ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

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.

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 : public acm114::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 };
```

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     // mutex to control access to the convergence criterion
8     pthread_mutex_t lock;
9
10    // constructor
11    Task(size_t workers, Grid & current, Grid & next) :
12       workers(workers), current(current), next(next), maxDeviation(0.0) {
13       pthread_mutex_init(&lock, 0);
14    }
15    // destructor
16    ~Task() {
17       pthread_mutex_destroy(&lock);
18    }
19 };
20
21 struct Context {
22    // thread info
23    size_t id;
24    pthread_t descriptor;
25    Task * task;
26 };
```

Threaded `Jacobi`: driving the update

```
28 void Jacobi::solve(Problem & problem) {
29    // initialize the problem
30    problem.initialize();
31    // do the actual solve
32    _solve(problem);
33    // compute and store the error
34    std::cout << " computing absolute error" << std::endl;
35    // compute the relative error
36    Grid & error = problem.error();
37    const Grid & exact = problem.exact();
38    const Grid & solution = problem.solution();
39
40    for (size_t j=0; j < exact.size(); j++) {
41       for (size_t i=0; i < exact.size(); i++) {
42          if (exact(i,j) == 0.0) {
43             error(i,j) = std::abs(solution(i,j));
44          } else {
45             error(i,j) = std::abs(solution(i,j) - exact(i,j))/exact(i,j);
46          }
47       }
48    }
49    std::cout << " --- done." << std::endl;
50    return;
51 }
```