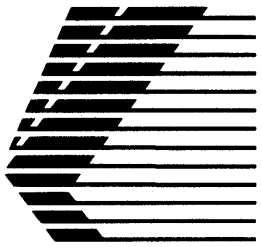

BUTTERFLY™ Parallel Processing

The Uniform System Approach To Programming The Butterfly™ Parallel Processor

BBN Advanced Computers Inc.
A Subsidiary of Bolt Beranek and Newman Inc.

THE UNIFORM SYSTEM APPROACH TO PROGRAMMING THE BUTTERFLY™ PARALLEL PROCESSOR

BBN Report No. 6149

Version 2

June 16, 1986

DRAFT

Preface

Will Crowther developed the Uniform System approach to programming the Butterfly Parallel Processor and wrote documentation for the initial version of the Uniform System Library. David Mankins and Bob Thomas have refined and extended the Uniform System Library. This document was written by Bob Thomas. Numerous people in the Butterfly Group at BBN Laboratories contributed suggestions on how to present this material. Peter Keville provided valuable editorial assistance.

Preparation of this document was supported by the Defense Advanced Research Projects Agency of the Department of Defense.

Table of Contents

List of Figures

1. Introduction

To date, two distinct approaches to programming the Butterfly™ Parallel Processor have seen widespread use. One approach is based on the notion of cooperating sequential processes as described by Dijkstra, Hoare and others. The second is the Uniform System approach developed by BBN Laboratories. It emphasizes the computational tasks that comprise an application and de-emphasizes the notion of processes. A third approach to programming the machine uses the Butterfly Lisp implementation currently under development at BBN Laboratories. This approach will assume greater importance as the Lisp implementation comes into general use. Combinations of two or more of these approaches are possible, as are completely different approaches.

The Uniform System approach is the subject of this document. Other approaches to programming the Butterfly system are described elsewhere. The purpose of this document is to provide enough information about the Uniform System approach to enable an application programmer to write programs for the Butterfly Parallel Processor.

This document assumes familiarity with the C programming language[1] and the Unix™ operating system. It also assumes that the reader has read the document Butterfly Parallel Processor Overview and has access to the reference manual for the Chrysalis™ operating system, Chrysalis Programmers Manual. Basic information about using the machine and various software tools can be found in the Butterfly Parallel Processor Tutorial.

The Uniform System methodology is supported by a set of subroutines collected in a program library called the Uniform System Library. The Uniform System approach and the supporting library are evolving as experience with them in various applications grows. Therefore, this document represents a snapshot of the Uniform System. While the basic concepts of the approach (e.g., the notion of task generation, the notions of globally shared memory and of scattering data uniformly about the machine) are unlikely to change, its details are.

There are two versions of the Uniform System Library: one for the Butterfly and one for the the front end machine (typically a VAX or Sun Workstation). The front end machine version implements all the routines in the Butterfly version and emulates

---

[1] The Uniform System can also be used from Butterfly Fortan programs.

enough of the Chrysalis functions to permit most programs to be (partially) debugged on the front end machine in a uniprocessor environment before moving them to the Butterfly Parallel Processor.

Section 2 describes the philosophy of the Uniform System approach to programming the Butterfly Parallel Processor. Section 3 explains how to use Chrysalis and the Uniform System Library in applications. Several example programs that use the Uniform System are presented in Section 4. Section 5 contains additional information that is useful for running and tuning programs that use the Uniform System. The mechanics of compiling, loading and running programs on the Butterfly system are described in Appendix I. Finally, Appendix II is organized as a reference manual for the Uniform System Library; it contains descriptions of each routine found in the library.

2. Philosophy of the Uniform System Approach

The Butterfly hardware and Chrysalis operating system comprise a foundation on which to build a variety of software structures. A teachable, efficient programming style for using this foundation has evolved from experiments with a wide range of software applications. This style, called the Uniform System approach, has proven to be particularly effective for applications containing a few frequently repeated tasks; e.g., much of scientific computing. It has also been successfully used in applications with less homogeneous task structures.

Beyond the usual concerns of programming, there are two key considerations specific to the Butterfly Parallel Processor: storage management and processor management. The goal of storage management 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 keep all the processors equally busy, thereby preventing the inefficiency that occurs when some processors are overloaded and others sit idle without work to do.

Memory Management

The Butterfly switch provides low delay, high bandwidth access to all of the 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 quite 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 consist of up to 256 separate segments, 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 that must be shared by two or more processors is allocated without regard to which processors will be using it. Of course the stack and variables local to individual processors are kept locally, and like code, are not fetched across the Butterfly switch.

Collectively, the memories of Butterfly 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 uniformly 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 provides a memory allocator that scatters data structures in a way that allows straightforward addressing conventions. It also supports a set of more specialized techniques for cases where that allocator is either inappropriate or ineffective.

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

1. Use of a single large address space shared by all processes to simplify programming; and
2. Scattering application data uniformly 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 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, and we feel this benefit greatly outweighs the modest cost.

The need to make certain operations on memory atomic is another aspect of memory management. This is not unique to parallel systems; it is also necessary in

multiprogrammed uniprocessors. 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 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 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 N^2 parallel sort longer to run on a 128 processor Butterfly than it takes an N log N sort to run on a single 68000.

2. Attempt to do the same number and kind of steps as 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 aggregate 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 (FFT's) 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 Parallel Processor can work very efficiently with individual tasks a few milliseconds in length; if necessary, it can 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 P concurrent tasks for P processors. The other strategy recommends using many more than P tasks, typically an order of magnitude or more. 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 attempts to manipulate 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 relies on having a large number of tasks relative to processors to minimize end effects: some waiting occurs at the end of the problem, but this waiting is generally acceptable since it is small relative to the total program execution time.

The Uniform System encourages the dynamic approach. For many applications the dynamic approach is simpler and more reliable, since it is unnecessary to know in advance how long an individual piece of work will take. Furthermore, it is adaptable to varying numbers of processors and sizes of problems.

After the programmer has determined the processing that is to occur in parallel, he must then control the Butterfly 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, the programmer is required to structure an application into two parts:

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

To illustrate this, consider matrix multiplication as an example. One way to structure a matrix multiplication program would be to write a routine that computes the dot product of a row and a column; and to ensure that the routine for the dot product task gets called once for each element of the result matrix, using the appropriate row and column of the operand matrices as parameters.

Usually a well designed program will be structured as a set of subroutines to improve program modularity, whether or not it is intended for parallel execution. Normally, there will be a subroutine per 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 case of the matrix multiplication example, there is a single task type, computing dot products, and corresponding to that task type, the dot product routine, whose row and column parameters specify particular tasks.

The second part of the application code comprises one or more subroutines able to 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 (GA) first builds a task descriptor data structure (TD) that specifies the task generator in terms of the worker (W) procedure, the data (D), and the task generation (TG) procedure. It then "activates" the generator by making the task descriptor (TD) available to other processors. The processor that invoked the generator activator along with other available processors then use the task descriptor (TD) and the task generation procedure (TG) to make repeated calls on the worker procedure (W), specifying subsets of the data to work upon. Each call of the worker procedure (W) is a task. When the last task is done, the processor that called the generator activator procedure (GA) continues execution of its program, while the other processors that worked on the tasks look for something else to do. 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. Occasionally a generator will involve a simple queue, in which case it would operate much like a process scheduler found in many time sharing systems; the next task for execution would be the one at the front of the queue. In general, though, a large number of applications can be constructed from a small set of generators. The Uniform System Library includes a collection of commonly used generators, and others will be added over time.

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[2].

Often an algorithm will require 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

---

[2]This section has been careful to use the terms *generator activator procedure* and *task generation procedure*. The rest of this document 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.

requirements are met.

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

1. The generator mechanism is very efficient. It is implemented using one process per processor in a way that ensures no unnecessary context swaps occur. 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 by finding an appropriate generator activator procedure in the library. Both the task generation and the worker procedures 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.

2. 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 important, programs are able to run on "reduced" configurations: for example, one where processors have been removed for repair.

3. The load can be balanced dynamically. Whenever a processor becomes free, a generator identifies the next task to be executed. 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.

3. Using the Uniform System

When the Uniform System approach is used on the Butterfly Parallel Processor, programs are written much the same way as for a uniprocessor. In fact, if a program never invokes a task generator, it will run on a single Butterfly processor. 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 section describes each routine found in the Uniform System Library, as well as some frequently used Chrysalis routines. Several example programs that illustrate how to use the Uniform System routines are contained in Section 4. Section 5 describes how to run Uniform System programs on the Butterfly system, and it also discusses some issues in tuning program performance. The descriptions of the library routines in this section are narrative in nature. The information presented in this section is repeated in Appendix II, which is organized for use as a reference manual for the Uniform System Library.

Initialization

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 finally initializes the Uniform System storage allocator. *InitializeUs* must be called before any other Uniform System routine, and it should be called only once.

Configuration Information

It may be desirable for a program to know the number of processors and memory banks available on a machine. The routines

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

return configuration information. *ProcsInUse* does not count processors that have

been removed by the *TimeTest* routine (see "Measuring Your Program" below). *MemoriesAvailable* counts memory in units of 64 KBytes. *DistinctMemoriesAvailable* 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 (for example, when other software, such as the Ethernet routines, have taken it all).

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 front end machine version of the Uniform System, *Proc_Node* is arbitrarily set. The particular numbers used as hardware processor numbers for a Butterfly machine 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 usually 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 first end version of the Uniform System, *UsProc_Node* is always 0.

It is important to note that the mapping between virtual processor number and hardware processor number may change from run to run. This can happen, for example, if some processors are missing from the configuration when the program is run.

The routines

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

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

Synchronization and Atomic Operations

Sometimes two processors need to work on the same data at the same time. If the order of work doesn't matter (incrementing a counter, for example), then the principal concern is that the processors don't interfere with one another (i.e., that one finishes before the other starts). If the order of work does matter (A is writing and B is reading, say), the program logic is probably 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 there is a Chrysalis atomic operation (e.g., *Atomic_add*, *Atcmic_ior*, etc.) that performs the desired operation. The Chrysalis atomic operations 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 pointed to 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" functionality provided by *Atomic_add*. It is also important to note that in its current implementation *Atomic_add_long* is atomic only with respect to other *Atomic_add_long* calls. In particular, it is possible for the execution of a read operation to 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) is propagated to the higher order bits[3].

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:

```
LOCK(lock, n)
UNLOCK(lock)
```

The *LOCK* 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 unset state with

---

[3]This anomalous behavior may be eliminated in a future release of Chrysalis that provides full support for atomic operations on 32 bit quantities.

value 0), and *n* is an *int* 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 1 millisecond[4].

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 *int* that specifies the time to wait in tens of microseconds. As with *LOCK*, using zero for *n* forces use of a default which is about 1 millisecond.

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. The Chrysalis operations include:

```
Make_DualQ
Enq_DualQ
Deq_DualQ
Wait_DualQ
Pool_DualQ
Make_Event
Post_Event
Wait
```

Consult the Chrysalis Manual for details of these and related operations.

Memory Management

Two classes of memory are available to Uniform System programs:

1. Process private memory. As the name suggests, data in process private memory can be accessed only by one process.
2. Globally shared memory. Data in globally shared memory is accessible by all Uniform System processes[5].

Within these two classes several quite 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 1):

---

[4]Note that if you nest these operations casually, you can achieve deadlock.

[5]It is possible, using the Chrysalis *Map_Obj* operation, to have memory that is shared among some, but not all, processes. We recommend you 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.

