Threads for Interoperable Parallel Programming

L. V. Kalé, J. Yelon, and T. Knauff

Dept. of Computer Science,
University of Illinois,
Urbana Illinois 61801,
jyelon@cs.uiuc.edu, kale@cs.uiuc.edu

Abstract. Many thread packages are freely available on the Internet. Yet, most parallel language design groups seem to have rejected all existing packages and implemented their own. This is unsurprising. Existing thread packages were designed for sequential computers, not parallel machines, and do not fit well in a parallel environment. Also importantly, existing thread packages try to impose a number of design decisions, especially in regard to scheduling and preemption. Designers of parallel languages are simply not willing to have scheduling methods decided for them, nor are they willing to allow the threads package to decide how concurrency control will work. In this paper, we explore the special issues raised when threads packages are used on parallel machines, particularly as parts of new parallel languages and systems. We describe the Converse threads subsystem, whose goals are to support the special needs of parallel programs, and to support interoperability among parallel languages. We then demonstrate how the Converse threads subsystem addresses the problems created when threads are used on a parallel computer.

1 Introduction

Many parallel programming languages rely upon lightweight threads as a fundamental component. Threads are extremely useful in implementing a vast range of constructs for parallelism. Use of threads can provide a number of benefits to a parallel program, such as improved latency tolerance, higher degrees of parallelism, and the potential for adaptive scheduling. When designing a set of thread primitives, it makes sense to take into account the special needs of parallel languages. There are many such special needs: parallel language runtime systems tend to impose very strict requirements on the properties of the thread package. Since parallel languages differ so much, each language offering its own set of advantages, it becomes critical to be able to use different languages for different modules of a parallel program. As a result, future parallel applications may contain many languages, with each language's runtime system making different demands of the thread subsystem. The existing threads packages, such as those based on the POSIX threads standard, are not designed for use in parallel programs, and are thus seen to be inadequate for parallel programming. This has led the implementors of new parallel languages to develop their own thread packages tuned to the needs of their language. This obviously makes the development and maintenance of runtime systems for such languages more complex than necessary. At the same time, such specialized thread packages make it harder, if not impossible, to achieve *interoperability* across languages — i.e. allowing multiple modules written in different multi-threaded languages to coexist and interleave their execution in a single program.

In this paper, we present the design and implementation of the Converse threads subsystem, which supports such interoperability. It addresses the issues created by the diverse needs of coexisting parallel languages. The Converse threads subsystem comprises a modular and flexible thread abstraction, related facilities, and protocols, which are used in conjunction with the remainder of the Converse system to implement runtime systems of multi-threaded languages. In the remainder of this section, we examine several of the special problems created when multiple parallel languages all try to share a threads package in a single application.

1.1 Thread Scheduling

Most thread packages execute threads in a round-robin fashion. This behavior is often inappropriate for parallel languages and applications. As an example, round-robin scheduling fails to take into account the fact that parallel programs generally have critical paths, unlike sequential applications that use threads. Some applications may require a specialized scheduler with knowledge of critical paths. A second example: round-robin scheduling destroys locality of reference when large numbers of threads are being used. When hundreds of threads coexist, randomly transferring control from one to another randomizes memory access patterns, and as a consequence, cache performance is reduced to nil. Again, a customized policy with knowedge of locality may be needed for some applications. A third example: round-robin scheduling, when used with tree-search applications, tends to produce breadth-first search. This is usually undesirable, depth-first search is usually more efficient, and breadth-first sometimes overflows the available memory. Again, one needs a specialized scheduling policy. In short, round-robin scheduling may be a good general-purpose policy, but there are many cases where a customized policy is needed.

One "solution" to this problem is to add priorities to the scheduler. This provides a means by which a language can encourage the scheduler to pick the "right" thread. However, there are some cases in which priorities aren't sufficiently expressive. Two of the three cases listed above are not amenable to prioritizing. Attempting to encourage depth-first search with priorities is clumsy and is not possible unless the priorities have a large number of significant bits. Encouraging locality with priorities is not feasible. In short, priorities are quite useful for some applications, but there will be other scheduling objectives that cannot be achieved in this manner.

One could attempt to devise a more sophisticated scheduler to try to handle these complex needs. However, it is obvious that such a scheduler would be language-dependent, and therefore could not be part of the threads package. Fortunately, there is at least one threads package, QuickThreads [6], which allows one to implement one's own scheduler. Thus, each language's runtime system could contain a scheduler appropriate to its needs. Unfortunately, though, this approach causes an arbitration problem: if an applications programmer wishes to combine multiple parallel languages (with multiple runtime systems) in a single application, then that programmer must remove the schedulers from the runtime systems of the languages and write a new scheduler which is compatible with all. It is clearly undesirable to force the application programmer to undertake such a task.

One of the facilities provided by the Converse threads subsystem is a hierarchical scheduling model, which makes it possible for one module or language to implement its own scheduling behaviors without sacrificing the ability to coexist with other modules using the standard scheduler.

1.2 Portability Concerns

The second problem facing thread users today is the lack of a single threads package which runs on all platforms. Many threads packages claim to be "portable", but very few actually compile without modification on any significant number of UNIX platforms. The user who wishes to run threads on a machine with an unusual CPU, unusual compiler, or different operating system typically ends up having to do his own port of the threads package. Unfortunately, parallel machines tend to count as "unusual" in this sense: they often use custom processors, compilers, and operating systems. And, usually, it's the parallel language implementor who has to maintain the port, since thread package implementors are rarely interested in supporting some strange supercomputer to which they don't have access.

