A Pthreads Wrapper for Fortran 2003

OMAR AWILE and IVO F. SBALZARINI, ETH Zurich

With the advent of multicore processors, numerical and mathematical software relies on parallelism in order to benefit from hardware performance increases. We present the design and use of a Fortran 2003 wrapper for POSIX threads, called forthreads. Forthreads is complete in the sense that is provides native Fortran 2003 interfaces to all pthreads routines where possible. We demonstrate the use and efficiency of forthreads for SIMD parallelism and task parallelism. We present forthreads/MPI implementations that enable hybrid shared-/distributed-memory parallelism in Fortran 2003. Our benchmarks show that forthreads offers performance comparable to that of OpenMP, but better thread control and more freedom. We demonstrate the latter by presenting a multithreaded Fortran 2003 library for POSIX Internet sockets, enabling interactive numerical simulations with runtime control.

Categories and Subject Descriptors: D.1.3 [**Concurrent Programming**]: Parallel Programming; G.4 [**Mathematical Software**]: Parallel and Vector Implementations

General Terms: Algorithms, Languages, Performance, Standardization

Additional Key Words and Phrases: POSIX threads, pthreads, Fortran, Fortran 2003, scientific computing, shared-memory programming, mathematical software, parallel particle-mesh, PPM library

ACM Reference Format:
Awile, O. and Sbalzarini, I. F. 2014. A pthreads wrapper for Fortran 2003. *ACM Trans. Math. Softw.* 40, 3, Article 19 (April 2014), 15 pages.
DOI:http://dx.doi.org/10.1145/2558889

1. INTRODUCTION

In 2005 the semiconductor industry made a technological move that has significantly influenced computer science. It had become clear that traditional chip designs were not able to meet the ever-increasing demand for performance. In the previous two decades the miniaturization of transistors and the increase in processor clock speed were the two main drivers of progress in the hardware industry. Consequently, software was able to benefit from an increased performance when executed on newer hardware. However, physical limitations such as the heat dissipation and power consumption of chips, as well as design limitations such as an increasing gap between CPU and memory speeds, led chip manufacturers to rethink the chip design [Asanovic et al. 2009; Geer 2005].

Instead of further increasing the clock frequency, chips now comprise multiple cores. Newer designs have become more power efficient, cooler, and more performant. Since 2005, the number of cores has steadily increased.

I. F. Sbalzarini is now affiliated with MOSAIC Group, Center of Systems Biology Dresden (CSBD), Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany.
O. Awile was supported by grant no. 200021-132064 from the Swiss National Science Foundation, to I. F. Sbalzarini.
Authors' address: O. Awile and I. F. Sbalzarini (corresponding author), MOSAIC Group, Institute of Theoretical Computer Science, ETH Zurich, Switzerland; email: ivos@mpi-cbg.de.

DOI:http://dx.doi.org/10.1145/2558889

These changes have presented new challenges to algorithms and software engineering. In order to optimally exploit the new processors, algorithms and software have to be designed with parallelism in mind [Asanovic et al. 2009]. Most of the difficulties in programming shared-memory parallel systems stem from the fact that resources have to be shared between several concurrent execution threads. Some of the pioneering work in concurrent programming is decades old. This includes *mutual exclusions (mutex)* [Dijkstra 1965] that allow two processes (or threads) to access the same shared-memory location without interfering with each other. *Transactional memory* allows a pair of load/store instructions to be executed atomically. Transactional memory has been implemented both in hardware [Herlihy and Moss 1993] and software [Shavit and Touitou 1997], and it is used in IBM's Blue Gene\Q supercomputer design.

In order to further ease building and updating parallel software, frameworks and libraries providing the tools for shared-memory parallel programming have become crucial to the computer industry and the scientific community.

Scientific computing has particularly benefited on several levels from the advent of multicore processors. On the one hand, multicore processors (and even more so manycore processors) turn the personal computers and workstations of engineers and scientists into powerful parallel computing machines. This makes it desirable that programming such machines be accessible to nonexperts, such that the users of such workstations can optimally exploit their computing resources [Brodtkorb et al. 2010; Marowka 2007; Perez et al. 2008]. On the other hand, multicore chips have enabled the construction of petascale supercomputers, while the next-generation manycore CPUs will be at the heart of upcoming exascale supercomputers. These systems are heterogeneous in nature, featuring not only multicore CPUs, but recently also GPGPUs, APUs, and FPGAs [Tsoi and Luk 2010].

Especially for tightly coupled problems, it has become increasingly important to optimize implementations for multicore and manycore architectures. This has, for example, been discussed by Dubey et al. [2010], Speck et al. [2011], and Madduri et al. [2009]. Furthermore, Madduri et al. [2011], and provide an extensive discussion of the opportunities and challenges of multi- and manycore architectures for Particle-In-Cell (PIC) methods. They concluded that manycores and GPUs may both offer considerable speedups. However, the benefit and cost of employing GPUs and accelerators must be carefully assessed for the particular problem at hand. One important challenge of large heterogeneous platforms, which has also been pointed out by Rabenseifner et al. [2009], is load balancing. Dynamic load balancing may be expensive and incur a communication overhead in purely distributed-memory parallel implementations. By using hybrid shared-/distributed-memory programming, however, it is possible to alleviate this problem by saving some of the communication overhead and providing dynamic workitem scheduling constructs. One such example is OpenMP's `schedule=dynamic` clause.

