Chapter 3

Models of Parallel Computation

Silicon Graphics, makes multiprocessor computer systems. You can use any of several programming models to exploit the parallel capabilities of the hardware. This chapter reviews the parallel programming models, supplying enough information that you can select one model. Pointers to more detailed documentation of each model are included. The major topics are:

- "Parallel Hardware and Programming Models" on page 123 provides a quick survey of the programming models and their relationship to the hardware.
- "Using Statement-Level Parallelism" on page 132 discusses using fine-grained parallel execution in Fortran and C.
- "Using Process-Level Parallelism" on page 137 provides an overview of the use of coordinated UNIX processes for parallel execution.
- "Using MPI and PVM" on page 142 compares these two interfaces.

Parallel Hardware and Programming Models

Silicon Graphics makes a variety of multiprocessor systems, including

- The CHALLENGE/Onyx systems (and their POWER versions) are symmetric multiprocessor (SMP) computers. In these systems at least 2, and as many as 36, identical microprocessors access a single, common memory and a common set of peripherals through a high-speed bus.
- The POWER CHALLENGEarray™ comprises 2 or more POWER CHALLENGE™ systems connected by a high-speed local HIPPI network. Each node in the array is an SMP with 2 to 36 CPUs. Nodes do not share a common memory; communication between programs in different nodes passes through sockets. However, the entire array can be administered and programmed as a single entity.

Most programs that run on these systems execute as if they were in a uniprocessor, employing the facilities of a single CPU. The IRIX operating system applies CPUs to different programs in order to maximize system throughput.

However, it is possible to write a program so that it makes use of more than one CPU at a time. The software interface you use for this is the parallel programming model. Each model is designed around a different set of assumptions about the hardware, and in particular about the memory system available.

Parallel Programs on Uniprocessors

It might seem a contradiction, but it is possible to execute some parallel programs in uniprocessors. Obviously you would not do this expecting the best performance. However:

- It is possible to restrict and assign the available CPUs of a multiprocessor so that only one CPU is available to execute a given program, even a program intended to use parallel execution. This can arise as a brief transient condition as IRIX dynamically assigns CPUs to programs, or it can arise through the operator using commands such as *mpadmin* (see the mpadmin(1) reference page).
- It is easier to debug a parallel program by running it in the more predictable environment of a uniprocessor.

Most parallel programming models adapt to the available hardware, running concurrently on multiple CPUs (up to some programmer-defined limit) when the CPUs are available, but running on a limited number, or even just one CPU, when necessary. For example, the Fortran programmer can control the number of CPUs used by a MIPpro Fortran 77 program by setting environment variables before the program starts (see “Using Statement-Level Parallelism” on page 132).

Memory Systems

The key memory issue is this: Can one process access memory data belonging to another concurrent process, and if so, what is the time penalty for doing so? The answer depends on the hardware architecture, and determines the optimal programming model.

Single Memory Systems

The CHALLENGE/Onyx system architecture uses a very high speed system bus to connect all components of the system.

One component is the physical memory system, which plugs into the bus and is equally available to all other components. Other units that plug into the system bus are I/O adapters, such as the VME bus adapter. CPU modules containing two MIPS R4000, R8000, or R10000 CPUs are also plugged into the system bus.

In the CHALLENGE/Onyx architecture, the single, common memory has these features:

- There is a single address map; that is, the same word of memory has the same address in every CPU.
- There is no time penalty for communication between processes because every word is accessible in the same amount of time from any CPU.
- All peripherals are equally accessible from any process.

The effect of a single, common memory is that processes running in different CPUs can share pages of memory, and can update the identical memory locations concurrently. For example, suppose there are four CPUs available to a Fortran program that processes a large array of data. You can divide a single DO-loop so that it executes concurrently on the four CPUs, each CPU working in one-fourth of the array in memory.

As another example, IRIX allows processes to “map” a single segment of memory into the virtual address spaces of two or more concurrent processes. Two processes can transfer data at memory speeds, one putting the data into a mapped segment and the other process taking the data out. They can coordinate their access to the data using semaphores located in the shared segment.

