CS495 **T. C. Mowry**
Parallel Computer Architecture and Programming **Date: 01/25/02**
Spring 2002

ANL Parallel Processing Macro Package Tutorial

Introduction

The Argonne National Laboratory's (ANL) parallel processing macro package provides a virtual machine that consists of a shared global memory and a number of processors with their own local memory. The macros themselves are a set of process control, synchronization, and communication primitives implemented as C-language `m4` macros.

The use of macros has the advantage of portability. Unfortunately, the use of macros also makes debugging more difficult, since error messages are given in reference to the C program generated from the original program. (While these problems have been resolved in the subsequent "p4" macros from ANL, unfortunately I have not been able to find a verion of "p4" which has been ported to SGI Origin machines—if you find one, let me know.)

This document describes a subset of the ANL macros as an introduction to their use. A more complete description of the ANL macro package appears in *Portable Programs for Parallel Processors* by Ewing Lusk et. a, published by Holt, Rinehart, Winston, Inc., New York.

Macro Description

In this document the macros are grouped by function: environment specification, process control, shared memory allocation, synchronization and timing. Notice that some macros expand into definitions or declarations of data structures, while other expand into executable code.

Environment Specification Macros

Some of the macros assume the existence of certain data structures. The MAIN_ENV and the EXTERN_ENV macros contain the necessary definitions and declarations; MAIN_INITENV performs the required initialization.

MAIN_ENV contains types and structures used internally in the macro package. It should appear in exactly one file (typically the main file) in the static definitions section before any other macros usage.

EXTERN_ENV contains definitions and external declarations and should appear in the static definitions section of each separately compiled module in which MAIN_ENV does not appear.

MAIN_INITENV is an executable macro that initializes data structures defined by MAIN_ENV. The code generated by this macro must be executed before that of any other macro, thus typically appearing very early in the program's main function.

MAIN_END should be placed at the end of your main routine, just before your program exits. It cleans up any structures used by the ANL macros. On the SGI machine, it removes any shared memory which has been allocated. You **must** put this in all your programs to avoid leaving shared memory allocated when your program exits.

Shared Memory Allocation Macros

It is a good idea to declare a single structure, say *gm_t*, as global memory, and use a single call to G_MALLOC to allocate this structure, say in variable *gm*. Parts of global memory can then be referenced as *gm->someVar*.

G_MALLOC(*size*) behaves like the Unix/C malloc call, except that the pointer returned points to globally shared memory which is accessible to all pointers. For example,

```
gm = (struct gm_t*)G_MALLOC(sizeof(struct gm_t));
```

where `gm_t` is a structure declared earlier.

G_FREE(*ptr*) de-allocates memory allocated by G_MALLOC, and is similar to the Unix/C free procedure.

Process Control Macros

CREATE(*entryProc*) causes a process to be created and start executing the procedure *entryProc*. No arguments can be passed to the new process, or as parameters to *entryProc*. The process is a Unix-style process and, in fact, CREATE uses the *fork* system call.

Each process will be assigned to a different processor until there are no processors left. Processes are Unix processes, and compete for system resources with all other processes on the system. As one would expect, process creation is typically relatively slow, which sets a lower limit to useful task granularity.

Note: At the point when a process is created, all of the parent's static data, including the pointer to global shared memory, is *copied* once into a separate address space for the created process. The only memory that is shared is the memory explicitly allocated by G_MALLOC. Globally allocated data is static.

WAIT_FOR_END(*nProcs*) waits for *nProcs* processes created by this process to exit.

Synchronization Macros

There are macros provided for locking, barriers, and distributed loops. In each case, there is a macro for declaration (its name ends in DEC), *the declaration macro should appear within a structure that is allocated with* G_MALLOC, so that it will be globally shared and accessible to all processes. Another macro contains initialization code (its name ends in INIT); *the initialization must occur before any use*.

Lock Macros

LOCKDEC(*lockName*) contains a lock declaration.

LOCKINIT(*lockName*) initializes the lock *lockName*.

LOCK(*lockName*) attempts to acquire ownership of the lock named *lockName*. If no other process currently owns the lock, then the process becomes the owner of the lock and proceeds. Otherwise, it is delayed until it can acquire the lock.

UNLOCK(*lockName*) relinquishes ownership of the lock named *lockName*. If other processes are waiting to acquire the lock, one of them will succeed.

When multiple locks need to be acquired, deadlocks can occur. Perhaps the simplest strategy to avoid deadlocks in this case is to have all processes acquire the locks in the same order.

