A Loader For Executing Multi-binary Applications on the Thinking Machines CM-5: It's Not Just For SPMD Anymore

Jeffrey C. Becker[1]

Report NAS-95-007 February 1995

NAS Systems Division
NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

Abstract

The Thinking Machines CM-5 platform was designed to run single program, multiple data (SPMD) applications, i.e., to run a single binary across all nodes of a partition, with each node possibly operating on different data. Certain classes of applications, such as multi-disciplinary computational fluid dynamics codes, are facilitated by the ability to have subsets of the partition nodes running different binaries. In order to extend the CM-5 system software to permit such applications, a multi-program loader was developed. This system is based on the dld loader which was originally developed for workstations.This paper provides a high level description of dld, and describes how it was ported to the CM-5 to provide support for multi-binary applications. Finally it elaborates how the loader has been used to implement the CM-5 version of MPIRUN, a portable facility for running multi-disciplinary/multi-zonal MPI (Message-Passing Interface Standard [5]) codes.

1. Computer Sciences Corporation, NASA Contract NAS 2-12961, Moffett Field, CA 94035-1000

1.0 Introduction

One of the goals of the Numerical Aerodynamic Simulation (NAS) facility at NASA Ames Research Center is to provide a sustained Teraflops computing rate on Computational Fluid Dynamics (CFD) codes. To this end, NAS has been evaluating parallel systems such as the Thinking Machines CM-5, as replacements for shared-memory vector computers such as the Cray C90. One class of CFD programs that is targeted is the so-called multi-disciplinary codes in which several disciplines are brought together and coordinated to perform a simulation [1]. For example, one code that is being run with increasing frequency at NAS is a fluids-structures code in which a fluids code component and a structures code component interact at each time step to perform a simulation of an aircraft structure such as a wing.

The CM-5 was primarily intended to run data parallel codes written in CMF or C*. While the CMMD message passing library can be used to communicate between nodes that are executing different parts of the same code, this is the limit to the degree of MIMD programming that is available. The CMost operating system and time sharing software running on the CM-5 only provide facilities for jobs consisting of a single executable on a partition, with each node possibly operating on different data. Users submit jobs to a partition manager, a Sun SparcStation 2 front end controlling some multiple of 32 nodes. The operating system gang schedules all the nodes of a partition to run each job. Different users can timeshare on a partition, but each user's job is considered as a single (parallel) process by the OS. In fact, the operating system process model is restricted to this single process view. System calls for process creation are not available as in standard UNIX. Thus, the system had to be enhanced to provide a means of running multiple binaries (executables) on the same partition.

Several approaches for developing a multi-binary loader were evaluated:

- Modify CMOST
- Use Linda/Piranha loader as starting point
- Use GNU dld loader as starting point

The GNU dld loader program originally designed for UNIX workstations was selected. The dld system allows a process to load an executable file into its image and access functions from the loaded binary. This could be used to provide the desired functionality on the CM-5 as follows. First, a common loader program is submitted to run on all the nodes of a partition, using the existing system software. This could then load a (possibly) different binary on each node. Thus, dld was acquired and ported to the CM-5.

As a major test of the loader, it was used as a basis for porting the MPIRUN loader [3] to the CM-5. This system was designed by Sam Fineberg at NAS to enhance the Message Passing Interface (MPI) standard [5] with a capability for program loading. MPIRUN has already been ported to several other parallel computers including the Intel Paragon and IBM SP2. However, these systems all provide some form of program loading similar to the standard UNIX fork and exec process spawning facilities, so porting MPIRUN on these machines was relatively straightforward. As mentioned earlier, the CM-5 system software does not provide this capability. However, the dld based loader provides a way to emulate it. It was relatively easy (compared to porting dld itself) to implement MPIRUN on top of the dld loader.

2.0 CM-5 System

The CM-5 at NAS has 128 SPARC processor nodes, with four attached vector units. Each node has 32 MegaBytes (MB) of local memory, and peak performance of 128 MFLOPS. The interconnection network is configured in a fat-tree topology. The system contains a Scalable Disk Array, a RAID device comprising 48 GB of fast parallel disk storage. Three SparcStation front ends act as partition managers. Finally, there are three HiPPI interfaces for connecting to other machines from the CM-5.

