C/C++ Programming Environment on the Cray XT3 System

Geir Johansen, *Cray Inc.*

ABSTRACT: *A description of the issues involving the Cray C & C++ Programming Environment of the Cray XT3 system. The goal of this paper is to provide C and C++ programmers information to readily write and port codes to the Cray XT3 system. Discussion will include the difference between the Catamount system libraries and Linux system libraries, and the unique features of the PGI C/C++ compiler.*

KEYWORDS: Cray XT3, Cray Red Storm, C/C++ Programming

1.0 Introduction

The goal of this paper is for the user to gain an understanding of the unique features of the Cray XT3 system and programming environment to enable them to readily write and port C/C++ code to the Cray XT3. The paper is not meant as a comprehensive description of the Cray XT3 programming environment (see Cray Inc. document *Cray XT3 Programming Environment User's Guide S-2396*), rather it is intended to highlight the issues that C/C++ programmers have encountered.

All of the information presented in this paper is valid for the Cray Red Storm system. As a caveat, the software for the Cray XT3 is evolving at a rapid pace, so it is likely that some of the information in this paper will be out of date in the near future. Also, any possible future features discussed do not represent a commitment by Cray Inc. to implement these features.

2.0 Programming Environment

2.1 Extremely Brief Cray XT3 Architecture Description

The programming environment for Cray system is essentially a cross compiler environment. The compiler and linker are executed on Cray XT3 login service nodes that run the Linux operating system, while the resulting executables are invoked on compute nodes that run the Catamount microkernel. Other relevant information about the Cray XT3 that affect the programming environment include:

- Portland Group (PGI) compilers are the only supported compilers for the compute nodes.
- Catamount only supports static libraries (i.e. no dynamic libraries).
- x86-64 code.
- Not an SMP. Each PE has its own memory.
- Catamount has a subset of the standard glibc functionality.
- Application has dedicated use of the processor and memory on compute node.
- I/O performed by service nodes running Linux.

2.2 Modules

Similar to other Cray Inc. systems, the Cray XT3 uses the *modules* utility to initialize the programming environment for the user. The *modules* utility will set the appropriate environment variables so the compilers will find the correct header files and libraries to create an executable for the Cray XT3 compute nodes. The main module file is *PrgEnv*, which when loaded will load the other programming environment modules and system modules needed to build code for the Cray XT3. The following table is a list of the Cray XT3 module files:

Module and Package Name	Description
PrgEnv	Main programming environment module that loads all the programming environment modules, plus all system modules (*xt-libc*, *xt-catamount*, *xt-pbs*, …).
pgi	PGI compilers
xt-pe	Compiler drivers
xt-mpt	Cray MPICH2 Message Passing Interface 2 (MPI-2), SHMEM routines
acml	AMD Core Math Library
xt-libsci	Cray XT3 LibSci scientific library routines
gcc	Gnu C Library 3.2.3 routines

Table 1. Programming Environment *modules* files

2.2.1 Current Reality of Cray XT3 Modules

One of the main features of using the *modules* utility is the ability to change versions of software. For example, to change from using PGI 5.2-4 to PGI 6.0-1, the user would simply execute the command "*modules swap pgi/5.2.4 pgi/5.2.4*". At this point in time of the Cray XT3 life cycle, there are several dependencies that inhibit the loading of different versions of software. For example, Cray MPT 1.0 is currently built with PGI 5.2, so because of an incompatibility between PGI 5.2 and 6.0, it is not possible to use this version of MPI-2 libraries with PGI 6.0. Another example is that the user will want to use the same Catamount glibc routines that were released with Catamount microkernel.

2.3 Compiler Drivers

The compiler commands (see Table 2) are shell scripts that read in the environment variables that have been initialized by modules files and proceed to call the compiler executable with the appropriate arguments. Only the listed compiler commands should be used to compile code targeted for the compute nodes. Using another compiler shell script or calling the compiler directly will likely result in an important option being missed that is essential to execute on the compute nodes. For example, an '*mpicc*' executable does exist on the system, however, the '*cc*' command is the correct command to compile MPI C code.