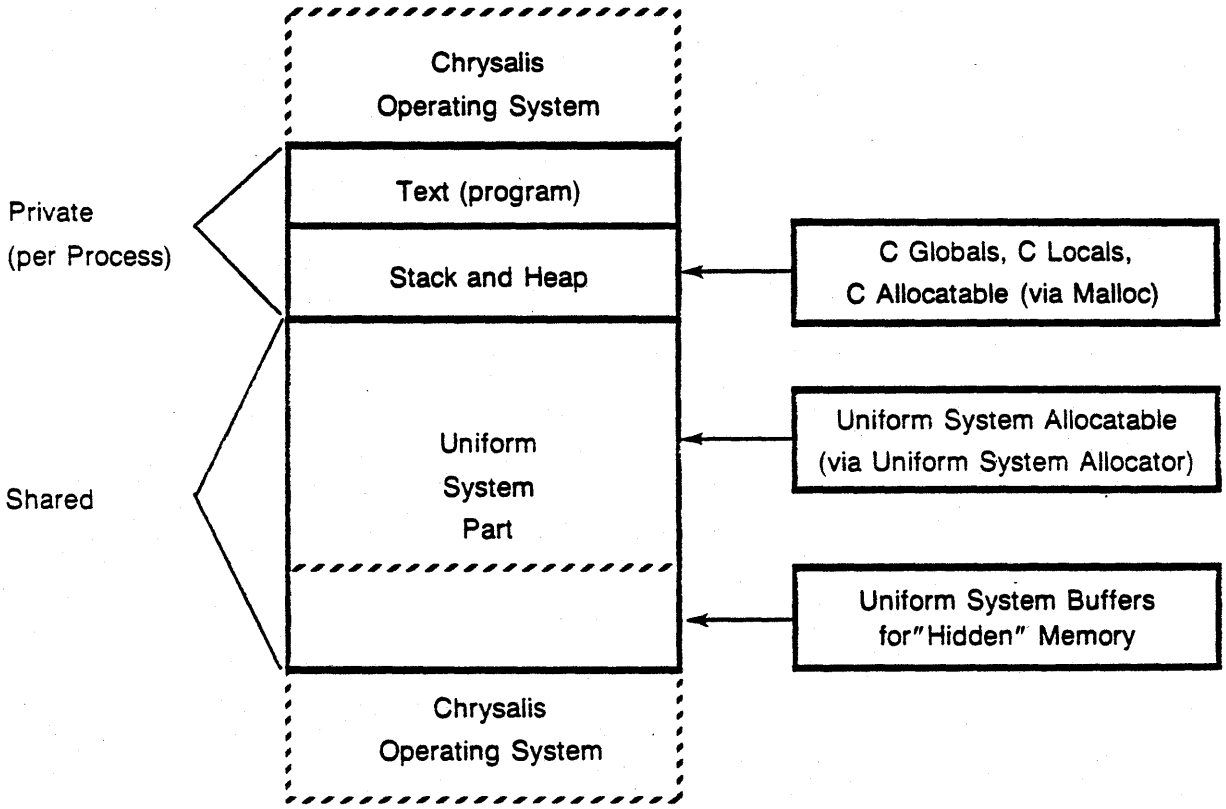

Figure 1: Address space of a Uniform System process.

- 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.

- Shared storage. Storage of this type is obtained using the Uniform System allocators *Allocate*, *AllocScatterMatrix*, and the like, and it is globally shared. There is one instance of a Uniform System allocated variable per Butterfly machine. Since this is globally shared storage, you can pass pointers from processor to processor, and use them on whatever processor you like. This is the only way to communicate between different processors and tasks, unless you choose to bypass the Uniform System and 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 *Share* and *ShareSM* below.)

- Hidden Storage. Storage of this type is globally shared. A Butterfly node is limited to a 24 bit virtual address (16 MBytes) by its 68000 processor. The Uniform System allows the user to access nearly that amount of memory directly (there is an area at the top and one at the bottom of memory taken by Chrysalis). However, Butterfly systems with more than 16 processors have more than 16 MBytes of real memory. To support the use of that memory the Uniform System supports the notion of "hidden" memory. Hidden memory is allocated much like regular memory, but the allocator returns a descriptor for the block of memory rather than a pointer. The user program can use Uniform System operations to copy data between hidden memory and directly accessible memory. Thus hidden memory acts rather like fast secondary storage. (The hidden memory feature is not yet implemented.)

The Uniform System storage allocator creates and manages the globally shared memory region of the process address space (see Figure 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.

Implementing globally shared memory is somewhat more involved than it might seem at first. Since the Butterfly computer uses a standard 68000 C compiler, the language provides no help when it comes to allocating globally accessible storage. If a

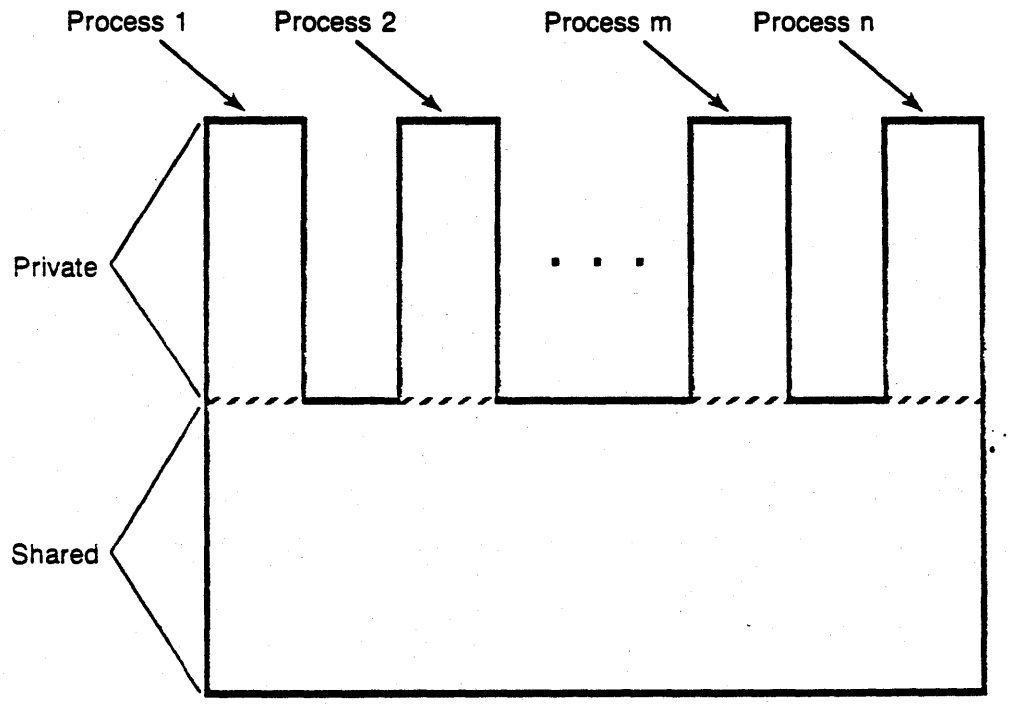

Figure 2: Uniform System processes share a large portion of their address spaces.

program variable is declared to be a C global, that only means that the variable is visible to the program modules 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 *alloc* system calls supported by Chrysalis allocate memory that is process private rather than globally shared. The Uniform System uses the Chrysalis Object Management System to implement globally shared memory.

Storage Allocation

You can allocate a block of storage in globally shared memory with:

```
Allocate(SizeInBytes);
```

The Uniform System allocates the block from the memory with the most free space.

If you want to allocate globally shared storage on the local processor, use

```
AllocateLocal(SizeInBytes);
```

If you want to specify a particular processor, you can use:

```
AllocateOnUsProc(Processor, SizeInBytes);
```

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

```
AllocateOnPhysProc(PhysProcessor, SizeInBytes);
```

Proper storage management on the Butterfly computer is important! If your data isn't uniformly distributed over all available memory, you may get poor performance. It usually doesn't 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 (described below) 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. For more complex data structures, *AllocateOnUsProc* and *AllocateOnPhysProc* can be used to scatter data across the machine. In addition, 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 below in "Making Copies of Process Private Data") and the generator "initialization" routines (described below in "Generators") are useful for making such copies.

The data structures required by many applications can be represented naturally by 2-dimensional matrices. Furthermore, higher dimensional matrices can be represented in a straightforward way by 2-dimensional matrices, as can one dimensional vectors. For example, a 3-dimensional matrix can be thought of as a 2-dimensional matrix, each element of which is a vector. Hence, 2-dimensional matrices can be used as a fundamental building block for supporting many application data structures. To reduce potential memory contention, it is desirable to scatter these data structures across the machine.

The routine

```
AllocScatterMatrix(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[6]. The Uniform System allocates the vectors in separate memories. The vector of pointers, a pointer to which is returned to the caller, is filled in with pointers to the scattered row vectors (see Figure 3). Elements of an array *A* allocated in this way can be referenced using standard C array notation:

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

Currently, the Uniform System storage allocator is fairly simple. In particular, it cannot free storage piecemeal. You can free it all using

```
FreeAll();
```

or your program can simply live with its garbage.

Processor Management

The Uniform System processor management mechanism is accomplished through the use of task generators. A "task" is the basic unit of parallel computation; 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

---

[6] At present, *ncols***element_size* must be <= 64K bytes, and *nrows**4 must be <= 64K bytes.

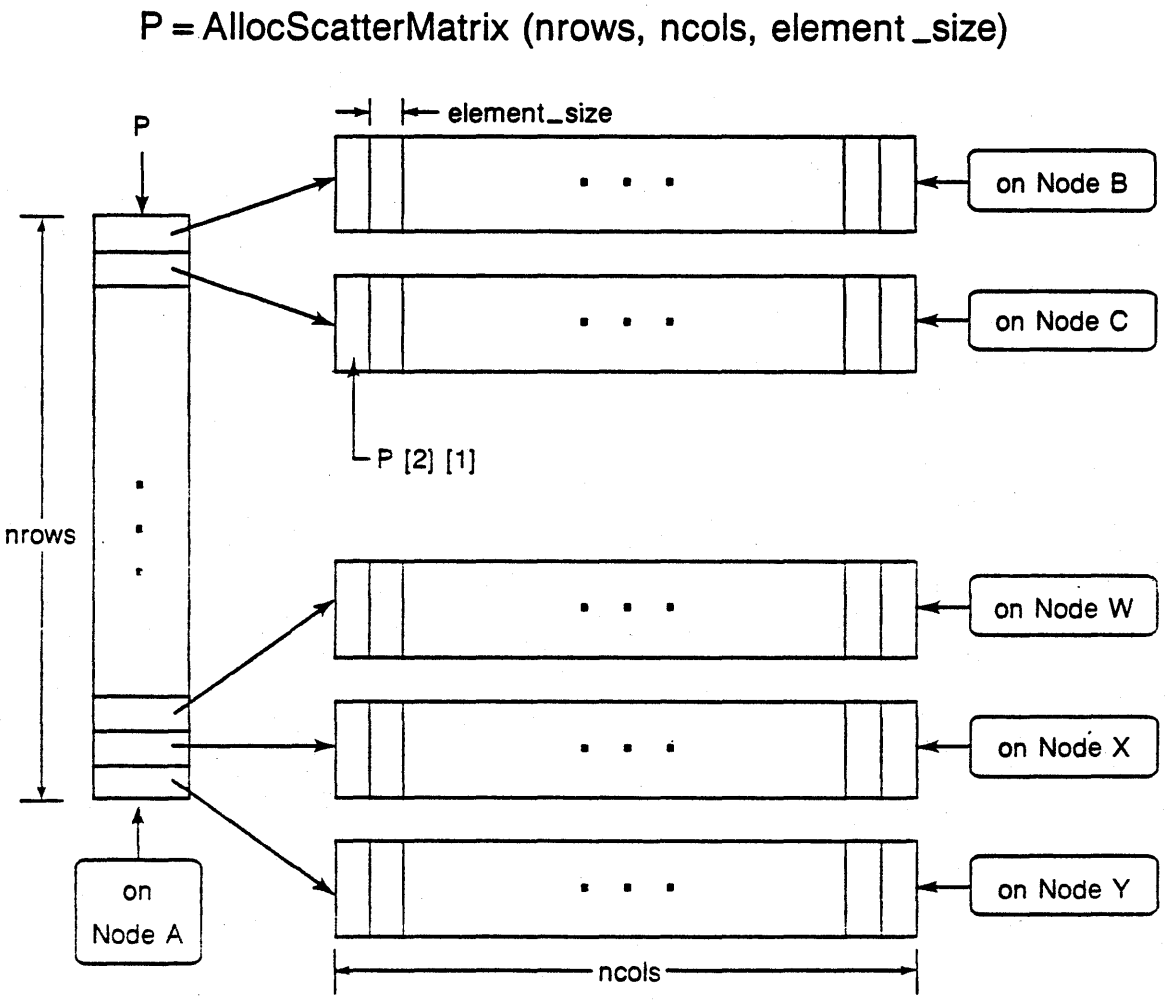

Figure 3: A scattered matrix created by AllocScatterMatrix.

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 routine that is able to 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, of course, since task generation is distributed). However, a generator may 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 a large effect on system efficiency.

While the application programmer is expected to provide both task generation and task implementor (worker) routines, experience has shown that the relatively small set of generators (or more precisely, generator activator procedures) supported by the Uniform System Library (see "Generators" below) are sufficient for a wide range of applications. The way to achieve parallel operation is to structure your program to fit the mold of one of these task generators.

The Uniform System supports two generator control disciplines:

1. **Synchronous** generators. When a process invokes a synchronous generator, control returns from the generator procedure after all of the tasks generated have been processed. Furthermore, the processor that calls the generator works on the tasks that are generated.

2. **Asynchronous** generators. When an asynchronous generator is invoked, control is returned to the calling process as soon as the generator has been activated. This enables the process to work on other things. The calling process may later work on tasks generated if it so chooses.

The Uniform System matches available processors to the tasks generated. Its System processor manager 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 labeled idle. As long as there are active generators, there are no idle processors.

It is perfectly reasonable to nest calls to generators. In fact, that is an expected mode of operation. When calls to generators are nested, the Uniform System assumes the order the generators are dealt with is unimportant, and it picks 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 which can be searched by N tasks performed in parallel; 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 the difference 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 activator generator procedures. All these generator activator procedures make use of a "universal" generator activator procedure. Use of the universal generator activator procedure is described below in the section "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".

– 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 index values between zero and *Range−1*[7]. *Arg* is typically a pointer to a problem description data structure.

---

[7]Earlier versions of the Index family generators required *Range* to be less than 2^{15}. That limitation has been removed. However, 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.

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, the routine *Final* is called once on each processor used to perform tasks for the generator after the last call of the *Worker* routine on each such processor. 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 which 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, GenHandle);
```

