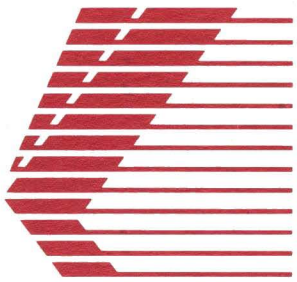
Programming with the Uniform System



BUTTERFLY PLUS

PROGRAMMING WITH

THE UNIFORM SYSTEM

October 28, 1987

RELEASE LEVEL

This manual conforms to the Beta Test version of the Butterfly Plus Parallel Processor released October, 1987, and Version 4.0 of the Chrysalis operating system.

NOTICE

BBN Advanced Computers Inc. (BBNACI) has prepared this manual for the exclusive use of BBN customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by BBNACI. BBNACI assumes no responsibility for any errors that appear in this document.

The information in this publication is subject to change without notice.

Contents

Chapter 3 Uniform System Examples

Chapter 4 Tuning Programs for Performance

Chapter 5 Uniform System Library Routines

Figures

Chapter 1

The Uniform System Approach

The Butterfly Plus™ hardware and Chrysalis™ operating system are a foundation on which a variety of software structures may be built. Experimentation with a wide range of software applications has produced a teachable, efficient programming style for using this foundation. This style, called the Uniform System approach, has proven particularly effective for applications containing a few frequently repeated tasks (*e.g.*, much of scientific computing). It has also been used successfully in applications with less homogeneous task structures.

The Uniform System is a library of subroutines that can be used with C language or FORTRAN programs. This manual is written from the point of view of the C language. For the most part, the FORTRAN and C language implementations of the Uniform System are functionally equivalent. Any minor differences are pointed out in this manual.

There are two versions of the Uniform System library: one for the Butterfly Plus and one for the frontend machine (typically a VAX or Sun workstation). The frontend machine version implements all the routines in the Butterfly Plus version and emulates enough of the Chrysalis functions to permit most C language programs to be partially debugged on the frontend machine in a uniprocessor environment. The partially debugged programs can then be moved to the Butterfly Plus parallel processor.

Beyond the usual concerns of programming, there are two key considerations specific to the Butterfly Plus parallel processor: storage management and processor management. The goal of storage management is to use the full memory bandwidth of the machine; that is, to keep all the memories in the machine equally busy, thereby preventing the slowdown that occurs when many processors attempt to access a single memory. The goal of processor management is to utilize the full processor bandwidth of the machine; that is, to keep all the processors equally busy, thereby preventing the inefficiency that occurs when some processors are overloaded while others sit idle.

MEMORY MANAGEMENT

The Butterfly Plus switch provides low delay, high bandwidth access to all memory in the machine. To help the programmer take advantage of this "common memory," the Uniform System implements a large shared memory for application programs, and provides means to spread application data uniformly across the memories of the machine.

The Chrysalis operating system provides "memory mapping" operations that enable processes to manage their address spaces, and hence the memory they access. Two or more processes can share memory by mapping the same memory segment.

In practice, memory sharing among processes is typically used in two different ways. One approach to programming the machine is to isolate processes from one another by mapping memory so that only a relatively small subset of each process address space is accessible to other processes. This subset can be changed at any time, and is often different for different groups of processes. This method facilitates debugging by limiting the number of processes likely to have touched a particular data structure.

The Uniform System uses a different approach, which is to share a single large block of memory by mapping the block into the address space of each process. This frees the application programmer from the need to

manipulate memory maps, and simplifies programming by implementing a large shared address space for application programs. Data two or more processors must share is allocated without regard to which processors will be using it. The stack and variables local to individual processors are kept locally, and like code, are not fetched across the Butterfly Plus switch.

Collectively, the memories of Butterfly Plus processor nodes form the shared memory of the machine. This means the large shared memory an application program sees is implemented by a collection of separate memories. If all the shared data used by an application happened to be located in a single physical memory, contention for that memory (as many processors attempt to access the data) would force the processors to proceed serially, thereby slowing program execution. Since the aggregate memory bandwidth of the machine is very large (10 gigabits per second for a 256 processor machine), slowdowns due to memory contention can be reduced by scattering application data across the physical memories of the machine. When many processors access data that has been scattered, their references tend to be distributed across the memories and can make use of the full memory bandwidth of the machine. The Uniform System Library memory allocator scatters data structures in a way that allows straightforward addressing conventions. The system also supports a set of more specialized techniques for use if the allocator is either inappropriate or ineffective.

To summarize, the approach to memory management used by the Uniform System is based on two principles:

- Use a single large address space shared by all processes to simplify programming; and
- Scatter application data across all memories of the machine to reduce possible memory contention.

This memory management strategy has a cost, due both to the slower access to remote memory and to possible contention in the switch and at the memories. This cost is an increase in execution time, typically from 4% to 8%, and is due less to contention than to the slightly slower remote

access. The benefit of this memory management strategy is that the programmer can treat all processors as identical workers, each able to do any application task, since each has access to all application data. This greatly simplifies programming the machine, a benefit that far outweighs the modest cost.

Another aspect of memory management is the need to make certain memory operations atomic. The Chrysalis kernel provides an extensive repertoire of primitive atomic operations. When the atomic operations required are more complex than these primitives provide, the primitives can be used to build simple locks that, in turn, can be used to implement arbitrarily complex atomic operations.

PROCESSOR MANAGEMENT

The most novel aspect of programming the Butterfly Plus is processor management. This falls naturally into two separate parts: identification of the parallel structure inherent in a chosen algorithm, and controlling the processors to achieve the determined parallelism. In many applications the parallel structure is both obvious and rich. In others, the structure is less clear and may require reworking the algorithm. Occasionally, an application will be inherently serial, and cannot be structured to take advantage of parallel processing. We can, however, offer a few guidelines:

1. Start with the best existing algorithm that implements the application. A Butterfly Plus system with **P** processors can do no more than speed up an algorithm by a factor of **P**. Speeding up a poor algorithm may not overcome its inefficiencies. For example, it may take an $O(N^2)$ parallel sort longer to run on a 128 processor Butterfly Plus than it takes an $O(N\log N)$ sort to run on a single processor.

2. Attempt to do the same number and kind of steps as those in the best algorithm. The order of steps in an algorithm can often be manipulated to achieve parallelism. This may involve adding logic in the form of simple locks to ensure the atomicity of selected operations.

3. Look for parallel structure at all levels and in all sizes: the more the better. If necessary, it is usually relatively easy to combine small tasks at a later stage into larger more manageable sizes; it is often more difficult to divide a task at a later stage into smaller ones. For example, if an application requires fast Fourier transforms (FFTs) on a number of different channels, the programmer should plan to exploit both the parallelism inherent in an individual FFT and the parallelism due to different channels.

   The Butterfly Plus parallel processor can work very efficiently with individual tasks a few milliseconds in length; if necessary, it can also work on tasks in the hundreds of microseconds. For shorter tasks, various overheads begin to interfere with good performance.

There are two strategies for determining the desirable number of concurrent operations to have at any stage in the processing. One strategy recommends a relatively static approach, using exactly as many concurrent tasks as there are processors. The other strategy uses many tasks per processor. Both strategies attempt to deal with end effects—the processor idle time that occurs toward the end of a stage when some processors have finished and others are still working. The first approach minimizes the effect by explicit construction: here the programmer tries to apportion the work so that all processors finish at approximately the same time. The second approach allocates tasks to processors dynamically in an attempt to balance the load. As a processor finishes a task, it is assigned the "next" task ready for execution. This approach minimizes end effects by having many more tasks than processors. Some wait time occurs at the end of the problem, but is generally small relative to the total program execution time.

The Uniform System encourages the dynamic approach, but also supports the static approach. For many applications the dynamic approach is simpler and more reliable, since it is not necessary to know in advance how long an individual piece of work will take. Furthermore, this approach adapts readily to varying numbers of processors and sizes of problems.

Once the programmer determines what processing will occur in parallel, he or she must then control the Butterfly Plus parallel processor to make this happen. There are several ways to do this. The Chrysalis kernel provides a rich collection of relatively low level operations for starting processes on various processors and for communicating among them. The Uniform System provides a higher level abstraction for managing the processors; one that is natural and efficient for a large class of applications.

The Uniform System treats processors as a group of identical workers, each able to do any task. To use the Uniform System, a programmer must structure the application into two parts:

- A set of subroutines that perform various application tasks; and,
- One or more "generators" that identify the "next" task for execution.

To illustrate this, consider matrix multiplication. On a uniprocessor the following nested loop could be used to multiply **n** × **o** matrix **A** by **o** × **m** matrix **B** to produce **n** × **m** result matrix **C** :

```
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        DotProduct (i, j)
```

where the **DotProduct** routine computes the dot product of the *i*th row of **A** and the *j*th column of **B**, then stores the result as the (*i*, *j*)th element of **C**. One way to parallelize this loop would be to cause the **DotProduct** routine to execute as a task, ensuring that the **DotProduct** task gets performed once for each element in the result matrix. There is a task generator in the Uniform System library, called **GenOnA**, that can be used to do this; the uniprocessor nested loop would then be replaced by:

```
GenOnA (DotProduct, n, m);
```

The **GenOnA** task generator causes the **DotProduct** task to execute once for each (row, column) pair in the range (**n**, **m**), using available processors to perform the computation.

Whether intended for parallel execution or not, a well-designed program is usually structured as a set of subroutines to improve program modularity. There is normally a subroutine for each task type, each subroutine taking arguments that define individual tasks in terms of subsets of the program

data to be operated on. To use the Uniform system, the programmer simply insures that these subroutines correspond to the tasks he wants to do in parallel. In the matrix multiplication example there is one task type, computing dot products. Corresponding to that task type is the dot product routine, whose row and column parameters specify particular tasks.

The second part of a Uniform System application code structure consists of one or more subroutines that identify the "next" task for execution. Such a subroutine is called a "generator," since its function is to generate tasks. In a serial program the generator function is usually embedded in the control structure of the program (*e.g.*, do this, do that, then do 10 of these). For parallel processing via the Uniform System, the programmer is expected to make generation of the next task explicit. For the matrix multiplication example, the task generator would be responsible for generating a call on the dot product routine for each element in the result matrix.

It is helpful to think of the generator concept in terms of three procedures and a task descriptor data structure. A generator activator procedure (**GA**) takes as parameters a worker procedure (**W**), a description of data (**D**) upon which work is to be done, and a task generation procedure (**TG**):

```
GA (W, D, TG)
```

The generator activator procedure first builds a task descriptor data structure that specifies the tasks to be generated in terms of the worker procedure, the data , and the task generation procedure. It then "activates" the generator by making the task descriptor available to other processors. The processor that invoked the generator activator, along with other available processors, then uses the task descriptor and the task generation procedure to make repeated calls on the worker procedure, specifying subsets of the data to work upon. Each call of the worker procedure is a task. When the last task is done, the processor that called the generator activator procedure continues execution of its program, while the other processors that worked on the tasks look for other work. In the matrix multiplication example, the worker procedure is the dot product routine, and the data is the operand and result matrices. The dot product worker routine is called once for each combination of row and column index; these indices are

stored in the task descriptor and are incremented indivisibly each time the task generation procedure is executed by a processor.

Conceptually, the generator notion is similar to the various "map" functions in the Lisp language. The unique thing about the Uniform System is that it achieves parallel operation by using processors as they are available to execute the various calls upon the worker procedure. Task generation and the processor management associated with it are implemented in a distributed fashion in the sense that each processor performing tasks participates in their generation.

Often the required generator is quite simple. In the matrix multiplication example, where a dot product is computed for every element in the result matrix, the generator can find the next task by incrementing row and column counters that identify the element in the result matrix to be computed next. Occasionally a generator must be more complex. A generator that selects the next node to process in an alpha-beta tree walk, for example, would rely heavily on using the most up-to-date information about the state of processing of the tree. Sometimes a generator uses a simple queue, in which case it operates much like the process scheduler found in many timesharing systems, where the next task for execution is the one at the front of the queue. In general, though, the generators included in the Uniform System library suffice to build many applications.

The Uniform System library provides a way to bind task generation procedures to worker procedures. The basis for this binding mechanism is a "universal" generator activator procedure. To use this universal generator activator procedure directly, application programs specify both a worker procedure and a task generation procedure. The library also includes a set of generator activator procedures that embody many commonly used task generation procedures. When an application program calls one of these "specific" generator activator procedures, it specifies only the worker procedure. The generator activator passes its associated task generation procedure and a task descriptor to the universal generator activator along with the worker procedure supplied by the application program.[1]

1. This section has been careful to use the terms *generator activator procedure* and

Often an algorithm requires multiple, perhaps nested, instances of generators. As long as the algorithm does not depend upon the order of task generation between different generators, the programmer is free to make multiple calls to task generators to start the system working on all of them at once. If the algorithm does depend upon the order, the programmer must either provide a task generation procedure to properly answer the question about what to do next, or carefully manage the use of existing generator activator procedures to ensure the algorithm's ordering requirements are met.

The Uniform System approach to processor management offers three important benefits:

- The generator mechanism is very efficient. It is implemented using one process per processor in a way that prevents unnecessary context swaps. Each processor executes a tight loop consisting of "generate next task—execute next task." The programmer supplies both the task generation and worker procedures, usually choosing an appropriate generator activator procedure from the library. Both the task generation procedure and the worker procedure execute at the application level. As a result, once a generator gains control of a processor, the Chrysalis kernel need not be involved until the generator has exhausted all the work it knows how to find.
- Programs that use the Uniform System task generation mechanism to exploit parallelism are insensitive to the number of processors. It is possible to debug programs on small configurations and run them on larger ones. Should an application grow to exceed the capacity of its current configuration, it can be moved without modification to a larger one. Perhaps more importantly, programs can also run on "reduced" configurations (*e.g.*, where processors have been removed for repair).

---

task generation procedure. The rest of this manual uses the term *generator*, both when referring to the generator activator procedure and when referring to the result of activating a task generator. We use the more specific terms only when it is important to distinguish between generator activation and task generation.

- The load can be balanced dynamically. Whenever a processor becomes free, a generator identifies the next task to execute. Since the task generation procedures are supplied by the application programmer, the task choice can be based on the current state of the computation and the requirements of the application.

Chapter 2

Using the Uniform System

When the Uniform System approach is used on the Butterfly Plus parallel processor, programs are written in much the same way as for a uniprocessor. In fact, a program that never invokes a task generator can run on a single Butterfly Plus processor node. The program is loaded into all of the processors, however, so the potential for parallel processing is there. Since Chrysalis runs a process scheduler on every processor, it is possible to have several independent application processes running on a single processor. However, when the Uniform System is used, there is usually only one process per processor.

This chapter describes each routine found in the Uniform System library. Several example programs that illustrate how to use the Uniform System routines are contained in Chapter 3. Chapter 4 describes some of the ways programmers can tune programs for better performance. The descriptions of the library routines in this chapter are narrative in nature. The information presented in this chapter is repeated in Chapter 5, which is organized for use as a reference manual for the Uniform System Library.

INCLUDE FILE

All Uniform System programs must include the header file *us.h*:

```
#include <us.h>
```

INITIALIZING THE UNIFORM SYSTEM

The routine:

```
InitializeUs();
```

initializes the Uniform System. This routine creates and starts a Uniform System process on every available processor, sets up the memory that is globally shared among all Uniform System processes, and initializes the Uniform System storage allocator. **InitializeUs** should be called before any other Uniform System routine except **SetUsConfig** or **ConfigureUs**. Normally it should be called only once in a program.

The routine:

```
TerminateUs();
```

can be used to undo the effects of **InitializeUs**. It should be called only by the process that called **InitializeUs**. **TerminateUs** kills all of the processes running on other nodes, and unmaps and deallocates the memory used to support the Uniform System shared memory. **TerminateUs** is useful when a program needs to release resources used by the Uniform System. For example, prior to entering a computational phase the program may not need the Uniform System functions and may require the resources the system was using. **TerminateUs** can also be used if a change in the Uniform System configuration is necessary. **TerminateUs** need not be called immediately prior to exiting a program.

OBTAINING CONFIGURATION INFORMATION

It is sometimes necessary to refer to processors by number. There are two separate numbering schemes for processors, and routines for converting between them. The first scheme uses the hardware processor number, an 8-bit number assigned when the machine is assembled. The hardware processor number for the processor on which a process is running is directly accessible through the Chrysalis variable **Proc_Node**. For the frontend machine version of the Uniform System, **Proc_Node** is set arbitrarily. The particular numbers used as hardware processor numbers for a Butterfly Plus with P processors depend upon the size of the switch and the way the processors are connected to the switch; the hardware

processor numbers used can range from 0 to 255. The important point to note is that the hardware numbering scheme often has gaps.

Because it is generally easier for application software to deal with consecutively numbered processors, the Uniform System implements a second processor numbering scheme that uses virtual processor numbers. These virtual processor numbers form a dense set, consecutively numbered from 0 to P–1, where P is the number of processors available to the program. The virtual processor number for the processor on which a process is running is directly accessible through the Uniform System variable **UsProc_Node**. For the frontend version of the Uniform System, **UsProc_Node** is always 0. The mapping between virtual processor number and hardware processor number may change from run to run. The routines:

```
UsProc = PhysProcToUsProc(PhysProc);
PhysProc = UsProcToPhysProc(UsProc);
```

can be used to convert between hardware processor number and Uniform System processor number.

A program sometimes needs to know the number of processors and the amount of memory available to it. The routines:

```
TotalProcsAvailable();
ProcsInUse();
MemoriesAvailable();
DistinctMemoriesAvailable();
```

return such configuration information. **TotalProcsAvailable** returns the number of processors in the Uniform System configuration; it includes processes that have been removed by the **TimeTest** routine (see the section entitled "Measuring Your Program" later in this chapter). **ProcsInUse** does not count processors that have been removed by the **TimeTest** routine. **MemoriesAvailable** counts memory in units of 64 kilobytes. **DistinctMemoriesAvailable** counts memory modules and is usually the same as **TotalProcsAvailable**, but there are cases when the Uniform System initialization routine (**InitializeUs**) cannot obtain memory on a particular processor node (*e.g.*, when other software, such as the Ethernet routines, have taken it all).

MEMORY MANAGEMENT

Two classes of memory are available to Uniform System programs:

- Process private memory. As the name suggests, data in process private memory can be accessed by only one process.
- Globally shared memory. Data in globally shared memory is accessible by all Uniform System processes.[1]

Within these two memory management classes, several different types of storage are available to C programs. These storage types are best described in terms of the types of variables available to C programs (see Figure 2-1):

- C local variables. Local variables are process private and are stored on the stack. A local variable is visible only within the routine that declares it. There is one instance of the variable for every routine call. Hence, the variable is private to the routine call, and hidden from every other call.
- C globals. C global variables are process private. There is one instance of each such variable per process. These variables are shared by subroutine calls within the same process, but are hidden from all other processes.
- C dynamic storage. Storage of this type, obtained by *malloc* and related routines, is process private. There is one instance of an allocated variable per process. These variables can be accessed by subroutines within the same process (providing the necessary pointers have been made available), but are hidden from all other processes. In particular, while you can pass a pointer from one process to another, if you try to use it within another process you will either get a hardware fault or (worse) access a random chunk of memory in that process.