A solution to this problem is to design a threads package which can be implemented in a machine-independent manner. The machine-independent implementation may not be as efficient as the carefully tuned assembly-language implementations, however, its presence makes it possible for users of unusual parallel machines to simply plug the threads package into their application, knowing that it will work. An optimized implementation can be written later if necessary. The Converse threads subsystem can be (and has been) implemented in a fully machine-independent manner, and has been successfully tested on all the machines supported by Converse.

1.3 Preemption and Nonpreemption

There is no general consensus regarding whether or not a standardized threads package should support preemptive context-switching or not. There are strong arguments in favor of both preemptive and nonpreemptive threads.

Preemptive threads have several disadvantages in terms of convenience and efficiency. On a distributed memory machine, where concurrency control is not normally needed, preemptive threads require that one rewrite all the libraries in a reentrant fashion. This is usually achieved by adding locking to the nonreentrant routines, but the cost of of such locking can be quite significant when it surrounds an otherwise cheap routine. This performance hit is compounded by the fact that one must pay for operating system interrupts and context switching far more often than one would in a nonpreemptive system. Finally, preemptive threads can easily violate the semantics of a parallel programming language if the language was not designed for concurrent access to variables.

However, there are also strong arguments in favor of preemption. Preemption can significantly improve IO performance in systems where IO is polled. Preemption is useful for keeping priorities up to date — without preemption, high-priority tasks can end up waiting for lower-priority tasks to yield. Consider, for example, the situation in which a high-priority thread is waiting for a message. With preemption, this can be easily handled: when the message arrives, one simply preempts any low-priority thread. Preemption can significantly improve interactive response time. Finally, preemption is a required part of many parallel programming models.

Clearly, this is a tradeoff. Some runtime system designers will need preemption, others will find it unacceptable. Given this state of affairs, it doesn't make sense to design a threads package with a fixed preemption policy. In fact, in an interoperable environment, where multiple languages are linked together in a single application, it is clear that preemptive and nonpreemptive threads need to coexist. The Converse threads subsystem supports the use of preemptible and nonpreemtible threads in the same program.

1.4 Compiler-Supported Threads

Currently, most thread packages do not get any explicit support from the compiler. However, thread-packages which have the compiler's support can be significantly more efficient, as is demonstrated by such languages as Concert [11]. It therefore seems likely that popular compilers like gcc will eventually be extended to include support for high-speed threads.

However, by the time this occurs, most users will already be using a conventional thread package. If that conventional thread package has a well-designed interface, it will be possible to efficiently reimplement it as an abstraction layer on top of the high-speed compiler-supported threads. If not, then users of that thread package will need to discard their software and start over to gain the efficiency advantages of compiler-supported threads. Therefore, it is important that thread primitives be designed in such a way that they can be efficiently reimplemented with compiler support.

For a set of threads primitives to be efficient in a compiler-supported context, it must not utilize inherently expensive abstractions. If a threads package is to be reimplemented as a layer on top of an extremely fast compiler-supported threads package, it is critical that the abstractions not add excessive overhead to what could otherwise be a very fast system.

An example of a costly abstraction is the "mutex" variable. Each mutex operation is actually built from a number of simpler primitives, such as as explicit context-switching, thread queues, and monolithic monitors. By forcing a language designer to implement a basic primitive (e.g., the future) in terms of a full-featured abstraction like the mutex (when it could be implemented using much less costly abstractions) robs the language designer of needed performance.

Given the inefficiency of currently available thread packages, expensive abstractions are rarely noticed: the overhead associated with using powerful abstractions is irrelevant in comparison to the large context-switch overhead. However, when compiler-supported threads become available, software designers who used a threads package with an efficient interface will simply download a faster version of the threads package, whereas designers who used a threads package with an expensive abstraction layer will need to rewrite much code to gain the efficiency they desire.

1.5 A Model for Shared and Private Data

Designers of thread packages for workstations have clearly reached consensus regarding their model of shared and private data: global variables are to be shared among all threads, and thread-private data is to be obtained by function calls. Unfortunately, this model is unusable on parallel computers. Global sharing is simply too expensive (if available at all) to be the standard meaning for global variables. As a consequence, designers of thread packages for parallel computers have rejected the workstation-oriented model of global sharing, and without any consensus to adhere to, each thread package designer chooses a different meaning for global variables. For example, with Solaris threads, global variables turn out to be shared among all threads, while with QuickThreads on distributed memory machines, they turn out to be partially shared, and with the vendor supplied thread on the Convex Exemplar they turn out to be private.

The unfortunate consequence is this: if the programmers use a global variable in a parallel multithreaded program, their code is not portable. This is a *severe* problem, and one that can only be dealt with by the threads implementor. If the threads package does not provide a consistent interpretation of whether a given variable is to be private or shared across threads, then code using the package is *not* portable.

No solution is possible until a sharing model is adopted which is more consistent with the properties of parallel computers. Converse commits to a model which acknowledges the fact that several levels of sharing are possible, each with its own advantages. Converse therefore provides primitives to declare and access data at each level of sharing.

