

# 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

```
1 #> mm laplace
2 #> laplace
3 #> cat laplace.csv
4 0,0,0.3827,0.7071,0.9239,1,0.9239,0.7071,0.3827,1.225e-16
5 1,0,1,1,1,1,1,1,1,0
6 2,0,1,1,1,1,1,1,1,0
7 3,0,1,1,1,1,1,1,1,0
8 4,0,1,1,1,1,1,1,1,0
9 5,0,1,1,1,1,1,1,1,0
10 6,0,1,1,1,1,1,1,1,0
11 7,0,1,1,1,1,1,1,1,0
12 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 convergence
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 exchange 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 acml14::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 acml14::laplace::Jacobi : public acml14::laplace::Solver {
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 };
```