1. It is possible, using the Chrysalis **Map_Obj** operation, to have memory that is shared among some, but not all, processes. We recommend that you do not use Chrysalis memory management operations directly within Uniform System programs unless you understand the implementation of the Uniform System memory management discipline in detail.

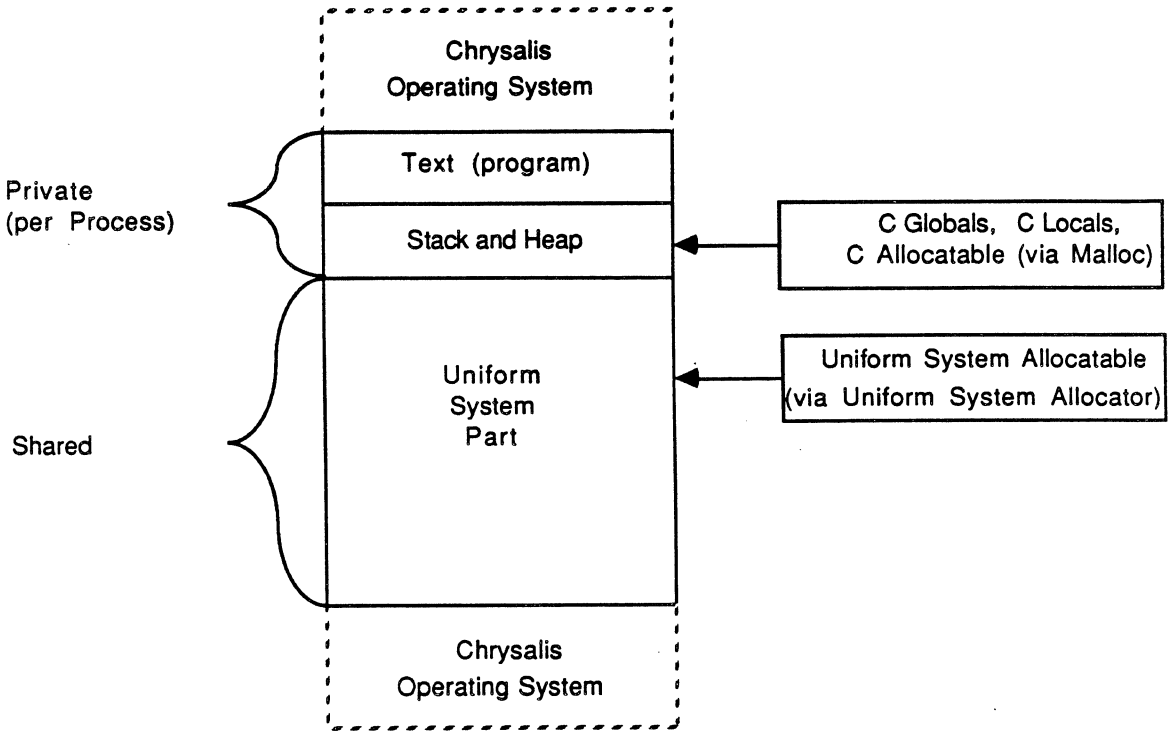


Figure 2-1
Address Space of a Uniform System Process

- Shared storage. Storage of this type is obtained using the Uniform System allocators **UsAlloc**, **UsAllocScatterMatrix**, and the like, and it is globally shared. There is one instance of a Uniform System allocated variable per Butterfly Plus machine. Since this storage is globally shared, pointers to it are valid on all processors and can be passed freely among them. This is the only way to communicate between different processors and tasks, unless you choose to use the Chrysalis mechanisms directly. To get started, most of the Uniform System task generators allow the user to pass a pointer to newly generated tasks. The passed pointer is typically the root of a user-specified data structure. (See also the discussion of the **Share** mechanism later in this chapter.)

The Uniform System storage allocator creates and manages the globally shared memory region of the process address space (see Figure 2-2). A program can ask the allocator for space that is scattered about the

machine, or for space in the memory of a particular processor node. Once such globally shared space has been allocated to a program, the program is free to pass pointers to variables in the space from one processor to another.

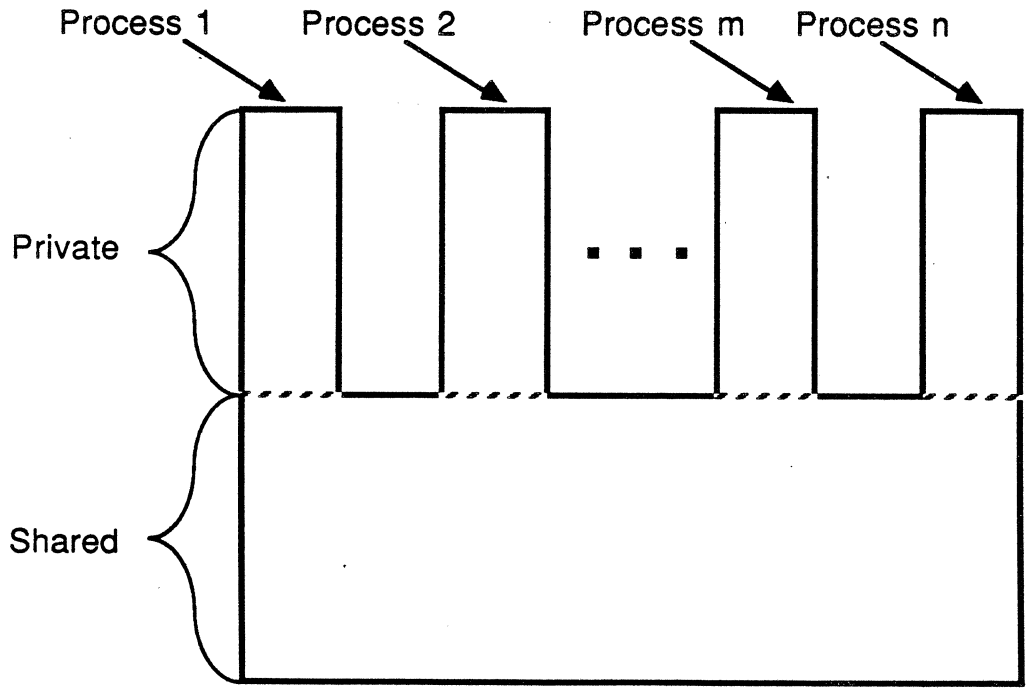


Figure 2-2
Processes Share Much of Their Address Spaces

Keeping the distinction between globally shared memory and process private memory clearly in mind is critical to using the Uniform System. If a program variable is declared to be a C *global*, for example, that only means that the variable is accessible by all the program modules that are linked together to make up a particular process. Since C *globals* are process private, if an identical copy of that process is created on another processor (or on the same processor), the new process will have its own copies of any variables declared as C *globals*. Similarly, the *malloc* and *calloc* system calls allocate memory that is process private rather than globally shared. The Uniform System uses the Chrysalis object management system to implement globally shared memory.

MEMORY ALLOCATORS

The Uniform System provides a variety of memory allocators that allocate storage in globally shared memory. (The normal allocators, such as *malloc*, can be used with Uniform System programs to allocate storage in process private memory.)

Like the normal allocators, the Uniform System allocators return a pointer to the block of memory allocated. If an allocator is unable to obtain the requested amount of memory, it returns the null pointer (*i.e.*, zero).

To allocate a block of storage in globally shared memory, use:

```
UsAlloc(SizeInBytes);
```

The Uniform System allocates the block from the memory with the smallest amount of previously allocated space. To allocate globally shared storage on the local processor, use:

```
UsAllocLocal(SizeInBytes);
```

To specify a particular processor, use:

```
UsAllocOnUsProc(Processor, SizeInBytes)
```

where **Processor** is a Uniform System virtual processor number. If **Processor** exceeds the number of available memories, the space is allocated on node **Processor** modulo **P**, where **P = DistinctMemoriesAvailable()**. If you want to specify the node by its hardware processor number, use:

```
UsAllocOnPhysProc(PhysProcessor, SizeInBytes);
```

Proper storage management on the Butterfly Plus computer is important. If your data is not distributed over all available memory, you may get poor performance. It usually does not save much (a few percent) to keep data near the processor using it. However, clumping a lot of data in a single processor node's memory can result in contention for that memory by multiple processors, and can be devastating to program performance.

The Uniform System library provides storage allocation routines for regular data structures, such as arrays and matrices. These routines scatter data across the memories of the machine in order to reduce memory contention. More complex data structures can be scattered across the machine via repeated calls to **UsAllocOnUsProc**, **UsAllocOnPhysProc**,

or **UsAllocAndReportC** (described in "Tagging Memories" later in this chapter) that specify different memories.

The data structures required by many applications can be represented naturally by two-dimensional matrices. Furthermore, higher dimensional matrices can be represented in a straightforward way by two-dimensional matrices, as can one dimensional vectors. For example, a three-dimensional matrix can be thought of as a two-dimensional matrix, each element of which is a vector. Hence, two-dimensional matrices can be used as a fundamental building block for supporting many application data structures. To reduce potential memory contention, scatter these data structures across the machine.

The routine:

```
UsAllocScatterMatrix(nrows, ncols, element_size)
```

allocates a matrix that is scattered by row over the memories of the machine. It does this by allocating a vector of pointers **nrows** long, and **nrows** separate vectors, each containing **ncols** items of size **element_size** bytes. The Uniform System allocates the row vectors in separate memories. **UsAllocScatterMatrix** returns a pointer to the vector of pointers. The vector of pointers is itself filled in with pointers to the scattered row vectors (see Figure 2-3). Elements of an array **A** allocated in this way can be referenced using standard C array notation:

```
A[i][j]
```

The FORTRAN-callable version of **UsAllocScatterMatrix** will scatter the matrix by columns rather than by rows. It is otherwise identical to the C-language-callable versions. Elements of an array allocated as a scatter matrix can be referenced using standard FORTRAN array notation:

```
A (i, j)
```

Internally, **UsAllocScatterMatrix** uses the routine **UsAllocAndReportC** (described in "Tagging Memories" later in this chapter) to scatter the row vectors.

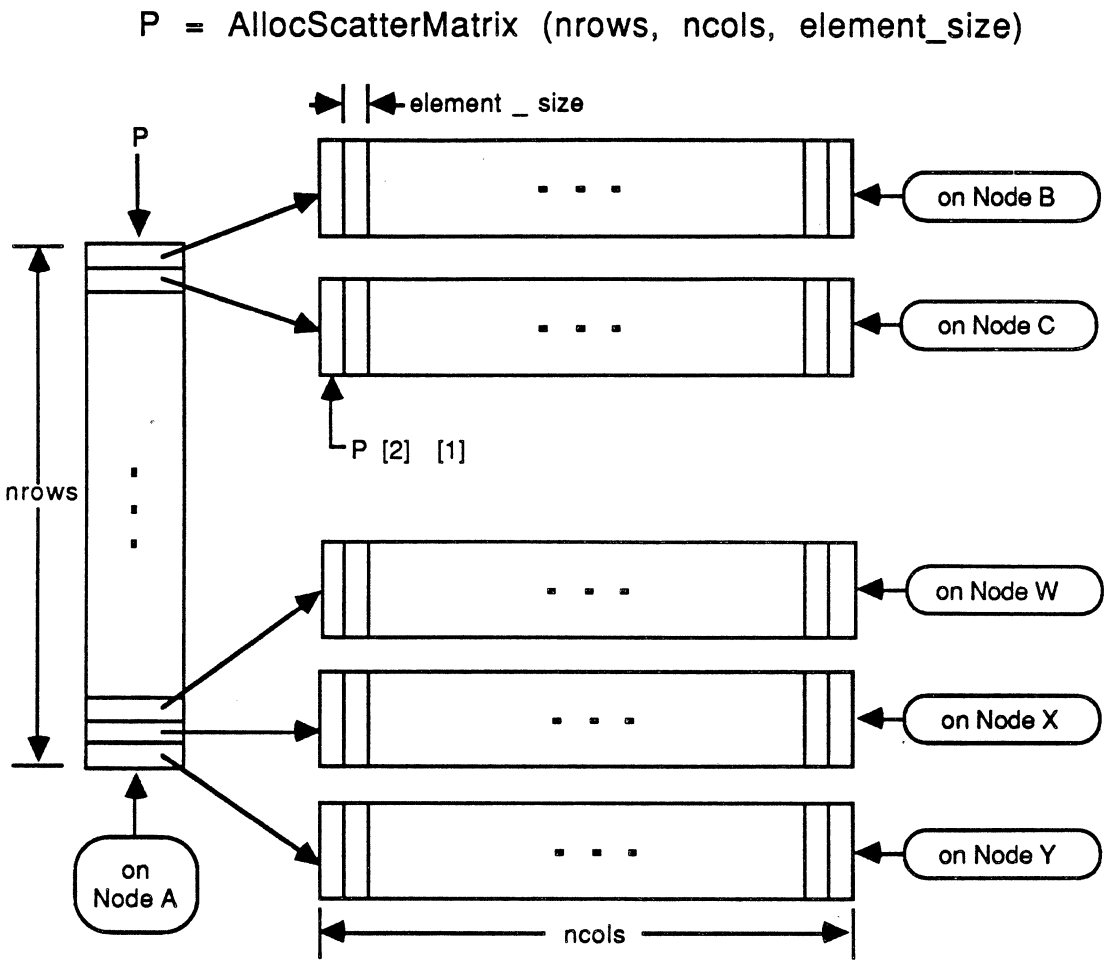


Figure 2-3
A Scattered Matrix Created by UsAllocScatterMatrix

Space allocated by the **UsAlloc...** storage allocators can be deallocated on a block-by-block basis. To free space previously allocated by one of the **UsAlloc...** allocators and pointed to by **p**, use:

```
UsFree(p)
```

Note that **UsFree** works for matrices scattered by **UsAllocScatterMatrix** as well as for simple blocks of storage. The routine:

```
FreeAll()
```

can be used to free all dynamically allocated shared memory. **FreeAll** should be used with care. It reinitializes the Uniform System storage

allocation system, freeing all shared memory, including memory allocated by the **UsAlloc...** allocators and that used by the **Share** mechanism.

It is always worth considering whether to copy the constants used by an application into the local memory in order to avoid possible contention for them. The **Share** routines (described later in this chapter in "Making Copies of Process Private Data") and the generator "initialization" routines (described later in "Task Generators") are useful for making such copies.

SYNCHRONIZATION AND ATOMIC OPERATIONS

Sometimes two processors need to work on the same data at the same time. If the order of work does not matter (*e.g.*, incrementing a counter), the principal concern is that the processors do not interfere with one another (*i.e.*, that one finishes before the other starts). If the order of work does matter (*e.g.*, task A is writing and task B is reading), the program logic may be flawed in the sense that task B is really not ready to run, and should not have been generated until A finished.

In many cases one of the atomic operations supported by Chrysalis is sufficient to prevent processes from interfering with one another. These Chrysalis operations implement a set of *fetch and op* functions that atomically read the value in a memory location, perform an operation on the value, store the operation result back into the memory location, and return the value originally read. The Chrysalis atomic operations (**Atomic_add**, **Atomic_ior**, etc.) work on 16-bit quantities.

Some situations require atomic 32-bit operations. The operation:

```
Atomic_add_long(loc, val);
```

implements 32-bit atomic addition. It atomically adds **val** to the location addressed by **loc**. **Atomic_add_long** is similar to the Chrysalis **Atomic_add** operation. It differs in that it operates on 32-bit quantities and does not support the "fetch" part of the "fetch and add" function provided by **Atomic_add**.[2]

2. In its current implementation, **Atomic_add_long** is atomic only with respect to other **Atomic_add_long** calls. In particular, the execution of a read operation may be

Some cases may require more than a simple atomic operation. In these cases it may be necessary to construct a lock around the code as follows:

```
lock;
  code to do what you want
unlock;
```

The Uniform System provides lock and unlock operations:

```
UsLock(lock, n)
UsUnlock(lock)
```

The **UsLock** operation is a "busy wait" type of lock, where **lock** is a pointer to a *short* variable used as the lock (assumed to have been initialized in the unlocked state with value zero), and **n** is an integer that specifies the time to wait in tens of microseconds between attempts to set the lock. Using zero for **n** forces use of a default, which is about one millisecond. Note that when nesting these operations, care must be taken to avoid deadlock.

If a program simply needs to wait until something occurs, and if "busy" waiting is acceptable, it can use **UsWait**:

```
while (something has not occurred)
    UsWait(n);
```

where **n** is an integer that specifies the time to wait in tens of microseconds. As with **UsLock**, using zero for **n** forces use of a default of about one millisecond.

To wait less than 100 microseconds, **UsWait** "spins" by executing a loop enough times to delay the requested amount. For waits longer than 100 microseconds, **UsWait** uses the realtime clock to determine when to stop waiting. To force the waiting to be done using the realtime clock, regardless of the requested amount, use:

```
UsWaitRtc (n);
```

To force the waiting to be done using the spin loop method, regardless of the requested amount, use:

---

interleaved with an **Atomic_add_long** operation in a way that returns an inconsistent result to the read. This can occur if the high-order 16 bits returned by the read are obtained after the low-order 16 bits are incremented by the **Atomic_add_long**, but before the carry (if any) is propagated to the higher-order bits.

```
UsWaitSpin (n);
```

For a given requested delay, the number of times **UsWait** or **UsWaitSpin** must execute the spin loop depends upon the speed of the processor. The Uniform System uses a wait factor:

```
double UsWaitFactor;
```

to determine how many loop iterations are required for a requested delay. This factor is set assuming that the machine is configured with MC68020 processor nodes. The factor can be recalibrated for an MC68000 (or simply for greater accuracy) by using the routine **UsWaitGetFactor**, which computes a correction to the current factor by timing the loop for a specified time (using the current factor). The following FORTRAN code fragment resets the **UsWaitFactor**:

```
external UsWaitRetFactor, UsWaitGetFactor
double precision UsWaitRetFactor, UsWaitGetFactor
double precision f
f = UsWaitRetFactor()
f = f * UsWaitGetFactor(100000)
call UsWaitSetFactor(f)
```

If "busy" waiting is not acceptable, the Chrysalis operations that manipulate dual queues and events can be used to construct an appropriate wait and signalling discipline.

PROCESSOR MANAGEMENT

The Uniform System processor management mechanism is accomplished using task generators. A task is the basic unit of parallel computation; a Uniform System task is a subroutine call. At any instant there is a set of runnable tasks that must be mapped to the available set of processors. The Uniform System takes the view that both the set itself and the priority of items within the set are dynamically changing; as a result, a simple queue is not an adequate model of the task structure. Instead, the Uniform System requires a user-supplied task generation procedure that can answer the question, "What is the current most important task to run at this instant?"

