Implementation and Usage of a Thread Pool based on POSIX Threads

Ronald Kriemann
Max-Planck-Institute for Mathematics in the Sciences,
Inselstr. 22-26, D-04103 Leipzig, Germany.
rok@mis.mpg.de

October 19, 2004

Abstract

When working on a shared memory system the thread model is an easy and efficient way to change existing serial programs to make use of more than one processor. A widely used interface to threads are POSIX threads. Unfortunately, the interface is sometimes too difficult to use, especially for beginners. Therefore a wrapper is introduced which implements a pool of threads, where a job can be send to and is executed in parallel. This article describes the implementation and usage of this thread pool.

1 Introduction

Although distributed computing in the form of clusters gains more and more popularity, shared memory machines, especially with only a few number of processors, are still more often used as workstations or compute servers. In such a situation often the wish arises, that a serial program makes use of all processors in this machine.

Sometimes an easy way of doing this is using *threads*, which represent parallel computation paths in a single process. A widely adopted interface to threads are *POSIX Threads* or *Pthreads*. Most computer systems supply a software library which implements the functions of this interface.

Unfortunately the handling of threads is complicated and often distracts from the real problem to solve. Since most of the time, only a few of the Pthread functions are used, an easier interface is wanted. Also, because the creation of threads comes with some costs, using several hundreds of them might not be efficient.

One way out of this situation is a *thread pool*. A thread pool consists of a fixed number, usually the same as the processor number, of threads and delegates incoming jobs to these threads in an efficient and easy way.

This paper describes one way on how to implement such a thread pool and how to use it. Section 2 shows the implementation in the programming language C++. Typical examples for the usage are given in section 3.

A basic knowledge about C++ and POSIX threads is assumed. For a good reference about Pthreads see [1].

The complete source code for the described implementation is freely available from [4].

2 Implementation

A thread pool as implemented below was designed to allow at most p jobs to work in parallel. But it should also be able to handle more than p requests.

By implementing these requirements, the thread pool is capable of working in two different kinds. In the first model, at most p jobs are started simultaneously, working in parallel and are finally synchronised at the same time by the calling thread. This working mode is usable if the load is already balanced between the processors and the thread pool is only used to simplify the starting of the threads.

In the other mode an arbitrary number of jobs are given to the thread pool at arbitrary times and are synchronised by the calling thread at the same time. This kind of programming is especially helpful, if no load balancing is done, but the number of requests is much bigger than p and the work per request is more or less the same.

The implementation is done in C++ and is inspired by the thread classes defined in the *Java* programming language (see e.g. [3]).

2.1 Thread Class

The class `TThread` is implemented as a wrapper around the Pthread functions. The interface is very similar to the thread classes in Java.

The member variables include the Pthread ID and attribute (`_thread_id` and `_thread_attr`), a boolean variable indicating whether the thread is running or not (`_running`) and an integer to store a given thread number (`_thread_no`). This thread number can be used to identify the local processor in the parallel algorithms.

The member functions of `TThread` include methods to access the local data and start or stop the thread. The special function `run()` is abstract.

It must be overloaded in derived classes and is called when the thread is running.

The whole interface of `TThread` is listed in the following source.

```
class TThread {
protected:
  pthread_t       _thread_id;    // ID of the thread
  pthread_attr_t  _thread_attr;  // thread attributes
  bool            _running;      // true if thread is running
  int             _thread_no;    // opt. number of thread
public:
  // constructor and destructor
  TThread ( int p = -1 );
  virtual ~TThread ();
  // access local data
  int  thread_no     () const  { return _thread_no; }
  void set_thread_no ( int p ) { _thread_no = p; }
  bool on_proc       ( int p ) const;
  // user interface
  virtual void run   () = 0;
  virtual void start ( bool d = false,
                       bool s = false ) { create( d, s ); }
  virtual void stop  () { cancel(); }
  // thread management
  void create ( bool d = false, bool s = false );
  void join   ();
  void cancel ();
};
```

The default thread number of -1 can be used, if the algorithm needs no identification of the local processor. If `_thread_no` is assign to this value, the method `on_proc(int)` always returns `true`. In any other case `on_proc(int)` only returns true if the given integer equals the local thread number or the argument is -1.