Compiler Command	Compiler
cc	C compiler
CC	C++ compiler
ftn	Fortran compiler for Fortran 90 and Fortran 95
f77	Fortran 77 compiler

Table 2. Compiler commands

2.3.1 Compiling Code for Service Nodes

In order to compile code that is to be executed on the login node, such as a utility, then the compilers can be called directly. For example, to compiler a C code, the PGI C compiler *pgcc* or the GNU C compiler *gcc* can be invoked to compile the code. These compilers will find the appropriate header files and libraries in their normal Linux locations.

2.3 Libraries

The module files and compiler commands are used to ensure that appropriate libraries are linked for the executable to run on the compute nodes. The following table list the libraries that are included when using the C/C++ compiler commands:

Library Name	Description
libmpich.a	MPICH2 library
libacml.a	AMD Core Math Library
liblustre.a	Lustre file system I/O routines
libpgc.a	PGI C compiler library
libm.a	Catamount glibc math libraries
libcatamount.a	Catamount system routines
libsysio.a	File system I/O routines
libportals.a	Portals routines, low-level message passing interface
libc.a	Catamount glibc routines
libC.a	PGI C++ Standard Library, based on STLport
libgcc.a	GNU C library routines
libsma.a	SHMEM library (Non-default, need to specify *-lsma*)
libpapi.a	PAPI library (Non-default, need to specify *-lpapi*)
libgmalloc.a	glibc version of malloc (Non-default, need to specify *–lgmalloc)*.

Table 3 Libraries searched by C/C++ compiler drivers

2.4 Linking Linux Libraries

If a program requires linking a Linux library, such as *libjpeg.a*, the user will want to copy it from its normal Linux directory (*/usr/lib64*) to another directory to prevent other Linux system libraries, such as *libc.a*, being linked in the executable.

2.4 MPICH2 and C++ Incompatibility

A name conflict exists when a C++ program includes the *mpi.h* and *stdio.h* (Note: C++ header file *iostream* includes *stdio.h*) header files. Both header files define SEEK_SET, SEEK_CUR, and SEEK_END. This name conflict is not unique to the Cray XT3, but exists on other platforms that use MPI-2 libraries. The compiler will abort the compilation if this name conflict is detected. Assuming that the MPI naming is not needed, the code can be compiled by using the compiler option *-DMPICH_IGNORE_CXX_SEEK*, for example:

```
$ cat seek1a.C

#include <iostream>
#include <mpi.h>
// stdio.h version of SEEK_SET,
// SEEK_CUR, and SEEK_END are used

void t() { }

$ CC -c -DMPICH_IGNORE_CXX_SEEK seek1a.C
/opt/xt-pe/1.1.02/bin/snos64/CC: INFO:
catamount target is being used
$
```

Alternatively, the *mpi.h* header file can be included before *stdio.h* is included:

```
$ cat seek1b.C

#include <mpi.h>  //include MPI header
                  //before I/O header
#include <iostream>
// stdio.h version of SEEK_SET,
//SEEK_CUR, and SEEK_END are used

void t() { }

$ CC -c seek1b.C
/opt/xt-pe/1.1.02/bin/snos64/CC: INFO:
catamount target is being used
$
```

If the MPI definitions are needed, then the solution is to *#undef* these names prior to including *mpi.h*:

```
$ cat seek2.C

#include <iostream>
#undef SEEK_SET
#undef SEEK_CUR
#undef SEEK_END
#include <mpi.h>
// MPI version of SEEK_SET, SEEK_CUR,
// and SEEK_END are used

void t() { }

$ CC -c seek2.C
/opt/xt-pe/1.1.02/bin/snos64/CC: INFO:
catamount target is being used
```

$

Some vendors do not implement the C++ MPI seek definitions, so this is why the problem is not seen on some other systems. In a future release of Cray XT3 MPICH2 the MPI definitions will be turned off by default and the option *–DMPICH_ENABLE_CXX_SEEK* will be used to explicitly define them.

3. Catamount Microkernel Issues

The Catamount microkernel developed by Sandia National Laboratories provides support for application execution without the overhead of a full operating system. The following sections describe the issues involved when writing and porting code targeted for the Catamount microkernel.

3.1 Target Machine Macros

The following predefined macros can be used by *#ifdef* statements to provide information that the code being is targeted for the Cray XT3 compute nodes.

