# USING A FORTRAN INTERFACE TO POSIX THREADS

Richard J. Hanson
*Rice University*
*Houston, Texas, USA*

Clay P. Breshears and Henry A. Gabb
*Kuck & Associates Inc., an Intel Company*
*Champaign, Illinois, USA*

**Abstract** Pthreads is the library of POSIX standard functions for concurrent, multithreaded programming. The POSIX standard only defines an application programming interface (API) to the C programming language, not to Fortran. Many scientific and engineering applications are written in Fortran. Also, many of these applications exhibit functional, or task-level, concurrency. They would benefit from multithreading, especially on symmetric multiprocessors (SMP). We summarize here an interface to that part of the Pthreads library that is compatible with standard Fortran. The contribution consists of two primary source files: a Fortran module and a collection of C wrappers to Pthreads functions. The Fortran module defines the data structures, interface and initialization routines used to manage threads. The stability and portability of the Fortran API to Pthreads is demonstrated using common mathematical computations on three different systems.

This paper is a shortened and slightly modified version of a complete Algorithm submitted for publication to the journal *ACM Trans. Math. Software*, during July, 2000.

**Keywords:** POSIX Threads, Fortran, scientific computing, symmetric multiprocessor, mathematical software, barrier routine

## 1. INTRODUCTION

Pthreads is a POSIX standard library [6] for expressing concurrency on single processor computers and symmetric multiprocessors (SMPs). Typical multithreaded applications include operating systems, database search and manipulation, and other transaction-based systems with

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35407-1_22
R. F. Boisvert et al. (eds.), *The Architecture of Scientific Software*

shared data. These programs are generally coded in C or C++. Hence the POSIX standard only defines a C interface to Pthreads. The lack of a Fortran interface has limited the use of Pthreads for scientific and numerically intensive applications. However, since many scientific computations contain opportunities for exploiting functional, or task-level concurrency, certain Fortran applications would benefit from multithreading.

A thread represents an instruction stream executing within a single address space; multiple threads, of the same process, share this address space. Threads are sometimes called 'lightweight' processes because they share many of the properties and attributes of full processes but require minimal system resources to maintain. When an operating system switches context between processes, the entire memory space of the executing process must be saved and the memory space of the process scheduled for execution must be restored. When switching context between threads there is no need to save and restore large portions of memory because the threads are executing within the same memory space. This savings of system resources is a major advantage of using threads.

The Pthreads library provides a means to control the spawning, execution, and termination of multiple threads within a single process. Concurrent tasks are mapped to threads. Threads within the same process have access to their own local, private memory but also share the memory space of the global process. Executing on SMPs, the system may execute threaded tasks in parallel.

As useful as the Pthreads standard is for concurrent programming, a Fortran interface is not defined. The POSIX 1003.9 (FORTRAN Language) committee was tasked with creating a FORTRAN (77) definition to the base POSIX 1003.1-1990 standard [3]. There is no evidence of any POSIX standard work to produce a FORTRAN equivalent to the Pthread standard. Fortran 90 has addressed many shortcomings of FORTRAN 77 that may have prevented the formulation of such a standard. There are no serious technical barriers to implementing a workable API in Fortran 90.

We review the implementation and testing of a Fortran API to Pthreads (referred to in what follows as **FPTHRD**). Our tests indicate that the API is standard-complying with Fortran 90 or Fortran 95 compilers. For this reason we use "Fortran" to mean compliance with both standards. The following section gives some general information on the threaded programming model with a specific example taken from the POSIX library functions. More complete descriptions of the POSIX thread library can be found in [2], [7], and [8].

In the complete paper [5] details of implementation of the Fortran API package to the Pthreads library are provided. Benchmarks are presented for threaded example problems and a comparison of their execution performance on three separate SMPs, each with a different native Pthreads implementation. These results show that thread programming has merit in terms of improving single processor performance on typical scientific computations.

