Chapter 11. Multiprocessing Advanced Features

A number of features are provided so that you can override the multiprocessing defaults and customize the parallelism to your particular applications. The following sections provide brief explanations of these features.

11.1 Run−time Library Routines

The Silicon Graphics multiprocessing C and C++ compiler provides the following routines for customizing your program.

11.1.1 mp_block and mp_unblock

The `mp_block` routine puts the slave threads into a blocked state using the `blockproc` system call. The slave threads stay blocked until a call is made to the `mp_unblock` routine. These routines are useful if the job has bursts of parallelism separated by long stretches of single processing, as with an interactive program. You can block the slave processes so they consume CPU cycles only as needed, thus freeing the machine for other users. The system automatically unblocks the slaves on entering a parallel region if you neglect to do so.

11.1.2 mp_setup, mp_create, and mp_destroy

The `mp_setup`, `mp_create`, and `mp_destroy` subroutine calls create and destroy threads of execution. This can be useful if the job has only one parallel portion or if the parallel parts are widely scattered. When you destroy the extra execution threads, they cannot consume system resources; they must be recreated when needed. Use of these routines is discouraged because they degrade performance; the `mp_block` and `mp_unblock` routines should be used in almost all cases.

`mp_setup` takes no arguments. It creates the default number of processes as defined by previous calls to `mp_set_numthreads`, by the `MP_SET_NUMTHREADS` environment variable, or by the number of CPUs on the current hardware platform. `mp_setup` is called automatically when the first parallel loop is entered to initialize the slave threads.

`mp_create` takes a single integer argument, the total number of execution threads desired. Note that the total number of threads includes the master thread. Thus, `mp_create(`*n*`)` creates one thread less than the value of its argument. `mp_destroy` takes no arguments; it destroys all the slave execution threads, leaving the master untouched.

When the slave threads die, they generate a `SIGCLD` signal. If your program has changed the signal handler to catch `SIGCLD`, it must be prepared to deal with this signal when `mp_destroy` is executed. This signal also occurs when the program exits; `mp_destroy` is called as part of normal cleanup when a parallel job terminates.

11.1.3 mp_blocktime

The slave threads spin wait until there is work to do. This makes them immediately available when a parallel region is reached. However, this consumes CPU resources. After enough wait time has passed, the slaves block themselves through `blockproc`. Once the slaves are blocked, it requires a system call to `unblockproc` to activate the slaves again (refer to the `unblockproc(2)` man page for details). This makes the response time much longer when starting up a parallel region.

This trade−off between response time and CPU usage can be adjusted with the `mp_blocktime` call. The `mp_blocktime` routine takes a single integer argument that specifies the number of times to spin before blocking. By default, it is set to 10,000,000; this takes roughly one second. If called with an argument of 0, the slave threads will not block themselves no matter how much time has passed. Explicit calls to `mp_block`, however, will still block the threads.

This automatic blocking is transparent to the user's program; blocked threads are automatically unblocked when a parallel region is reached.

11.1.4 mp_numthreads, mp_suggested_numthreads, mp_set_numthreads

Occasionally, you may want to know how many execution threads are available. The `mp_numthreads` routine is a zero−argument integer function that returns the total number of execution threads for this job. The count includes the master thread. In addition, this routine has the side effect of freezing (for eternity) the number of threads to the returned value, so this routine should be used sparingly. To determine the number of threads without this freeze property, use `mp_suggested_numthreads`.

`mp_suggested_numthreads` takes an unsigned integer and uses the supplied value as a hint about how many threads to use in subsequent parallel regions. It returns the previous value of the number of threads to be employed in parallel regions. It does not affect currently executing parallel regions, if any. The implementation may ignore this hint depending on factors such as overall system load. This routine may also be called with the value 0, in which case it simply returns the number of threads to be employed in parallel regions.

`mp_set_numthreads` takes a single integer argument. It changes the default number of threads to the specified value. A subsequent call to `mp_setup` will use the specified value rather than the original defaults. If the slave threads have already been created, this call will not change their number. It has an effect only when `mp_setup` is called.

