Using Pthreads in Fortran

By V.Ganesh, Research Student, University of Pune, INDIA

This article describes the way to use pthreads library in Fortran programs. With most of the modern day processors having more and more built in capability of parallelism, there is ample need of utilizing this power at the application level, especially in scientific applications which involves lots of number crunching and multi GB disk handling. To utilize the power of multi threading for scientific applications, where FORTRAN still seems to be the de-facto choice, many compiler vendors like Intel have started supporting Open MP based parallelism. Though Open MP is a very good way of introducing parallelism in existing sequential FORTRAN programs, it only provides you with a mostly declarative style of programming where all the parallelization is handled at the compiler level. If you need to have explicit programming control by writing code (like handling disk based quantum chemical calculations, where explicit buffering is extremely useful), there is a need to use threading libraries like the pthread library. I will stick to using pthread in this article, but similar techniques can be used with any other threading library. I have also made the choice of using pthreads because it is widely available and recognized by POSIX standards. For detailed discussion on pthreads and programming with it refer the book from O'Reilly[1].

Any basic threaded application requires the following functionality from any underlying threading APIs:

1. Ability to create a thread
2. Assign a task to a thread, say a subroutine is executed in a thread
3. Mechanism for synchronization and locking
4. Wait for/ or Yield to execution of a thread

Taking these basic requirements into mind, and the requirements of my application, I have exposed only a few of the pthreads APIs (see Listing 1). Before I start giving details about my implementation, it is also important to know what compiler you use, because the name decorations

1 Pthread programming, O'Reilly publications

are different in different compilers. I have used the non-commercial version of the Intel FORTRAN compiler and the standard GCC compiler.

The first section of `#define`s are introduced to gracefully handle porting to any compiler. All the functions are named with only six letters so that even older FORTAN compilers may be used. The name decoration used in case of Intel FORTAN compiler consists of an underscore after the function name, with the name as lower case letters. Hence to avoid confusion all the C function names are declared in upper case. If you want the same to be used with g77 (GNU FORTAN compiler), then you just need to add two underscores instead of one in the `#define` and so on.

The first function of interest is the `CRETRD()` function. The first argument to it is the function pointer, which will be invoked by the thread library. To obtain a function/subroutine pointer in FORTRAN just pass the function name. In certain cases it is required to declare the function before it is passed to `CRETRD()` function using `INTERFACE` keyword:

```
INTERFACE IOTRD
   SUBROUTINE IOTRD
   END
END INTERFACE
INTEGER IDCPU

CALL CRETRD(CPUTRD, IDCPU)
```

The integer variable `IDCPU` is used to track the thread ID. Its important to note one valid assumption here that `pthread_t` in C is `typedef` for `unsigned int`, and so can be safely stored in FORTRAN integer type in case of 32-bit applications.

A call to `SLFTRD()` from with in a thread returns its ID, while `YLDTRD()` is a simple wrapper over `pthread_yield()` which is used to relinquish control over to other ready threads, if any. The functions `JONTRD()` and `EXITRD()` are trivial wrappers over `pthread_join()` and `pthread_exit()` respectively.

The handling of synchronization mechanism with the help of mutexes in FORTRAN was not as trivial, but the code was definitely simple and elegant to write. To understand how function wrappers `GETMTX()`, `RELMTX()`, `LOKTRD()` and `ULKTRD()` work we must first understand how the function or subroutine arguments are passed in C and FORTAN.