The CM-5 runs the CMost operating system, a parallel but limited version of UNIX. The OS allows each job to run the same binary on all nodes of a single partition, with each node accessing data stored in its local memory. As mentioned above, each job is run as a single process gang-scheduled across the partition. The OS supports a limited form of timesharing on the nodes, in which a (potentially different) user's parallel process is gang-scheduled on the partition in each time slice. Only one user process is allowed on the partition per time slice. Jobs are submitted through the Distributed Job Manager [6]. Standard UNIX features not implemented by CMost include system calls for process management, such as fork and exec, and virtual memory. Message passing is accomplished by calling the CMMD communication library. Vector unit access is provided implicitly in the data parallel languages CMF and C*, or explicitly through the DPEAC assembly language. There is no way to access the vector units with standard C or Fortran.

3.0 Extending the CM-5 to Run Multiple Binaries on a Partition

Probably the most direct way to run multi-disciplinary codes on the CM-5 is to have all nodes in a partition run a code resembling the following:

```
1.    switch (node number)
2.          case range i: do discipline i
```

In fact, this basically represents the extent to which the CM-5 can run MIMD codes, i.e., the multiple instruction streams must come from a single binary. This is a very inefficient use of memory since all application disciplines will be contained entirely on every node, but only a single discipline will be run by each.

Clearly, it would be better to have each node only load and run only the binary corresponding to its discipline.

The CM-5 system software in its current state is not able to load different binaries on a partition. If this could be accomplished, the CMMD message passing library could be used to communicate between the different codes. Thus, the first step was to enumerate the options available to carry out this task. Basically, there are three possible solutions.

The first solution is to modify the CMost source code. The principal part of the code requiring modification is the page table management routines. The OS currently has one page table per job running on each partition. In order to allow each job to be comprised of several binaries, this would have to be changed to several page tables per job, one per binary. For example, each job could be associated with a linked list of page tables. In addition, for each code comprising a job, the system would have to keep track of the nodes it was running on. Assuming that each code runs on a contiguous range of nodes, a pair of integers could be associated with each job. While the additional functionality is not very substantial, it seemed that modifying CMost is a very difficult task as there are over 6000 source files, and it is not trivial to pinpoint all the necessary changes, and take care of all possible interactions affected by those changes. In particular, much of the code depends on the single process model referenced earlier. Thus even a simple change could have wide effect. In addition, it would be very difficult to maintain this system.

The second option was mentioned by our on-site TMC support staff [7]. As part of the Linda project at Yale University, Eric Freeman developed a loader to support loading multiple binaries on a CM-5 partition [2]. It works as follows. First, the different binaries are each linked to begin execution at a fixed address. At run-time, a single bootstrap loader is started up on all nodes of a partition. Each node receives a message from the partition manager telling it the size of the code it is to receive. Memory is then allocated in the data segment of the loader using the valloc system call. Once the space is allocated, the partition manager then broadcasts each binary to the target subset of nodes. Normally, the CM-5 broadcasts can only be done to all nodes in a partition. In order to circumvent this, special low level calls are made to the network interface of the nodes not receiving the binary to tell them to abstain from the broadcast. These must be reset between each subset broadcast, and before the binaries start execution. When all loading is complete, all nodes jump to the prespecified location.

Although this effectively accomplished the desired goal, two aspects to its implementation make it unattractive. The first is the implementation's inefficient memory utilization. The valloc call allocates memory aligned on a page boundary. Thus any memory between the end of the loader text segment and the next boundary is wasted. The other shortcoming is the requirement that codes be linked to start at a fixed address. This is not desirable since each new version of the loader could require specifying a new address. At the outset of the project, the possible use and evolution of the loader was unclear. Thus having to change the start address several times was not practical. Of course, we could have used a large starting address in anticipation of loader text segment (code) growth, but this would have exacerbated the problem of memory waste.

The solution which was eventually adopted and ported to the CM-5 was found in the GNU software repository. This code, dld, was developed by Wilson Ho while he was a graduate student at the University of California at Davis [4]. It is a dynamic linker system designed to link one binary into the data segment of another at run-time. It reads in the text and data of the desired binary, allocating only as much memory as desired. In addition, this information is read in to memory immediately following the memory used by the executable being read into. Unlike the Linda loader described above, no memory is wasted and the entry point to the loaded binary is not fixed. The code does not have to linked to load at a particular address. This has the additional advantage that codes do not have to be relinked prior to use with dld. Thus, for example, each component of a multidisciplinary application can be developed and tested individually and the resulting binaries can be fed directly to the loader. The final reason for selecting dld is that it was already a complete buildable source package, whereas the previous loader source comprised pieces of code extracted from the Linda system. Thus, dld seemed to provide a better starting point.

4.0 Dld

Dld is a library that provides the ability to dynamically link in relocatable object files or libraries. Although several functions are provided, the salient ones for our purposes are described below.

