# ACM/CS 114
# Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

# Improving the update loop

- the plan is to keep the workers alive and updating the grid while either we converge or `max_iterations` is reached
- the main thread
  - loops to spawn all the threads
  - and immediately enters a loop to harvest them
- the workers use a condition variable to synchronize among themselves
  - they iterate, updating the grid
  - grab a mutex, deposit their local maximum deviation from the last iterations, update a counter that records how many workers have completed their update, and release the lock
  - enter another critical section with the termination logic
    - everybody uses a condition variable to wait for the slowest worker
    - the slowest worker checks the convergence criterion and updates the termination flag, swaps the grid blocks and signals everybody else
    - if the termination flag is set, or if the maximum number of iterations has been reached, all threads exit

# Threaded `Jacobi`: the updated master thread

```
1  void Jacobi::_solve(Problem & problem) {
2     Grid & current = problem.solution();
3
4     // create and initialize temporary storage
5     Grid next(current.size());
6     problem.initialize(next);
7
8     // shared thread info
9     Task task(_workers, current, next);
10    // per-thread information
11    Context context[_workers];
12
13    // let's get going
14    std::cout << "jacobi: spawning " << _workers << " threads" << std::endl;
15    for (size_t tid=0; tid < _workers; tid++) {
16       context[tid].id = tid;
17       context[tid].task = &task;
18
19       int status = pthread_create(&context[tid].descriptor, 0, _update, &context[tid]);
20       if (status) {
21          throw ("error in pthread_create");
22       }
23    }
24    // harvest the threads
25    std::cout << "jacobi: harvesting " << _workers << " threads" << std::endl;
26    for (size_t tid = 0; tid < _workers; tid++) {
27          std::cout << "jacobi: waiting for thread " << tid << " to finish" << std::endl;
28          pthread_join(context[tid].descriptor, 0);
29          std::cout << "jacobi: thread " << tid << " done" << std::endl;
30       }
31    // done
32    std::cout << "jacobi: done." << std::endl;
33    return;
34 }
```

# Threaded `Jacobi`: updated thread data

```
35 struct Task {
36    // shared information
37    size_t workers; // the number of threads
38    double tolerance; // the covergence tolerance
39    Grid & current;
40    Grid & next;
41
42    bool done; // is there more work?
43    double maxDeviation; // the value
44    size_t contributions; // the number of threads that have deposited contributions
45    pthread_mutex_t gridUpdate_lock; //the mutex
46    pthread_cond_t gridUpdate_check;
47
48    Task(size_t workers, double tolerance, Grid & current, Grid & next) :
49       workers(workers), tolerance(tolerance), current(current), next(next),
50       done(false), maxDeviation(0.0), contributions(0),
51       gridUpdate_lock(), gridUpdate_check() {
52       // initialize the grid update lock
53       pthread_mutex_init(&gridUpdate_lock, 0);
54       pthread_cond_init(&gridUpdate_check, 0);
55    }
56
57    ~Task() {
58       pthread_mutex_destroy(&gridUpdate_lock);
59       pthread_cond_destroy(&gridUpdate_check);
60    }
61 };
```

# Threaded `Jacobi`: the main thread

```
1  void Jacobi::_solve(Problem & problem) {
2     Grid & current = problem.solution();
3
4     // create and initialize temporary storage
5     Grid next(current.size());
6     problem.initialize(next);
7
8     // shared thread info
9     Task task(_workers, _tolerance, current, next);
10    // per-thread information
11    Context context[_workers];
12    // spawn the threads
13    std::cout << "jacobi: spawning " << _workers << " workers" << std::endl;
14    for (size_t tid=0; tid < _workers; tid++) {
15       context[tid].id = tid;
16       context[tid].task = &task;
17
18       int status = pthread_create(&context[tid].descriptor, 0, _update, &context[tid]);
19       if (status) {
20          throw ("error in pthread_create");
21       }
22    }
23    // harvest the threads
24    for (size_t tid = 0; tid < _workers; tid++) {
25          pthread_join(context[tid].descriptor, 0);
26       }
27    // done
28    std::cout << "jacobi: done." << std::endl;
29    return;
30 }
```