A number of petascale simulations performed on heterogeneous multi and manycore systems have recently demonstrated the opportunities and challenges of these systems. Winkel et al. [2012] extended the PEPC library, a MPI Barnes-Hut tree code, using pthreads and made it scale up to almost 300,000 cores, Using 4,000 GPUs and 16,000 CPU cores, Shimokawabe et al. [2011] achieved a 1 PFlop/s simulation of metal alloy solidification. Finally, Bernaschi et al. [2011] performed a biofluidics simulation at nearly 1 PFlop/s of blood flow through the human coronary arteries at the resolution of single red blood cells.

A common approach to writing scalable software for heterogeneous hardware platforms is to combine a distributed-memory parallelization library, such as MPI, with a threading library, like POSIX threads (pthreads) or OpenMP. MPI is then used to parallelize the application on the level of networked hosts, while the thread library is used to parallelize within each MPI process. The processor cores can thus be used to execute multiple threads in parallel. This strategy has been successfully used, for example, by Winkel et al. [2012]. It has several advantages over executing one MPI process per core. First, running several threads per process instead of running several processes results in a smaller overall memory footprint. This is not only due to the overhead incurred by process management, but also due to increased memory requirements for data replications, such as halo layers for decomposed domains [Rabenseifner et al. 2009; Winkel et al. 2012]. The problem is further aggravated since the size and bandwidth of main memory are not scaling with the number of cores [Dubey et al. 2010]. Second, using shared-memory parallelism allows for improved dynamic memory access on NUMA architectures. Finally, using hybrid threads-MPI programming models allows the programmer to designate tasks such as inter-node communication [Song et al. 2009; Winkel et al. 2012], job management, and job monitoring to be executed within one thread and, depending on the workload, to be dedicated to one core. This is harder to achieve in a pure SPMD programming model.

Pthreads is a POSIX standard for threads that is implemented in all POSIX-compliant operating systems, ranging from BSD derivatives and Linux to MacOS X and Solaris. Microsoft Windows offers an implementation too, albeit not natively. The original standard was published in 1995, but a number of threads implementations predate POSIX threads [Powell et al. 1991; Stein and Shah Sunsoft 1992]. In fact, the POSIX threads standard was created in an effort to consolidate existing libraries under one common interface, allowing programmers to write threaded applications that are portable across many operating systems.

Pthreads provides an API for the C programming language, offering functions to manage threads, mutexes, condition variables, and thread-specific data, and allowing synchronization between threads using locks and barriers. This API has been ported (i.e., wrapped) to several other programming languages, giving a wide audience access to the threading programming model. Extensive support for Fortran 2003, however, has been lacking. Fortran has a long-standing history in scientific computing and high-performance computing. The Fortran 2003 standard includes many desirable extensions and also supports object-oriented programming. A large number of numerical libraries and applications have been created for Fortran and are actively maintained and used. BLAS, Lapack, FFTW, PETSc, PEPC, and the NAG Fortran library are examples of widely used libraries written in Fortran or providing a native Fortran interface. To date, few nonproprietary, albeit partial implementations of Fortran pthreads wrappers exist [Breshears et al. 1998; Hanson et al. 2002; Nagle 2013]. Furthermore, IBM provides a proprietary pthreads Fortran interface for its AIX platform. While it is possible to directly call C routines from Fortran, this is often difficult and sometimes impractical and not easily accessible to many programmers. The wrapper by Hanson et al. [2002] provides access to the basic pthreads functionality and describes an example application for multithreaded matrix-vector products. This wrapper, however, is incomplete, and support for barriers, spin-locks, and reader-writer locks is, for example, missing. Moreover, pthreads-internal data types are not treated as opaque objects, in principle violating the POSIX standard.

Here we present forthreads, a new Fortran 2003 threads library that aims at providing a complete wrapper for POSIX threads that respects the opacity of internal types without requiring knowledge about Fortran 2003's ISO C interoperability extensions. We achieve this by using Fortran 2003 language features to implement a clean Fortran/C interface. Our work hence extends the few existing Fortran pthreads wrappers [Hanson et al. 2002; Nagle 2013], providing additional thread synchronization and management interfaces, and increasing portability. We also provide three examples covering a wide range of use cases for mixed shared-/distributed-memory parallelism.

Our examples extend the Parallel Particle-Mesh (PPM) library, a middleware for distributed-memory parallel particle-mesh simulations [Awile et al. 2010; Sbalzarini et al. 2006] with shared-memory threads support. The PPM library provides an abstraction layer consisting of a set of high-level data types and operations for hybrid particle-mesh methods [Sbalzarini 2010]. This abstraction layer hides the low-level message passing between distributed-memory processes behind high-level, domain-specific parallel abstractions. PPM uses adaptive geometric domain decompositions and communication through halo layers [Sbalzarini et al. 2006] to enable transparent parallelization of particle-mesh simulations.

We extended PPM with SIMD shared-memory parallelism for particle-mesh interpolations, a mixed shared-/distributed-memory multigrid Poisson solver with computation-communication overlap, and a multithreaded socket server for interactive monitoring and runtime control of PPM-based simulations. We show that using forthreads in our examples does not incur a significant performance toll, while providing the programmer with a simple yet comprehensive Fortran 2003 interface to POSIX threads.

2. FEATURES AND LIMITATIONS OF THE FORTHREADS LIBRARY

The present implementation provides routines and derived types covering almost all POSIX threads [POSIX 2004] capabilities, including optional specifications implemented in Linux and Linux-specific extensions. The provided Fortran interfaces cover the following.