A real POSIX thread is created by a call to `create(bool,bool)` or, indirectly by `start(bool,bool)`. The first boolean argument to both functions determines, whether the thread should be started in a *joinable* or *detached* state. If the thread is in the joinable state, other threads can synchronise with the termination of the thread and gather informations about the termination status. If the thread is detached, all resources of the thread are freed after the termination. It is therefore not possible to join with such a thread. By default all threads start in the joinable state.

The second argument to `create(bool,bool)` and `start(bool,bool)`

sets the *contention scope* of the thread. The default is *process contention scope*. On most systems, such threads are cheaper to create, because they compete for resources (e.g. processor time) only among themselves. If the argument is `false`, the scope is set to *system contention scope*, where the threads compete against all threads in the system. Although more costly when creating a thread, system contention scope usually results in a more predictable behaviour of the runtime and a better utilisation of system resources.

A termination request to finish the execution can be send to the thread by the method `cancel()`. After the termination by `cancel()` the thread is still join-able if it was created in this state.

There are several other functions defined in the Pthread interface, like `pthread_exit` or `pthread_kill`. Usually their functionality is not needed, especially not in the thread pool, and therefore they are not included in the thread class.

The following listing shows the implementation of the constructor, the destructor and the method `on_proc(int)`.

```
TThread::TThread ( int p )
        : _running(false), _thread_no(p)
{}
TThread::~TThread ()
{  if ( _running )
      cancel();
}
bool TThread::on_proc ( int p ) const
{  if ((p == -1) || (_thread_no == -1) || (p == _thread_no))
      return true;
    else
      return false
}
```

The method `create(bool,bool)` first checks, if the thread is already running. After this, the attributes of the thread are initialised and, optionally, the detached state and system contention scope are set. Finally the POSIX thread is created.

```
void
TThread::create ( bool detached, bool sscope )
{
  if ( ! _running )
  {
```

```
    int status;
    pthread_attr_init( & _thread_attr );
    if ( detached )
      pthread_attr_setdetachstate( & _thread_attr,
                                   PTHREAD_CREATE_DETACHED );
    if ( sscope )
      pthread_attr_setscope( & _thread_attr,
                             PTHREAD_SCOPE_SYSTEM );
    pthread_create( & _thread_id, & _thread_attr,
                    _run_thread, this );
    _running = true;
  }
  else
    printf( ``ERROR : thread is already running\n'' );
}
```

Argument 3 of the function `pthread_create` is not the member function `run()` but an external, non-virtual function. Because of the way a virtual method is called in C++, it is not possible to obtain a pointer to such a function. Instead a pointer to the function `_run_thread(void*)` is supplied which does nothing but call the member function `run()`:

```
extern ``C'' void *
_run_thread ( void * arg )
{  if (arg != NULL)
     ((TThread*) arg)->run();
   return NULL;
}
```

The implementation of the methods `join()` and `cancel()` is straightforward: after ensuring, that a thread is running, the appropriate Pthread function is called.

2.2 Classes for Mutices and Condition Variables

Two other ingredients are very important when working with threads: *mutices* and *condition variables*.

A mutex allows the synchronisation between different threads, especially important if several threads write to the same memory area.

The following listing shows a simple wrapper around the Pthread functions for mutices.

```
class TMutex
```

```
{
  friend class TCondition;
protected:
  // the mutex itself and the mutex attribute
  pthread_mutex_t     _mutex;
  pthread_mutexattr_t _mutex_attr;
public:
  TMutex ()
  { pthread_mutexattr_init( & _mutex_attr );
    pthread_mutex_init( & _mutex, & _mutex_attr );
  }
  ~TMutex ()
  { pthread_mutex_destroy( & _mutex );
    pthread_mutexattr_destroy( & _mutex_attr );
  }
  // lock and unlock mutex (return 0 on success)
  int lock   () { return pthread_mutex_lock(   & _mutex ); }
  int unlock () { return pthread_mutex_unlock( & _mutex ); }
  // try a lock (return 0 if lock was successful)
  int trylock () { return pthread_mutex_trylock( & _mutex ); }
};
```

