ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

```
77  // main program
78  int main(int argc, char* argv[]) {
79      // default values for our user configurable settings
80      size_t N = 10;
81      double tolerance = 1.0e-6;
82      const char* filename = "laplace.csv";
83
84      // read the command line
85      int command;
86      while ((command = getopt(argc, argv, "N:e:o:")) != -1) {
87          switch (command) {
88          // get the convergence tolerance
89          case 'e':
90              tolerance = atof(optarg);
91              break;
92          // get the grid size
93          case 'N':
94              N = (size_t) atof(optarg);
95              break;
96          // get the name of the output file
97          case 'o':
98              filename = optarg;
99          }
100     }
```

Sequential implementation - driving the solver

```
102     // allocate space for the solution
103     Grid potential(N);
104
105     // initialize and apply our boundary conditions
106     initialize(potential);
107
108     // call the solver
109     laplace(potential, tolerance);
110
111     // open a stream to hold the answer
112     std::fstream output(filename, std::ios_base::out);
113
114     // build a visualizer and render the solution in our chosen format
115     Visualizer visualizer;
116     visualizer.csv(potential, output);
117
118     // all done
119     return 0;
120 }
```

Sequential implementation - the preamble

- ▶ back up to the beginning of the file

```
1 #include <getopt.h>
2 #include <cmath>
3 #include <cstdlib>
4 #include <fstream>
5 #include <iostream>
6
7 // forward declarations
8 class Grid;
9 class Visualizer;
10
11 // the solver; does nothing for the time being
12 void initialize(Grid & grid) {};
13 void laplace(Grid & grid, double tolerance){};
```

- ▶ we have separated out *visualization* in a different object to support different formats without disturbing the data representation
- ▶ initialize and laplace have trivial implementations for now
    - ▶ enables testing the scaffolding without worrying about the solver implementation just yet

Sequential implementation - the grid object stub

```
15  // the solution representation
16  class Grid {
17      // interface: TBD
18  public:
19
20      // meta methods
21  public:
22      Grid(size_t size);
23      ~Grid();
24
25      // private data members: TBD
26  private:
27
28      // disabled interface
29      // grid will own dynamic memory, so don't let the compiler screw up
30  private:
31      Grid(const Grid &);
32      const Grid & operator= (const Grid &);
33  };
34
35  // the grid implementation
36  Grid::Grid(size_t size) {
37  }
38
39  Grid::~Grid() {
40  }
```

Sequential implementation - the visualizer stub

```
97   // the visualizer class
98   class Visualizer {
99       // local type aliases
100  public:
101      typedef std::ostream stream_t;
102
103      // interface
104  public:
105      void csv(const Grid & grid, stream_t & stream);
106
107      // meta methods
108  public:
109      inline Visualizer() {}
110  };
111
112  // the Visualizer class implementation
113  void Visualizer::csv(const Grid & grid, Visualizer::stream_t & stream) {
114      return;
115  }
```

- the code now compiles and links
    - consistency check that the object collaborations are ok, for now
    - can be tested for command line option parsing

Fleshing out the initializer

```
1  // the grid initializer:
2  // clear the grid contents and apply our boundary conditions
3  void initialize(Grid & grid) {
4      // ask the grid to clear its memory
5      grid.clear(1.0);
6      // apply the dirichlet conditions
7      for (size_t cell=0; cell < grid.size(); cell++) {
8          // evaluate sin(pi x)
9          double sin = std::sin(cell * grid.delta() * pi);
10         // along the x axis, at top and bottom
11         grid(cell, 0) = sin;
12         grid(cell, grid.size()-1) = sin * std::exp(-pi);
13         // along the y axis, left and right
14         grid(0, cell) = 0.0;
15         grid(grid.size()-1, cell) = 0.0;
16     }
17
18     return;
19 }
```

- the grid knows its size, its spacing δ, and can initialize its memory
- access to grid elements happens through an overloaded `operator()` so we can *encapsulate* the indexing function

The grid class declaration

```
29  // the solution representation
30  class Grid {
31      // interface
32  public:
33      // set all cells to the specified value
34      void clear(double value=0.0);
35      // the grid dimensions
36      size_t size() const {return _size;}
37      // the grid spacing
38      double delta() const {return _delta;}
39      // access to the cells
40      double & operator()(size_t i, size_t j) {return _block[j*_size+i];}
41      double operator()(size_t i, size_t j) const {return _block[j*_size+i];}
42      // meta methods
43  public:
44      Grid(size_t size);
45      ~Grid();
46      // data members
47  private:
48      const size_t _size;
49      const double _delta;
50      double* _block;
51      // disable these
52  private:
53      Grid(const Grid &);
54      const Grid & operator= (const Grid &);
55  };
```

The grid class implementation

```
57  // the grid implementation
58  // interface
59  void Grid::clear(double value) {
60      for (size_t i=0; i < _size*_size; i++) {
61          _block[i] = value;
62      }
63
64      return;
65  }
66
67  // constructor
68  Grid::Grid(size_t size) :
69      _size(size),
70      _delta((1.0 - 0.0)/(size-1)),
71      _block(new double[size*size]) {
72  }
73
74  // destructor
75  Grid::~Grid() {
76      delete [] _block;
77  }
```

Grid visualization

```
1  // the visualizer class
2  class Visualizer {
3      // local type aliases
4  public:
5      typedef std::ostream stream_t;
6      // interface
7  public:
8      void csv(const Grid & grid, stream_t & stream);
9      // meta methods
10 public:
11     inline Visualizer() {}
12 };
13
14 // the Visualizer class implementation
15 void Visualizer::csv(const Grid & grid, Visualizer::stream_t & stream) {
16     for (size_t j=0; j < grid.size(); j++) {
17         stream << j;
18         for (size_t i=0; i < grid.size(); i++) {
19             stream << "," << grid(i,j);
20         }
21         stream << std::endl;
22     }
23
24     return;
25 }
```

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?)