Multiple Memory Systems

In an Array system such as a POWERCHALLENGEarray, each node is a computer built on the CHALLENGE/Onyx architecture. However, the only connection between nodes is the high-speed HIPPI bus between nodes. The system does not offer a single system memory; instead, there is a separate memory subsystem in each node. As a result:

- There is not a single address map. A word of memory in one node cannot be addressed at all from another node.
- There is a time penalty for some interprocess communication. When data passes between programs in different nodes, it passes through a software socket and over the HIPPI network, which takes longer than a memory-to-memory transfer.
- Peripherals are accessible only in the node to which they are physically attached.

Nevertheless it is possible to design an application that executes concurrently in multiple nodes of an Array. The message-passing interface (MPI) is designed specifically for this.

Types of Parallel Models

The IRIX system supports a variety of parallel programming models. You can compare these models on two features:

Granularity	The relative size of the units of computation that are affected: single statements, functions, or entire processes.
Communication channel	The basic mechanism by which the independent, concurrent units of the program exchange data and synchronize their activity.

A summary comparison of the available models is shown in Table 3-1.

Table 3-1 Comparing Parallel Models

Model	Granularity	Communication
Power Fortran™, IRIS POWER C™	Looping statement (DO or *for* statement)	Shared variables in a single user address space.
Ada95 tasks	Ada Procedure	Shared variables in a single user address space.
Lightweight UNIX processes (**sproc()**)	C function	Arena memory segment in a single user address space.
General UNIX processes (**fork()**, **exec()**)	Process	Arena segment mapped to multiple address spaces.
Remote Procedure Call (RPC)	Process	Memory copy within node or UDP or TCP network between nodes.
Portable Virtual Memory (PVM)	Process	Memory copy within node or TCP socket between nodes.
Message-Passing (MPI)	Process	Memory copy within node or TCP socket between nodes.

Statement-Level Parallelism

Parallelism at the finest level of granularity is provided for three languages:

- MIPSpro Fortran 77 supports compiler directives that command parallel execution of the bodies of DO-loops. The MIPSpro POWER Fortran 77 product is a preprocessor that automates the insertion of these directives in a serial program.
- MIPSpro Fortran 90 supports parallelizing directives similar to MIPSpro Fortran 77, and the MIPSpro POWER Fortran 90 product automates their placement.
- MIPSpro POWER C supports compiler pragmas that command parallel execution of segments of code. The IRIS POWER C analyzer automates the insertion of these pragmas in a serial program.

In all three languages, the run-time library—which provides the execution environment for the compiled program—contains support for parallel execution. The compiler generates library calls that create subprocesses and distribute loop iterations to them.

The run-time support can adapt itself dynamically to the number of available CPUs. Alternatively, you can control it—using program source statements or using environment variables at execution time—to use a certain number of CPUs.

Statement-level parallel support is based on using common variables in memory, and so it can be used only within the bounds of a single-memory system, a CHALLENGE or a single node in a POWERCHALLENGEarray.

Thread-Level Parallelism

A thread is an independent execution state within the context of a larger program. A UNIX process normally consists of an address space and one thread, together with a large collection of state information: a table of open files, a set of signal handlers, a process ID, an effective user ID, and so on.

There are three key differences between a thread and a process:

- A UNIX process has its own set of UNIX state information, for example, its own effective user ID, signal handlers, and set of open file descriptors.

  Threads exist within a process and do not have distinct copies of these UNIX state values. Threads share the single state belonging to their process.

- Normally, each UNIX process has a unique address space of memory segments that are accessible only to that process (lightweight processes created with **sproc()** share an address space; see “Process-Level Parallelism” on page 129).

  Threads within a process share the single address space belonging to their process.

- Processes are scheduled by the IRIX kernel. A change of process requires two context changes, into the kernel domain and back to the user domain of the next process. Since a process carries a large amount of state information, the change from the context of one process to the context of another can entail many instructions.