A very convenient way to have some kind of signalling between threads are condition variables. They are using mutices, which is the reason for the `friend` declaration inside the `TMutex` class.

```
class TCondition
{
protected:
  // the Pthread condition variable
  pthread_cond_t  _cond;
public:
  // constructor and destructor
  TCondition  () { pthread_cond_init( & _cond, NULL ); }
  ~TCondition () { pthread_cond_destroy( & _cond ); }
  // condition variable related methods
  void wait ( TMutex & m )
  { pthread_cond_wait( & _cond, & m._mutex ); }
  void signal ()
  { pthread_cond_signal( & _cond ); }
  void broadcast ()
  { pthread_cond_broadcast( & _cond ); }
};
```

Condition variables in the Pthread implementation can also be initialised with attributes, but usually the standard attributes are sufficient and therefore the are not included in the wrapper.

2.3 Thread Pool Class

The thread pool works by starting p local threads during the initialisation of the pool. These threads are blocked, until a job is given to the thread pool. This job is then associated to one of these threads which executes the job. After finishing, the thread is again blocked until the next job arrives.

If no free thread is available for an incoming job, the calling thread blocks, until a local thread of the pool has finished and is idle.

2.3.1 Jobs for the Thread Pool

The implementation starts with a class for a job, which can be handled by the pool. This class is similar to the thread class above, but should not be a thread itself. Instead it just has to hold some informations for the thread pool.

```
class TJob
{
protected:
  // number of processor this job was assigned to
  int         _job_no;
  // associated thread in thread pool
  TPoolThr  * _pool_thr;
public:
  // constructor
  TJob ( int p ) : _job_no(p), _pool_thr(NULL) {}
  // running method
  virtual void run ( void * ptr ) = 0;
  // access local data
  int    job_no       () const { return _job_no; }
  TPoolThr * pool_thr ()       { return _pool_thr; }
  void set_pool_thr ( TPoolThr * t ) { _pool_thr = t; }
  // compare if given processor is local
  bool on_proc ( int p ) const;
};
```

The variable `_job_no` and the method `on_proc(int)` have the same purpose as `_thread_no` and `on_proc(int)` in the `TThread` class and the implementation is identical.

For synchronisation purposes, which become clear later, a pointer (`_pool_thr`) to the associated thread in the thread pool is kept within each job object.

As in the class for a thread, the function `run(void*)` has to be overwritten by a derived class and should contain the real computation to be performed.

2.3.2 A Thread in a Thread Pool

The class `TPoolThr` is derived from `TThread` and contains mutices and condition variables for the synchronisation with the thread pool. Also a pointer to the actual pool it belongs to is saved.

```
class TPoolThr : public TThread
{
protected:
  // thread pool we belong to
  TThreadPool    * _pool;
  // job to run and data for the job
  TJob           * _job;
  void           * _data_ptr;
  // condition and mutex for waiting for job
  // and pool synchronisation
  TCondition       _work_cond,  _sync_cond;
  TMutex           _work_mutex, _sync_mutex;
  // indicates work-in-progress, end-of-thread
  // and a mutex for the local variables
  bool             _work, _end;
  TMutex           _var_mutex;
  // mutex for synchronisation with destructor
  TMutex           _del_mutex;
  // should the job be deleted upon completion
  bool             _del_job;
public:
  // constructor and destructor
  TPoolThr ( int n, TThreadPool * p );
  // running method
  void run ();
  // access local variables
  void set_end  ( bool f );
  void set_work ( bool f );
  void set_job  ( TJob * j, void * p, bool del );
  bool is_working () const { return _job != NULL; }
  TJob        * job        () { return _job; }
  TCondition  & work_cond  () { return _work_cond; }
```

```
  TMutex      & work_mutex () { return _work_mutex; }
  TCondition  & sync_cond  () { return _sync_cond; }
  TMutex      & sync_mutex () { return _sync_mutex; }
  TMutex      & del_mutex  () { return _del_mutex; }
};
```

All changes to the local variables, e.g. done by `set_end(bool)`, `set_work(bool)` and `set_job(TJob*,void*)`, are guarded by `_var_mutex`.