— *Thread creation, joining, cancellation and basic management* provide basic threading functionality, such as creating and initializing new threads, calling of initialization routines, determining thread IDs, and comparing threads.
— *Mutexes*, or mutual exclusions, provide multiple threads with exclusive access to shared resources. Forthreads exposes all functions provided by pthreads mutex handling.
— *Conditional variables* are used in conjunction with mutexes, allowing threads to atomically check the state of a condition.
— *Barriers* are synchronization points at which participating threads must wait until all their peers have called the wait function.
— *Spin-locks* provide a busy-wait type of locking for threads. A thread trying to acquire a spin-lock that is already locked by a peer checks in a loop for the availability of the lock and returns as soon as its peer has returned. Spin-locks are more expensive in terms of resources than conventional locks based on the process or kernel scheduler, but offer a superior reaction time.
— *Readers-writer locks*, also known as shared exclusive locks, allow multiple threads acting as readers to acquire the lock at the same time in order to read a shared resource, while only one thread acting as a writer is allowed to acquire the lock for the shared resource. Forthreads offers wrappers for all RW lock pthread functions.
— *Thread attribute objects* are provided by pthreads to allow reading and modifying miscellaneous options, such as scheduling policy, stack size, or scheduling priorities.

Our current implementation expands upon the one by Hanson et al. [2002], in particular by providing wrappers for barriers, spin-locks, and readers-writer locks. These constructs are useful additions to mutexes and conditional variables and offer the programmer a set of flexible tools for thread synchronization.

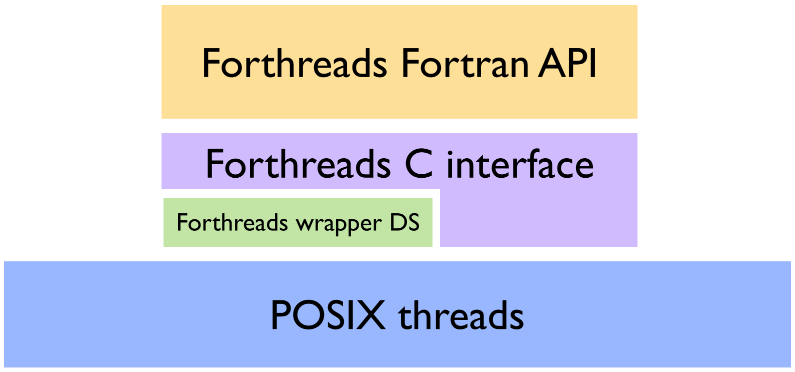


Fig. 1. The forthreads library consists of two parts. First, the pthreads functions and opaque types are wrapped with C code that exposes Fortran-friendly data types and function interfaces. Then, a set of Fortran routines provides the user with a native Fortran interface calling internally the forthreads C functions and passing all necessary pointers and data.

The only pthreads API functions that could not be wrapped in the present Fortran implementation are as follows.

— `pthread_cleanup_push` and `pthread_cleanup_pop`. These functions allow the programmer to register callback functions into a calling thread's cancellation cleanup stack that will be popped and executed in order whenever the thread exits, is cancelled, or calls `pthread_cleanup_pop` itself. These functions cannot be wrapped, as push and pop must be called in pairs in the same scope. Hence, the POSIX standard foresees their implementation to be done using C macros [POSIX 2004].
— Pthread thread-specific data management routines (`pthread_key_*` and `pthread_getspecific` / `pthread_setspecific`). These routines heavily rely on the C programming language's `void` pointers. Unfortunately, such pointers are not available in Fortran without exposing Fortran ISO C bindings in the library interface. It is a design choice of the present library that the user shall not require such knowledge. Therefore, it seems difficult to provide portable and safe wrappers to these functions.

Finally, in contrast to pthreads, the current implementation only allows `INTEGER` pointers to be passed to the thread-start routine. This is for the same reason as the aforementioned limitations on thread-specific data management routines.

The POSIX threads standard states that all pthread-specific types are opaque and that their specification should be treated as unknown to the user. Because of this limitation we chose to implement forthreads in two layers (Figure 1). In principle it would have been possible to recreate all pthread-specific types in Fortran as opaque types (using `PRIVATE` type members). This would, however, violate the POSIX threads standard by rendering forthreads OS dependent. C language functions and data structures are first used to manage and store all pthreads objects and expose only indexes, primitive type variables, and types defined in forthreads itself. A set of Fortran 2003-derived types and routines wrapping the forthreads C routines defines the actual forthreads API. The routines make heavy use of Fortran ISO C bindings introduced in Fortran 2003. They allow seamless interaction with the library without any knowledge of C/-Fortran interoperability.

To illustrate our approach we provide in the listings in Figures 2 and 3 the code required to wrap the `pthread_mutex_lock` function. This function locks the mutex with the given ID. If it is already locked by another thread, then the calling thread blocks until it can acquire the lock on the mutex. Figure 2 shows the forthreads C interface

```c
typedef struct array_tag {
  void **data;
  int size;
  int after;
  pthread_mutex_t mutex;
} array_t;

array_t *mutexes;

void thread_mutex_lock(int *mutex_id, int *info) {
  *info = FT_OK;
  if (!is_initialized) {
    *info = FT_EINIT;
    return;
  }
  if (!is_valid(mutexes,*mutex_id)) {
    *info = FT_EINVALID;
    return;
  }
  *info = pthread_mutex_lock((pthread_mutex_t*)
                  (mutexes->data[*mutex_id]));

}
```

Fig. 2. The forthreads C wrapper code for `pthread_mutex_lock`.

and the required data structures to wrap pthreads' opaque `pthread_mutex_t` identifiers. The user passes the previously obtained mutex identifier (`mutex_id`) to the function. Forthreads in turn passes to the `pthread_mutex_lock` the `pthread_mutex_t` object that had previously been stored in the `mutexes` array. Figure 3 shows the ISO C binding interface to Fortran 2003, and the implementation of the Fortran wrapping routine. It is not strictly necessary to use these Fortran routines as interface, but they free the user of dealing with the intricacies of Fortran-C interoperability.

