ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

Printing out the initial grid

- we should be able to print out the initialized grid

```
#> mm laplace
#> laplace
#> cat laplace.csv
0,0,0.3827,0.7071,0.9239,1,0.9239,0.7071,0.3827,1.225e-16
1,0,1,1,1,1,1,1,1,0
2,0,1,1,1,1,1,1,1,0
3,0,1,1,1,1,1,1,1,0
4,0,1,1,1,1,1,1,1,0
5,0,1,1,1,1,1,1,1,0
6,0,1,1,1,1,1,1,1,0
7,0,1,1,1,1,1,1,1,0
8,0,0.01654,0.0306,0.03992,0.04321,0.0399,0.03056,0.01654,0
```

- notice that
  - the top line contains some recognizable values
  - the left and right borders are set to zero
  - the interior of the grid is painted with our initial guess
- still to do:
  - write the update
  - build a grid with the exact solution
  - build the error field (why?)

Fleshing out the solver

```
// the solver driver
void laplace(Grid & current, double tolerance) {
   // create and initialize temporary storage
   Grid next(current.size());
   initialize(next);
   // put an upper bound on the number of iterations
   long max_iterations = (long) 1e4;;
   for (long iterations = 0; iterations<max_iterations; iterations++) {
      double max_dev = 0.0;
      // do an iteration step
      // leave the boundary alone
      // iterate over the interior of the grid
      for (size_t j=1; j < current.size()-1; j++) {
         for (size_t i=1; i < current.size()-1; i++) {
            // update
            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;
            }
         }
      }
      // swap the blocks between the two grids
      Grid::swapBlocks(current, next);
      // check covergence
      if (max_dev < tolerance) {
         break;
      }
   }
   return;
}
```

Adding the new grid interface

- here is the declaration of `Grid::swapBlocks`

```
class Grid {
   // interface
   public:
   ...
   // exchange the data blocks of two compatible grids
   static void swapBlocks(Grid &, Grid &);
   ...
};
```

- and its definition

```
void Grid::swapBlocks(Grid & g1, Grid & g2) {
   // bail out if the two operands are not compatible
   if (g1.size() != g2.size()) {
      throw "Grid::swapblocks: size mismatch";
   }
   if (g1.delta() != g2.delta()) {
      throw "Grid::swapblocks: spacing mismatch";
   }
   // but if they are, just exhange their data buffers
   double * temp = g1._block;
   g1._block = g2._block;
   g2._block = temp;
   // all done
   return;
}
```

Reworking the driver

```
    // build a visualizer
    Visualizer vis;

    // compute the exact solution
    Grid solution(N);
    exact(solution);
    std::fstream exact_stream("exact.csv", std::ios_base::out);
    vis.csv(solution, exact_stream);

    // allocate space for the solution
    Grid potential(N);
    // initialize and apply our boundary conditions
    initialize(potential);
    // call the solver
    laplace(potential, tolerance);
    // open a stream to hold the answer
    std::fstream output_stream(filename, std::ios_base::out);
    // build a visualizer and render the solution in our chosen format
    vis.csv(potential, output_stream);

    // compute the error field
    Grid error(N);
    relative_error(potential, solution, error);
    std::fstream error_stream("error.csv", std::ios_base::out);
    vis.csv(error, error_stream);

    // all done
    return 0;
}
```

Computing the exact solution and the error field

```
void exact(Grid & grid) {
   // paint the exact solution
   for (size_t j=0; j < grid.size(); j++) {
      for (size_t i=0; i < grid.size(); i++) {
         double x = i*grid.delta();
         double y = j*grid.delta();
         grid(i,j) = std::exp(-pi*y)*std::sin(pi*x);
      }
   }
   return;
}

void relative_error(
   const Grid & computed, const Grid & exact, Grid & error) {
   // compute the relative error
   for (size_t j=0; j < exact.size(); j++) {
      for (size_t i=0; i < exact.size(); i++) {
         if (exact(i,j) == 0.0) { // hm... sloppy!
            error(i,j) = std::abs(computed(i,j));
         } else {
            error(i,j) = std::abs(computed(i,j) - exact(i,j))/exact(i,j);
         }
      }
   }
   return;
}
```