Task generators are often rather simple. A common parallel operation is to apply some function to each item of a structure (list or array) where the

order is immaterial. For example, this might be the semantics for a PARALLEL DO extension to FORTRAN. In this case the task generation routine need only identify the next item in the list, which it can do by incrementing a counter (atomically, since task generation is performed by each processor for itself). However, a generator can be arbitrarily complex. For example, a generator used in a chess playing program might do alpha-beta pruning of a game tree, using the most up-to-date information to decide where to devote its resources next. In this case, most of the complexity of the code and the execution time of the program might reside in the task generation procedure.

It is good practice to make the tasks themselves small. The responsiveness of the system to changes in priorities depends on the size of a task, because once a task is started, the system runs it to completion. Also, even if the priorities are not changing, there will come a time toward the end of a task generator when all of the tasks have been generated by the task generation procedure. When that happens, if there are no other active generators, some processors will sit idle while others finish the last tasks. If the tasks are small in size, the idle time will not have much impact on program efficiency.

Although the programmer must provide both task generation and task implementor (worker) procedures, experience has shown that the relatively small set of generators (or more precisely, generator activator procedures) supported by the Uniform System library is sufficient for a wide range of applications. The easiest way to achieve parallel operation is to structure the program to fit the mold of one of these task generators.

The Uniform System supports two generator control disciplines:

- **Synchronous** generators return to the caller after all of the generated tasks have been processed. Furthermore, the processor that calls a synchronous generator always works on the tasks that are generated.
- **Asynchronous** generators return to the caller as soon as the generator has been activated. This enables the calling process to do other work. The calling process can later work on generated tasks if it so chooses.

The Uniform System matches available processors to the generated tasks and keeps track of active task generators. Whenever a processor has nothing to do, it obtains a task using the task generation procedure for one of the active generators. When a Uniform System program begins execution, all the processors, except the one used to start the program, are idle. As long as there are active generators with tasks to be done, there are no idle processors.

Generator calls can be nested, in which case the Uniform System deals with the generators in an arbitrary order that depends largely upon the stochastic nature of interprocessor timing. However, because the Uniform System guarantees that at least one processor is working within each synchronous generator, forward progress is assured on each.

There are some situations where it may be possible to place an upper bound on the number of tasks required by a problem, but where the number actually required may be data dependent. For example, consider a search where the search space can be partitioned into N disjoint regions that can be searched by N tasks; if the first task finds the object in the first region, there is little utility in searching the remaining N–1 regions. The Uniform System supports *abortable* generators for such situations. An abortable generator can be terminated before all the tasks it describes have been generated and executed. After an abortable generator has been aborted, it will generate no more tasks; however, any tasks started before the generator was aborted will be processed.

Normally when a generator is active, processors, as they become free, begin working on the generator until either all processors are working on it, or all the tasks have been generated. In situations where several classes of tasks can be active simultaneously, it may be desirable to control the number of processors used for each task class. The Uniform System provides *limited* generators, which use only a specified number of processors (or fewer), for such situations.

Generators are very efficient. It takes a little overhead to get a processor to notice a generator, but once the processor does, it will continue

generating and working on the tasks defined by the generator at a cost of about one extra subroutine call per task.

It is not easy to cause deadlocks using generators, but it is possible. For synchronous generators, since there is always at least one processor working on each generator (perhaps recursively), progress should be made unless that processor hangs. It is, of course, bad practice to write code so that a processor can hang. Unfortunately, it is good practice to write code where processors take turns accessing some resource in an atomic way, and it is not always easy to tell if a program will cause deadlocks just by looking at the code. The distinction, of course, is that accesses made by deadlock-free programs eventually (and usually quickly) give up the resource. With asynchronous generators, more care needs to be taken to avoid race and deadlock conditions.

The Uniform System Library includes a collection of generator activator procedures that embody various commonly-used task generation procedures. The next section describes the synchronous generator activator procedures in the library. The section following that describes the asynchronous generator activator procedures. All these generator activator procedures make use of a "universal" generator activator procedure. Use of the universal generator activator procedure is described later, in the section entitled "Building a Generator."

SYNCHRONOUS GENERATORS

The Uniform System Library supports several major "families" of generators:

- *Index* family. Given an integer range, generators in the index family generate a task for each value (index) within the range.
- *Array* family. Given two integer ranges (which can be thought of as array dimensions), generators in the array family generate a task for each pair of values (which can be thought of as row and column indices) within the ranges.

- *Half array* family. Given two integer ranges, which can be thought of as array dimensions, generators in the half array family generate a task for each array element that is beneath the "diagonal."

Each family of generators has a simple version of the call to the generator, an *abortable* version that allows a process to stop the generator at any time, a *limited* version that restricts task generation to a subset of the processors, and a *full* version that is a superset of all the other calls. The full version of a generator call requires all the arguments and can be used instead of any other call in the same generator family. There are also *asynchronous* versions of all the generators that return control to the process directly after the generator has been called. The full versions of the generators are described first, in the remainder of this chapter, and descriptions of the simpler versions follow.

The Index Family of Generators

Consider a subroutine **Worker(Arg, index, ...)**, which is to be called for all values of **index** from zero through **Range–1**. A call of the form:

```
code = GenOnIFull (Init, Worker, Final, Arg, Range, Limited, Abortable);
```

causes **Worker** to be executed in parallel for the values of **index** between zero and **Range**–1; **Range** must be less than 2^{31}. Task generation is somewhat faster if **Range** is less than 2^{15}, since the task generation procedures can use **Atomic_add** to increment the index. **Arg** is typically a pointer to a problem description data structure. Elements of **Arg** might point to the multiplier, multiplicand, and product matrices in a matrix multiplication problem, for example.

To facilitate application bookkeeping, before the generator calls **Worker** for the first time on a particular processor, it will call:

```
Init(Arg);
```

on that processor. Typically, the **Init** routine is used to copy frequently referenced constants from globally shared memory into process private memory or to initialize process private temporaries. By convention, 0 specifies that there is no **Init** routine.

Similarly, after the last call of the **Worker** routine on each processor used to perform tasks for the generator, the routine **Final** is called once on each such processor used. The **Final** routine is called with **Arg** as a parameter:

```
Final(Arg);
```

and is typically used for per-processor post-processing associated with tasks. By convention, 0 specifies that there is no **Final** routine.

The **Limited** parameter indicates the number of processors to which the generator is to be restricted. A value of 0 or −1 signals no limitation; a positive value ensures that no more than that number of processors will be used on the tasks.

The **Abortable** parameter is a *boolean* variable that indicates whether or not the generator can be aborted. The value of **Abortable** determines the arguments passed to the **Worker** routine. If **Abortable** is *FALSE*, two arguments are passed to **Worker**:

```
Worker(Arg,index);
```

otherwise, if **Abortable** is *TRUE*, each call to **Worker** takes an additional argument:

```
Worker (Arg, index, GenID);
```

where **GenID** is an "identifier" for the generator (C type = **UsGenDesc***, defined in the *#include* file *us.h*).

If the generator identified by **GenID** is abortable, it can be aborted using:

```
AbortGen(GenID, termination_code);
```

where **termination_code** is an integer. **AbortGen** prevents the generation of new tasks. Any tasks in progress when **AbortGen** is called will run to completion.

All synchronous generators in the index family return a value. If a generator is abortable and was aborted, it returns the **termination_code** argument supplied to **AbortGen**. (More than one processor may call **AbortGen** to abort a generator. In such a case, the value returned is the smallest **termination_code** supplied to **AbortGen**.) If all of a generator's tasks have been performed, either the generator was not abortable or it was

abortable but it was not aborted. In either case, the generator returns the code **genEXHAUSTED.**

Other synchronous generators in the index family are useful in situations not requiring the full flexibility of **GenOnIFull.** For example, since these routines take no **Arg** routine, they can be used when calls to **Share** (described below) and its companion routines eliminate the need to pass problem description data structures around.

The generator:

```
code = GenOnI (Worker, Range);
```

generates tasks of the form:

```
Worker(0, index);
```

Note that the **Worker** routine is passed a dummy **Arg** parameter. The generator:

```
code = GenOnILimited (Worker, Range, nprocs);
```

is like **GenOnI,** differing in that it limits the generator to the specified number of processors. The generator:

```
code = GenOnIAbortable (Worker, Range);
```

is like **GenOnI,** differing in that it is abortable; it generates tasks of the form:

```
Worker(0, index, GenID);
```

The Array Family of Generators

The generator:

```
code = GenOnAFull (Init, Worker, Final, Arg, Range1,
                   Range2, Limited, Abortable);
```

is similar to **GenOnIFull** except that **Worker** takes a second index, which runs over **Range2.** More specifically, if **Abortable** is *FALSE,* **GenOnAFull** generates tasks of the form:

```
Worker(Arg,index1,  index2);
```

and if **Abortable** is *TRUE,* it generates tasks of the form:

```
Worker(Arg,index1,  index2, GenID);
```

As with the index family, several additional generators in the array family are useful in situations that do not require the full flexibility of **GenOnAFull**. The generator:

```
code = GenOnA (Worker, Range1, Range2);
```

generates tasks of the form:

```
Worker(0, index1, index2);
```

The generator:

```
code = GenOnALimited (Worker, Range1, Range2, nprocs);
```

is like **GenOnA** except that it limits the generator to the specified number of processors. The generator:

```
code = GenOnAAbortable (Worker, Range1, Range2);
```

is like **GenOnA** except that it is abortable. It generates tasks of the form:

```
Worker(0, index1, index2, GenID);
```

The Half Array Family of Generators

The generator:

```
code = GenOnHAFull(Init,  Worker, Final, Arg, Range1,
                    Range2, Limited, Abortable);
```

is similar to **GenOnA**, except for the range of the **index1**, **index2** arguments passed to the **worker** routine. The sequence of **(index1, index2)** values span the "half" array beneath the diagonal of a **Range1** × **Range2** array as follows:

```
index2 = 0,   index1 = 1,...,(Range1-1)
index2 = 1,   index1 = 2,...,(Range1-1)
 ...
index2 = R-2, index1 = (R-1),...,(Range1-1)
```

where:

```
R = min(Range1,Range2)
```

Similarly, the generators:

```
code = GenOnHA (Worker, Range1, Range2);
code = GenOnHALimited (Worker, Range1, Range2, nprocs);
code = GenOnHAAbortable (Worker, Range1, Range2);
```

are analogous to the corresponding routines in the array family.

It may appear that more variants are needed for half arrays; for example, those that include the diagonal. However, **GenOnHA** can be used with some simple tricks to get the desired behavior; for example, to include the diagonal, add one to the ranges in the call to **GenOnHA** and subtract one from **index1** and **index2** in the **Worker** routine.

Miscellaneous Generators

The generator:

```
GenTaskForEachProc (call, arg);
```

generates exactly one task, **call(arg)**, for every processor (that has not been removed by the **TimeTest** routine).

The generator:

```
GenTaskForEachProcLimited (call, arg, nprocs);
```

generates exactly one task, **call(arg)**, for each of **nprocs** different processors.

The generator:

```
GenTasksFromList (routine_list, arg_list, n);
```

where **routine_list** is an array of routines of length **n**, **r1,...,rn**, and **arg_list** is an array of "arguments" of length **n**, **arg1,...,argn**, generates **n** tasks. The **i**th task is **ri(argi)**.

ASYNCHRONOUS GENERATORS

There are asynchronous versions of each of the generators in the index, array, and half array generator families. Although the form of the tasks generated by these generators varies from family to family, the asynchronous generators use a common control discipline.

Suppose **AsyncGen...** is an asynchronous generator. The call:

```
GenID = AsyncGen...(...);
```

activates the generator and then returns control immediately to its caller along with **GenID**, an "identifier" for the generator activated.

The processor that invokes an asynchronous generator can choose to work on tasks generated by the generator, or can do other things. To work on tasks from the generator, it uses the call:

```
code = WorkOn (GenID);
```

After all of the tasks generated have been processed, **WorkOn** returns a **code** to the caller. The **code** indicates either that the generator exhaustively produced all of its tasks or that it was aborted via **AbortGen**.

The sequence:

```
GenID = AsyncGen...(...);
code = WorkOn (GenID);
```

is functionally equivalent to the corresponding synchronous generator.

A processor that has previously invoked an asynchronous generator can use the call:

```
code = WaitForTasksToFinish (GenID);
```

to wait until all the tasks associated with the specified generator have been completed. As with **WorkOn**, the returned **code** indicates whether the generator exhaustively produced all its tasks or was aborted.

Both **WorkOn** and **WaitForTasksToFinish** should be used only by the process that activated the generator in question, and only if that process is not already working on the generator. Furthermore, a process may invoke either **WorkOn** or **WaitForTasksToFinish**, but not both.

All task generators (synchronous and asynchronous) use a task descriptor data structure to record information about the generator, such as the identity of the worker routine, and to keep track of generator progress. When a generator is invoked, a task descriptor data structure is allocated, and when the generator completes, the task descriptor data structure is deallocated. For synchronous generators, the deallocation occurs before the generator call returns. For asynchronous generators, the deallocation is done within the **WorkOn** or **WaitForTasksToFinish** routine. The present implementation limits the number of generators that may be active at any time to 256 (of course, each generator can describe thousands of tasks). Therefore, a program that makes more than 256 calls to

asynchronous generators must use **WorkOn** or **WaitForTasksToFinish** to force the deallocation of at least some of the task descriptor data structures to proceed beyond the first 256 calls.

The asynchronous generators currently supported by the Uniform System are:

Index family:

```
GenID = AsyncGenOnIFull (Init, Worker, Final, Arg, Range, Limited,
                         Abortable);
GenID = AsyncGenOnI(Worker,  Range);
GenID = AsyncGenOnILimited (Worker, Range, nprocs);
GenID = AsyncGenOnIAbortable (Worker, Range);
```

Array family:

```
GenID = AsyncGenOnAFull (Init, Worker, Final, Arg, Range1, Range2,
                         Limited, Abortable);
GenID = AsyncGenOnA(Worker,  Range1, Range2);
GenID = AsyncGenOnALimited (Worker, Range1, Range2, nprocs);
GenID = AsyncGenOnAAbortable (Worker, Range1, Range2);
```

Half Array Family:

```
GenID = AsyncGenOnHAFull (Init, Worker, Final, Arg, Range1, Range2,
                          Limited, Abortable);
GenID = AsyncGenOnHA (Worker, Range1, Range2);
GenID = AsyncGenOnHALimited (Worker, Range1, Range2, nprocs);
GenID = AsyncGenOnHAAbortable (Worker, Range1, Range2);
```

Each of these corresponds to one of the synchronous generators described above.

COPYING PROCESS PRIVATE DATA

It is often useful for each processor to have its own copy of certain frequently referenced variables declared as C globals. These copies eliminate the memory contention that might otherwise occur as multiple processors access shared copies of the variables. For example, as part of initialization one processor might set C global variables that other processors must access. Recall that C globals are in process private memory. One way to make the values of these variables accessible to the other processors is to pass the values in the data structure argument to a task generator and have the generator "initialization" routine make copies on each processor. Often a more convenient way to achieve this effect is to use one of the

Share routines.

Assume that X is a 4-byte data item (*e.g.*, an integer) declared as *global* or *static*; X is therefore process private. The effect of:

```
Share(&X);
```

is to copy the value of X into each processor that performs tasks generated by subsequent task generators. The value copied is the value X has when **Share** is invoked. The value of X is copied to each processor prior to the call of the task initialization routine for the next task generator handled by that processor. For generators that have no explicit initialization routine, X is set prior to the first call of the task worker routine on that processor. The effect of **Share** is illustrated schematically in Figure 2-4. Note that when processor P executes **Share**(&X) the effect is as if:

1. P allocates space in shared memory to hold the value of X;
2. On the next generator called:

   For the **Arg** parameter for the generator, P passes a pointer to the shared memory location that holds the copy of X;

   For the **Init** parameter for the generator, P passes a routine that copies the value of X from the shared memory location pointed to by **Arg** to the location of X in process private memory.

In particular, note that the value of X is propagated to other processors by the **Share** mechanism, but the variable X itself is not shared. Therefore, should one processor change its copy of X, only that processor will see the changed value.

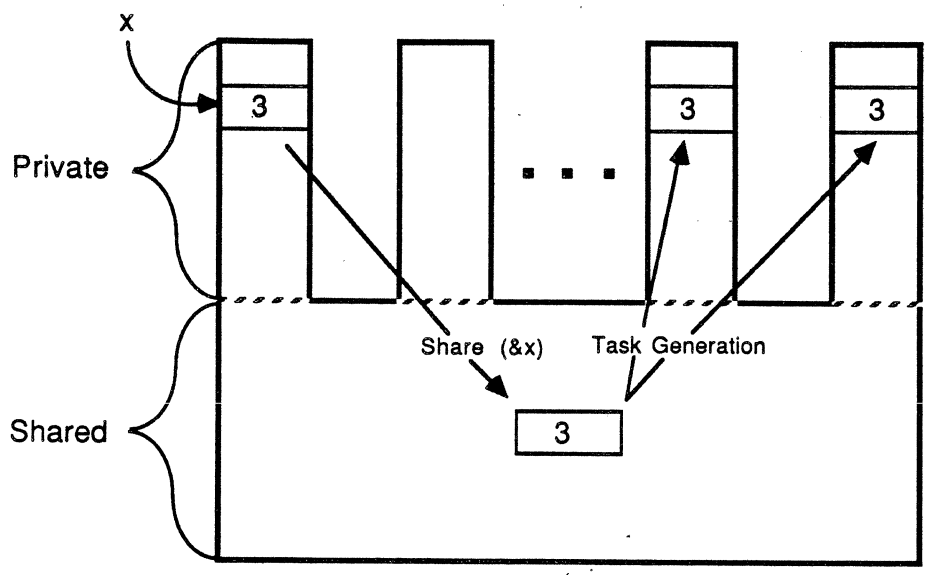


Figure 2-4
Share Passes Copies of Process Private Variables

A block of data declared *global* or *static* can be passed to other processors by:

```
ShareBlk(&X, size);
```

where size is the size of the block, in bytes, and &X is its starting address.

A pointer variable P, which is declared to be *global* or *static*, and the block of data it points to can be passed to other processors by:

```
SharePtrAndBlk(&P, size);
```

where size is the size of the block in bytes. The following code fragment allocates and initializes a block of data in process private memory and then uses **SharePtrAndBlk** to propagate the data to other processors:

```
int * p;
 ...
p = (int *) malloc (10 * sizeof (int));
for (i = 0; i < 10; i++)
    p [i] = i;
SharePtrAndBlk (& p, 10 * sizeof (int));
```

When many processors make frequent references to many elements of an array allocated by **UsAllocScatterMatrix**, it is often desirable for each processor to have its own copy of the vector of pointers created by **UsAllocScatterMatrix**. This reduces contention for those pointers, which are

all stored in a single memory and which must be referenced to access the array elements. The routine:

```
ShareScatterMatrix(&P, nrows);
```

where **P** is a C global allocated by:

```
P = UsAllocScatterMatrix(nrows, ncols, element_size);
```

causes such copies to be made. Each processor that performs tasks generated by task generators called after the call to **ShareScatterMatrix** will have its **P** set to point to a local copy of the vector of pointers (the local copy is allocated in globally shared memory). As with **Share**, **ShareBlk** and **SharePtrAndBlk**, the value of **P** in each such processor will be set prior to the call of the task initialization routine for the next task generator handled by that processor. The call:

```
ShareScatterMatrix(&P, nrows);
```

is equivalent to:

```
SharePtrAndBlk(&P, 4 × nrows);
```

in terms of the copies made. However it differs in two important ways: **ShareScatterMatrix** maintains information about the scatter matrix required for proper specification of **UsFree** (see below), whereas **SharePtrAndBlk**. In addition, **ShareScatterMatrix** operates faster on larger Butterfly Plus configurations, since it is careful to avoid memory contention by making copies from other copies as well as from the original.