3. USING FORTHREADS IN HYBRID MPI/PTHREAD PROGRAMS

Different design patterns exist for combining distributed- and shared-memory parallelism.

The *SIMD pattern* uses multiple threads to distribute a large number of identical (and preferably independent) workitems. Each thread executes the same subprogram on different data. This pattern is most prominently used in OpenMP, which employs preprocessor directives placed around sections of the code to be executed in parallel. The compiler then generates additional instructions to spawn and execute the threads. The same can also be achieved using pthreads (and hence forthreads). MPI is then used to parallelize the computation across multiple memory address spaces.

The *task parallelism* pattern assigns different tasks to different threads, executing possibly different code. A thread could, for example, be tasked with performing interprocess communication or message passing (e.g., using MPI) while other threads can run the program's main computations. Task-parallel threads can also be used to compute real-time in situ visualizations, or to allow user interaction of a running program. Pthreads and forthreads offer the full flexibility required for task-parallel applications through their various interfaces for thread management and synchronization. Also OpenMP has in its recent versions gained support for task-level parallelism through `task` constructs, which continue to be improved.

```
! ——— ciface.h ———

interface
  subroutine thread_mutex_lock(mutex_id,info) bind(c)
  use iso_c_binding

  integer(c_int), intent(in)  :: mutex_id
  integer(c_int), intent(out) :: info

  end subroutine thread_mutex_lock
end interface

! ——— forthread.f03 ———

subroutine forthread_mutex_lock(mutex_id,info)
implicit none

include 'ciface.h'

integer, intent(in)  :: mutex_id
integer, intent(out) :: info

call thread_mutex_lock(mutex_id,info)

end subroutine forthread_mutex_lock
```

Fig. 3. The forthreads Fortran 2003 wrappers built on top of the C interface shown in Figure 2.

In the *thread pool pattern* a number of worker threads are typically created that receive work tasks through a queue. A master thread manages the creation and destruction of worker threads based on the workload and interprocess communication. Such systems are particularly useful when the workload of each (MPI) process varies during the computation. OpenMP internally uses the thread pool pattern, but gives the programmer only limited freedom in adjusting the mode of operation through optional clauses to its preprocessor directives.

3.1. Extending the PPM Library with Shared-Memory Support

The PPM library [Awile et al. 2010; Sbalzarini et al. 2006] is a programming middleware for parallel hybrid particle-mesh methods. It provides abstract data structures and operations allowing the programmer to reason in terms of particles, meshes, and the associated compute operations [Sbalzarini 2010]. Data are distributed according to adaptive geometric domain decompositions (Figure 4), and subdomains are assigned to processors such that one processor is typically assigned multiple subdomains in order to provide sufficient granularity for load balancing. All data communication is internally handled by the library using MPI. Decomposition and communication are transparent, but made explicit to the programmer in order to expose the incurred communication overhead. Particles and meshes communicate at processor boundaries through halo (ghost) layers, which are kept up-to-date by user-called mapping operations. Numerical algorithms and solvers are applied locally per processor or per subdomain. Simulation codes are implemented as clients to the PPM library by calling functions from the PPM API. This has proven to reduce code development time while

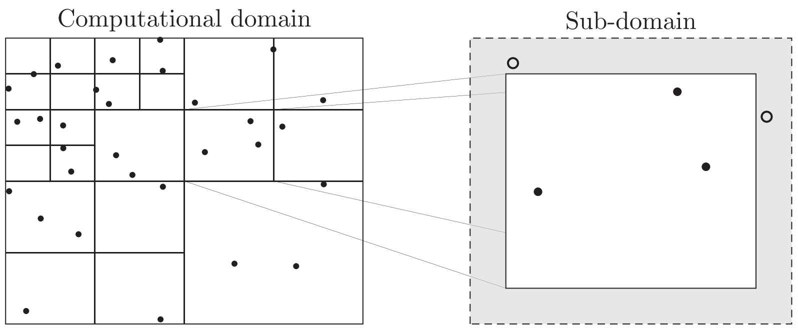


Fig. 4. The PPM library adaptively decomposes the computational domain (left) into subdomains (right). Each subdomain is extended by a ghost layer (gray shading) through which particles and mesh nodes from neighboring processes are accessible. Therefore, the ghost layer is populated with copies (circles) of real particles (dots) from neighboring subdomains using PPM's `ghost-get` mapping [Sbalzarini et al. 2006].

ALGORITHM 1: Multithreaded PPM particle-mesh interpolation.

```
(1) Allocate derived type object for routine arguments
(2) Allocate a threads array containing the IDs of created threads
(3) Copy pointers to particle positions and properties, meshes, and parameters to routine argu-
    ment object
(4) For each subdomain
    (a) Create a new thread passing the interpolation subroutine as start routine and the sub-
        domain ID as routine argument
    (b) store the thread ID in the threads array
(5) For each subdomain
    (a) Retrieve the associated thread ID and join the thread
```

maintaining state-of-the-art parallel performance [Chatelain et al. 2008; Sbalzarini 2010; Sbalzarini et al. 2006; Walther and Sbalzarini 2009].

We demonstrate the use of forthreads by adding three threading extensions to the PPM library: Multithreaded particle-mesh interpolation, a multigrid Poisson solver with computation-communication overlap using a dedicated communication thread, and interactive computing with PPM using a runtime socket server running in a separate thread. These extensions allow PPM to make better use of multicore architectures, offering opportunities for improved scalability and usability.