- ***__QK_USER__*** Code is targeted to run under the Catamount OS
- ***__LIBCATAMOUNT__*** Code uses Catamount libraries

3.2 Catamount glibc support

The Catamount microkernel only supports a subset of glibc functionality, some of the routines not supported include:

- Sockets, pipes, remote procedure calls, or other TCP/IP communication routines.
- Dynamic process control routines, such as *fork*, *exec*, and *system*.
- Share memory routines (*shm_open*).
- Dynamic library routines (*dlopen*).
- Pthreads.
- *getcwd* routines.
- Functions that require a database, for example *getuid* and related routines are not supported.
- Limited support for signal routines and *ioctl*.

Appendix A of the *Cray XT3 Programming Environment User's Guide (S-2396)* contains a full list of the *glibc* functions that are supported in Catamount.

The practical experience with porting codes to the Cray XT3 is that there have been a few codes that cannot be ported to the Cray XT3 because of the Catamount *glibc* limitation. For example, codes that rely on the use of pthreads or sockets cannot be ported. Most codes have required no or minor modifications to allow them to run the Cray XT3. The following sections describe some of the issues that have required work-arounds to the code.

3.2.1 malloc

The Catamount malloc routine is a customized version that has been optimized for the Catamount non-virtual memory operating system. It is tuned to work with applications that allocate large, contiguous data arrays. The heap_info routine is a Catamount routine that returns information about heap memory usage. Here is an example of the information provided by this routine:

```
$ cat mem_check.c
#include <stdio.h>
#include <catamount/catmalloc.h>

main ()
{
  size_t fragments;
  unsigned long total_free, largest_free,
total_used;
  if (heap_info(&fragments, &total_free,
      &largest_free, &total_used) == 0) {
   printf("heap_info fragments=%lu \
       \n total_free=%lu \
       \n largest_free=%lu \
       \n total_used =%lu\n",
       fragments, total_free, largest_free,
total_used);
  } else {
    printf("non zero return code from \
heap_info\n");
  }
  return;
}
$ cc -o  mem_check  mem_check.c
/opt/xt-pe/1.1.02/bin/snos64/cc: INFO:
catamount target is being used
mem_check.c:
geir@nid00004:/ufs/home/users/geir> yod -sz
1 ./mem_check
heap_info fragments=300
 total_free=918419968
 largest_free=918413840
 total_used =132560
$
```

The glibc version of malloc is available to users by specifying the *–lgmalloc* option on the compiler command line.

3.2.2 mmap

Catamount does not support the *mmap* function. Applications that use the *mmap* function with the MAP_ANONYMOUS flag to allocate memory space can instead use *malloc* to perform this function.

3.2.3 times

Catamount does not support the *times, _rtc,* and *clock* routines. The Catamount *dclock* routine is used to determine the elapsed time of a program segment. In addition to *dclock*, the functions *gettimeofday*, *getrusage*, *MPI_Wtime*, and Fortran *cpu_time* can be used to calculate elapsed time. All of these routines use the same clock, however, *dclock* will have the lowest calling overhead. Here is an example using *dclock*:

```
$ cat dclock.c
#include <catamount/dclock.h>

main()
{
    double start_time, end_time;
    start_time = dclock();
    sleep(3);
    end_time = dclock();
    printf("\nElapsed time =
%f\n",(end_time - start_time));
}
$ yod -sz 1 ./dclock

Elapsed time = 3.000008
$
```

gettimeofday example:

```
$ cat gettimeofday.c
#include <sys/time.h>

main()
{
    struct timeval tv;
    struct timezone tzp;
    double start_time, end_time;

    gettimeofday(&tv,&tzp);
    start_time = (double) tv.tv_sec
             + (double) tv.tv_usec * 1.e-6;
    sleep(3);
    gettimeofday(&tv,&tzp);
    end_time = (double) tv.tv_sec
             + (double) tv.tv_usec * 1.e-6;
    printf("\nElapsed time =
%f\n",(end_time - start_time));
}
$ yod -sz 1 ./gettimeofday

Elapsed time = 3.000009
$
```