The function that creates and initializes mutex in pthreads library is `pthread_mutex_init()`. This function is required to pass a pointer to a `typedef struct pthread_mutex_t`. Though structures are supported in FORTRAN, to avoid complexity its best to avoid them in FORTRAN source in some way, the best way being using pointers. While in FORTAN every thing is always passed by reference, in C every thing is passed by value, even pointers are passed by value in C. Thus every function which accepts an argument is declared to accept a pointer argument, as this is the convention in FORTRAN. The `GETMTX()` function is declared to accept an argument of pointer to pointer type, thus in FORTRAN this subroutine will be called as follows:

```
INTEGER LOCK1

CALL GETMTX(LOCK1)
```

Doing so stores value of pointer to mutex in the integer variable `LOCK1` and this variable can be used for any further handling of lock. For e.g. the following code illustrates a typical scenario for handling synchronized sections :

```
INTEGER LOCK1

CALL GETMTX(LOCK1)
CALL LOKTRD(LOCK1)
!....
! ... SOME CRITICAL SECTION GOES HERE...
!...
CALL ULKTRD(LOCK1)
CALL RELMTX(LOCK1)
```

Lastly, I have successfully used these wrapper functions to incorporate parallel i/o and cpu operations in disk based *ab initio* quantum chemical application, which has not been discussed here because of its inherent

complexity.

Conclusions:

Pthread libraries are one of the most powerful, cross-platform threading libraries available today at programmers disposal. Though there are some implementations of pthread for FORTRAN (for e.g. on AIX platform), they are not widely available. So the only way these could have been used as of today was to write wrappers over them, yet being simple and succinct enough to be used in a real application. I have tried to describe a very simple way to utilize this powerful threading library in FORTAN and hence effectively using it to extend the functionality of the programming paradigm. The code snippet that I have provided can easily be extended to other functionalities of pthread library as the need by the programmers using it.

Listing 1: The pthread interface to FORTRAN

```
/**
 * fthread.c
 *
 * pthread library interface to Fortran, for OSs supporting pthreads
 *
 * @author V.Ganesh
 */

#include <stdio.h>
#include <pthread.h>

/**
 * The following names are published to Fortran source
 */
#define CRETRD cretrd_
#define YLDTRD yldtrd_
#define SLFTRD slftrd_
#define JONTRD jontrd_
#define EXITRD exitrd_
#define LOKTRD loktrd_
#define ULKTRD ulktrd_
#define GETMTX getmtx_
#define RELMTX relmtx_

/**
 * CRETRD() - create a new fortran thread through a subroutine.
 *
 * @param thread_func: a function pointer, pointing to a subroutine
 * @param theThread: the thread ID
 */
void CRETRD(void *(*thread_func)(void *), pthread_t *theThread) {
     pthread_create(theThread, NULL, thread_func, NULL);
} /* end of function CRETRD() */

/**
 * YLDTRD() - yeild control to othre threads
 */
void YLDTRD() {
     pthread_yield();
} /* end of function YLDTRD() */

/**
 * SLFTRD() - get the thread ID
 */
pthread_t SLFTRD() {
     return pthread_self();
} /* end of function SLFTRD() */

/**
 * LOKTRD() - locks the execution of all threads till we have
 *            the mutex
 */
void LOKTRD(pthread_mutex_t **theMutex) {
     pthread_mutex_lock(*theMutex);
}

/**
 * ULKTRD() - unlocks the execution of all threads that were
 *            stopped due to this mutex
 */
void ULKTRD(pthread_mutex_t **theMutex) {
     pthread_mutex_unlock(*theMutex);
}

/**
 * GETMTX() - get a new mutex object
 */
void GETMTX(pthread_mutex_t **theMutex) {
     *theMutex = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
     pthread_mutex_init(*theMutex, NULL);
}
```

```
/**
 * RELMTX() - release a mutex object
 */
void RELMTX(pthread_mutex_t **theMutex) {
     pthread_mutex_destroy(*theMutex);
     free(*theMutex);
}

/**
 * JONTRD() - waits for thread ID to join
 */
void JONTRD(pthread_t *theThread) {
     int value = 0;

     pthread_join(*theThread, (void **)&value);
} /* end of function JONTRD() */

/**
 * EXITRD() - exit from a thread
 */
void EXITRD(void *status) {
     pthread_exit(status);
} /* end of function EXITRD() */
```