The FORTRAN-callable version of **ShareScatterMatrix** takes as its second parameter the number of columns rather than the number of rows. Stated somewhat differently, the second parameter of **ShareScatterMatrix** is the number of scattered entities, which for C language matrices are rows and for FORTRAN matrices are columns.

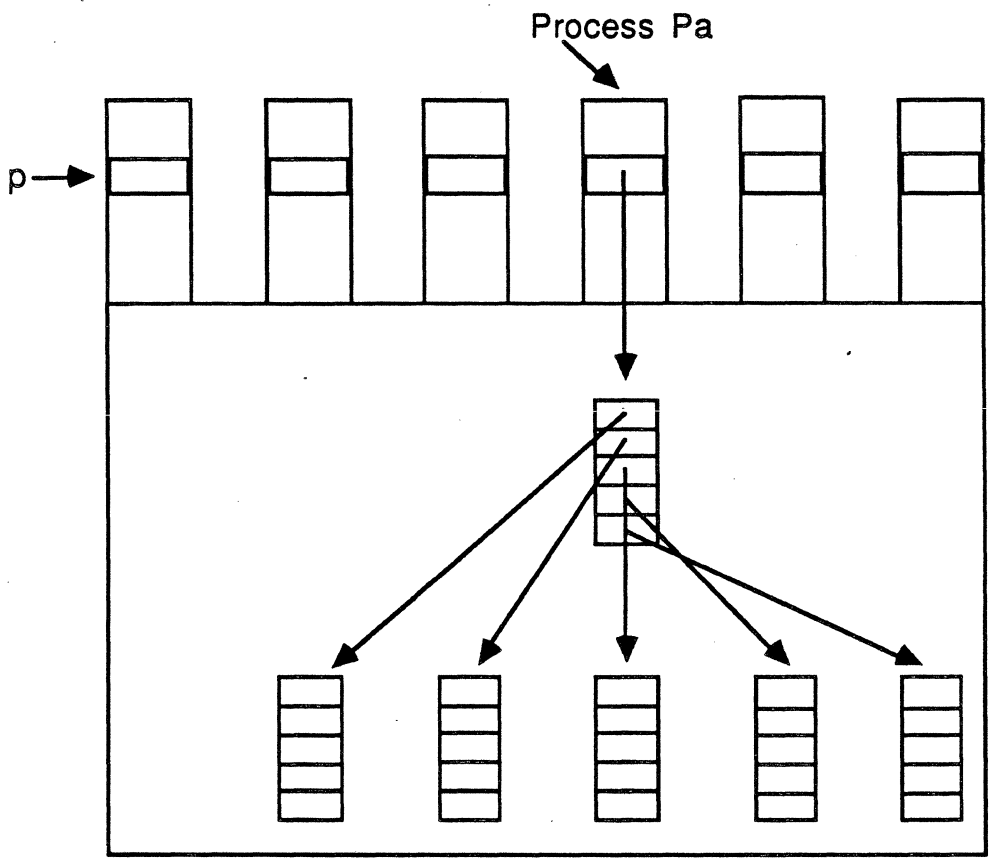


Figure 2-5
Between UsAllocScatterMatrix and ShareScatterMatrix Calls

Figures 2-5 and 2-6 show the effect when a process **Pa** executes the code sequence:

```
p = UsAllocScatterMatrix (n, m, size);
ShareScatterMatrix (&P, n);
GenOn...;
```

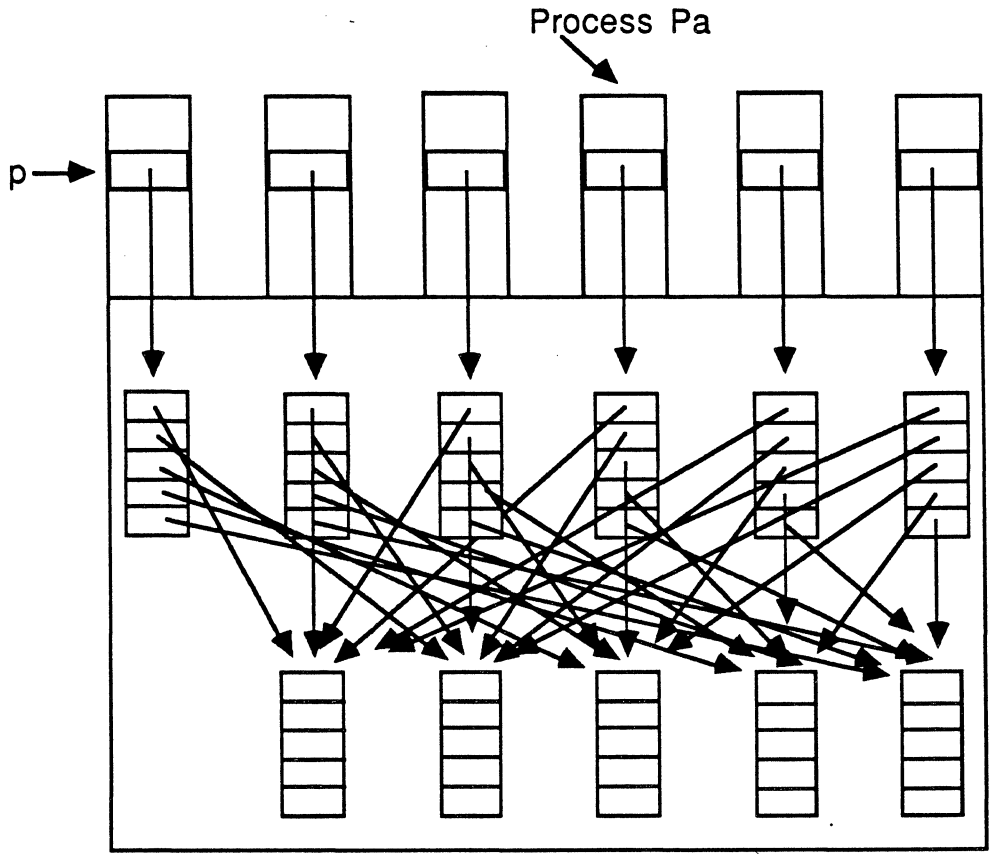


Figure 2-6
After All Processors Start Working on GenOn... Tasks.

To deallocate a scatter matrix after **ShareScatterMatrix** has been used to make copies of the vector of pointers, **UsFree** must be called for each of the vector of pointers. When **UsFree** is called for the last vector of pointers, the rows of the scatter matrix (columns for FORTRAN arrays) are deallocated.

The **Share** mechanism was developed to facilitate program initialization by making it relatively easy to propagate the values of process private variables set during program initialization to all processes. However, the mechanism is also often used to propagate updates to process private variables to all processes.

Normally, the **Share** mechanism automatically propagates copies of such data. The copies are "refreshed" each time a process starts working on tasks from a new task generator. This is done within the generator

mechanism by checking to see whether there are new values to be copied before the process gets its first task from the new generator.

Although this automatic mechanism is adequate when the **Share** mechanism is used to propagate values of process private variables set during program initialization, it is often not adequate when the **Share** mechanism is used in other ways. The routine:

```
RefreshLocalShareVariables();
```

can be used by a process to force a refresh of its copies of any process private variables that may have been updated and re-**Shared** by other processes.

As part of its operation, the **Share** mechanism allocates memory. In addition to the memory it allocates and makes available to the user's program via **ShareScatterMatrix** and **SharePtrAndBlk**, it allocates memory for internal bookkeeping. If there is insufficient memory, the **Share** mechanism will fail. This failure could occur when **Share** (**ShareBlk**, **SharePtrAndBlk** or **ShareScatterMatrix**) is called, or it could occur when a copy of the data being shared is being made on a processor as part of generator initialization for that processor. Depending upon how a **Share** operation is being used, its failure may or may not be "fatal." For example, the failure of **Shares** being done to initialize process private variables are probably fatal, whereas the failure of **Shares** being done to reduce memory contention probably are not.

Each of the **Share** routines normally returns a boolean value (1 = TRUE, 0 = FALSE) that indicates whether or not the routine succeeded in setting up the requested **Share**. In addition, **SetUsConfig** (see the section entitled "Configuring the Uniform System" later in this chapter) can be used with the **configStopOnShareFail** and **configWarningOnShareFail** configuration codes to control the behavior of a program when a failure in the Share mechanism occurs. The default behavior when a **Share** failure occurs is for the process detecting the failure to print a warning message and suspend itself. The FORTRAN versions of the **Share** routines do *not* provide this return value, and are called as subroutines.

SHARING VARIABLES AMONG PROCESSORS

The share mechanism described in the previous section propagates copies of variable values from one processor to others. Situations often occur where it is desirable to share variables among processors in a more dynamic fashion, such that when one processor changes the value of such a variable all the processors see the change.

Ideally, one would like to use a storage class specifier, similar to *static* or *extern*, to declare that a variable is to be shared in this fashion; for example:

```
shared int N;
int M;
```

would cause **N** to be allocated in the globally shared portion of the address space, and **M** to be allocated in the process private portion of the address space. However, as noted earlier, the Butterfly Plus C compiler is a standard uniprocessor C compiler that does not support the notion of globally shared storage.

The Uniform System supports a mechanism that achieves the effect of a globally shared storage class by facilitating the creation and use of dynamically shared variables. This mechanism allows a programmer to declare and use a set of variables that are globally shared among all processors. There are three parts to the mechanism:

1. Declaration of the shared variables. The declaration:

```
BEGIN_SHARED_DECL
    int N;
    char c;
     ...
END_SHARED_DECL;
```

declares **N**, **c**, and the other variables between **BEGIN_SHARED_DECL** and **END_SHARED_DECL**, to be globally shared.

2. Allocation of the shared variables. The macro:

```
MakeSharedVariables;
```

which must be called after **InitializeUs** and before using the shared variables, allocates space for the variables and propagates knowledge of where they are to all processors.

3. Referencing the shared variables. To reference a globally shared variable that has been declared in this way, the programmer must explicitly specify that it is shared via the SHARED prefix; for example:

```
SHARED N = (x + SHARED N) / 12;
if (SHARED c == '1') break;
```

The constructs **BEGIN_SHARED_DECL**, **END_SHARED_DECL**, **SHARED**, and **MakeSharedVariables** are all macros processed by the C language preprocessor. They are not available to FORTRAN programs.

When using this mechanism, some important limitations must be kept in mind:

- **BEGIN_SHARED_DECL** may appear only once in a program. All variables to be shared via this mechanism must be declared in one place.
- All of the shared variables are allocated in the same physical memory. Hence, contention for that memory could be a performance bottleneck. (See Chapter 4, *Tuning Programs for Performance*, for a discussion of the performance implications of memory contention.)

Despite these limitations, the mechanism is useful in many situations.

MEASURING YOUR PROGRAM

You may want to measure the performance of your program on different numbers of processors. The Uniform System offers a utility routine called **TimeTest** that facilitates this kind of measurement:

```
TimeTest(Init, Execute, PrintResults);
```

To use **TimeTest**, you need to divide your application into three major subroutines: one that does all of the initialization (**Init**), another that does the real work of the program (**Execute**), and a third that prints results (**PrintResults**). **TimeTest** takes the names of these subroutines as arguments, and runs your application on various configurations of the machine by calling **Init**, **Execute**, and **PrintResults** in sequence. It times the middle routine only (**Execute**), and passes the execution time, the number of processors, and the effective number of processors to the specified display

routine (**PrintResults**) at the end of each pass:[3]

```
PrintResults(time, procs, effprocs);
int time, procs;
float effprocs;
```

The Uniform System provides a simple **PrintResults** routine called **TimeTestPrint** that outputs **time**, **procs**, and **effprocs**. You may prefer to supply your own display routine that prints other information.

At the start of the run, **TimeTest** prompts the user for the number of processors to use. The user is asked how many processors should be used the first time the **Execute** routine is timed (**start**), how many processors should be added for each iteration of the time test (**delta**), and how many processors should be used for the last time test (**end**).

```
Please enter start, delta (0=exp), and end for time test: 4 2 16
using start = 4, delta = 2, end = 16
```

The effect of this interaction is to start a program on four nodes and increase the number of nodes by two for each timed run until 16 nodes are used. If there are only 15 processors available to the program, only 15 processors will be used for the last timed run. If the **start** parameter is set to 0 or 1, the first timed run will use one processor. If the **delta** parameter is set to zero, the number of processors will increase exponentially (*i.e.*, 1, 2, 4, 8). If the **end** parameter is set to zero, the final timed run uses all available processors.

A variant on **TimeTest** eliminates the need to obtain the processor configurations to be timed from the user:

```
TimeTestFull(Init, Execute, PrintResults, start delta, end);
```

TimeTestFull allows a **start**, increment (**delta**) and **end** value to be specified for a set of runs. The first test is run on **start** processors, the next on **start** + **delta** processors, and so forth, up to the final test that is run on **end** processors. **TimeTestFull** is particularly useful on bigger machines, where incrementing by one processor can be tedious. If **start**

3. The effective number of processors is a *float* equal to (time 1 proc)/(time **n** procs), which is a good measure of the speedup your program achieves over one processor when **n** processors are used. If the first test run uses more than one (=**k**) processor, then the effective number of processors is **k** (time k proc)/(time n procs).

(or **end**) is zero, the test is run from (to) the end of the range of available processors, and in particular, it is run for the limiting processor case whether or not it is in the normal progression specified by **delta**.

If **delta** is specified to be zero, the number of processors used increases by powers of two (1, 2, 4, 8, etc.). The rules for **start** and **end** still apply. If the **delta** specified is negative, **start** and **end** are ignored and **TimeTest-Full** asks the user to supply values for **start**, **delta**, and **end**. This is the normal usage for timing many programs, and is equivalent to **TimeTest**.

Although all processor nodes in a Butterfly Plus system are functionally equivalent, there are situations in which some nodes may appear to be slower to application programs than others. A node will appear to be slower than others if it is running a window manager process. When benchmarking a program, avoid using such nodes so that the measurements are not affected by the processing requirements of the window manager. Chrysalis releases since 3.0 provide a multiple user capability based on the notion of partitions of processor nodes, called *clusters*, that are allocated to users. Each user has at least one cluster, and one of the processors in one of the clusters runs the user's window manager process. The proper way to avoid a node running a window manager process is to create a new cluster with the desired number of nodes and to run the program in that cluster. The following sequence of shell interactions illustrates how this can be done:

```
(cluster 5) [8] make-cluster 8   ; Create cluster with 8 nodes
NewCluster = 7
(cluster 5) cluster 7            ; Cause programs to run in new cluster
Cluster = 7
(cluster 7) [8]
    ...                          ; Run your program in original cluster
(cluster 7) cluster 5
(cluster 5) free-cluster 7       ; Delete new cluster
```

Prior to Chrysalis release 3.0, the node to be avoided was the *king node* and the way to avoid it was for a program to use the routine:

```
InitializeUsForBenchmark();
```

rather than **InitializeUs** to initialize the Uniform System, and to start the program on a node other than the king node.

READING THE CLOCK

A program can read the Butterfly Plus clock using the routine:

```
GetRtc();
```

which returns the time since the system was booted in units of 62.5 microseconds. On the Butterfly Plus the clock value is the same (plus or minus two ticks) on every processor. The frontend version of the Uniform System Library uses the real time clock on the frontend machine to implement **GetRtc**, and converts to these 62.5 microsecond units. If you merely want the clock to measure the speed of your program, see the section entitled "Measuring Your Program" earlier in this chapter.

INPUT AND OUTPUT

The routines **printf** and **scanf** are available for terminal I/O. The operation of these functions is generally the same as that of their UNIX counterparts.

A RAMfile package is available that uses Butterfly Plus memory to implement a "file" system. Programs can create, read, and write RAMfiles, and there are utilities for moving RAMfiles between the Butterfly Plus and the frontend host file system. In addition, support for the standard UNIX file I/O functions for files on the frontend host is being developed. In the interim, a simple mechanism, supported by a *streams* package, has been developed to permit a program running on the Butterfly Plus to read and write files on the frontend computer. Consult the *Chrysalis Programmer's Manual* for details on how to use the streams package.

CONFIGURING THE UNIFORM SYSTEM

Normally **InitializeUs** creates a process for its program on every available processor in the system, and seizes as much memory as it can use from each processor node in order to establish the Uniform System globally shared address space. Although this is appropriate in many cases, there are situations that may require finer control of the resources used by Uniform System programs. In such situations, the routine:

```
SetUsConfig(configuration_code, value);
```

can be used prior to calling **InitializeUs**. The **configuration_code** serves to identify a configuration parameter to be set, and **value** specifies its value.

To set more than a single parameter, use the routine;

```
ConfigureUs(Spec, n);
```

where **Spec** is an array (of integers) that specifies the configuration in terms of **n** parameter specification blocks. Each parameter specification block contains a **configuration_code** that identifies the parameter being set, followed by the value for the parameter.

Note: Before setting configuration parameters using **configuration_codes** you must include the *#include* file *usgen.h*.