2 Converse Threads Subsystem

Converse [4] is a machine interface for parallel systems and languages. It seeks to achieve the following major goals:

1. portability: that programs written on top of Converse be executable without modification on a wide range of platforms,
2. generality: to be able to implement the full spectrum of parallel constructs on top of Converse.

3. interoperability: the ability for unrelated and highly dissimilar modules, often written in different parallel languages at different institutions, to *concurrently* execute, exchange data, and share the machine interface without conflicts.

Interoperability is the feature that sets Converse apart from most runtime systems — Converse provides a variety of resource arbitration techniques not found in other parallel runtime systems. Chief among these resources is CPU time itself. An important part of Converse's mechanisms for allocating and managing CPU time is the Converse threads subsystem, which is the focus of this paper. The other parts of Converse are discussed elsewhere [4].

The Converse threads subsystem is not a threads package in the traditional sense of the phrase. It includes some software, to be sure, but its goal is not just to provide some useful subroutines. Instead, it seeks to guarantee that software written with the Converse threads subsystem will be both portable and "interoperable" — in other words, it seeks to guarantee that contention over thread facilities will never be a source of incompatibility between two modules. Therefore, the Converse threads subsystem includes not just software, but also a set of "rules" which, if followed, enable threaded modules written by different research groups to coexist.

The Converse threads subsystem contains the following major components:

The Thread Objects Module: the most fundamental elements of a thread system: routines for creating threads and context switching.

A Standardized Scheduler Interface: a protocol whereby blocking routines can be written without knowing what scheduling policies are in place.

The Converse Scheduler: A scheduler provided by Converse. In addition to providing a convenient scheduler for many languages, it serves as a central clearinghouse for CPU time.

Yielding Mechanisms: A set of mechanisms that threads may use to temporarily give up control. Both nonpreemptive (manual) and preemptive (automatic) yielders are available.

A Model of Thread-Private and Thread-Shared Data: Converse defines the concepts of the address space, the processor, and the thread, and provides ways of declaring data with several levels of sharing.

The following sections describe the elements of the Converse threads subsystem in greater detail.

2.1 Thread Objects

The thread-object module, like most thread packages, provides a function for creating threads, one for destroying threads, one for explicitly transferring control to another thread, and one for retrieving the currently-executing thread. The following calls are the most important ones:

```
typedef struct CthThreadStruct *CthThread;
```

This is an opaque type defined in the Converse header files. It represents a first-class thread object. No information is publicized about the contents of a

CthThreadStruct. The fact that threads are first-class objects makes it possible to express many operations elegantly.

`CthThread CthCreate(void (*fn)(void *), void *arg, int size)`

Creates a new thread object. `CthCreate` returns the thread identifier of the newly created thread. The newly-created thread is not yet executing.

`void CthResume(CthThread t)`

Immediately context-switches (transfers control to) thread `t`. Note: normally, the user of a thread package wouldn't explicitly choose which thread to transfer to. Instead, the user would rely upon a scheduler to choose the next thread. Therefore, this routine is primarily intended for people who are implementing schedulers, not for end-users. Instead of calling `CthResume`, most threads will use a scheduler-provided function `CthSuspend` to context switch, this is described in the next section.

`CthThread CthSelf()`

Returns the currently-executing thread.

`void CthFree(CthThread t)`

Releases the memory associated with thread `t`. You may even `CthFree` the currently-executing thread, although the free will actually be postponed until the thread suspends.

2.2 Schedulers and the Standardized Scheduler Interface

We formalize the idea of a scheduler in the following manner: when a thread stops (context-switches out), another thread must take over the CPU. At that moment, a decision must be made regarding which thread will take over the CPU. Any subroutine or module making such decisions is termed a "scheduler".

Converse includes a standard scheduler. However, we recognize that the Converse scheduler uses a policy that may not be applicable to all parallel programming languages. Therefore, we allow language designers to write their own schedulers. Each thread will be managed by one scheduler, of its own choosing.

Since there will be multiple schedulers available, there is significant potential for incompatibilities. For example, suppose an implementor wished to write a subroutine that reads from the keyboard. The subroutine, upon discovering that no characters were available, would need to call some "suspend" subroutine. However, the keyboard IO routine could conceivably be called from many different languages, each using a different set of scheduling subroutines. Therefore, the keyboard IO routine would need to figure out which subroutine to call to ask for suspension. To make this feasible, we define a standardized way in which threads ask to suspend and ask to be reawakened. This interface is the "standard scheduler interface".

The standard scheduler interface consists of two functions: `CthSuspend` and `CthAwaken`. From the point of view of the thread, these methods perform as follows. When the thread wishes to block, it calls `CthSuspend()`. This causes a transfer of control to some "ready" thread. A thread `t` which has called

`CthSuspend()` is considered not-ready, and it remains not-ready until somebody calls `CthAwaken(t)`. After this time, the thread is considered ready, and it remains ready until it suspends again.

From a scheduler's point of view, the function `CthSuspend()` means "transfer control to a thread in the ready-set". `CthAwaken(t)` means "insert thread `t` into the ready-set". The scheduler's job is to record the contents of the ready-set, and choose elements from it as necessary.

