A Common Library Interface to Shared-Memory Multiprocessors

Lawrence A. Crowl

Computer Science Department, Oregon State University

Abstract

In writing a highly-portable parallel program, we developed a library of parallel primitives to shield our application code from the various multiprocessors. Rather than adapt the different multiprocessors to a program-specific library via extensive implementation, we chose to first find a common 'intersection' of the virtual machine models provided by each vendor, define a common interface to that model, and then develop our application based on that model.

This paper describes the resulting common virtual machine model and a library interface to that model. We have implementations for that model on seven different shared-memory multiprocessors, and two different uniprocessors. Each implementation averages less than 100 lines of code. The common virtual machine model requires more programming effort than any one vendor's library, but the resulting program executes reasonably efficiently on all machines.

1: Introduction

Despite recent parallel language standardization efforts (e.g. PCF [15], ANSI Parallel C [16], and HPF [8]), application programmers cannot yet expect that any given parallel programming language will be available on a wide variety of machines [14]. Indeed, K&R C [10] and Fortran 77 [1] are the only languages widely available on multiprocessors. To overcome the sequential nature of these languages, vendors usually provide parallel programming primitives via user-callable libraries. Unfortunately, the vendor-provided libraries often differ substantially. In addition to providing differing mechanisms to create and synchronize processes; libraries often have different models of how the processes share and allocate memory within their address space, that is, the libraries present different virtual machine models.

Programmers can overcome the differences in vendor-supplied libraries by accessing parallel primitives through their own library and then implementing that library in terms of the vendor-supplied libraries. There are two approaches to developing a parallel library for portable applications:

1. Define a convenient virtual machine model, write the program in terms of that model, and then implement that model on the different machines.

2. Find a virtual machine model that is common to all machines, write the trivial interface between the common model and each machine, and then write the program in terms of the common model.

With the first approach, it may be difficult to implement the library on a given machine because the convenient model may be incompatible with the vendor's model, for example when the convenient model shares global variables and the vendor's model does not. Efficiency may also suffer when models are inconsistent, for example when an application with a model of uniform memory access times (UMA) is ported to a machine with with non-uniform access times (NUMA).

The second approach assumes that the 'intersection' of the vendor models is a useful, programmable, virtual machine model. The 'intersection' of models means that:

1. A capability that appears in the common model must appear directly, or have a trivial implementation, in each vendor model.

2. Programming practices for the common model will yield good performance on every vendor model.

The common virtual machine model approach to parallel libraries is much more likely to yield efficient programs because the performance aspects of the various machines are already embedded in the common virtual machine model. The cost is that the programmer's machine model is less convenient. By restricting programs to the functionality of the common model, we avoid programs that some machine cannot efficiently support.

This paper describes a parallel programming library for the C programming language developed based on the second approach. We developed this library in conjunction with a program to solve the subgraph isomorphism problem in parallel. This program executes efficiently on seven shared-memory multiprocessors, covering a range of processor organization and technology: the BBN Butterfly One and TC2000, the IBM 8CE, the Kendall Square KSR1, the Sequent Balance and Symmetry, and the Silicon Graphics Iris. While developed for a single program, we believe this library has general applicability.

The library supports an address space model that is consistent with all seven multiprocessors, a mechanism to create parallelism, and a set of synchronization primitives. Because the library presents mechanisms that are common to all vendors, the implementations are extremely transparent and implementations of the library average less than 100 lines of code each. The transparency of the implementations aids efficiency of execution.

The common virtual machine model defined by the library is usable, but is somewhat inconvenient. As one might expect, it is more difficult to program for the common model than it is to program for any *one* of the vendor models. However, we believe it is easier to program for the common model than to program for *all* of the vendor models.

Section 2 develops the virtual machine model underlying the library. Section 3 describes the interface to the model. Section 4 discusses programming with the library and its performance implications. Section 5 presents our conclusions. Appendix A briefly describes each of the seven multiprocessors used in this study. Finally, appendix B provides the complete implementation of the library for three of the seven multiprocessors.

2: Common Virtual Machine Model

Vendor-supplied parallel programming libraries generally export a *virtual* machine model that is very close to the *physical* machine. To construct a common parallel programming library, we must first define a common virtual machine model. If programs written using the common library are to be efficient, the corresponding virtual machine model must have an efficient mapping to each of the physical machines.