  In contrast, threads are scheduled by code that operates almost entirely in the user domain without kernel interference. Since threads have less state information, thread scheduling is faster than process scheduling.

At this time, IRIX supports only one thread per process. However, Silicon Graphics has announced the intention of supporting the POSIX standard for multithreaded applications in a future release.

In the meantime, the Silicon Graphics implementation of the Ada 95 language includes support for multitasking Ada programs—using what are essentially threads in the meaning used here. For a complete discussion of the Ada 95 task facility, refer to the *Ada 95 Reference Manual*, which installs with the Ada 95 compiler (GNAT) product.

Process-Level Parallelism

A UNIX process consists of an address space, a varied set of state values, and one thread of execution. The main task of the IRIX kernel is to create processes and to dispatch them to different CPUs so as to maximize the utilization of the system.

IRIX contains a variety of interprocess communication (IPC) mechanisms, which are discussed in Chapter 2, "Interprocess Communication." These mechanisms can be used to exchange data and to coordinate the activities of multiple, asynchronous processes within a single-memory system. (Processes running in different nodes of an array must use one of the abstract models described in the next topic.)

In traditional UNIX practice, one process creates another with the system call **fork()**, which makes a duplicate of the calling process, after which the two copies execute concurrently. Typically the new process immediately uses the **exec()** function to load a new program.

The fork(2) reference page contains a complete list of the state values that are duplicated when a process is created. The exec(2) reference page details the process of creating a new program image for execution.

IRIX also supports the system function **sproc()**, which creates a lightweight process. A process created with **sproc()** shares some of its state values with its parent process (the sproc(2) reference page details how this sharing is specified).

In particular, a lightweight process does not have its own address space; it continues to execute in the address space of the original process. In this respect, a lightweight process is like a thread (see "Thread-Level Parallelism" on page 128). However, a lightweight process differs from a true thread in two significant ways:

- A lightweight process still has a full set of UNIX state values, including its own signal handlers. Some of these values, for example the table of open file descriptors, can be shared with the parent process, but in general a lightweight process carries more state information than a thread.
- Dispatch of lightweight processes is done in the kernel, and a context switch between lightweight processes, even when they share the same address space, is time-consuming.

The library support for statement-level parallelism is based on the use of lightweight processes, coordinating their activities through semaphores (see "Statement-Level Parallelism" on page 127 and "Using IRIX Semaphores" on page 45).

Portable, Abstract Models

There are three portable, abstract models of parallel execution that are supported by Silicon Graphics systems. Each provides a method of distributing a computation within a single-memory system or across the nodes of a multiple-memory system, without having to reflect the system configuration in the source code. The three programming models are:

- Message-Passing Interface (MPI)
- Portable Virtual Memory (PVM)
- Remote Procedure Call (RPC) interface

Each of the three has its particular strengths and weaknesses.

Message-Passing Interface (MPI) Model

MPI is a portable standard programming interface for the construction of a portable, parallel application in Fortran 77 or in C, especially when the application can be decomposed into a fixed number of processes operating in a fixed topology (for example, a pipeline, grid, or tree).

A highly tuned, efficient implementation of MPI is included with the Array software CD for Array systems such as the POWER CHALLENGEarray. MPI is the recommended parallel model for use with Array products.

MPI is discussed in more detail under "Using MPI and PVM" on page 142.

Portable Virtual Machine (PVM) Model

PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous, concurrent computing framework on interconnected computers of varied architecture. Using PVM, you can create a parallel application that executes as a set of concurrent processes on a set of computers. The set can include Silicon Graphics uniprocessors, multiprocessors, and nodes of Array systems.

An implementation of PVM is included with the Array software CD for Silicon Graphics Array systems. PVM has a better ability to deal with a heterogenous computer network than MPI does. In every other way, MPI is preferable. When the application runs in the context of a single Array system, an MPI design has better performance.

PVM is discussed in more detail under "Using MPI and PVM" on page 142.