Shortcomings

- numerics:
  - it converges very slowly; other update *schemes* improve on this
  - our approximation is very low order, so it takes very large grids to produce a few digits of accuracy
  - the convergence criterion has some unwanted properties; it triggers
    - prematurely: large swaths of constant values may never get updated
    - it would trigger even if we were updating the wrong grid!
- design:
  - separate the problem specification from its solution
  - there are other objects lurking, waiting to be uncovered
  - someone should make the graphic visualizer
  - restarts anybody?
  - how would you try out different convergence criteria? update schemes? memory layouts?
- usability:
  - supporting interchangeable parts requires damage to the top level driver
    - to enable the user to make the selection
    - to expose new command line arguments that configure the new parts

Assessing our fundamental

- `Grid` is a good starting point for abstracting structured grids
  - assumes ownership of the memory associated with a structured grid
  - encapsulates the indexing function
  - extend it to
    - support different memory layout strategies
    - support non-square grids (?)
    - support non-uniform grids (?)
    - higher dimensions
    - if you need any of these, consider using one of the many excellent class libraries written by experts
- `Visualizer`, under another name, can form the basis for a more general persistence library
  - to support HDF5, NetCDF, bitmaps, voxels, etc.

The `Problem` class: the interface

```
// the solution representation
class acm114::laplace::Problem {
   //typedefs
public:
   typedef std::string string_t;
   // interface
public:
   inline string_t name() const;
   inline const Grid & exact() const;
   inline const Grid & deviation() const;
   inline Grid & solution();
   inline const Grid & solution() const;
   inline Grid & error();
   inline const Grid & error() const;
   // abstract
   virtual void initialize() = 0;
   virtual void initialize(Grid &) const = 0;
   // meta methods
public:
   inline Problem(string_t name, double width, size_t points);
   virtual ˜Problem();
   // data members
```

The Problem class: the data

```
protected:
   string_t _name;
   double _delta;
   Grid _solution;
   Grid _exact;
   Grid _error;
   Grid _deviation;
   // disable these
private:
   Problem(const Problem &);
   const Problem & operator= (const Problem &);
};
```

The Example class

```
class acm114::laplace::Example : public acm114::laplace::Problem {
   // interface
public:
   virtual void initialize();
   virtual void initialize(Grid &) const;

   // meta methods
public:
   inline Example(string_t name, double width, size_t points);
   virtual ˜Example();

   // disable these
private:
   Example(const Example &);
   const Example & operator= (const Example &);
};
```

The Solver class

```
class acm114::laplace::Solver {
   // interface
public:
   virtual void solve(Problem &) = 0;

   // meta methods
public:
   inline Solver();
   virtual ~Solver();

   // data members
private:

   // disable these
private:
   Solver(const Solver &);
   const Solver & operator= (const Solver &);
};
```

The Jacobi class

```
class acm114::laplace::Jacobi : public acm114::laplace::Solver {
   // interface
public:
   virtual void solve(Problem &);

   // meta methods
public:
   inline Jacobi(double tolerance, size_t workers);
   virtual ~Jacobi();

   // implementation details
protected:
   virtual void _solve(Problem &);
   static void * _update(void *);

   // data members
private:
   double _tolerance;
   size_t _workers;

   // disable these
private:
   Jacobi(const Jacobi &);
   const Jacobi & operator= (const Jacobi &);
};
```

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;
   pthread_mutex_t lock; // mutex to control access to the convergence criterion

   Task(size_t workers, Grid & current, Grid & next) :
      workers(workers), current(current), next(next), maxDeviation(0.0) {
      pthread_mutex_init(&lock, 0);
   }
};

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;
}
```