At any given moment, each thread is connected to one scheduler. When the thread suspends, its scheduler picks the next thread. When a thread is awakened, its scheduler records the fact that it is now ready. A thread is associated with a scheduler using the function `CthSetStrategy`, described below.

The following is a brief description of the interface via which threads talk to their schedulers.

`void CthSuspend()`

Threads should call `CthSuspend` when they wish to give up control of the CPU. `CthSuspend` will then automatically transfer control to some other thread that wants the CPU. `CthSuspend` will select the thread to transfer control to by calling the scheduler-supplied, thread-specific "choosefn" described in `CthSetStrategy` below.

`void CthAwaken(CthThread t)`

Should be called only when `t` is suspended. Indicates to thread `t`'s scheduler that `t` is no longer suspended, that it needs to be selected for execution. This function actually just calls the scheduler-supplied, thread-specific "awakenfn" described in `CthSetStrategy` below.

```
void CthSetStrategy(CthThread t,
                    void (*awakenfn)(CthThread t),
                    CthThread (*choosefn)(void))
```

Specifies the scheduling functions to be used with thread `t`. Subsequent to this call, any attempt to `CthAwaken(t)` will cause then `t`'s `awakenfn` will be called. If `t` calls `CthSuspend`, then thread `t`'s `choosefn` will be called to pick the next thread. One may use the same functions for all threads (which is the common case), but the specification on a per-thread basis gives you maximum flexibility in controlling scheduling.

2.3 The Converse Scheduler

The Converse Scheduler is a fully-functional scheduler with supporting a powerful priority system, and support for a highly efficient form of stackless thread (for short, nonblocking tasks) in addition to its support for standard threads. A standard thread can be attached to the Converse scheduler using the following function:

`void CthSetStrategyDefault(CthThread t)`

After this call, the central Converse scheduler will automatically pick the next thread whenever `t` suspends or yields, and it will arrange for control to return to `t` when `t` is awakened.

The following subroutine is available for priority management:

```
void CthSetPrio(CthThread t, int strategy, int priolen, int *prio)
```

Sets the priority and queuing behavior of thread `t`. The priority is a sequence of bits representing an arbitrary-precision number. The strategy can be LIFO or FIFO (the thread is inserted on the ready queue either before or after threads of the same priority).

For many programs, the Converse scheduler will be quite sufficient in itself. It is quite common for several modules written in different multithreaded languages and running concurrently to simply share the Converse scheduler.

Although the Converse scheduler is an excellent basic scheduler, it cannot account for all the peculiarities of the languages implemented using Converse — for example, it cannot anticipate critical paths, whereas a programming language's runtime system could conceivably have good critical-path heuristics. Language designers may therefore wish to write their own schedulers to manage their own threads. In Converse's spirit of interoperability, these home-brewed schedulers must coexist with the Converse scheduler and other home-brewed schedulers. Therefore, the Converse scheduler has a second role: as a central clearinghouse for CPU time.

The principle of hierarchical scheduling is based on the idea that the scheduler is actually an allocator of CPU-time, and can be compared to memory allocators. Arranging memory allocators into hierarchies is quite common: for example, a lisp programmer might allocate memory using `cons`, which originally obtained its memory from `malloc`, which in turn obtained its memory from the UNIX primitives `brk` and `sbrk`. Hierarchical schedulers are based on exactly the same principle: initially, all CPU time is owned by the Converse scheduler. A different scheduler can request an indefinite amount of time from the central scheduler by inserting a thread into the central scheduler's queue. When the thread receives the CPU, it can then allocate time segments to other threads by calling `CthResume` to restart them. Those must eventually return control to their scheduler, which must eventually return control to the central Converse scheduler.

2.4 Yielding Mechanisms

The facilities we have shown thus far make it possible to write uninterrupted threads — threads that run continuously until they call `CthSuspend`. Such threads retain the CPU until they have no computation left to perform.

Many thread-packages use a preemptive context-switching policy. Converse, however, recognizes that preemptive context-switching can be very destructive to program correctness: it introduces a number of concurrency control issues, can destroy nonreentrant subroutines, and can violate the semantics of many parallel programming languages. On the other hand, preemptive context-switching can provide significant advantages in terms of IO performance and in terms of keeping priorities current. Therefore, Converse provides facilities for preemptive context-switching, but in a very conservative manner. Each thread must individually request preemptive behavior. If a thread does not ask to be preemptible, then that thread is not preempted. By providing preemptible threads for languages that need it, and allowing those threads to coexist with nonpreemptible threads for languages without concurrency control, Converse preserves interoperability.

Even nonpreemptible threads may wish to yield occasionally. In fact, we have found that manual yielding is sufficient for almost all purposes, and it does not create concurrency control problems. Therefore, we provide the following manual yielding mechanism.

`void CthYield()`

Temporarily suspends the current thread. This requires no additional support from the scheduling module, since it is implemented as follows: The thread simply adds itself to it's scheduler's ready-set using the scheduler's "awaken" method, and then transfers control to another thread using its schedulers "choose-next" method.

Converse provides the following facility for traditional preemptive yielding:

`void CthAutoYield(int microsec)`

Called by a thread to request "traditional" preemption. After this call, the thread that called it will be automatically preempted every `microsec` microseconds, approximately. This does not cause any other thread to be preempted.