The function `run()` is implemented as an infinite loop, waiting for a job to arrive and executing it. To prevent the destruction of the object before the thread has finished its execution, the mutex `_del_mutex` is used for the synchronisation of the destruction of the thread object.

```
void
TThreadPool::TPoolThr::run ()
{
  _del_mutex.lock();
  while ( ! _end )
  {
    // wait for work
    _work_mutex.lock();
    while ((_job == NULL) && ! _end )
      _work_cond.wait( _work_mutex );
    _work_mutex.unlock();
    // check again if job is set and execute it
    if ( _job != NULL )
    {
      _job->run( _data_ptr );
      // detach thread from job
      _job->set_pool_thr( NULL );
      if ( _del_job )
          delete _job;
      set_job( NULL, NULL );
      _sync_mutex.unlock();
    }
    // append thread to idle list
    _pool->append_idle( this );
  }
  _del_mutex.unlock();
}
```

The condition variable is guarded by the associated mutex and the predicate (`_job == NULL`) is checked twice before assuming it valid. After the

execution of the job, the connection between the thread and the job is reset. That way, any synchronisation with the job can be skipped. Finally the thread is inserted into the list of idle threads of the thread pool.

2.3.3 The Thread Pool Class

In the constructor of the thread pool, p threads of type `TPoolThr` are created and p is saved in a local variable (`_max_parallel`). Because all threads in the pool are only created once, system contention scope (see 2.1) is chosen for a better system utilisation. The threads are stored in an array and also, because they are immediately blocked, in a list holding the idle threads (`_idle_threads`).

`TArray` and `TSLL` are classes representing dynamic arrays and single-linked lists. The implementation of these classes is omitted in this paper.

The complete definition of `TThreadPool` is shown in the next listing:

```
class TThreadPool
{
public:
  class TJob      ... ;
  class TPoolThr  ... ;
protected:
  // maximum degree of parallelism
  uint                  _max_parallel;
  // array of threads, handled by pool
  TArray< TPoolThr * >  _threads;
  // list of idle threads, mutices and condition for it
  TSLL< TPoolThr * >    _idle_threads;
  TMutex                _idle_mutex, _list_mutex;
  TCondition            _idle_cond;
public:
  // constructor and destructor
  TThreadPool ( uint max_p );
  ~TThreadPool ();
  // access local variables
  uint max_parallel () const { return _max_parallel; }
  // run, stop and synchronise with job
  void  run  ( TJob * job, void * ptr = NULL, bool del = false );
  void  sync ( TJob * job );
  void  sync_all ();
  // return idle thread form pool
  TPoolThr * get_idle ();
  // insert idle thread into pool
  void append_idle ( TPoolThr * t );
```

```
};
```

In the first section of `TThreadPool` are the definitions for the above described classes `TJob` and `TPoolThread`, which are subclasses of `TThreadPool`.

The implementation of the constructor is as described above. Optionally a call to `pthread_setconcurrency` might follow at the end of the constructor to tell the Pthread implementation, how many concurrent threads to expect. The destructor starts by synchronising with all threads. After that, each thread is terminated by setting the variable `_end` in the pool-thread and signalling the termination. Finally, after ensuring that each thread has finished by locking `_del_mutex`, all resources are freed.

```
TThreadPool::TThreadPool ( uint p )
{
    _max_parallel = p;
    _threads.set_size( p );
    for ( uint i = 0; i < p; i++ )
    {
        _threads[i] = new TPoolThr( i, this );
        _idle_threads.append( _threads[i] );
        _threads[i]->start( true, true );
    }
    // pthread_setconcurrency( p + pthread_getconcurrency() );
}
TThreadPool::~TThreadPool ()
{
    sync_all();
    for ( uint i = 0; i < _max_parallel; i++ )
    {
        _threads[i]->sync_mutex().lock();
        _threads[i]->set_end( true );
        _threads[i]->set_job( NULL, NULL );
        _threads[i]->work_mutex().lock();
        _threads[i]->work_cond().signal();
        _threads[i]->work_mutex().unlock();
        _threads[i]->sync_mutex().unlock();
    }
    // cancel still pending threads and delete them all
    for ( uint i = 0; i < _max_parallel; i++ )
    {
        _threads[i]->del_mutex().lock();
        delete _threads[i];
    }
```

```
}
```

