ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

The grid class declaration

```
// the solution representation
class Grid {
   // interface
public:
   // set all cells to the specified value
   void clear(double value=0.0);
   // the grid dimensions
   size_t size() const {return _size;}
   // the grid spacing
   double delta() const {return _delta;}
   // access to the cells
   double & operator()(size_t i, size_t j) {return _block[j*_size+i];}
   double operator()(size_t i, size_t j) const {return _block[j*_size+i];}
   // meta methods
public:
   Grid(size_t size);
   ~Grid();
   // data members
private:
   const size_t _size;
   const double _delta;
   double* _block;
   // disable these
private:
   Grid(const Grid &);
   const Grid & operator= (const Grid &);
};
```

The grid class implementation

```
// the grid implementation
// interface
void Grid::clear(double value) {
   for (size_t i=0; i < _size*_size; i++) {
      _block[i] = value;
   }

   return;
}

// constructor
Grid::Grid(size_t size) :
   _size(size),
   _delta((1.0 - 0.0)/(size-1)),
   _block(new double[size*size]) {
}

// destructor
Grid::~Grid() {
   delete [] _block;
}
```

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 fundamentals

- `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.