`void CthSigYield()`

This function should be called either from inside a UNIX signal handler or a Converse event handler. (Converse generates some events, which are much like UNIX signals. Active message arrival is an example of a Converse event.) It causes the currently active thread on the current processor to yield. Accepting such yields is optional, see `CthSigYieldEnable` below. Signal-based yielding is particularly useful if the system has just received a high-priority interrupting message that needs to be processed.

`int CthSigYieldEnable(int flag)`

This functions enables preemptive yielding if `flag` is nonzero, and disables it if the flag is zero. Returns the old value of the flag. Initially, threads do not allow signal-based yielding. They must explicitly turn it on with `CthSigYieldEnable`. If a `CthSigYield` occurs and signal-based yielding is not being accepted by the current thread, the yield is delayed until the next manual yield or until signal-based yielding is enabled.

2.5 A Model of Private and Shared Data

Programming languages for uniprocessors make no presupposition about whether or not global variables, common blocks and the like are to be shared across threads. Therefore, thread-packages for parallel computers make inconsistent decisions about global variables. For example, Solaris threads [14] treats global variables as shared across threads, whereas the Convex vendor-supplied threads treat them as private. Other thread packages have other policies. The practical consequence for those attempting to write portable code is this: global variables have completely unpredictable behavior, and therefore, undefined semantics. Without a known behavior for global variables, it becomes extremely difficult to use any sort of globally-scoped data, private or shared.

Writing a useful, portable package requires a clear model of what kinds of sharing are possible. Converse identifies several levels of sharing, based on the following machine model. A parallel machine consists of a number of address spaces. Each address space contains a number of processors (or virtual processors). Each processor can support an arbitrary number of threads. The number of address spaces varies, and may be only one, likewise with the number of processors per address space. Given this machine model, there can be three kinds of variables: those where one copy per thread exists (thread-private), those with one copy per processor (processor-private), and those with one copy per address-space (shared). Converse provides macros to define variables at each level of sharing. Here are the most important macros for declaring, initializing, and accessing thread-private data:

`CtvDeclare(type, varname)`

Used in a fashion analogous to a C global variable declaration. One may place this macro at the top level of one's program to declare thread-private variables.

`CtvInitialize(varname)`

This macro must be invoked once for each thread-private variable that is declared in the program. It must be called exactly once per thread-private variable declaration, before any threads are created. It is therefore convenient to do so temporally near the start of `main`.

`CtvAccess(varname)`

Thread-private variables must be accessed with this macro. For example, to add one to the thread-private variable `X` one might say `CtvAccess(X) = CtvAccess(X) + 1`, or just `CtvAccess(X)++`.

The macros for declaring processor-private and shared variables are identical, except that they begin with `Cpv` and `Csv` respectively. With the help of these macros, it is possible for the parallel programmer to meaningfully use non-local variables.

The macros for declaring variables at each level of sharing have been carefully optimized. For example, the macro for accessing a thread-private variable takes approximately 3 Sparc instructions per fetch, which can be compared to 2 Sparc instructions for fetching a global variable.

2.6 Synchronization Mechanisms

Parallel languages use a variety of interesting and complex synchronization abstractions. It would be impossible for a threads package to provide all of them. Instead, to be usable in a parallel environment, a threads subsystem must provide efficient support for user-level implementation of synchronization abstractions. In this section, we show an implementation of such an abstraction, using the primitives described in previous chapters. The implementation illustrates the relative simplicity of building synchronization abstractions in Converse.

The synchronization mechanism we will use for demonstration purposes is the mutex. Mutexes may be in one of two states, either locked or unlocked, and they support two operations, “lock” and “unlock”. A “lock” cannot proceed unless the mutex is in the unlocked state, therefore, the locking subroutine waits (blocks) until the mutex is in the unlocked state. These routines can be easily implemented using the Converse primitives. The mutex data type itself has the following structure:

```
typedef struct {
    int locked;
    queue_of_threads queue;
} *mutex;
```

If a thread (hereafter known as the “locker”) wishes to lock the mutex, it must first check whether the mutex is already locked. If not, the locker can simply lock the mutex. If the mutex is already locked, however, the locker must wait for the mutex to enter the unlocked state. It pushes itself onto the queue of lockers. When the mutex is unlocked, the unlocker promises to 1, remove a locker from the queue, 2, relock the mutex on behalf of the locker, and 3, reawaken the locker. Therefore, when a locker is reawakened after being in the queue, the mutex has been locked on its behalf, and it may go on. Figure 2 shows the code executed by the locker and unlocker respectively.

```
void MutexLock(mutex m)
{
  int oldenable
    = CthSigYieldEnable(0);
  if (m->locked==0) m->locked=1;
  else {
    push(CthSelf(), m->queue);
    CthSuspend();
  }
  CthSigYieldEnable(oldenable);
}
```

```
void MutexUnlock(mutex m)
{
  int oldenable
    = CthSigYieldEnable(0);
  if (empty(m->queue))
    m->locked=0;
  else {
    CthAwaken(pop(m->queue));
  }
  CthSigYieldEnable(oldenable);
}
```

Fig. 1. Locking and Unlocking a mutex using Converse primitives.

