ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

Solving a simple PDE on a uniform structured grid

- Laplace equation over some domain $\Omega \in \mathbb{R}^d$, subject to Dirichlet boundary conditions

$$\nabla^2 \phi = 0 \quad \text{with} \quad \phi(\partial\Omega) = f \tag{1}$$

- let the grid be uniform: $\delta_x = \delta_y$
- in two dimensions, using first order central differences, Eq. 1 becomes

$(\partial_{xx} + \partial_{yy})$  $= 0$ (2)

4  $=$  $+$  $+$  $+$  (3)

- and translates into the following constraint among grid elements

 $= \frac{1}{4}$  (4)

using a shorthand for the sum of the neighboring cells

An example

- specifically,
  - let Ω be the unit box in two dimensions
  - and let ϕ satisfy the following boundary conditions

$$\begin{array}{rcccll} & & \phi(x,0) & = & \sin(\pi x) & 0 \leq x \leq 1 \\ & & \phi(x,1) & = & e^{-\pi}\sin(\pi x) & 0 \leq x \leq 1 \\ \phi(0,y) & = & \phi(1,y) & = & 0 & 0 \leq y \leq 1 \end{array} \tag{5}$$

- the exact solution is given by

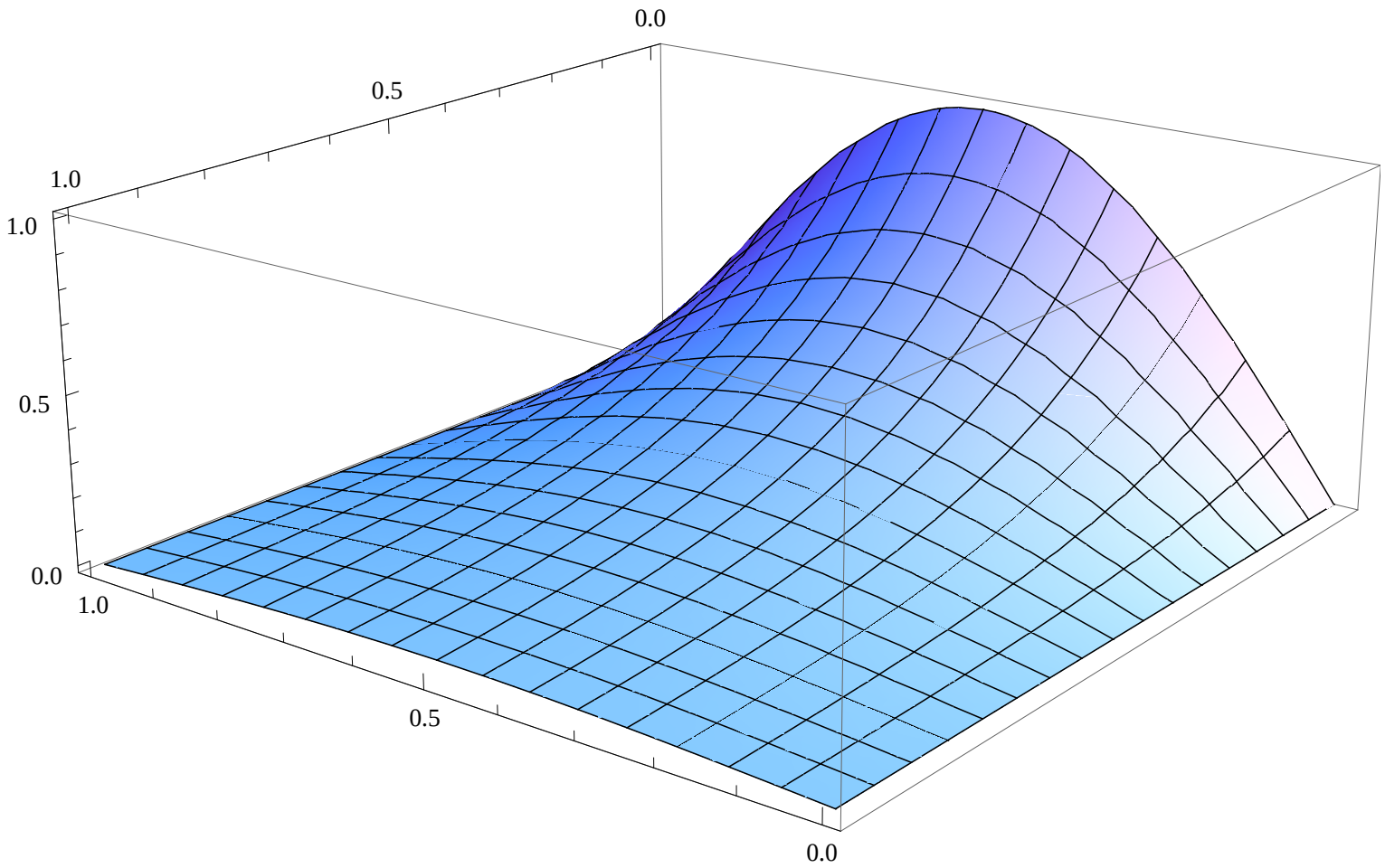
$$\phi(x,y) = e^{-\pi y}\sin(\pi x) \tag{6}$$