# Threaded `Jacobi`: workers, part 1

```
31 // the threaded update
32 void * Jacobi::_update(void * arg) {
33    Context * context = static_cast<Context *>(arg);
34
35    size_t id = context->id;
36    Task * task = context->task;
37
38    const size_t workers = task->workers;
39    Grid & current = task->current;
40    Grid & next = task->next;
41
42    size_t maxIterations = (size_t) 1e4;
43    // iterate, updating the grid until done
44    for (size_t iteration = 0; iteration < maxIterations; iteration++) {
45       // thread 0: print an update
46       if (id == 0 && iteration % 100 == 0) {
47          std::cout << " " << iteration << std::endl;
48       }
49
50       double max_dev = 0.0;
51       // do an iteration step
52       // leave the boundary alone
53       // iterate over the interior of the grid
54       for (size_t j=id+1; j < current.size()-1; j+=workers) {
55          for (size_t i=1; i < current.size()-1; i++) {
56             next(i,j) = 0.25*(current(i+1,j)+current(i-1,j)+current(i,j+1)+current(i,j-1));
57             // compute the deviation from the last generation
58             double dev = std::abs(next(i,j) - current(i,j));
59             // and update the maximum deviation
60             if (dev > max_dev) {
61                max_dev = dev;
62             }
63          }
64       }
65       // done with the grid update
```

# Threaded `Jacobi`: workers, part 2

```
66          // grab the grid update lock
67          pthread_mutex_lock(&task->gridUpdate_lock);
68          // update the global maximum deviation
69          if (task->maxDeviation < max_dev) {
70             task->maxDeviation = max_dev;
71          }
72          // leave a mark
73          task->contributions++;
74          // bookkeeping at the end of the update
75          if (task->contributions == workers) {
76             // if i am the slowest worker
77             // swap the blocks between the two grids
78             Grid::swapBlocks(current, next);
79             // check covergence
80             if (task->maxDeviation < task->tolerance) {
81                std::cout
82                   << " +++ thread " << id << ": convergence in " << iteration << " iterations"
83                   <<std::endl;
84                task->done = true;
85             }
86             // reset our accounting and signal everybody
87             task->contributions = 0;
88             task->maxDeviation = 0;
89             pthread_cond_broadcast(&task->gridUpdate_check);
90          } else {
91             // all but the slowest wait here
92             pthread_cond_wait(&task->gridUpdate_check, &task->gridUpdate_lock);
93          }
94          // release
95          pthread_mutex_unlock(&task->gridUpdate_lock);
96          // check whether we are done
97          if (task->done) {
98             break;
99          }
100      }
101      return 0;
102   }
```