11.1.5 mp_my_threadnum

The `mp_my_threadnum` routine is a zero−argument function that allows a thread to differentiate itself while in a parallel region. If there are *n* execution threads, the function call returns a value between zero and $n - 1$. The master thread is always thread zero. This function can be useful when parallelizing certain kinds of loops. Most of the time the loop index variable can be used for the same purpose. Occasionally, the loop index may not be accessible, as, for example, when an external routine is called from within the parallel loop. This routine provides a mechanism for those cases.

11.1.6 mp_setlock, mp_unsetlock, mp_barrier

The `mp_setlock`, `mp_unsetlock`, and `mp_barrier` zero−argument subroutines provide convenient (although limited) access to the locking and barrier functions provided by `ussetlock`, `usunsetlock`, and `barrier`. These subroutines are convenient because you do not need to initialize them; calls such as `usconfig` and `usinit` are done automatically. The limitation is that there is only one lock and one barrier. For most programs, this amount is sufficient. If your program requires more complex or flexible locking facilities, use the `ussetlock` family of subroutines directly.

11.1.7 mp_set_slave_stacksize

The `mp_set_slave_stacksize` routine sets the stack size (in bytes) to be used by the slave processes when they are created (using `sprocsp`). The default size is 16 MB. Slave processes only allocate their local data onto their stack, shared data (even if allocated on the master's stack) is not counted.

11.2 Run−time Environment Variables

The Silicon Graphics multiprocessing C and C++ compiler provides the following environment variables that you can use to customize your program.

11.2.1 MP_SET_NUMTHREADS, MP_BLOCKTIME, MP_SETUP

The MP_SET_NUMTHREADS, MP_BLOCKTIME, and MP_SETUP environment variables act as an implicit call to the corresponding routine(s) of the same name at program start−up time.

For example, the following csh command causes the program to create two threads regardless of the number of CPUs actually on the machine, as does the source statement below it:

csh command:

```
% setenv MP_SET_NUMTHREADS 2
```

Source statement:

```
mp_set_numthreads (2)
```

Similarly, the following sh commands prevent the slave threads from autoblocking, as does the source statement:

sh commands:

```
% set MP_BLOCKTIME 0  
% export MP_BLOCKTIME
```

Source statement:

```
mp_blocktime (0);
```

For compatibility with older releases, the environment variable NUM_THREADS is supported as a synonym for MP_SET_NUMTHREADS.

To help support networks with several multiprocessors and several CPUs, the environment variable MP_SET_NUMTHREADS also accepts an expression involving integers +, −, min, max, and the special symbol "all," which stands for the number of CPUs on the current machine. For example, the following command selects the number of threads to be two fewer than the total number of CPUs (but always at least one):

```
% setenv MP_SET_NUMTHREADS max(1,all-2)
```

11.2.2 *MP_SUGNUMTHD, MP_SUGNUMTHD_MIN, MP_SUGNUMTHD_MAX, MP_SUGNUMTHD_VERBOSE*

In an environment with long running jobs and varying workloads, it may be preferable to vary the number of threads during execution of some jobs.

Setting MP_SUGNUMTHD causes the run−time library to create an additional, asynchronous process that periodically wakes up and monitors the system load. When idle processors exist, this process increases the number of threads, up to a maximum of MP_SET_NUMTHREADS. When the system load increases, it decreases the number of threads, possibly to as few as 1. When MP_SUGNUMTHD has no value, this feature is disabled and multithreading works as before.

The environment variables MP_SUGNUMTHD_MIN and MP_SUGNUMTHD_MAX are used to limit this feature as desired. When MP_SUGNUMTHD_MIN is set to an integer value between 1 and MP_SET_NUMTHREADS, the process will not decrease the number of threads below that value.

When MP_SUGNUMTHD_MAX is set to an integer value between the minimum number of threads and MP_SET_NUMTHREADS, the process will not increase the number of threads above that value.

If you set any value in the environment variable MP_SUGNUMTHD_VERBOSE, informational messages are written to stderr whenever the process changes the number of threads in use.