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

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

```
AbortGen(GenHandle, termination_code);
```

where *termination_code* is an *int*. 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*[8]. If all of a generator's tasks have

---

[8]More than one processor may call *AbortGen* to abort a generator. In such a case, the value returned is the smallest *termination_code*.

been performed (i.e., either it is not abortable, or it is abortable but it was not aborted), the generator returns the *code genEXHAUSTED*.

There are other synchronous generators in the Index family which 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 descriptions 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, GenHandle);
```

– 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, GenHandle)
```

As with the Index family, there are several additional generators in the Array family that 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, GenHandle);
```

– 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. The sequence of *(index1, index2)* values span the "half" array beneath the diagonal of a *Range1 x 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 the variable *i* (in the *Worker* routine).

– Miscellaneous Generators.

The generator

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

generates exactly 1 task, *call(arg)*, for every processor (that has not been removed by the *TimeTest* routine).

The generator

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

exactly 1 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, $r0,...,rn-1$, and *arg_list* is an array of "arguments" of length n, $arg1,...,argn$, generates n tasks; the ith task is $ri(argi)$.

Asynchronous Generators

There are asynchronous versions of each of the generators in the Index, Array and Half Array generator families. While 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

```
GenHandle = AsyncGen... (...);
```

activates the generator and then returns control immediately to its caller along with *GenHandle*, an "identifier" for the generator activated. The processor that invokes an asynchronous generator can choose to work on tasks generated by the generator by using the call

```
code = WorkOn (GenHandle);
```

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*. Alternatively, the processor that invokes an asynchronous generator can do other things.

The sequence

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

is functionally equivalent to the corresponding synchronous generator.

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

```
code = WaitForTasksToFinish (GenHandle)
```

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

The asynchronous generators currently supported by the Uniform System are:

```
Index Family:
    GenHandle = AsyncGenOnIFull (Init, Worker, Final, Arg, Range,
                                 Limited, Abortable)
    GenHandle = AsyncGenOnI (Worker, Range)
    GenHandle = AsyncGenOnILimited (Worker, Range, nprocs)
    GenHandle = AsyncGenOnIAbortable (Worker, Range)

Array Family:
    GenHandle = AsyncGenOnAFull (Init, Worker, Final, Arg, Range1, Range2,
                                 Limited, Abortable)
    GenHandle = AsyncGenOnA (Worker, Range1, Range2)
    GenHandle = AsyncGenOnALimited (Worker, Range1, Range2, nprocs)
    GenHandle = AsyncGenOnAAbortable (Worker, Range1, Range2)

Half Array Family:
    GenHandle = AsyncGenOnHAFull (Init, Worker, Final, Arg, Range1, Range2,
                                  Limited, Abortable)
    GenHandle = AsyncGenOnHA (Worker, Range1, Range2)
    GenHandle = AsyncGenOnHALimited (Worker, Range1, Range2, nprocs)
    GenHandle = AsyncGenOnHAAbortable (Worker, Range1, Range2)
```

Each of these corresponds to one of synchronous generator described above.

Making Copies of 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 could occur as multiple processors access the variables. For example, as part of initialization one processor might set C global variables which 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 is to use one of the *Share* routines.

The effect of

```
Share(&N);
```

where *N* is declared as a global *int* (and is therefore process private), is to cause the value of *N* (in the processor invoking *Share* at the time *Share* is invoked) to be copied into *N* in each processor used to perform tasks generated by subsequent task generators. The value of *N* is set in each such processor prior to the call of the task initialization routine for the next task generator handled by that processor[9]. The effect of *Share* is illustrated schematically in Figure 4. Note that only the value of *N* is propagated to other processors by the *Share* mechanism. Therefore, should one processor change its copy of *N*, only that processor will see the changed value.

A non-integer variable *X* can be passed to other processors by

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

where *size* is the size of *X* in bytes. A pointer variable *P* and the block of data it points to can be passed to other processors by

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

where *size* is the size in bytes of the block of data pointed to by *P*.

When many processors make frequent references to many elements of an array allocated by *AllocScatterMatrix*, it is often desirable for each processor to have its own copy of the vector of pointers created by *AllocScatterMatrix*. 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 = AllocScatterMatrix(nrows, ncols, element_size);
```

will cause such copies to be made. Each processor used to perform tasks generated by task generators called after the call to *ShareScatterMatrix* will have its *P* set to

---

[9]Generators that have no explicit initialization routine (see the section on "Synchronous Generators" above) can be thought of as having a null or no-op initialization routine. *N* is set prior to the (non-existent) call of the null initialization routine, and therefore prior to the first call of the task worker routine on that processor.

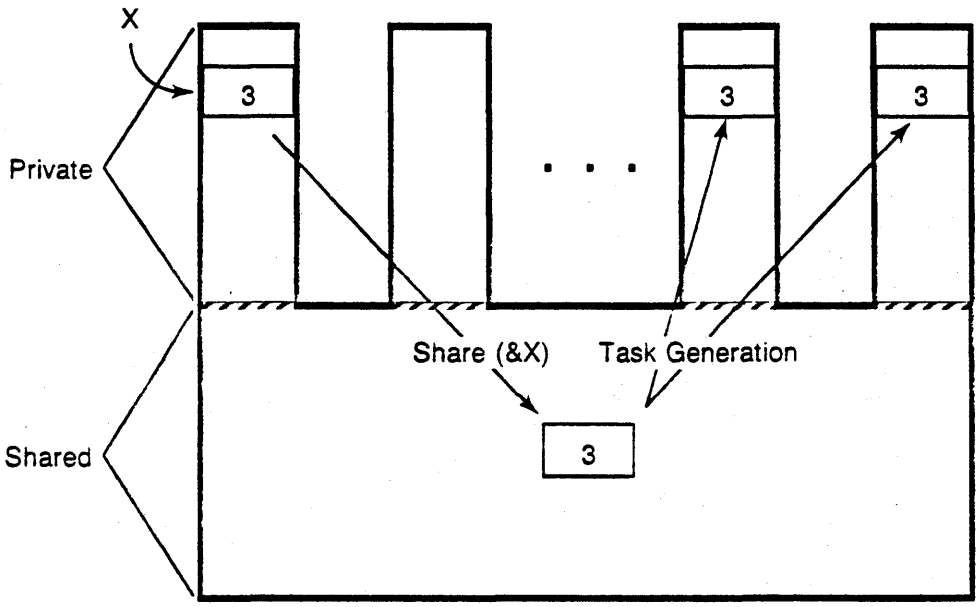

Figure 4: *Share* is used to pass copies of process private variables.

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[10].

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,

```
globally_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 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.

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. The "statement"

```
MakeSharedVariables;
```

---

[10] *ShareScatterMatrix(&P, nrows)* is logically equivalent to *SharePtrAndBlk(&P, 4*nrows)*, but operates much faster, since it is careful to make copies from other copies as well as from the original.

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. To reference a globally shared variable that has been declared in this way, the programmer must explicity specify that it is shared via the *SHARED* prefix; for example,

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

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

1. *BEGIN_SHARED_DECL* may appear only once in a program. That is all variables to be shared via this mechanism must be declared in one place.

2. All of the shared variables are allocated in the same physical memory. Hence contentions for that memory could be a performance bottleneck. (See Section 5 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. It times the middle routine only (*Execute*), and passes the execution time, the number of processors, and the effective number of processors[11] on to the specified display routine (*PrintResults*) at the end of each pass:

```
PrintResults(time, procs, effprocs)     /* Written by the user. However note */
int time, procs;                        /* that the Uniform System provides */
float effprocs;                         /* a simple version, see below. */
```

[11] 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) processors, then the effective number of processors is (time k proc) / (k * (time n procs)).

At the start of the run, *TimeTest* asks you to specify from the keyboard the configurations to be timed. If, for example, all possible configurations are specified, *TimeTest* will run the three specified routines in order on a single processor, then call them again with a two processor configuration, and so forth until it has run the program on every possible processor configuration up to the size of the machine being used.

The display routine

```
TimeTestPrint(time, procs, effprocs)
```

can be used to print *time*, *procs* and *effprocs*. You may prefer to supply your own display routine if you want to print other information.

A variant on *TimeTest* gives somewhat more control over the test runs:

```
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 which is run on *end* processors. *TimeTestFull* is particularly useful on bigger machines, where incrementing by one processor can be tedious. If *start* (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, *TimeTestFull* asks the user to supply values for *start*, *delta*, and *end* at the start of the run. This is the normal usage for timing many programs, and is what you get with the simpler *TimeTest*.

Although all Processor Nodes in a Butterfly system are functionally equivalent, there is a distinguished King Node that is special in two ways: it is the node to which the console terminal is connected; and it controls the machine while the operating system is being booted. Because a terminal handler and window manager run on the King Node, it appears about 8%–10% slower than the other nodes to application programs. When benchmarking a program, it is desirable to avoid using the King Node to ensure that the measurements were not affected by the processing requirements of the terminal handler and window manager,

The King Node can be avoided by using the routine

```
InitializeUsForBenchmark();
```

rather than *InitializeUs*, and starting the program on some node other than the King Node[12].