Note that the use of `CthSigYieldEnable` to achieve atomicity is only correct in situations where the mutex is controlling access to processor-private data (a `Cpv` variable). If the data were shared across the node (a `Csv` variable), one would need to add a test-and-set or other interprocessor synchronization directive to the mutex implementation. (Converse provides a fairly standard set of interprocessor synchronization facilities). Note that the use of interprocessor synchronization primitives would make the lock significantly more expensive, it is therefore advantageous to be able to use this less expensive single-processor lock where appropriate. In general, the ability to design synchronization mechanisms with as little or as much generality as one needs is likely to improve the efficiency of parallel languages implemented with Converse.

3 Converse Scheduling, Messaging, and other Facilities

The thread abstractions defined above could be demonstrated in the context of a single-language uniprocessor program. However, this would not adequately demonstrate their special properties. Instead, we demonstrate them in the context of a multi-language program with message-passing and other concurrency mechanisms. Therefore, we briefly describe the Converse scheduling and messaging subsystems [4], which together with the threads make up the heart of Converse.

The Converse scheduler is much like the scheduler for a normal threads package. However, in addition to regular threads, the Converse scheduler supports stackless threads. Their lack of their own stack makes them more efficient than regular threads, though it also means they cannot suspend or yield. Their high efficiency makes it possible to use Converse's prioritized ready-queue for very small, short tasks in situations where creating a true thread for the task would be too expensive. In fact, the Converse scheduler is hand-tuned specifically for this type of thread: the very short-lived, non-blocking, rapidly-generated threads that so commonly are created by parallel programs.

The Converse scheduler provides a task pool on each processor. It repeatedly selects tasks (either threads or stackless threads) from the ready-pool and allocates time segments to them. Unlike traditional multithreaded systems, time segments can either be of a fixed length (the thread is preempted), or can be of indefinite duration (the thread runs until it decides it is ready to yield). In the case of stackless threads, however, the thread's time segment lasts until the thread completes.

The Converse messaging system is integrated with the Converse scheduling facility: it makes it possible to insert a stackless thread into another processor's ready-queue. With the ability to insert work into each other's queues, processors can easily demand arbitrary work of each other – for example, they can PUT/GET each other's memory, they can simulate the reception of MPI messages, and so forth. Most of Converse's distributed operations are built on top of this basic primitive.

To use the Converse messaging system, one first builds a "message", which is essentially a stackless thread encoded in the form of a sequence of bytes.

Messages only contain a function pointer[1] and a block of data to be passed to the function. Allocating and building the message structure is done manually by the user. One then inserts the message into another processor's ready-queue with a function similar to this one:

`void CmiSyncSend(int destPE, int size, void *msg)`

Inserts a stackless thread (as represented by the message `msg` whose size is `size` bytes) into into the ready-queue of processor `destPE`. The "sync" in the name of this functions refers only to its buffer management policy, it does *not* wait for the data to be received, it only waits for it to be extracted from the message buffer. Therefore, it returns nearly immediately. The stackless thread has extremely high priority in the target processor's ready-queue.

There are several other send-functions in Converse. Some of them perform broadcast (insertion of a stackless thread into the ready-queues of all processors). Some use varying buffer-management policies which are more efficient than this one (Converse has a variety of buffer management policies designed to minimize overhead). Further information about Converse's messaging mechansisms and buffering policies can be found in [4].

The overall structure of a Converse program can be seen in figure 3.

```
/* the following code is executed by all processors */
void user_main(int argc, char **argv)
{
  ConverseInit(argv);  /* initialize Converse runtime */
  CreateWork();        /* create some threads */
  CsdScheduler(-1);    /* run until user termination */
  ConverseExit();      /* clean up and exit */
}
```

Fig. 2. The structure of a typical program using Converse

The function `CreateWork` would be user-written. It might use `CthCreate` and `CthAwaken` to create and start some standard threads. Alternatively, it might use the messaging routines described above to create some stackless threads and insert them into the scheduler's ready-queue.

The function `CsdScheduler` is provided by Converse: it chooses threads and stackless threads from the ready-queue and allocates time segments to them. It continues until the function `CsdExitScheduler` is called by one of the threads.

In addition to its scheduler, threads, and messaging, Converse contains a number of utility modules designed to facilitate the implementation of parallel

[1] actually, a network-transmittable encoding of a function pointer.

languages. A good example of such a module would be the Converse message manager, designed to facilitate the implementation of languages with tagged message transmission. This module provides a data type, the `CmmTable`, which can store data and index it according to a set of tags. One can insert data into the table along with a sequence of integer keys, and one can then retrieve the data by specifying the keys, possibly specifying some keys as wildcards. While it would be quite feasible for the Converse user to implement this himself, it is such frequently-needed functionality that it is convenient to provide this storage structure as a part of Converse.

Collectively, the subsystems in Converse seek to provide all the facilities needed to easily implement the runtime system for a parallel programming language. The facilities are designed for ease of use, for efficiency, and to handle the resource-arbitration issues created when multiple parallel languages must coexist. In the following sections, we demonstrate how Converse can be used to implement multithreaded languages quickly and efficiently.

4 Implementing Multithreaded Languages in Converse

The purpose of this section is to explore the design of a multithreaded parallel language, and determine how this language would be implemented in an interoperable way using the facilities provided by Converse.

