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

```
169 // the solver driver
170 void laplace(Grid & current, double tolerance) {
171    // create and initialize temporary storage
172    Grid next(current.size());
173    initialize(next);
174    // put an upper bound on the number of iterations
175    long max_iterations = (long) 1e4;;
176    for (long iterations = 0; iterations<max_iterations; iterations++) {
177       double max_dev = 0.0;
178       // do an iteration step
179       // leave the boundary alone
180       // iterate over the interior of the grid
181       for (size_t j=1; j < current.size()-1; j++) {
182          for (size_t i=1; i < current.size()-1; i++) {
183             // update
184             next(i,j) = 0.25*(
185                current(i+1,j)+current(i-1,j)+current(i,j+1)+current(i,j-1));
186             // compute the deviation from the last generation
187             double dev = std::abs(next(i,j) - current(i,j));
188             // and update the maximum deviation
189             if (dev > max_dev) {
190                max_dev = dev;
191             }
192          }
193       }
194       // swap the blocks between the two grids
195       Grid::swapBlocks(current, next);
196       // check covergence
197       if (max_dev < tolerance) {
198          break;
199       }
200    }
201    return;
202 }
```

Adding the new grid interface

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

```
30 class Grid {
31    // interface
32    public:
33    ...
34    // exchange the data blocks of two compatible grids
35    static void swapBlocks(Grid &, Grid &);
36    ...
37 };
```

- and its definition

```
69 void Grid::swapBlocks(Grid & g1, Grid & g2) {
70    // bail out if the two operands are not compatible
71    if (g1.size() != g2.size()) {
72       throw "Grid::swapblocks: size mismatch";
73    }
74    if (g1.delta() != g2.delta()) {
75       throw "Grid::swapblocks: spacing mismatch";
76    }
77    // but if they are, just exhange their data buffers
78    double * temp = g1._block;
79    g1._block = g2._block;
80    g2._block = temp;
81    // all done
82    return;
83 }
```

Reworking the driver

```
239     // build a visualizer
240     Visualizer vis;
241
242     // compute the exact solution
243     Grid solution(N);
244     exact(solution);
245     std::fstream exact_stream("exact.csv", std::ios_base::out);
246     vis.csv(solution, exact_stream);
247
248     // allocate space for the solution
249     Grid potential(N);
250     // initialize and apply our boundary conditions
251     initialize(potential);
252     // call the solver
253     laplace(potential, tolerance);
254     // open a stream to hold the answer
255     std::fstream output_stream(filename, std::ios_base::out);
256     // build a visualizer and render the solution in our chosen format
257     vis.csv(potential, output_stream);
258
259     // compute the error field
260     Grid error(N);
261     relative_error(potential, solution, error);
262     std::fstream error_stream("error.csv", std::ios_base::out);
263     vis.csv(error, error_stream);
264
265     // all done
266     return 0;
267 }
```

Computing the exact solution and the error field

```
143 void exact(Grid & grid) {
144    // paint the exact solution
145    for (size_t j=0; j < grid.size(); j++) {
146       for (size_t i=0; i < grid.size(); i++) {
147          double x = i*grid.delta();
148          double y = j*grid.delta();
149          grid(i,j) = std::exp(-pi*y)*std::sin(pi*x);
150       }
151    }
152    return;
153 }
154
155 void relative_error(
156    const Grid & computed, const Grid & exact, Grid & error) {
157    // compute the relative error
158    for (size_t j=0; j < exact.size(); j++) {
159       for (size_t i=0; i < exact.size(); i++) {
160          if (exact(i,j) == 0.0) { // hm... sloppy!
161             error(i,j) = std::abs(computed(i,j));
162          } else {
163             error(i,j) = std::abs(computed(i,j) - exact(i,j))/exact(i,j);
164          }
165       }
166    }
167    return;
168 }
```

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

```
1  // the solution representation
2  class acm114::laplace::Problem {
3     //typedefs
4  public:
5     typedef std::string string_t;
6     // interface
7  public:
8     inline string_t name() const;
9     inline const Grid & exact() const;
10    inline const Grid & deviation() const;
11    inline Grid & solution();
12    inline const Grid & solution() const;
13    inline Grid & error();
14    inline const Grid & error() const;
15    // abstract
16    virtual void initialize() = 0;
17    virtual void initialize(Grid &) const = 0;
18    // meta methods
19 public:
20    inline Problem(string_t name, double width, size_t points);
21    virtual ~Problem();
22    // data members
```

The `Problem` class: the data

```
23 protected:
24    string_t _name;
25    double _delta;
26    Grid _solution;
27    Grid _exact;
28    Grid _error;
29    Grid _deviation;
30    // disable these
31 private:
32    Problem(const Problem &);
33    const Problem & operator= (const Problem &);
34 };
```

The `Example` class

```
1  class acm114::laplace::Example : public acm114::laplace::Problem {
2     // interface
3  public:
4     virtual void initialize();
5     virtual void initialize(Grid &) const;
6
7     // meta methods
8  public:
9     inline Example(string_t name, double width, size_t points);
10    virtual ~Example();
11
12    // disable these
13 private:
14    Example(const Example &);
15    const Example & operator= (const Example &);
16 };
```