If the created processes all try to output to standard output at once, there can be trouble - so use a lock to access standard output, or let only the main process generate output.

Barrier Macros

A barrier is used to hold processes at a particular point *p* in a program until a specified number of processes have reached *p*.

BARDEC(*barName*) declares a barrier with the given name.

BARINIT(*barName*) is an executable macro that initializes the barrier.

BARRIER(*barName*, *nProcs*) stops all processes reaching this barrier until *nProcs* processes have reached it. When that happens,

1. Barrier *barName* is reinitialized; it is not necessary to call BARINIT(*barName*) again.
2. All the processes continue on from the BARRIER statement.

Distributed Loops: Get Subscript

These macros aid in coordinating a distributed or self-scheduled loop. A self-scheduled loop is executed in parallel; each process dynamically acquires the next iteration to be executed (in this case, by first obtaining its corresponding index value).

GSDEC(*name*) declares an instance of a distributed loop.

GSINIT(*name*) initializes internal variables of the distributed loop.

GETSUB(*name, subscript, maxSub, nProcs*) sets *subscript* to the next available subscript. When all subscripts in the range 0 to *maxSub* (inclusive) have been returned, the following will happen to a process executing GETSUB, in that order:

1. The GETSUB operation is delayed until *nProcs* processes have requested an out-of-range subscript.
2. Loop instance *name* is reinitialized; it is not necessary to call GSINIT(*name*) again.
3. A value of -1 is returned for *subscript*.

NOTE: a common mistake is to pass in the *address* of an object as one of the arguments to GETSUB—instead, you should pass in the *name* of the object (the address of the object is taken implicitly as part of macro expansion).

Timing Macros

Execution time of part or whole programs can be measured using the CLOCK macro. It gives the current elapsed time in some time unit (1 microsecond on the KSR and SGI), no actual CPU time, which means that it is important in general that no other programs run during time measurements.

CLOCK(*time*) sets time to the current timer value, from 0 to 2^{32} - 1, where time is declared as:

```
unsigned int time;
```


Since the timer is a system-level timer, it's possible for the value to wrap around while your program is running. This means that you should output both the start and end time, as well as the actual elapsed time (the difference). Then you can check whether the timer wrapped around while the program was executing.

Using the ANL Macros on the NCSA SGI Origin 2000 Machines

Debugging

Your programs should be written so that they will run with a variable number of processes/processors (allow the number to be specified as a command line argument). It is a very good idea to begin by debugging your program using a single processor somewhere. If that works, then move to two processors. Only if that works should you try more than two processors. You can run on a single processor anywhere (even on your own UNIX machine), by using the Makefile in *ASSTDIR*/`example_uniprocessor`.

The SGI debugger (`dbx`) is capable of debugging parallel codes. Note that *in order to use dbx or udb, you must compile with the "`-g`" rather than the "`-O2`" `CCFLAGS` option*. However, for your timing runs, you should use "`-O`" to get the best performance.

When the compiler complains about an error in your `*.c` file, the line number that it gives you refers to this `*.c` file, and not your original `*.U` file. Hence you must look in your `*.c` file first to determine where the problem is in your `*.U` file. (Oh, the joys of using m4 macros.)

Note that the `m4` macros can be delicate; a misplaced space or semicolon can lead to seemingly bizarre behavior—keep this in mind when debugging your programs. For example, a space cannot occur between a macro name and the opening '(' for its argument list. Thus, `G_FREE(ptr)` calls the macro `G_FREE` with the argument *ptr*, while `G_FREE (ptr)` calls the macro `G_FREE` with no argument (note that this will not necessarily lead to an error message). See the `m4` man pages for a quick overview on the use of `m4` macros.

Kill Those Stray Processes!

When your parallel program fails on an SGI machine, the associated processes will not necessarily terminate. Always check to see if any processes are still running by typing "`ps -ef | grep yourLoginName`". If so, use "`kill -9 processID`" to kill the associated processes.

Free those shared memory segments!

On an SGI machine, be very careful to release any shared memory segments that your programs reserve. If your program always executes "`MAIN_END`" when it exits the program, then you should be okay. Note: do *not* exit the program by simply saying "`exit(1)`" - this does *not* free the shared memory segment.

To check whether any shared memory segments are outstanding, use "`ipcs`". To free an outstanding memory segment, type "`ipcrm -m <segment_id>`", and to free a synchronization object, type "`ipcrm -s <segment_id>`".