3.1.1. Particle-Mesh Interpolation Using Forthreads. The current parallelization model of PPM foresees that one MPI process holds several subdomains of the decomposed computational domain. This can conveniently be taken advantage of to execute independent operations on the different subdomains in parallel using threads. We thus modify the particle-mesh interpolation routines of PPM to execute on a single subdomain. The main interpolation routine spawns one thread per subdomain and executes the interpolations on the different subdomains in parallel (Algorithm 1). The arguments to the interpolation routines must be passed in heap memory, instead of the call stack, because of forthreads' restriction to allow only one `INTEGER` pointer to be passed as an argument to the thread-start routine. Since all subroutine arguments are identical for the different subdomains and they are only *read* by the interpolation routine, it is sufficiant to store only one set of arguments. The subdomain ID is passed as the sole argument to the thread. The threads are created and afterwards immediately joined,

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) FIRSTPRIVATE(lda, dxxi, dxyi, dxzi) &
!$OMP& SHARED(topo, store_info, list_sub, xp, up, min_sub, max_sub, field_up)
    DO isub = 1,nsubs
        ! perform interpolation for subdomain isub
    END DO
!$OMP END PARALLEL DO
```

```
MODULE ppm_p2m_interpolation
  TYPE t_shared_interp_args
    INTEGER :: topoid
    REAL, DIMENSION(:), POINTER :: wp
    REAL, DIMENSION(:,:), POINTER :: xp
    REAL, DIMENSION(:,:,:), POINTER :: field_wp
    REAL, DIMENSION(3) :: dx
  END TYPE
  TYPE(t_shared_interp_args) :: args
  !...