For *getrusage*, user time and system time will be the same time. The compute node running the Catamount microkernel is dedicated for the users application. Adding the elapsed user and system time will simple result in the doubling of the actual elapsed time:

```
$ cat getrusage.c
#include <sys/time.h>
#include <sys/resource.h>

main()
{
  struct rusage ru;
  double u_start_time, u_end_time;
  double s_start_time, s_end_time;

  getrusage(RUSAGE_SELF,&ru);
  u_start_time = (double)ru.ru_utime.tv_sec
    + (double) ru.ru_utime.tv_usec * 1.e-6;
  s_start_time = (double)ru.ru_stime.tv_sec
    + (double) ru.ru_stime.tv_usec * 1.e-6;
  sleep(3);
  getrusage(RUSAGE_SELF,&ru);
  u_end_time = (double) ru.ru_utime.tv_sec
    + (double) ru.ru_utime.tv_usec * 1.e-6;
  s_end_time = (double) ru.ru_stime.tv_sec
    + (double) ru.ru_stime.tv_usec * 1.e-6;
  printf("\nElapsed time (user) = %f\n",
         (u_end_time - u_start_time));
  printf("\nElapsed time (system) = %f\n",
         (s_end_time - s_start_time));
}
$ yod -sz 1 ./getrusage

Elapsed time (user) = 3.000009

User Elapsed time (system) = 3.000009
$
```

3.2.4 system routine

The *system* routine performs a call to *fork* and *exec*, which are not supported by the Catamount microkernel. Often the call to execute a command can be replaced by a library routine. For example, *system("mkdir /dir")* can be replaced by a call to *mkdir("/dir", 0750)* . In other cases, users have written routines to replace the command being called. For example, *system("cp src dest")* could be replaced by a call to routine that copies one file to another.

3.2.5 getpid

While the *getpid* function is supported by Catamount it may not return information that is useful to the program. On Catamount, the *getpid* function returns an integer from 1 – 5. Different processes within the same parallel program can return the same *getpid* number. To get a unique value for each process, the *nid* value can be used. For example, in the program below a *getnid* function is written to return a unique value for each process.

```
$ cat getpid.c
#include <catamount/data.h>

unsigned getnid() {
    return((unsigned)_my_pnid);
}

main()
{
    printf("getpid=%d, getnid=%d\n",
            getpid(), getnid());
}
$ cc -o getpid getpid.c
/opt/xt-pe/1.1.02/bin/snos64/cc: INFO:
catamount target is being used
getpid.c:
$ yod -sz 8 ./getpid
```

```
getpid=2, getnid=82
getpid=5, getnid=196
getpid=3, getnid=199
getpid=3, getnid=200
getpid=2, getnid=201
getpid=4, getnid=197
getpid=2, getnid=204
getpid=2, getnid=202
$
```

3.2.6 getrlimit, setrlimit

An application running on a compute node will have dedicated use of the processor and memory, so *getrlimit* will show that many resources limits have a value of *RLIM_INFINITY*. For file I/O related limits, the Catamount does not having limitations, however, the specific file system on the Linux service partition may have limits that are unknown to the Catamount microkernel. The following code shows the limits being returned by *getrlimit* for each system resource:

```
$ cat rlimit.c

#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

main () {
  struct rlimit rl;
printf("RLIM_INFINITY=%d\n",RLIM_INFINITY);
  getrlimit(RLIMIT_CPU,&rl);
  printf("RLIMIT_CPU limit = %d\n",
rl.rlim_cur);
  getrlimit(RLIMIT_DATA,&rl);
  printf("RLIMIT_DATA limit = %d\n",
rl.rlim_cur);
  getrlimit(RLIMIT_FSIZE,&rl);
  printf("RLIMIT_FSIZE limit = %d\n",
rl.rlim_cur);
  getrlimit(RLIMIT_LOCKS,&rl);
  printf("RLIMIT_LOCK limit = %d\n",
rl.rlim_cur);
  getrlimit(RLIMIT_MEMLOCK,&rl);
  printf("RLIMIT_MEMLOCK limit = %d\n",
rl.rlim_cur);
  getrlimit(RLIMIT_NOFILE,&rl);
  printf("RLIMIT_NOFILE limit = %d\n",
rl.rlim_cur);
  getrlimit(RLIMIT_NPROC,&rl);
  printf("RLIMIT_NPROC limit = %d\n",
rl.rlim_cur);
  getrlimit(RLIMIT_RSS,&rl);
  printf("RLIMIT_RSS limit = %d\n",
rl.rlim_cur);
  getrlimit(RLIMIT_STACK,&rl);
  printf("RLIMIT_STACK limit = %d\n",
rl.rlim_cur);
}
$ cc -o rlimit rlimit.c
/opt/xt-pe/1.1.02/bin/snos64/cc: INFO:
catamount target is being used
rlimit.c:
$ yod -sz 1 ./rlimit
RLIM_INFINITY=-1
RLIMIT_CPU limit = -1
RLIMIT_DATA limit = 918659072
RLIMIT_FSIZE limit = -1
RLIMIT_LOCK limit = -1
RLIMIT_MEMLOCK limit = -1
RLIMIT_NOFILE limit = -1
RLIMIT_NPROC limit = 1
RLIMIT_RSS limit = -1
RLIMIT_STACK limit = 16777216
$
```

