MOOSE: A Multi-Tasking Operating System for Hypercubes

John Salmon*, Sean Callahan*, Jon Flower*†, Adam Kolawa*†

* California Institute of Technology, Pasadena CA 91126
† Current Address: ParaSoft Corp., 27415 Trabuco Circle, Mission Viejo, CA 92692

1. Introduction

The MOOSE project was begun at Caltech in Summer 1986. Its goal is to produce a powerful, flexible multi-tasking operating system suitable for research into load-balancing and decomposition of irregular and dynamic problems. Sec. 2 of this report describes the features of the MOOSE system in some detail. In Sec. 3 we discuss where we expect developments in MOOSE to take place, and in Sec. 4 we review some of the lessons learned from the MOOSE project.

MOOSE offers some distinct advantages over other operating systems available on Caltech's hypercubes (CrOS III [Fox 88], Time Warp [Jefferson 85,88], and the commercial systems from Intel and NCUBE). It is especially attractive as a vehicle for load balancing, because it allows for run-time multi-tasking. It is possible to write multi-tasking programs for the Intel iPSC but the structure of tasks must be specified at compile time, and cannot be changed during the course of the program. Unlike Time Warp, MOOSE does not promote a specific programming model. The MOOSE programmer is left to explore different programming models that might suit his problem. One could even implement the Time Warp model on top of MOOSE. MOOSE is more akin to the Time Warp 'Machine Interface' than the entire Time Warp system.

MOOSE is operational on our Mark II hypercubes, our NCUBE systems and our Intel iPSC. The features are essentially the same on each, except that the real-time clock facilities are not implemented on the NCUBE.

2. System Overview.

MOOSE consists of several distinct but overlapping sub-systems. The *tasking* sub-system deals with the creation, destruction and scheduling of tasks. The communication sub-system allows tasks to communicate through *pipes*. The synchronization sub-system allows the programmer to synchronize tasks using *semaphores*. An I/O subsystem based on the CUBIX model [Salmon 87] allows programs to transparently access the host's operating system. In the following sections, we treat each of these sub-systems in turn.

2.1 Tasks

Currently, MOOSE loads the entire executable module, consisting of user code and data, as well as operating system code and data, into each processor of the hypercube. This strategy is dictated by the boot-loaders already present on the hypercubes to which we have ported MOOSE, which we elected not to replace or override. Thus, all processors have access to all code, as well as a single private copy of all data declared with C storage class *extern*.

As far as the programmer is concerned, execution begins with the function *main*, in processor 0 only. In order to start the other processors doing useful work the *main* task must create tasks on those processors, using the task system call which will be discussed shortly.

A task is a thread of control, with its own instruction pointer and its own stack, or *automatic* data. Within a task instructions are executed in the usual order, but there may be long delays between the execution of one instruction and the next, since the scheduler may decide to execute another task for a while. This type of behavior is familiar to (although conveniently overlooked by) programmers used to operating systems like UNIX and VMS. When a UNIX process is begun, it appears to the programmer that instructions are executed in a well-defined sequence, without interruption. In fact, many processes may be sharing the CPU, and long delays may interrupt any given process when those other processes gain control of the CPU. Although the underlying CPU is executing many separate processes, and maintaining many independent threads of control, the programming model apparent to the user is that of a unique thread of control winding its way through his program.

MOOSE's primary goal is to support multi-tasking on a parallel computer. In order to achieve that goal, a mechanism must be created which allows for the creation of new tasks. The system call *task* serves this purpose.

```
#include <task.h>
#include <pipes.h>

void task(int (*funcptr)(), int processor, int stacksize,
      int priority, PDES inpipe, PDES outpipe);
```

Example 1. The *task* system call.

The first argument, *funcptr*, is a pointer to a function, and causes the new task to begin executing at that address. Since all the code is present and identical on all processors, such pointers are unambiguous, and may be dereferenced independently of processor. The next argument, *processor*, allows the programmer to specify on which processor the task should run. The *stacksize* argument instructs the system to allocate a fixed stack of the specified size, to hold the local data used by the new task. It is an error for the stack to grow beyond this point. The *priority* is used by the scheduler to determine the order of task execution. The last two arguments are the task's input and output pipes, respectively. These provide the task's primary connection with its parent and siblings. Pipes will be discussed in Section 2.4.