The `Solver` class

```
1  class acm114::laplace::Solver {
2     // interface
3  public:
4     virtual void solve(Problem &) = 0;
5
6     // meta methods
7  public:
8     inline Solver();
9     virtual ~Solver();
10
11    // data members
12 private:
13
14    // disable these
15 private:
16    Solver(const Solver &);
17    const Solver & operator= (const Solver &);
18 };
```

The `Jacobi` class

```
1  class acm114::laplace::Jacobi {
2     // interface
3  public:
4     virtual void solve(Problem &);
5
6     // meta methods
7  public:
8     inline Jacobi(double tolerance, size_t workers);
9     virtual ~Jacobi();
10
11    // implementation details
12 protected:
13    virtual void _solve(Problem &);
14    static void * _update(void *);
15
16    // data members
17 private:
18    double _tolerance;
19    size_t _workers;
20
21    // disable these
22 private:
23    Jacobi(const Jacobi &);
24    const Jacobi & operator= (const Jacobi &);
25 };
```

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

```
1  struct Task {
2     // shared information
3     size_t workers;
4     Grid & current;
5     Grid & next;
6     double maxDeviation;
7     pthread_mutex_t lock; // mutex to control access to the convergence criterion
8
9     Task(size_t workers, Grid & current, Grid & next) :
10       workers(workers), current(current), next(next), maxDeviation(0.0) {
11       pthread_mutex_init(&lock, 0);
12    }
13 };
14
15 struct Context {
16    // thread info
17    size_t id;
18    pthread_t descriptor;
19    Task * task;
20 };
```

Threaded `Jacobi`: driving the update

```
22 void Jacobi::solve(Problem & problem) {
23    // initialize the problem
24    problem.initialize();
25    // do the actual solve
26    _solve(problem);
27    // compute and store the error
28    std::cout << " computing absolute error" << std::endl;
29    // compute the relative error
30    Grid & error = problem.error();
31    const Grid & exact = problem.exact();
32    const Grid & solution = problem.solution();
33
34    for (size_t j=0; j < exact.size(); j++) {
35       for (size_t i=0; i < exact.size(); i++) {
36          if (exact(i,j) == 0.0) {
37             error(i,j) = std::abs(solution(i,j));
38          } else {
39             error(i,j) = std::abs(solution(i,j) - exact(i,j))/exact(i,j);
40          }
41       }
42    }
43    std::cout << " --- done." << std::endl;
44    return;
45 }
```

Threaded `Jacobi`: the master thread

```
46 void Jacobi::_solve(Problem & problem) {
47    Grid & current = problem.solution();
48
49    // create and initialize temporary storage
50    Grid next(current.size());
51    problem.initialize(next);
52
53    // shared thread info
54    Task task(_workers, current, next);
55    // per-thread information
56    Context context[_workers];
57
58    // let's get going
59    std::cout << "jacobi: tolerance=" << _tolerance << std::endl;
60
61    // put an upper bound on the number of iterations
62    const size_t max_iterations = (size_t) 1.0e4;
```

Threaded `Jacobi`: the master thread, part 2

```
63    for (size_t iterations = 0; iterations<max_iterations; iterations++) {
64       if (iterations % 100 == 0) {
65          std::cout << " " << iterations << std::endl;
66       }
67       // reset the maximium deviation
68       task.maxDeviation = 0.0;
69       // spawn the threads
70       for (size_t tid=0; tid < _workers; tid++) {
71          context[tid].id = tid;
72          context[tid].task = &task;
73
74          int status = pthread_create(&context[tid].descriptor, 0, _update, &context[tid]);
75          if (status) {
76             throw ("error in pthread_create");
77          }
78       }
79       // harvest the threads
80       for (size_t tid = 0; tid < _workers; tid++) {
81          pthread_join(context[tid].descriptor, 0);
82       }
83
84       // swap the blocks between the two grids
85       Grid::swapBlocks(current, next);
86       // check covergence
87       if (task.maxDeviation < _tolerance) {
88          std::cout << " ### convergence in " << iterations << " iterations!" << std::endl;
89          break;
90       }
91    }
92    std::cout << " --- done." << std::endl;
93
94    return;
95 }
```

Threaded `Jacobi`: update in the worker threads

```
96  void * Jacobi::_update(void * arg) {
97     Context * context = static_cast<Context *>(arg);
98
99     size_t id = context->id;
100    Task * task = context->task;
101
102    size_t workers = task->workers;
103    Grid & current = task->current;
104    Grid & next = task->next;
105    pthread_mutex_t lock = task->lock;
106
107    double max_dev = 0.0;
108    // do an iteration step
109    // leave the boundary alone
110    // iterate over the interior of the grid
111    for (size_t j=id+1; j < current.size()-1; j+=workers) {
112       for (size_t i=1; i < current.size()-1; i++) {
113          next(i,j) = 0.25*(current(i+1,j)+current(i-1,j)+current(i,j+1)+current(i,j-1));
114          // compute the deviation from the last generation
115          double dev = std::abs(next(i,j) - current(i,j));
116          // and update the maximum deviation
117          if (dev > max_dev) {
118             max_dev = dev;
119          }
120       }
121    }
122
123    // grab the lock and update the global maximum deviation
124    pthread_mutex_lock(&lock);
125    if (task->maxDeviation < max_dev) {
126       task->maxDeviation = max_dev;
127    }
128    pthread_mutex_unlock(&lock);
129
130    return 0;
131 }
```