply loops until another
outine can return at this point,
(sets it to 1) after reading the 0.

```
se address is lockp.
```

```
1)
;
```

```
hose address is lockp.
```

r, except that it may place an
s. If the ALM lock is locked when
repeatedly attempts to read the
in the process, until the lock is
igure C-2.) Since accesses to the
ıpete with accesses to MULTIBUS
routine may degrade system per-

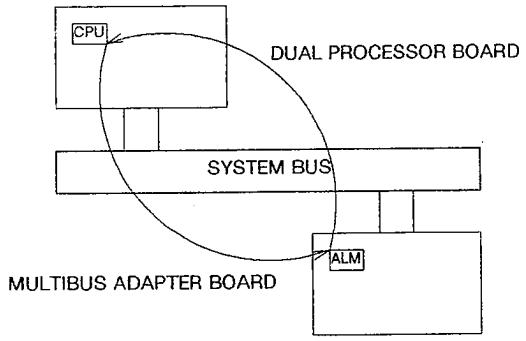


Fig. C-2. Spinning on ALM lock uses System Bus.

Eliminating Unnecessary Bus Usage

An alternative approach is to spin on a *shadow* of the ALM lock—i.e., a copy of the lock in shared memory. Reads from system memory are cached by the dual-processor board. The first time the processor reads from the shadow variable, the block of memory that contains the shadow variable is stored in the processor's cache. Subsequent reads are satisfied by the cache until the processor holding the lock writes a 0 to the shadow variable (i.e., unlocks the lock). When the cache controller sees the write occur, it invalidates the cache block that contains the shadow variable, and the next read returns the new value (0) out of memory. (See Figure C-3.)

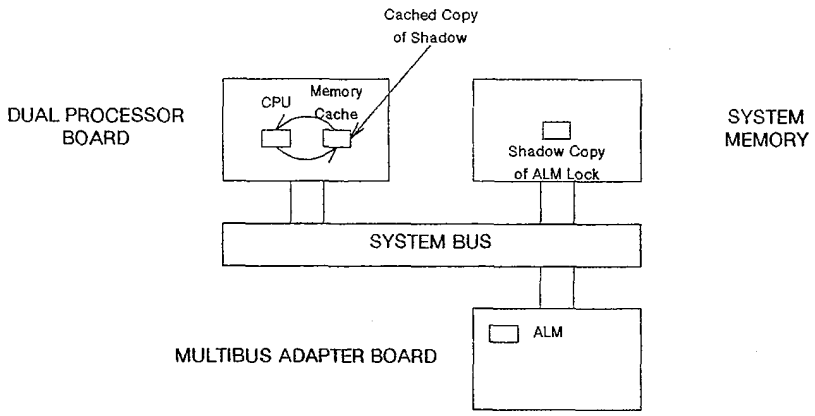


Fig. C-3. Spinning on shadow of lock uses cache.

The following code illustrates `lock()` and `unlock()` routines using this technique:

```
struct lock_t {
  char    *lk_alm;        /* address of ALM lock */
  char    lk_shadow;      /* shadow in memory */
};

/*
 * Lock the ALM lock whose address is lockp.
 */
lock (lockp)
        register struct lock_t   *lockp;
{
        /* Go for the ALM lock. */
        while ( *(lockp->lk_alm) & 1) {
                /*
                 * Didn't get it.  Spin until shadow
                 * is unlocked and try again.
                 */
                while (lockp->lk_shadow)
                        continue;
        }
```



```
        /* Got the ALM lo
        lockp->lk_shadow
}

/*
 * Unlock the ALM lock wh
 */
unlock (lockp)
        struct lock_t   *
{
        lockp->lk_shadow
        *(lockp->lk_alm)
}
```

Multiplexed Locks

Some applications may require more hardware. To solve this problem, you guard multiple "soft" locks. Each soft value of 1 (locked) or 0 (unlocked). unlock a soft lock or to spin waiting for before locking a soft lock, you must ol lock to ensure that no other process is time. Since the hardware lock is held being changed to the locked state, th negligible.