The method `run(TJob*,void*)` takes a given job with an optional argument, looks for an idle job, attaches the job to the thread and signals the thread to begin the execution.

```
void
TThreadPool::run ( TThreadPool::TJob * job, void * ptr, bool del )
  TPoolThr * t = get_idle();
  // and start the job
  t->sync_mutex().lock();
  t->set_job( job, ptr, del );
  // attach thread to job
  job->set_pool_thr( t );
  t->work_mutex().lock();
  t->work_cond().signal();
  t->work_mutex().unlock();
```

To synchronise with the termination of a job, `sync(TJob*)` is used. This function uses the pointer to the pool-thread, stored in the `TJob` object. If this pointer exists, the synchronisation is done with this thread. In case the pointer is `NULL`, e.g. the thread finished before calling `sync(TJob*)`, the method returns. Finally the connection between the job and the pool-thread is nullified.

```
void
TThreadPool::sync ( TJob * job )
  if ( job == NULL )
    return;
  TPoolThr  * t = job->pool_thr();
  // check if job is already released
  if ( t == NULL )
      return;
  // look if thread is working and wait for signal
  t->sync_mutex().lock();
  t->set_job( NULL, NULL );
  t->sync_mutex().unlock();
  // detach job and thread
  job->set_pool_thr( NULL );
```

To synchronise with all running threads, `sync_all()` was implemented. Instead of using the thread pointer in the job objects, it uses the pointer to the threads, stored in the thread pool directly. To not block, when checking

if a thread is running, the function first tries to block, if this is successful, the thread is already idle, otherwise `sync_all()` waits for the thread to unlock the mutex.

```
void
TThreadPool::sync_all ()
  for ( uint i = 0; i < _max_parallel; i++ )

    if ( _threads[i]->sync_mutex().trylock() )
      _threads[i]->sync_mutex().lock();
    _threads[i]->sync_mutex().unlock();
```

As mentioned above, the thread pool blocks if no idle thread is available to execute the incoming job. This is done by the function `get_idle()`. It blocks, until the list of idle threads (`_idle_list` is non-empty and returns the first thread in that list. All accesses to the list are guarded by a mutex.

```
TThreadPool::TPoolThr *
TThreadPool::get_idle ()
{
    while ( true )
    {
        // wait for an idle thread
        _idle_mutex.lock();
        while ( _idle_threads.size() == 0 )
            _idle_cond.wait( _idle_mutex );
        _idle_mutex.unlock();
        // get first idle thread
        _list_mutex.lock();
        if ( _idle_threads.size() > 0 )
        {
            TPoolThr * t = _idle_threads.behead();
            _list_mutex.unlock();
            return t;
        }
        _list_mutex.unlock();
    }
}
```

The condition variable used in `get_idle()` is signalled by the method `append_idle(TPoolThr*)` which inserts an idle thread into the list of the pool. Before this, the list is tested, whether the given thread is already

inside or not. This is mainly for safety reasons. After the thread is inserted, a signal is send to the condition variable `_idle_cond`, which wakes up any blocking threads waiting for it.

```
void
TThreadPool::append_idle ( TThreadPool::TPoolThr * t )
{
  _list_mutex.lock();
  // check if given thread is already in list
  // and only append if not so
  TSLL< TPoolThr * >::TIterator  iter = _idle_threads.first();
  while ( ! iter.eol() )
  {
    if ( iter() == t )
    {
      _list_mutex.unlock();
      return;
    }
    ++iter;
  }
  _idle_threads.append( t );
  _list_mutex.unlock();
  _idle_mutex.lock();
  _idle_cond.broadcast();
  _idle_mutex.unlock();
}
```

3 Usage

As mentioned is section 2, the thread pool was implemented with two different kinds of usage in mind. The first case where the number of subproblems n is much larger than the number of processors p and the costs between the subproblems only vary little, *list scheduling* (see [2]) can be used for load balancing.

List scheduling works by assigning the next available, not executed job to the first free processor (or thread). This kind of scheduling is in general not optimal, e.g. it approximates the time achieved by the best possible scheduling by a factor of up to $2 - \frac{2}{n-1}$. But this worst case only occurs, if a small number of very costly jobs exists which are executed last. If the costs per job do not differ much, as assumed above, list scheduling works quite well.

The second method is based on load balancing done by the user and only helps to simplify the parallel execution. Using this strategy n should be in the order of p.