Calls to mp_numthreads and mp_set_numthreads are taken as a sign that the application depends on the number of threads in use. The number in use is frozen upon either of these calls; and if MP_SUGNUMTHD_VERBOSE is set, a message to that effect is written to stderr.

11.2.3 *MP_SCHEDTYPE, CHUNK*

These environment variables specify the type of scheduling to use on for loops that have their scheduling type set to RUNTIME. For example, the following csh commands cause loops with the RUNTIME scheduling type to be executed as interleaved loops with a chunk size of 4:

```
% setenv MP_SCHEDTYPE INTERLEAVE  
% setenv CHUNK 4
```

The defaults are the same as on the `#pragma pfor` directive; if neither variable is set, `SIMPLE` scheduling is assumed. If `MP_SCHEDTYPE` is set, but `CHUNK` is not set, a `CHUNK` of 1 is assumed. If `CHUNK` is set, but `MP_SCHEDTYPE` is not, `DYNAMIC` scheduling is assumed.

11.2.4 MP_SLAVE_STACKSIZE

The stack size of slave processes can be controlled through the environment variable `MP_SLAVE_STACKSIZE`, which may be set to the desired stacksize in bytes. The default value is 16 MB (4 MB for more than 64 threads).

11.2.5 MPC_GANG

`MPC_GANG` specifies gang scheduling. Set `MPC_GANG` to `ON` to enable gang scheduling. To disable gang scheduling, set `MPC_GANG` to `OFF`.

11.3 Communicating Between Threads Through Thread Local Data

The routines described in this section allow you to perform explicit communication between threads within their multiprocessing C program. These communication mechanisms are similar to message−passing, one−sided−communication, or `shmem`, and may be desirable for reasons of performance and/or style.

The operations allow a thread to fetch from (`get`) or send to (`put`) data belonging to other threads. Therefore, these operations can be performed only on data that has been declared to be `-Xlocal` (that is, each thread has its own private copy of that data; see the `ld(1)` man page for details on `Xlocal`). A get operation requires that the *source* parameter point to `Xlocal` data, while a put operation requires that the *target* parameter point to `Xlocal` data.

The following routines are available as part of the Message Passing Toolkit (MPT) and are similar to the original `shmem` routines (see the `shmem` reference page), but are prefixed by `mp_`:

```
void mp_shmem_get32  (int *target,  
int *source,  
int length,  
int source_thread)  
  
void mp_shmem_put32  (int *target,  
int *source,  
int length,  
int target_thread)  
  
void mp_shmem_iget32 (int *target,  
int *source,  
int target_inc,  
int source_inc,  
int length,  
int source_thread)  
  
void mp_shmem_iput32 (int *target,  
int *source,  
int target_inc,  
int source_inc,  
int length,  
int target_thread)  
  
void mp_shmem_get64(long long *target,  
```

```
long long *source,  
int length,  
int source_thread)  
  
void mp_shmem_put64  (long long *target,  
long long *source,  
int length,  
int target_thread)  
  
void mp_shmem_iget64 (long long *target,  
long long *source,  
int target_inc,  
int source_inc,  
int length,  
int source_thread)  
  
void mp_shmem_iput64 (long long *target,  
long long *source,  
int target_inc,  
int source_inc,  
int length,  
int target_thread)
```

The following rules apply to the preceding listed routines:

- Both *source* and *target* are pointers to 32−bit quantities for the 32−bit versions, and to 64−bit quantities for the 64−bit versions of the calls. The actual type of the data is not important, because the routines perform a bit−wise copy.
- For a `put` operation, the target must be `Xlocal`. For a `get` operation, the source must be `Xlocal`.
- *length* specifies the number of elements to be copied, in units of 32 or 64−bit elements, as appropriate.
- *source_thread* and *target_thread* specify the thread−number of the remote processing element (PE).
- A get operation copies from the remote PE. A put operation copies to the remote PE.
- *target_inc* and *source_inc* are specified for the strided `iget` and `iput` operations. They specify the increment (in units of 32−bit or 64−bit elements) for source and target when performing the data transfer. The number of elements copied during a strided `put` or `get` operation is still determined by *length*.