2.1: Memory Organization

The organization of memory within a multiprocessor has a great deal of impact on the programming methods needed to obtain efficient execution. Unfortunately, the multiprocessors in this study differ substantially in their memory organization. The Balance and Symmetry provide a single, physically-shared, global memory. Historically, these are called uniform memory-access multiprocessors (UMA). These machines are relatively insensitive to data locality and contention. The Butterfly and TC2000 provide one local memory per processor, but provide remote memory access from one processor to another. Memory is shared with respect to the instruction set architecture, but is not physically shared. Historically, these are called non-uniform memory-access multiprocessors (NUMA). These machines are highly sensitive to both data locality and contention. Programmers must be carefully distribute data and processing to provide good performance. The Iris and KSR1 provide a global memory, but with caches that are so large that they act like local memories. We call these machines cached memory-access multiprocessors (CMA). These machines are not as sensitive to physical location as NUMA multiprocessors, but are very sensitive to data contention or migration. Even though these machines appear to have a memory organization like that of UMA machines, only programming techniques like those used for NUMA machines will yield good performance. The primary difference between CMA and NUMA is that CMA machines will eventually migrate or replicated most data with little or no write sharing automaticly, whereas NUMA machines will not. The 8CE is somewhat unusual in that it provides both local memories (with remote access) and a global memory. It also needs NUMA programming techniques.

The common view of these disparate memory models is the more complex one — one global memory, one local memory per processor, and remote access to other processor's local memories. While a UMA view of memory, such as in Presto [4], is substantially easier to program, programs with the UMA view perform poorly on NUMA machines. On the other hand, programs with a NUMA view perform reasonably well on all machines. (See section 4.) Furthermore, the trend in shared-memory multiprocessors indicates that programs written with a UMA view will perform poorly on future machines [11]. So, the common view of memory is similar to the organization of future machines.

The common view of memory organization maps relatively efficiently to each of the hardware models. In the case of global-memory-only multiprocessors, the virtual local memories simply get mapped to a region of the physical global memory. In the case of multiple-local-memory-only multiprocessors, the virtual global memory gets mapped to some physical local memory. Because this latter mapping could introduce a point of contention, the model requires a certain style of programming if programs are to be efficient. We address this issue in section 4.

With the exception of the TC2000, each multiprocessor provides a sequentially-consistent view of memory. In the case of the TC2000, the memory view can be kept consistent by not caching data from shared memory. Because all multiprocessors support a sequentially-consistent view of memory, and because this view is easer to program than an inconsistent view, the common library provides a sequentially-consistent view of memory.

2.2: Process Organization

Each of the multiprocessors provides at least eight processors, and may provide many more. Rather than assume any specific number of processors, we assume there will be at least one processor but potentially many processors. The program indicates the number of processors it wishes during initialization.

The vendor libraries differ substantially in the assignment of processes to processors. Some libraries assign processes to a single processor and leave there for the lifetime of the process, while others while others will migrate processes to balance workload. However, all libraries make it possible to provide each process with exclusive access to a single processor, provided that there are no more processes than processors. The notion of exclusive access is important because not all vendor libraries provide mechanisms to multiprogram individual processors, and some rely on busy waiting for synchronization. In practice, the occasional interruption by system processes has little effect on program execution, but general multiprogramming in the presence of busy waiting can seriously degrade performance [6]. The common virtual machine model is that of a one process per processor with no multiprogramming.

Several multiprocessors provide memories local to a given processor. In order to ensure locality, programmers must be able to co-locate processing and data. To support this, our library indexes virtual processors and their corresponding local memories by a number in the range 0 to $n-1$. Furthermore, for those systems that provide local memories, the processes will not migrate among processors.

2.3: Address Space Organization

Each process has an address space that defines its view of memory. The portions of the address space that are shared have a significant impact on the programmer. Like most vendor libraries, we limit our scope to single-program-multiple-data computing. Each multiprocessor places all code in identical regions of the address space of each process, so the common library also does this. A consequence is that processes may pass function pointers across process boundaries, which makes possible the efficient implementation of user-task schedulers. The model replicates code to local memories if the multiprocessors have local memories.

The vedor libraries differ in whether or not C global variables are shared among all processes or are private to a single process. Those that make global variables private provide either a mechanism to share certain explicitly declared global variables, or a mechanism to replicate the value of global variables among the processors. Thus, the common view for C global variables is somewhat complicated — programmers cannot rely on global variables being private, programmers must explicitly declare variables that are shared, and programmers must explicitly replicate shared variables.

The vendor systems also differ in whether or not dynamically allocated (`malloc`ed) memory is shared. In the 8CE, KSR1 and Iris, `malloc`ed memory is shared, but each `malloc` yields a distinct region of memory. On the other hand, `malloc` on the Butterfly, TC2000, Balance and Symmetry yields process-private memory. The common view is that `malloc`ed memory is distinct among processes, but may not be assumed protected from other processes.