`int dld_init (char *path)`: performs initialization and loads the symbols of the executing program (located at path) into memory so that they may be accessed by code to be dynamically linked in. It must be called prior to any other dld functions. Normally, dld_init takes the path of the executable that contains the call as a parameter.

`int dld_link (char *filename)`: performs dynamic linking of the object file or library file stored in the file named filename. All memory required for this is allocated dynamically within this function.

`unsigned long dld_get_func (char *func)`: returns a pointer to the named function, func.

In order to use dld for the CM-5 loader, each node would basically do the following:

```
1.  dld_init(loader_path)
2.  for each required library(libc.a, libm.a, etc.),
    dld_link(library)
3.  dld_link(file to be run on this node)
4.  func = dld_get_func(main)
5.  * func()
```

FIGURE 1. Dld loader pseudocode

Thus, the OS is told to run the same binary on all the nodes as usual. However, after the first step, each node could load a completely different code, and run in true MIMD fashion. Note that dld_link is for relocatable object files. Thus any libraries not already used by the "parent" binary (into which linking is done) must be linked in as well.

Dld also provides functions to unlink object files or libraries. This can be used to reclaim space and possibly link in an entirely different binary, thus allowing codes to dynamically alter their function several times during their execution.

5.0 Porting dld to CM-5

Although the version of dld obtained from the GNU project built fine and ran the sample test codes provided with the source correctly on the CM-5 Sparcstation partition managers, there were several problems that had to be resolved before dld could be run on the CM-5 nodes. This is not surprising, as dld was not originally designed for parallel machines. Although most of the problems that were uncovered had to do with this factor, several remaining bugs were also found (and fixed). Our tests probably uncovered paths in the code that had not been exercised.

The dld code had to be ported to use the CM-5's I/O facilities. There are four I/O modes provided in the CMMD library. Every file descriptor has one of the following modes associated with it.

CMMD_local: nodes can read and write different files; each node maintains its own file descriptors and file pointers.

CMMD_independent: files in this mode are opened by all nodes for independent reading and writing; a single file descriptor is stored in the process' file descriptor table.

CMMD_sync_bc: allows simultaneous reading or writing of a file by all nodes; uses special broadcast hardware and is very fast.

CMMD_sync_seq: allows scatter reads of sequential file data to and gather writes from nodes; uses special hardware for speed.

In order to achieve good performance and maintain simplicity, the CMMD_sync_bc mode was used for all loading. The prototype loader is first initialized by having all nodes read in the symbols of its executable (step 1 above). It then loads a common set of libraries on all the nodes (step 2). Synchronous broadcasts are very efficient for these operations.

At step 3, each node loads a potentially different binary. However, it's likely that each binary is run by a group of nodes, and the number of such groups is not usually very large. Thus it is efficient to do the file operations in the CMMD_sync_bc mode. The primary drawback to using this mode is that all nodes must participate. In order to conserve memory, nodes that do not use a given binary unlink it after loading is complete. Note that this would not be as important if CMost supported virtual memory since only the necessary pages would be loaded on each node. A future version of the loader could use the low level calls as used in the Yale loader to cause nodes to abstain from broadcasts of binary files intended for different nodes, thus eliminating the need for unlinking.

The unlinking step also unlinks any libraries associated with the binary being unlinked. Since those same libraries may be needed by the binary that is meant to be loaded on that node, this may cause an error. To fix this, a function to specially tag library symbols as preloaded was added to dld. The unlink function was also modified to only unlink symbols that did not have the special preload tag.

Certain symbols that are normally included by the standard linking process appeared to be missing. To fix this problem, a dummy source file defining them was created and compiled to an object file that was loaded at run time.

Step 4 in Figure 1 shows the format of the dld_get_func call for C programs. For F77 codes, the compiler uses the symbol "MAIN_" to indicate the main program. Thus this parameter is given to dld_get_func instead.

The resulting loader only works with codes written in the C and Fortran languages. The CM-5 data-parallel languages CMF and C* are not currently supported. As stated earlier, this means that the only access to the vector units is through the assembly language DPEAC. Several difficult problems would have to be resolved to support these languages, e.g., getting collective communication to work correctly within groups of nodes. Thus, this was not attempted in this initial version of the loader.

6.0 Using the loader for multi-disciplinary application support: MPIRUN

MPIRUN is a portable facility for running multi-disciplinary/multi-zonal MPI codes. It was developed at NAS by Sam Fineberg, in order to add the capability of loading multiple programs to MPI. The MPI standard does not say how to load processes. The MPIRUN system is described in detail in [3]. It provides an interface specification that describes variables and functions available to mpirun programs, and an implementation which is somewhat independent of the interface. A basic description of the implementation follows.