Unlike processes in UNIX, MOOSE tasks can communicate through shared memory, *if they are executing on the same processor*. This feature is a source of great convenience, as well as considerable headache. As mentioned above, all tasks on the same processor share code and external data. Memory declared with storage class *extern*, and that obtained from the system call *malloc* is sharable by all tasks running on the same processor. Although the memory allocated by *malloc* is sharable, only the task that called *malloc* will "know" where that memory is, unless the value returned by *malloc* is placed in a shared external variable. Since external identifiers refer to the same data in different contexts, one should take great care in using them as loop counters, etc. A set of semaphore primitives is provided (see Section 2.3), to assist in synchronization of access to such shared data.

The currently running task is always available under the pointer *thistask*, which is declared in the include file *task.h*. It is useful, because the current task's input and output pipes are contained in the *inpipe* and *outpipe* fields of *thistask*, as in Example 2.

```
#include <task.h>

...
pread(thistask->inpipe, ...);
pwrite(thistask->outpipe, ...);
...
```

Example 2. Use of the *thistask* structure.

2.2. Scheduling and the Real-Time Clock

Each task has a programmer selected *priority*, which is assigned when the task is created, and which is used by the scheduler to decide which task to run. The scheduler attempts to ensure that the highest priority *runnable* task is always running. If there is more than one runnable task with the highest priority, then those tasks execute in a round-robin order, each getting the CPU for one *tick*, or about 100msec. The MOOSE scheduler is pre-emptive on the Mark II and Intel systems. That is, if a high priority task becomes runnable through some asynchronous event like a timer interrupt or receipt of a message, it immediately gains control of the CPU. On the NCUBE, we have not implemented pre-emption, and the running task will retain control of the CPU until it makes a system call.

One way for a task to enter a state other than runnable, is to execute the *sleep* system call.

```
void sleep(int nticks);
```

Example 3. The *sleep* system call.

The argument, *nticks*, specifies a number of ticks of the real-time clock that should elapse before the task becomes runnable again.

2.4 Semaphores

A semaphore is a data structure, which may be accessed only through the primitives displayed in Example 4.

The header file *<sem.h>* defines the type SDES, or semaphore descriptor. A semaphore descriptor identifies a unique semaphore which resides on a particular processor somewhere in the system. The function *screate* returns a new semaphore with the specified initial value. The semaphore resides on the named *processor* but the semaphore may be used by tasks on other processors. A semaphore descriptor, like the pipe descriptors described in Section 2.4,

```
#include <sem.h>

extern SDES NULLSEM;

SDES screate(int processor, int initial_val);

sdelete(SDES sd);

int ISNULL(SDES sd);

swait(SDES sd);

ssignal(SDES sd);

ssignaln(SDES sd, int ntimes);

sreset(SDES sd, int new_value);

int scount(SDES sd);
```

Example 4. Semaphore primitives.

contains information which allows the semaphore primitive routines to uniquely locate the semaphore's internal state variables, regardless of processor. A semaphore descriptor is not a pointer, and hence its value is valid on any processor. A semaphore descriptor may be placed in a pipe, and then used by a task on another processor. The function *sdelete* removes the named semaphore, signaling all tasks that may have been waiting for it. It is an error to refer to a semaphore that has been deleted. The special semaphore descriptor, NULLSEM, never refers to a valid descriptor, and the macro ISNULL, may be used to compare any descriptor with

NULLSEM. If a non-fatal error is encountered in *screate*, NULLSEM is returned.

The complementary functions *swait* and *ssignal* constitute the primary interface to semaphores. Each semaphore contains a hidden integer variable called *s*, for the purpose of discussion, and a list of *waiting* tasks. When *swait* is called on a semaphore, the count, *s*, is decremented by one and if it becomes negative the task that executed *swait* enters the *waiting* state, and is placed on the list of tasks associated with the semaphore. The complimentary function *ssignal* increments the hidden variable *s* and if *s* remains negative or zero, the highest priority task in the list waiting waiting tasks is made *runnable*. Thus, several tasks can be waiting for the same semaphore, and they are made runnable one at a time by execution of *ssignal*.