Remote Procedure Call (RPC) Model

RPC is a standard programming interface originally developed at Sun Microsystems, Inc. and used as the basis of Sun's Network File System (NFS) standard. RPC is used extensively within the IRIX system (and in most current UNIX implementations) to provide NFS and network management services.

The purpose of the RPC interface is to distribute services across a network, so that one program can easily supply a service to all others. An RPC server program registers the services it can provide with RPC. A client program anywhere in the network can issue a remote procedure call for a registered service, and the RPC interface takes care of locating the server program, invoking its service, and returning the result values to the caller.

RPC by itself does not support concurrent execution. A remote procedure call, like a local procedure call, is synchronous; that is, the caller is blocked until the called procedure completes its work. RPC is a method of distributing a computation over a network, not a method of parallel execution. However, RPC can be combined with other parallel execution models. For example, a thread or lightweight process can issue remote procedure calls.

RPC libraries are included in IRIX. For an overview of RPC programming, see the *IRIX Network Programming Guide*. For further details, refer to the rpc(3R) reference page.

Using Statement-Level Parallelism

As noted under "Statement-Level Parallelism" on page 127, you can use statement-level parallelism in three language packages: Fortran 77, Fortran 90, and C. This type of parallelism is unique in that you begin with a normal, serial program, and you can always return the program to serial execution by recompiling. Every other parallel model requires you to plan and write a parallel program from the start.

The parallel features of all three of these languages are documented in detail in the manuals listed in Table 3-2.

Table 3-2 Documentation for Statement-Level Parallel Products

Manual	Document Number	Contents
IRIS POWER C User's Guide	007-0702-0x0	Use of the IRIS POWER C Analyzer, including all pragmas.
MIPSpro Fortran 77 Programmer's Guide	007-2361-00x	General use of Fortran 77, including parallelizing assertions and directives.
MIPSpro Power Fortran 77 Programmer's Guide	007-2363-00x	Use of the Power Fortran source analyzer to place directives automatically.
MIPSpro Fortran 90 Programmer's Guide	007-2761-001	General use of Fortran 90, including parallelizing assertions and directives.
MIPSpro Power Fortran 90 Programmer's Guide	007-2760-001	Use of the Power Fortran 90 source analyzer to place directives automatically.

In addition to these products from Silicon Graphics, the High Performance Fortran (HPF) compiler from the Portland Group is a compiler for Fortran 90 augmented to the HPF standard. It supports automatic parallelization. (Refer to http://www.pgroup.com for more information).

The FORGE products from Applied Parallel Research (APRI) contain a Fortran 77 source analyzer that can insert parallelizing directives, although not the directives supported by MIPSpro Fortran 77. (Refer to http://www.infomall.org/apri for more information.)

Creating Parallel Programs

In each of the three languages, the language compiler supports explicit statements that command parallel execution (#pragma lines for C; directives and assertions for Fortran). However, placing these statements is a demanding, error-prone task. It is easy to create a suboptimal program, or worse, a program that is incorrect in subtle ways. Furthermore, small changes in program logic can invalidate parallel directives in ways that are hard to foresee, so it is difficult to maintain a program that has been manually made parallel.

For each language, there is a source-level program analyzer that is sold as a separate product (IRIS POWER C, MIPSpro Power Fortran 77, MIPSpro Power Fortran 90). The analyzer identifies sections of the program that can safely be executed in parallel, and automatically inserts the parallelizing directives. After any logic change, you can run the analysis again, so that maintenance is easier.

The source analyzer makes conservative assumptions about the way the program uses data. As a result, it often is unable to find all the potential parallelism. However, the analyzer produces a detailed listing of the program source, showing each segment that could or could not be parallelized, and why. Directed by this listing, you insert source assertions that give the analyzer more information about the program.