## 2. THREADED PROGRAMMING CONCEPTS

Multithreading is a concurrent programming model. Threads may execute concurrently on a uniprocessor system. Parallel execution, however, requires multiple processors sharing the same memory; i.e., SMP platforms.

Threads perform concurrent execution at the task or function level. A single process composed of independent tasks may divide these computations into a set of concurrently executing threads. Each thread is an instruction stream, with its own stack, sharing the global memory space assigned to the process. Upon completion, the threads resources are recovered by the system. All POSIX threads executing within a process are peers. Thus, any thread may cancel any other thread; any thread may wait for the completion of any other thread; and any thread may block any other thread from executing protected code segments. There is no explicit parent-child relationship unless the programmer specifically implements such an association.

With separate threads executing within the same memory address space, there is the potential for memory access conflicts; i.e., write/write and read/write conflicts (also known as race conditions). Write/write conflicts arise when multiple threads attempt to concurrently write to the same memory location; read/write conflicts arise when one thread is reading a memory location while another thread is concurrently writing to that same memory location. Since scheduling of threads is largely nondeterministic, the order of thread operations may differ from one execution to the next. It is the responsibility of the programmer to recognize potential race conditions and control them. Fortunately, Pthreads provides a mechanism to control access to shared, modifiable data. Locks, in the form of mutual exclusion (mutex) variables, prevent threads from entering critical regions of the program while the lock is held by another thread. Threads attempting to acquire a lock (i.e., enter a protected code region) will wait if another thread is already in the protected region.

Threads acquire and release locks using function calls to the Pthreads library.

Pthreads provides an additional form of synchronization through condition variables. Threads may pause execution until they receive a signal from another thread that a particular condition has been met. Waiting and signaling is done using Pthreads function calls.

## 2.1. POSIX CONSIDERATIONS

The Pthreads header file for the C wrapper code (**summary.h**) contains system dependent definitions for data structures that are used with the Pthreads routines. These are typically C structures and are intended to be opaque to the programmer. Manipulation of and access to the contents of the structures should only be done through calls to the appropriate Pthreads functions. Since programmers do not need to deal with differences of structure definitions between platforms, this style enables codes to be portable. Standard names for error codes that can be returned from system calls are established by the POSIX standard. Integer valued constants are defined with these standard names within system header files. As with Pthreads structures, the actual value of any given error code constant may change from one operating system to the next. The intention is to keep the specific values given to each error code hidden from the programmer. Thus, the programmer need only compare a function's return value against the named constant to determine if a specific error condition has arisen.

# 3. DESIGN AND IMPLEMENTATION OF FORTRAN API

The Pthreads library is relatively small, consisting of only 61 routines that can loosely be classified into three categories: thread management, thread synchronization, and thread attributes control. Thread management functions deal with the creation, termination, and other manipulation of threads. The two methods available for guaranteeing the correct and synchronous execution of concurrent threads are mutex and condition variables. These constructs, and the functions to handle them, are used to ensure the integrity of data being shared by threads. The Pthreads standard defines attributes in order to control the execution characteristics of threads. Such attributes include detach state, stack address, stack size, scheduling policies, and execution priorities.

*Table 1* Example code: Use of static initializers

```
TYPE (fpthrd_mutex_t) any_mutex
...
any_mutex = FPTHRD_MUTEX_INITIALIZER
```

## 3.1. FORTRAN INTERFACE DETAILS

The **FPTHRD** package consists of a Fortran module and file of C routines. The module defines Fortran derived types, parameters, interfaces, and routines to allow Fortran programmers to use Pthread routines. The C functions provide the interface from Fortran subroutine calls and map parameters into the corresponding POSIX routines and function arguments. Dependencies on compiler and Pthreads versions are managed within the C functions.

The following sections describe some of the design decisions we faced and the similarities and differences between **FPTHRD** and the Pthreads standard.