The other primitives are simple variations on the theme of *swait*, *ssignal*, *screate* and *sdelete*. The function *signaln* atomically calls *ssignal* a number of times. The function *sreset* awakens all tasks sleeping on the named semaphore and then resets the semaphore's counter to *new_value*, and *scount* returns the value of the named semaphore's counter. This value is "current" at the time that *scount* returns, but it may have changed before the calling program has a chance to examine the value. It should be used for advisory purposes only. Basing a synchronization algorithm on the value returned by *scount* is almost certainly an error.

The power of semaphores is demonstrated by the code fragment in Example 5. which enforces mutual-exclusion between execution of critical sections of code, i.e. even if many tasks are simultaneously executing the subroutine containing this code fragment, only one task can be executing instructions inside the critical section at any one time. Thus, it is safe to touch shared data structures within the critical section, without fear of interference.

```
#include <sem.h>
extern SDES sd;
sd = screate(proc, 1);      /* executed only once */
...
swait(sd);
/* CRITICAL SECTION */
ssignal(sd);
...
```

Semaphores were introduced in [Dijkstra 1968]. An illuminating discussion appears in [Ben-Ari 1982]. The semaphores used in MOOSE are similar to those in the Xinu operating system [Comer 1984].

2.4 Communication and pipes.

MOOSE tasks executing on the same processor may communicate through shared memory. MOOSE also provides a more general communication mechanism via *pipes*, so that any two tasks can communicate regardless of whether they share the same processor or not. This is desirable because it allows for a uniform coding style and often avoids complicated synchronization algorithms involving semaphores.

A pipe is a depository for data in fixed length records. The records in the depository are read and written in FIFO order. The depository resides on a particular processor, even though tasks on any processor may refer to the pipe. The system calls that refer to pipes are shown in Example 6.

```
#include <pipes.h>

PDES pipopen(int processor,
     int record_length, int nrecords);

int pwrite(PDES pd, void *dataptr, int nbytes);

int pread(PDES pd, void *dataptr, int nbytes);

void pclose(PDES pd);
```

Example 6. Pipe primitives.

A pipe is created by the system call, *pipopen*, with a depository on the named *processor*. All subsequent reference to the pipe is through the pipe-descriptor (PDES) returned by *pipopen*. It is important to realize that a PDES is a structure that contains enough information to uniquely identify any pipe in the system. A PDES is not a pointer. In contrast to a pointer, its value is independent of the processor on which it is used. The records that will be exchanged through the pipe must be of size *record_length* or smaller, and there may be no more than *nrecords* unread records sitting in the pipe at any one time.

The system call *pwrite* copies *nbytes* of data beginning at the pointer, *ptr*, to the first free record in the pipe identified by *pd*. A shortcoming of the current implementation is that *pwrite* will fail if the pipe is full, or if the value of *nbytes* exceeds the record length of the named pipe.

The system call *pread* copies *nbytes* from the first record in the pipe into the space under the pointer, *ptr*, and deletes the record. It is up to the user to guarantee that this record contains *nbytes* of valid data. If the record was placed in the pipe by a call to *pwrite* with *nbytes* less than that in the call to *pread*, then only the initial bytes are valid. If there are no records in the pipe at the time *pread* is called, then the calling task enters the *waiting* state. It is awakened when a record is written into the pipe by some other task. Internally, the pipe system uses a semaphore protocol to control waiting. In the interest of speed, when a *pread* is pending on a pipe, the corresponding call to *pwrite* deposits the data directly into its final destination, avoiding the unnecessary copy into and out from the pipe's central depository.

The system call *pclose* deletes a pipe from the system. Any attempt to access the pipe after a call to *pclose* is an error.

