ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

Required changes to the sequential solution

- what is needed
  - an object to hold the problem information shared among the threads
  - the per-thread administrative data structure that holds the thread id and the pointer to the shared information
    - this is the argument to `pthread_create`
  - a mutex to protect the update of the global convergence criterion
  - a `pthread_create` compatible worker routine
  - a change at the top-level driver to enable the user to choose the number of threads
- and a strategy for managing the thread life cycle
  - synchronization is trivial if
    - we spawn our threads to perform the updates of a single iteration
    - harvest them
    - check the convergence criterion
    - stop, or respawn them if another iteration is necessary
  - can the convergence test be done in parallel?
    - so we don't have to pay the create/harvest overhead?
    - if so, how do we guarantee correctness and consistency?

Threaded Jacobi: thread data

```
struct Task {
   // shared information
   size_t workers;
   Grid & current;
   Grid & next;
   double maxDeviation;
   // mutex to control access to the convergence criterion
   pthread_mutex_t lock;

   // constructor
   Task(size_t workers, Grid & current, Grid & next) :
      workers(workers), current(current), next(next), maxDeviation(0.0) {
      pthread_mutex_init(&lock, 0);
   }
   // destructor
   ~Task() {
      pthread_mutex_destroy(&lock);
   }
};

struct Context {
   // thread info
   size_t id;
   pthread_t descriptor;
   Task * task;
};
```

Threaded Jacobi: driving the update

```
void Jacobi::solve(Problem & problem) {
   // initialize the problem
   problem.initialize();
   // do the actual solve
   _solve(problem);
   // compute and store the error
   std::cout << " computing absolute error" << std::endl;
   // compute the relative error
   Grid & error = problem.error();
   const Grid & exact = problem.exact();
   const Grid & solution = problem.solution();

   for (size_t j=0; j < exact.size(); j++) {
      for (size_t i=0; i < exact.size(); i++) {
         if (exact(i,j) == 0.0) {
            error(i,j) = std::abs(solution(i,j));
         } else {
            error(i,j) = std::abs(solution(i,j) - exact(i,j))/exact(i,j);
         }
      }
   }
   std::cout << " --- done." << std::endl;
   return;
}
```

Threaded `Jacobi`: the master thread

```
void Jacobi::_solve(Problem & problem) {
   Grid & current = problem.solution();

   // create and initialize temporary storage
   Grid next(current.size());
   problem.initialize(next);

   // shared thread info
   Task task(_workers, current, next);
   // per-thread information
   Context context[_workers];

   // let's get going
   std::cout << "jacobi: tolerance=" << _tolerance << std::endl;

   // put an upper bound on the number of iterations
   const size_t max_iterations = (size_t) 1.0e4;
```

Threaded Jacobi: the master thread, part 2

```
    for (size_t iterations = 0; iterations<max_iterations; iterations++) {
        if (iterations % 100 == 0) {
            std::cout << " " << iterations << std::endl;
        }
        // reset the maximium deviation
        task.maxDeviation = 0.0;
        // spawn the threads
        for (size_t tid=0; tid < _workers; tid++) {
            context[tid].id = tid;
            context[tid].task = &task;

            int status = pthread_create(&context[tid].descriptor, 0, _update, &context[tid]);
            if (status) {
                throw ("error in pthread_create");
            }
        }
        // harvest the threads
        for (size_t tid = 0; tid < _workers; tid++) {
            pthread_join(context[tid].descriptor, 0);
        }

        // swap the blocks between the two grids
        Grid::swapBlocks(current, next);
        // check covergence
        if (task.maxDeviation < _tolerance) {
            std::cout << " ### convergence in " << iterations << " iterations!" << std::endl;
            break;
        }
    }
    std::cout << " --- done." << std::endl;

    return;
}
```

Threaded `Jacobi`: update in the worker threads

```
void * Jacobi::_update(void * arg) {
   Context * context = static_cast<Context *>(arg);

   size_t id = context->id;
   Task * task = context->task;

   size_t workers = task->workers;
   Grid & current = task->current;
   Grid & next = task->next;
   pthread_mutex_t lock = task->lock;

   double max_dev = 0.0;
   // do an iteration step
   // leave the boundary alone
   // iterate over the interior of the grid
   for (size_t j=id+1; j < current.size()-1; j+=workers) {
      for (size_t i=1; i < current.size()-1; i++) {
         next(i,j) = 0.25*(current(i+1,j)+current(i-1,j)+current(i,j+1)+current(i,j-1));
         // compute the deviation from the last generation
         double dev = std::abs(next(i,j) - current(i,j));
         // and update the maximum deviation
         if (dev > max_dev) {
            max_dev = dev;
         }
      }
   }

   // grab the lock and update the global maximum deviation
   pthread_mutex_lock(&lock);
   if (task->maxDeviation < max_dev) {
      task->maxDeviation = max_dev;
   }
   pthread_mutex_unlock(&lock);

   return 0;
}
```