The following code illustrates `lock()` plexed locks:

```
typedef unsigned char

#define L_UNLOCKED   0
#define L_LOCKED 1

/*
 * ALM_HASH() is used to
 */

extern   char*_alm_base;

#define ALM_HASH(x)  ((i
#define ALM_UNLOCKED 0
#define ALM_LOCKED   1


/*
```

```
        /* Got the ALM lock.  Lock the shadow. */
        lockp->lk_shadow = 1;
}

/*
 * Unlock the ALM lock whose address is lockp.
 */
unlock (lockp)
        struct lock_t   *lockp;
{
        lockp->lk_shadow = 0;
        *(lockp->lk_alm) = 0;
}
```

Multiplexed Locks

Some applications may require more locks than are available in the hardware. To solve this problem, you can use a single hardware lock to guard multiple "soft" locks. Each soft lock is a byte in memory with a value of 1 (locked) or 0 (unlocked). No hardware lock is required to unlock a soft lock or to spin waiting for it to become unlocked. However, before locking a soft lock, you must obtain the corresponding hardware lock to ensure that no other process is locking the soft lock at the same time. Since the hardware lock is held only while one of its soft locks is being changed to the locked state, the effect on System Bus traffic is negligible.

The following code illustrates `lock()` and `unlock()` routines for multiplexed locks:

```
typedef unsigned char     slock_t; /* 's' for "spin"-lock */

#define  L_UNLOCKED   0
#define  L_LOCKED 1

/*
 * ALM_HASH() is used to hash an address to an ALM offset.
 */

extern   char *_alm_base;       /* virt addr of mapped ALM's */

#define  ALM_HASH(x)  ((int)(&(x)) & (0xFF << 2))
#define  ALM_UNLOCKED 0
#define  ALM_LOCKED   1


/*
```

```
 * lock() provides in-line access to locks for C programs;
 */

#define lock(lp) { \
    register char    *lock_alm = &_alm_base[ALM_HASH(*(lp))];
    for (;;) { \
        /* Wait for lock to be available */ \
        while (*(lp) == L_LOCKED) \
            continue; \
        /* Grab ALM gate for atomic access to lock */ \
        while (*lock_alm & ALM_LOCKED) \
            continue; \
        /* Can race with others trying to get the lock */ \
        if (*(lp) == L_UNLOCKED) { \
            /* No race (or won it) -- grab the lock */ \
            *(lp) = L_LOCKED; \
            *lock_alm = ALM_UNLOCKED; \
            break; \
        } \
        /* Lost race, try again */ \
        *lock_alm = ALM_UNLOCKED; \
    } \
}

/*
 * unlock() provides in-line unlocking for C programs;
 */
#define unlock(lp)   (*(lp) = L_UNLOCKED)
```

C.1.2 Symmetry Systems: Locked Instructions

The Symmetry locking mechanism is basically the same as the Balance locking mechanism: bytes of memory are used as locks. The difference is that Symmetry systems do not require processes to map ALM regions. Instead, any byte of memory may be used as a lock.

The LOCK Prefix

On Symmetry systems, locking is handled by the System Bus hardware. Locking mechanisms are therefore implemented in Symmetry assembly language. These can be included in C programs as `asm` functions. (For information on `asm` functions, refer to the *Sequent C Compiler User's Manual*.) They can also be implemented as out-of-line locking subroutines such as `s_lock` and `s_unlock`.