2.5 I/O and Asynchronous CUBIX.

MOOSE tasks communicate with the outside world with an asynchronous version of CUBIX. CUBIX is a runtime library of routines for the processors, and a universal program that runs on the host which makes most of the host operating system transparently available to the MOOSE program [Salmon 1986]. In contrast to loosely synchronous CUBIX, asynchronous CUBIX imposes absolutely no requirements of synchronization or argument agreement. Each processor's

interaction with the host is logically separate from all others. If used naively, this feature can lead to considerable overhead, e.g. when all processors are reading identical data from a file. If the system call *read* or the standard I/O call *fread* is made by each processor, then the data will be read by the host, and transmitted from the host to processor 0 once for each processor in the cube. Of course, nothing prevents the programmer from noting this fact and calling *read* only once in processor 0, and then distributing the data to the rest of the cube using a fast tree-based or ring-based algorithm.

The complete independence of the individual processors, coupled with the fact that there is really only one host process to multiplex all their access to the UNIX system leads to some difficulties. A single UNIX process can have, at most, 20 open files. Thus, processors may not open separate files for their input or output. All together, the processors may have files opened with at most 20 distinct names. The code fragment in Example 7. simply will not work on a system with more than 20 processors. (Note that each processor is trying to open a different file.)

```
char s[20];
sprintf(s, "wont_work%d", procnum);
fp = fopen(s, "w");
```

Example 7. An improper use of CUBIX.

All processors wishing to do I/O should open the same file, or one processor (processor 0, say) should be designated as the I/O controller and all I/O should be funnelled through it.

Asynchronous CUBIX manages the case in which several processors open a single file by maintaining a record of each processor's position in the file. Thus, each call to *read* or *write* is accompanied by a call to *lseek* which is hidden from the programmer.

During output, all processors will be writing to the same file (or files), but each one has a unique pointer into the file. The user must arrange to keep them from interfering with one another. Two slightly obscure features of the UNIX file system deserve comment because they are extremely useful in this regard. The first is that 'holes' in a file do not consume disk space. Thus, a file created as follows:

```
fp = fopen("striped", "w");
fseek(fp, procnum*100000L, 0);
fprintf(fp, "hello world from processor %d.\n", procnum);
```

Example 8. The use of striped files.

will use approximately one disk block per processor, even though its length is 100kbytes times the number of processors. This trick is useful for keeping the output of different processors well separated. Unfortunately, it does not work in NCUBE's AXIS operating system.

Another feature of the UNIX file system is the O_APPEND file mode. When a file is open for output with the O_APPEND flag bit on, all writing is done at the end of the file, regardless of the position of the file pointer.

2.6 Debugging tools.

We took a rather *ad hoc* approach to the debugging of MOOSE, i.e. we developed tools as we needed them. These tools are available to the application programmer. Additionally, since we can hardly expect that MOOSE is bug free, we have left debugging code in the system which can be activated at run time. This detracts from the speed and elegance of the system, but it is necessary at this early stage of development.

Debugging loosely synchronous CUBIX programs is generally accomplished by inserting print statements into the code until the bug is sufficiently localized that it can be found by inspecting the code. While hardly an advanced debugging technique, it is generally adequate. Unfortunately, this method has the serious drawback that it relies on the communication system and I/O system to be operational in order to be of any value. If either of these systems has failed, the programmer gets no information. Since the systems we have been using (except the iPSC) do not have memory protection, MOOSE cannot protect itself, and hence its communication and I/O sub-systems are susceptible to accidental disruption by the application program.

We have partially avoided the reliance on working subsystems by introducing the concept of a RAM stream. A RAM stream is used the same as any other standard I/O stream (i.e. FILE pointer) except that data written to it is never flushed. Instead, the data remains in the memory of the processors, and can be retrieved, even after complete collapse of the operating system, by utility programs run from the host. The standard I/O stream, *stdsys*, is automatically opened as a RAM stream. By default, slightly less than 32k bytes of memory are reserved for *stdsys*. It is not fool-proof, but it is extremely useful.

When MOOSE itself detects an error, or if the application program calls the function *error*, an extensive set of diagnostics including stack backtraces is written into *stdsys*. This information can then be obtained with utilities run from the host that retrieve *stdsys*.