  SUBROUTINE p2m_interp(...)
    !...
    args%topoid = topoid
    !...
    args%dx = dx
    DO isub = 1,nsubs
      CALL forthread_create(thread_id, attr_id, p2m_interp_sub, isub, info)
    END DO
    !...
  END SUBROUTINE p2m_interp
END MODULE
```

Fig. 5. Listing showing the code required to shared-memory parallelize the particle-mesh subroutine in PPM using OpenMP and forthreads. The forthreads code requires storing shared data in the module; here we use a derived type to group all needed variables. Each thread is then created with the start routine `p2m_interp_sub` and the thread-specific argument `isub`.

which amounts to a barrier at the end of the interpolation, ensuring all subdomains have been completely interpolated before the simulation proceeds.

We compare our forthreads approach with a reference OpenMP implementation. OpenMP offers a simple and efficient solution to this specific problem, as we use an SIMD pattern (Figure 5). Both approaches are comparable in terms of the required code modifications. However, OpenMP provides a more compact syntax (compare the two implementations in Figure 5). More importantly, both implementations perform comparably well in terms of performance, as shown in Figure 6. All timings were performed on an AMD Opteron 8380 using 8 threads on 8 cores and a problem size of $512 \times 512 \times 512$ with 1 particle per mesh cell. All benchmark code was compiled using GCC 4.6.2 and the `-O3` flag.

3.1.2. Multigrid Poisson Solver with Computation-Communication Overlap. We demonstrate forthreads' use as a Fortran library for shared-memory task parallelism by porting PPM's multigrid Poisson solver for heterogeneous platforms. Multigrid (MG) methods use a hierarchy of successively coarsened meshes in order to efficiently invert a matrix, resulting, for example, from spatial discretization of a partial differential equation (we refer to Trottenberg et al. [2001] for details on the numerical method and the multigrid algorithm). The numerics part of the PPM library includes an MG solver for

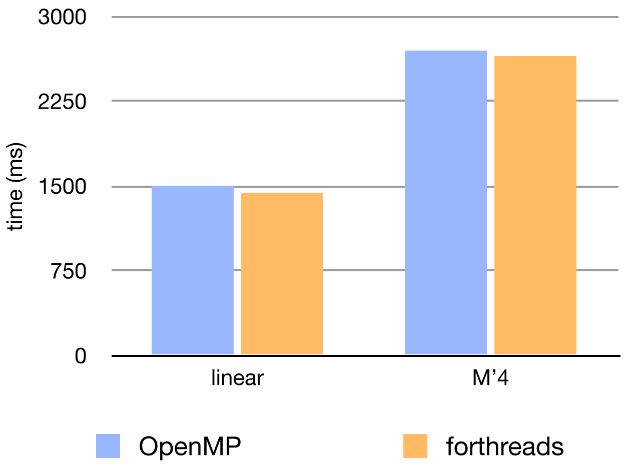


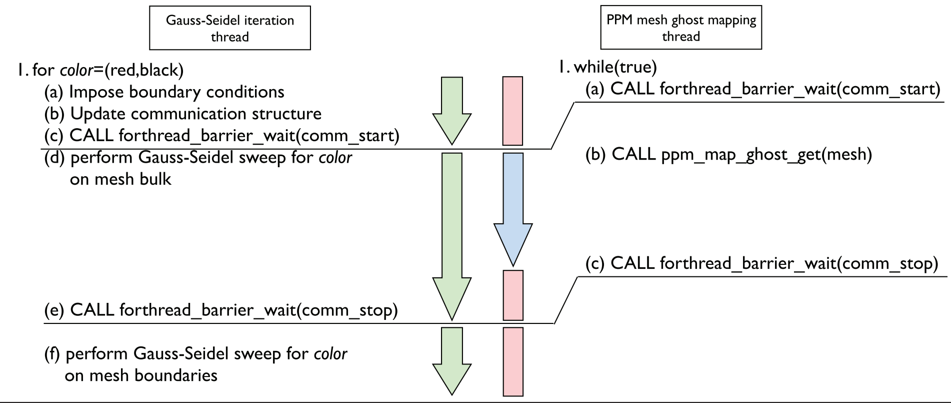
Fig. 6. Timings for linear and M_4' [Monaghan 1985] particle-mesh interpolation using OpenMP (blue) and forthreads (orange). All timings were performed on an AMD Opteron 8380 using 8 threads on 8 cores. The used problem size is $512 \times 512 \times 512$ with 1 particle per mesh cell.

finite-difference discretizations of the Poisson equation. The solution is found by iteratively applying a linear system solver, such as the Gauss-Seidel method or successive overrelaxation, restricting the error of the solution onto a coarser mesh, possibly applying more solver iterations, and finally interpolating back onto the original mesh. The MG method has its roots in the 1960s, but was popularized by Brandt [1977]. Today, MG solvers are widespread computational tools in science and engineering.

To make use of multicore platforms, we extend the current MG implementation in PPM by encapsulating the calls to PPM mesh-ghost mapping routines in a separate, concurrent thread. This enables overlapping computation and communication. Algorithm 2 describes the forthreads MG implementation. Apart from thread creation in the MG initialization routine, we only need to adapt the iterative solver (in this case a Gauss-Seidel method) and add a new routine encapsulating the calls to the PPM communication abstractions in a concurrent thread. The communication thread is executed in an infinite loop, waiting at the `comm_start` barrier. When the main computation thread enters the Gauss-Seidel solver routine and reaches the `comm_start` barrier, the communication thread becomes active. It updates the mesh-ghost layers using communication from neighboring processes (provided by the PPM mapping abstraction). At the same time, the next solver iteration is performed on the bulk mesh, that is, away from the boundaries that are currently being communicated. Both threads subsequently synchronize at the `comm_stop` barrier before the computation thread continues solving the mesh boundaries using the new ghost values.

Even though our implementation successfully overlaps MPI communication with computation, its runtime is higher than that of the MPI-only implementation. We believe the reason for this is twofold: First, the newly added iteration index calculations that are necessary to filter boundary mesh points from the main bulk iterations incur a large overhead. The red-black Gauss-Seidel iterations on mesh bulks require initializing the mesh indices according to a number of state variables, which prevented them from being vectorized by the compiler. Precomputing these mesh indices could potentially alleviate the problem and improve the efficiency of the multithreaded solver routine. Second, PPM's ghost mapping communication schedule requires multiple communication rounds in order to prevent network conflicts and deadlocks. A 3D Cartesian topology on 8 processors, for example, requires 8 communication rounds. On