Tagging Memories

Sometimes it is useful to partition the node memories into classes. For example, the *Allocate* and *AllocScatterMatrix* 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 *int*, makes the memory of the specified Processor Node a member of the specified *class*. All memories are initially in class 0.

The allocation routines

```
AllocateC(SizeInBytes, class)
AllocScatterMatrixC(Processor, SizeInBytes, class)
AllocateOnUsProcC(nrows, ncols, element_size, class)
```

where *class* is an *int*, are similar to *Allocate*, *AllocScatterMatrix*, and *AllocateOnUsProc*, differing in that they allocate space only on memories in the specified *class*. *AllocateOnUsProcC* will fail if *proc* is not in *class*.

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 obtain from each processor node in order to establish the Uniform System globally shared address space.

While this is appropriate in many cases, there are situations which may require finer control of the resources used by Uniform System programs. In such situations, the routine

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

---

[12]The *-on* switch of the *run* command or the *us* utility can be used to start a program on a non-King node. See Section 5 and the Chrysalis Programmers Manual for details.

can be used prior to calling *InitializeUs* to specify values for configuration parameters that differ from the values normally used by *InitializeUs*. *Spec* is an array (of *int*'s) which specifies the configuration; 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) integer(s) which specify the value for the parameter.

Currently the following *configuration_code*'s are defined:

Code	Parameter
configProcs	integer = number of processors to include in the Uniform System configuration.
configMaxSars	integer = number of segment attribute registers (SARs) to use to define process address spaces.

As an example, the code fragment:

```
spec [0] = configProcs;
spec [1] = 6;
ConfigureUs (spec, 1);
InitializeUs ();
```

limits the Uniform System program to (a maximum of) 6 processors.

Clock

Your program can read the Butterfly clock using the routine

```
GetRtc()
```

that returns the time since the system was booted in units of 62.5 microseconds. On the Butterfly the clock value is the same (plus or minus two ticks) on every processor. The front end version of the Uniform System Library uses the real time clock on the front end 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 "Measuring Your Program" above.

I/O

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.

The Chrysalis utility *tftp* provides means for a user to manually transfer files to and from the front end host. Consult the *tftp* section of the Chrysalis Manual for information on how to use this utility. In addition, support for the standard Unix file

I/O functions for files on the front end 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 to read and write files on the front end computer. Consult the Chrysalis Manual for deails on how to use the *streams package*.

Building a Generator[13]

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.

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, Gen,
            Async, MaxProcsToUse, Abortable, ResultP)
```

As above, *Init*, *Worker*, and *Final* are routines, *Arg* is typically a pointer to a data structure, and *Range1* and *Range2* are *short*'s. *Type* is *GENERATOR*. *Gen* is a task generation routine you supply to generate the next task (described in more detail below). *Async* is a boolean which specifies whether the generator is to be activated asynchronously (*true*) or synchronously (*false*). *MaxProcsToUse* specifies the number of processors to which the generator is to be restricted; as with the "limited" forms of the generators, *0* or *−1* indicates no limitation, and a positive value ensures that no more than that number of processors will be used. *Abortable* is a boolean which specifies whether the generator is to be abortable (*true*) or not (*false*). Finally, *ResultP* is a pointer that is used if *Abortable* is *true*; it specifies a location where the *ActivateGen* should store the generator "result code" (= *genEXHAUSTED* if all tasks are generated; or the termination code parameter of *AbortGen* if the generator is aborted).

The *Gen* task generation routine has the form:

```
Gen(TD)
UsGenDesc *;
```

where *TD* is a pointer to a task descriptor data structure of the form:

---

[13]This is an advanced topic. While the general approach to building generators is not likely to change, the details may.

```
struct
{  short started;
   short type;
   /* Defined types are: */
        #define IDLETASK  1
        #define GOAWAYTASK 2
        #define GENERATOR 4
   short incarnation_number;
   short state;
   /* Defined states are: */
        #define ACTIVE 1
        #define INACTIVE 2
   short us_lock;
   short lock;
   int   (*init)();
   int   (*call)();
   int   (*gen)();
   int   (*final)();
   int   arg;
   char  *currentShare;
   int   range;
   int   range2;
   QH    returnQ;
   int   post_pending;
   short MaxProcsToUse;
   int   end;
   int   abortable;
   short retcode;
   /* Defined retcodes are: */
        #define genEXHAUSTED -1
   short endlock;
   union {long Long; short Short;} index;
   union {unsigned long Long; unsigned short Short;} index2;
   short locka[nlocks];
   short index1a[nlocks];
   short index2a[nlocks];
}
```

The task descriptor data structure is declared in the header file *usgen.h* which you must *#include* with your program when using *ActivateGen*. The *Worker*, *Init*, *Final*, *Gen*, *Arg*, *Range1*, *Range2*, *Type*, *MaxProcsToUse*, and *Abortable* parameters of *ActivateGen* are used to initialize the *call*, *init*, *final*, *gen*, *arg*, *range*, *range2*, *type*, *MaxProcsToUse*, and *abortable* fields of the task descriptor structure. The *lock* and the *locka* array fields of the task descriptor structure are initialized to 0 and are available for use as locks by the *Gen* routine you supply; and, the *index* and *index2* fields, and the *index1a* and *index2a* array fields are initialized to 0 and are available for use by your *Gen* routine for bookkeeping associated with generating the tasks. *ActivateGen* uses the remaining fields (*started*, *shareCount*, *returnQ*, etc.) for internal bookkeeping.

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

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

is desired which 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, 0, false, 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);   /* Generate the next index value. */
    if (index >= range) break;  /* Finished if range exceeded. */
    (*worker) (arg, index);     /* Otherwise, call the worker routine. */
  }
}
```

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 (*TD*) and *GenShortIdx* to generate and execute tasks.

4. Examples

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

4.1. Multiprocessor "hello world"

This example illustrates the use of the task generator *GenOnI*, the variable *Proc_Node*, and the *TotalProcsAvailable*, *PhysProcToUsProc* and *Share* routines. The example 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. Figure 5 is the typescript produced by running it on a large Butterfly system. The *-sars 200* option to *run* is explained in Section 5. The line *init1 -- find memory ...* is generated by *InitializeUs* as a debugging aid, and is likely to be eliminated in a future version of the Uniform System.

The multiprocessor "hello world" program is shown in Figure 6. The program uses *Allocate* to allocate 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 to print the "Hello" message from each processor. The only tricky part is ensuring that each processor performs exactly one task. Without some form of coordination, it is possible, in general, that some processors would get more than one task, and others would get none. For this program, the coordination is simple. Each processor simply prints its message, 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.

```
[0] run -sars 200 Hello
loading Hello from VAX...

 init1 — find memory      init2 — map memory     init3 — start processors

There are 123 nodes on this machine

Hello world from node #0 (= hardware node #0)
Hello world from node #3 (= hardware node #6)
Hello world from node #2 (= hardware node #4)
Hello world from node #4 (= hardware node #8)
Hello world from node #5 (= hardware node #a)
Hello world from node #1 (= hardware node #2)
Hello world from node #6 (= hardware node #c)
Hello world from node #7 (= hardware node #e)
Hello world from node #8 (= hardware node #10)
Hello world from node #9 (= hardware node #12)

...

Hello world from node #79 (= hardware node #a2)
Hello world from node #72 (= hardware node #94)
Hello world from node #46 (= hardware node #5e)
Hello world from node #44 (= hardware node #5a)
Hello world from node #49 (= hardware node #64)
Hello world from node #60 (= hardware node #7c)
Hello world from node #58 (= hardware node #78)
Hello world from node #113 (= hardware node #e8)
Hello world from node #85 (= hardware node #ae)
Hello world from node #76 (= hardware node #9c)
Hello world from node #65 (= hardware node #86)
Hello world from node #81 (= hardware node #a6)
Hello world from node #25 (= hardware node #32)
Hello world from node #66 (= hardware node #88)
Hello world from node #120 (= hardware node #f6)
Hello world from node #59 (= hardware node #7a)
Hello world from node #82 (= hardware node #a8)
Hello world from node #41 (= hardware node #54)
Hello world from node #47 (= hardware node #60)
Hello world from node #108 (= hardware node #de)
Hello world from node #53 (= hardware node #6e)
Hello world from node #86 (= hardware node #b0)
Hello world from node #24 (= hardware node #30)
Hello world from node #118 (= hardware node #f2)
Hello world from node #107 (= hardware node #dc)
Hello world from node #29 (= hardware node #3a)
Hello world from node #32 (= hardware node #40)
Hello world from node #90 (= hardware node #b8)
Hello world from node #104 (= hardware node #d4)
Hello world from node #95 (= hardware node #c2)
Hello world from node #91 (= hardware node #ba)
Hello world from node #64 (= hardware node #84)
Hello world from node #22 (= hardware node #2c)
Hello world from node #48 (= hardware node #62)
Hello world from node #61 (= hardware node #7e)
Hello world from node #96 (= hardware node #c4)
Hello world from node #112 (= hardware node #e6)
[0]
```

Figure 5: Typescript from the multiprocessor "Hello world" program.

```
 1  /* Multiprocessor "Hello" program */
 2
 3  #include <us.h>
 4
 5  short * nodecount;
 6
 7  PrintHello(dummy, index)
 8      int dummy, index;
 9  {   printf("Hello world from node #%d (= hardware node #%x)\n",
10              PhysProcToUsProc(Proc_Node), Proc_Node);
11      Atomic_add(nodecount, -1);
12      while (*nodecount != 0) UsWait(0);
13  }
14
15  main()
16  {   InitializeUs();
17      nodecount = (short *) Allocate(sizeof(short));
18      * nodecount = TotalProcsAvailable();
19      printf("\nThere are %d nodes on this machine\n\n", *nodecount);
20      Share(&nodecount);
21      GenOnI(PrintHello, *nodecount);
22  }
```

Figure 6: The multiprocessor "Hello world" program.

4.2. Matrix Multiplication

This example illustrates use of the *AllocScatterMatrix* 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 ith row of b and the jth column of c.

The program is written to run on a set of processor configurations specified from the keyboard. Figure 7 is a typescript produced by running the matrix example program on a small Butterfly system. The line *please enter start ...* is used to gather specification of the processor configurations to be used for the run. It is output by the *TimeTest* routine; see the discussion of *TimeTest* in Section 3 for an explanation of the *start*, *del* and *end* parameters. The line [*8*] *time = 12657 ...* is output by *TimeTestPrint*. It indicates that the matrix example program took 12657 ticks or .79 seconds on 8 processors, and that it achieved a speedup of 7.8 over 1 processor (= 7.8 effective processors), utilizing the 8 processors with 98.70 % efficiency.

The program itself is shown in Figure 8. It parallelizes matrix multiplication by computing the individual elements of the product matrix a in parallel. Each element is the dot product of a row of matrix b and a column of matrix c.