  Note: Call these routines only after the threads have been created (typically, the first `pfor`/parallel region). Performing these operations while the program is still serial leads to a run−time error because each thread's copy has not yet been created.

In the example below, compiling with `-Wl,-Xlocal, myvars` ensures that each thread has a private copy of `x` and `y`.

```
struct {  
      int x;  
      double y[100];  
} myvars;
```

The following example copies the value of `x` on thread 3 into the private copy of `x` for the current thread.

```
mp_shmem_get32 (&x, &x, 1, 3)
```

The next example copies the value of *localvar* into the thread 5 copy of `x`.

```
mp_shmem_put32 (&x, &localvar, 1, 5)
```

The example below fetches values from the thread 7 copy of array `y` into `localarray`.

```
mp_shmem_get64 (&localarray, &y, 100, 7)
```

The next example copies the value of every other element of `localarray` into the thread 9 copy of `y`.

```
mp_shmem_iput64 (&y, &localarray, 2, 2, 50, 9)
```

11.4 Synchronization Intrinsics

The intrinsics described in this section provide a variety of primitive synchronization operations. Besides performing the particular synchronization operation, each of these intrinsics has two key properties:

- The function performed is guaranteed to be atomic (typically achieved by implementing the operation using a sequence of load–linked and/or store–conditional instructions in a loop).
- Associated with each instrinsic are certain memory barrier properties that restrict the movement of memory references to visible data across the intrinsic operation (by either the compiler or the processor).

  A visible memory reference is a reference to a data object potentially accessible by another thread executing in the same shared address space. A visible data object can be one of the following:

– C/C++ global data

– Data declared `extern`

– Volatile data

– Static data (either file–scope or function–scope)

– Data accessible via function parameters

– Automatic data (local–scope) that has had its address taken and assigned to some visible object (recursively)

The memory barrier semantics of an intrinsic can be one of the following three types:

acquire barrier	Disallows the movement of memory references to visible data from after the intrinsic (in program order) to before the intrinsic. (This behavior is desirable at lock–acquire operations.)
release barrier	Disallows the movement of memory references to visible data from before the intrinsic (in program order) to after the intrinsic. (This behavior is desirable at lock–release operations.)
full barrier	Disallows the movement of memory references to visible data past the intrinsic (in either direction), and is thus both an acquire and a release barrier. A barrier restricts only the movement of memory references to visible data across the intrinsic operation: between synchronization operations (or in their absence), memory references to visible data may be freely reordered subject to the usual data–dependence constraints.

By default, it is assumed that a memory barrier applies to all visible data. If you know the precise set of data objects that must be restricted by the memory barrier, you can specify the set of data objects as additional arguments to the intrinsic. In this case, the memory barrier restricts the movement of memory references to the specified list of data objects only, possibly resulting in better performance. The specified data objects must be simple variables and cannot be expressions (for example, `&p` and `*p` are disallowed).

Caution: Conditional execution of a synchronization intrinsic (such as within an `if` or a `while` statement) does not prevent the movement of memory references to visible data past the overall `if` or `while` construct.

11.4.1 Atomic fetch–and–op Operations

The fetch–and–op operations are as follows:

```
<type> __fetch_and_add (<type>* ptr, <type>   
value, ...)  
<type> __fetch_and_sub (<type>* ptr, <type>   
value, ...)  
<type> __fetch_and_or  (<type>* ptr, <type>   
value, ...)  
```

```
<type> __fetch_and_and (<type>* ptr, <type>   
value, ...)  
<type> __fetch_and_xor (<type>* ptr, <type>   
value, ...)  
<type> __fetch_and_nand(<type>* ptr, <type>   
value, ...)  
<type> __fetch_and_mpy (<type>* ptr, <type>   
value, ...)  
<type> __fetch_and_min (<type>* ptr, <type>   
value, ...)  
<type> __fetch_and_max (<type>* ptr, <type>   
value, ...)
```

<type> can be any of the following:

```
int  
long  
long long  
unsigned int  
unsigned long  
unsigned long long
```

The ellipses (...) refer to an optional list of variables protected by the memory barrier.

