ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

Parallelization using threads

- the shared memory implementation requires
  - a scheme so that threads can update cells without the need for locks
  - while maximizing locality of data access
  - even the computation of the convergence criterion can be parallelized
- parallelization strategy
  - we will focus on parallelizing the iterative grid update
    - grid initialization, visualization, computing the exact answer and the error field do not depend on the *number of iterations*
  - the finest grain of work is clearly an individual cell update based on the value of its four nearest neighbors
  - for this two dimensional example, we can build coarser grain tasks using
    - horizontal or vertical strips
    - non-overlapping blocks
    - the strategy gets more complicated if you want to perform the update in place
  - the communication patterns are trivial for the double buffering layout; only the final update of the convergence criterion requires any locking
  - each coarse grain task can be assigned to a thread

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 $\times$ 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 $\times$ 1000 grid

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