The *setrlimit* function will always return successfully when called with a valid resource name and valid pointer to an *rlimit* structure. This *rlimit* passed by *setrlimit* is ignored by Catamount.

3.2.7 Other Routines

Here are other examples of cases where missing Catamount libraries have affected the porting of code:

tmpnam The *mkstemp* routine can be used to create a temporary file with a unique name.

statfs Routine to get file system statistics is not supported.

***fcntl* (*F_SETLK*)** The *fcntl* routine is supported, however, it cannot be used to set file locks. The error *EINVAL* (Invalid argument) will be returned.

3.3 Catamount Standard I/O

Standard I/O (*stdin*, *stdout*, and *stderr*) on the compute nodes is unbuffered by default. As a result, the performance of read and writes to standard I/O is very poor. A user may dramatically improve performance by calling *setvbuf* to buffer *stdin*, *stdout*, or *stderr*. In the following example, the *stdout* buffer size is set to 1024, but larger buffer sizes can be used if needed.

```
$ cat setvbuf.c
#include <catamount/dclock.h>
#include <stdio.h>

main()
{
    int i;
    char *buf;
    double start_time, end_time;

    start_time = dclock();
    for(i=0;i<128;i++)
        printf("sixteen chars!!\n");
    fflush(stdout);
    end_time = dclock();
    fprintf(stderr,"   Time for unbuffer
I/O = %f\n",
            (end_time - start_time));
```

```
    buf = (char *)malloc(1024);
    setvbuf(stdout, buf, _IOFBF, 1024);
    start_time = dclock();
    for(i=0;i<128;i++)
        printf("sixteen chars!!\n");
    fflush(stdout);
    end_time = dclock();
    fprintf(stderr,"   Time for buffer I/O
= %f\n",
            (end_time - start_time));
}
$ yod -sz 1 ./setvbuf  >set.out
   Time for unbuffer I/O = 25.609129
   Time for buffer I/O = 0.200042
$
```

4. PGI C/C++ Compiler Issues

The Portland Group compilers are currently the only supported compilers for code to be executed on the Cray XT3 compute nodes. The PGI compilers do provide a good combination of features and performance for HPC programming. PGI has been very responsive in regards to the support of their compilers. The following sections describe the PGI specific issues that have occurred when compiling code for the Cray XT3.

4.1 PGI 5.2 & 6.0 Incompatibilities

The *PGI 6.0 Release Notes* indicate that object files created using the 6.0 compilers are incompatible with object files from previous releases. One reason for this is the C++ name mangling has changed in PGI C++ from previous releases.

4.2 C99 Standard

The Portland Group has indicated that full conformance to the C99 standard is a likely feature of the PGI 6.1 release. The following are examples of how not conforming to the C99 standard have affected compilation of code for the Cray XT3.

4.2.1 C++ style comments

The PGI C compiler by default does not interpret the C++-style comments ("//") in source code. The '*-B*' option can be specified on the compiler command line to allow the C compiler to understand that // designates comments in the code.