Each of these operations behaves as follows:

- Atomically performs the specified operation with the given value on **ptr*, and returns the old value of **ptr*.

  ```
  {tmp = *ptr; *ptr <op>= value; return tmp;}
  ```

- Full barrier

11.4.2 Atomic op−and−fetch Operations

The op−and−fetch operations are as follows:

```
<type> __add_and_fetch (<type>* ptr, <type>   
value, ...)  
<type> __sub_and_fetch (<type>* ptr, <type>   
value, ...)  
<type> __or_and_fetch  (<type>* ptr, <type>   
value, ...)  
<type> __and_and_fetch (<type>* ptr, <type>   
value, ...)  
<type> __xor_and_fetch (<type>* ptr, <type>   
value, ...)  
<type> __nand_and_fetch(<type>* ptr, <type>   
value, ...)  
<type> __mpy_and_fetch (<type>* ptr, <type>   
value, ...)  
<type> __min_and_fetch (<type>* ptr, <type>   
value, ...)  
<type> __max_and_fetch (<type>* ptr, <type>   
value, ...)
```

<type> can be any of the following:

```
int  
long  
long long  
unsigned int  
unsigned long  
unsigned long long
```

Each of these operations behaves as follows:

- Atomically performs the specified operation with the given value on **ptr*, and returns the new value of **ptr*.

```
{*ptr <op>= value; return *ptr;}
```

· Full barrier

11.4.3 Atomic compare–and–swap Operation

The compare–and–swap operation is as follows:

```
int __compare_and_swap (<type>*   
ptr, <type> oldvalue, <type>   
newvalue, ...)
```

`<type>` can be one of the following:

```
int  
long  
long long  
unsigned int  
unsigned long  
unsigned long long
```

This operation behaves as follows:

· Atomically compares **ptr* to *oldvalue*. If equal, it stores the new value and returns 1, otherwise it returns 0.

```
if (*ptr != oldvalue) return 0;  
else {  
      *ptr = newvalue;  
      return 1;  
}
```

· Full barrier

11.4.4 Atomic synchronize Operation

The synchronize operation is as follows:

```
__synchronize (...)
```

The ellipses (`...`) refer to an optional list of variables protected by the memory barrier.

This operation behaves as follows:

· Issues a `sync` operation

· Full barrier

11.4.5 Atomic lock and unlock Operations

11.4.5.1 Atomic lock–test–and–set Operation

The lock–test–and–set operation is as follows:

```
<type> __lock_test_and_set (<type>* ptr  
, <type> value, ...)
```

`<type>` can be any of the following:

```
int  
long  
long long  
unsigned int  
unsigned long  
unsigned long long
```

This operation behaves as follows:

- Atomically stores the supplied value in *ptr and returns the old value of *ptr

  ```
  {tmp = *ptr; *ptr = value; return tmp;}
  ```

- Acquire barrier

11.4.5.2 Atomic lock−release Operation

The lock_release operation is as follows:

```
void __lock_release (<type>* ptr, ...)
```

<type> can be one of the following:

```
int  
long  
long long  
unsigned int  
unsigned long  
unsigned long long
```

This operation behaves as follows:

- Issues sync then sets *ptr to 0 and flushes it from the register

  ```
  {*ptr = 0}
  ```

- Release barrier

11.4.6 Example of Implementing a Pure Spin−Wait Lock

The following example shows implementation of a spin−wait lock:

```
int lockvar = 0;  
while (__lock_test_and_set (&lockvar, 1) != 0); /* acquire the lock */  
      ...   read and update shared variables ...  
__lock_release (&lockvar);                      /* release the lock */
```

The memory barrier semantics of the intrinsics guarantee that no memory reference to visible data is moved out of the above critical section, either ahead of the lock−acquire or past the lock−release.

Note: Pure spin−wait locks can perform poorly under heavy contention.

If the data structures protected by the lock are known precisely (for example, x, y, and z in the example below), then those data structures can be precisely identified as follows:

```
int lockvar = 0;  
while (__lock_test_and_set (&lockvar, 1, x, y, z) != 0);  
       ...   read/modify the variables x, y, and z ...  
__lock_release (&lockvar, x, y, z);
```