The following **configuration_codes** are defined:

configProcs Specifies the number of processors to include in the Uniform System configuration. The number should be an integer less than or equal to the number of nodes in the cluster. If **configProcs** is set greater than the number of available nodes, the Uniform System uses only the nodes available to it.

configSuppressInitMsgs Specifies whether to print messages that report the progress of **InitializeUs**: 1 means suppress the messages, 0 means print the messages. The default is 1.

configTimeTestViaReinit Specifies behavior of the **TimeTest** mechanism (see "Measuring Your Program"). 1 means completely reinitialize the Uniform System for each configuration timed. This ensures that only the resources of the processors being timed are used; in particular, only the memory of those processors is used. 0 means use the

Uniform System as is, by "diverting" enough processors to an idle loop in order to time a given processor configuration. This means that only the CPU resources of the processors in the configuration are used, but the memory resources of all processors, including those that have been diverted, may be used. The default is 0. This parameter may be specified anytime before calling **TimeTest**.

configAllocAcross64K Specifies whether Uniform System memory allocators (see "Memory Allocators") may allocate blocks of memory that cross 64-kilobyte boundaries in a process address space. Early versions of the Uniform System would not allocate blocks that cross 64-kilobyte boundaries. 1 means allow allocation across 64-kilobyte blocks; 0 means don't allow allocation across 64-kilobyte blocks. The default is 1.

configWarningOnShareFail
Specifies behavior on a failure of the **Share** mechanism (see "Copying Process Private Data"); 1 means print a warning message on a **Share** failure; 0 means don't print a warning message on a **Share** failure. The default is 1.

configStopOnShareFail Specifies behavior on a failure of the **Share** mechanism (see "Copying Process Private Data"); 1 means suspend process on a **Share** failure; 0 means allow process to continue execution on a **Share** failure. The default is 1.

configMemObjsFree Specifies the number of 64-kilobyte memory objects to leave free on each processor node. Setting this configuration option overides any previous use of **configMaxMemObjs**.

configMaxMemObjs Specifies the maximum number of 64-kilobyte memory objects to obtain from each processor when making the Uniform System shared memory. Setting this configuration option overides any previous use of **configMemObjsFree**.

configObjsRetRoot During **InitializeUs**, the Uniform System obtains memory in 64-kilobyte blocks to be used to build its shared address space. It obtains as many 64-kilobyte blocks as it can on each processor node in the configuration. It then returns some 64-kilobyte blocks on each node to allow operations requiring memory to occur on the nodes; for example, running the various Chrysalis utilities such as **ps** and **showmem** require memory. This configuration code is used to specify the number of 64-kilobyte blocks the Uniform System should return for the processor node on which the Uniform System program is started (the root processor). The parameter is interpreted only if **configMaxMemObjs** and **configMemObjsFree** have not been specified. The default is 2.

configObjsRetChild Similar to **configObjsRetRoot**, but specifies the number of 64-kilobyte blocks to be returned for child processor nodes. The parameter is interpreted only if **configMaxMemObjs** and **configMemObjsFree** have not been specified. The default is 2.

configMaxSars Specifies the maximum size of the shared portion of the process address space in terms of 64-kilobyte blocks or "segments."[4] This

4. Prior to the Butterfly Plus, Butterfly processor nodes contained a custom memory

parameter should be an integer greater than 15. The default is 237.

configTotalSars Specifies the maximum size of the process address space in terms of 64-kilobyte blocks or "segments."[5] This includes the space consumed by the program, the stack, process private data, and Uniform System shared memory. The default value for this parameter is 256. The maximum allowable value for this parameter is also 256. This value restricts Uniform System programs to a 16-megabyte address space.

As an example, the code fragment:

```
#include <usgen.h>
SetUsConfig(configProcs, 6);
InitializeUs();
```

limits the Uniform System program to (a maximum of) six processors. The code fragment:

```
SetUsConfig(configprocs,  3);
SetUsConfig(configSuppressInitMsgs,0);
```

is equivalent to the code fragment:

```
int config [4];
config [0] = configProcs;
config [1] = 3;
config [2] = configSuppressInitMsgs
config [3] = 0;
ConfigureUs(config,  2);
```

---

management unit that made use of registers called Segment Attribute Registers (SARs). On those machines, the Uniform System used one SAR for each 64-kilobyte segment of the shared portion of the process address space.

5. It was useful to use this configuration code prior to the Butterfly Plus to reduce the number of SARs required by a program, because SARs were a relatively scarce processor node resource. Since Butterfly Plus processor nodes do not contain SARs, Butterfly Plus programs should not need to use this configuration code.

TAGGING MEMORIES

Sometimes it is useful to partition the node memories into classes. For example, the **UsAlloc** and **UsAllocScatterMatrix** routines use all of the memories of the machine. It may be desirable to limit allocation to a smaller set of memories; for example, only the memories of processor nodes being used to run a program. The routine:

```
UsSetClass(proc, class);
```

where **proc** is a physical processor number and **class** is an integer, makes the memory of the specified processor node a member of the specified **class**. The function:

```
UsGetClass (proc);
```

returns the class of which **proc** is a member. All memories are initially in class 0.

The allocation routines:

```
UsAllocC(nbytes, class);
UsAllocScatterMatrixC(nrows, ncols, nybtes, class);
UsAllocOnUsProcC(usproc, nbytes, class);
```

where **class** is an integer, are similar to **UsAlloc**, **UsAllocScatterMatrix**, and so on, differing in that they allocate space only on memories in the specified **class**. **UsAllocOnUsProcC** will fail if **proc** is not in **class**.

The allocation routine:

```
UsAllocAndReportC (usproc, wherep, nbytes, class)
```

attempts to allocate a block of size **nbytes** on a processor in the specified class and, if successful, sets the location pointed to by **wherep** (an **int** *) to the Uniform System ID for the processor on which the block was allocated. The routine first attempts to allocate the space on **usproc**; should that fail, it tries **usproc+1**, and so forth (wrapping around to processor 0), until it either succeeds, or has tried all processors in the class. **UsAllocAndReportC** is useful for building allocators for scattered data structures, such as the scatter matrices allocated by **UsAllocScatterMatrix**.

The following program fragment illustrates the use of these routines:

```
UsSetClass(10,3);
```

```
UsSetClass(4,3);
UsAllocC(64,3);
```

It sets processors 10 and 4 to class 3 and allocates 64 bytes on either 10 or 4, whichever has the least memory previously allocated.

BUILDING A GENERATOR

The Uniform System Library contains a set of useful generators for a wide range of applications. Occasionally it may be necessary, however, to construct a generator for a particular application. Building a generator is an advanced topic, and although the general approach to building generators is not likely to change, the details may. The generator activators supported by the library all make use of the "universal" generator activator procedure. This procedure can be called directly by application programs, and can be used to build new generator activator procedures:

```
ActivateGen(Init, Worker, Final, Arg, Range1, Range2, Type, GenProc,
                Async, MaxProcsToUse, Abortable, ResultP, Language);
```

Init is the per-processor initialization routine, **Worker** is the task worker routine, and **Final** is the per-processor post-processing routine. **Arg** is a pointer to a data structure, which is passed to the **Init**, **Worker**, and **Final** routines. **Type** must be set to GENERATOR, and **Range1** and **Range2** are integers. **GenProc** is a task generation routine described in more detail below. The **Async** parameter specifies whether the generator is to be synchronous or asynchronous. It should be set to *TRUE* for synchronous or *FALSE* for asynchronous. **MaxProcsToUse** specifies the maximum number of processors that the generator can generate tasks for. To use all available processors, **MaxProcsToUse** should be set to **TotalProcsAvailable()**. The **Abortable** parameter determines whether the generator is to be abortable. It should be set to *TRUE* if the generator is to be abortable or *FALSE* if the generator is not to be abortable. **ResultP** is a pointer used when **Abortable** is true. It specifies where to store the generator "result code" if the generator is aborted so that the generator activator routine can find it. Finally, **Language** specifies the programming language from which the generator is being called (0 for the C langauge, 1 for FORTRAN).

GenProc is the task generation routine. It is of the form:

```
GenProc(TD);
```

```
UsGenDesc * TD;
```

where TD is a pointer to a task descriptor data structure in globally shared memory. The task descriptor data structure, UsGenDesc, which is defined in *us.h*, is shown in Figure 2-7:

```
typedef struct UsGenDescStruct
{   int       id;
    short *   completion_location;
    struct UsGenDescStruct * prev_ptr;
    struct UsGenDescStruct * next_ptr;
    int       in_hash_table;
    char *    currentShare;
    int       end;
    short     started;
    short     us_lock;
    short     retcode;
# define genEXHAUSTED -1
    short     endlock;
    int       (* init)();
    int       (* gen)();
    int       (* final)();
    int       arg;
    int       (* call)();
    int       range;
    int       range2;
    int       abortable;
    short     max_procs_to_use;
    short     language;
# define C_CALLING 0
# define FORTRAN_CALLING 1
    union
    {    longLong;
    short Short;
    }         index;
    union
    {    unsigned long Long;
    unsigned short Short;
    }         index2;
    short     lock;
} UsGenDesc;
```

Figure 2-7
The UsGenDesc typedef

The fields **id** through **language** are used internally by the Uniform System generator mechanism. The **Worker, Init, Final, GenProc, Arg, Range1, Range2, MaxProcsToUse,** and **Language** parameters of **ActivateGen** are used to initialize the **call, init, final, gen, arg, range, range2,**

max_procs_to_use, abortable, and **language** fields of the data structure. The other fields between **id** and **language** are used by the generator mechanism for internal bookkeeping. The remaining fields (**index** through **lock**) are initialized to 0, and are available for use by the **GenProc** route for any generator-specific bookkeeping associated with generating tasks.

After **ActivateGen** initializes the task descriptor data structure, it makes the descriptor accessible to other processors. If **Async** is *TRUE*, **ActivateGen** then returns its caller a pointer to the task descriptor data structure; otherwise, the processor running **ActivateGen** calls the **GenProc** task generation procedure. That processor, and others as they become free, use the task generator descriptor (**TD**) and the **GenProc** task generation procedure to generate and execute calls to the **Worker** procedure.

An example may help illustrate use of **ActivateGen** to build a generator. Suppose a generator:

```
GenOnShortIndex(Init, Worker, Arg, Range);
```

is desired that is to be similar to **GenOnI**, differing in that it takes an **Init** routine and an **Arg** parameter, and that the **Range** is to be restricted to a *short*. **GenOnShortIndex** could be implemented by calling:

```
ActivateGen(Init, Worker, 0, Arg, Range, 0, GENERATOR, GenShortIdx, FALSE,
                    TotalProcsAvailable, FALSE, 0, 0);
```

where **GenShortIdx** is:

```
GenShortIdx(TD) UsGenDesc *TD;
{ register int index;
  register short * p1=(short *)&TD->(index.Short);
  register short range = TD->range;
  register int (*worker)()=  TD->call;
  register int arg = TD->arg;
  for (;;)
  { index = Atomic_add(p1,1);    /* make next index */
    if (index >= range) break;   /* range exceeded? */
    (*worker) (arg, index);      /* no: call worker */
  }
}
```

ActivateGen initializes a task generator descriptor (**TD**, a **UsGenDesc** data structure) from its parameters, and makes the descriptor accessible to other processors. The processor on which **ActivateGen** is invoked then calls **GenShortIdx**. That processor, along with others as they become

free, use the task generator descriptor and **GenShortIdx** to generate and execute tasks.

Chapter 3

Uniform System Examples

This section presents several example programs that illustrate use of the Uniform System.

MULTIPROCESSOR "HELLO WORLD"

This example illustrates the use of the task generator **GenOnI**, the variable **Proc_Node**, and the routines **TotalProcsAvailable**, **PhysProcToUsProc**, and **Share**. It is a multiprocessor version of the "hello world" program in Kernighan and Ritchie's *The C Programming Language*, and is only a little more complicated. The program causes each processor to print out, "Hello world from node *n*," exactly once. The output produced by running it on a large Butterfly Plus system is shown in Figure 3-1.

```
(cluster 14) [c] Hello

There are 32 nodes on this machine

Hello from node #29 (= hardware node #c)
Hello from node #5 (= hardware node #9c)
    ...
Hello from node #13 (= hardware node #ac)
Hello from node #3 (= hardware node #98)
(cluster 14) [c]
```

Figure 3-1
Output from "Hello World" Program

The multiprocessor "hello world" program uses **UsAlloc** to reserve space in globally shared memory for **nodecount**, a variable used for bookkeeping by the processors. **Nodecount** is initialized with the number of processors on the machine, a number obtained via **TotalProcsAvailable**. After using **Share** to propagate the location of **nodecount** to other processors, the program then uses **GenOnI** to generate tasks that print the "hello" message from each processor. The only tricky part is ensuring that each processor performs exactly one task. In general, without some form of coordination, some processors could get more than one task and others might get none. For this program, the coordination is simple. After printing its message, each processor atomically decrements a counter maintained in globally shared memory (**nodecount**), and then waits until the counter indicates that all messages have been printed. This guarantees that no processor finishes its task until all messages have been printed; therefore all tasks are generated before any processor finishes. The program code is shown in Figure 3-2.

```
/* Multiprocessor "Hello" program */

#include <us.h>

short * nodecount;

PrintHello (dummy, index)
    int dummy, index;
{   printf ("Hello from node #%d (= hardware node #%x)0,
        PhysProcToUsProc(Proc_Node), Proc_Node);
    Atomic_add (nodecount,-1);
    while (*nodecount!=  0) UsWait (0);
}

main ()
{   InitializeUs ();
    nodecount = (short *) UsAlloc (sizeof (short));
    * nodecount = TotalProcsAvailable ();
    printf ("\nThere are %d nodes on this machine\n\n", *nodecount);
    Share (& nodecount);
    GenOnI (PrintHello, * nodecount);
}
```

Figure 3-2
Program Code for "Hello World" Program

MATRIX MULTIPLICATION

This example illustrates use of the **UsAllocScatterMatrix** storage allocator, the **GenOnA** task generator, and the routines **InitializeUs**, **Share**, **TimeTest**, and **TimeTestPrint**. The example is an unoptimized program that multiplies two matrices. The program computes the matrix **a** = **b** * **c**. Recall that the product (**a**) of two matrices (**b** and **c**) is the matrix whose *(i,j)*th component is the sum of the products of the corresponding elements (the dot product) of the *i*th row of **b** and the *j*th column of **c**.

The program is written to run on a set of processor configurations specified from the keyboard. The output produced by running the matrix example program on a small Butterfly Plus system is shown in Figure 3-3.

```
(cluster 14) [c] MatrixExample

Starting Matrix Multiply
Matrix Size: 20

 Please enter start, delta (0=exp), and end for time test: 1 0 8
 Using start = 1, delta = 0, end = 8

a row 0 0. 60. 120. 180. 240. 300.
a row 1 3. 63. 123. 183. 243. 303.
a row 2 6. 66. 126. 186. 246. 306.
a row 3 9. 69. 129. 189. 249. 309.
a row 4 12. 72. 132. 192. 252. 312.
a row 5 15. 75. 135. 195. 255. 315.
[1] time = 6287 ticks =    .39 sec;  ep =   1.0;  eff = 1.0000

a row 0 0. 60. 120. 180. 240. 300.
a row 1 3. 63. 123. 183. 243. 303.
a row 2 6. 66. 126. 186. 246. 306.
a row 3 9. 69. 129. 189. 249. 309.
a row 4 12. 72. 132. 192. 252. 312.
a row 5 15. 75. 135. 195. 255. 315.
[2] time = 3580 ticks =    .22 sec;  ep =   1.7;  eff = .8780

a row 0 0. 60. 120. 180. 240. 300.
a row 1 3. 63. 123. 183. 243. 303.
a row 2 6. 66. 126. 186. 246. 306.
a row 3 9. 69. 129. 189. 249. 309.
a row 4 12. 72. 132. 192. 252. 312.
a row 5 15. 75. 135. 195. 255. 315.
[4] time = 1798 ticks =    .11 sec;  ep =   3.4;  eff = .8741

a row 0 0. 60. 120. 180. 240. 300.
a row 1 3. 63. 123. 183. 243. 303.
a row 2 6. 66. 126. 186. 246. 306.
a row 3 9. 69. 129. 189. 249. 309.
a row 4 12. 72. 132. 192. 252. 312.
a row 5 15. 75. 135. 195. 255. 315.
[8] time = 1146 ticks =    .07 sec;  ep =   5.4;  eff = .6857
(cluster 14) [c]
```

Figure 3-3
Output from Matrix Multiplication Program

The line:

```
please enter start ...
```

is used to specify the processor configurations for the run. It is printed by the TimeTest routine. See the previous chapter for an explanation of the

start, **delta** and **end** parameters. The line:

```
[8] time = 1146 ...
```

is printed by **TimeTestPrint**. It indicates that the matrix example program took 1,146 ticks or 0.07 seconds on eight processors, and that it achieved a speedup of 5.4 over one processor (= 5.4 effective processors), utilizing the eight processors with 68.6% efficiency. The program itself is shown in Figure 3-4.

```
/* Matrix multiply - unoptimizedexample  program */

#include <us.h>

int Size;
float * * a, * * b, * * c;

InitProblemOnce ()
{ int i, j;
  a = (float * *) UsAllocScatterMatrix (Size, Size, sizeof(float));
  b = (float * *) UsAllocScatterMatrix (Size, Size, sizeof(float));
  c = (float * *) UsAllocScatterMatrix (Size, Size, sizeof(float));
  ShareScatterMatrix (& a, Size);  Share (& b);  Share (& c);
  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++)
    {  if (i==j) b[i][j] = 3.; else b[i][j] = 0.;
       c[i][j] = Size * i + j;
    }
}

InitPerRun ()
{ int i, j;
  for (i=0; i<Size; i++)
    for (j=0; j<Size; j++)
      a[i][j] = 0.;
}

DotProduct (dummy, i, j)
  int dummy, i, j;
{ int k;  float * bb, * cc, temp;
  temp = 0.0;     bb = b[i];     cc = c[j];
  for (k=0; k<Size; k++)
    temp += *bb++ * *cc++;
  a[i][j] = temp;
}

Body ()
{ GenOnA (DotProduct, Size, Size);
}

PrintAnswer(time,  procs, speedup)
  int time, procs; float speedup;
{ int i, j;
  for (i=0; i<6; i++)
  { printf ("\na row %d ", i);
    for (j=0; j<6; j++)
      printf ("%d. ", (int) a[i][j]);
  }
  printf ("\n");
  TimeTestPrint (time, procs, speedup);
}

main ()
{ InitializeUs ();
```

```
    printf ("\nStarting Matrix Multiply\nMatrix Size: "); scanf ("%d", &Size);
    Share (&Size);
    InitProblemOnce ();
    TimeTest (InitPerRun, Body, PrintAnswer);
}
```

Figure 3-4
Program Code for Matrix Multiplication Program

This program parallelizes matrix multiplication by computing the individual elements of product matrix **a** in parallel. Each element is the dot product of a row of matrix **b** and a column of matrix **c**. Chrysalis starts the program by calling the routine **main** on a single processor. The program has six routines:

1. **InitProblemOnce**, as its name suggests, is an initialization routine, called once per invocation of the program, that reserves space in globally shared memory for the result matrix, **a**, and the two operand matrices, **b** and **c**, using the Uniform System allocator, **UsAllocScatterMatrix**. The variables **a**, **b**, and **c** are C *global*s and, hence, process private. Next, **InitProblemOnce** uses **Share** to make copies of **a**, **b**, and **c** available to any processors used in tasks generated to do the matrix multiplication. Finally, it initializes the **b** and **c** matrices (with dummy data) using nested *for* loops. Since matrix **b** will be accessed by row, and matrix **c** will be accessed by column, **b** is scattered by row and **c** is scattered by column. That is, **b[i][j]** is the element in row **i**, column **j** of **b**, whereas **c[i][j]** is the element in row **j**, column **i** of **c**.

2. **InitPerRun** is an initialization routine called before each run of the matrix multiplication code on a given configuration of processors. It simply zeros answer matrix **a**. Strictly speaking, since every element of **a** is written during the matrix multiplication, it is not necessary to zero them between runs. They are zeroed here only to illustrate the use of an initialization routine for **TimeTest**. Note that the rows of the matrix could be zeroed in parallel if the matrix was very big.