The systems that provide private `malloc` memory also provide a mechanism to dynamically allocate shared memory. The common library also provides such a mechanism. As discussed in section 2.1, programmers may allocate from specific local memories or from the global memory.

Because many multiprocessors manage memory in pages, programmers should not place both process-private and shared variables within the same region of memory. That is, programmers should allocate memory for process-private variable separately from that for shared variables.

2.4: Synchronization

Not all multiprocessors provide atomic reads and writes for C scalar variables, so the common virtual machine model does not assume such reads and writes are atomic. For higher-level synchronization, Some multiprocessors provide locks, and some provide test-and-set or fetch-and-add primitives. Because efficient synchronization is important to many programs, the common view must provide transparent access to hardware-based synchronization primitives. To this end, the common view must support the union of synchronization mechanisms. Our library provides locks, signals, and atomic counters. Our library does not provide barriers because our application did not need barriers. (Mellor-Crummey and Scott [12] present a good method for implementing synchronization efficiently on a wide variety of machines.)

3: Common Library Interface

This section describes the common library interface. We reserve discussions of programming and usage to section 4. The interface is built with a combination of C preprocessor macros, typedefs, global variables, and functions. All macros[1] appear in upper case and start with "`SHML_`", while all variables, typedefs, and functions appear in lower case and start with "`shml_`". Note that we use prototypes in our descriptions, but that the library does not use prototypes.

3.1: Processors

The preprocessor symbol `SHML_MAX_PROCS` is an integer constant specifying the maximum number of processors that may appear on a given machine. This symbol may be used to size array variables.

The library is initialized by calling `shml_initialize`, with the desired number of processors as a parameter. Programmers must call this function before any other library function, and may not call the function more than once.

```
void shml_initialize( int num_procrs ) ;
```

The processes are spawned with a call to `shml_spawn`. This function accepts two parameters, the number of processors desired[2] and a function to execute on each processor. The parameter function receives an argument specifying the virtual processor on which it executes.

```
void shml_spawn( int num_procrs,
        void (*body_func)( int procr_number ) ) ;
```

3.2: Memory and Address Space

The common library provides a `SHML_SHARED` storage class for all C global variables that are to be shared among processes. These variables must be explicitly replicated with the `SHML_REPLICATE` operation, which accepts a pointer to the global variable and the size of the variable. The `SHML_REPLICATE` operation may only be called after `shml_initialize` and before `shml_spawn`, which means that programmers cannot modify shared C global variables once parallel execution has begun. Typical uses of shared global variables are to contain read-only data, and to store read-only pointers to dynamically allocated read-write shared memory.

```
void SHML_REPLICATE( any *var_ptr, int size ) ;
```

Programmers may dynamically allocate shared memory on a specific local memory with a call to `SHML_PALLOC`, which accepts a virtual processor number on which to allocate the memory and a size to allocate in bytes. Programmers may dynamically allocate global shared memory with a call to `SHML_GALLOC`, which accepts a size to allocate in bytes. Programmers may continue to use `malloc` to allocate private memory on local processors, though programmers should not expect such memory to be protected from other processes.

```
void *SHML_PALLOC( int processor_number,
                   unsigned size ) ;
void *SHML_GALLOC( unsigned size ) ;
```

The `SHML_COPY` operation copies memory from a given source address to a given target address for a given number of bytes. This operation may be significantly faster than normal C memory copies on some machines because not all vendors modify the standard `memcpy` to take advantage of special hardware. The `SHML_ZERO` operation zeros memory at a given address for a given number of bytes.

```
void SHML_COPY( any *source_address,
        any *target_address, int size ) ;
void SHML_ZERO( any *address, int size ) ;
```

3.3: Synchronization

The library provides three synchronization primitives, locks, signals, and atomic counters.

The type `shml_lock` represents atomic lock variables. `SHML_LOCK_INIT` initializes a lock. `SHML_LOCK_WAIT` busy waits to acquire a lock. `SHML_LOCK_VISIT` attempts to acquire a lock and returns true if successful and false otherwise. `SHML_LOCK_RELEASE` releases a lock.

```
void SHML_LOCK_INIT( shml_lock *lock_ptr ) ;
void SHML_LOCK_WAIT( shml_lock *lock_ptr ) ;
int SHML_LOCK_VISIT( shml_lock *lock_ptr ) ;
void SHML_LOCK_RELEASE( shml_lock *lock_ptr ) ;
```

[1] Programmers should not pass arguments with side effects to macros because the implementation may evaluate the argument more than once.

[2] The specification of the number of processors in `shml_spawn` is potentially redundant with that for `shml_initialize`, but it simplifies the implementation.

