Chapter 2.
Building MPI Applications

This chapter provides procedures for building MPI applications on UNICOS, UNICOS/mk, and IRIX systems.

2.1 Building Applications on UNICOS Systems

On UNICOS systems, the MPI library supports either a shared memory or a Transmission Control Protocol (TCP) communications model.

The following section provides information about the shared memory model, and information about compiling and linking shared memory MPI programs.

Note: Software included in the 1.2 release of the Message Passing Toolkit is designed to be used with the Cray Programming Environment. When building an application that uses the shared memory version of MPI, you must be using the Programming Environment 3.0 release or later. Before you can access the Programming Environment, the `PrgEnv` module must be loaded. For more information on using modules, see *Installing Programming Environment Products*, publication SG−5191, or, if the Programming Environment has already been installed on your system, see the online ASCII file `/opt/ctl/doc/README`.

Section 2.1.3, provides information that applies to MPI programs that use TCP for communication.

2.1.1 Building Applications That Use Shared Memory MPI on UNICOS Systems

To build an executable file that makes use of shared memory, perform the following steps:

Procedure 2

1. Convert all global and static data to `TASKCOMMON` data.

In a multitasking environment, all members of the multitasking group can access all global or static data because they share one user address space. An MPI process is described as a separate address space. To emulate an MPI process in a multitasking environment, all global or static data that can be modified during the course of execution of a program must be treated as data local to each task. This is done by placing the data in `TASKCOMMON` blocks. `TASKCOMMON` storage is a mechanism that is used in multitasked programs to provide a separate copy of data for each member of the multitasking group. `TASKCOMMON` data is still globally accessible across functions within a multitasked program, but it is private to each specific task. Fortran examples of global or static data that must be placed in `TASKCOMMON` include data that resides in `COMMON` blocks and data that appears in `DATA` or `SAVE` statements. In C, you must place all data that is declared static (either locally or globally) or data declared at a global level (outside of any function) in `TASKCOMMON`.

Because changing your program so that all global and static data is private is tedious and makes a program less portable, Cray Research provides support in the form of compile−time command line options to do the conversions. You can convert most global and static data to `TASKCOMMON` data automatically by using the following command−line options:

- For C programs:

```
cc −h taskprivate
```

- For Fortran programs:

```
f90 −a taskcommon
```

When you are placing data in TASKCOMMON, there may be cases in which the compiler cannot do the conversion because of insufficient information. The compiler notes these cases by issuing a warning during compilation. For such cases, you must convert the data by hand. Most of the time, these cases are related to initialization that involves Fortran DATA or SAVE statements or C initialized static variables, and you might need to change only how or when the data is initialized for it to be successfully placed in TASKCOMMON.

The following is an example of a case that the compiler cannot handle:

```
int a;

int b = &a
```

If variable a resides in TASKCOMMON, its address will not be known until run time; therefore, the compiling system cannot initialize it. In this case, the initialization must be handled within the user program.

2. Use the cc(1) or f90(1) commands to build your shared memory MPI program, as in the following examples:

C programs:

```
cc -htaskprivate -D_MULTIP_ -L$MPTDIR/lib/multi file.c
```

For C programs, the -D and -L options are needed to access the reentrant version of libc. This version is required to provide safe access to libc routines in a multitasking environment. When the mpt module is loaded, the module software sets $MPTDIR automatically and points to the default MPT software library. (For information on using modules, see *Installing Programming Environment Products*, publication SG−5191.) To make compiling in C easier, the environment variable $LIBCM is also set automatically when the mpt module is loaded. You can use $LIBCM with the cc(1) command to request the reentrant version of libc. $LIBCM is set to the following value:

```
-D_MULTIP_ -L$MPTDIR/lib/multi
```

The following example uses $LIBCM:

```
cc -htaskprivate $LIBCM file.c
```

Fortran programs:

```
f90 -ataskcommon file.f
```

3. Select private I/O if private Fortran file unit numbers are desired.

In a multitasking environment, Fortran unit numbers are, by default, shared by all members of the multitasking group. This behavior forces all files to be shared among MPI processes. The user can request that files be private to each MPI process by specifying the private I/O option on the assign(1) command. The examples in Table 1, request private I/O.