3. **DotProduct** is a worker routine called by the **GenOnA** task generator. It computes the vector dot product of row **i** of the **b** matrix and column **j** of the **c** matrix and stores the result in element **a[i][j]** of the result

matrix. It uses a *for* loop to accumulate the individual products in a temporary variable, which it then stores in the result matrix. The variable **bb** is a pointer to row **i** of matrix **b** and variable **cc** is a pointer to column **j** of matrix **c**. Since matrix **b** is scattered by row and matrix **c** is scattered by column, successive elements of the *i*th row of **b** and the *j*th column of **c** can be accessed by incrementing and de-referencing the **bb** and **cc** pointers. Using ***bb** rather than **b[i][j]** avoids accessing **b[i]** (which is constant since *i* does not change) in each iteration of the *for* loop. This helps avoid contention for the memory that holds the **b** vector of pointers. A similar comment applies to the use of **cc**.

4. **Body** is the routine that computes the matrix product. It uses the **GenOnA** task generator to spawn tasks that execute in parallel to compute the individual dot products that make up the result matrix. The generator ensures that **DotProduct** is called for all combinations of **i** and **j** for **i**[0 ≤ **i** Size] and **i**[0 ≤ **j** Size].

5. **PrintAnswer** is the display routine called by **TimeTest**. It prints out part of the result matrix and then calls **TimeTestPrint** to print the runtime, number of processors, and the speedup obtained over one processor by a particular processor configuration.

6. The program starts in **main**. After initializing the Uniform System, **main** asks for the size of the matrices (square matrices are assumed) and stores the reply in the C *global*, process private variable **Size**. Next, it calls **Share** to copy the value of **Size** in all processors that execute any tasks subsequently generated. It then calls **InitProblemOnce** to allocate and initialize the **a**, **b**, and **c** matrices. Finally, it calls **TimeTest** to run the matrix multiplication on the range of processor configurations specified by the user. The routines **InitPerRun**, **Body**, and **PrintAnswer** are called in order by **TimeTest** on each processor configuration, and **Body** is timed for each configuration.

CONVOLUTION

This example illustrates use of the **GenOnIFull** task generator and the Chrysalis block transfer operation. The example is an unoptimized program that performs a convolution operation on an input image to produce

a new output image. Each pixel in the output image is the weighted sum of the corresponding pixel in the input image and pixels adjacent to it. The weighting is specified by a mask. For the example program a specific 3-pixel by 3-pixel mask is used:

```
-1  -1  -1
-1   8  -1
-1  -1  -1
```

The value of each pixel in the output image is eight times the value of the corresponding pixel in the input image minus the values of each of the eight adjacent input image pixels. The output from running the program on a small Butterfly Plus configuration is shown in Figure 3-5.

```
(cluster 14) [c] convolve

Image size = 256

 Please enter start, delta (0=exp), and end for time test: 1 0 8
 Using start = 1, delta = 0, end = 8

[1] time = 50321 ticks =   3.14 sec;  ep =   1.0;  eff = 1.0000
[2] time = 24772 ticks =   1.54 sec;  ep =   2.0;  eff = 1.0156
[4] time = 12407 ticks =    .77 sec;  ep =   4.0;  eff = 1.0139
[8] time = 6413 ticks =    .40 sec;  ep =   7.8;  eff = .9808
(cluster 14) [c]
```

Figure 3-5
Output from Convolution Program

The program parallelizes the convolution operation by computing rows of pixels in the output image in parallel. The **GenOnIFull** task generator is called with a **Range** parameter equal to the number of input image rows minus two to generate the tasks. The top and bottom rows, and the left and right columns are not convolved because they are on the edge of the image, and therefore have insufficient adjacent pixels. The program code is shown in Figure 3-6.

```
/*   Image convolution- unoptimizedexample program   */

#include <us.h>

#define true  1
#define false 0

int N, End;
int * * im, * * an;
int * row, * row_m1, * row_p1, * row_ans;

InitProblemOnce ()
{   int i, j;
    im=(int * *) UsAllocScatterMatrix (N, N, sizeof(int));
    an=(int * *) UsAllocScatterMatrix (N, N, sizeof(int));
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        im[i][j] = i % 2;
    Share (& N); Share (& im); Share (& an);
}

InitforProc(dummy)
    int dummy;
{   End = N - 1;
    row = (int *) malloc (N*sizeof(int));
    row_m1 = (int *) malloc (N*sizeof(int));
    row_p1 = (int *) malloc (N*sizeof(int));
    row_ans = (int *) malloc (N*sizeof(int));
}

DoConvol (dummy ,r)
    int dummy, r;
{   int c;
    if (r & 1)
      r = N-r-2;
    Do_bt (im[r++], row_m1, N*sizeof(int));
    Do_bt (im[r++], row, N*sizeof(int));
    Do_bt (im[r--], row_p1, N*sizeof(int));
    for (c = 1; c < End; c++)
      row_ans[c] =  -row[c-1] + (row[c] << 3) - row[c+1]
                    -row_m1[c-1] - row_m1[c] - row_m1[c+1]
                    -row_p1[c-1] - row_p1[c] - row_p1[c+1];
    Do_bt (row_ans, an[r], N*sizeof(int));
}

FinalforProc ()
{   free (row);
    free (row_m1);
    free (row_p1);
    free (row_ans);
}

Body ()
{   GenOnIFull (InitforProc, DoConvol, FinalforProc, 0, N-2, 0, false);
```

```
}

main ()
{   InitializeUs ();
    printf ("\nImage size = ");  scanf ("%d", &N);
    InitProblemOnce ();
    TimeTest (0, Body, TimeTestPrint);
}
```

Figure 3-6
Program Code for Convolution Program

The program has six routines.

1. **InitProblemOnce** allocates space in globally shared memory for the input (**im**) and output (**an**) images (square images of dimension **N** by **N** are assumed). The images are scattered by row across the memories of the machine. It then generates pixel values for the input image. Next, it uses **Share** to make copies of **N**, **im**, and **an** available to processors used in tasks generated to do the convolution.

2. **InitforProc** is the "initialization" routine passed to **GenOnIFull**. It is called once on each processor that executes tasks generated by **GenOnIFull** before any of the tasks themselves are. **InitforProc** allocates process private space, to be used by **DoConvol**, for four rows of image pixels: **row**, **row_m1**, **row_p1**, and **row_ans**.

3. The **DoConvol** routine computes one row of the output image. Calls to it are generated by the **GenOnIFull** task generator. Before computing output pixels, **DoConvol** makes local copies in process private memory of the pixel values it needs using the Chrysalis **Do_bt** block transfer operation. Each iteration of the *for* loop computes one pixel of the output image. As their values are computed, the output pixels are accumulated in process private memory in **row_ans**. After all have been computed, **row_ans** is copied to the output image by a block transfer.

   The four block transfer operations are motivated by two performance considerations. When referencing many contiguous items, it is more efficient to first use block transfer to make a local copy and then

reference the copied values locally than it is to reference the items one at a time through the Butterfly Plus switch. After a small amount of setup, the block transfer occurs at the full 32-megabit per second rate of the Butterfly Plus switch, whereas individual remote references would be slower, since they incur setup overhead for each remote reference. Using the block transfer operation to put frequently referenced data in local memory is the Butterfly Plus analogy to using *register* variables in C to hold data in faster memory. The second performance consideration is that potential multiprocessor contention for the memory holding the pixel values is reduced, since the single block transfer ties up the memory for less time than the individual remote references.

The *if* statement that changes **r** when it is odd is also motivated by memory contention considerations. Since each instance of **DoConvol** references three rows of the input image, processors working on adjacent rows need to access two rows in common. To reduce the contention that could occur when the processors attempt to block transfer copies of the same rows, processors that are passed an even **r** index use the index directly as a row index, whereas those with an odd **r** index use the index as an offset from the bottom of the image. (As written, the program assumes that **N** is even.) This tends to spread the processors out on the image; processors start both at the top of the image and work down on even rows, and at the bottom of the image and work up on odd rows. This scheme assumes that **GenOnIFull** generates index values in sequence, which in fact it does. Note that there is still a potential for contention with this approach since, for example, the processors working on rows 2 and 4 both access row 3. A slightly more complex scheme would eliminate this contention.

4. **FinalforProc** is the "finalization" routine passed to **GenOnIFull**. It is called on each processor used for tasks generated by **GenOnIFull** after the last such task has been executed on the processor. **FinalforProc** deallocates the space for **row**, **row_m1**, **row_p1**, and **row_ans**.

5. **Body** is the routine timed by **TimeTest**. It uses **GenOnIFull** to generate the tasks that compute rows of output image pixels in parallel.

6. The program starts with **main**, which simply initializes the Uniform System, obtains the size of the image to be convolved from the user, and times the parallel convolution on the processor configurations specified by the user.

Chapter 4

Tuning Programs for Performance

This section presents a few suggestions for tuning the multiprocessor performance of Uniform System programs. Programs are often developed in two stages. The first stage focuses on getting the program to function correctly, and the second stage focuses on achieving an acceptable level of performance by tuning the correctly functioning program. We recommend this two-stage approach to multiprocessor programs: first, get the program to work, and then tune its performance. Although this section is concerned with tuning a program's multiprocessor behavior, the uniprocessor behavior should, of course, also be tuned.

Multiprocessor performance bottlenecks in Uniform System programs may occur for several reasons. Performance bottlenecks can occur if:

- There are insufficient tasks
- The tasks are not long enough
- There is memory contention.

The following paragraphs briefly consider each of these.

INSUFFICIENT TASKS

If there are insufficient tasks, processor starvation occurring as task generators finish up can limit program performance. For example, assume a

system with 128 processors, and an application with 129 tasks, each of which takes about T time units to perform. One processor will perform two tasks and the remaining 127 processors a single task. Therefore, the time to run on 128 processors will be 2T, and the maximum speedup attainable over running on a single processor is the execution time on one processor divided by the execution time on 128 processors, equal to 129T divided by 2T, or 64.5, which results in a processor utilization of only 50%. On a speedup plot (a plot of actual processors versus effective processes) processor starvation effects will show up as a periodic "saw tooth" superimposed on a generally monotonically increasing curve.

The obvious way to remedy this situation is to increase the number of tasks. (In a large application, with many generators active at once, having a relatively small number of tasks for some generators need not be a concern.) In some cases, this is straightforward. For example, if it were necessary to increase the number of tasks in the convolution example of the previous chapter, the number of tasks could be doubled by having each task process only half of an image row.

TASKS NOT LONG ENOUGH

When the tasks are not long enough, poor performance may be due to two factors:

- If task generation time is a significant fraction of total run time, the overhead of the task generator may be unacceptably high. Speedup curves will often be linear in this situation.
- Task generators typically contain an internal "critical" region through which processors must proceed one at a time. For example, **GenOnIndex** must atomically increment a counter to step through the **Range** parameter (see the section entitled "Building a Generator" in Chapter 2).

Critical regions in task generation may limit the number of processors that can be used efficiently. To see this, let T be the time it takes to execute a task. T includes the time to generate the task (**Tgen**) and the time to

perform the task computation (**Twork**).

$$T = T_{gen} + T_{work}$$

T_{gen} is made up of time spent in the critical region (T_{crit}) and in the non-critical region ($T_{noncrit}$). Hence,

$$T = T_{crit} + T_{noncrit} + T_{work}$$

Letting T_{rest} be the sum of $T_{noncrit}$ and T_{work} gives:

$$T = T_{crit} + T_{rest}$$

Since processors must proceed through the critical region serially, the maximum number of processors that can be fully utilized (*i.e.*, used without waiting to proceed though the critical region) is:

$$\text{Max \# procs} = T/T_{crit} = (T_{crit} + T_{rest})/T_{crit} = 1 + T_{rest}/T_{crit}$$

For example, if the critical region is half the total task time, only two processors can be fully utilized. This effect will usually manifest itself as a flattening of the speedup curve, asymptotically approaching T/T_{crit} effective processors.

The effects of both factors can be minimized by increasing task length. The convolution example in the previous chapter is an intermediate version in a sequence that led to an optimized program. An earlier version parallelized the convolution by computing single pixels in the output image in parallel. That task took about 45 microseconds and was far too small, since the critical region in the **GenOnArray** task generator used was about 10 microseconds.

MEMORY CONTENTION

Finally, if there is significant memory contention, processors are forced to proceed serially as they contend for "hot" memory. Hot spots typically show as a flattening of the speedup curve. If the hot spot is severe, the curve may turn down or oscillate. The remedy for this situation is to remove the hot spot. In practice this is usually a two step process: detecting the hot spot, and then removing it.

In some cases hot spots can be identified by studying the code. In other cases the hot spots are not so obvious. In such cases, the Butterfly Plus program profiling utility can be used to determine where, if at all, there is significant memory contention. Consult the *Chrysalis Programmer's Manual* for detailed information on using the profiler.

After hot spots are identified, they must be eliminated. Eliminating them is usually application dependent. However, a few general guidelines can be offered:

- Distribute the program's data across the machine. **UsAllocScatterMatrix** can be used to do this.
- Make local copies of frequently accessed data items. **Share** and **ShareScatterMatrix**, or more specialized code in the per-processor initialization routines of task generators, can be used to do this.
- Distribute references to frequently accessed data across multiple copies of the data. In some cases it may neither be necessary nor practical to have a copy of frequently accessed data on every processor. In many cases, a few copies are sufficient. (If there are **n** copies, processor **p** would access copy **p** modulo **n**.) Of course, if the copied data changes as the computation proceeds and multiple processors need to see the changes, managing the copies can become complex.
- Make local cache copies of data structures before referencing them, as in the convolution example in Chapter 3. **Do_bt** can be used to do this.

Chapter 5

Uniform System Library Routines

This section documents each of the operations supported by the Uniform System. The operations are ordered alphabetically.

- **AbortGen**

```
AbortGen(GenID, code)
UsGenDesc * GenID;
int code;
```

AbortGen aborts an active task generator by preventing the generation of new tasks. Any tasks in progress will run to completion. **GenID** is an identifier for the generator. It must specify an abortable generator. **Code** is returned as the result code for the generator. If **AbortGen** is called more than once for a given generator, the smallest **code** is returned as the generator result code.

- **ActivateGen**

```
UsGenDesc * ActivateGen(Init, Worker, Final, Arg, Range1, Range2, Type,
                        Gen, Async, MaxProcsToUse, Abortable,
                        ResultP, lang)
int (* Init)(), (*Worker)(), (* Final)();
int Arg, Range1, Range2, Type, (* Gen)();
int Async, MaxProcsToUse, Abortable, * ResultP, lang;
```

ActivateGen is the "universal" generator activator procedure. It is called by all of the generators to activate parallel activity. Application programs can use **ActivateGen** directly to build new generators.

Init is the per-processor initialization routine, **Worker** is the task worker routine, and **Final** is the per-processor post-processing routine. **Arg** is a pointer to a data structure, which is passed to the **Init**, **Worker**, and **Final** routines. **Type** must be set to GENERATOR, and **Range1** and **Range2** are integers. **GenProc** is a task generation routine described in more detail below. The **Async** parameter specifies whether the generator is to be synchronous or asynchronous. It should be set to *TRUE* for synchronous or *FALSE* for asynchronous. **MaxProcsToUse** specifies the maximum number of processors that the generator can generate tasks for. To use all available processors, **MaxProcsToUse** should be set to **TotalProcsAvailable()**. The **Abortable** parameter determines whether the generator is to be abortable. It should be set to *TRUE* if the generator is to be abortable or *FALSE* if the generator is not to be abortable. **ResultP** is a pointer used when **Abortable** is true. It specifies where to store the generator "result code" if the generator is aborted, so the generator activator routine can find it. Finally, **Language** specifies the programming language from which the generator is being called (0 for the C language, 1 for FORTRAN).

GenProc is the task generation routine. It is of the form:

```
GenProc(TD);
UsGenDesc * TD;
```

where **TD** is a pointer to a task descriptor data structure in globally shared memory. The task descriptor data structure, **UsGenDesc**, which is defined in *us.h*, is:

```
typedef struct UsGenDescStruct
{   int       id;
    short *   completion_location;
    struct UsGenDescStruct * prev_ptr;
    struct UsGenDescStruct * next_ptr;
    int       in_hash_table;
    char *    currentShare;
    int       end;
    short     started;
    short     us_lock;
    short     retcode;
# define genEXHAUSTED -1
    short     endlock;
    int       (* init)();
    int       (* gen)();
    int       (* final)();
    int       arg;
    int       (* call)();
```

```
    int       range;
    int       range2;
    int       abortable;
    short     max_procs_to_use;
    short     language;
# define C_CALLING 0
# define FORTRAN_CALLING 1
    union
    {    longLong;
    short Short;
    }         index;
    union
    {    unsigned long Long;
    unsigned short Short;
    }         index2;
    short     lock;
} UsGenDesc;
```

The fields **id** through **language** are used internally by the Uniform System generator mechanism. The **Worker, Init, Final, GenProc, Arg, Range1, Range2, MaxProcsToUse** and **Language** parameters of **ActivateGen** are used to initialize the **call, init, final, gen, arg, range, range2, max_procs_to_use, abortable,** and **language** fields of the data structure, and the other fields between **id** and **language** are used by the generator mechanism for internal bookkeeping. The remaining fields (**index** through **lock**) are initialized to 0, and are available for use by the **GenProc** routine for any generator-specific bookkeeping associated with generating tasks.

After **ActivateGen** initializes the task descriptor data structure, it makes the descriptor accessible to other processors. If **Async** is *TRUE*, **ActivateGen** then returns its caller a pointer to the task descriptor data structure; otherwise, the processor running **ActivateGen** calls the **GenProc** task generation procedure. That processor, and others as they become free, use the task generator descriptor (**TD**) and the **GenProc** task generation procedure to generate and execute calls to the **Worker** procedure.

- **AsynchGenOnA**

```
UsGenDesc *
AsynchGenOnA(Worker, Range1, Range2)
int (* Worker)();
int Range1, Range2;
   ...
Worker(0, index1, index2, GenID)
int index1, index2;
```

AsyncGenOnA is the asynchronous version of **GenOnA**. It is equivalent to:

```
AsyncGenOnAFull(0, Worker, 0, 0, Range1, Range2, 0, FALSE)
```

- **AsyncGenOnAAbortable**

```
UsGenDesc *
AsyncGenOnAAbortable(Worker, Range1, Range2)
int (* Worker)();
int Range1, Range2;
   ...
Worker(0, index1, index2, GenID)
int index1, index2;
UsGenDesc * GenID;
```

AsyncGenOnAAbortable is the asynchronous version of **GenOnAAbortable**. It is equivalent to:

```
AsyncGenOnAFull(0, Worker, 0, 0, Range1, Range2, 0, TRUE)
```

- **AsyncGenOnAFull**

```
UsGenDesc *
AsyncGenOnAFull(Init, Worker, Final, Arg, Range1, Range2, Limited, Abortable)
int (*Init)(), (* Worker)(), (* Final)();
int Arg, Range1, Range2, Limited, Abortable;
   ...
Worker(0, index1, index2)          /* If Abortable = FALSE */
int index1, index2
    or
Worker(0, index1, index2, GenID)   /* If Abortable = TRUE */
int index1, index2;
UsGenDesc * GenID
   ...
Init(Arg)
int Arg;
   ...
Final(Arg)
int Arg;
```

AsyncGenOnAFull is the asynchronous version of **GenOnAFull**. It returns to the caller as soon as the task generator is activated, enabling the caller to work on other things while the tasks are executed. **AsynchGenOnAFull** returns a generator handle that can be used with the **WorkOn** or **WaitForTasksToFinish** routines. See the description of **GenOnAFull** for an explanation of the parameters.