The program has 6 routines[14]:

1. *InitProblemOnce*. As its name suggests, this is an initialization routine called once per invocation of the program. *InitProblemOnce* allocates space in globally shared memory for the result matrix a, and the two operand matrices b and c, using the Uniform System allocator *AllocScatterMatrix*. Note that the variables a, b, and c are C globals 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*. This is an initialization routine called before each run of the matrix multiplication code on a given configuration of processors. It simply

---

[14] Chrysalis starts the program by calling the routine *main* on a single processor.

```
[0] run -sars 200 MatrixExample
loading MatrixExample from VAX...

 init1 — find memory      init2 — map memory     init3 — start processors

Starting Matrix Multiply
Matrix Size: 20

 please enter start, del(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 = 99944 ticks =   6.24 sec;  ep =   0.9;  eff = 0.9999
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 = 50204 ticks =   3.13 sec;  ep =   1.9;  eff = .9953
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 = 25079 ticks =   1.56 sec;  ep =   3.9;  eff = .9962
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 = 12657 ticks =    .79 sec;  ep =   7.8;  eff = .9870
[0]
```

Figure 7: Typescript of the matrix example program.

```
1  /* Matrix multiply - unoptimized example program */
2
3  #include <us.h>
4
5  int Size;
6  float **a,**b,**c;
7
8  InitProblemOnce()
9  { int i,j;
10   a = (float **) AllocScatterMatrix(Size,Size,sizeof(float));
11   b = (float **) AllocScatterMatrix(Size,Size,sizeof(float));
12   c = (float **) AllocScatterMatrix(Size,Size,sizeof(float));
13   Share(&a);  Share(&b);  Share(&c);
14   for (i=0; i<Size; i++)
15     for (j=0; j<Size; j++)
16     {  if (i==j) b[i][j] = 3.; else b[i][j] = 0.;
17        c[i][j] = Size * i + j;
18     }
19 }
20
21 InitPerRun()
22 { int i,j;
23   for (i=0; i<Size; i++)
24     for (j=0; j<Size; j++)
25       a[i][j] = 0.;
26 }
27
28 DotProduct(dummy,i,j)
29   int dummy,i,j;
30 { int k;  float *bb, *cc, temp;
31   temp = 0.0;     bb = b[i];     cc = c[j];
32   for (k=0; k<Size; k++)
33     temp += *bb++ * *cc++;
34   a[i][j] = temp;
35 }
36
37 Body()
38 { GenOnA(DotProduct, Size, Size);
39 }
40
41 PrintAnswer(time,procs,speedup)
42   int time,procs; float speedup;
43 { int i,j;
44   for (i=0; i<6; i++)
45   { printf ("\na row %d ", i);
46     for (j=0; j<6; j++)
47       printf ("%d. ", (int) a[i][j]);
48   }
49   TimeTestPrint(time,procs,speedup);
50 }
51
52 main()
53 { InitializeUs();
54   printf("\nStarting Matrix Multiply\nMatrix Size: "); scanf("%d",&Size);
55   Share(&Size);
56   InitProblemOnce();
57   TimeTest(InitPerRun, Body, PrintAnswer);
58 }
```

Figure 8: The matrix example program.

zeros the answer matrix *a*[15]. Note that the rows of the matrix could be zeroed in parallel if the matrix was very big.

3. *DotProduct*. This 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 ith row of *b* and the jth 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* doesn't 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*. This 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 $0 <= i < Size$ and $0 <= j < Size$.

5. *PrintAnswer*. This 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 1 processor by a particular processor configuration.

6. *main*. The program starts in *main*. After initializing the Uniform System, *main* asks the user to supply 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 make a copy of 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.

4.3. 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

---

[15]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 "init" routine for *TimeTest*.

3 pixel x 3 pixel mask is used:

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

The value of each pixel in the output image is 8 times the value of the corresponding pixel in the input image minus the values of each of the 8 adjacent input image pixels.

Figure 9 is a typescript from running the program on a small Butterfly configuration. The convolution program is shown in Figure 10.

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 2 to generate the tasks[16].

The program has six routines:

1. *InitProblemOnce*. This routine allocates space in globally shared memory for the input (*im*) and output (*an*) images (square images of dimension $N \times N$ 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*. This is the "init" 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. *DoConvol*. This 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 means of block transfer.

   The four block transfer operations are motivated by two performance considerations. First, when referencing a large number of contiguous items, it is more efficient to first use block transfer to make a local copy of them and then reference the copied values locally, than it is to reference the items one at a time through the switch. After a small amount of setup, the block transfer occurs at the full 32 Mbit/second rate of the Butterfly switch, whereas the individual remote references do not, since setup overhead must be incurred for each remote reference. Using the block transfer operation to put frequently referenced data in local memory is the

---

[16]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.

```
[0] run -sars 200 convolve
loading convolve from VAX...

 init1 — find memory      init2 — map memory     init3 — start processors

Image size = 256

 please enter start, del(0=exp), and end for time test: 1 0 8

 using start = 1, delta = 0, end = 8
[1] time = 147647 ticks =   9.22 sec;  ep =   0.9;  eff = 0.9999
[2] time = 73397 ticks =   4.58 sec;  ep =   2.0;  eff = 1.0058
[4] time = 36786 ticks =   2.29 sec;  ep =   4.0;  eff = 1.0034
[8] time = 18498 ticks =   1.15 sec;  ep =   7.9;  eff = .9977
[0]
```

Figure 9: Typescript from the convolution program.

```
1  /*   Image convolution - unoptimized example program   */
2
3  #include <us.h>
4
5  #define true 1
6  #define false 0
7
8  int     N, End;
9  int     ** im, ** an;
10 int     * row, * row_m1, * row_p1, * row_ans;
11
12 InitProblemOnce ()
13 {   int i, j;
14     im=(int **)AllocScatterMatrix(N, N, sizeof(int));
15     an=(int **)AllocScatterMatrix(N, N, sizeof(int));
16     for (i = 0; i < N; i++)
17         for (j = 0; j < N; j++)
18             im[i][j] = i % 2;
19     Share(&N); Share(&im); Share(&an);
20 }
21
22 InitforProc(dummy)
23     int dummy;
24 {   End = N - 1;
25     row = (int *) malloc (N*sizeof(int));
26     row_m1 = (int *) malloc (N*sizeof(int));
27     row_p1 = (int *) malloc (N*sizeof(int));
28     row_ans = (int *) malloc (N*sizeof(int));
29 }
30
31 DoConvol(dummy ,r)
32     int dummy, r;
33 {   int c;
34     if (r & 1)
35         r = N-r-2;
36     Do_bt (im[r++], row_m1, N*sizeof(int));
37     Do_bt (im[r++], row, N*sizeof(int));
38     Do_bt (im[r--], row_p1, N*sizeof(int));
39     for (c = 1; c < End; c++)
40         row_ans[c] =  -row[c-1] + (row[c] << 3) - row[c+1]
41                       -row_m1[c-1] - row_m1[c] - row_m1[c+1]
42                       -row_p1[c-1] - row_p1[c] - row_p1[c+1];
43     Do_bt (row_ans, an[r], N*sizeof(int));
44 }
45
46 FinalforProc ()
47 {   free (row);
48     free (row_m1);
49     free (row_p1);
50     free (row_ans);
51 }
52
53 Body()
54 {   GenOnIFull(InitforProc, DoConvol, FinalforProc, 0, N-2, 0, false);
55 }
56
57 main()
58 {   int TimeTestPrint();
59     InitializeUs();
60     printf("\nImage size = ");  scanf("%d", &N);
61     InitProblemOnce();
62     TimeTest(0, Body, TimeTestPrint);
63 }
```

Figure 10: The convolution example program.

Butterfly 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[17]. 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[18] .

4. *FinalforProc.* This is the "final" 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.* This is the routine timed by *TimeTest.* It uses *GenOnIFull* to generate the tasks that compute rows of output image pixels in parallel.

6. *main.* The program starts with *main. Main* 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.

---

[17]As written, the program assumes that N is even.

[18]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.

5. Running and Tuning Uniform System Programs

This section presents information needed to run programs that use the Uniform System. In addition, it offers a few suggestions for tuning the multiprocessor performance of Uniform System programs.

Running Uniform System Programs

When a program uses the Uniform System, it has access to a large globally shared region of memory. Implementation of the shared memory region requires use of more hardware mapping registers (called *sars* for Segment Attribute Registers) than the Butterfly shell (*bshell*) *load* and *run* commands normally use for application programs. Therefore, the *bshell* must be instructed to use the required number of *sars* for Uniform System programs via the *-sars* option of the *load* or *run* command:

```
load -sars 200 program
run -sars 200 program
```

or via the *us* utility:

```
us program
```

The *us* utility is equivalent to *run -sars 200*; it also starts the program on a non-King node, sets the switch timeout to a value appropriate for Uniform System programs, (see discussion below) and enables alternate switch paths, if any (see discussion below).

If the *us* utility is not used, then it is often advisable to use the Chrysalis *toset* and *alten* utilities prior to running Uniform System programs, as described in the following paragraphs.

Chrysalis manages the value of a timeout controlling the length of time processor nodes will try to get a message (e.g., a request to read or write a remote memory location) through the Butterfly switch. Transmission of a message may fail for a variety of reasons: contention within the switch or at a memory, failure of a switching element, software or hardware failure of the destination processor node, and so forth. When an attempt to send a message fails, the sending node repeatedly retransmits the message until either the message is successfully transmitted or the timeout period elapses. If the timeout elapses before the message is successfully transmitted, Chrysalis signals an exception condition to the application program by means of a "throw"[19]. Chrysalis uses 10 milliseconds as a default timeout period.

[19] See the Chrysalis Programmers Manual or the appendix of the Butterfly Parallel Processor Overview for a discussion of the throw mechanism.

Experience has shown that this is too small for many programs on moderately sized and larger machines (>= 16 processors). The *toset* utility may be used to change the switch timeout to a larger value. A value of 4 seconds works well;

```
toset 4000
```

The switches for larger Butterfly systems (> 16 processors) are typically configured with alternate paths. Unless it is explicitly told to do so, Chrysalis will not use the alternate paths. The *alten* utility may be used to enable the use of alternate paths;

```
alten 2
```

enables the use of two paths between source and destination.

Performance Tuning

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.

For Uniform System programs, multiprocessor performance bottlenecks may occur for several reasons. Performance bottlenecks can occur if:

1. There are insufficient tasks;
2. The tasks are too small;
3. There is memory contention.

The following paragraphs briefly considers each of these.

If there are insufficient tasks, processor starvation occurring as task generators finish up can limit program performance. For example, assuming that there are 128 processors, consider the simple case of an application with 129 tasks, each of which takes about T time units to perform. One processor will perform 2 tasks and the remaining 127 processors a single task. Therefore, the time to run on 128 processors will be

$$2\ T$$

and the maximum speedup attainable over running on a single processor is

```
Max speedup = (Time on 1 proc) / (Time on 128 procs)
            = 129 T / 2 T
            = 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[20]. In some cases, this is straightforward. For example, if it were necessary to increase the number of tasks in the convolution example of Section 4, the number of tasks could be doubled by having each task process only half of an image row.

When the tasks are too short, poor performance may be due to two factors:

1. 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.

2. 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 "Building a Generator" in Section 3). 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 = Tgen + Twork
   ```

   Tgen is made up of time spent in the critical region *Tcrit* and in the noncritical region (*Tnoncrit*); hence,

   ```
   T = Tcrit + Tnoncrit + Twork
   ```

   Letting *Trest* be the sum of *Tnoncrit* and *Twork* gives

   ```
   T = Tcrit + Trest.
   ```

   Since processors must proceed through the critical region serially, the maximum number of processors that can be fully utilized[21] is:

---

[20] 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.

[21] That is, used without waiting to proceed though the critical region.

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

$$\sim T_{rest} / T_{crit} \quad \text{for } T_{rest} \gg T_{crit}.$$

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 / Tcrit* effective processors.

The effects of both factors can be minimized by increasing task length. The convolution example in Section 4 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.

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 program profiling utility can be used to determine where, if at all, there is significant memory contention. Consult the Chrysalis 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:

1. Distribute the program's data across the machine. *AllocScatterMatrix* can be used to do this.
2. 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.
3. 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[22]. Of course, if the copied data changes as the computation proceeds and multiple processors need to see the changes,

---

[22] If there are n copies, processor p would access copy (p mod n).

managing the copies can become complex.

4. Make local cache copies of data structures before referencing them, as in the convolution example in Section 4. *Do_bt* can be used to do this.

I. Appendix

Compiling, Loading, and Running Programs

This appendix gives step by step instructions for compiling, linking and running Uniform System programs on the Butterfly Parallel Processor. The Butterfly Parallel Processor Tutorial and the Chrysalis Programmers Manual should be consulted for more detailed information on the mechanics of using the Butterfly system.