The type shml_signal represents atomic boolean variables. SHML_SIGNAL_CLEAR clears a signal. SHML_SIGNAL_SET sets a signal. SHML_SIGNAL_TEST returns true if a signal is set. SHML_SIGNAL_WAIT busy waits for a signal to be set. This abstraction has very efficient implementations on all machines, so should be used where possible in preference to locks.

```
void SHML_SIGNAL_CLEAR( shml_signal *signal_ptr ) ;
void SHML_SIGNAL_SET( shml_signal *signal_ptr ) ;
int SHML_SIGNAL_TEST( shml_signal *signal_ptr ) ;
void SHML_SIGNAL_WAIT( shml_signal *signal_ptr ) ;
```

The type shml_counter represents atomic unsigned counter variables. SHML_ADD atomicly adds the increment to the counter. SHML_FETCH_AND_ADD atomicly adds the increment to the counter and returns the value of the counter before the increment. In addition, programmers may read and write counters atomicly, as they would other unsigned variables.

```
void SHML_ADD(
    shml_counter *counter_ptr, int incr ) ;
unsigned short SHML_FETCH_AND_ADD(
    shml_counter *counter_ptr, int incr ) ;
```

3.4: Timing

Many parallel programs keep track of their execution time. The shml_time_now function returns a double representing the current time in seconds, relative to some unspecified base. On multiprocessors, the time is wall-clock time. On uniprocessors, the time may be either wall-clock time or CPU time. Because the value provided by shml_time_now has no particular base, programmers must compute elapsed time by subtracting values returned by two successive calls to shml_time_now.

```
double shml_time_now( ) ;
```

4: Programming and Performance

Our library's virtual machine model requires programmers to adopt a few techniques if they are to obtain reasonable performance over a wide range of architectures. We discuss these techniques and then describe the program organization they encourage. We also provide an example building a convenient parallel for loop out of raw processes.

4.1: Distribution and Replication

The common virtual machine model consists of both local and global memories. On some multiprocessors, access to non-local memory is significantly more expensive than access to local memory. In addition, the virtual global memory may be mapped to a local memory on some multiprocessors, which could be a source of excessive contention. Given the higher cost of non-local memory access and the potential contention of global memory, programmers should use local memory wherever possible. Not only will this approach yield better performance on NUMA multiprocessors, but may also yield better performance on CMA systems.

The first technique for using local memory is to distribute data so that processors have the portion of the data that they process within their local memory. Distributing data often requires an additional level of pointers in the implementation of data structures, which could reduce performance by requiring processors to chase an additional pointer when accessing the data. Figure 1 shows the effect of such data distribution on two multiprocessors: the TC2000 which is a local-memory-only NUMA multiprocessor, and the Balance which is a global-memory-only UMA multiprocessor. The figure shows that failing to distribute

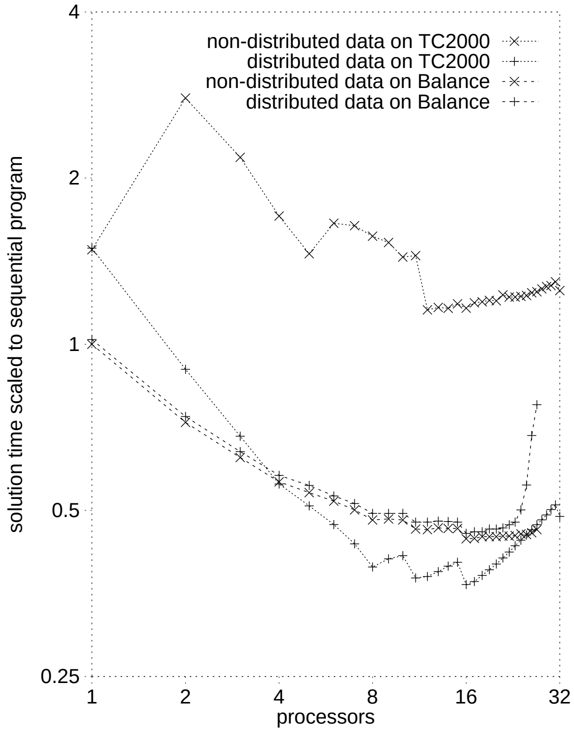

Figure 1: Effect of Data Distribution on Performance

introduced a factor of four performance penalty on the NUMA multiprocessor. On the other hand, distributing data introduced only a three percent performance penalty on the UMA multiprocessor up until the point where its shared memory bus became saturated. (The relatively poor speedup of this graph is due to the nature of the subgraph isomorphism problem, and not a consequence of the library.)