**3.1.1 Naming Conventions.** The names of the **FPTHRD** routines are derived from the Pthreads root names; i.e., the string following the prefix **pthread_**. The string **fpthrd_** replaces this prefix. In this way, a call to the Pthreads function **pthread_create()** translates to a call to the Fortran subroutine **fpthrd_create()**. Our initial thoughts were to prefix the full POSIX names with the character **f**, which would yield the prefix string **fpthread_** before each root name. However, the Fortran standard [1] limits subroutine and variable names to 31 characters. The longest POSIX defined name is 32 characters in length. Since the **fpthrd_** prefix yields a net loss of one character over the POSIX prefix, we can guarantee that routine names in our package will have no more than 31 characters. All the Fortran routine names are therefore standard-compliant and all the Pthreads root names remain intact.

For consistency, all POSIX data types (Table 2 below) and defined constants (Table 2, [5]) prefixed with **pthread_** (**PTHREAD_**) are defined with the prefix **fpthrd_** (**FPTHRD_**) within the Fortran module. Besides those defined specifically for Pthreads types, other POSIX types are used as parameters to Pthreads functions. For these additional structures a corresponding definition is included within the module with the prefix character **f** added to the POSIX name.

*Table 2* Fortran derived types and Pthreads structures

| Fortran Derived Type Name | POSIX Structure Name |
|---|---|
| **TYPE (fpthrd_t)** | pthread_t |
| **TYPE (fpthrd_once_t)** | pthread_once_t |
| **TYPE (fpthrd_attr_t)** | pthread_attr_t |
| **TYPE (fpthrd_mutex_t)** | pthread_mutex_t |
| **TYPE (fpthrd_mutexattr_t)** | pthread_mutexattr_t |
| **TYPE (fpthrd_cond_t)** | pthread_cond_t |
| **TYPE (fpthrd_condattr_t)** | pthread_condattr_t |
| **TYPE (fsched_param)** | sched_param |
| **TYPE (ftimespec)** | timespec |
| **TYPE (fsize_t)** | size_t |

**3.1.2 Structure Initialization.** Besides the routines specifically designed for initialization, the Pthreads library includes predefined constants that can be used to initialize mutexes, condition variables, and 'once block' structures to their default values. Corresponding derived type constants for initialization have been defined and included in **FPTHRD**. The type and names of these initialization constants for a condition variable, mutex variable, and 'once block' variable are:

```
TYPE (fpthrd_cond_t) FPTHRD_COND_INITIALIZER

TYPE (fpthrd_mutex_t) FPTHRD_MUTEX_INITIALIZER

TYPE (fpthrd_once_t) FPTHRD_ONCE_INIT
```

To use these initialized data types with default attributes, assign the value in a program unit with the assignment operator , viz. Table 1.

**3.1.3 Parameters.** The Fortran API preserves the order of the arguments of the C functions and provides the C function value as the final argument. This style of using Fortran subroutines for corresponding C functions with the status argument appended to the parameter list is used in the Fortran API for both MPI [9] and PVM [4]. A return value of zero indicates that the routine call did not yield any exception; any nonzero return value indicates that an exception condition occurred during execution. Whether an exception condition is an error or can be ignored is determined in the context of the application. The POSIX standard defines names for specific conditions and requires fixed integer values

be attached to these error codes. The Fortran module defines integer constants with the same names as the POSIX standard for all potential error codes that might be returned from Pthreads functions. The values of these Fortran constants are the same as their POSIX counterparts on the target platform. The routines **fpthrd_self()** and **fpthrd_equal()** have no status argument since they do not return exception flags.

Fortran provides compile-time checking of argument type, number, kind, and rank using interface blocks. This is an advantage over the C programming language, which does not provide argument checking. Besides the compile-time checking, interface blocks also provide for argument overloading. This feature allows the use of **TYPE(C_NULL)** parameters where an optional `NULL` could be used in the underlying C functions. Fortran interface blocks also make it possible for the status parameter to be optional in Fortran routine calls. The module in our package provides interface blocks for the Fortran routines that call corresponding C functions with the exception of routine **fpthrd_create()**. Since the argument type for the threaded subroutine is chosen by the program author, it is necessary to exclude type checking for **fpthrd_create()**. The status parameter is not optional in calls to this routine.