- **AsyncGenOnALimited**

```
UsGenDesc *
AsyncGenOnALimited(Worker, Range1, Range2, MaxProcsToUse)
int (* Worker)();
int Range1, Range2, MaxProcsToUse;
   ...
Worker(0, index1, index2)
int index1, index2;
```

AsyncGenOnALimited is the asynchronous version of **GenOnALimited**. It is equivalent to:

```
AsyncGenOnAFull(0, Worker, 0, 0, Range1, Range2, MaxProcsToUse, FALSE)
```

- **AsyncGenOnHA**

```
UsGenDesc *
AsyncGenOnHA(Worker, Range1, Range2)
int (* Worker)();
int Range1, Range2;
   ...
Worker(Arg,index1,  index2)
int Arg, index1, index2;
```

AsyncGenOnHA is the asynchronous version of **GenOnHA**. It is equivalent to:

```
AsyncGenOnHAFull(0, Worker, 0, 0, Range1, Range2, 0, FALSE)
```

- **AsyncGenOnHAAbortable**

```
UsGenDesc *
AsyncGenOnHAAbortable(Worker, Range1, Range2)
int (* Worker)();
int Range1, Range2;
   ...
Worker(0, index1, index2, GenID)
int index1, index2;
UsGenDesc * GenID;
```

AsyncGenOnHAAbortable is the asynchronous version of **GenOnHAAbortable**. It is equivalent to:

```
AsyncGenOnHAFull(0, Worker, 0, 0, Range1, Range2, 0, TRUE)
```

- **AsyncGenOnHAFull**

```
UsGenDesc *
AsyncGenOnHAFull(Init, Worker, Final, Arg, Range1,
                 Range2, Limited, Abortable)
int (* Init)(), (* Worker)(), (* Final)();
```

```
int Arg, Range1, Range2, Limited, Abortable;
   ...
Worker(0, index1, index2)          /* If Abortable = FALSE */
int index1, index2
     or
Worker(0, index1, index2, GenID)   /* If Abortable = TRUE */
int index1, index2;
UsGenDesc * GenID
   ...
Init(Arg)
int Arg;
   ...
Final(Arg)
int Arg;
```

AsyncGenOnHAFull is the asynchronous version of **GenOnHAFull**. It returns to the caller as soon as the task generator is activated, enabling the caller to work on other things while the tasks are executed. **AsyncGenOnHAFull** returns a generator handle that can be used with the **WorkOn** or **WaitForTasksToFinish** routines. See the description of **GenOnAFull** for an explanation of the parameters.

- **AsyncGenOnHALimited**

```
UsGenDesc *
AsyncGenOnHALimited(Worker, Range1, Range2, MaxProcsToUse)
int (* Worker)();
int Range1, Range2, MaxProcsToUse;
   ...
Worker(Arg,index1,  index2)
int Arg, index1, index2;
```

AsyncGenOnHALimited is the asynchronous version of **GenOnHALimited**. It is equivalent to:

```
AsyncGenOnHAFull(0, Worker, 0, 0, Range1, Range2, MaxProcsToUse, FALSE)
```

- **AsyncGenOnI**

```
UsGenDesc *
AsyncGenOnI(Worker, Range)
int (* Worker)();
int Range;
   ...
Worker(0,index)
int index;
```

AsyncGenOnI is the asynchronous version of **GenOnI**. Control is returned to the process that executed **AsyncGenOnI** without waiting for the tasks to complete. It is equivalent to:

```
AsyncGenOnIFull(0, Worker, 0, 0, Range, 0, FALSE)
```

• AsyncGenOnIAbortable

```
UsGenDesc *
AsyncGenOnIAbortable(Worker, Range)
int (* Worker)();
int Range;
   ...
Worker(0,index,GenID)
int index;
UsGenDesc * GenID;
```

AsyncGenOnIAbortable is the asynchronous version of **GenOnIAbortable**. Control is returned to the process that executed **AsyncGenOnIAbortable** without waiting for the tasks to complete. It is equivalent to:

```
AsyncGenOnIFull(0, Worker, 0, 0, Range, 0, TRUE)
```

• AsyncGenOnIFull

```
UsGenDesc *
AsyncGenOnIFull(Init, Worker, Final, Arg, Range, Limited, Abortable)
int (* Init)(), (* Worker)(), (* Final)();
int Arg, Range, Limited, Abortable;
   ...
Worker(0, index)          /* If Abortable = FALSE */
int index
    or
Worker(0, index, GenID)   /* If Abortable = TRUE */
int index;
UsGenDesc * GenID
   ...
Init(Arg)
int Arg;
   ...
Final(Arg)
int Arg;
```

AsyncGenOnIFull is the asynchronous version of **GenOnIFull**. It returns to the caller as soon as the task generator is activated, enabling the caller to work on other things while the tasks are executed. **AsyncGenOnIFull** returns a generator handle that can be used with the **WorkOn** or **WaitForTasksToFinish** routines. See the description of **GenOnIFull** for an explanation of the parameters.

• AsyncGenOnILimited

```
UsGenDesc *
```

```
AsyncGenOnILimited(Worker, Range, MaxProcsToUse)
int (* Worker)();
int Range, MaxProcsToUse;
   ...
Worker(0,index)
int index;
```

AsyncGenOnILimited is the asynchronous version of **GenOnILimited**. It is equivalent to:

```
AsyncGenOnIFull(0, Worker, 0, 0, Range, MaxProcsToUse, FALSE)
```

- **Atomic_add_long**

```
Atomic_add_long(loc, val)
int * loc, val;
```

Atomic_add_long atomically adds **val** to the location addressed by **loc**. It is similar to the Chrysalis 16-bit **Atomic_add** operation except that it operates on 32-bit quantities and does not support the fetch part of the "fetch and add" function provided by **Atomic_add**.

Atomic_add_long is atomic only with respect to other **Atomic_add_long** calls. In particular, execution of a read operation can be interleaved with an **Atomic_add_long** operation in a way that returns an inconsistent result to the read. This can occur if the high-order 16 bits returned by the read are obtained after the low-order 16 bits are incremented by the **Atomic_add_long**, but before the carry (if any) propagates to the higher order bits.

- **BEGIN_SHARED_DECL and END_SHARED_DECL**

```
BEGIN_SHARED_DECL
    ...
    (normal C declarations;)
    ....
END_SHARED_DECL;
```

BEGIN_SHARED_DECL and **END_SHARED_DECL** are macros that declare variables to be globally shared among all of the processors. They create a structure that contains all the variables. Space is allocated for the structure via the macro **MakeSharedVariables**. Variables in the structure are referenced via the macro **SHARED**.

Only one BEGIN_SHARED_DECL and END_SHARED_DECL declaration can appear in a Uniform System program. All variables declared via BEGIN_SHARED_DECL and END_SHARED_DECL are allocated on the same physical memory. In some situations this may lead to memory contention.

- **ConfigureUs**

```
ConfigureUs(Spec, n)
int * Spec, n;
```

ConfigureUs can be used prior to calling **InitializeUs** to specify values for configuration parameters that differ from the default values used by **InitializeUs**. **Spec** is an array of integers that specifies the configuration parameters to be set; it contains *n* parameter specification blocks. Each parameter specification block contains an integer configuration code that serves to identify the parameter being set, followed by one or more integers that specify the value for the parameter. See **SetUsConfig** for a list of the configuration codes currently defined.

- **DistinctMemoriesAvailable**

```
DistinctMemoriesAvailable()
```

DistinctMemoriesAvailable returns the number of memories available for use by the application program. This number is usually the same as **TotalProcsAvailable**, but there are cases where it will be a smaller number because memory cannot be obtained on a particular processor node.

- **FreeAll**

```
FreeAll()
```

FreeAll reinitializes the Uniform System memory allocator by freeing *all* globally allocated storage, including memory allocated by any of the allocators.

- **GenOnA**

```
GenOnA(Worker, Range1, Range2)
int (* Worker)();
int Range1, Range2;
```

```
   ...
Worker(0, index1, index2)
int index1, index2;
```

GenOnA generates tasks that execute the worker routine in parallel for all combinations of two ranges of values. The worker routine will be executed **Range1** × **Range2** times. The indexes are the specific values given to the **worker** routine each time it is executed. **Index1** ranges from 0 to (**Range1**−1). **Index2** ranges from 0 to (**Range2**−1). The processor that invokes **GenOnA**, and possibly other processors, will execute the generated tasks. When **GenOnA** returns, all generated tasks will have finished. A call to **GenOnA** is equivalent to:

```
GenOnAFull(0, Worker, 0, 0, Range1, Range2, 0, FALSE)
```

- **GenOnAAbortable**

```
GenOnAAbortable(Worker, Range1, Range2)
int (* Worker)();
int Range1, Range2;
   ...
Worker(0, index1, index2, GenID)
int index1, index2;
UsGenDesc * GenID;
```

GenOnAAbortable is the abortable version of **GenOnA**. **GenID** is an identifier for the task generator. It is used with the **AbortGen** routine to abort it. **GenOnAAbortable** returns a value that indicates whether **AbortGen** was used to abort the generator. It is equivalent to:

```
GenOnAFull(0, Worker, 0, 0, Range1, Range2, 0, TRUE)
```

- **GenOnAFull**

```
GenOnAFull(Init, Worker, Final, Arg, Range1, Range2, Limited, Abortable)
int (* Init)(), (* Worker)(), (* Final)();
int Arg, Range1, Range2, Limited, Abortable;
   ...
Worker(0, index1, index2)          /* If Abortable = FALSE */
int index1, index2
    or
Worker(0, index1, index2, GenID)   /* If Abortable = TRUE */
int index1, index2;
UsGenDesc * GenID
   ...
Init(Arg)
int Arg;
   ...
Final(Arg)
```

```
int Arg;
```

GenOnAFull generates tasks on an array. It is the complete version of the **GenOnA** generator family. The **Abortable** parameter determines whether the generator is abortable. The parameter should be set to *FALSE* if the generator is not abortable or *TRUE* if the generator is abortable. If the generator is not abortable, the worker routine is:

```
Worker(Arg,index1,  index2)
int Arg, index1, index2;
```

If the generator is abortable, the worker routine is:

```
Worker(Arg,index1,  index2, GenID)
int Arg, index1, index2;
UsGenDesc * GenID;
```

The **Init** routine is called once on each processor used to execute the generated tasks. It is called before the **Worker** routine runs for the first time on that processor. The **Final** routine is called once on each processor used to execute the generated tasks, after the **Worker** routine runs for the last time on that processor. The **Limited** parameter controls the number of processors used by the generator. If **Limited** is set to 0 or −1, the generator may use all available processors. If **Limited** is set to a positive value, the generator will use no more than that number of processors. It may use less than the maximum number of processors.

If **GenOnAFull** returns without being aborted, all the generated tasks have finished and the value **genEXHAUSTED** is returned. If the **Abortable** parameter was set to *TRUE* and the generator was aborted, some of the tasks may not have been performed and the code that was passed to **AbortGen** is returned.

- **GenOnALimited**

```
GenOnALimited(Worker, Range1, Range2, MaxProcsToUse)
int (* Worker)();
int Range1, Range2, MaxProcsToUse;
   ...
Worker(0, index1, index2)
int index1, index2;
```

GenOnALimited is the limited version of **GenOnA**. It is equivalent to:

```
GenOnAFull(0, Worker, 0, 0, Range1, Range2, MaxProcsToUse, FALSE)
```

• GenOnHA

```
GenOnHA(Worker, Range1, Range2)
int (* Worker)();
int Range1, Range2;
   ...
Worker(Arg,index1, index2)
int Arg, index1, index2;
```

GenOnHA generates tasks that execute the worker routine in parallel for certain combinations of two ranges of values. The combinations of values span the half array below the diagonal of the array. The indexes are the specific values given to the **Worker** routine each time it is executed. The indexes range as follows:

```
index2 = 0,   index1 = 1,...,(Range1-1)
index2 = 1,   index1 = 2,...,(Range1-1)
   ...
index2 = R-2, index1 = (R-1),...,(Range1-1)
```

where **R** is the lesser of **Range1** and **Range2**. The processor that invokes **GenOnHA**, and possibly other processors, will execute the generated tasks. When **GenOnHA** returns, all of the generated tasks will have finished. **GenOnHA** is equivalent to:

```
GenOnHAFull(0, Worker, 0, 0, Range1, Range2, 0, FALSE)
```

• GenOnHAAbortable

```
GenOnHAAbortable(Worker, Range1, Range2)
int (* Worker)();
int Range1, Range2;
   ...
Worker(0, index1, index2, GenID)
int index1, index2;
UsGenDesc * GenID;
```

GenOnHAAbortable is the abortable version of **GenOnHA**. **GenID** is an identifier for the task generator. It is used with **AbortGen** to abort it. **GenOnHAAbortable** returns a value that indicates whether **AbortGen** aborted the generator. It is equivalent to:

```
GenOnHAFull(0, Worker, 0, 0, Range1, Range2, 0, TRUE)
```

• GenOnHAFull

```
GenOnHAFull(Init, Worker, Final, Arg, Range1, Range2, Limited, Abortable)
int (* Init)(), (* Worker)(), (* Final)();
int Arg, Range1, Range2, Limited, Abortable;
```

```
   ...
Worker(0, index1, index2)          /* If Abortable = FALSE */
int index1, index2
    or
Worker(0, index1, index2, GenID)   /* If Abortable = TRUE */
int index1, index2;
UsGenDesc * GenID
   ...
Init(Arg)
int Arg;
   ...
Final(Arg)
int Arg;
```

GenOnHAFull generates tasks on half of an array. It is the complete version of the **GenOnHA** generator family. The **Abortable** parameter determines whether the generator is abortable. The parameter should be set to *FALSE* if the generator is not abortable or *TRUE* if the generator is abortable. If the generator is not abortable, the worker routine is:

```
Worker(Arg,index1,  index2)
int Arg, index1, index2;
```

If the generator is abortable, the worker routine is:

```
Worker(Arg,index1,  index2, GenID)
int Arg, index1, index2;
UsGenDesc * GenID;
```

The **Init** routine is called once on each processor used to execute the generated tasks. It is called before the **Worker** routine runs for the first time on that processor. The **Final** routine is called once on each processor used to execute the generated tasks, after the **Worker** routine runs for the last time on that processor. The **Limited** parameter controls the number of processors used by the generator. If **Limited** is set to 0 or −1, the generator may use all available processors. If **Limited** is set to a positive value, at most, the generator will use that number of processors. It may use less than the maximum number of processors.

If **GenOnHAFull** returns without being aborted, all the generated tasks have finished and the value **genEXHAUSTED** is returned. If the **Abortable** parameter was set to *TRUE* and the generator was aborted, some of the tasks may not have been performed and the code that was passed to **AbortGen** is returned.

- **GenOnHALimited**

```
GenOnHALimited(Worker, Range1, Range2, MaxProcsToUse)
int (* Worker)();
int Range1, Range2, MaxProcsToUse;
   ...
Worker(Arg,index1,  index2)
int Arg, index1, index2;
```

GenOnHALimited is the limited version of **GenOnHA**. It is equivalent to:

```
GenOnHAFull(0, Worker, 0, 0, Range1, Range2, MaxProcsToUse, TRUE)
```

- **GenOnI**

```
GenOnI(Worker, Range)
int (* Worker)();
int Range;
   ...
Worker(0,index)
int index;
```

GenOnI generates tasks that execute a worker routine for a range of values. **Range** is the number of times that the **Worker** routine will be executed. **index** is the specific value given to the **Worker** routine each time it is executed. **index** ranges from 0 to (**Range**–1). The processor that invokes **GenOnI**, and possibly other processors, will execute the generated tasks. When **GenOnI** returns, all of the generated tasks will have finished. A call to **GenOnI** is equivalent to:

```
GenOnIFull(0, Worker, 0, 0, Range, 0, FALSE)
```

- **GenOnIAbortable**

```
GenOnIAbortable(Worker, Range)
int (* Worker)();
int Range;
   ...
Worker(0,index,GenID)
int index;
UsGenDesc * GenID;
```

GenOnIAbortable is the abortable version of **GenOnI**. **GenID** is an identifier for the task generator. It is used with the **AbortGen** routine to abort it. **GenOnIAbortable** returns a value that indicates whether **AbortGen** was used to abort the generator. **GenOnIAbortable** is equivalent to:

```
GenOnIFull(0, Worker, 0, 0, Range, 0, TRUE)
```

• GenOnIFull

```
GenOnIFull(Init, Worker, Final, Arg, Range, Limited, Abortable)
int (*Init)(), (* Worker)(), (* Final)();
int Arg, Range, Limited, Abortable;
   ...
Worker(0, index)          /* If Abortable = FALSE */
int index
    or
Worker(0, index, GenID)   /* If Abortable = TRUE */
int index;
UsGenDesc * GenID
   ...
Init(Arg)
int Arg;
   ...
Final(Arg)
int Arg;
```

GenOnIFull generates tasks on an index. It is the complete version of the **GenOnI** generator family. The **Abortable** parameter determines whether the generator is abortable. The parameter should be set to *FALSE* if the generator is not abortable or *TRUE* if the generator is abortable. If the generator is not abortable, the worker routine is:

```
Worker(Arg,index)
int Arg, index;
```

If the generator is abortable, the worker routine is:

```
Worker(Arg,index,  GenID)
int Arg, index;
UsGenDesc * GenID;
```

The **Init** routine is called once on each processor used to execute the generated tasks. It is called before the **Worker** routine runs for the first time on that processor. The **Final** routine is called once on each processor used to execute the generated tasks, after the **Worker** routine runs for the last time on that processor. The **Limited** parameter controls the number of processors used by the generator. If **Limited** is set to 0 or –1, the generator may use all available processors. If **Limited** is set to a positive value, at most, the generator will use that number of processors. It may use less than the maximum number of processors.

If **GenOnIFull** returns without being aborted, all the generated tasks have finished and the value **genEXHAUSTED** is returned. If the **Abortable** parameter was set to *TRUE* and the generator was aborted, some of the tasks may not have been performed and the code that was passed to

AbortGen is returned.

- **GenOnILimited**

```
GenOnILimited(Worker, Range, MaxProcsToUse)
int (*worker)();
int Range;
   ...
Worker(0,index)
int index;
```

GenOnILimited is the limited version of **GenOnI**. It will only generate tasks for a limited number of processors. **MaxProcsToUse** is the number of processors to use. **GenOnILimited** is equivalent to:

```
GenOnIFull(0, Worker, 0, 0, Range, MaxProcsToUse, FALSE)
```

- **GenTaskForEachProc**

```
GenTaskForEachProc(Worker, Arg)
int (* Worker)();
int Arg;
   ...
Worker(Arg)
int Arg;
```

GenTaskForEachProc generates exactly one call on the worker routine for every processor.