To set the bus lock, precede a
prefix. This prefix assures th
prefixes. The `LOCK` prefix can
instructions for 8, 16, and 32
`XCHG`, `ADD`, `OR`, `ADC`, `SBB`, `ANI`
to the *Symmmetry Series Asse*
information on these instruction

The `XCHG` *instruction*
preceded by the `LOCK` *p*

Simple Lock and Unlock Rou

Symmetry locking and unlocl
instruction to perform atomic te
The following example shows on

```
asm void LOCK(locka
{
%reg lockadd; lab l

loop: movb  $LOCK,
      xchgb %dl, (l

      cmpb  $UNLOCF

      je    done
spin: cmpb  $UNLOCF
      je    loop
      jmp   spin
done:
}
```

Notice that because this rout
atomic test-and-set operation,
also that if the routine's first a
spins in cache while waiting fc
traffic on the System Bus.

```
cess to locks for C programs;


m = &_alm_base[ALM_HASH(*(lp))];

>e available */ \
:KED) \

 atomic access to lock */ \
.M_LOCKED) \

:rs trying to get the lock */ \
:ED) { \
>n it) -- grab the lock */ \
; \
JNLOCKED; \


ıin */ \
:KED; \



unlocking for C programs;

 L_UNLOCKED)
```

.ctions

lly the same as the Balance
ed as locks. The difference is
esses to map ALM regions.
. a lock.

y the System Bus hardware.
nted in Symmetry assembly
ams as `asm` functions. (For
Sequent C Compiler User's
s out-of-line locking subrou-

To set the bus lock, precede an assembly instruction with the LOCK prefix. This prefix assures the atomicity of the instruction that it prefixes. The LOCK prefix can be used with the following assembler instructions for 8, 16, and 32-bit operations: BT, BTS, BTR, BTC, XCHG, ADD, OR, ADC, SBB, AND, SUB, XOR NOT, NEG, and INC. (Refer to the *Symmmetry Series Assembler User's Manual* for more detailed information on these instructions.)

NOTE

The XCHG *instruction is always locked, whether it is preceded by the* LOCK *prefix or not.*

Simple Lock and Unlock Routines

Symmetry locking and unlocking routines typically use the XCHG instruction to perform atomic test-and-set and test-and-clear operations. The following example shows one implementation of a locking routine:

```
asm void LOCK(lockadd)
{
%reg lockadd; lab loop, spin, done;

loop: movb  $LOCK, %dl          /* lock byte to register */
      xchgb %dl, (lockadd)      /* atomic test-and-set   */
                                /* on "soft" lock in mem */
      cmpb  $UNLOCK, %dl        /* if mem location was   */
                                /* unlocked, we got lock */
      je    done                /* we're finished        */
spin: cmpb  $UNLOCK, (lockadd) /* spin in cache until   */
      je    loop                /* unlocked, then try    */
      jmp   spin                /* again for lock        */
done:
}
```

Notice that because this routine uses the XCHG instruction for the atomic test-and-set operation, it does not need the LOCK prefix. Notice also that if the routine's first attempt to set the lock is unsuccessful, it spins in cache while waiting for the lock and does not create additional traffic on the System Bus.

The following example shows one implementation of an unlocking routine:

```
asm void UNLOCK(lockadd)
{
%reg lockadd;

        movb  $UNLOCK, %al   /* unlock byte to register */
        xchgb %al, (lockadd) /* atomic test-and-clear   */
}
```

Again, notice that this routine uses the XCHG instruction for the atomic test-and-set operation, so it does not need the LOCK prefix. When the address of the lock is sent out on the System Bus, any processor spinning in cache and waiting for a lock will see the address on the bus and try again to set the lock.

C.1.3 Shared Memory

The *mmap* (2) entry in Volume 1 of the *DYNIX Programmer's Manual* is a detailed specification of the `mmap()` system call, upon which the DYNIX shared-memory implementation is based. The following paragraphs examine certain features of `mmap()` that may be of interest to a programmer writing a parallel programming support package.

Mapping Shared Memory

In general, `mmap()` can be used to map a portion of any file or any region of the system's physical address space into a process's virtual address space. A process creates a shared-memory region by opening an ordinary file, then using `mmap()` to map the file into the process's virtual address space. If the high end of the mapped region is above the current program "break" (as returned by the `sbrk()` system call), the "break" is set to the high end of the mapped region. However, any memory between the old break and the low end of the mapped region is inaccessible (unless it is subsequently `mmap`-ed).