The MPIRUN implementation is comprised of two parts. The first part parses the command line or a file specified by the user. This is to determine how many nodes each executable is to run on. It then configures some data structures to record this information. Each code comprising the application needs to know which application group it is in as well as the leaders of the other application groups so that they can perform inter-group communication. This information is also recorded in the initialization of MPIRUN.

Following the initialization a routine, *start_procs*, is called. This routine loads the appropriate code onto the nodes as specified in the initialization, and then the codes are started. Each code must call *mpirun_init* in order to receive the (application) group and group leader information. This information is computed and stored in a file by *mpirun*. In *mpirun_init*, node 0 reads this file and sends the information to the other nodes. Communication groups for set of nodes running a particular application are also set up in *mpirun_init*. This function is in a library, *libmpirun.a*, that MPIRUN codes must be linked with.

Porting MPIRUN to the CM-5 was a relatively simple process. The only part of the system that required porting was the start_procs routine. When a job is run on the CM-5, all the nodes in the partition are used. However, the MPIRUN job may have requested fewer nodes. Thus one of the first steps in start_procs calls CMMD_reset_partition_size in order to have the correct number of nodes. This is required since the current partition is used as MPI_COMM_WORLD, the MPI variable specifying the pool of all nodes. Unfortunately, the unused partition nodes are idle while the job runs, and are not available to other jobs. This is due to a limitation of the gang-scheduling system used by CMost.

Once the partition size is reset, start_procs basically executes the pseudo-code in Figure 1 on all the active nodes. The parameters of start_procs include a list of the applications and the nodes on which they are to run. Thus step 3 is repeated in a loop through all the applications. When this is done, all nodes do step 4 and then call the main program to start their particular application.

The MPIRUN implementation was built to run using the 7/22/94 version of the public domain version of MPI from Argonne National Lab and Mississippi State University. It should be noted that testing of this MPIRUN port was done using the same test suite used to test MPIRUN on other platforms (with equivalent test codes for the CM-5). Although none of these comprise a large multi-disciplinary application, the suite tests the basic components required for their operation.

7.0 Summary

Dld provided a viable solution for allowing multi-disciplinary/multi-zonal codes to be loaded and run on the CM-5. While aspects of the CM-5's I/O and time-sharing systems made the implementation less efficient than it potentially could have been, they did not hamper its functionality. Using the loader as a basis for porting MPIRUN to the CM-5 was also successful. This can largely be attributed to the fact that MPIRUN was written to be portable. Most of the changes were localized to a single procedure. Note that this is true for other platforms as well, not just the CM-5.

The most difficult part of this work had to do with porting dld to the CM-5. At the start of this project, it looked like dld basically accomplished the desired goal, and although it was clear that some porting work would be required, it seemed that using dld would save much code development time. In retrospect, it may have been easier to write the loader from scratch. It took much time and effort to understand the dld source code, and as was mentioned earlier, it still had a few bugs left in it. While software reuse is a good general software engineering principle, one must carefully assess how much effort is involved.

8.0 Acknowledgments

I would like to thank Parkson Wong for all his help in porting dld to the CM-5, and Sam Fineberg for his assistance with MPIRUN. I'd also like to thank Sam Fineberg and Bill Saphir for their helpful comments on this report.

9.0 References

[1] Barszcz, Eric, Sisira Weeratunga, and Eddy Pramono, "A Model for Executing Multidisciplinary and Multizonal Programs", Technical Report RNR-93-009, NASA Ames Research Center, March 1993.

[2] Carriero, Nicholas, Eric Freeman, and David Gelernter "Adaptive Parallelism on Multiprocessors: Preliminary Experience with Piranha on the CM-5", Technical Report 969, Yale University, May 1993.

[3] Fineberg, S.,"Implementing Multidisciplinary and Multizonal Applications Using MPI", 5th Symposium on the Frontiers of Massively Parallel Computation, February, 1995.

[4] Ho W.W., and R.A. Olsson, "An approach to genuine dynamic linking", Software, Practice and Experience, v. 21, no. 4, April 1991.

[5] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", May, 1994.

[6] Minnesota Supercomputer Center, "The Distributed Job Manager Administration Guide", 2nd ed., 1993.

[7] Saphir, W.C., personal communication, August 1993.

[8] Thinking Machines Corporation, "CMMD reference manual", v. 3.0, 1993.