The second technique for using local memory is to replicate read-only data to the local memories. On NUMA machines, this technique will improve performance dramatically. On other machines, this technique should introduce little performance penalty, provided sufficient memory is available. If programmers use the library's `SHML_SHARED` and `SHML_REPLICATE` mechanisms, then the data will not actually be replicated on systems without physical local memory, but will reside in physical global memory, thus conserving memory space.

The third technique is to replicate read-write variables to local memories. This is possible for those variables for which there is a single writer and many readers. Each write to the original variable is replaced by a write to a copy on each processor. This technique is not applicable to lock and atomic counter variables, but often applies to signal variables.

Because signal variables can be replicated, programmers should use them in preference to locks or counters. Indeed, it may be better to rewrite synchronization using locks or counters into synchronization using signals. For example, many parallel programs maintain a counter of the number of processes that have completed initialization and are ready to begin execution. Processes then start when the counter reaches the expected value. Under our library, this counter would be placed in virtual global memory and may introduce a source of contention. Another approach, more appropriate to our model, is for each process to maintain its own signal indicating when it has completed initialization. Because each process has its own signal, there is only one writer, and signals may be replicated. To synchronize program initialization, each process writes all copies of its signal and then waits until all processes have set its copy of their signals.[3]

Replicating signal variables has three effects on performance: it increases the number of memory writes; it reduces the contention because processes spin on their own copy of the signal rather than on a shared copy; and it reduces the inter-processor communication because processes only communicate with local memory rather than with remote memory.

In summary, programmers should distribute or replicate data and synchronization where possible, and limit the use of virtual global memory to variables that cannot be replicated or distributed.

[3] A barrier would serve here.

4.2: Program Organization

Our library encourages a program organization in which substantial work is devoted during initialization to setting up the shared memory regions.

The main program generally has the following sequence of events.

> determine number of processors desired
>   and call `shml_initialize`
> initialize unshared global variables
> allocate shared global memory with
>   `SHML_GALLOC`
> initialize shared global variables
>   and call `SHML_REPLICATE` on each
> spawn processes with `shml_spawn`

On starting up, each child process generally has the following sequence of events.

> allocate local private memory with
>   `malloc`
> allocate local shared memory with
>   `SHML_PALLOC`
> copy pointers to local memory
>   into global memory
> indicate process ready with `SHML_SIGNAL_SET`
>   on a signal in global memory
> wait for all other processes to signal ready
> copy pointers to other processors'
>   local memories into local memory
> begin work

Alternatively, the main program may allocate the local memory for each process and set up the corresponding pointers before spawning.

This organization is necessarily complex because it must handle a NUMA virtual machine model.

4.3: Programming Example — Forall

As an example of programming with our library, we outline how to implement a parallel `for` loop. This loop is implemented as a function which takes the loop bounds and a loop-body function as parameters. The implementation uses a master/slave process organization. The master executes the program's main and sequential code, and occasionally executes the `forall` function that requests parallel execution of a function. Each slave process executes a `forall_server` function that waits for work, executes it, and waits again. These functions synchronize through an 'idle' signal for each process. The master clears the idle signal when it has provided work; the slave waits for the signal to clear, does its work, and sets the signal when finished; and then the master waits until all processes have set their idle signal. The cycle may then repeat.

The outline of `forall` is:

```
divide loop iterations based on
      the number of processors
for each other processor
      write loop sub-bounds, function pointer,
            and parameters to processor
      clear other processor's idle
            with SHML_SIGNAL_CLEAR
do local iterations
for each other processor
      wait for idle with SHML_SIGNAL_WAIT
```

The outline of `forall_server` is

```
while program still running
      if still idle
            delay and repeat
      else
            read loop sub-bounds, function pointer,
                  and parameters
            execute loop iterations
            set idle with SHML_SIGNAL_SET
```

This example illustrates the concept of using distributed signals rather than single-point synchronization variables, and demonstrates that programmers can build more convenient programming abstractions than raw virtual processors in a machine-independent manner.

5: Conclusions

The library has efficient implementations on all seven multiprocessors, and programs can be written that are reasonably efficient on each multiprocessor, provided programmers adopt the virtual machine model and programming techniques outlined in this paper. However, optimal execution on each multiprocessor may require changes to the parallelism a program exploits, which is beyond the scope of a library. For a discussion of these issues, see [7].

Our library is portable and reasonably efficient, but not particularly convenient. The need to explicitly manage replication and distribution places a significant burden on the programmer. This burden is not unique to our library, but is common to most NUMA multiprocessors. Given that the NUMA architecture is likely to persist in future multiprocessors, we anticipate that programmers relying on libraries, rather than compilers, will continue to invest in replication and distribution, and our library imposes no special burden in that regard.

