The ADMB pthread_manager Class

svn+ssh://admb-project.org/branches/threaded2

http://www.admb-project.org/svn/branches/threaded2/

The ADMB pthread_manager class was introduced in the June 2013 ADMB Developer Workshop in Seattle. At that time, the performance benefits of using the pthreads were unclear. The following examples, show that multi-threading can substantially decrease convergence time in some models. Two of the examples are based on the examples in the original ADMB manual. A third example demonstrates passing data among "slave" threads. The final example demonstrates application of the pthread_manager class to the tagest diffusion model, and adamply of inserting the class in some terrible old legacy code.

All examples were tested on two different architectures: Xeon x5690 @3.47GHz x 12; 11.7GiB RAM and Core i7-3630QM CPU @ 2.40GHz × 8; 15.6 GiB RAM

If anyone is interested they could do some more testing after checking out the threaded2 branch from the subversion repository (svn+ssh://admb-project.org/branches/threaded2, for now).

The "multisimple" example revisited

The folder `examples/threaded/multisimple` contains two revised `.tpl` files, both derived from the classic admb `simple.tpl` example. `msimple.tpl` is linear regression on a largish (prime) number of data points (1000003) chopped into segments so that sums of squares contribution for each segment are computed on separate threads independently of other segments. The sums of squares from each thread are summed to compute the likelihood. `msimple-nothreads.tpl` performs the same regression without any threading. The purposes of this example are to illustrate how one might go about adapting a model for multithreading and to examine the scaling properties of the pthread_manager class. The revisions to the `msimple.tpl` since the Seattle meeting were provoked by reading an interesting piece in Dr. Dobbs, but actually have little to do with "false sharing".

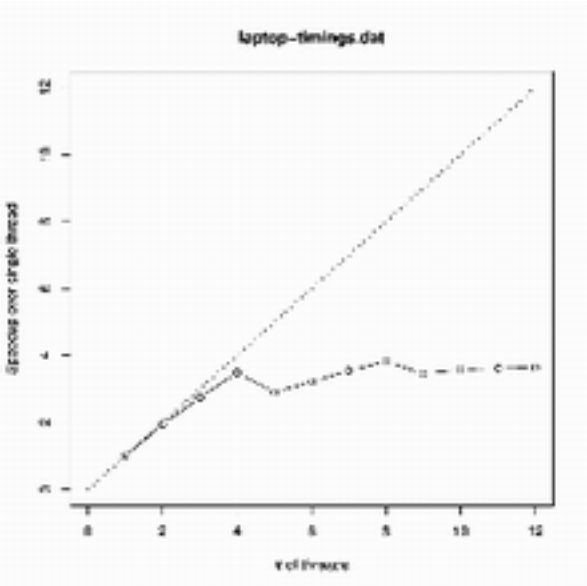
Timing

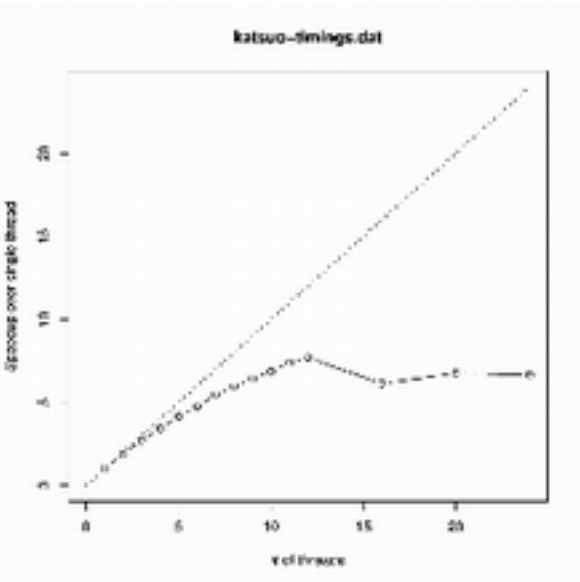
Here are estimates of time to convergence of msimple running on a Core i7 laptop. Seconds refers to the "Real" time returned from the linux time utility.

Threads	Seconds	Final F	max|G|	Fn Evals
1	6.88	8.8535e+06	-5.0337e-05	24
2	3.56	8.8535e+06	-5.0337e-05	24
3	2.51	8.8535e+06	-5.0353e-05	24
4	1.97	8.8535e+06	-5.0337e-05	24
5	2.37	8.8535e+06	-5.0352e-05	24
6	2.14	8.8535e+06	-5.0352e-05	24
7	1.94	8.8535e+06	-5.0337e-05	24
8	1.80	8.8535e+06	-5.0338e-05	24
9	1.99	8.8535e+06	-5.0335e-05	24
10	1.92	8.8535e+06	-5.0337e-05	24
11	1.90	8.8535e+06	-5.0337e-05	24
12	1.89	8.8535e+06	-5.0335e-05	24
50	1.82	8.8535e+06	-5.0337e-05	24

Performance Scaling

The following graphs show the speedup running on different numbers of threads relative to running on a one thread on two different architectures. The speedup relative to using a single thread is roughly proportional to the number of threads. The dotted line shows theoretical linear scaling. Scaling is almost linear when the number of threads does not exceed the number of CPUs available.