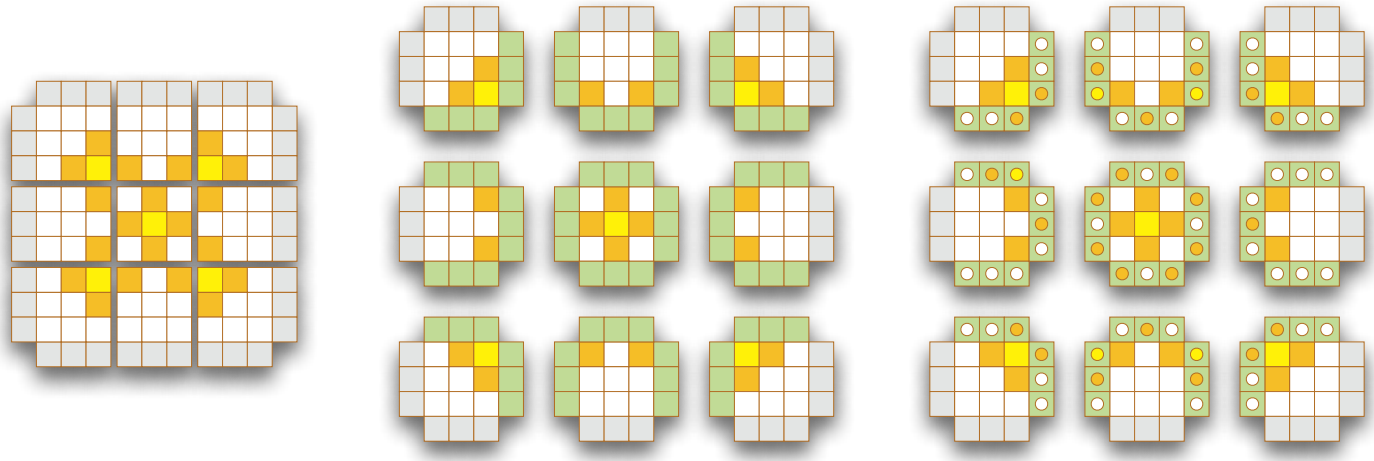
# Assessing the improved implementation

- the improved threading scheme is not much more complex
  - we keep track of how many threads have computed their grid update
  - the slowest worker check the convergence criterion and performs all the necessary bookkeeping
  - while everybody else waits
  - use `pthread_cond_brodacast` to wake the other workers
- here is the performance comparison for 10,000 iterations on a $1000 \times 1000$ grid on the same 8-core MacPro

| threads | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| previous($s$) | 413.306 | 211.050 | 109.509 | 98.279 | 74.087 |
| updated($s$) | 408.636 | 208.832 | 107.015 | 59.043 | 61.481 |

# Parallelization with MPI

- the MPI implementation will require careful data management
  - we must partition the mesh among processes
  - each process work on its own subgrid
    - it will allocate its own memory, for both actual data and the guard zones
    - it must locate its patch in physical space
  - communication is required every iteration
    - so that neighbors can synchronize their boundaries
    - think of the synchronization as a kind of boundary condition!
  - parallel convergence testing involves a collective operation



# A little bit of help

- `MPI` supports this common use case through a Cartesian *virtual topology*
  - a special communicator with a map from a $d$-dimensional virtual process grid to the normal linear process ranks
  - and local operations that enable you to discover the ranks of your virtual neighbors
  - there is even a special form of send/receive so that you don't have to worry about contention and race conditions during the boundary synchronization
- to create a Cartesian communicator

```
1 int MPI_Cart_create(MPI_Comm old,
2       int ndims, int* layout, int* periods, MPI_Comm* new);
```

- to find out the coordinates of a process in the virtual grid given its rank

```
1 int MPI_Cart_coords(MPI_Comm cartesian,
2       int rank, int ndims, int* coords);
```

- you can also find out the ranks of your neighbors

```
1 int MPI_Cart_shift(MPI_Comm cartesian,
2       int dimension, int shift, int* origin, int* neighbor);
```

# The MPI driver, part 1

```
26 int main(int argc, char* argv[]) {
27    int status;
28    // initialize mpi
29    status = MPI_Init(&argc, &argv);
30    if (status) {
31       throw("error in MPI_Init");
32    }
33    // get my rank in the world communicator
34    int worldRank, worldSize;
35    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
36    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
37    size_t processors = static_cast<size_t>(std::sqrt(worldSize));
38
39    // default values for our user configurable settings
40    size_t n = 9; // points per processor
41    size_t threads = 1;
42    double tolerance = 1.0e-3;
43
44    // read the command line
45    int command;
46    while ((command = getopt(argc, argv, "n:e:t:")) != -1) {
47       switch (command) {
48       // get the convergence tolerance
49       case 'e':
50          tolerance = atof(optarg);
51          break;
52       // get the grid size
53       case 'n':
54          n = (size_t) atof(optarg);
55          break;
56       // get the number of threads
57       case 't':
58          threads = (size_t) atoi(optarg);
59          break;
60       }
61    }
```

# The MPI driver, part 2

```
62    // print out the chosen options
63    if (worldRank == 0) {
64       for (int arg = 0; arg < argc; ++arg) {
65          std::cout << argv[arg] << " ";
66       }
67       std::cout
68          << std::endl
69          << "  grid size: " << n << std::endl
70          << "    workers: " << threads << std::endl
71          << "  tolerance: " << tolerance << std::endl;
72    }
73
74    // instantiate a problem
75    Example problem("cliche", 1.0, processors, n);
76
77    // instantiate a solver
78    Jacobi solver(tolerance, threads);
79    // solve
80    solver.solve(problem);
81    // save the results
82    Visualizer vis;
83    vis.csv(problem);
84
85    // initialize mpi
86    status = MPI_Finalize();
87    if (status) {
88       throw("error in MPI_Finalize");
89    }
90
91    // all done
92    return 0;
93 }
```