A shared-memory allocator analogous to `malloc()` (see *malloc* (3)) can be built using `mmap()` to acquire needed memory in the same way that `malloc()` uses `sbrk()`. In fact, the Parallel Programming Library routines `shmalloc()`, `shbrk()`, and `shsbrk()` use `mmap()` in this way.

Mapped regions created with `mmap()` are inherited (i.e., shared) by the process's children. Thus, in an application involving a parent process and one or more identical (not exec-ed) children, the parent first maps

the necessary shared-memory an
locks or other shared variables, t
Programming Library handles ini
Balance systems, ALM by calling
a program's `main()` routine. Th
data segment into shared memor
sary, and performs miscellaneous
routines.) Unrelated processes car
mapping the same file into their vi

Note, however, that `mmap()` affe
subsequently forked children. I
memory region, the expansion will
If B tries to access a variable set
dress space, B will receive a **SIGS**
course, B can catch this signal an
to grow its own shared-memory re
used by the Parallel Programming
memory regions up to date.

Mapped Files

The Parallel Programming Librar
file that it uses to create the shar
many ways to use the file that is

- The file acts like a pagin
The memory contents are
is swapped or when it exi
mapped by the last proce
can be useful in post-mort
- If the mapped portion of
mapped, the contents of
memory." (Technically, t
needed.) Thus, a prev
memory can be easily res
- An application-specific mc
ecuting parallel applic
mapped file into its own a
- `Read()` and `write()` op
file also affect the corres
ties such as **cp** can be u
memory.

mentation of an unlocking rou-

```
/* unlock byte to register */
/* atomic test-and-clear   */
```

XCHG instruction for the atomic
ed the LOCK prefix. When the
ystem Bus, any processor spin-
see the address on the bus and

'YNIX *Programmer's Manual* is
system call, upon which the
is based. The following para-
() that may be of interest to a
ing support package.

p a portion of any file or any
space into a process's virtual
d-memory region by opening an
the file into the process's virtual
pped region is above the current
k() system call), the "break" is
egion. However, any memory
the mapped region is inaccessi-

`malloc()` (see *malloc*(3)) can
memory in the same way that
Parallel Programming Library
`shsbrk()` use `mmap()` in this

e inherited (i.e., shared) by the
tion involving a parent process
children, the parent first maps

the necessary shared-memory and ALM regions, then initializes any locks or other shared variables, then forks the children. (The Parallel Programming Library handles initialization of shared memory and, for Balance systems, ALM by calling the `_ppinit()` routine before calling a program's `main()` routine. This routine maps the program's shared data segment into shared memory, allocates a block of ALM if necessary, and performs miscellaneous run-time initilization for other library routines.) Unrelated processes can also share memory by independently mapping the same file into their virtual memory .

Note, however, that `mmap()` affects only the calling process and any **subsequently** forked children. If child process A expands its shared-memory region, the expansion will not show up in its sibling process, B. If B tries to access a variable set up by A in the new portion of A's address space, B will receive a **SIGSEGV** (segmentation fault) signal. Of course, B can catch this signal and use it as an indication that B needs to grow its own shared-memory region to match A's. This mechanism is used by the Parallel Programming Library to keep all processes' shared-memory regions up to date.

Mapped Files

The Parallel Programming Library immediately unlinks the temporary file that it uses to create the shared memory region. However, there are many ways to use the file that is mapped into a shared memory region:

- The file acts like a paging area for the mapped memory region. The memory contents are copied out to the file when the process is swapped or when it exits, or when the region is otherwise unmapped by the last process that has it mapped. Thus, the file can be useful in post-mortems.
- If the mapped portion of the file already exists when the file is mapped, the contents of the file are immediately available "in memory." (Technically, the contents are paged in as they are needed.) Thus, a previously obtained snapshot of shared memory can be easily restored.
- An application-specific monitor or debugger can plug in to an executing parallel application by mapping the application's mapped file into its own address space.
- `Read()` and `write()` operations to the mapped regions of the file also affect the corresponding memory. Thus, ordinary utilities such as **cp** can be used to capture the contents of shared memory.