Table 1. assign examples

Example	Description
assign -P private u:10	Specifies that unit 10 should be private to any MPI process opens it.
assign -P private p:%	Specifies that all named Fortran units should be private to a MPI process that opens them. This includes all units conne regular files and excludes units such as 5 and 6, which are connected to stdin, stdout, or stderr by default.
assign -P global u:0 assign -P global u:5 assign -P global u:6 assign -P global u:100 assign -P global u:101 assign -P global u:102	This set of assign commands can be used in conjunction assign -P private g:all to retain units connected by default to stdin, stdout, and stderr as global units. A u connected to these standard files cannot be a private unit.

For more information on private I/O functionality on Cray PVP systems, see the assign(1) man page.

4. Run the application. To start or run an application that uses the shared memory version of MPI, you must use the `-nt` option on the `mpirun(1)` command (for example, `mpirun -nt 4 compute`).

You should also consider using Autotasking instead of message passing whenever your application is run on a UNICOS system. The communications overhead for Autotasking is orders of magnitude less than that for sockets, even on the same system, so it might be better to have only one fully autotasked MPI process on the UNICOS system. In many cases, you might be able to achieve this simply by invoking the appropriate compiler options and sending a larger file of input data to the MPI process on the UNICOS system.

2.1.2 Shared Memory MPI Limitations

Emulating a shared memory environment with the use of Cray Research multitasking software might cause unexpected program behavior. The goal is to preserve the original behavior as much as possible. However, it is not efficient or productive to completely preserve the original MPI behavior in a multitasked environment. The intent is to document possible changes in behavior. For example, changes in behavior might occur with the use of signals; therefore, it is not recommended that signals be used with the shared memory version of MPI.

The shared memory implementation of MPI supports the running of only 32 MPI processes within a multitasking group. Because MPI processes must share CPU resources, running with more than the number of physical CPUs available on the UNICOS system will begin to degrade performance.

2.1.3 Building Files on UNICOS Systems When Using TCP

On UNICOS systems, after you have added the MPI function calls described in Procedure 1, Step 1, a simple MPI program can be linked as follows:

```
cc -o compute compute.o
```

This command links the `compute.o` object code to produce the `compute` executable file.

If you are using more than one host, the executable files should be installed on the Cray Research systems that you will be using. By default, MPI uses the path name of the user−initiated executable file on all systems. You can override this file by using a process group file.

In some installations, certain hosts are connected in multiple ways. For example, an Ethernet connection may be supplemented by a high−speed FDDI ring. Usually, alternate host names are used to identify the high−speed connection. You must put these alternate names in your machine file.

If your hosts are connected in multiple ways, you must not use `local` in your machine file to identify the local host, but must use the name of the local host instead. For example, if hosts `host1` and `host2` have Asynchronous Transfer Mode (ATM) connected to `host1-atm` and `host2-atm`, respectively, the correct machine file is as follows:

```
host1-atm

host2-atm
```

2.2 Building Applications on UNICOS/mk Systems

On UNICOS/mk systems, after you have added MPI function calls to your program, as described in Procedure 1, Step 1, you can compile and link an MPI program, as in the following examples:

```
cc -o compute compute.c
```

or

```
f90 -o compute -X4 compute.f
```

If you have loaded the `mpt` module, the directory that contains the MPI include files is automatically searched, and the MPI library is automatically loaded.

2.3 Building Applications on IRIX Systems

On IRIX systems, after you have added MPI function calls to your program, as described in Procedure 1, Step 1, you can compile and link the program, as in the following examples:

To use the 64−bit MPI library:

```
cc −64 compute.c −lmpi
```

To use the 32−bit MPI library:

```
cc −n32 compute.c −lmpi
```

If you are using modules and have loaded the `mpt` module file, the directory that contains the MPI include files is automatically searched and the MPI library is automatically loaded; therefore, you do not need to specify the `−lmpi` option, and you can compile as in the following example:

```
cc −64 compute.c
```