# The `Jacobi` declaration

```
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    // data members
12 private:
13    double _tolerance;
14    size_t _workers;
15
16    // disable these
17 private:
18    Jacobi(const Jacobi &);
19    const Jacobi & operator= (const Jacobi &);
20 };
```

# The `Problem` declaration

```
1  class acm114::laplace::Problem {
2     //typedefs
3  public:
4     typedef std::string string_t;
5     // interface
6  public:
7     string_t name() const;
8     inline MPI_Comm communicator() const;
9     inline int rank() const;
10    // access to my grid
11    inline Grid & solution();
12    inline const Grid & solution() const;
13    // interface used by the solver
14    virtual void initialize();
15    virtual void applyBoundaryConditions() = 0;
16    // meta methods
17 public:
18    Problem(string_t name, double interval, int processors, size_t points);
19    virtual ~Problem();
20    // data members
21 protected:
22    string_t _name;
23    double _delta, _x0, _y0;
24    int _rank, _size, _processors;
25    int _place[2];
26    MPI_Comm _cartesian;
27    Grid _solution;
28    // disable these
29 private:
30    Problem(const Problem &);
31    const Problem & operator= (const Problem &);
32 };
```

# The `Problem` constructor

```
94  Problem::Problem(
95     string_t name, double interval, int processors, size_t points) :
96     _name(name),
97     _delta(interval/((points-2)*processors+1)),
98     _x0(0.0), _y0(0.0),
99     _rank(0), _size(0), _processors(processors), _place(),
100    _cartesian(),
101    _solution(points) {
102
103    // build the intended layout
104    int layout[] = { processors, processors };
105    // find my rank in the world communicator
106    int worldRank;
107    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
108    // build a Cartesian communicator
109    int periods[] = { 0, 0 };
110    MPI_Cart_create(MPI_COMM_WORLD, 2, &layout[0], periods, 1, &_cartesian);
111    // check whether i can paritcipate
112    if (_cartesian != MPI_COMM_NULL) {
113       // get my rank in the cartesian communicator
114       MPI_Comm_rank(_cartesian, &_rank);
115       MPI_Comm_size(_cartesian, &_size);
116       // get my logical position on the process grid
117       MPI_Cart_coords(_cartesian, _rank, 2, &_place[0]);
118       // now compute my offset in physical space
119       _x0 = 0.0 + (points-2)*_place[0]*_delta;
120       _y0 = 0.0 + (points-2)*_place[1]*_delta;
121    } else {
122       // i was left out because the total number of processors is not a square
123       std::cout
124          << "world rank " << worldRank << ": not a member of the cartesian communicator "
125          << std::endl;
126    }
127 }
```

# The `Example` declaration

```
1  class acm114::laplace::Example : public acm114::laplace::Problem {
2     // interface
3  public:
4     virtual void applyBoundaryConditions();
5
6     // meta methods
7  public:
8     inline Example(
9        string_t name, double interval, int processors, size_t points);
10    virtual ~Example();
11
12    // disable these
13 private:
14    Example(const Example &);
15    const Example & operator= (const Example &);
16 };
```

# The implementation of `Jacobi::solve`, part 1

```
1  void Jacobi::solve(Problem & problem) {
2     // initialize the problem
3     problem.initialize();
4
5     // get a reference to the solution grid
6     Grid & current = problem.solution();
7     // build temporary storage for the next iterant
8     Grid next(current.size());
9
10    // put an upper limit on the number of iterations
11    size_t maxIterations = (size_t) 1e4;
12    for (size_t iteration = 0; iteration < maxIterations; iteration++) {
13       // print out a progress repot
14       if ((problem.rank() == 0) && (iteration % 100 == 0)) {
15          std::cout
16             << "jacobi: iteration " << iteration
17             << std::endl;
18       }
19       // enforce the boundary conditions
20       problem.applyBoundaryConditions();
21       // reset the local maximum change
22       double localMax = 0.0;
23       // update the interior of the grid
24       for (size_t j=1; j < next.size()-1; j++) {
25          for (size_t i=1; i < next.size()-1; i++) {
26             // the cell update
27             next(i,j) = .25*(current(i+1,j)+current(i-1,j)+current(i,j+1)+current(i,j-1));
28             // compute the change from the current cell value
29             double dev = std::abs(next(i,j) - current(i,j));
30             // and update the local maximum
31             if (dev > localMax) {
32                localMax = dev;
33             }
34          }
35       } // done with the grid update
```