Note, however, that a file cannot be truncated while it is mapped. Thus,

```
cp saved_mem mapped_file
```

will not work.

Also note that if you map a file whose size is not an integral multiple of the file system block size (usually 8192), `mmap()` will pad the file with null bytes to the end of the block. If you do not have write access to the file, `mmap()` will fail.

Mapping Shared Memory from Unrelated Processes

The following pages contain examples showing how to use the **mmap** system call to create shared memory for unrelated processes. The examples illustrate two techniques. The first, and simplest, technique is to create a single shared file and to use the DYNIX loader, **ld**, to locate the shared data in memory. The second technique is to create multiple shared files and use assembler directives to locate the shared data.

Creating a Single Shared File. Creating a single shared file is a two-part process:

1. Set up a `_ppinit` subprogram to call **mmap** and initialize shared files. This procedure is automatically called before the main program.
2. Use **ld** to declare the necessary global variable or common block as shared and to declare its location in memory.

The following examples illustrate this process.

NOTE

These examples do not use a full pathname for the shared file, so they must be executed in the same directory.

The following two FORTRAN programs declare the common block `SHARED` and then take turns writing values to the shared file. The first program, *x1.f*, waits for the other program to write the shared variable `A`, writes the shared variable `B`, waits for the other program to write `C`, then exits.


```
      COMMON /SHAR
      INTEGER*4 A,

      WRITE(0,1)
    1 FORMAT( 12H

   10 CONTINUE
      IF ( A .EQ.

      WRITE(0,2)
    2 FORMAT( 9H W

      B = 1

      WRITE(0,3)
    3 FORMAT( 12H

   20 CONTINUE
      IF ( C .EQ.

      STOP
      END
```

The second program, *x2.f*, writ other program to write the sha C, then exits.

```
      COMMON /SHAI
      INTEGER*4 A,

      WRITE(0,1)
    1 FORMAT( 9H W

      A = 1

      WRITE(0,2)
    2 FORMAT( 12H

   10 CONTINUE
      IF ( B .EQ.

      WRITE(0,3)
    3 FORMAT( 9H W

      C = 1

      STOP
      END
```

```
      COMMON /SHARED/ A,B,C
      INTEGER*4 A,B,C

      WRITE(0,1)
    1 FORMAT( 12H WAIT FOR A )

   10 CONTINUE
      IF ( A .EQ. 0 ) GOTO 10

      WRITE(0,2)
    2 FORMAT( 9H WRITE B )

      B = 1

      WRITE(0,3)
    3 FORMAT( 12H WAIT FOR C )

   20 CONTINUE
      IF ( C .EQ. 0 ) GOTO 20

      STOP
      END
```

The second program, *x2.f*, writes the shared variable A, waits for the other program to write the shared variable B, writes the shared variable C, then exits.

```
      COMMON /SHARED/ A,B,C
      INTEGER*4 A,B,C

      WRITE(0,1)
    1 FORMAT( 9H WRITE A )

      A = 1

      WRITE(0,2)
    2 FORMAT( 12H WAIT FOR B )

   10 CONTINUE
      IF ( B .EQ. 0 ) GOTO 10

      WRITE(0,3)
    3 FORMAT( 9H WRITE C )

      C = 1

      STOP
      END
```

The following file, *ppinit.c*, is linked with both FORTRAN programs and is called automatically when the programs are run. This subprogram initializes the shared file and rounds the size of the shared memory segment up to the nearest page boundary:

```
/*
 * _ppinit.c
 *  Parallel program run-time
 *  environment initialization.
 */

#include <a.out.h>
#include <strings.h>
#include <sys/errno.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/mman.h>

#include <machine/pmap.h>
#include "parc.h"

/*
 * _ppinit()
 *  Parallel startup for C programs.
 *
 */

extern int errno;
int _pgoff;
extern shared char _shstart_, _shend_;


_ppinit()
{
    int fd;
    int szshared;

    fd = open("SHARED_FILE", O_RDWR|O_CREAT, 0666);
    if (fd < 0)
        bad_init("open", errno);

    _pgoff = getpagesize() - 1;

    szshared = (int) PGRND(&_shend_ - &_shstart_);
    if (MMAP(fd, &_shstart_, szshared, 0) < 0)
        bad_init("mmap", errno);
```



```
}


/*
 * bad_init()
 *  For some reason, cc
 *  complain and exit w
 */

static
bad_init(msg, err)
    char *msg;
    int err;
{
    perror(msg);
    _exit(err);
}
```