Speedup is roughly proportional to the number of threads up to the point where the number of threads exceeds the number of processors. After that point the speedup is more or less constant. It is puzzling why the initial speed up is not strictly linear. It could be due to the thread scheduling policies that Matthew pointed out in Seattle or the perhaps overhead of pthread creation. The Corei7 architecture consists of 4 dual core processors, so it is also puzzling why the speedup stops at 4 and not 8 threads. (Is the Xeon a dual core processor?)

The thread scheduling policy notion was explored in a cursory way. Setting the scheduling policy to SCHED_FIFO in **`void adpthread_manager::create_all(void * ptr)`** caused the program to hang, probably because the "slave" threads blocked the "master". There is more to this than meets the eye.

Over-fitting Distraction

During preliminary explorations of the scaling properties of this model, it was noticed that the time to convergence increased for certain numbers of threads and attributed it to some inexplicable instance of false sharing. Closer examination revealed that the numbers of function evaluations were larger in these instances. Further the derivatives at convergence were very small, ~10^{-6} and the derivative checker showed the derivatives to be wrong. The data used in the msimple model are generated by a linear regressions with normally distributed observation errors, exactly the model being fit. This appears to be a case of "over-fitting" whereby the minimizer attempts to make the derivative really small by doing more function evaluations. The errors reported by derivative checker are possibly caused by roundoff in the finite difference approximation. The convergence criterion was relaxed to 10^{-3} (from the default 10^{-4}) to obtain the above results.

Funnels and threads - the forest example

Chapter 3 of the ADMB manual introduces the notion of a "funnel" to reduce the size of the temporary storage buffers. The computations in one pass through the funnel do not depend on the results of previous passes through the funnel. Code that can be executed using a funnel is therefore a good candidate for implementing on threads. Directory `examples/threaded/mforest` contains `mforest.tpl` which is a multithreaded implementation of the forestry model described in Chapter 3.

The example also illustrates the use of a specialized class to encapsulate variables needed to run the model as discussed in Seattle. Each thread creates a unique instance of this class. In the original `forest.tpl` example, the functions used in the numerical integration, `dvariable trapzd(...)` and `dvariable adromb(...)` were included in the in the `function_minimzer` class so that they were be inherited by the `model_data` class defined by the tpl. For this example, these functions were moved to the `thread_funnel` class to eliminate the possibility of false sharing of data (particularly the static variables in `trapzd)` and functions. It also avoids the need to pass a pointer to function and the necessity to work around C++ rules about taking the address of a class member.

This table below shows the time in seconds to reach convergence as reported by the "REAL" field of the unix time command.

Architecture	Unthreaded	Threaded	Speedup
Core i7	2.102	0.605	3.5
Xeon	2.725	0.611	4.5

The model converged to the same function value after 37 function evaluations for both the threaded and unthreaded. Speed-up is 3 to 4 fold on both architectures tested.

newertpl

This example demonstrates passing data among "slave" threads and the use of thread groups to designate thread numbers for data passing. This code is based on something developed to handle tag cohorts in MFCL and resulted in a 2x speedup. Further documentation might be forthcoming (or not).

Tuna tagging diffusion model - tagest

Three versions of the tagest code were developed. The “Benchmark” is a slightly simplified version of the standard (`svn+ssh://katsuo/movemod/25`) code. This code uses and advection-diffusion-reaction model to predict the density of tagged fish over time on a finite difference grid. Parameters are estimated using ADMB to minimize a Poisson log-likelihood function of observed and predicted tag recaptures. Each monthly cohort of tagged fish (all the fish tagged and released in one month) is assumed tb independent of all other tagged fish. Since the cohorts are independent, a **`funnel_dvariable`** object is to compute the likelihood for each cohort.
The “Threaded” code aggregates all of the computations into a single function - including computation of movement and mortality fields from model parameters and intermediate tridiagonal matrices for solving the PDE. This function is implemented as member of specialized class and invoked by the task running on the thread.

The standard model domain is a 95 x 40 finite difference grid with 1 degree spatial steps. The PDE is solved with ¼ month time steps. Twentyeight cohorts were followed for 24 months, and there are 76 active parameters. The value of the likelihood function and the maximum gradient for the three code versions werw the same after 100 function evaluations.

The following table shows the time in seconds to compute 100 function evaluations of the three code on two different architectures as reported by the “REAL” field of the unix time command.

Code Version	Xeon	Core i7
Benchmark	585.093	502.694
Threaded (28 threads)	161.542	382.267
No-threads	1501.985	1346.751
Speedup relative to benchmark	3.6	1.3
Speedup relative to no-threads	9.3	3.5

The speedup relative to the no-threads code is deceptively impressive on the Xeon, but the no-threads version is considerably slower than the benchmark. The speedup relative to the benchmark is the comparison of interest and depends critically on the number of processors, 3.6 for 12 core for the Xeon and 1.3 for for the four core Corei7

A more demanding test is to use a higher resolution model domain, a 190 x 80 finite difference grid with 1/2 degree spatial steps. At this spatial resolution, PDE should be solved with smaller time steps to avoid numerical instability. Preliminary tests with 1/12 month time step following 28 cohorts for 24 months caused both architectures to run out of RAM and to begin to use swap memory. The following table shows results for 100 function evaluations with ⅙ month time steps following cohorts for 12 months (not the most stable solution). Unfortunately this configuration eats up all the RAM on the Xeon machine.