A more specific problem with our library is that programmers cannot rely on process-private global variables, and hence must pass pointers to the process-local memory regions as parameters to all functions that may access process-local variables by name. This problem is not inherent to the different machines, but represents different choices about the meaning of global variables on the part of the different vendors. Had all vendors chosen to make global variables private to a process, the common virtual machine would have been much easier to program. So, while our library can serve as a short-term solution for some programmers, it certainly will not substitute for current parallel programming standards efforts [13].

Our experience developing our application and this library was difficult for the first three multiprocessors (the Butterfly, the Balance, and the Iris) because each new port highlighted additional incorrect assumptions embedded in the original application. These assumptions generally resulted in addressing exceptions due to stray pointers, and substantial program rewrites. Subsequent ports to the final four multiprocessors were relatively easy, with locating documentation as the most difficult task. The extremely short 100-line implementations for this library make it easy for programmers to include the library directly within their programs and to implement the library for additional multiprocessors.

While parallel programming standards are still not widely accepted, programmers are not helpless in the face of diverse programming environments for shared-memory multiprocessors. We have described a shared-memory parallel programming library, its virtual machine model, and a set of techniques for programming with the library that enable programmers to write programs that execute effectively on seven different shared-memory multiprocessors.

Acknowledgements

We thank Mark Crovella for the original KSR1 implementation of the library. We also thank Argonne National Laboratories for the use of their TC2000, International Business Machines for providing the 8CE, Donna Bergmark and the Cornell Theory Center for their assistance and the use of their KSR1, and Sequent Computer Systems for providing the Balance and Symmetry.

References

[1] American National Standards Institute. *American National Standard Programming Language: FORTRAN, X3.9-1978*, 1978.

[2] BBN Laboratories, Cambridge, Massachusetts. *Chrysalis Programmer's Manual*, June 1985.

[3] BBN Laboratories, Cambridge, Massachusetts. *The Uniform System Approach To Programming the Butterfly Parallel Processor*, October 1985.

[4] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[5] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, Litchfield Park, Arizona, December 1989. also appeared in Operating Systems Review 23(5), December 1989.

[6] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on multiprocessors. Technical Report 385, Computer Science Department, University of Rochester, February 1991.

[7] L. A. Crowl, M. E. Crovella, T. J. LeBlanc, and M. L. Scott. Beyond data parallelism: The advantages of multiple parallelizations in combinatorial search. Technical Report 92-80-06, Department of Computer Science, Oregon State University, December 1992.

[8] High Performance Fortran Forum, Rice University. *DRAFT High Performance Fortran Language Specification*, January 1993.

[9] IEEE. *POSIX Threads, Standard P1003.4a*, 1993.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, 1978.

[11] E. P. Markatos and T. J. LeBlanc. Shared-memory multiprocessor trends and the implications for parallel program performance. Technical Report 420, Computer Science Department, University of Rochester, May 1992.

[12] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1992.

[13] C. M. Pancake. What should we expect from parallel language standards? *International Journal of Supercomputer Applications*, 6(1):112–117, Spring 1992.

[14] C. M. Pancake. Multithreaded languages for scientific and technical computing. *Proceedings of the IEEE*, 81(2):288–304, February 1993.

[15] Parallel Computing Forum. Pcf parallel fortran extensions. *ACM Fortran Forum*, 10(3), September 1991.

[16] W. G. Rudd (document editor). *ANSI X3H5 93-SD3 Revision D of the Draft C binding to the Parallel Processing Model for High-Level Languages (ANSI X3H5 93-SD2 Revision I)*. ANSI, 1993.

[17] Sequent Computer Systems, Inc. *Sequent Guide to Parallel Programming*, 1987.

[18] Silicon Graphics, Inc. *Parallel Programming on Silicon Graphics Computer Systems*, June 1991.

A: Multiprocessor Machine Descriptions

Our library has implementations for seven shared-memory multiprocessor systems covering a range of operating system, processor, and interconnect technologies.

The BBN Butterfly One uses the Motorola 68000 processor, each with its own local memory. Each processor may access another's memory through a switching network similar to the Omega network. The access times to local memory and another processor's memory are in the ratio 1:5. There is no memory cache. The Butterfly runs the Chrysalis operating system [2]. Rather than use Chrysalis directly, we use the Uniform System library [3].

The BBN TC2000 uses the Motorola 88100 processor, each with its own local memory. Each processor may access another's memory through a switching network similar to the Omega network. The 8 KB processor data caches are used for private data, but are not used for shared data because they are not kept consistent. The TC2000 runs a variant of the Mach operating system. Again, we use the Uniform System library. This TC2000 version of the library differs from the Butterfly version in several details.