**3.1.4 fpthrd_join() Parameters.** One special case should be mentioned with respect to parameters. The second parameter of **fpthrd_join()** is used to return an exit code from the **fpthrd_exit()** call of the thread being joined. The Pthreads library uses a C language `void**` type to allow the return of any defined data value or C structure.If no value is expected or needed by the joining thread, a `NULL` value may be used. Due to the difference in the way C and Fortran pointers are implemented (see 3.4 What's Not Included in the Package, for further discussion) and the desire to keep the programming of the interface as simple as possible, it was decided to restrict the type of this parameter to `INTEGER`.

This type restriction is repeated in the single parameter of the **fpthrd_exit()** routine which generates the value. Within scientific applications, it was thought that this exit value would be used mostly for returning a completion code to the joining thread. Special codes could be designed to signify the success or failure, and the cause of any failure, of the joined thread. Should more elaborate data structures be required to be passed from a thread to that thread which joins it, the integer value can be used as a unique index into a global array of results.

## 3.2. SUPPORT AND UTILITY ROUTINES

This section contains details on several routines that are not included in the Pthreads standard. These routines have been included in **FPTHRD** to provide the programmer the ability to give the runtime system a hint as to the number of active threads desired, to initialize the Fortran API routines and check parameter values and derived type sizes, and to manipulate POSIX defined data types required by certain Pthreads functions for which there is no Fortran compliant method generally available.

Many systems that support multithreading have an included function to inform the runtime system of the number of threads the system should execute concurrently. This seems to be particularly relevant for uniprocessor systems and is intended to allow finer control of system resources by the thread programmer. We have included the **fpthrd_setconcurrency()** and **fpthrd_getconcurrency()** subroutines in **FPTHRD** to give the programmer a chance to request the number of kernel entities that should be devoted to threads; i.e., the number of threads to be executing concurrently. If the target platform does not support this functionality, calls to these routines will return without altering anything.

An initial data exchange is required as a first program step before using other routines in **FPTHRD**. The routine **fpthrd_data_exchange()** is used as an initialization for the **FPTHRD** library. This routine is similar in functionality to the `MPI_INIT()` routine from MPI. The data exchange was found to be necessary because the parameters defined in Fortran or constants defined in C are not directly accessible in the alternate language. One such value of note is the parameter `NULL` passed from Fortran to C routines. This integer is used as a signal within the C wrapper code to substitute a `NULL` pointer for the corresponding function argument. The derived type **TYPE(C NULL)**, while available to programmers, is not meant for use except to define the special parameter value `NULL`.

The working space for the C structures of Pthreads data types are declared as Fortran derived types. Each of the definitions for derived types is an integer array with the **PRIVATE** attribute. Pthreads structures are opaque. The **PRIVATE** attribute prevents the Fortran program from inadvertently accessing the data in these structures. One other task performed by the **fpthrd_data_exchange()** routine is to ensure that the Fortran derived types are of sufficient size to hold the corresponding C structures.

*Table 3* Example code: Use of error codes

```
DO
  CALL fpthrd_create(tid, NULL, thread_routine, routine_arg, ierr)
  IF (ierr /= EAGAIN) EXIT
  CALL wait_some_random_time
END DO
```

Five additional routines are included to give the programmer the ability to manipulate those C structures used by Pthreads that are not a direct part of the Pthreads definition. The Fortran names defined in **FPTHRD** for these data types are **TYPE(fsize t)**, **TYPE(ftimespec)**, and **TYPE(fsched param)** (as shown in Table 2). For these data types there are routines to set and retrieve values from objects of each type.

## 3.3. ERROR CHECKING

The POSIX standard defines a set of error codes that may be returned from calls to Pthreads functions that signal when exceptional conditions have occurred. These exception codes are available from the routines in **FPTHRD** through the optional status parameter. Examination of the returned value of the status parameter allows codes to dynamically react to possibly fatal conditions that may arise during execution.