The method of creating an optimized parallel program is as follows:

1. Write a complete application that runs on a single processor.
2. Completely debug and verify the correctness of the program in serial execution.
3. Apply the source analyzer and study the listing it produces.
4. Add assertions to the source program. These are high-level statements that describe the program's use of data, not explicit commands to parallelize.
5. Repeat steps 3 and 4 until the analyzer finds as much parallelism as possible.
6. Run the program on a single-memory multiprocessor.

When the program requires maintenance, you make the necessary logic changes and, simultaneously, you remove any assertions about the changed code—unless you are certain that the assertions are still true of the modified logic. Then repeat the preceding procedure from step 2.

Managing Parallel Execution

The run-time library for each of the languages uses IRIX lightweight processes to implement parallel execution (see "Process-Level Parallelism" on page 129).

When a parallel program starts, the run-time support creates a pool of lightweight processes using the **sproc()** function. Initially the extra processes are blocked, and one process executes the opening passage of the program. When execution reaches a parallel section, the run-time support unblocks as many processes as necessary. Each one begins to execute the same block of statements. The processes share global variables, while each has its own copy of variables that are local to one iteration of a loop, such as a loop index.

When a process completes its portion of the work of that section, it returns to the run-time library code, where it picks up another portion of work if any work remains, or simply blocks until the next time it is needed. At the end of the parallel section, all extra processes are blocked and the original process continues to execute the serial code following the parallel section.

Controlling the Degree of Parallelism

You can specify the number of lightweight processes that are started by a program. In IRIS POWER C, you can use *#pragma numthreads* to specify the exact number of processes to start, but it is not a good idea to embed this number in a source program. In all implementations, the run-time library by default starts enough processes that there is one for each CPU in the system. That default is often too high, since usually at least one of the CPUs is dedicated to other work (often more than one).

The run-time library checks an environment variable, MPC_SET_NUM_THREADS, for the number of processes to start. You can use this environment variable to choose the number of processes used by a particular run of the program, thereby tuning the program's requirements to the system load. You can even force a parallelized program to execute on a single CPU when necessary.

MIPSpro Fortran 77 and MIPSpro Fortran 90 also recognize additional environment variables that specify a range of process numbers, and use more or fewer processes within this range as system load varies. (See the *Programmer's Guide* for the language for details.)

At certain points the multiple processes must wait for one another before continuing. They do this by waiting in a busy-loop for a certain length of time, then by blocking until they are signaled. You can specify the amount of time that a process should spend spinning before it blocks, using either source directives or an environment variable (see the *Programmer's Guide* for the language for system functions for this purpose).

Choosing the Loop Schedule Type

Most parallel sections are loops. The benefit of parallelization is that some iterations of the loop are executed in one CPU, concurrent with other iterations of the same loop in other CPUs. But how are the different iterations distributed across processes? All three languages support four possible methods of scheduling loop iterations, as summarized in Table 3-3. The variables used in Table 3-3 are as follows:

N Number of iterations in the loop, determined from the source or at run-time.

P Number of available processes, set by default or by environment variable (see "Controlling the Degree of Parallelism" on page 134).

Q Number of a process, from 0 to *N*-1.

C "Chunk" size, set by directive or by environment variable.

Table 3-3 Loop Scheduling Types

Schedule	Purpose
SIMPLE	Each process executes $\lfloor N/P \rfloor$ iterations starting at $Q*(\lfloor N/P \rfloor)$. First process to finish takes the remainder chunk, if any.
DYNAMIC	Each process executes *C* iterations of the loop, starting with the next undone chunk unit, returning for another chunk until none are left undone.
INTERLEAVE	Each process executes *C* iterations at *C***Q*, *C**2*Q*, *C**3*Q*...
GSS	Each process executes chunks of decreasing size, (*N*/2*P*), (*N*/4*P*), ...

The effects of the scheduling types depend on the nature of the loops being parallelized. For example:

- The SIMPLE method works well when *N* is relatively small. However, unless *N* is evenly divided by *P*, there will be a time at the end of the loop when fewer than *P* processes are working, and possibly only one.
- The DYNAMIC and INTERLEAVE methods allow you to set the chunk size so as to control the span of an array referenced by each process. You can use this to reduce cache effects. When *N* is very large so that not all data fits in memory, INTERLEAVE may reduce the amount of paging compared to DYNAMIC.
- The guided self-scheduling (GSS) method is good for triangular matrices and other algorithms where loop iterations become faster toward the end.