The IBM 8CE uses the ROMPC processor (as found in the IBM RT). In addition to a global shared memory, each processor has its own local memory. Access to non-local memory is through a shared bus, and the access times to local, global and another processor's local memory are in the ratio 1:2:5. There is no memory cache. The 8CE has an experimental variant of the Mach operating system [5] that hides the presence of local memories by dynamically placing shared pages in the global shared memory and placing unshared pages in local memories. We use the Mach Cthreads library.

The Kendal Square Research KSR1 uses a custom 64-bit processor. All memory in the system is managed as a set of caches, with each processor containing a 32 MB cache and a 512 KB subcache. Access times to the subcache, the local cache, and a remote cache are in the ratio 1:9:88. Cache block movement is through a high-speed ring network.[4] The KSR1 uses the OSF/1 operating system, a variant of Mach. We use the POSIX Pthreads library [9].

The Sequent Balance uses the National Semiconductor 32032 processor. There is a single global shared memory, accessed through a shared bus. Each processor has an 8KB write-through cache. The Balance runs a variant of BSD Unix. We use the Sequent Parallel Programming Library [17].

The Sequent Symmetry uses the Intel 80386 processor. There is a single global shared memory, accessed through a shared bus. Each processor has an 64KB write-back cache. The Symmetry runs a variant of System V Unix. Again, we use the Sequent Parallel Programming Library.

The SGI Iris uses the MIPS R3000 processor. There is a single global memory accessed through a shared bus. Each processor has 64 KB write-through primary cache and a 1 MB write-invalidate secondary cache, which is large enough that the Iris is most effectively programmed as though it had local memories. The Iris runs a variant of System V Unix. We use the SGI Parallel Programming Library [18].

B: Common Library Implementations

In addition to the implementations for the seven multiprocessors presented in appendix A, we have implementations for the the BBN Butterfly One as a uniprocessor, and a generic uniprocessor Unix. We provide the implementations for the Butterfly and the 8CE. These two implementations are representative of the techniques used in the other implementations.

B.1: BBN Butterfly One and the Uniform System

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

#define SHML_SHARED
#define SHML_REPLICATE( variable_ptr, size ) \
    ShareBlk( (int *)(variable_ptr), (size) )

extern char *AllocateOnUsProc( /* int node,
                                  unsigned size */ ) ;
#define SHML_PALLOC( procr_num, size ) \
     (AllocateOnUsProc( (procr_num), (size) ))
#define SHML_GALLOC( size ) \
     (AllocateOnUsProc( 0, (size) ))

extern char *memcpy( /* char *target_adr,
                    char *source_adr, int size */ ) ;
#define SHML_COPY( source_adr, target_adr, size ) \
     ((void)block_copy( (char *)(source_adr), \
               (char *)(target_adr), (size) ))

#define SHML_ZERO( adr, size ) \
     bzero( (char *)(adr), (size) )

void shml_delay( count )
    int count ;
  { while ( count-- > 0 ) ;
  }

typedef volatile short shml_signal ;
#define SHML_SIGNAL_CLEAR( signal_ptr ) \
    (*(signal_ptr) = 0)
#define SHML_SIGNAL_SET( signal_ptr ) \
    (*(signal_ptr) = 1)
#define SHML_SIGNAL_TEST( signal_ptr ) \
    (*(signal_ptr))
#define SHML_SIGNAL_WAIT( signal_ptr ) \
    (shml_signal_wait( signal_ptr ))

shml_signal_wait( signal_ptr )
    shml_signal *signal_ptr ;
  { while ( !SHML_SIGNAL_TEST( signal_ptr ) )
        shml_delay( 10 ) ;
  }

typedef volatile short shml_lock ;
#define SHML_LOCK_INIT( lock_ptr ) \
    (*(lock_ptr) = 0)
#define SHML_LOCK_WAIT( lock_ptr ) \
    (shml_lock_wait( lock_ptr ))
#define SHML_LOCK_VISIT( lock_ptr ) \
    (Atomic_ior( (lock_ptr), 1 ))
#define SHML_LOCK_RELEASE( lock_ptr ) \
    (*(lock_ptr) = 0)

shml_lock_wait( lock_ptr )
    shml_signal *lock_ptr ;
  { while ( !SHML_SIGNAL_TEST( lock_ptr ) )
        shml_delay( 10 ) ;
  }

typedef volatile unsigned short shml_counter ;
#define SHML_ADD( cntr_ptr, incr ) \
    (Atomic_add( cntr_ptr, incr ))
#define SHML_FETCH_AND_ADD( cntr_ptr, incr ) \
    (Atomic_add( cntr_ptr, incr ))

double shml_time_now( )
  { return (unsigned)GetRtc( ) / 16000.0 ;
  }

#define SHML_MAX_PROCS 128
```

[4] The KSR1 has a multi-level ring architecture, but all our work was done on a single ring.

