Portable Message-Passing Programming for MPPs and PVPs

Margaret A. Cahir, Cray Research, Inc,. Eagan, Minnesota

ABSTRACT: *The number of different computer architectures currently available creates problems for application developers who would like to maintain just one common source base. Early experiences in using the prototype software for the Message Passing Toolkit (MPT) to develop a single source code to run on both CRAY MPP (T3D/T3E) and parallel-vector processor (PVP) architectures as well as other types of machines will be presented. Use of the upcoming private I/O option is also documented. It is shown that this software provides an extremely portable, high-performance method of implementing message-passing while still taking advantage of many of the features of a shared-memory environment.*

1 Overview of the MPT Software

A Message Passing Toolkit is scheduled for release later this year. The initial release of the Toolkit will consist of an optimized, multitasked version of PVM, a version of MPI and a subset of the high-performance SHMEM library for the CRAY J90/YMP, C90 and T90. Eventually, the toolkit will also be available on the T3D/T3E and will contain an optimized version of MPI. These libraries allow for the easy porting of existing codes that may come from machine types ranging from clusters of workstations to true MPPs. This implementation allows for the mixing of SHMEM routines in the same application with PVM or MPI. This feature gives the application developer the option of coding critical portions of a code with the SHMEM routines to take full advantage of the benefits of shared memory for efficient data transfer and synchronization between tasks.

The PVM and SHMEM portions of this software grew out of the T3D emulator software which was used on CRAY PVP machines before the CRAY T3D was generally available. It relies on macrotasking, which is a mature and stable product. The user may find it helpful to think of the MPT as providing a PVM or MPI interface to macrotasking.

1.1 Program Initiation

Two basic options exist for initiating tasks in an application, one designed to be more familiar to network-PVM users and one designed to be familiar to CRAY T3D users.

In the first method, the master task issues a "pvm_spawn" call with the "PvmMtSpawn" flag which will cause the spawned processes to be started in a new multitasked group.

In the second approach, tasks are initiated by the master task when it calls a routine "START_PES(NPES)". A zero value passed to this routine will cause the value of the environment variable NPES to be used, thus allowing the number of tasks to be determined at run time. This approach was designed to allow codes that currently run on the CRAY-T3D to port easily while still allowing the flexibility of specifying the number of tasks to use at run time.

The second method effectively produces a single, "hostless" (in the PVM sense) executable. This is the method utilized in the codes ported to date.

1.2 Data Transfer

Data transfer between tasks is accomplished by memory copies, although exactly how this is done depends on whether PVM or SHMEM is used. In the case of PVM, data is copied from the sender's area to a shared buffer, where it sits until it is "received" by another task.

SHMEM data transfers are performed by copying memory directly from one task's stack to another task's stack. In order for one task to know the address of another task's data object, some restrictions on the storage type of the data objects must be made. We define the term "symmetric" to mean an object that exists in the same relative place on all tasks' stack and require that all data used by the SHMEM routines be symmetric. For C and Fortran codes on PVP machines, this implies that the data reside in TASKCOMMON.

On the CRAY MPP machines, symmetric data includes FORTRAN data placed in COMMON storage, C static data storage class, data allocated with shpalloc/shmalloc (FORTRAN/ C), and data declared to be symmetric via a new compiler directive (#pragma symmetric for C, !DIR$ symmetric for F90).

1.3 Barriers

The basic method for performing a barrier in the PVP version of the Toolkit is based on the macrotasking "BARSYNC" routine, which is a spin-wait barrier implemented in user's space. The simpliest, most direct way to synchronize all tasks is to use the "BARRIER()" call. Alternatively, the PVM_BARRIER and BARSYNC routines can be used to synchronize all tasks or a sub-group. Although other barrier methods exist, this one was chosen as it provides good performance and doesn't impact batch processing negatively.

2 Primary Experiences

To date we have ported several codes utilizing the prototype software. One code came from the T3D and was implemented with the SHMEM library. Two other codes had been running on a network of workstations and were based on network-PVM. The largest code was over 500,000 lines of Fortran. Compatibility issues encountered can be categorized into the following areas: private datatype specification; task initiation; integer size; and private I/O. Below we explain how we dealt with these areas in the codes.