You can use source directives or pragmas within the program to specify the scheduling type and chunk size for particular loops. Where you do not specify the scheduling, the run-time library uses a default method and chunk size. You can establish this default scheduling type and chunk size using environment variables.

Using Process-Level Parallelism

Software products from Silicon Graphics use process-level parallelism in order to exploit the power of single-memory multiprocessors. For example, the IRIS Performer graphics library normally creates a separate lightweight process to manage the graphics pipe in parallel with rendering work. The run-time library for statement-level parallelism creates a pool of lightweight processes and dispatches them to execute parts of loop code in parallel (see "Managing Parallel Execution" on page 134).

Parallelism in Real-Time Applications

In real-time programs such as aircraft or vehicle simulators, separate processes are used to distribute the work of the simulation onto multiple CPUs. In these demanding applications, the programmer frequently uses IRIX facilities to

- reserve one or more CPUs of a multiprocessor for exclusive use by the application
- isolate the reserved CPUs from all interrupts
- assign specific processes to execute on specific, reserved CPUs

These facilities are described in detail in the *REACT Real-Time Programmer's Guide* (007-2499-00x). Also covered in that book is the use of the Frame Scheduler, an alternate process scheduler. The normal process scheduling algorithm of the IRIX kernel attempts to keep all CPUs busy and to keep all processes advancing in a fair manner. This algorithm is in conflict with the stringent needs of a real-time program, which needs to dedicate predictable amounts of hardware capacity to its processes, without regard to fairness.

The Frame Scheduler seizes one or more CPUs of a multiprocessor, isolates them, and executes a specified set of processes on each CPU in strict rotation. The Frame Scheduler has much lower overhead than the normal IRIX scheduler; and it has features designed for real-time work, including detection of overrun (when a scheduled process does not complete its work in the necessary time) and underrun (when a scheduled process fails to execute in its turn).

At this writing there are no real-time applications that use multiple nodes of an Array system.

Process Synchronization and Share Groups

IRIX provides a variety of features to make it possible to build an application consisting of multiple, lightweight processes. In general, a lightweight process is one that shares the address space of its parent process (see "Process-Level Parallelism" on page 129). The parent process and the sibling processes that it creates are a *share group*. IRIX provides special services to share groups.

Process Communication and Coordination

IRIX supports a wide range of interprocess communication (IPC) facilities. These are discussed in detail in Chapter 2, "Interprocess Communication." They include:

- The use of shared arenas for common memory (see "Initializing a Shared Arena" on page 36 and the following topics).
- IRIX semaphores ("Using IRIX Semaphores" on page 45), locks ("Using Locks" on page 47) and barriers ("Using Barriers" on page 49) for coordination and mutual exclusion.

  The IRIX semaphores and locks are especially tuned to efficiency in a multiprocessor environment.
- Portable support for interprocess messages, shared memory, and semaphores ("System V IPC Overview" on page 51).

The REACT™/Pro product includes a number of examples of real-time programs that use IRIX IPC features. The *REACT Real-Time Programmer's Guide* includes the source code of additional examples.

Process Creation

The **sproc()** and **sprocsp()** functions create a lightweight process (see the sproc(2) reference page). The difference between the calls is that **sproc()** allocates a new memory segment to serve as the stack for the new process. You use **sprocsp()** to specify a stack segment that you have already allocated—for example, a block of memory that you allocate and lock against paging using **mpin()**.

In the traditional **fork()** call, the new process executes the identical program text as the old one; that is, both processes "return" from **fork()** and you distinguish them by the return code, which is 0 in the child process and the new process ID in the parent.

The **sproc()** call differs in that it takes as an argument the address of the function that should be executed by the new process. Often, each child process has a particular role to play, and the function that represents that work.

