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. In fact, there are two layers of macros: a small set of primitive, machine-dependent macros written in terms of target machine functions and a larger set of machine-independent macros written in terms of the machine-dependent macros, so only few machine-dependent macros need to be ported to a new machine. 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.

This document describes a subset of the ANL macros as an introduction to their use.

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 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 macro 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_INTENV 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 it typically appears very early in the program's main function.

MAIN_END must be the last thing in your main() routine. It is used for any cleanup necessary to support the programming environment.

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, say, *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 processes. For example,

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

where gm_t is a structure declared earlier.

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

P_MALLOC(*size*) behaves the same as the Unix/C malloc call, but is used for individual processes which returns a pointer that is not accessible to other processes.

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. 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.*

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 given by *lockName.* If other processes are waiting to acquire the lock, one of them will succeed. Shen 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 fixed 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.

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 statement 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 this 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*.

Timing Macros

Execution time of part of whole programs can be measured using the CLOCK macro. It gives the current elapsed time in some time unit, not actual CPU time, which means that it is in general important 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*;

The CLOCK macro will typically be used to computer time difference, so the fact that *time* may wrap around does not matter, as long as your program takes less than 2^{32} time units.