The instructions below refer to various directories, header files, tools, etc. These all come on Butterfly software distribution tapes. Instructions for installing software distribution tapes on your front end host can be found in the Chrysalis Programmers Manual.

STEP 1: Set up search paths on the Unix front end machine. You will need to access several directories besides the one containing your sources. To do this you will have to edit your *.profile* file (if a bourne shell user) or your *.login* file (if a cshell user).

The changes for the borne shell are:

1. Add *BFlyDir/bin* to your Unix search path, where *BFlyDir* is the root Butterfly directory (usually /usr/butterfly).

2. Add to your *.profile* file the lines:

```
CHRYS=BFlyDir/chrys/release
B_PATH=:.:$CHRYS/tools:$CHRYS/net-tools:$CHRYS
export B_PATH CHRYS
```

   where *BFlyDir* is the root Butterfly directory (usually /usr/butterfly).

The changes for the cshell are:

1. Add *BFlyDir/bin* to your Unix search path, where *BFlyDir* is the root Butterfly directory (usually /usr/butterfly).

2. The changes to your *.login* file for the cshell are:

```
setenv CHRYS BFlyDir/chrys/release
setenv B_PATH .:$CHRYS/tools:$CHRYS/net-tools:{$CHRYS}
```

   where *BFlyDir* is the root Butterfly directory (usually /usr/butterfly).

STEP 2: Set up your directory. Make a directory on the front end machine in which to build your program.

Construct with your favorite editor both a source file (extension *.c*) and a makefile for your program. We strongly recommend that you start with an existing template for both of these files[23]. Figure 8, the matrix multiply example program, and Figure 11 are suitable to serve as templates. (These templates may be found in the directory *$(CHRYS)/us* in the files *MatrixExample.c* and *us-prog.makefile*.)

STEP 3: Customize your makefile. Edit the first line in the makefile template replacing *name* with the name of your program. If your program is named *prog.c*, the line should be:

```
MINE = prog
```

If your program requires several files to be linked together, you will need to change one of the "rules" in the makefile template – consult the Unix documentation for *make*.

STEP 4: You are now ready to edit, compile, and run programs. Construct your source file. It must be in the directory you just created, and it must have extension .c. At the start of the file, add the line:

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

This incorporates a few definitions that allow compilation of the same program for both the Butterfly and the front end without modification.

STEP 5: Compile your file. To compile for the front end, *make prog.out*. To compile for the Butterfly, *make prog.68*.

STEP 6: Debug your program on the front end machine. *Make prog.out* creates the usual front end executable version of your program; debug it using whatever techniques you prefer.

STEP 7: Go to a Butterfly console terminal.

STEP 8: Type *ctl-c*. If you don't get a prompt, consult your local Butterfly system expert. You are now talking to USD; a very simple minded ROM-based loader/debugger program[24].

STEP 9: Type *T* (capitals matter), then *return*. You should now be speaking to the front end. Log out if necessary, then log in and change directory to the one that contains your program.

NOTE: You can get to the front end whenever you want by typing *ctl-c* and *T*. If you wish, you can edit and compile while connected to the front end in this manner.

---

[23] Alternatively, you might choose to use the *genmake* utility to construct your makefile. Consult the Chrysalis Programmers Manual for details.

[24] There are several variations to the scenario described in this step and steps 9 through 13. Consult the Butterfly Parallel Processor Tutorial for details.

STEP 10: Decide whether you want to reload Chrysalis. Usually you won't need to. If you need to reload Chrysalis, reenter the front end and execute:

```
bld  chrys.68
```

STEP 11: Start Chrysalis. Return to the Butterfly by typing *ctl-c*, then type *G*.

STEP 12: Configure the machine. Chrysalis will tell you which processors are available, and which you are currently using. It will then ask if you want to change things around. The dialog is self explanatory. Usually, it is fine to use all of the processors that are available.

STEP 13: When Chrysalis asks you for the terminal type you are using, answer it. After you respond, Chrysalis will clear your screen and print a prompt of the form "[n]" where n is the number of the processor node that your terminal happens to be connected to. At this point, you are talking to to the Butterfly Shell (*bshell*) command interpreter. The commands that the *bshell* will execute are documented in the Chrysalis Programmers Manual. Only the bare minimum are given below.

STEP 14: To run your program, type the line:

```
us prog
```

where *prog* is the name of your program[25]. Your program will now be in control. If it executes correctly, you are done and you may logout.

STEP 15: If your program doesn't execute correctly, it will hang, print forever, or print some error message. You have a couple of options at this point. If you do not care to learn more about the detailed workings of the Butterfly, you must use *printf*'s (the standard C programmer's fallback). Otherwise, you can learn to read the information that appears in the error messages, and you can learn about the debugging tools.

STEP 16: The surest and simplest way to regain control of the machine is to type *ctl-c*. After that, you can type *G* to restart Chrysalis. There are less catastrophic ways to stop the machine[26] which will save you time in the compile/edit/debug cycle (restarting Chrysalis is not a quick operation). Typing *ctl-g k* will terminate your program and return you to *bshell*. At this point you can return to the front end by typing *ctl-g h*. After editing and recompiling your program you can return to the Butterfly by typing *ctl-g x*, where *x* is any character other than *ctl-g*; to send *ctl-g* to the front end, type

---

[25] The *us* utility provides a simple way for starting Uniform System programs which is adequate for many situations. See Section 5 for a discussion of other ways to start a Uniform System program.

[26] See the Butterfly Parallel Processor Tutorial for details.

ctl–g ctl–g. To run your program again, you should first remove the old version by typing

```
rm prog
```

to the *bshell*. Now you can run the new version of your program by going back to STEP 14.

You should now be able to go through the compile/edit/debug cycle.

```
MINE        =   name
BF          =   Butterfly root directory
VER         =   release
CHRYS       =   $(BF)/chrys/$(VER)
u           =   $(CHRYS)/us
IDIR        =   $(CHRYS)/include
IFLAGS      =   -I$(IDIR)
x           =   $(CHRYS)/chrys.68
h           =   $(CHRYS)/include
CC68        =   $(BF)/bin/bcc
CC68FLAGS   =   -O -DBFLY
LNK68LIBS   =   $(CHRYS)/lib/libcs.a $(CHRYS)/lib/libtools.a
CFLAGS      =   -c -Im -g
LFLAGS      =   -Im -g
prefix      =   $(CHRYS)/lib/boss.o68
sys         =   $x $(prefix) $(LNK68LIBS)
hdrs        =   $h/public.h $h/stdio.h $h/us.h

all:    $(MINE).68 $(MINE)

# Object file dependencies

$(MINE).68:       $(sys) $(MINE).o68
$(MINE).out:      $u/us.o $(MINE).o

# Source file dependencies

$(MINE).o68:      $(hdrs)

#-------------------------------------------------------------------

.SUFFIXES:
.SUFFIXES:  .out  .o
.SUFFIXES:  .68 .o68 .a68 .s68 .c68 .c

# Source file suffixes:
# .c68  - C source code file
# .a68  - assembler source code file

# Intermediate file suffixes:
# .o68  - relocatable output of assembler (butterfly)
# .o    - relocatable output of assembler (vax)
# Executable file suffixes:
# .68   - executable 68000 file in a.out format (stable version)

.c68.o68:       ; rm -f $*.168; $(CC68) $(IFLAGS) $(CC68FLAGS) -c  $*.c68
.c.o68:         ; rm -f $*.168; $(CC68) $(IFLAGS) $(CC68FLAGS) -c  $*.c
.o68.68:        ; $(CC68) -o $@ $*.o68 $(LNK68LIBS); splitsyms $@
.c.o:           ; rm -f $*.o; cc -I$(u) $(CFLAGS) -O $*.c
.o.out:         ; cc -o $* $*.o $u/us.o $(LFLAGS); touch $*.out
```

Figure 11: "Makefile" template for Uniform System programs.

II. Appendix

Uniform System Library Routines

This appendix documents each of the operations supported by the Uniform System Library. The operations are ordered alphabetically. The page references refer to the narrative descriptions for the various operations in Section 3.

- AbortGen

```
AbortGen(GenHandle, code)                                    page 23
UsGenDesc * GenHandle;
int code;
```

Abort the active task generator specified by *GenHandle* by preventing the generation of new tasks. Any tasks in progress will run to completion. The value of *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.

GenHandle must specify an abortable generator.

- ActivateGen

```
UsGenDesc *
ActivateGen(Init, Worker, Final, Arg, Range1, Range2, Type      page 35
            Gen, Async, MaxProcsToUse Abortable, ResultP)
int (* Init)(), (* Worker)(), (* Final)();
int Arg, Range1, Range2, Type;
int (* Gen)();
int Async, MaxProcsToUse, Abortable, * ResultP;
```

ActivateGen is the "universal" generator activator procedure. It is called by all of the *GenOn...* generator activator procedures. *ActivateGen* may be used directly by application programs to construct new generators.

As with the generators described elsewhere in this appendix, *Init*, *Worker*, and *Final* are respectively, the per processor initialization routine, the task worker routine, and 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 *GENERATOR*, and *Range1* and *Range2* are integers. *Gen* is a task generation routine described in more detail below. *Async* is a boolean that specifies if the generator is synchronous (*true*) or asynchronous *false*. *MaxProcsToUse* specifies the processor limit for the generator; *0* or *-1* indicates no processor limitation; a positive values indicates the maximum number of processors to be used on the generator. *Abortable* is a boolean which indicates whether the generator is to be abortable. Finally, *ResultP* is a pointer used when *Abortable* is *true*; it specifies a location where the generator "result code" should be stored if the generator is aborted (so that the generator activator routine can find it).

The task generation routine is of the form:

```
Gen(TD);
UsGenDesc * TD;
```

where *TD* is a pointer to a task descriptor data structure in globally shared memory of the form (the type *UsGenDesc* is defined in *usgen.h* an *#include* file which must be used when *ActivateGen* is used):

```
struct
{  short started;
   short type;
   /* Defined types are: */
        #define IDLETASK  1
        #define GOAWAYTASK 2
        #define GENERATOR 4
   short incarnation_number;
   short state;
   /* Defined states are: */
        #define ACTIVE 1
        #define INACTIVE 2
   short us_lock;
   short lock;
   int (*init)();
   int   (*call)();
   int   (*gen)();
   int   (*final)();
   int   arg;
   char *currentShare;
   int   range;
   int   range2;
   QH returnQ;
   int   post_pending;
   short MaxProcsToUse;
   int   end;
   int   abortable;
   short retcode;
   /* Defined retcodes are: */
        #define genEXHAUSTED -1
   short endlock;
   union {long Long; short Short;} index;
   union {unsigned long Long; unsigned short Short;} index2;
   short locka[nlocks];
   short index1a[nlocks];
   short index2a[nlocks];
}
```

The *Worker*, *Init*, *Final*, *Gen*, *Arg*, *Range1*, *Range2*, *Type*, *MaxProcsToUse*, and *Abortable* parameters of *ActivateGen* are used to initialize the *call*, *init*, *final*, *gen*, *arg*, *range*, *range2*, *type*, *MaxProcsToUse*, and *abortable* fields of the task descriptor data structure. The *lock* and the *locka* array fields are initialized to 0 and are available for use as locks by the *Gen* routine; and, the *index* and *index2* fields, and the *index1a* and *index2a* array fields of the task descriptor data structure are initialized to 0 and are available for use by the *Gen* routine for bookkeeping associated with generating the tasks. The remaining fields (e.g., *started*, *state*, *shareCount*, *returnQ*, etc.) are used by *ActivateGen* for internal bookkeeping.