ALGORITHM 2: PPM numerics multigrid Poisson solver using forthreads. The arrows visualize the control flow and thread states (green/blue: run, red: wait). The communication thread (right) is executed in an infinite loop. The computation thread (left) performs a given number of iterations; we show here one iteration. The communication thread updates the field's elements at the thread boundaries while the computation thread is performing a Gauss-Seidel sweep on the bulk of the field. Barriers are used to ensure correct synchronization between the threads and to prevent the boundaries from being overwritten before they have been used for the completion of the Gauss-Seidel sweep.


Gauss-Seidel iteration thread
PPM mesh ghost mapping thread
1. for *color*=(red,black)
(a) Impose boundary conditions
(b) Update communication structure
(c) CALL forthread_barrier_wait(comm_start)
(d) perform Gauss-Seidel sweep for *color* on mesh bulk
(e) CALL forthread_barrier_wait(comm_stop)
(f) perform Gauss-Seidel sweep for *color* on mesh boundaries
1. while(true)
(a) CALL forthread_barrier_wait(comm_start)
(b) CALL ppm_map_ghost_get(mesh)
(c) CALL forthread_barrier_wait(comm_stop)

64 processors, 27 communication rounds are required. Consequently the volume-to-surface ratio of the subdomains should be increased in order to mask the increased ghost mapping time with a matching amount of bulk-mesh computation time.

3.1.3. Interactive Computing with PPM Using Forthreads . Many modern applications use task parallelism and threads to allow for quick, responsive interaction with the user. We use forthreads together with a POSIX Internet sockets Fortran wrapper to provide a prototypic server instance allowing remote clients to connect to and control a running PPM simulation. The server is capable of handling an arbitrary number of concurrent client connections. Such server extensions allow PPM-based simulations to be directly controlled by the user. In addition, they also enable them to communicate at runtime with other applications, such as visualization tools, cluster management, databases, and Web browsers.

In order to extend PPM with an Internet server thread, we first build a simple wrapper for the POSIX Internet socket API for Fortran (which we call "fsocket"), abstracting some of the intricacies of the sockets API. This wrapper, however, is not complete. It is specifically geared toward providing the necessary functionality for building TCP Internet servers in PPM. It provides the following functions.

— `fsocket_init()` must be called before any other fsocket routine. It creates and initializes an internal data structure for mainting the open connections and file descriptors;
— `fsocket_server_create()` is a shortcut for the `socket()` and `bind()` functions. It creates a socket address structure and requests a file descriptor;
— `fsocket_listen()` wraps the `listen()` function indicating that the caller is ready to accept incoming connections;
— `fsocket_accept()` creates a new client address object, then calls the `accept()` function, which returns as soon as an incoming connection is to be established. Once the connection is established, the client address and file descriptor are stored in the internal data structure and a unique ID is returned to the caller;

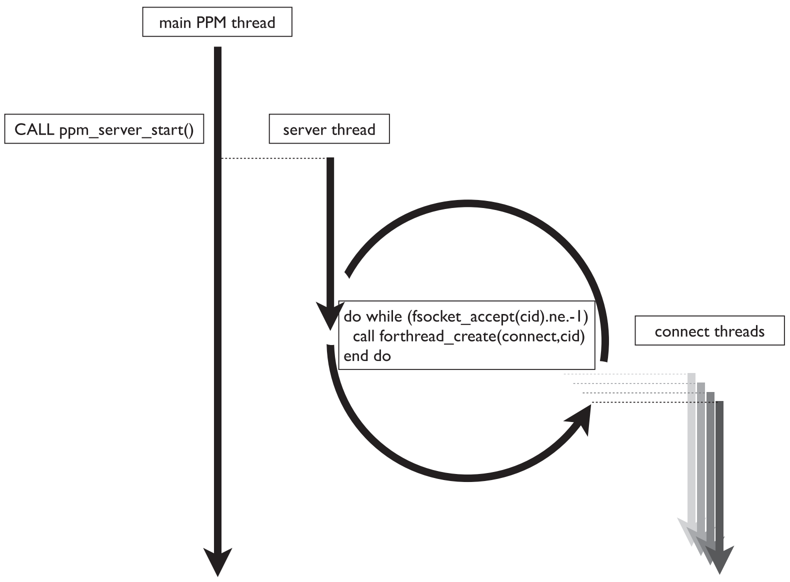


Fig. 7. PPM server. The arrows visualize the thread control flow; dashed lines are thread spawns. The PPM server creates a thread for the server listen loop and for each newly established connection.

— `fsocket_read()` and `fsocket_write()` are simple wrappers for the `read()` and `write()` functions. They allow reading and writing character buffers from/to the socket;
— `fsocket_close_conn()` and `fsocket_close_server()` wrap the `close()` function, either passing the client connection file descriptor or the server file descriptor.

In order to extend the current version of fsocket to a general-purpose Fortran sockets interface, one should at least separate the `socket()` and `bind()` functions, provide generic interfaces for building socket address structures, and extend the `fsocket_read()` and `fsocket_write()` routines with a type argument allowing arbitrary Fortran primitive types, similar to MPI's communication routines.

We extend the PPM core library by adding a simple Fortran module providing a single user-facing subroutine. This routine is responsible for spawning a new thread (using the forthreads library) that creates an Internet server socket and enters the main server listen loop. Whenever a new client connection is established, this loop advances by one iteration and creates an additional thread for handling the new client connection. This mechanism ensures that the running PPM application remains responsive to all connecting clients while at the same time continuing its normal operation. The mode of operation of this PPM server is summarized in Figure 7.

Since the PPM server is executed independently by each process, the user may directly address and interact with specific MPI processes of a running PPM simulation. An example is shown in Figure 8, where a connection is established to a PPM process running on the local host, and the dimensionality of the currently solved problem is inquired.

4. SUMMARY AND CONCLUSION

We have developed a comprehensive binding of the POSIX threads API to Fortran 2003 that gives the programmer access to almost all thread management functions and all thread synchronization constructs. Using this library, the programmer is only