2.7. Performance.

Speed was of paramount importance in the design of MOOSE, but not in its implementation. Thus, we consistently asked the question: "Can it be done quickly?", when considering a new feature or algorithm. On the other hand, we rarely had the time or patience to actually do it 'quickly', analyzing code for 'hot spots', coding in assembly language, etc. In fact, MOOSE still contains numerous expensive calls to debugging routines, which slow it down but which will help us to track down the remaining bugs. Nevertheless, one is still interested in just how fast or slow it is. The following figures are for our NCUBE implementation. We believe they could be improved by a factor of three or more with careful optimization.

The time to communicate messages between adjacent processors when neither processor undergoes a context switch is approximately 500μsec + (message length)3μsec/byte.

The time to context switch, using semaphores, is approximately 1–2 msec.

3. The Future

MOOSE is believed to "work" in in its current state. However, several important extensions are impossible without more sophisticated hardware. The kinds of developments which we envision are described in the following sections. In almost all these cases, the existence of memory management hardware is almost essential for the work to proceed in a meaningful way. In principle, one can do many of these tasks without hardware support, but the software expense, would be exorbitant, and would prevent one from learning the correct lessons from the exercise. The following sections outline the various directions in which research on MOOSE can continue. Since they are predictive rather than reflective, they are necessarily sketchy. Much of this work is being carried on by Jeff Koller and the interested reader is referred to his paper in this proceedings [Koller 88].

3.1. Memory Management, and Protection.

The MOOSE system calls and their associated data structures should be protected from accidental corruption by the user program. This is absolutely essential for a robust system, and it would be extremely helpful for program development as well. Currently, a simple error in the application program can destroy the code and data of the operating system. Such errors, when they are detected at all, often appear as internal MOOSE errors, suggesting a bug in the operating system, instead of a bug in the application program. Debugging would be vastly simpler if the hardware detected illegal references, and immediately aborted the application program with an informative message. Jeff Koller, [Koller 88], has ported MOOSE to the Intel iPSC, making use of the memory management features of the Intel 80286 processor. That work is presented elsewhere in this proceedings.

3.2. Teams.

MOOSE's strategy of sharing all code and data between all processes executing on a processor is born of expedience rather than design. Given memory management hardware capable of translating virtual addresses into absolute addresses, it becomes possible to protect tasks from one another. In such an environment, the programmer can decide that certain tasks will share code and data, which will be distinct from the code and data of other tasks on the same processor. Such a collection is termed a *team*. Teams and tasks are complimentary in that teams define a program's use of memory, and tasks define a program's use of the CPU. With this realization, it is clear that the tasks and teams are not simply the beginning of an infinite hierarchy. With the exception of global structures like global memory, and a file system (see Section 3.4 below), there are no more resources to be allocated and partitioned.

With the introduction of teams, the system call *task* loses some of its complexity. The *task* system call will create a task in the parent's team. There is no need to specify the processor on which the task will run, and the specification of input and output pipes may also become unnecessary. The migration of teams from one processor to another may be under programmer control or under the control of a load-balancer.

3.3. Relocation and Load Balancing.

Since *teams* will be completely self-contained, with their own code and data, they may be moved from processor to processor. Thus, strategies for automatic load balancing can be studied after the implementation of teams. This is a very large and exciting area, deserving of more than a single paragraph.

3.4. Global memory and leagues.

The ability to share small control structures, such as array sizes, timesteps, filenames, etc., would make programming considerably easier. A global dictionary is an appropriate mechanism for sharing such data. Since this dictionary constitutes another system resource, logically independent of, and larger than teams and tasks, it suggests the need for another level of the team/task hierarchy. We provisionally call objects at this level *leagues*, since they are primarily collections of teams. Another appropriate name is *program*, since objects at this level are really complete and self-contained entities solving a single problem.