After *ActivateGen* initializes the task descriptor data structure, it makes the descriptor accessible to other processors. If *Async* is *true*,, *ActivateGen* then returns control to its caller along with a pointer to the task descriptor data structure; otherwise, the processor on which *ActivateGen* is invoked calls the *Gen* task generation procedure. That processor, and others as they become free, use the task generator descriptor (*TD*) and the

Gen task generation procedure to generate and execute calls on the *Worker* procedure.

- Allocate

```
char * Allocate(size)                                          page 17
int size;
```

Allocate a block of storage of *size* bytes in globally shared memory. The block is allocated from the memory with the most free space.

- AllocateC

```
char * AllocateC(size, class)                                  page 33
int size, class;
```

Allocate a block of storage of *size* bytes in globally shared memory. The block is allocated from the memory in the *class* specified with the most free space. See also *UsSetClass*.

- AllocateLocal

```
char * AllocateLocal(size)                                     page 17
int size;
```

Allocate from the memory of the local processor a block of globally shared storage of *size* bytes.

- AllocateOnPhysProc

```
char * AllocateOnPhysProc(physproc, size)                      page 17
int physproc, size;
```

Allocate from the memory of the processor whose hardware processor number is *physproc* a block of globally shared storage of *size* bytes.

- AllocateOnUsProc

```
char * AllocateOnUsProc(proc, size)                            page 17
int proc, size;
```

Allocate from the memory of the processor whose Uniform System processor number is *proc* a block of globally shared storage of *size* bytes.

- AllocateOnUsProcC

```
char * AllocateOnUsProcC(proc, size, class)                    page 33
int proc, size, class;
```

Similar to *AllocateOnUsProc*, differing in that the block of memory will be allocated only if the processor is in the specified *class*; otherwise, it fails. See also *UsSetClass*.

- AllocScatterMatrix

```
char * * AllocScatterMatrix(nrows, ncolumns, element_size)     page 18
int nrows, ncolumns, element_size;
```

Allocate a matrix that is scatted by row over the memories of the machine. A vector of pointers *nrows* long is allocated, and *nrows* separate vectors, each containing *ncols* items of size *element_size* bytes. The vectors are allocated in separate memories. The vector of pointers, a pointer to which is returned to the caller, is filled in with pointers to the scattered row vectors. Elements of an array *A* allocated in this way can be referenced using standard C array notation:

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

o AllocScatterMatrixC

```
char * * AllocScatterMatrixC(nrows, ncolumns,                    page 33
                             element_size, class)
int nrows, ncolumns, element_size;
```

Similar to *AllocScatterMatrix*, differing in that only memories of the machine that are in the specified *class* are used to hold the scattered rows of the matrix and the vector of row pointers. See also *AllocScatterMatrix* and *UsSetClass*.

o AsyncGenOnA

```
UsGenDesc *
AsyncGenOnA(Worker, Range1, Range2)                              page 27
int (* Worker)();
int Range1, Range2;
```

Asynchronous version of *GenOnA*. *AsyncGenOnA* is equivalent to

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

o AsyncGenOnAAbortable

```
UsGenDesc *
AsyncGenOnAAbortable(Worker, Range1, Range2)                     page 27
int (* Worker)();
int Range1, Range2;
```

Asynchronous version of *GenOnAAbortable*. *AsyncGenOnAAbortable* is equivalent to

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

o AsyncGenOnAFull

```
UsGenDesc *
AsyncGenOnAFull(Init, Worker, Final, Arg, Range1, Range2,        page 27
                Limited, Abortable)
int (*Init)(), (* Worker)(), (* Final)();
int Arg, Range1, Range2, Limited, Abortable;
```

Asynchronous version of *GenOnAFull*. *AsyncGenOnAFull* 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. *AsyncGenOnAFull* returns a generator handle that can be used with *WorkOn* or *WaitForTasksToFinish*. See the description of *GenOnAFull* for an explanation of the parameters.

o AsyncGenOnALimited

```
UsGenDesc *
AsyncGenOnALimited(Worker, Range1, Range2, MaxProcsToUse)        page 27
int (* Worker)();
int Range1, Range2, MaxProcsToUse;
```

Asynchronous version of *GenOnALimited*. *AsyncGenOnALimited* is equivalent to

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

- AsyncGenOnHA

```
UsGenDesc *
AsyncGenOnHA(Worker, Range1, Range2)                             page 27
int (* Worker)();
int Range1, Range2;
```

Asynchronous version of *GenOnHA*. *AsyncGenOnHA* is equivalent to

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

- AsyncGenOnHAAbortable

```
UsGenDesc *
AsyncGenOnHAAbortable(Worker, Range1, Range2)                    page 27
int (* Worker)();
int Range1, Range2;
```

Asynchronous version of *GenOnHAAbortable*. *AsyncGenOnHAAbortable* is equivalent to

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

- AsyncGenOnHAFull

```
UsGenDesc *
AsyncGenOnHAFull(Init, Worker, Final, Arg, Range1, Range2,       page 27
                 Limited, Abortable)
int (* Init)(),  (* Worker)(), (* Final)();
int Arg, Range1, Range2, Limited, Abortable;
```

Asynchronous version of *GenOnHAFull*. *AsyncGenOnHAFull* 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. It returns a generator handle that can be used with *WorkOn* or *WaitForTasksToFinish*. See the description of *GenOnHAFull* for an explanation of the parameters.

- UsGenDesc * AsyncGenOnHALimited

```
AsyncGenOnHALimited(Worker, Range1, Range2, MaxProcsToUse)       page 27
int (* Worker)();
int Range1, Range2, MaxProcsToUse;
```

Asynchronous version of *GenOnHALimited*. *AsyncGenOnHALimited* is equivalent to

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

- AsyncGenOnI

```
UsGenDesc *
AsyncGenOnI(Worker, Range)                                    page 27
int (* Worker)();
int Range;
```

Asynchronous version of *GenOnI*. *AsyncGenOnI* is equivalent to

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

o AsyncGenOnIAbortable

```
UsGenDesc *
AsyncGenOnIAbortable(Worker, Range)                           page 27
int (* Worker)();
int Range;
```

Asynchronous version of *GenOnIAbortable*. *AsyncGenOnIAbortable* is equivalent to

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

o AsyncGenOnIFull

```
UsGenDesc *
AsyncGenOnIFull(Init, Worker, Final, Arg, Range,              page 27
                Limited, Abortable)
int (* Init)(), (* Worker)(), (* Final)();
int Arg, Range, Limited, Abortable;
```

Asynchronous version of *GenOnIFull*. *AsyncGenOnIFull* 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. It returns a generator handle that can be used with *WorkOn* or *WaitForTasksToFinish*. See the description of *GenOnIFull* for an explanation of the parameters.

o AsyncGenOnILimited

```
UsGenDesc *
AsyncGenOnILimited(Worker, Range, MaxProcsToUse)              page 27
int (* Worker)();
int Range, MaxProcsToUse;
```

Asynchronous version of *GenOnILimited*. *AsyncGenOnILimited* is equivalent to

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

o Atomic_add_long

```
Atomic_add_long(loc, val)                                     page 12
int * loc, val;
```

Atomically add *val* to the location pointed to by *loc*. *Atomic_add_long* is similar to the Chrysalis 16 bit *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" functionality provided by *Atomic_add*.

It is also important to note that in its current implementation *Atomic_add_long* is atomic only with respect to other *Atomic_add_long* calls. In particular, ii is possible for the execution of a read operation to 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) is propagated to the higher order bits.

- BEGIN_SHARED_DECL

```
BEGIN_SHARED_DECL                                          page 30
    ...
    normal C declarations;
    ....
END_SHARED_DECL;
```

BEGIN_SHARED_DECL is a macro. It is used with *END_SHARED_DECL* to delimit the declaration of variables that are to be globally shared among all of the processors. Variables declared in this way are referenced using the *SHARED* prefix. Space for variables declared in this way must be allocated via *MakeSharedVariables* after *InitializeUs* is called and before they are referenced.

Only one *BEGIN_SHARED_DECL/END_SHARED_DECL* declaration can appear in a Uniform System program.

All of the variables declared via *BEGIN_SHARED_DECL/END_SHARED_DECL* are allocated on the same physical memory. In some situations this may lead to memory contention.

- ConfigureUs

```
ConfigureUs(Spec, n)                                       page 33
int * Spec, n;
```

ConfigureUs can be used prior to calling *InitializeUs* to specify values for configuration parameters that differ from the values normally used by *InitializeUs*. *Spec* is an array (of *int*'s) which specifies the configuration; 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) integer(s) which specify the value for the parameter.

The following *configuration_code*'s are currently defined:

Code	Parameter
configProcs	integer = number of processors to include in Uniform System configuration.
configMaxSars	integer = number of segment attribute registers (SARs) to use to define process address spaces.

- DistinctMemoriesAvailable

```
DistinctMemoriesAvailable()                                page 10
```

Return an integer which is 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.

- END_SHARED_DECL page 30

  END_SHARED_DECL is a macro used with *BEGIN_SHARED_DECL* to delimit the declaration of variables that are to be globally shared.

- FreeAll

  ```
  FreeAll()                                                    page 18
  ```

  Deallocate all globally allocated storage.

- GenOnA

  ```
  GenOnA(Worker, Range1, Range2)                               page 25
  int (* Worker)();
  int Range1, Range2;
  ```

  Generate tasks that cause *Worker(0, index1, index2)* to be executed in parallel for all combinations of *index1* and *index2* for *0 <= index1 < Range1* and *0 <= index2 < Range2*. The processor that invokes *GenOnA*, and possibly other processors, will be used to execute the tasks generated. When *GenOnA* returns, all of the tasks generated will have been completed.

- GenOnAAbortable

  ```
  GenOnAAbortable(Worker, Range1, Range2)                      page 25
  int (* Worker)();
  int Range1, Range2;
  ```

  Abortable version of *GenOnA*. The tasks generated are calls of the form *Worker(0, index1, index2, GenHandle)*, where *GenHandle* is an identifier for the task generator that can be used with *AbortGen* to abort it. *GenOnAAbortable* is equivalent to

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

  Note that *GenOnAAbortable* returns a value which indicate whether *AbortGen* was used to abort the generator.

- GenOnAFull

  ```
  GenOnAFull(Init, Worker, Final, Arg, Range1, Range2)         page 24
             Limited, Abortable)
  int (* Init)(), (* Worker)(), (* Final)();
  int Arg, Range1, Range2, Limited, Abortable;
  ```

  Generate tasks that cause *Worker(Arg, index1, index2)* (if *Abortable* is *false*) or *Worker(Arg, index1, index2, GenHandle) (if Abortable is true)* to be executed in parallel for all combinations of *index1* and *index2* for *0 <= index1 < Range1* and *0 <= index2 < Range2*. The processor that invokes *GenOnAFull*, and possibly other processors, will be used to execute the tasks generated.

  The routine *Init(Arg)* is called on each processor used to execute the tasks generated before the *Worker* routine is called for the first time on the processor. The routine *Final(Arg)* is called on each processor used to execute the tasks generated after the *Worker* routine is called for the last

time on the processor. The *Limited* parameter is used to control the number of processors used by the generator; if *Limited* is *0* or *-1* there is no limitation on the number of processors; a positive value limits the processors used to that number or fewer.

