ANL Parallel Processing Macro Package

Introduction

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

The advantage of using these macros is portability. Different machines and operating systems provide different support for the constructs often needed in parallel programs. The macros shield programmers from knowledge of a machine's specific implementation and allow them to concentrate on writing a correct and efficient solution to their problem. Unfortunately, the use of macros complicates the debugging process because compiler error messages are given in reference to the C program after these macros have been expanded.

This handout 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 Parallel Programs for Parallel Processors* by Ewing, Lusk et. al, published by Holt, Rinehart, Winston, Inc., New York.

Macro Descriptions

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 others expand into executable code. In general, macro statements **never end with a semicolon** (while this is arguably bad design, you have to remember it or you will find some weird errors). Note that **G_MALLOC** is a function and as such part of a C statement that does end with a semicolon.

Environment Specification Macros

Many of the macros require 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`
Defines the global symbols, variables, and functions that are used internally by the other macros. This macro should appear at the beginning of the main source file for the application, outside of any procudure. (Every application should contain exactly one instance).

`EXTERN_ENV`
Similar to `MAIN_ENV`, but this macro declares variables with the extern qualifier to indicate that the actual definition appears in another file. This macro should appear at the beginning of each source file except the main source file containing `MAIN_ENV`.

`MAIN_INITENV`
Initializes data structures defined by `MAIN_ENV`. The code generated from this macro must be executed before that of any other macro. It typically appears as the first executable statement of the application (i.e. inside the main procedure).

MAIN_END
Cleans up any structures used by the ANL macros. On parallel machines, it will also remove any shared memory which has been allocated. It is typically placed at the end of the main routine and **must** be executed to avoid leaving shared memory allocated when your program exits.

Shared Memory Allocation Macros

It is a good idea to declare a single structure, such as `gmem_type`, as global memory and use a single call to `G_MALLOC` to allocate this structure. for a single variable such as `gmem`. Parts of the allocated global memory can then be accessed as `gmem->variable`. This will become clearer upon examination of the sample program.

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. The example discussed above would be written as:

```
gmem = (struct gmem_type *)G_MALLOC(sizeof(struct gmem_type));
```

G_FREE(*pointer*)
De-allocates the memory allocated to this pointer in `G_MALLOC` and behaves the similarly to the UNIX/C `free()` procedure.

Process-Control Macros

These functions provide the highest level functionality of parallel programming. You will call `CREATE()` once for every processor that you want to use. You should make sure that the number of processes you `CREATE`() doesn't outnumber the available processors. You should call `WAIT_FOR_END()` at the end of your program to avoid leaving "zombie" processes running when your main process exits.

CREATE(*procedure-name*)
Causes a process to be created and to start executing the procedure sent as the parameter. No arguments can be passed to the new process or to the procedure that will begin execution. The process is a Unix-style process and is actually created by a call to UNIX's `fork()`.

On the SGI machine, each process will be assigned to a different processor until there are no processes left. These processes compete for system resources with all other processes on the system. Process creation is expensive, so we advocate creating processes once and then terminating them at the end of the parallel part of the program.

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(*num*)
This macro typically is placed towards the end of the main procedure and causes the process to wait for *num* processes created by this process to exit.

Synchronization Macros

There are a number of macros provided for locks, barriers, and distributed loops. In each case, there is a macro for declaring the parallel construct before it can be used. **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**. Other macros contain the initialization code for each these constructs. These initialization macros must be executed before any use of the corresponding global synchronization variables.

Lock Macros

Locks form the foundation of parallel programs and are used to ensure mutual exclusion in critical sections of code.

`LOCKDEC`(*lockName*)
Declares a lock variable with the specified *name*. Only declare locks in shared memory.

`LOCKINIT`(*lockName*)
Initializes the lock variable with the specified *name*. Upon creation, locks are unlocked.

`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, the process is delayed until it can acquire the lock. When multiple locks need to be acquired, deadlocks can occur. One of the simplest strategies that you can use to avoid deadlocks is to have all processes acquire the locks in the same fixed order.

`UNLOCK`(*lockName*)
Relinquishes ownership of the lock given by *lockName*. If other processes are waiting to acquire the lock, one of them will succeed.

NOTE: If a number of created processes attempt to output to the standard output at the same time, the output can be garbled. The easiest way to get around this is to use a lock to protect the standard output. This is not an issue if only one process generates the output.

Barrier Macros

A barrier is used to hold processes at a particular point in their executions until a specified number of processes have reached this barrier.

`BARDEC`(*barrierName*)
Declares a barrier with the specified *barrierName*. Similarly to locks, barriers should only be declared in shared memory.

`BARINIT`(*barrierName*)
Performs initialization on the barrier *barrierName*.