As an example, consider a code that requires the creation of a large number of threads. During execution, resources may be temporarily unavailable to create new threads. Rather than abort the entire computation at this exception, it would be prudent to pause the creation of new threads until resources become available, since this is deemed certain to happen. In the event that the **fpthrd_create()** status parameter return value be equal to the **EAGAIN** error constant, the spawning thread would wait for some amount of time before attempting to create another thread (Table 3). As long as the **EAGAIN** exception value is returned from **fpthrd_create()**, the spawning thread will continue to wait before attempting to create the new thread.

While each platform may have different values for **EAGAIN** and all other error constants, the initial data exchange routine accounts for these differences. All the programmer needs to do is use the symbolic name; e.g., **EAGAIN**. The possible error constants that may be returned from each routine in **FPTHRD** are detailed in the documentation for each Pthreads routine.

Since most routines in **FPTHRD** have several possible exception codes, rather than specifically check for each one, a method to print out what exception code was returned may be desired. This is especially true when debugging threaded applications. **FPTHRD** contains the routine, **ferr_abort()**, that provides the functionality described above as well as aborting further processing by all threads. A brief description of the **ferr_abort()** subroutine and its parameters is given below.

```
SUBROUTINE ferr_abort(sequence_number, status, text_string)
  INTEGER, INTENT(IN) :: sequence_number
  INTEGER, INTENT(IN) :: status
  CHARACTER, DIMENSION(*), INTENT(IN) :: text_string
```

The sequence number is an arbitrary identifying integer printed with the error message. The status argument is a variable holding the exception code value returned from a prior call to some routine in **FPTHRD**. If the status value is non-zero, a message containing the corresponding error constant is printed along with the text of the third argument. A Fortran `STOP 'Abort'` is also executed to terminate the computation. If the status value is zero, no action is taken by the **ferr_abort()** routine. Thus, it is safe (and very wise) to insert calls to **ferr_abort()** after calls to **FPTHRD** routines when fatal errors are possible. Where it is possible that non-fatal exceptions may be encountered, these should be dealt with directly by the application code.

## 3.4. WHAT'S NOT INCLUDED IN THE PACKAGE

The functionality of several routines included in the Pthreads library is outside the scope of Fortran. We describe these functions in this section and state our reasons for their exclusion from **FPTHRD**.

The functions **pthread_cleanup_push()** and **pthread_cleanup_pop()** allow the programmer to place and remove function calls into a stack structure. Should a thread be cancelled before the corresponding pop calls have been executed, the functions are removed from the stack and executed. In this way, threads are able to "cleanup" details such as freeing allocated memory or acquired mutex variables even if normal termination is thwarted.

While the functionality of these routines is desirable, they are typically implemented as macros defined in the **pthread.h** header file in order to ensure paired push and pop calls. Upon further examination, we have found undocumented system calls and data structures used within these macros. Since the targets for **FPTHRD** are scientific computation and numerical codes, it was concluded that such functionality might not be

as useful as other functions. With that in mind, it was decided the effort required to develop a simple, general algorithm to implement equivalent cleanup functions in Fortran outweighed the potential benefit.

The heart of the problem for the functions associated with thread-specific data: **pthread_getspecific()**, **pthread_key_create()**, **pthread_key_delete()**, and **pthread_setspecific()** is illustrated by **pthread_getspecific()**. This function returns a void C pointer to a data object associated with the calling thread. This allows local data pertinent to a user's threaded function to be available before the thread terminates. Fortran defines a pointer attribute for intrinsic and derived types (from [1]):

> "... a pointer is a descriptor with space to contain information about the type, type parameters, rank, extents, and location of a target. Thus a pointer to a scalar object of type real would be quite different from a pointer to an array of user-defined type. In fact each of these pointers is considered to occupy a different unspecified storage unit."

A C pointer is simply a memory address. As evidenced from the above passage, Fortran cannot access or manipulate memory addresses directly. In other words, the C and Fortran languages share the word 'pointer' but not the logical content. At this time, we can find no portable way to implement the thread-specific data functions without imposing obstructive constraints.