An interesting style of parallel programming is the model where threads on different processors communicate by sending messages to each other. Chant[2] is an example of a prominent system that supports such a capability. In this section, we describe a simple "language" that supports threads sending messages to and receiving messages from each other. This language supports a subset of the features of Chant, and so is easier to describe here. We will show how our thread interface, in conjunction with the Converse framework, facilitates implementations of such languages. The language, which we call "simple messaging plus threads" or `SMT` for brevity, provides the following major functions:

`CthThread CsmStartThread(void (*function)(void *), void *arg)`

This call allocates a new thread, and enters it in the main scheduler's ready-queue. When scheduled, this thread will start executing the function pointed to by `function`, with a single parameter `arg`.

`void CsmTSend(int pe, int tag, char *buffer, int size)`

A message is sent to the given processor `pe` containing `size` bytes of data from `buffer`, and tagged with the given tag. The calling thread continues after depositing the message with the runtime system.

`int CsmTRecv(int tag, char *buffer, int size, int *rtag)`

Waits until a message with a matching tag is available, and copies it into the given buffer. A wildcard value, `SMTWildCard`, may be used for the tag. In this case, any available message is considered a matching message. The tag with which the message was sent is stored in the location to which `rtag` points. The number of bytes in the message is returned.

This language has been designed to test Converse, to determine whether or not it is an adequate vehicle for implementing multithreaded parallel languages such as Chant.

The SMT runtime has been implemented with the help of mechanisms and data structures provided by Converse. Figure 3 shows a snapshot of those data structures as they might exist on some processor of a running SMT program, with each data structure labeled according to the module that supports it. In this particular snapshot, threads 1, 3, and 4 are waiting for messages: they have inserted pointers to themselves into the table of waiting threads and have suspended. Threads 2 and 6 are waiting for the CPU, they wait in the scheduler's ready-set. Thread 5 must be the currently running thread on this processor, if (as the diagram suggests) it is not stored in any synchronization structure at all. Three messages have arrived and been stored for future receipt.

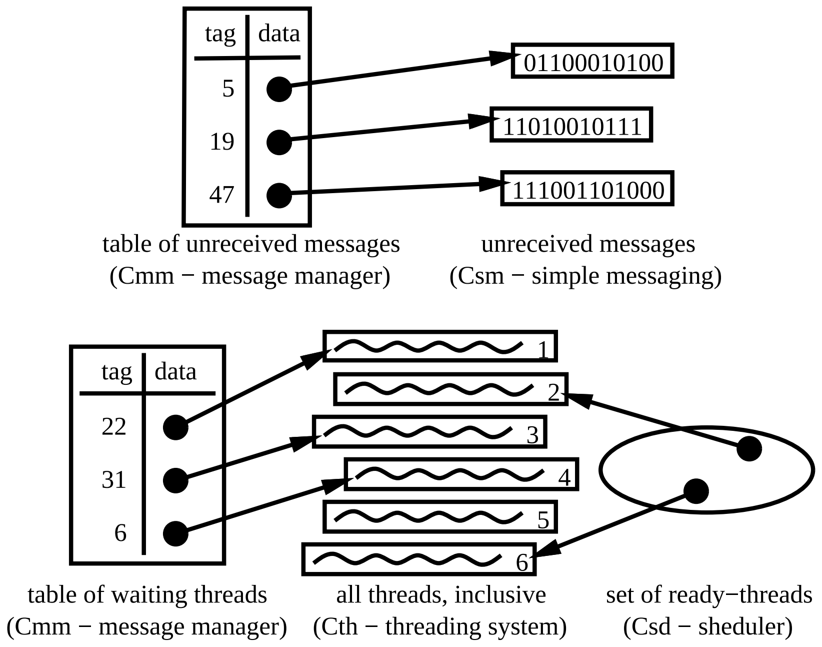

Fig. 3. Snapshot of data structures internal to Csm

The implementation of `CsmTSend` is as follows: it creates a stackless thread (also known as a Converse message), it stores the SMT message data and SMT tag in the stackless thread, and configures the thread to run an internal function `CsmTHandler`. It then inserts the stackless thread into the target processor's ready-set.

The stackless thread begins executing CsmTHandler on the target processor. CsmTHandler finds the SMT message data and SMT tag in its thread's data area, and inserts a pointer to the message data into the table of unreceived messages. It then checks to see if any Csm threads were waiting for the specified tag (by checking the table of waiting threads), and if so, it removes them from the table and awakens them. CsmTHandler returns, and the stackless thread is done.

CsmTRecv is implemented as follows: it first checks the table of unreceived messages, and if the message is already there, CsmTRecv extracts it and returns it. If not, it obtains its own thread identifier and inserts itself into the table of waiting threads. It then suspends. When it awakens, it knows that it has been awakened by CsmTHandler. It therefore extracts its message from the table of unreceived messages, and returns.

The CsmStartThread function is simply layered on top of CthCreate and CthSetStrategyDefault, it creates the thread and specifies that it should use the central Converse scheduler.

The entire code for this "runtime", which uses the message manager and the thread object, takes less than a hundred lines. The size of the runtime system would probably increase somewhat if an effort were made to optimize it, and a more sophisticated language such as Chant would require more code, but substantially less than a from-scratch implementation. The simplicity of the implementation demonstrates the success of our thread design.