The following header file, *parc.h*, de used in *finit.c*.

```
/*
 * parc.h
 *  Parallel C support
 */

/*
 * MMAP() is short-hand
 */

#define MMAP(fd,va,sz,
    mmap(va, sz, PROT_
/*
 * PGRND() rounds up a
 */

#define PGRND(x) (char
```

Finally, the following file, *Makefile*, various sections of this application. lines use the loader option **-F** to de the loader option **-ZO** to declare 100 data segment.

```
all : x1 x2
x1  : x1.f ppinit.o
      fortran -F/SHARE

x2  : x2.f ppinit.o
```

```
}


/*
 * bad_init()
 *   For some reason, couldn't init --
 *   complain and exit with error status.
 */

static
bad_init(msg, err)
    char*msg;
    int err;
{
    perror(msg);
    _exit(err);
}
```

The following header file, *parc.h*, defines the `MMAP` and `PGRND` macros used in *finit.c*.

```
/*
 * parc.h
 *   Parallel C support library definitions.
 */

/*
 * MMAP() is short-hand for calling mmap().
 */

#define MMAP(fd,va,sz,pos)   \
    mmap(va, sz, PROT_RDWR, MAP_SHARED, fd, pos)
/*
 * PGRND() rounds up a value to next page boundary.
 */

#define PGRND(x)(char *) (((int)(x) + _pgoff) & ~_pgoff)
```

Finally, the following file, *Makefile*, compiles, links, and executes the various sections of this application. Notice that the **fortran** command lines use the loader option **-F** to declare the shared common block and the loader option **-ZO** to declare 10000 as the base address of the shared data segment.

```
all : x1 x2
x1  : x1.f ppinit.o
      fortran -F/SHARED/ -ZO10000 -e -o x1 x1.f finit.o

x2  : x2.f ppinit.o
```

```
        fortran -F/SHARED/ -Z010000 -e -o x2 x2.f finit.o

ppinit.o: ppinit.c

clean    :
    rm -f x1 x2 *.o SHARED_FILE

run :
    rm -f SHARED_FILE
    x1 &
    sleep 5
    x2 &
```

Creating Multiple Shared Files. Creating multiple shared files is a three-part process:

1. Set up your main programs to explicitly call a subprogram that initializes shared memory.
2. Set up the subprogram to call **mmap** and initialize shared files.
3. Set up a file of assembler directives that define the starting address of each shared file.

The following examples illustrate this process. (Some of these examples are similar or identical to those in the previous section.)

NOTE

These examples do not use a full pathname for the shared file, so they must be executed in the same directory.

The following two FORTRAN programs declare the common block SHARED, call the subroutine FINIT to initialize shared memory, and then take turns writing values to the shared file. The first program, *x1.f*, waits for the other program to write the shared variable A, writes the shared variable B, waits for the other program to write C, then exits.


```
      COMMON /SHARED/
      INTEGER*4 _START

      EXTERNAL _FINIT

      CALL _FINIT(_STA

      WRITE(0,1)
    1 FORMAT( 12H WAIT

   10 CONTINUE
      IF ( A .EQ. 0 )

      WRITE(0,2)
    2 FORMAT( 9H WRITE

      B = 1

      WRITE(0,3)
    3 FORMAT( 12H WAIT

   20 CONTINUE
      IF ( C .EQ. 0 )

      STOP
      END
```