2.1 Private Datatype Specification

The first issue that must be dealt with is to ensure that all data is private. On the CRAY T3D and on networks of workstations, all data is private unless explicitly defined otherwise. On the PVP shared-memory machines, all data that is global or static is shared among tasks. For FORTRAN codes, this means data that is in COMMON or that is used in DATA or SAVE statements. Data in C that is global includes data declared outside the function level and data that is statically declared. These shared data objects may be converted to task-private via a TASKCOMMON compiler directive. This can be done with the '-a taskcommon' and '-h taskcommon' options on the F90 and C compilers, respectively. As no equivalent option exists for the CF77 compiler, each COMMON must be declared as TASKCOMMON. Each of the codes ported so contained a mix of Fortran 77 and C source. The conversion to TASKCOMMON of the Fortran 77 code was accomplished automatically using SED scripts.

An additional consideration when using the SHMEM library is that all data to be transferred must be "symmetric", i.e., all data used as the target or source when calling the get and put routines must be in TASKCOMMON. This is required in order for the library to know where data on the other task's stack is placed.

2.2 Task Initiation

Changes needed to initiate tasks should be relatively minor, provided the master task is generated by the same executable used for the slave tasks. The MPT provides some extenstions that users may find handy to simplify the startup process. For example, the "START_PES(NPES)" routine starts up tasks simply, and the "SHMEM_N_PES()" and "SHMEM_MY_PE()" funtions return the number of PEs (npes) and the PE number in the range of 0:npes-1, respectively. Alternatively, the functions "MY_PE()" and "NUM_PES()" can be used to return the PE number and number of PEs, respectively. When "START_PES()" is used, command line options passed to the executable will exist for all tasks, so there is no need for the master task to read and broadcast the values. The same input file may be read by different tasks so that that data need not necessarily be read by one process and then broadcast to the other processes.

2.3 Integer Size

A PVM compatibilty issue that must be dealt with is integer size. Standard network PVM only implements 4-byte integers, whereas CRAY machines are based on 8-byte integers. It should be easy for the user to supply a variable or to utilize macros to ensure that the correct datatype is passed to the PVM pack/unpack routines. However, many users find the method of sending a PVM message (init, pack and send) combersome and have written a single routine to perform these three functions. They pass a pointer to the data through this routine and rely on all data to be packed as integer. This is not a healthy practice in general and will not work on CRAY machines. Better methods are to use pvm_psend/precv, which passes data as bytes, or to pass an argument indicating the variable type to be used in packing the data. Another option is to pass all data as blocks of bytes. A disadvantage of the byte-based methods is that it still requires the programmer to be aware of the relative datatype sizes on the machine used.

2.4 I/O

Multitasked codes on CRI systems, by default, use shared I/O. This is exactly opposite to the situation that exists on clustered machines, where I/O is all private and a single file cannot be shared. I/O done through C can circumvent the issue by using unique file descriptors for each task, but for existing Fortran codes, the workaround may involve significant source code modification. To address this issue, the Programming Environment (PE 2.0) introduces the capability of specifying files to be private to each task.

The user can assign the "private" attribute via the assign command or system call. This approach has the advantage of allowing the application to specify global or private i/o on a per file basis, whichever is more convenient for the given application.

3 Performance Notes and Results

Scalability results for the three codes are shown in Tables 1, 2 and 3. The first example from an academic code called FLO67. This code is primarily used for benchmarking purposes. A version of this code that was running on the CRAY-T3D using the SHMEM library was ported to the C90 using the MPT prototype software. All communication was done on the C90 with the SHMEM routines. The code employs a multigrid method and the smallest mesh used determines the amount of parallelism in the code. For this size example, this was the limiting factor to the scalability. The speedups of 1.8 on 2 processors, 2.9 on 4 processors and 4 on 8 processors are quite good and similar to the highly optimized autotasking implementation of the code.

The second table shows the performance of a fluids code. The original code was based on PVM and only scaled up to 8 processors with a disappointing speedup of 4.4 on 8 processors. The communication intensive portions of the code were changed to use SHMEM routines, and some synchronization points were eliminated. This caused the speedups to improve to 2.0 on two processors, 4.0 on 4 processors, 7.3 on 8 processors and 11.0 on 16 processors. Work on single node performance and further revisions in the parallel version resulted in improved speedups of 7.7 on 8 processors and 12.1 on 16 processors.