4.2.2 Variable Length Arrays

PGI 6.0 C compiler does not support variable length arrays (VLAs). For example, the following code will not compile:

```
void vla(int size) {
    char dummy[size];
      .
      .
```

The above code can be rewritten as:

```
void vla(int size) {
    char *dummy;
    dummy = (char *)malloc(size);
      .
      .
```

4.2.3 ISO C99 Library Routines

The PGI compiler will leave the macro *__USE_ISOC99* undefined, so the code in the header files that depends on this macro being set (i.e. *#ifdef __USE_ISOC99*) will not be included. The Catamount *glibc* and *libm* libraries contain some ISO C99 specific routines, and they are linked in even if they have not been prototyped in a header file. This is a problem for the PGI C++ compiler, as it requires all routines that are used to be prototyped. Here is an example where the routine *trunc* is located in the Catamount version of *libm.a*, but is not prototyped in the header file because *__USE_ISOC99* is not set. Manually prototyping the *trunc* routine works around the problem:

```
$ cat trunc.C

#include <cmath>
#ifdef __USE_ISOC99
  #error 1 //__USE_ISOC99 is unset
#endif
#include <iostream>

#ifdef TRUNC
extern "C" double trunc(double);
#endif

int main() {
   double a = trunc(1.2);
   std::cout << a << std::endl;
   return 0;
}

$ CC -D__USE_ISOC99 -o trunc trunc.C
/opt/xt-pe/1.1.02/bin/snos64/CC: INFO:
catamount target is being used
trunc.C:
"trunc.C", line 13: error: identifier
"trunc" is undefined
     double a = trunc(1.2);
                ^

1 error detected in the compilation of
"trunc.C".
$ CC -DTRUNC -o trunc trunc.C
/opt/xt-pe/1.1.02/bin/snos64/CC: INFO:
catamount target is being used
trunc.C:
$ yod -sz 1 ./trunc
1
geir@nid00004:/ufs/home/users/geir>
```

4.3 Compiler Options

The PGI C/C++ compiler has many options to specify features and optimization techniques to be performed by the compiler. Chapter 2 of the *PGI User's Guide* provides a good overview of optimization options available for the PGI compilers. The following list is a sampling of the PGI compiler options that have been used in compiling the code for the Cray XT3:

- ***-fastsse*** This flag is a collection of optimization options that PGI suggests for targets with SSE/SSE2 capability. The specific optimization flags that are specified by the –fastsse are: "*-fast -Mvect=sse -Mscalarsse -Mcache_align –Mflushz*", where *–fast* specifies "*-O2 -Munroll=c:1 -Mnoframe –Mlre*"
- ***-Mnontemporal*** Informs compiler to force generation of nontemporal move and prefetch instructions.
- ***-Mprefetch=distance:8,nta*** *distance* option for the *prefetch* flag sets the fetch-ahead distance to 8 cache lines. The *nta* option instructs compiler to use the *prefetchnta* instruction.
- ***-Msafeptr*** Optimization option that instructs the compiler that pointers do not have data dependencies. Option is similar to the Cray C/C++ *restrict* option.
- ***-Mipa (=fast)*** Invokes interprocedural analysis. The fast option is collection of IPA sub-options that are generally optimal for the targeted machine.
- ***-Minline=levels:X*** Informs the inliner to perform X levels of inlining, where the default is 1. This is an important option for C++ code. The PGI User Guide suggests using *–Minline-levels:10* for C++ code.
- ***-Kieee*** Floating-point operations are performed in conformance with the IEEE 754 standard. This option is useful for producing bit identical results. In PGI 6.0, a performance penalty has been observed when using this option.
- ***-O3*** The *-fastsse* option contains *'-O2'*, so this option must appear after the *–fastsse* option on the command line. PGI informs us there is not significant difference between *–O2* and *–O3*.
- ***-Minfo*** Outputs messages of optimizations the compiler performed.
- ***-Mneginfo*** Outputs messages on why certain optimizations were not performed.
- ***-Mnodepchk*** The compiler assumes that potential data dependencies do not conflict. Option can produce incorrect code if there are data dependencies. Option is similar to the Cray *ivdep* compiler option.
- ***-help*** Displays useful information about the options specified on the command line**.**
- -***v*** Displays how the compiler, assembler, and linker were called.
- ***-tp k8-64, -tp amd64*** Specifies that you are targeting code to run on a AMD64 processor 64-bit mode. This option is not necessary when compiling on a Cray XT3 system

4.4 C++ Template Instantiaion

The single largest problem area involving the PGI C/C++ compiler has been with template instantiation. The PGI 5.2 compiler used a prelinker process to instantiate templates for the C++ program. The prelinker instantiation method was problematic for the following reasons:

- The process for building libraries required that all the object files be prelinked before they are added to a library. Many software makefiles assume that g++-style instantiation is available, so makefiles needed to be altered in order to use PGI 5.2 method of building libraries containing C++ code.
- The PGI compiler did not allow the use of the *–g* option to be used when building the object files for the library. This prevented the code from being examined by a debugger.
- The prelinking process requires that some source files to be recompiled in order to instantiate templates. This process adds to the overall build time of an executable.
- The build process is not robust in that it requires additional supporting files (i.e. *.ii, *.ti) to be maintained. Often undefined linking errors have been resolved by removing all object files and supporting instantiation information, then rebuilding from the entire source.

In PGI 6.0 the C++ compiler now uses a gnu-like style of template instantiation. A template is now instantiated each time it is referenced and placed in the object file. Archives and plain objects will contain multiple copies of templates, which will be discarded by the gnu linker. No special compiler or linker flags are required for this template instantiation method.

4.5 profile

Code generated using the PGI profile options *(-Mprof*) does not execute successfully on the compute nodes. Problem is likely a Catamount porting issue of the PGI profile library being linked in the code.

5. Future Opportunities

5.1 Large Memory Support

Currently the system software limits applications to 1GB of memory per compute nodes, but this restriction will be removed in the near future. In order for the executable to use data sections that are greater than 2GB per node, the code will need to be compiled and linked with the PGI *-mcmodel=medium* option. This option requires that the static libraries being linked in must also be compiled with the *-mcmodel=medium* option. The programming environment will need to provide libraries compiled with this option.

5.2 Cross Compiler Environment

A request from Cray Inc. internal users has been for the implementation of a cross compiler environment to compile and build code. A main benefit of the cross compiler environment is to allow users to develop application code while not having access to the Cray XT3. At this time the Cray XT3 software is being updated frequently, so a version of cross compiler environment becomes obsolete quickly. As the Cray XT3 software stack becomes more mature with less updates, this would be a helpful feature.

5.3 Support of Additional Compilers

Currently the PGI compilers are the only officially supported compilers for applications running on the Cray XT3. A possible future enhancement is to support other compilers, such as the Gnu compilers or the Pathscale compilers.

While not supported, it is possible to create code for the compute nodes using other compilers. Compiling the Streams benchmark using the *gcc* compiler, and then using the *cc* command to link the executable resulted in an executable that could run on a compute node. The results from this executable were:

```
---------------------------------------------
Function Rate (MB/s) RMS time Min time Max time
Copy:    1807.5439    0.0886   0.0885   0.0888
Scale:   1866.3021    0.0858   0.0857   0.0858
Add:     2169.6453    0.1107   0.1106   0.1107
Triad:   2176.1226    0.1103   0.1103   0.1103
---------------------------------------------
```

The same software compiled with the PGI 6.0 compiler yielded the following results:

```
---------------------------------------------
Function Rate (MB/s) RMS time Min time Max time
Copy:    4148.8482   0.0386    0.0386   0.0388
Scale:   3542.561    0.0452    0.0452   0.0454
Add:     3870.0874   0.0620    0.0620   0.0620
Triad:   3845.9712   0.0624    0.0624   0.0624
---------------------------------------------
```

5.4 MPP Applications Running on Linux Kernel

A possible future enhancement to the Cray XT3 would be to extend parallel programming to nodes running the Linux kernel. This feature would allow certain codes to execute that were otherwise affected by the limitations of the Catamount microkernel.

Conclusion

Given that the Cray XT3 is still very early in its product lifecycle, the C/C++ programming environment has performed very well in enabling users to generate code for the Cray XT3. The Catamount microkernel glibc limitations have not been a major obstacle for porting many important codes to the Cray XT3. The PGI C/C++ compilers have also performed well, with one exception being template instantiation difficulties with PGI 5.2, which has been addressed in PGI 6.0

About the Author

Geir Johansen works in Software Product Support, Cray Inc. He is responsible for support of C, C++, libc, MPI, TotalView and other debuggers, and performance tools for the Cray X1 and Cray XT3 platforms. He can be reached at Cray Inc., 1340 Mendota Heights Road, Mendota Heights, MN 55120, USA; Email: geir@cray.com