When *GenOnAFull* returns, either: all of the tasks generated will have been completed in which case *GenOnAFull* returns the value *genEXHAUSTED*; or the *Abortable* parameter was *true*, the generator was aborted, and some of the tasks may not have been performed, in which case *GenOnAFull* returns the code passed to *AbortGen* when it was aborted..

o GenOnALimited

```
GenOnALimited(Worker, Range1, Range2, MaxProcsToUse)          page 25
int (* Worker)();
int Range1, Range2, MaxProcsToUse;
```

"Limited" version of *GenOnA*. *GenOnALimited* is equivalent to

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

o GenOnHA

```
GenOnHA(Worker, Range1, Range2)                               page 25
int (* Worker)();
int Range1, Range2;
```

Generate tasks that cause *Worker(Arg, index1, index2)* to be executed in parallel for combinations of *index1* and *index2* that span the "half" array beneath the diagonal of a *Range1 x 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)
```

The processor that invokes *GenOnHA*, and possibly other processors, will be used to execute the tasks generated. When *GenOnHA* returns, all of the tasks generated will have been completed.

o GenOnHAAbortable

```
GenOnHAAbortable(Worker, Range1, Range2)                      page 25
int (* Worker)();
int Range1, Range2;
```

Abortable version of *GenOnHA*. The tasks generated are calls of the form *Worker(0, index1, index2, GenHandle)*, where *GenHandle* is an identifier for the task generator that can be used with *AbortGen* to abort it. *GenOnHAAbortable* is equivalent to

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

Note that *GenOnHAAbortable* returns a value which indicate whether *AbortGen* was used to abort the generator.

o GenOnHAFull

```
GenOnHAFull(Init, Worker, Final, Arg, Range1, Range2,          page 25
            Limited, Abortable)
int (* Init)(), (* Worker)(), (* Final)();
int Arg, Range1, Range2, Limited, Abortable;
```

Generate tasks that cause *Worker(Arg, index1, index2)* (if *Abortable* is *false*) or *Worker(Arg, index1, index2, GenHandle) (if Abortable is true)* to be executed in parallel for combinations of *index1* and *index2* that span the "half" array beneath the diagonal of a *Range1 x 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)
```

The processor that invokes *GenOnHAFull*, and possibly other processors, will be used to execute the tasks generated.

The routine *Init(Arg)* is called on each processor used to execute the tasks generated before the *Worker* routine is called for the first time on the processor. The routine *Final(Arg)* is called on each processor used to execute the tasks generated after the *Worker* routine is called for the last time on the processor. The *Limited* parameter is used to control the number of processors used by the generator; if *Limited* is *0* or *-1* no limitation is placed on the number of processors; a positive value limits the processors used to that number or fewer.

When *GenOnHAFull* returns, either: all of the tasks generated will have been completed in which case *GenOnHAFull* returns the value *genEXHAUSTED*; or the *Abortable* parameter was *true*, the generator was aborted, and some of the tasks may not have been performed, in which case *GenOnHAFull* returns the code passed to *AbortGen* when it was aborted..

o GenOnHALimited

```
GenOnHALimited(Worker, Range1, Range2, MaxProcsToUse)          page 25
int (* Worker)();
int Range1, Range2, MaxProcsToUse;
```

"Limited" version of *GenOnHA*. *GenOnHALimited* is equivalent to

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

o GenOnI

```
GenOnI(Worker, Range)                                          page 24
int (* Worker)();
int Range;
```

Generate tasks that cause *Worker(0, index)* to be executed in parallel for all values of *index* in the range *0 <= index < Range*. The processor that invokes *GenOnI*, and possibly other processors, will be used to execute the tasks generated. When *GenOnI* returns, all of the tasks generated will have been completed.

o GenOnIAbortable

```
GenOnIAbortable(Worker, Range)                                    page 24
int (* Worker)();
int Range;
```

Abortable version of *GenOnI*. The tasks generated are calls of the form *Worker(0, index, GenHandle)*, where *GenHandle* is an identifier for the task generator that can be used with *AbortGen* to abort it. *GenOnIAbortable* is equivalent to

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

Note that *GenOnIAbortable* returns a value which indicate whether *AbortGen* was used to abort the generator.

o GenOnIFull

```
GenOnIFull(Init, Worker, Final, Arg, Range,                       page 22
           Limited, Abortable)
int (* Init)(), (* Worker)(), (* Final)();
int Arg, Range, Limited, Abortable;
```

Generate tasks that cause *Worker(0, index)* (if *Abortable* is *false*) or *Worker(Arg, index, GenHandle) (if Abortable is true)* to be executed in parallel for all values of *index* in the range *0 <= index < Range*. The processor that invokes *GenOnIFull*, and possibly other processors, will be used to execute the tasks generated.

The routine *Init(Arg)* is called on each processor used to execute the tasks generated before the *Worker* routine is called for the first time on the processor. The routine *Final(Arg)* is called on each processor used to execute the tasks generated after the *Worker* routine is called for the last time on the processor. The *Limited* parameter is used to control the number of processors used by the generator; if *Limited* is *0* or *-1* there is no limitation on the number of processors; a positive value limits the processors used to that number or fewer.

When *GenOnIFull* returns either: all of the tasks generated will have been completed in which case *GenOnIFull* returns the value *genEXHAUSTED*; or the *Abortable* parameter was *true*, the generator was aborted, and some of the tasks may not have been performed, in which case *GenOnIFull* returns the code passed to *AbortGen* when it was aborted.

o GenOnILimited

```
GenOnILimited(Worker, Range, MaxProcsToUse)                       page 24
```

"Limited" version of *GenOnI*. *GenOnILimited* is equivalent to

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

o GenTaskForEachProc

```
GenTaskForEachProc(Worker, Arg)                                   page 26
int (* Worker)();
int Arg;
```

Generate exactly one task of the form *Worker(Arg)* for every processor.

o GenTaskForEachProcLimited

```
GenTaskForEachProcLimited(Worker, Arg, NProcs)                    page 26
int (* Worker)();
int Arg, NProcs;
```

Generate exactly one task of the form *Worker(Arg)* for each of NProcs processor.

WARNING: If *ProcsInUse()* is less than *NProcs*, this call will hang.

o GenTasksFromList

```
GenTasksFromList(Routine_List, Arg_List, n)                       page 26
int * (* RoutineList)();
int * Arg_List;
int n;
```

Routine_List is a list of *n* routines, *r1,...,rn*, and *Arg_List* is a list of *n* arguments, *arg1,...,argn*. *GenTasksFromList* generates n tasks, where the *ith* task is of the form *ri(argi)*.

o GetRtc

```
GetRtc()                                                          page 34
```

Return the time since the system was booted in units of 62.5 microseconds.

o InitializeUs

```
InitializeUs()                                                    page 10
```

Initialize the Uniform System. This includes creating and starting a Uniform System process on every available processor, setting up the memory that is globally shared among all Uniform System processes, and initializing the Uniform System storage allocator. *InitializeUs* must be called before using any other Uniform System routine, and it should be called only once.

o InitializeUsForBenchMark

```
InitializeUsForBenchMark()                                        page 10
```

Initialize the Uniform System. Similar to *InitializeUs*, differing in that the King Node will not be used by the program if the program is started on a non-King node (via the *-on* switch of the *run* command or the *us* utility). This is useful when benchmarking a program, where it is desirable that the measurements not be affected by the processing requirements of the terminal handler and window manager which run on the King Node.

o LOCK

```
LOCK(lock, n)                                                     page 12
short * lock;
int n;
```

Set the "lock" specified by *lock*. The *short* pointed to by *lock* is assumed to have been initialized in the unset state to the value 0. *LOCK* implements a "busy wait" type of lock. The *int n* 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 1 millisecond. *LOCK* does not return until it has set the lock. (See *UNLOCK*.)

- MakeSharedVariables

```
MakeSharedVariables;                                    page 30
```

This is a macro. It allocates space in globally shared memory for variables declared as globally shared (via *BEGIN_SHARED_DECL* and *END_SHARED_DECL*) and makes the location of the variables known to other processors. *MakeSharedVariables* should be called after *InitializeUs*, and only if *BEGIN_SHARED_DECL* and *END_SHARED_DECL* have been used.

- MemoriesAvailable

```
MemoriesAvailable()                                     page 10
```

Return an integer that is the amount of memory available to the application program. The value returned is in units of 64 KBytes.

- PhysProcToUsProc

```
PhysProcToUsProc(PhysProc)                              page 11
int PhysProc;
```

Return the Uniform System processor number corresponding to the physical processor number *PhysProc*.

- ProcsInUse

```
ProcsInUse()                                            page 10
```

Return an integer which is the number of processors available to an application program. The value returned will not count any processors which have been removed by the *TimeTest* or *TimeTestFull* routines.

- Share

```
Share(N)                                                page 28
int * N;
```

Pass 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 4 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

```
ShareBlk(X, size)                                       page 28
int * X;
int size;
```

Pass the block of data of *size* bytes pointed to by *X* to all processors used to execute tasks generated subsequently. *X* 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 *X* (in the processor invoking *ShareBlk* at the time *ShareBlk* is invoked) to be copied into the

location beginning at *X* in each processor used to perform tasks generated by task generators activated subsequent to the call of *ShareBlk*.

- SHARED page 31

SHARED is a macro. It is used as a prefix to access variables which have been declared as globally shared using *BEGIN_SHARED_DECL/END_SHARED_DECL*. For example, if *N* has been declared in this way, it may be referenced as *SHARED N*:

```
SHARED N = SHARED N % 7;
```

WARNING: before such a variable can be referenced, it must space for it must be allocated using *MakeSharedVariables*.

- SharePtrAndBlk

```
SharePtrAndBlk(P, size)                                    page 28
int * * P;
int size;
```

Pass the pointer pointed to by *P* and the block of data of *size* 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* causes 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 *SharePtrAndBlk* is invoked) to be made for each processor used to perform tasks generated by task generators activated subsequent to the call of *SharePtrAndBlk* as follows: 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*.

- ShareScatterMatrix

```
ShareScatterMatrix(P, nrows)                               page 28
int * * * P;
int nrows;
```

P points to a global or static variable allocated by

```
AllocScatterMatrix(nrows, ncols, element_size)
```

ShareScatterMatrix makes a copy of the vector of row pointers allocated by *AllocScatterMatrix* 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 functionally equivalent to *SharePtrAndBlk*, but operates much faster, since it is careful to make its copies from other copies as well as from the original.

- TimeTest

```
TimeTest(Init, Execute, PrintResults)                      page 31
int (* Init)(), (* Execute)(), (* PrintResults)();
```

Time execution of the routine *Execute* on various processor configurations as specified by the user from the keyboard. *TimeTest* runs the routines *Init*,

Execute, and *PrintResutls* 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 (time k proc) / (k * (time n procs)).

The *PrintResults* routine is specified by the application program. The Uniform System Library contains a routine (see *TimeTestPrint* below) 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)     page 32
int (* Init)(), (* Execute)(), (* PrintResults)();
int start, delta, end;
```

TimeTestFull is similar to *TimeTest* (see above). 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)                          page 32
int runtime, procs;
float effprocs;
```

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 (= speedup achieved over 1 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*.)

o TotalProcsAvailable

```
TotalProcsAvailable()                                    page 10
```

Return 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*.

o UNLOCK

```
UNLOCK(lock)                                             page 12
short * lock;
```

Clear the lock specified by *lock*. (See *LOCK*.)

o UsProcToPhysProc

```
UsProcToPhysProc(UsProc)                                 page 11
int UsProc;
```

Return the physical processor number corresponding to the Uniform System processor number *UsProc*.

o UsSetClass

```
UsSetClass(proc, class)                                  page 33
int proc, class;
```

Add the memory of the specified Processor Node to the specified *class*. Initially all memories are in class 0. See also *AllocateC*, *AllocScatterMatrixC*, *AllocateOnUsProcC*.

o UsWait

```
UsWait(n)                                                page 13
int n;
```

Wait for *10 * n* microseconds. Using zero for *n* forces use of a default which is about 1 millisecond. *UsWait* is a "busy wait".

o WaitForTasksToFinish

```
WaitForTasksToFinish(GenHandle)                          page 27
UsGenDesc * GenHandle;
```

Wait 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*.

o WorkOn

```
WorkOn(GenHandle)                                        page 26
UsGenDesc * GenHandle;
```

Work 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*.

BBN Advanced Computers Inc.

A Subsidiary of Bolt Beranek and Newman Inc.

10 Fawcett Street
Cambridge, MA 02238
Telephone (617) 873-6000