The third table shows data from a finite element structures code. Work on this code is currently on-going. This data is from the initial port of the code, i.e., no optimizations were made yet. If we ignore the time spent during the data input and problem setup phase, it can be seen that the scalability is quite good. Further work is planned to run the code on more processors and with larger examples.

A few things became clear during the work that went into porting these codes. In general, the SHMEM routines provide the highest performance at the cost of reduced portability to non-CRAY machines. This performance difference is most apparent for short messages. The differences are due to the fact that the SHMEM routines do one memory copy whereas the PVM routines perform two memory copies (to and from the buffer) in addition to the overhead of the extra PVM protocol.

Secondly, the amount of time spent in the barrier routine has a great effect on overall performance. It is preferable to use a barrier on the whole set of tasks rather than a subset. The PVM_BARRIER works more efficiently when the Cray-extension "PVMALL" groupname is used, as this allows the function to skip the analysis of which tasks are in the named group.

The applications developer may need to decide whether to have a master task read in some data and broadcast it to the other tasks or to have each task read in the data from it's own private file. Although the latter method is easier to program, in general the cost of each processor doing it's own I/O increases as the number of PEs increases. Typically, it does not matter which method is used for small numbers of tasks, but as the number of tasks increases, the more efficient method is to have one task read the data and broadcast it to the other tasks.

A significant advantage of the CRAY PVP systems is that multitasked codes can be run during batch relatively easily. Some systems have difficulty running multiprocessed jobs on an active shared system, as any memory swapping or time-slicing causes problems in recognizing deadlock and/or seriously degrades overall system performance. This causes some organizations to throttle usage on their parallel computer such that only one process runs on a CPU at a time. The multitasking software that this new Toolkit is built on is a mature, reliable product and many of these system issues were addressed some time ago. In the course of this work, we have seen excellant speedups for 2 and 4 processors during a heavily loaded J90 system on a regular basis.

4 Summary

We have successfully demonstrated the usefulness of the prototype MPT software on several real-world applications. High performance is achieved through fast data transfers and synchronizations. Portability is achieved across the CRAY platforms and only minor changes are necessary to enable codes to run on non-CRAY architectures.

The adoption of a distributed-memory programming style does not prevent us from taking advantage of some of the features of the shared memory and bonafide parallel operating system of these machines. The shared filesytem provides the developer with the choice of using shared or private I/O, while still being able to take advantage of the fast I/O subsystems of the CRAY machines. The convenience of having one executable is provided, as is the convenience of allowing the end-user the capability of choosing the number of tasks at runtime. The applications developer will appreciate the automatic deadlock detection feature, the hardware-performance monitor and the full set of performance software tools available. The computer system can be fully utilized as these codes run well in a batch environment and do not degrade system throughput.

Table 1. FLO67: Multigrid Fluids Example.

FLO67: 96x16x24 mesh with three levels, 10 cycles

	1 CPU	**2CPU**	**4CPU**	**8CPU**
C90 Autotasking	1.0	1.75	2.75	3.84
C90 MPT SHEM Proto	1.0	1.79	2.91	4.14
	4 PE	8 PE	16PE	
T3D W/ SHEM	1.0	1.9	3.03	

Table 2. Fluids Code 157K Cell Example.

Performance Relative To 1 CPU

	1 CPU	**2 CPU**	**4 CPU**	**8 CPU**	**16 CPU**
J90 MPT-PVM					
Original Code	1.0	2.0	3.6	4.4	2.7
Improve Communications	1.0	2.1	4.0	7.3	11.0
More Optimizations	1.0	n.a.	n.a.	7.7	12.1
C90 Autotasking	1.0	1.5	1.9	n.a.	n.a.

Table 3. 4000 Finite Element Sturctures Example.

J90 Speedups

	1 CPU	**2 CPU**	**4 CPU**
Overall	1.0	1.95	2.96
One Statuc Increment	1.0	2.21	3.96