```
void shml_initialize( num_procrs )
    int num_procrs ;
  { static int spec[ ] = { configProcs, 1,
                configSuppressInitMsgs, 1 } ;
    spec[ 1 ] = num_procrs ;
    ConfigureUs( spec, 2 ) ;
    InitializeUsForBenchMark( ) ;
  }

/*ARGSUSED*/ void shml_helper( spawn_body, dummy )
    void (*spawn_body)( ) ;  int dummy ;
  { spawn_body( PhysProcToUsProc( pnn ) ) ;
  }

void shml_spawn( num_procrs, spawn_body )
    int num_procrs ;  void (*spawn_body)( ) ;
  { GenOnIndex( (int (*)())0, shml_helper,
                spawn_body, num_procrs ) ;
  }
```

B.2: IBM 8CE and Mach Cthreads

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/times.h>
#include <mach.h>
#include <cthreads.h>

#define SHML_SHARED
#define SHML_REPLICATE( variable_ptr, size )

#define SHML_PALLOC( procr_num, size ) \
    (shml_allocate( size ))
#define SHML_GALLOC( size ) (shml_allocate( size ))

char *shml_allocate( size )
    unsigned size ;
  { char *allocated ;
    vm_allocate( task_self( ), &allocated,
                 size, TRUE ) ;
    return allocated ;
  }

extern char *memcpy( /* char *target_adr,
                 char *source_adr, int size */ ) ;
#define SHML_COPY( source_adr, target_adr, size ) \
    ((void)memcpy( (char *)(target_adr), \
                   (char *)(source_adr), (size) ))

#define SHML_ZERO( adr, size ) \
    bzero( (char *)(adr), (size) )

void shml_delay( count )
    int count ;
  { while ( count-- > 0 ) ;
  }

typedef volatile short shml_signal ;
#define SHML_SIGNAL_CLEAR( signal_ptr ) \
    (*(signal_ptr) = FALSE)
#define SHML_SIGNAL_SET( signal_ptr ) \
    (*(signal_ptr) = TRUE)
#define SHML_SIGNAL_TEST( signal_ptr ) \
    (*(signal_ptr))
#define SHML_SIGNAL_WAIT( signal_ptr ) \
    (shml_signal_wait( signal_ptr ))

shml_signal_wait( signal_ptr )
    shml_signal *signal_ptr ;
  { while ( !SHML_SIGNAL_TEST( signal_ptr ) )
        shml_delay( 10 ) ;
  }

typedef volatile struct mutex shml_lock ;
#define SHML_LOCK_INIT( lock_ptr ) \
    (mutex_init( lock_ptr ))
#define SHML_LOCK_WAIT( lock_ptr ) \
    (mutex_lock( lock_ptr ))
#define SHML_LOCK_VISIT( lock_ptr ) \
    (mutex_try( lock_ptr ))
#define SHML_LOCK_RELEASE( lock_ptr ) \
    (mutex_unlock( lock_ptr ))

shml_lock shml_counter_lock ;
typedef volatile unsigned short shml_counter ;
#define SHML_ADD( cntr_ptr, increment ) \
    (mutex_lock( &shml_counter_lock ) \
     , *(cntr_ptr) += incr \
     , mutex_unlock( &shml_counter_lock ))
#define SHML_FETCH_AND_ADD( cntr_ptr, incr ) \
    (shml_fetch_and_add( cntr_ptr, incr ))

unsigned short shml_fetch_and_add( cntr_ptr, incr )
    shml_counter *cntr_ptr ;  int incr ;
  { unsigned short old_value ;
    mutex_lock( &shml_counter_lock ) ;
    old_value = (*cntr_ptr += incr) - incr ;
    mutex_unlock( &shml_counter_lock ) ;
    return old_value ;
  }

double shml_time_now( )
  { struct timeval tv ;
    (void)gettimeofday( &tv, (struct timezone *)0 ) ;
    return tv.tv_sec + tv.tv_usec / 1000000.0 ;
  }

#define SHML_MAX_PROCS 8

/*ARGSUSED*/ void shml_initialize( num_procrs )
    int num_procrs ;
  { mutex_init( &shml_counter_lock ) ;
  }

void shml_spawn( num_procrs, spawn_body )
    int num_procrs ;  void (*spawn_body)() ;
  { int procr ;
    for ( procr = 1 ; procr < num_procrs ; procr++ )
        cthread_detach( cthread_spawn( spawn_body,
                                       procr ) ) ;
    spawn_body( 0 ) ;
  }
```