The second program, *x2.f*, initializes then writes the shared variable A, the shared variable B, writes the sh

```
      COMMON /SHARED/
      INTEGER*4 _START

      EXTERNAL _FINIT

      CALL _FINIT(_ST

      WRITE(0,1)
    1 FORMAT( 9H WRITE

      A = 1

      WRITE(0,2)
    2 FORMAT( 12H WAI

   10 CONTINUE
      IF ( B .EQ. 0 )
```

```
-Z010000 -e -o x2 x2.f finit.o
```

```
D_FILE
```

eating multiple shared files is a

explicitly call a subprogram that

mmap and initialize shared files.

tives that define the starting ad-

rocess. (Some of these examples revious section.)

full pathname for the
cuted in the same direc-

ıms declare the common block) initialize shared memory, and shared file. The first program, ite the shared variable A, writes ıer program to write C, then ex-


```
      COMMON /SHARED/ _START,A,B,C,_END
      INTEGER*4 _START,A,B,C,_END

      EXTERNAL _FINIT

      CALL _FINIT(_START,_END)

      WRITE(0,1)
    1 FORMAT( 12H WAIT FOR A )

   10 CONTINUE
      IF ( A .EQ. 0 ) GOTO 10

      WRITE(0,2)
    2 FORMAT( 9H WRITE B )

      B = 1

      WRITE(0,3)
    3 FORMAT( 12H WAIT FOR C )

   20 CONTINUE
      IF ( C .EQ. 0 ) GOTO 20

      STOP
      END
```

The second program, *x2.f*, initializes itself in the same way as *x1.f*. It then writes the shared variable A, waits for the other program to write the shared variable B, writes the shared variable C, and exits.

```
      COMMON /SHARED/ _START,A,B,C,_END
      INTEGER*4 _START,A,B,C,_END

      EXTERNAL _FINIT

      CALL _FINIT(_START, _END)

      WRITE(0,1)
    1 FORMAT( 9H WRITE A )

      A = 1

      WRITE(0,2)
    2 FORMAT( 12H WAIT FOR B )

   10 CONTINUE
      IF ( B .EQ. 0 ) GOTO 10
```

```
      WRITE(0,3)
    3 FORMAT( 9H WRITE C )

      C = 1

      STOP
      END
```

The following file, *finit.c*, initializes the shared file and rounds the size of the shared memory segment up to the nearest page boundary:

```
/*
 * finit.c
 *   Parallel program run-time
 *   environment initialization.
 */

#include <a.out.h>
#include <strings.h>
#include <sys/errno.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/mman.h>

#include <machine/pmap.h>
#include "parc.h"

/*
 * finit()
 *   Parallel startup for C programs.
 *
 */

extern int errno;
int _pgoff;

finit(end, start)
    char *start, *end;
{
    int fd;
    int szshared;

    printf("start %x, end %x0", start, end);
    fd = open("SHARED_FILE", O_RDWR|O_CREAT, 0666);
    if (fd < 0)
        bad_init("open", errno);

    _pgoff = getpagesize() - 1;
```



```
    szshared = (int) P
    if (MMAP(fd, start
        bad_init("mmap
}
/*
 * bad_init()
 *   For some reason, c
 *   complain and exit
 */

static
bad_init(msg, err)
    char*msg;
    int err;
{
    perror(msg);
    _exit(err);
}
```

The following header file, *parc.h*, d
used in *finit.c*.

```
/*
 * parc.h
 *   Parallel C support
 */

/*
 * MMAP() is short-han
 */

#define MMAP(fd,va,sz,
    mmap(va, sz, PROT_
/*
 * PGRND() rounds up a
 */
#define PGRND(x) (char
```

The following assembly language fi
SHARED common block:

```
    .globl   /SHARED/
    .set /SHARED/,0x1(
```

)

hared file and rounds the size of
:arest page boundary:

:ime
ıtion.

: programs.