Some features of Chant are not provided by SMT. For example, we have not mentioned remote thread creation. Adding such a feature can be achieved by adding a few more routines based on Converse features already described. We have also not mentioned direct thread-to-thread transmission. To provide thread-to-thread transmission, one need only make the thread identifier part of the message tag (this may require enlarging the tag). Expanding the SMT runtime to provide most of the features of the Chant runtime is straightforward.

The implementation techniques described in this section can be directly used to implement runtime systems for several thread-based languages including MultiLisp [3], Cid [10], and so forth. The ease of their implementation, using the Converse facilities, and the automatic interoperability provided by the framework, suggests that one will be able to run multilingual programs in near future, each module implemented using the language that suits its structure the best.

It is also easy to extend a language such as SMT to support prioritized threads. Of course, it is straightforward to have each thread run at a different priority level, by simply having it call CthSetPrio. More interestingly, we can make the priority of a thread depend on the priority of the message it is waiting for, or has just received. In such a configuration, each message will carry a priority on it, set by the sender. The message handler in the runtime, when it receives such a message, will set the priority of the waiting thread to that of the message before *awakening* it. If the message arrives before a thread starts waiting for it, the *receive* call will change the priority of the thread, and force it to yield, if necessary. Such message-induced priority will be useful, for example, to implement server threads whose priority can be dictated by its client.

5 Conclusions

We described the design and rationale of the Converse threads subsystem, a framework designed to facilitate the implementation of multithreaded parallel languages. The Converse threads subsystem is unique in that it contains support for interoperability among multiple parallel languages. The thread abstractions in Converse modularly separate the thread-scheduling functionality from the suspension and resumption mechanisms for threads. The facilities provided by the Converse threads subsystem include a highly portable thread object, a hierarchical scheduling model with a powerful top-level scheduler, routines for obtaining preemption or nonpreemption on a per-thread basis, and support for declaring thread-private variables as well as shared variables at multiple levels of sharing. The Converse threads subsystem, when integrated with the other subsystems in Converse, provides a complete framework for implementing interoperable multithreaded parallel languages. The rich set of flexible primitives ensures that a diverse set of languages, with widely varying (and possibly non-standard) thread behaviors, can be easily implemented using Converse. The comprehensive scheduling model ensures that such languages can interoperate in a single program. We explored the implementation of the runtime system for a small multithreaded language `SMT`. In doing so, we demonstrated the feasibility and simplicity of implementing multithreaded languages using the Converse threads subsystem and the remainder of the Converse framework.

The following programming models have been implemented with the help of Converse:

- Charm++ [5], based on remote method invocation,
- IMPORT [9], a successor to Modsim, using time-warp [1] methodology,
- SMT, a message-passing threads model,
- PVM [15], a simple but popular message-passing model
- DP [7], a data-parallel Fortran-based language

Of these languages, Charm++ uses stackless threads unless the programmer explicitly creates a `Cth` thread, `DP` and `IMPORT` use stackless threads exclusively, and `SMT` and `PVM` are based upon full-fledged threads. The fact that languages of such widely varying structure can coexist under a single runtime system indicates that the goals of interoperability and generality have been fairly well attained.

Immediate goals for the Converse threads subsystem include a detailed analysis of the performance of the thread subsystem, possibly optimized implementations for specific platforms, an implementation of MPI [8] that runs within a set of threads, and a runtime layer for the multithreaded Cid [10] language.

When these are complete, the future of this research involves porting the runtime systems of a wide range of parallel languages to Converse, thereby making it possible to use what were previously isolated languages as parts of a comprehensive multi-language programming system, where one may choose the "right language for the job" on a module-by-module basis. Pre-existing libraries written in different languages can then be used in a single program. The utility of such multi-lingual programming will then be systematically examined by developing several multi-lingual applications.

Converse, including its thread subsystem, is available from the Parallel Programming Laboratory at `http://charm.cs.uiuc.edu/`.

References

1. D. Ball and S. Hoyt. The Adaptive Time-Warp Concurrency Control Algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 174–177, 1990.
2. M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing 1994*, Nov 1994.
3. R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, October 1985.
4. L.V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *International Parallel Processing Symposium 1996 (to appear)*, 1996.
5. L.V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.
6. David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.
7. E. Kornkven and L.V. Kalé. Efficient Implementation of High Performance Fortran via Adaptive Scheduling – An Overview. In V. K. Prasanna, V. P. Bhatkar, L. M. Patnaik, and S. K. Tripathi, editors, *Proceedings of the 1st International Workshop on Parallel Processing.* Tata McGraw-Hill, New Delhi, India, December 1994.
8. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.
9. Vance P. Morrison. Import/dome language reference manual. Technical report, US. Army Corps of Engineering Research Laboratory, ASSET group., 1995.
10. Rishiyur S. Nikhil. Parallel Symbolic Computing in Cid. In *Parallel Symbolic Languages and Systems*, 1995.
11. John Plevyak, Vijay Karamcheti, Xingbin Zhang, and Andrew A. Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Supercomputing '95.*
12. PORTS- POrtable Runtime System consortium. The PORTS0 Interface. Technical report, Jan 1995.
13. POSIX System Application Program Interface: Threads Extension to C Language. Technical Report POSIX 1003.4a Draft 8, Available from the IEEE Standards Department.
14. M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *Proceedings of the Winter 1991 USENIX Conference.*
15. V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

This article was processed using the LaTeX macro package with LLNCS style