In both cases, the parallel work has to be put into an object of type `TJob`. For this a new class must be derived and the `run` method has to by overloaded.

```
class TMyJob : public TThreadPool::TJob
{
public:
  TMyJob ( int p ) : TThreadPool::TJob(p)
  void run ( void * arg )
  { // do something }
};
```

The overhead involved in using the thread pool on different hardware and software systems is shown in the following table. In this benchmark 1 000 000 jobs are created and executed, while each job immediately returns from the `run()` method. The same is repeated using threads with process (proc) and system (sys) contention scope.

System	thread pool	Thread (proc)	Thread (sys)
Linux	20.7 s	76.8 s	77.0 s
Solaris 7	79.7 s	142.3 s	249.2 s
Solaris 9	49.1 s	100.4 s	100.3 s
HP-UX	76.8 s	192.8 s	194.4 s
IBM AIX	29.8 s	56.4 s	64.6 s

The advantage of the thread pool is obvious, with a factor between 3.7 (Linux) and 1.8 (Solaris 7) better than the execution times of Pthreads alone. A penality in the creation time of a thread with system contention scope is only visible in the Solaris 7 operating system. On the other systems either there is no increased costs in using system contention scope, or the Pthread implementation does not distinguish between both kind of contention scoping.

Typical examples for the above described programming paradigms are presented in the following sections.

3.1 Automatic Load Balancing

The example presented in this section uses a recursive function where the computations are done in the last level (e.g. the leafs) of the recursion. All

computations are assumed to be independent.

```
TThreadPool  * thread_pool;
void f ( int l )
{
  if ( l == 0 )
    thread_pool->run( new TMyJob() );
  else
  { f( l-1 ); f( l-1 ); }
}
void
main ()
{
  thread_pool = new TThreadPool( p );
  f( max_depth );
  thread_pool->sync_all();
}
```

The function creates new jobs until no idle threads remain in the thread pool. Any new job blocks until a previous one has finished. After all jobs have been given to the thread pool, the function returns and waits for the termination of the running threads.

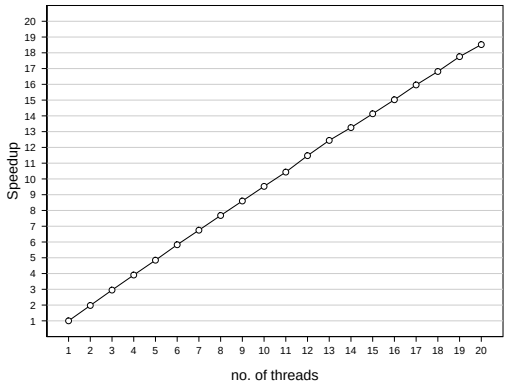

Figure 1: Speedup with List Scheduling

In figure 1 the speedup for an example using the described strategy is given. A total of 256 jobs is executed by an increasing number of threads. The work in each job varies randomly, whereat the minimal and maximal costs differ by a factor of 1.5. As can be seen, the speedup is almost optimal, indicating the good efficiency of list scheduling in such a case.

3.2 User Defined Load Balancing

In the following a load balancing is assumed, which distributes the work over all p processors. Then p jobs have to be created and given to the thread pool.

```
TThreadPool    thread_pool( p );
TMyJob      ** jobs = new TMyJob* [p];

for ( int i = 0; i < p; i++ ) jobs[i] = new TMyJob( i );
for ( int i = 0; i < p; i++ ) thread_pool.run( jobs[i] );
for ( int i = 0; i < p; i++ ) thread_pool.sync( jobs[i] );
for ( int i = 0; i < p; i++ ) delete jobs[i];
delete[] jobs;
```

The number of the local processor was assigned to each thread to allow the parallel routines to identify each thread. If this is not necessary, it can be omitted.

References

[1] D.R. Butenhof: *Programming with POSIX Threads*, Addison-Wesley, 1997.

[2] R.L. Graham: *Bounds on Multiprocessing Timing Anomalies*, SIAM Journal of Applied Mathematics, Volume 17(2), 1969, pp. 416-429.

[3] P. Hyde: *Java Thread Programming*, SAMS, 2001.

[4] *Max Planck Institute for Mathematics in the Sciences*, Scientific Computing, Internet-address: http://www.mis.mpg.de/scicomp/