%x0, start, end);
:", O_RDWR|O_CREAT, 0666);

:rrno);

- 1;


```
        szshared = (int) PGRND(end - start);
        if (MMAP(fd, start, szshared, 0) < 0)
            bad_init("mmap", errno);
    }
    /*
     * bad_init()
     *   For some reason, couldn't init --
     *   complain and exit with error status.
     */

    static
    bad_init(msg, err)
        char *msg;
        int err;
    {
        perror(msg);
        _exit(err);
    }
```

The following header file, *parc.h*, defines the `MMAP` and `PGRND` macros used in *finit.c*.

```
    /*
     * parc.h
     *   Parallel C support library definitions.
     */

    /*
     * MMAP() is short-hand for calling mmap().
     */

    #define  MMAP(fd,va,sz,pos)   \
        mmap(va, sz, PROT_RDWR, MAP_SHARED, fd, pos)
    /*
     * PGRND() rounds up a value to next page boundary.
     */
    #define  PGRND(x) (char *) (((int)(x) + _pgoff) & ~_pgoff)
```

The following assembly language file, *x.s*, sets the base address of the SHARED common block:

```
        .globl    /SHARED/
        .set /SHARED/,0x100000
```

Finally, the following file, *Makefile*, compiles, links, and executes the various sections of this application.

```
all     : x1 x2
x1      : x1.f x.o finit.o
            fortran -e -o x1 x1.f x.o finit.o

x2      : x2.f x.o finit.o
            fortran -e -o x2 x2.f x.o finit.o

x.o     : x.s

finit.o     : finit.c

clean       :
        rm -f x1 x2 *.o SHARED_FILE

run     :
        rm -f SHARED_FILE
        x1 & x2 &
```

C.2. Balance Configuration Requirements for ALM

For a program that uses ALM to run on your Balance system, the following conditions must be true. The associated configuration steps must be performed by the superuser.

NOTE

There are no special configuration requirements for Symmetry Systems, since they do not use ALM.

1. The `pmap` pseudo-device driver must be configured into the DYNIX kernel. Verify that your kernel configuration file (e.g., */sys/conf/DYNIX*) contains this line:

   ```
   pseudo-device   pmap            # phys-map driver
   ```

   If this line is not present, you need to add it to the end of your kernel configuration file and rebuild the kernel, as described in the *DYNIX System Administrator's Guide* .

2. The special files *alm0(* */dev/alm* directory. If t commands at the systen

   ```
   # cd /dev
   # MAKEDEV alm
   ```

3. The revision number of be 2:1 or greater: earlie execute the **MAKEDEV** not include ALM, **MAKI**

   ```
   OLD REV MBAD, NO AI
   DEVICES
   ```

   If your MULTIBUS ad than 2:1, contact your grade.

4. If the MULTIBUS ada TIBUS interface board adapter board only for it must be properly jumpe

Shared Memory

npiles, links, and executes the

```
x1 x1.f x.o finit.o

x2 x2.f x.o finit.o
```

IARED_FILE

quirements for ALM

ı your Balance system, the fol-
ociated configuration steps must

ıtion requirements for
o not use ALM.

· must be configured into the
ır kernel configuration file (e.g.,
line:

```
p          # phys-map driver
```

eed to add it to the end of your
ıuild the kernel, as described in
ır's Guide .

2. The special files *alm00* through *alm31* must reside in the */dev/alm* directory. If this directory does not exist, enter these commands at the system prompt:

```
# cd /dev
# MAKEDEV alm
```

3. The revision number of your MULTIBUS adapter board must be 2:1 or greater: earlier revisions do not contain ALM. If you execute the **MAKEDEV** `alm` command and your system does not include ALM, **MAKEDEV** will respond as follows:

```
OLD REV MBAD, NO ALM SUPPORT -- CAN'T INSTALL ALM
DEVICES
```

   If your MULTIBUS adapter board has a revision number less than 2:1, contact your local sales representative about an upgrade.

4. If the MULTIBUS adapter board is not connected to a MULTIBUS interface board (e.g., you are using the MULTIBUS adapter board only for its ALM), the MULTIBUS adapter board must be properly jumpered for this configuration.