```
>$ telnet 127.0.0.1 1337
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
PPM command server 0.1 - Welcome
> hello
Hi there!
> dim
ppm_dim is: 2
> exit
bye Connection closed by foreign host.
```

Fig. 8. Log of example PPM command server session.

exposed to native Fortran interfaces. Forthreads extends previous implementations, most notably by Hanson et al. [2002] and Nagle [2013]. We showed the versatility of forthreads in three examples using different design patterns. All examples extended the PPM library [Awile et al. 2010; Sbalzarini et al. 2006] using forthreads in order to provide new functionality and multicore support. First, we have extended the existing particle-mesh interpolation routines to spawn one thread per subdomain, executed on separate processor cores. The benchmarks showed that our implementation yields a performance comparable to an OpenMP reference implementation, while offering improved control over the threads. This comes, however, at the cost of a slight increase in code complexity. Second, we have redesigned and ported the existing implementation of the multigrid Poisson solver of the PPM library to use threads. The thread-enabled multigrid solver maintains a separate communication thread, allowing overlapping computation with communication. The current implementation, however, has significant shortcomings over the original implementation in terms of time efficiency. We do, however, suspect that this is not due to the use of forthreads, but is caused by the loss of code vectorization due to the index algebra needed to separate bulk-mesh nodes from boundary nodes. Third, we have implemented a Fortran wrapper for the POSIX Internet socket API and a new PPM module providing a control server for running PPM simulations. This server is capable of handling several simultaneous client connections, providing interactive computing capabilities to PPM.

Mixed shared-/distributed-memory parallel programming has in several instances shown significant improvements over pure distributed-memory parallelizations [Madduri et al. 2009; Song et al. 2009; Winkel et al. 2012]. The forthreads library is intended to offer a simple yet powerful alternative to existing parallelization frameworks for shared-memory parallelism in Fortran 2003. Forthreads is complete in the sense that it provides native Fortran 2003 interfaces to all POSIX threads routines where possible. Forthreads also maintains the opacity of the internal pthreads types and data structures, as required by the POSIX standard. Together with the fsocket wrapper for the POSIX Internet socket API, we believe that forthreads will be a useful tool for developing numerical software for multicore platforms. Forthreads is freely available on `https://github.com/ohm314/forthreads` and on `http://mosaic.mpi-cbg.de`. The presented examples, including the PPM MG solver, are available as part of the open-source PPM Library from `http://mosaic.mpi-cbg.de/?q=downloads/ppm_lib`.

ACKNOWLEDGMENTS

We thank all members of the MOSAIC Group (ETH Zurich and MPI-CBG Dresden) for many fruitful discussions.

REFERENCES

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. 2009. A view of the parallel computing landscape. *Comm. ACM 52*, 10, 56–67.

Awile, O., Demirel, O., and Sbalzarini, I. F. 2010. Toward an object-oriented core of the ppm library. In *Proceedings of the 8th AIP International Conference of Numerical Analysis and Applied Mathematics (ICNAAM'10)*. 1313–1316.

Bernaschi, M., Bisson, M., Endo, T., Matsuoka, S., Fatica, M., and Melchionna, S. 2011. Petaflop biofluidics simulations on a two million-core system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM Press, New York, 4:1–4:12.

Brandt, A. 1977. Multi-level adaptive solutions to boundary-value problems. *Math. Comput. 31*, 138, 333–390.

Breshears, C. P., Gabb, H. A., and Bova, S. 1998. Towards a fortran 90 interface to the posix threads library. In *Proceedings of the DoD High Performance Computing Modernization Program Users Group Conference.*

Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., and Storaasli, O. O. 2010. State-of-the-art in heterogeneous computing. *Sci. Program. 18*, 1, 1–33.

Chatelain, P., Curioni, A., Bergdorf, M., Rossinelli, D., Andreoni, W., and Koumoutsakos, P. 2008. Billion vortex particle direct numerical simulations of aircraft wakes. *Comput. Method. Appl. Mech. Engin. 197*, 1296–1304.

Dijkstra, E. W. 1965. Solution of a problem in concurrent programming control. *Comm. ACM 8*, 9, 569–569.

Dubey, A., Daley, C., and Weide, K. 2010. Challenges of computing with flash on largest hpc platforms. In *Proceedings of the 8th AIP International Conference of Numerical Analysis and Applied Mathematics (ICNAAM'10)*. Vol. 1281. 1773–1776.

Geer, D. 2005. Chip makers turn to multicore processors. *Comput. 38*, 5, 11–13.

Hanson, R. J., Breshears, C. P., and Gabb, H. A. 2002. Algorithm 821: A fortran interface to posix threads. *ACM Trans. Math. Softw. 28*, 3, 354–371.

Herlihy, M. and Moss, J. E. B. 1993. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News 21*, 2, 289–300.

Madduri, K., Williams, S., Ethier, S., Oliker, L., Shalf, J., Strohmaier, E., and Yelicky, K. 2009. Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM Press, New York, 48:1–48:12.

Madduri, K., Im, E.-J., Ibrahim, K. Z., Williams, S., Ethier, S., and Oliker, L. 2011. Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. *Parallel Comput. 37*, 9, 501–520.

Marowka, A. 2007. Parallel computing on any desktop. *Comm. ACM 50*, 9, 74–78.

Monaghan, J. J. 1985. Extrapolating b splines for interpolation. *J. Comput. Phys. 60*, 253–262.

Nagle, D. 2013. Fortran 2008 module pthread. http://www.daniellnagle.com/pub/pthread.f90.

Perez, J., Badia, R., and Labarta, J. 2008. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the IEEE International Conference on Cluster Computing*. 142–151.

POSIX. 2004. IEEE standard for information technology - Portable operating system interface (posix) base definitions. IEEE Std 1003.1, 2004 Edition. The open group technical standard base specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002, and IEEE Std 1003.1-2001/Cor 2-2004. Base.

Powell, M. L., Kleiman, S. R., Barton, S., Shah, D., Stein, D., and Weeks, M. 1991. Sunos multi-thread architecture. In *Proceedings of the Winter USENIX Conference*. 65–80.

Rabenseifner, R., Hager, G., and Jost, G. 2009. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. 427–436.

Sbalzarini, I. F. 2010. Abstractions and middleware for petascale computing and beyond. *Int. J. Distrib. Syst. Technol. 1*, 2, 40–56.

Sbalzarini, I. F., Walther, J. H., Bergdorf, M., Hieber, S. E., Kotsalis, E. M., and Koumoutsakos, P. 2006. PPM – A highly efficient parallel particle-mesh library for the simulation of continuum systems. *J. Comput. Phys. 215*, 2, 566–588.

Shavit, N. and Touitou, D. 1997. Software transactional memory. *Distrib. Comput. 10*, 99–116.

Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A., and Matsuoka, S. 2011. Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11).* ACM Press, New York, 3:1–3:11.

Song, F., YarKhan, A., and Dongarra, J. 2009. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis (SC'09).* ACM Press, New York, 19:1–19:11.

Speck, R., Arnold, L., and Gibbon, P. 2011. Towards a petascale tree code: Scaling and efficiency of the pepc library. *J. Comput. Sci. 2*, 2, 138–143.

Stein, D. and Shah Sunsoft, D. 1992. Implementing lightweight threads. In *Proceedings of the USENIX Summer Conference*. 1–9.

Trottenberg, U., Oosterlee, C., and Schueller, A. 2001. *Multigrid*. Academic Press, San Diego, CA.

Tsoi, K. H. and Luk, W. 2010. Axel: A heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'10).* ACM Press, New York, 115–124.

Walther, J. H. and Sbalzarini, I. F. 2009. Large-scale parallel discrete element simulations of granular flow. *Engin. Comput. 26*, 6, 688–697.

Winkel, M., Speck, R., Hübner, H., Arnold, L., Krause, R., and Gibbon, P. 2012. A massively parallel, multidisciplinary barnes–hut tree code for extreme-scale n-body simulations. *Comput. Phys. Comm. 183*, 4, 880–889.

Received October 2012; revised July 2013; accepted October 2013