- **GenTaskForEachProcLimited**

```
GenTaskForEachProcLimited(Worker, Arg, NProcs)
int (* Worker)();
int Arg, NProcs;
   ...
Worker(Arg)
int Arg;
```

GenTaskForEachProcLimited generates exactly one call on the worker routine for every processor. The number of processors to use is limited to **NProcs**. If **ProcsInUse()** is less than **NProcs**, this call will hang.

- **GenTasksFromList**

```
GenTasksFromList(Routine_List, Arg_List, n)
int * (* RoutineList)();
int * Arg_List;
int n;
```

GenTasksFromList generates *n* tasks from a list of tasks. **Routine_List**

is the list of routines to be executed, $r_1,...,r_n$. Arg_List is the list of arguments to the routines, $arg_1,...,arg_n$. There is one argument for each routine. The first task is of the form $r_1(arg_1)$. The C notation for the first task is:

```
(* Routine_List [0]) (Arg_List[0]);
```

- **GetRtc**

```
GetRtc()
```

GetRtc returns the time since the system was last reset in units of 62.5 microseconds.

- **InitializeUs**

```
InitializeUs()
```

InitializeUs initializes the Uniform System. It creates and starts a Uniform System process on every available processor and sets up the memory that is globally shared among all Uniform System processes. It also initializes the Uniform System storage allocator. **InitializeUs** must be called once in every program. It is called before other Uniform System routines except for **ConfigureUs** and **SetUsConfig**.

- **MakeSharedVariables**

```
MakeSharedVariables;
```

MakeSharedVariables is a macro that allocates space in globally shared memory for the structure created by BEGIN_SHARED_DECL and END_SHARED_DECL. It makes the location of the structure known to other processors. **MakeSharedVariables** must be called after **InitializeUs()** and before any of the shared variables are referenced.

- **MemoriesAvailable**

```
MemoriesAvailable()
```

MemoriesAvailable returns the amount of globally-shared memory available to the application program in units of 64 kilobytes.

- **PhysProcToUsProc**

```
PhysProcToUsProc(physproc)
```

```
int physproc;
```

PhysProcToUsProc returns the Uniform System virtual processor number corresponding to the physical processor number, **physproc**.

- **ProcsInUse**

```
ProcsInUse()
```

ProcsInUse returns the number of processors available to an application program, excluding any processors that were removed by **TimeTest** or **TimeTestFull**.

- **RefreshLocalShareValues**

```
RefreshLocalShareValues ()
```

The Uniform System **Share** mechanism propagates copies of process private data to all processes. It facilitates program initialization by making it relatively easy to propagate the values of variables set during program initialization to all processors. Copies of such data normally propagate to a process automatically. This occurs within the Uniform System generator mechanism, which checks to see whether there are any new values to be copied before a process gets its first task from a new generator.

Although automatic propagation of process private data is adequate when the **Share** mechanism is used to propagate initialized values of variables that have been allocated in non-shared memory, it is often inadequate when the **Share** mechanism is used in other ways. A process can use **RefreshLocalShareValues** to refresh its copies of any variables whose values may have been updated and propagated by other processes via the **Share** mechanism.

- **SetUsConfig**

```
SetUsConfig(code,  value)
int code, value;
```

SetUsConfig can be used prior to calling **InitializeUs** to specify a value for a configuration parameter that differs from the default value used by **InitializeUs**. **code** is the configuration code for the parameter name. **value** is the integer value of the parameter.

The following configuration codes are defined:

configProcs Specifies the number of processors to include in the Uniform System configuration. The number should be an integer less than or equal to the number of nodes in the cluster. If **configProcs** is set greater than the number of available nodes, the Uniform System uses only the nodes available to it.

configSuppressInitMsgs Specifies whether to print messages that report the progress of **InitializeUs**: 1 means suppress the messages, 0 means print the messages. The default is 1.

configTimeTestViaReinit

Specifies behavior of the **TimeTest** mechanism (see "Measuring Your Program"). 1 means completely reinitialize the Uniform System for each configuration timed. This ensures that only the resources of the processors being timed are used; in particular, only the memory of those processors is used. 0 means use the Uniform System as is, by "diverting" enough processors to an idle loop in order to time a given processor configuration. This means that only the CPU resources of the processors in the configuration are used, but the memory resources of all processors, including those that have been diverted, may be used. The default is 0. This parameter may be specified anytime before calling **TimeTest**.

configAllocAcross64K Specifies whether Uniform System memory allocators (see "Memory Allocators") may allocate blocks of memory that cross 64-kilobyte boundaries in a process address space. Early versions of the Uniform System would

not allocate blocks that cross 64-kilobyte boundaries. 1 means allow allocation across 64-kilobyte blocks; 0 means don't allow allocation across 64-kilobyte blocks. The default is 1.

configWarningOnShareFail
Specifies behavior on a failure of the **Share** mechanism (see "Copying Process Private Data"); 1 means print a warning message on a **Share** failure; 0 means don't print a warning message on a **Share** failure. The default is 1.

configStopOnShareFail Specifies behavior on a failure of the **Share** mechanism (see "Copying Process Private Data"); 1 means suspend process on a **Share** failure; 0 means allow process to continue execution on a **Share** failure. The default is 1.

configMemObjsFree Specifies the number of 64-kilobyte memory objects to leave free on each processor node. Setting this configuration option overides any previous use of **configMaxMemObjs**.

configMaxMemObjs Specifies the maximum number of 64-kilobyte memory objects to obtain from each processor when making the Uniform System shared memory. Setting this configuration option overides any previous use of **configMemObjsFree**.

configObjsRetRoot During **InitializeUs**, the Uniform System obtains memory in 64-kilobyte blocks to be used to build its shared address space. It obtains as many 64-kilobyte blocks as it can on each processor node in the configuration. It then returns some 64-kilobyte blocks on each node to allow operations requiring memory to occur on the nodes; for example, running the various Chrysalis utilities such as **ps** and

showmem require memory. This configuration code is used to specify the number of 64-kilobyte blocks the Uniform System should return for the processor node on which the Uniform System program is started (the root processor). The parameter is interpreted only if **configMaxMemObjs** and **configMemObjsFree** have not been specified. The default is 2.

configObjsRetChild Similar to **configObjsRetRoot**, but specifies the number of 64-kilobyte blocks to be returned for child processor nodes. The parameter is interpreted only if **configMaxMemObjs** and **configMemObjsFree** have not been specified. The default is 2.

configMaxSars Specifies the maximum size of the shared portion of the process address space in terms of 64-kilobyte blocks or "segments."[1] This parameter should be an integer greater than 15. The default is 237.

configTotalSars Specifies the maximum size of the process address space in terms of 64-kilobyte blocks or "segments."[2] This includes the space consumed by the program, the stack, process private data, and Uniform System shared memory. The default value for this parameter is 256. The maximum allowable value for this

1. Prior to the Butterfly Plus, Butterfly processor nodes contained a custom memory management unit that made use of registers called Segment Attribute Registers (SARs). On those machines, the Uniform System used one SAR for each 64-kilobyte segment of the shared portion of the process address space.
2. It was useful to use this configuration code prior to the Butterfly Plus to reduce the number of SARs required by a program, because SARs were a relatively scarce processor node resource. Since Butterfly Plus processor nodes do not contain SARs, Butterfly Plus programs should not need to use this configuration code.

parameter is also 256. This value restricts Uniform System programs to a 16-megabyte address space.

- **Share**

```
int
Share(N)
int * N;
```

Share passes the value pointed to by **N** to all processors used to execute tasks generated subsequently. **N** must point to a variable allocated in process private memory and declared to be a global or a static. In addition, the variable pointed to by **N** must be four bytes in size. **Share** causes the value pointed to by **N** (in the processor invoking **Share** at the time **Share** is invoked) to be copied into the location specified by **N** in each processor used to perform tasks generated by task generators activated subsequent to the call of **Share**.

- **ShareBlk**

```
int
ShareBlk(N,nbytes)
int * N;
int nbytes;
```

ShareBlk passes the block of data of **nbytes** bytes pointed to by **N** to all processors used to execute tasks generated subsequently. **N** must point to a variable allocated in process private memory and declared to be a global or a static. **ShareBlk** causes the block of data pointed to by **N** (in the processor invoking **ShareBlk** at the time **ShareBlk** is invoked) to be copied into the location beginning at **N** in each processor used to perform tasks generated by task generators activated subsequent to the call of **ShareBlk**.

- **SharePtrAndBlk**

```
int
SharePtrAndBlk(P, nbytes)
int * * P;
int nbytes;
```

SharePtrAndBlk passes the pointer pointed to by **P**, and the block of data of **nbytes** bytes to which it points, to all processors used to execute tasks generated subsequently. **P** must point to a pointer variable allocated in

process private memory and declared to be a global or a static. **SharePtrAndBlk** makes a copy of the pointer pointed to by **P** and the block of data to which it points (in the processor invoking **SharePtrAndBlk** at the time the routine is invoked) for each processor used to perform tasks generated by task generators activated subsequent to the call of **SharePtrAndBlk**. A block of storage is allocated in the memory of the processor and the block of data pointed to by the pointer pointed to by **P** is copied into the newly allocated storage block. A pointer to the newly-allocated storage block is stored in the location pointed to by **P**. For example, to share a pointer and block:

```
int *p;
p=(int *)UsAlloc(sizeof(data block that p points to));
     (fill in block of data)
SharePtrAndBlk(&p,sizeof(data block that p points to));
```

- **ShareScatterMatrix**

```
int
ShareScatterMatrix(P, nrows)
int * * * P;
int nrows;
```

P points to a global or static variable allocated by:

```
UsAllocScatterMatrix(nrows, ncols, element_size)
```

ShareScatterMatrix makes a copy of the vector of row pointers allocated by **UsAllocScatterMatrix** in the memory of each processor used to execute tasks generated subsequently. It then sets the location pointed to by **P** to point to that copy. **ShareScatterMatrix** is careful to make its copies from other copies as well as from the original in order to avoid memory contention on larger configurations.

- **SHARED**

SHARED is a macro used to access variables in the structure created by the **BEGIN_SHARED_DECL** and **END_SHARED_DECL** macros. For example, if **N** has been declared in this way, it may be referenced as **SHARED N**:

```
BEGIN_SHARED_DECL
     int N;
END_SHARED_DECL;
main ()
   . {InitializeUs();
```

```
    MakeSharedVariables;
      ...
    SHARED N = 5;
      ...
}
```

Before a variable can be referenced in this way, space for it must be allocated using **MakeSharedVariables**.

- **TimeTest**

```
TimeTest(Init, Execute, PrintResults)
int (* Init)(), (* Execute)(),(*  PrintResults)();
```

TimeTest times the execution of the routine **Execute** on various processor configurations as specified by the user from the keyboard. **TimeTest** runs the routines **Init**, **Execute**, and **PrintResults** in sequence on each of the processor configurations specified. It times only the **Execute** routine, and passes the execution time, the number of processors, and the effective number of processors to the specified **PrintResults** routine:

```
PrintResults(time, procs, effprocs)
int time, procs;
float effprocs;
```

The effective number of processors is a *float* equal to (time 1 proc)/(time *n* procs). This is a good measure of the speedup the **Execute** routine achieves over one processor when *n* processors are used. If the first test run uses more than one (=*k*) processors, then the effective number of processors is *k*(time *k* proc)/(time *n* procs).

The **PrintResults** routine is specified by the application program. The Uniform System Library contains a routine that can be used for this purpose, or the user can supply his own routine.

TimeTest asks the user to specify the processor configurations to be used by specifying a **start** configuration, a step (**delta**), and an **end** configuration. The first run uses **start** processors, the next uses **start** + **delta** processors, and so forth, up to the final run, which uses **end** processors. If **start** (or **end**) is zero, the test is run from (to) the end of the range of available processors. In particular, it is run for the limiting processor case whether or not it is in the normal progression specified by **delta**. If **delta** is specified to be zero, the number of processors used increases by

powers of two (1, 2, 4, 8, etc). The rules for **start** and **end** still apply.

- **TimeTestFull**

```
TimeTestFull(Init, Execute, PrintResults, start, delta, end)
int (*Init)(), (* Execute)(),(*  PrintResults)();
int start, delta, end;
```

TimeTestFull is similar to **TimeTest**. It differs only in that it accepts the **start**, **delta**, and **end** parameters that specify the processor configurations to be timed, rather than asking for them from the keyboard. If the **delta** specified is negative, **TimeTestFull** asks the user to supply values for **start**, **delta**, and **end** at the start of the run.

- **TimeTestPrint**

```
TimeTestPrint(runtime, procs, effprocs)
int runtime, procs;
float effprocs;
```

TimeTestPrint is used with **TimeTest** or **TimeTestFull** to print the timing results for a particular processor configuration. It prints the execution time, the number of processors used, the effective number of processors utilized (the speedup achieved over one processor), and the efficiency with which processors were used for the given processor configuration. **TimeTestPrint** outputs this information in the format:

```
[procs] time = runtime ticks = S sec; ep = effprocs; eff = E
```

where E = *effprocs/procs*. (See **TimeTest** and **TimeTestFull**.)

- **TotalProcsAvailable**

```
TotalProcsAvailable()
```

TotalProcsAvailable returns the total number of processors available to the application program. The value returned includes any processors that may have been removed by **TimeTest** or **TimeTestFull**.

- **UsAlloc**

```
char * UsAlloc (nbytes)
unsigned long nbytes;
```

UsAlloc allocates a block of storage of **nbytes** in globally shared memory. The block is allocated from the memory with the most free space.

- **UsAllocAndReportC**

```
char * UsAllocAndReportC (usproc, wherep, nbytes, class)
int usproc;
int * wherep;
unsigned long nbytes;
int class;
```

UsAllocAndReportC attempts to allocate a block of size **nbytes** on a processor in the specified class, and, if successful, sets the location pointed to by **wherep** to the Uniform System processor number for the processor on which the block was allocated. The routine first attempts to allocate the space on **usproc**; should that fail, it tries **usproc+1**, and so forth (wrapping around to processor 0) until it either succeeds, or has tried all processors in the class. **UsAllocAndReportC** is useful for building allocators for scattered data structures, such as the scatter matrices allocated by **UsAllocScatterMatrix**.

- **UsAllocC**

```
char * UsAllocC (nbytes,class)
unsigned long nbytes;
int class;
```

UsAllocC allocates a block of storage of **nbytes** in globally shared memory. The block is allocated from the memory in the specified **class** with the most free space. (See also **UsSetClass**.)

- **UsAllocLocal**

```
char * UsAllocLocal (nbytes)
unsigned long nbytes;
```

UsAllocLocal allocates **nbytes** of globally shared memory from the memory of the local processor.

- **UsAllocOnPhysProc**

```
char * UsAllocOnPhysProc (physproc, nbytes)
int physproc;
unsigned long size;
```

UsAllocOnPhysProc allocates **nbytes** bytes of globally shared memory from the memory of the processor whose physical processor number is **physproc**.

- **UsAllocOnUsProc**

```
char * UsAllocOnUsProc (usproc, nbytes)
int usproc;
unsigned long nbytes;
```

UsAllocOnUsProc allocates **nbytes** of globally shared memory from the memory of the processor whose Uniform System virtual processor number is **usproc**.

- **UsAllocOnUsProcC**

```
char * UsAllocOnUsProcC (usproc, nbytes, class)
int usproc;
unsigned long nbytes;
int class;
```

UsAllocOnUsProcC is similar to **UsAllocOnUsProc**, except that the allocation will succeed only if **usproc** is in the specified **class**.

- **UsAllocScatterMatrix**

```
char * * UsAllocScatterMatrix (rows, cols, nbytes)
int rows;
int cols;
int nbytes;
```

UsAllocScatterMatrix allocates space from global memory for a matrix. The space is scattered by row across the memories of the machine. Each row has a pointer to it. The pointers are put into a vector in the global memory on the node that called **UsAllocScatterMatrix**. **UsAllocScatterMatrix** returns a pointer to that vector. **nbytes** is the number of bytes in an element of the array.

- **UsAllocScatterMatrixC**

```
char * * UsAllocScatterMatrixC (rows, cols, nbytes, class)
int rows;
int cols;
int size;
int class;
```

UsAllocScatterMatrixC is similar to **UsAllocScatterMatrix**, except that only memories in the specified **class** will hold the scattered rows of the matrix and the vector of row pointers. **nbytes** is the number of bytes in an element of the array. (See also **UsSetClass**.)

- **UsFree**

```
UsFree (ap)
char * ap;
```

UsFree frees memory allocated by one of the Uniform System allocators. It is used to free a simple block of storage, as well as a scatter matrix allocated by **UsAllocScatterMatrix**.

- **UsGetClass**

```
int UsGetClass (proc)
int proc
```

UsGetClass returns the class of which **proc** is a member. **proc** is a physical processor number.

- **UsLock**

```
UsLock(lock, n)
short * lock;
int n;

UsUnlock(lock)
short * lock;
```

UsLock sets a lock. It implements a busy-wait type of lock. Before **UsLock** is called, storage for **lock** must be initialized to zero (the clear state). When a processor has locked the lock, * **lock** is non-zero (the set state). **N** specifies the time to wait between attempts to set the lock in tens of microseconds. If **n** is zero, the process will wait about 1 millisecond. **UsLock** does not return until the lock is set. For example:

```
short *lock;
 ...
lock = (short *) UsAlloc(sizeof(short));
*lock=0;
Share(&lock);
 ...
UsLock(lock, 10);      /* Code ``protected'' by lock */
counter = counter + 1;
other code;
UsUnlock(lock);
```

The first process to begin executing the code protected by the lockfinds the lock unset. It sets the lock and executes the code. The next process that tries to set the lock will find it already set and will wait 100 microseconds before attempting to set it again. When the first process finishes executing the code, it clears the lock.

- **UsProcToPhysProc**

```
UsProcToPhysProc(UsProc)
int UsProc;
```

UsProcToPhysProc returns the physical processor number corresponding to the Uniform System virtual processor number **UsProc**.

- **UsSetClass**

```
UsSetClass(proc, class)
int proc, class;
```

UsSetClass adds the memory of the specified processor node to the specified **class**. Initially all memories are in class 0. See also **UsAllocC**, **UaAllocScatterMatrixC**, **UsAllocOnUsProcC**.

- **UsWait**

```
UsWait(n)
int n;
```

UsWait waits for 10**n** microseconds. Using zero for **n** causes the process to wait about one millisecond. **UsWait** is a *busy wait*.

- **WaitForTasksToFinish**

```
WaitForTasksToFinish(GenHandle)
UsGenDesc * GenHandle;
```

WaitForTasksToFinish waits for the task generator specified by **GenHandle** to complete. **GenHandle** must specify an asynchronous generator activated by the calling process. **WaitForTasksToFinish** returns a value (the result code for the generator), which indicates whether the generator ran to completion or was aborted by **AbortGen**.

- **WorkOn**

```
WorkOn(GenHandle)
UsGenDesc * GenHandle;
```

WorkOn works on tasks generated by the task generator specified by **GenHandle**. **GenHandle** must specify an asynchronous generator activated by the calling process. **WorkOn** returns a value (the result code for the generator), which indicates whether the generator ran to completion or was aborted by **AbortGen**.

Index