Functions **pthread_attr_getstackaddr()** and **pthread_attr_setstackaddr()** manipulate a thread's stack address. As stated previously, Fortran has no facility to directly manipulate memory addresses. Besides, implementation details, such as a stack, are not addressed in the Fortran standard. Thus, there is a danger that a Fortran program that calls these routines may not recognize setting of the stack address.

The **pthread_atfork()**, **pthread_kill()**, and **pthread_sigmask()** functions deal with the `fork()` function and inter-thread signalling. Since support for these features from Fortran programs within the run-time system is unknown and perhaps even unsupported, especially between different operating systems, these functions are not included in **FPTHRD**.

## 3.5. PACKAGE CONTENTS

The **FPTHRD** API sample implementation consists of a Fortran module, `fpthrd.f90`, and a file of C functions, `ptf90.c`, together with an include or header file, `summary.h`. We also have included four test/verification programs, timing programs for matrix-vector product and matrix transpose, and other documentation.

# 4. EXAMPLE OF A BARRIER ROUTINE FOR THREADS

A routine often needed in scientific applications is thread synchronization or the use of a barrier routine. This need arises since concurrency is often ubiquitous in a computation, but the results cannot be used until all the participating threads have completed their work.

The Pthreads library, and the provided **FPTHRD** API, operates at a lower level of functionality than a barrier function. In fact a typical barrier is shown below, constructed from **FPTHRD** routines. Note that the example uses a mutex or lock variable (`LOCK`), a condition variable (`COND`), a simple counter (`COUNTER`), and a logical switch (`CYCLE`). The value `NTHREADS` is the number of threads to synchronize, which is set before the routine is called. The routine also relies on an external dummy routine (`VOLATILE()`) that is intended to force a load from memory for the argument.

Here are remarks about the codes given in Table 4

- The **FPTHRD** variables `LOCK`, `COND`, and the integer variables `CYCLE`, `SWITCH`, and `NTHREADS`, are declared in a module that contains the routine **SYNC_ALL()**. The variables `COND` and `LOCK` must be initialized (not shown here); the value of `COUNTER` is initialized to `NTHREADS`; and the value of `CYCLE` is initialized to zero (also not shown).
- Each thread stores a local copy of `CYCLE` as `LOCAL_CYCLE`. (Line 5)
- If there is one thread, there is no need to synchronize further, and the routine exits immediately. (Lines 4 & 11)
- Each thread, as it acquires the lock, decrements the value of `COUNTER`. One *distinguished thread* will decrease `COUNTER` to the value of zero. (Lines 6-7)
- The distinguished thread signals the alternate threads that `COUNTER` now has the value zero. It prepares the routine for further calls (Lines 9 & 13) and broadcasts a synchronous signal to the alternate threads that "individually wakes them up" with the mutex locked. (Line 16)
- The distinguished thread changes the value of `CYCLE`, (Line 13). This step is critically important, as explained next, in addition to preparing the routine for later calls.
- When the signal is broadcast, an alternate thread may not have yet entered the routine **FPTHRD_COND_WAIT()**. Testing