- we will solve this equation using the Jacobi iterative scheme:
  - make an initial guess for ϕ over a discretization of Ω
  - apply the boundary conditions
  - interpret Eq. 4 as an update step to compute the next iteration

$$\boxed{}_{t} = \frac{1}{4}\boxed{}_{t-1} \tag{7}$$

  - stop when a convergence criterion is met

The solution



Implementation strategy

- grid resolution:
  - ideally determined by analyzing the boundary conditions, since discrete sampling may wash out sharp features
  - for our simple example, this can be done as part of the solver initialization
  - we will use an $N \times N$ grid and let N be user specified so we can control the problem size

$$\delta_x = \delta_y = \frac{1}{N-1} \tag{8}$$

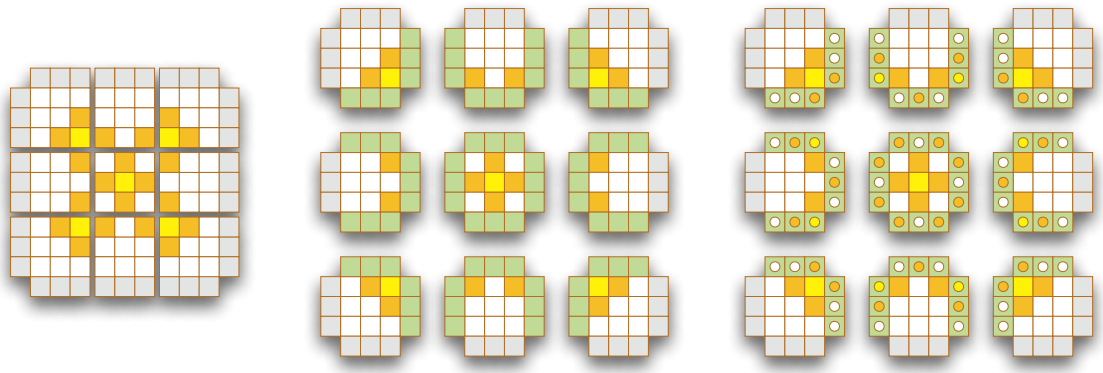
- data layout
  - investigate the effect of data locality by trying out various layouts
- setting up the update
  - we only need to keep track of two iterants
  - can be done in place; do you see how?
- convergence criterion
  - we will stop iterating when

$$\max_{\Omega} \left(\phi_t - \phi_{t-1}\right)) < \epsilon \tag{9}$$

  and let the user specify ϵ

Parallelization

- the finest grain of work is clearly the cell update based on the value of its four nearest neighbors
- 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
- with `MPI`
  - must partition the mesh among processes
  - each process work on its own subgrid
  - communication is required every iteration
  - parallel convergence testing involves a collective operation



Sequential implementation - user interface

```
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