Another design is possible. The **sproc()** function has considerable overhead. It is inefficient to continually create and destroy child processes. In some applications, you may have to manage a flow of many, relatively short, activities which should be done in parallel. You do not want to create a new child process for each activity and destroy it afterward. Instead, you can create a pool containing a small number of general-purpose processes. When a piece of work needs to be done, you can dispatch one process to do it. The fragmentary code in Example 3-1 shows the general approach.

Example 3-1 Partial Code to Manage a Pool of Processes

```
typedef void (*func)(void *arg) workFunc;
struct oneSproc {
   struct oneSproc *next;        /* -> next oneSproc ready to run */
   workFunc calledFunc;          /* -> function sproc is to call */
   void *callArg;                /* argument to pass to called func */
   usema_t *sprocDone;           /* optional sema to post on completion */
   usema_t *sprocWait;           /* sproc waits for work here */
} sprocList[NUMSPROCS];
usema_t *readySprocs;            /* count represents sprocs ready to work */
uslock_t sprocListLock;          /* mutex control of sprocList head */
struct oneSproc *sprocList;      /* -> first ready oneSproc */
/*
|| Put a oneSproc structure on the ready list and sleep on it.
|| Called by a child process when its work is done.
*/
void sprocSleep(struct oneSproc *theSproc)
{
    ussetlock(sprocListLock);       /* acquire exclusive rights to sprocList */
    theSproc->next = sprocList;  /* put self on the list */
    sprocList = theSproc;
    usunsetlock(sprocListLock);  /* release sprocList */
    usvsema(readySprocs);           /* notify master, at least 1 on the list */
    uspsema(theSproc->sprocWait);/* sleep until master posts me */
}
/*
|| Body of a general-purpose child process. The argument, which must
|| be declared void* to match the sproc() prototype, is the oneSproc
|| structure that represents this process.   The contents of that
|| struct, in particular sprocWait, are initialized by the parent.
*/
```

```
void childBody(void *theSprocAsVoid)
{
    struct oneSproc *mySproc = (struct oneSproc *)theSprocAsVoid;
    /* here one could establish signal handlers, etc. */
    for(;;)
    {
        sprocSleep(mySproc);      /* wait for work to do */
        mySproc->calledFunc(mySproc->callArg);  /* do the work */
        if (mySproc->sprocDone)   /* if a completion sema is given, */
            usvsema(mySproc->sprocDone); /* ..post it */
    }
}
/*
|| Acquire a oneSproc structure from the ready list, waiting if necessary.
|| Called by the master process as part of dispatching a sproc.
*/
struct oneSproc *getSproc()
{
    struct oneSproc *theSproc;
    uspsema(readySprocs);        /* wait until at least 1 sproc is free */
    ussetlock(sprocListLock);       /* acquire exclusive rights to sprocList */
    theSproc = sprocList;        /* get address of first free oneSproc */
    sprocList = theSproc->next;  /* make next in list, the head of list */
    usunsetlock(sprocListLock);  /* release sprocList */
    return theSproc;
}
/*
|| Start a function going asynchronously. Called by master process.
*/
void execFunc(workFunc toCall, void *callWith, usema_t *done)
{
    struct oneSproc *theSproc = getSproc();
    theSproc->calledFunc = toCall;     /* set address of func to exec */
    theSproc->callArg = callWith;      /* set argument to pass */
    theSproc->sprocDone = done;           /* set sema to post on completion */
    usvsema(theSproc->sprocWait);      /* wake up sleeping process */
}
```

Process Scheduling Features

The IRIX kernel supports special process scheduling rules for share groups. This permits you to increase the efficiency of a parallel program in some cases. The feature is controlled by the **schedctl()** kernel function (detailed in the schedctl(2) reference page).

When **schedctl()** is called with the SCHEDMODE argument, it sets one of three scheduling rules for the share group whose member issues the call:

SGS_FREE	The normal situation, in which each process is scheduled individually.
SGS_SINGLE	All but the master process of the share group are blocked. This permits the master process to perform initialization or error recovery without contention from other members of the group.
SGS_GANG	All processes of the group run concurrently, provided there are sufficient CPUs available.