3.5. Object-oriented programming

Initially, we intended MOOSE to incorporate object-oriented concepts at all levels. We hoped to use the C++ programming language for the implementation, and use the work of [Stroustrup, 84] as a starting point for multi-threaded programming. Unfortunately, deficiencies in certain compilers prevented us from realizing this idea. Since the available compilers have improved somewhat with time, we are in a position to reconsider this important idea. Incorporating an object-oriented methodology into MOOSE would assist programmers in writing portable, reliable and understandable code. Constructs such as teams and tasks are ideal candidates for 'objects' in such a system.

3.6. Miscellaneous

Our experience during both system development and application development has suggested that the programmer interface could be profitably redesigned. Semaphores, while provably sufficient for all synchronization tasks, are sometimes quite difficult to use. More structured objects, like *monitors* would provide the programmer with a more powerful system.

Pipes would be more efficient (and more deserving of the name) if they had buffers at both ends, rather than a single buffer. Such a change would require a major overhaul of the semantics as well as the code.

We have found that temporary pipes, which are used once and thrown away are very common both internally to MOOSE, and in application programs. Special system calls to support this type of pipe would simplify coding, and could lead to a significant performance increase.

4. Lessons

Using MOOSE has taught us some important lessons about parallel asynchronous programming. In retrospect, perhaps, these are things we should have realized at the outset. The most significant is that asynchronous parallel programming can be vastly more complex than synchronous

parallel programming. The difficulties arise because one's implicit assumptions about the behavior of a program may be incorrect. A multi–threaded program does not execute its instructions in an order readily determined from the structure of the code. Critical sections are not hard to implement, the hard part is realizing that they are needed. When a bug does appear, it is often non–repeatable and timing–dependent, and it can evade the standard "print-it-and-see" debugging techniques. Asynchronous programming desperately needs a set of rules of thumb, akin to those introduced twenty years ago for "structured programming."

One useful rule is to avoid unnecessary task creation. Don't get carried away with using tasks when a simple function call will do. Another is to put the communication and synchronization structures in place at the very beginning of a program, right at the beginning of *main*, and before tasks start creating and destroying one another. Finally, avoid the use of shared memory, except for "almost-constant" data that is written once at the beginning of the program.

These suggestions are clearly not the last word on structured asynchronous programming. Extensive application development needs to be undertaken, and lessons must be drawn from from a much larger body of experience. Nevertheless, abiding by the rules above is clearly an aid to writing understandable, bug-free code.

5. Acknowledgments

This work was supported in part by Department of Energy Grant No. DE–FG03–85ER25009, the Program Manager of the Joint Tactical Fusion Office and the ESD division of the USAF as well as grants from IBM and SANDIA. In addition, J.S. was partially supported by a Shell Foundation Fellowship.

6. References

Dijkstra, E.W., Co-operating Sequential Processes, in *Programming Languages*, F. Genuys Ed., Academic Press, New York, 1968, pp. 43-112.

Ben-Ari, M. *Principles of Concurrent Programming*, Prentice-Hall International, Englewood Cliffs N.J., 1982.

Comer, D., *Operating System Design: The Xinu Approach*, Prentice-Hall, Englewood Cliffs N.J., 1984.

Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D., *Solving Problems on Concurrent Processors*, Prentice-Hall, Englewood Cliffs N.J., 1988.

Jefferson, D., Sowizral, H., SCG Conference on Distributed Simulation, San Diego, 1985

Jefferson, D., *The Time Warp Operating System*, Proceedings of the Third Conference on Hypercube Computers and Applications, ed. G.C. Fox, SIAM, Philadelphia, 1988

Koller, J, *A Dynamic Load Balancer on the Intel Hypercube*, Proceedings of the Third Conference on Hypercube Computers and Applications, ed. G.C. Fox, SIAM, Philadelphia, 1988

Salmon, J., CUBIX: Programming Hypercubes Without Programming Hosts, in *Hypercube Multi-Processors 1987*, ed. M.T. Heath, p. 3, SIAM, Philadelphia, 1987.

Stroustrup, B., *A Set of C++ Classes for Co-routine Style Programming*, AT&T Bell Laboratories Computer Science Technical Report, AT&T, Murray Hill NJ, 1984