Assessing the threaded implementation

- the implemented synchronization scheme is very simple
  - each grid update step spawns some number of workers to update a subset of the cells
  - the workers are harvested after the grid is updated
  - the main thread checks for convergence
  - if another iteration is required, a new set of workers is spawned
- the simplicity of this strategy comes at a cost
  - *scalability* suffers when the overhead of creating and harvesting threads is comparable to amount of work done by each thread
  - for low thread counts, it is still an overall win, since the time to solution decreases and the machine utilization is better
  - but as the number of threads increases, the program becomes *slower*
    - timing a 100×100 grid to convergence on a recent MacPro

threads	1	2	4	8	16
time(s)	4.367	2.517	1.918	1.937	3.537

    - and 10,000 iterations of a 1000×1000 grid

threads	1	2	4	8	16
time(s)	413.306	211.050	109.509	98.279	74.087

Improving the update loop

- the plan is to keep the workers alive and updating the grid while either we converge or `max_iterations` is reached
- the main thread
  - loops to spawn all the threads
  - and immediately enters a loop to harvest them
- the workers use a condition variable to synchronize among themselves
  - they iterate, updating the grid
  - grab a mutex, deposit their local maximum deviation from the last iterations, update a counter that records how many workers have completed their update, and release the lock
  - enter another critical section with the termination logic
    - everybody uses a condition variable to wait for the slowest worker
    - the slowest worker checks the convergence criterion and updates the termination flag, swaps the grid blocks and signals everybody else
    - if the termination flag is set, or if the maximum number of iterations has been reached, all threads exit

Threaded Jacobi: the main thread

```
void Jacobi::_solve(Problem & problem) {
   Grid & current = problem.solution();

   // create and initialize temporary storage
   Grid next(current.size());
   problem.initialize(next);

   // shared thread info
   Task task(_workers, _tolerance, current, next);
   // per-thread information
   Context context[_workers];
   // spawn the threads
   std::cout << "jacobi: spawning " << _workers << " workers" << std::endl;
   for (size_t tid=0; tid < _workers; tid++) {
      context[tid].id = tid;
      context[tid].task = &task;

      int status = pthread_create(&context[tid].descriptor, 0, _update, &context[tid]);
      if (status) {
         throw ("error in pthread_create");
      }
   }
   // harvest the threads
   for (size_t tid = 0; tid < _workers; tid++) {
         pthread_join(context[tid].descriptor, 0);
      }
   // done
   std::cout << "jacobi: done." << std::endl;
   return;
}
```

Threaded `Jacobi`: updated thread data

```
struct Task {
   // shared information
   size_t workers; // the number of threads
   double tolerance; // the covergence tolerance
   Grid & current;
   Grid & next;

   bool done; // is there more work?
   double maxDeviation; // the value
   size_t contributions; // the number of threads that have deposited contributions
   pthread_mutex_t gridUpdate_lock; //the mutex
   pthread_cond_t gridUpdate_check;

   Task(size_t workers, double tolerance, Grid & current, Grid & next) :
      workers(workers), tolerance(tolerance), current(current), next(next),
      done(false), maxDeviation(0.0), contributions(0),
      gridUpdate_lock(), gridUpdate_check() {
      // initialize the grid update lock
      pthread_mutex_init(&gridUpdate_lock, 0);
      pthread_cond_init(&gridUpdate_check, 0);
   }

   ~Task() {
      pthread_mutex_destroy(&gridUpdate_lock);
      pthread_cond_destroy(&gridUpdate_check);
   }
};
```

Threaded Jacobi: workers, part 1

```
// the threaded update
void * Jacobi::_update(void * arg) {
   Context * context = static_cast<Context *>(arg);

   size_t id = context->id;
   Task * task = context->task;

   const size_t workers = task->workers;
   Grid & current = task->current;
   Grid & next = task->next;

   size_t maxIterations = (size_t) 1e4;
   // iterate, updating the grid until done
   for (size_t iteration = 0; iteration < maxIterations; iteration++) {
      // thread 0: print an update
      if (id == 0 && iteration % 100 == 0) {
         std::cout << " " << iteration << std::endl;
      }

      double max_dev = 0.0;
      // do an iteration step
      // leave the boundary alone
      // iterate over the interior of the grid
      for (size_t j=id+1; j < current.size()-1; j+=workers) {
         for (size_t i=1; i < current.size()-1; i++) {
            next(i,j) = 0.25*(current(i+1,j)+current(i-1,j)+current(i,j+1)+current(i,j-1));
            // compute the deviation from the last generation
            double dev = std::abs(next(i,j) - current(i,j));
            // and update the maximum deviation
            if (dev > max_dev) {
               max_dev = dev;
            }
         }
      }
      // done with the grid update
```

Threaded Jacobi: workers, part 2

```
      // grab the grid update lock
      pthread_mutex_lock(&task->gridUpdate_lock);
      // update the global maximum deviation
      if (task->maxDeviation < max_dev) {
         task->maxDeviation = max_dev;
      }
      // leave a mark
      task->contributions++;
      // bookkeeping at the end of the update
      if (task->contributions == workers) {
         // if i am the slowest worker
         // swap the blocks between the two grids
         Grid::swapBlocks(current, next);
         // check covergence
         if (task->maxDeviation < task->tolerance) {
            std::cout
               << " +++ thread " << id << ": convergence in " << iteration << " iterations"
               <<std::endl;
            task->done = true;
         }
         // reset our accounting and signal everybody
         task->contributions = 0;
         task->maxDeviation = 0;
         pthread_cond_broadcast(&task->gridUpdate_check);
      } else {
         // all but the slowest wait here
         pthread_cond_wait(&task->gridUpdate_check, &task->gridUpdate_lock);
      }
      // release
      pthread_mutex_unlock(&task->gridUpdate_lock);
      // check whether we are done
      if (task->done) {
         break;
      }
   }
   return 0;
}
```

Assessing the improved implementation

- the improved threading scheme is not much more complex
  - we keep track of how many threads have computed their grid update
  - the slowest worker check the convergence criterion and performs all the necessary bookkeeping
  - while everybody else waits
  - use `pthread_cond_brodacast` to wake the other workers
- here is the performance comparison for 10,000 iterations on a 1000×1000 grid on the same 8-core MacPro

threads	1	2	4	8	16
previous(s)	413.306	211.050	109.509	98.279	74.087
updated(s)	408.636	208.832	107.015	59.043	61.481