`BARRIER`(*barrierName*, *number*)

Stops all processes reaching this statement until the specified *number* of processes have reached it. When the final process reaches the barrier, *barrierName* is re-initialized. It is not necessary to re-execute `BARINIT`. At this point, all of the processes continue on with the instruction after the barrier.

Distributed Loops: Get Subscript

These macros aid in coordinating a distributed or self-scheduled loop. A self-scheduled loop is executed in parallel with each process dynamically acquiring the next iteration to be executed. This macro provides an easy mechanism for creating task queues.

`GSDEC`(*name*)
Declares a self-scheduled loop subscript with the specified *name*.

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

`GETSUB`(*name, subscript, maxSub, numProcs*)
Sets *subscript* to the next available loop index. When all subscripts in the range 0 to *maxSub* (inclusive) have been returned, the following will happen to a process executing the macro:
1. Each Process will wait at a `GETSUB()` until all *numProcs* processes have requested an out-of-range subscript.
2. Loop instance *name* is reinitialized; it is not necessary to call `GSINIT()` again.
3. A value of -1 is returned for *subscript*.

Timing Macros

Execution time of sections of programs or of complete programs can be measured using the CLOCK macro. It gives the current elapsed time in some time unit (Morgoth uses 1 microsecond), not in actual CPU time. This means that it is important that no other programs are running at the same time during timing runs.

`CLOCK`(*time*)
Sets *time* to the current timer value, from 0 to 2^{32} - 1, where time is an unsigned integer. since the timer is a system-level timer, it is 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 between start and end times). This allows you to check whether or not the timer wrapped around during execution.

Using the ANL Macros

You should use the macros on uniprocessor UNIX machines for code development and on the parallel machine to actually measure multiprocessor performance. The macro files you will need can be found in `~bilas/class/macrodir`.

On the uniprocessor machines, you should use the macros named `c.m4.not`. These macros are substitutes for any real machine ports and simply allow you to compile and run your programs on a uniprocessor. On morgoth you should use the macros named `c.m4.sgi`. The file `shmem.o` in the macrodir directory must be linked in when running on morgoth.

We will be using the convention that all source code containing unexpanded macros has the extension ".U" and the macro-expanded files have extension ".c". A makefile must first run the source ".U" files through the m4 macro processor and then compile and link the ".c" files. Example makefiles can be found in the class directory.

Morgoth Specifications

Morgoth is a single global address space shared-memory computer with 16 processing elements. Each processing element consists of a 150MHz MIPS R4400 processor and per-processor 2-level caches. The first level data and instruction caches are each 16 kilobytes, and there is a 1 megabyte unified second-level cache. There are 512 MB of centralized shared memory connected to the processors by a shared bus. You can use the `hinv` command to get more information.

Batch Runs on Morgoth

To accurately measure the amount of speedup that is being obtained through the use of multiple processors, special precautions must be taken. You must ensure that the program of interest is the only one executing on the machine during all timing runs. To facilitate this, we have created a batch facility. This reservation program is in `~bilas/class/reserve` and is invoked as:

```
reserve -f cmdFile -d day [-o outFile]
```

"cmdFile" is a script file that will be executed at some point during the batch run. "day" is the day of the month for this to be executed. Note that these runs will be run after midnight, so the day should be one more than you think.

DO NOT WORK INTERACTIVELY ON MORGOTH FROM MIDNIGHT UNTIL MORNING!

Debugging

Your programs should be written so that they will run with a variable number of processes/ processors. Allow this value to be specified as a command line argument. **You should debug your program using a single process on a uniprocessor workstation!!!** You will not be able to run with multiple processes on the uniprocessors. Only move onto morgoth when you are sure that the single process works correctly. Debug on morgoth using only two processes at first and only move up to more when this works correctly.

The debugger available on morgoth is `dbx`. Read the man pages for information on its multiprocessor support. Or use print statements in the code.

When your parallel program fails on morgoth, all of the associated processes will not necessarily terminate. **KILL THOSE STRAY PROCESSES!!!** Type `ps -u <username>` to see if there are any of your processes still running. If so, use the `kill -9` command to kill the associated PID. Please be sure that you check for stray processes every time you log out! Your program may also leave shared memory allocated if it exits abnormally. Run the command `ipcs` to check whether you have any shared memory allocated. If you do, use the command `ipcrm -m id_number` to remove them. Man pages are available for both of these commands.

Any compilation errors indicate line numbers for the expanded ".c" file and you have to look at that file to figure out where in the original source ".U" file these errors occur. This is not difficult, but if you forget this fact, you may waste a lot of time.

Note that `m4` macros can be quite 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 appear 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`<space>(*ptr*) calls the macro `G_FREE` with no

argument (this will not necessarily lead to any error message). See the `m4` man pages for a quick overview on the use of m4 macros.