Code Version	Xeon	Core i7
Benchmark	NA	1437.069
Threaded (28 threads)	NA	1200.664
No-threads	NA	3969.218
Speedup relative to benchmark	NA	1.2
Speedup relative to no-threads	NA	3.3

More RAM is on order for the Xeon machine.

Using legacy classes

The code listing below is the function that runs on the threads in the diffusion model. It is similar in structure to the code used in the mforest example. The primary parameter class, **par_t_reg,** is successively derived from two base classes, contains AUTODIF container classes for both constant and variable objects, contains instances of other specialized C++ classes, and even uses templates to distinguish variable and constant objects (bad idea). These features make it a pretty good test case for running class members on threads. C++ assumes that all classes have a default constructor (if not it will invent one). All AUTODIF default constructors create instances of themselves, but do not allocate memory for data. All AUTODIF assignment operators check to see if the object on the left of the operator has allocated memory. If not, memory is allocated to exactly correspond to the object on the right of the operator. Once memory is allocated, the data are copied to the object on the left. This action ensures a "deep copy" of the object on the right. The pthread_manager class uses this feature to ensure that threads do not inadvertently share data.

```
#include <admodel.h>
#include <adthread.h>
#include "par_t_reg.h"

void mp_computeCohortLikelihood(void* ptr)
{
   // cast the data pointer to the proper type
   new_thread_data * tptr = (new_thread_data *) ptr;

   // set up the gradient stack for the thread
   gradient_structure::set_CMPDIF_BUFFER_SIZE(  150000000L);
   gradient_structure::set_GRADSTACK_BUFFER_SIZE(12550000L);
   gradient_structure::set_MAX_NVAR_OFFSET(1000);
   gradient_structure gs(20000000);

   // get the thread number
   // assumed to be equal the the cohort number
   const int sno=tptr->thread_no;

   // need to get this set first so that it knows which buffer to
```

```
   // use for this instance
   ad_comm::pthread_manager->set_slave_number(sno);

   // use default constructor to create instance of par_t_reg class
   // with unallocated and unitialized class members
   par_t_reg param;

   // allocate memory and values for constant class members
   // from master (thread 0) using the constant data buffer
   // and allocate memory for variable class members
   param.get_constant_data(0);

   do
   {
      ad_comm::pthread_manager->cread_lock_buffer(0);
      int lflag=ad_comm::pthread_manager->get_int(0);
      ad_comm::pthread_manager->cread_unlock_buffer(0);
      if (lflag==0)
          break;

      // get values of variable class members from the master thread
      // for this function evaluation
      param.get_variable_data(0);

      // get dates over which to compute likelihood from the constant buffer
      ad_comm::pthread_manager->cread_lock_buffer(0);
      int year = ad_comm::pthread_manager-> get_int(0);
      int month = ad_comm::pthread_manager-> get_int(0);
      year_month local_start_date(year,month);
      year = ad_comm::pthread_manager-> get_int(0);
      month = ad_comm::pthread_manager-> get_int(0);
      ad_comm::pthread_manager->cread_unlock_buffer(0);
      year_month final_date(year,month);


      // get initial tag density matrix from master using the variable buffer
      ad_comm::pthread_manager->read_lock_buffer(0);
      dvar_matrix release = ad_comm::pthread_manager->get_dvar_matrix(0);
      ad_comm::pthread_manager->read_unlock_buffer(0);

      // use par_t_reg class member to compute likelihood for this cohort
      int cohort=sno;
      dvariable like = param.computeCohortLikelihood(local_start_date,
                                                     final_date, release,
cohort);

      // return likelihood to master
      ad_comm::pthread_manager->write_lock_buffer(0);
      ad_comm::pthread_manager->send_dvariable(like, 0);
      ad_comm::pthread_manager->write_unlock_buffer(0);

      // compute derivative contribution for this thread
      slave_gradcalc();
   }
   while(1);
```

```
  // terminate this thread and ths instance of par_t_reg
   pthread_exit(ptr);
}
```

Next Steps

Incorporation of pthreads into ADMB applications can produce useful speedup depending on the model and the hardware on which it is run. The ADMB pthread_manager class appears to be useful but not yet a finished product. Several features need further development, but most importantly ADMB users need to create applications for the pthread_manager class. If such applications are not developed, it is impossible to really understand how to fully integrate pthreads into ADMB. Some useful short-term modifications to the pthread_manager class include:

1. Simplifying the API
2. Rationalizing the destructor
3. Implementing the -ams, -gbs , and -ams flags to apply both the the master thread and the slaves
4. Computing the Hessian in threads
5. Writing some DOX for the API
6. What would be required for a THREAD_FUNCTION tpl keyword?