Under gang scheduling, IRIX tries to run all processes of a share group concurrently. When this is possible (in other words, when there are enough available CPUs in the multiprocessor), gang scheduling can greatly reduce lock conflicts between processes.

Without gang scheduling, one member of the share group can acquire a lock and then be suspended. Another member, attempting to acquire the lock, is also suspended until the first process is dispatched again and releases the lock.

With gang scheduling, when a second member attempts to acquire the lock, the first process is almost certainly executing at the same time, and releases the lock while the second member is still spinning.

Process Management Features

The **prctl()** kernel function provides a variety of process-related management tools (detailed in the prctl(2) reference page). One feature useful for parallel programs is the PR_MAXPPROCS query. This returns the number of different CPUs that the calling process could use for execution. The returned number is 1 when the caller has been assigned to a particular CPU. Otherwise it is the number of unrestricted CPUs in the system. A parent process could use this during initialization to find out the degree of parallelism it can hope to achieve.

The **sysmp()** kernel function provides information about a multiprocessor (detailed in the sysmp(2) reference page). Some of the queries useful to a parallel program include MP_NPROCS, return number of CPUs in the system, and MP_NAPROCS, return the number of CPUs available for normal process scheduling.

Using MPI and PVM

MPI (see "Message-Passing Interface (MPI) Model" on page 130) and PVM (see "Portable Virtual Machine (PVM) Model" on page 131) are two approaches to the same problem: how to distribute a concurrent program across a cluster of computers.

Choosing Between MPI and PVM

Silicon Graphics has adopted the MPI interface as the primary and preferred model for concurrent applications on Array processors. There are occasions when you may elect to use PVM instead, but in general MPI is strongly recommended for new applications and for applications that are being ported to an Array system.

In many ways, MPI and PVM are similar:

- Each is designed, specified, and implemented by third parties who have no direct interest in selling hardware.
- Support for each is available "on the net" at low or no cost.
- Each provides a set of portable, high-level, functions with which a group of processes can make contact and exchange data without having to be aware of the communication medium.

- Each supports C and Fortran 77.
- Each provides for automatic conversion between different representations of the same kind of data so that processes can be distributed over a hetergeneous computer network.

The primary reason MPI is preferred for Array systems is performance. The design of MPI is such that a highly optimized implementation could be created for the homogenous environment of Silicon Graphics Array systems. PVM implementations for Array systems do not perform as well for reasons that are rooted in the PVM interface design.

Another difference is in the support for the "topology" (the interconnect pattern: grid, torus, or tree) of the communicating processes. In MPI, the group size and topology are fixed when the group is created. This permits low-overhead group operations and, because the topology is normally inherent in the algorithmic design, the lack of run-time flexibility is not a problem. In PVM, group composition is dynamic, which requires the use of a "group server" process and causes more overhead in common group-related operations.

Other reasons can be found in the design details of the two interfaces. MPI, for example, supports asynchronous and multiple message traffic, so that a process can wait for any of a list of message-receive calls to complete, and can initiate concurrent sending and receiving. MPI provides for a "context" qualifier as part of the "envelope" of each message. This permits you to build encapsulated libraries that exchange data independently of the data exchanged by the client modules. MPI also provides several elegant data-exchange functions for use by a program that is emulating an SPMD parallel architecture.

PVM is possibly more suitable for distributing a program across a heterogenous network, including both uniprocessors and multiprocessors and including computers from multiple vendors. When the application runs in the environment of a Silicon Graphics Array system, MPI is the recommended interface.

Porting From PVM to MPI

Because MPI and PVM address similar problems in ways that are conceptually similar, you can consider porting a program from PVM to MPI in order to get better performance on an Array system. A detailed discussion of this process, with examples, appears in Appendix B, "Converting PVM Applications to MPI."