*Table 4* Typical barrier routine

```
 1. SUBROUTINE SYNC_ALL()
 2.   INTEGER LOCAL_CYCLE
 3. !  START of basic barrier code:
 4.   IF(NTHREADS > 1) call fpthrd_mutex_lock(LOCK)
 5.   LOCAL_CYCLE=CYCLE
 6.   COUNTER=COUNTER-1
 7.   IF(COUNTER == 0) THEN
 8. ! Reset counter to number of threads.
 9.     COUNTER=NTHREADS
10. ! When there is only one thread, synchronizing is not required.
11.     IF(NTHREADS == 1) RETURN
12. ! Throw switch in alternate direction.
13.     CYCLE=1-CYCLE
14. ! These steps prepare the routine for another use.
15. ! The distinguished thread wakes up the waiting threads.
16.     call fpthrd_cond_broadcast(COND)
17.   END IF
18. ! Waiting threads wake up, or see if the switch has changed.
19. ! The use of this function to test the value of CYCLE prevents
20. ! compiler optimization from using the wrong value of CYCLE.
21. ! An alternate thread will change the value, and each thread must
22. ! fetch its current value every time the test is made.
23.   DO WHILE(VOLATILE(CYCLE) == LOCAL_CYCLE)
24.     call fpthrd_cond_wait(COND, LOCK)
25.   END DO
26.   call fpthrd_mutex_unlock(LOCK) !  END of basic barrier code.
27. END SUBROUTINE

28. FUNCTION VOLATILE(INT)
29. ! This external dummy function is intended to prevent code
30. ! optimization from removing a critical test. It does not do
31. ! anything except prevent compiler optimization errors.
32.   INTEGER VOLATILE
33.   INTEGER,INTENT(INOUT) :: INT
34.   VOLATILE=INT
35.   INT=VOLATILE
36. END FUNCTION
```

the value of CYCLE protects against this and avoids missing the signal. Another possibility is that alternate threads may spuriously wake up — returning from routine **FPTHRD_COND_WAIT()** — and find this signal is not intended for them. If the value of CYCLE has not changed they again call routine **FPTHRD_COND_WAIT()** , unlock the mutex, and continue waiting, (Lines 23-25). The use of an external dummy integer subprogram, **VOLATILE()**, is intended to guard against aggressive compiler optimization. For example, without the use of this function, the machine code could keep the value of CYCLE and LOCAL_CYCLE within integer registers. When the wakeup signal is broadcast, the test (Line 23) would always be satisfied with the register values and thus appear to be a spurious wakeup signal. This would cause the program to fail since another signal would not appear.

## Acknowledgments

This work was supported in part by a grant of from the DoD HPC Modernization Program, Contract DAHC94-96-C0002. The authors would like to thank Dr. Wayne Mastin, Academic PET Director, for facilitating the work.

## References

[1] Adams, J. C., Brainerd, W. S., Martin, J. T., Smith, B. T., and Wagener, J. L. 1997. *Fortran 95 Handbook.* The MIT Press, Cambridge, MA.

[2] Butenhof, D. R. 1997. *Programming with POSIX Threads.* Addison-Wesley, Reading, MA.

[3] Butenhof, D. R. 1999. Personal communication.

[4] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. 1994. *PVM: A Users' Guide and Tutorial for Network Parallel Computing.* The MIT Press, Cambridge, MA.

[5] Hanson, R. J., Breshears, C. P., Gabb, H. A. "A Fortran Interface to POSIX Threads," submitted to *ACM Trans. Math. Software*, during July, 2000.

[6] IEEE. IEEE Standards Press, 1996. 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition] Information Technology, Portable Operating System Interface (POSIX), Part 1: System Application: Program Interface (API) [C Language] (ANSI).

[7] Lewis, B. and Berg, D. J. 1998. *Multithreaded Programming with Pthreads.* Sun Microsystems Press, Mountain View, CA.

[8] Nichols, B., Buttlar, D., and Farrell, J. P. 1996. *Pthreads Programming.* O'Reilly and Associates, Sebastopol, CA.

[9] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. 1998. *MPI The Complete Reference: Volume 1, the MPI Core.* The MIT Press, Cambridge, MA.

## DISCUSSION

*Speaker: Richard Hanson*

**Brian T. Smith :** What were the comparisons with transpose and matrix multiply? Were the intrinsics threaded by the supplier or were they serial?

**Richard Hanson :** These are comparisons of "ordinary code" with the array intrinsics `TRANSPOSE` and `MATMAL`. One expects the vendors to do a good job on these operations. In fact, at this problem size, threading is more effective on all the platforms we have ported to.

**John Rice :** Where is the gain of efficiency made in the threads implementation of the transpose?

**Richard Hanson :** Quite probably each computational unit is using its cache and the system resources concurrently and efficiently.