# The implementation of `Jacobi::solve`, part 2

```
36          // swap the blocks of the two grids, leaving the solution in current
37          Grid::swapBlocks(current, next);
38          // compute global maximum deviation
39          double globalMax;
40          MPI_Allreduce(&localMax, &globalMax, 1, MPI_DOUBLE, MPI_MAX, problem.communicator());
41          // convergence check
42          if (globalMax < _tolerance) {
43             if (problem.rank() == 0) {
44                std::cout
45                   << "jacobi: convergence in " << iteration << " iterations"
46                   << std::endl;
47             }
48             break;
49          }
50          // otherwise
51       }
52       // when we get here, either we have converged or ran out of iterations
53       // update the fringe of the current grid
54       problem.applyBoundaryConditions();
55       // all done
56       return;
57    }
```

# Boundary conditions and data exchanges, part 1

```
1  void Example::applyBoundaryConditions() {
2     // a reference to my grid
3     Grid & g = _solution;
4     // my rank;
5     int rank = _rank;
6     // the ranks of my four neighbors
7     int top, right, bottom, left;
8     // get them
9     MPI_Cart_shift(_cartesian, 1, 1, &rank, &top);
10    MPI_Cart_shift(_cartesian, 0, 1, &rank, &right);
11    MPI_Cart_shift(_cartesian, 1, -1, &rank, &bottom);
12    MPI_Cart_shift(_cartesian, 0, -1, &rank, &left);
13
14    // allocate send and receive buffers
15    double * sendbuf = new double[g.size()];
16    double * recvbuf = new double[g.size()];
```

# Boundary conditions and data exchanges, part 2

```
17    // shift to the right
18    // fill my sendbuf with my RIGHT DATA BORDER
19    for (size_t cell=0; cell < g.size(); cell++) {
20       sendbuf[cell] = g(g.size()-2, cell);
21    }
22    // do the shift
23    MPI_Sendrecv(
24              sendbuf, g.size(), MPI_DOUBLE, right, 17,
25              recvbuf, g.size(), MPI_DOUBLE, left, 17,
26              _cartesian, MPI_STATUS_IGNORE
27              );
28    if (left == MPI_PROC_NULL) {
29       // if i am on the boundary, paint the dirichlet conditions
30       for (size_t cell=0; cell < g.size(); cell++) {
31          g(0, cell) = 0;
32       }
33    } else {
34       // fill my LEFT FRINGE with the received data
35       for (size_t cell=0; cell < g.size(); cell++) {
36          g(0, cell) = recvbuf[cell];
37       }
38    }
```

# Boundary conditions and data exchanges, part 4

```
39    // shift to the left
40    // fill my sendbuf with my LEFT DATA BORDER
41    for (size_t cell=0; cell < g.size(); cell++) {
42       sendbuf[cell] = g(1, cell);
43    }
44    // do the shift
45    MPI_Sendrecv(
46              sendbuf, g.size(), MPI_DOUBLE, left, 17,
47              recvbuf, g.size(), MPI_DOUBLE, right, 17,
48              _cartesian, MPI_STATUS_IGNORE
49              );
50    if (right == MPI_PROC_NULL) {
51       // if i am on the boundary, paint the dirichlet conditions
52       for (size_t cell=0; cell < g.size(); cell++) {
53          g(g.size()-1, cell) = 0;
54       }
55    } else {
56       // fill my RIGHT FRINGE with the received data
57       for (size_t cell=0; cell < g.size(); cell++) {
58          g(g.size()-1, cell) = recvbuf[cell];
59       }
60    }
```

# Boundary conditions and data exchanges, part 5

```
62    // shift up
63    // fill my sendbuf with my TOP DATA BORDER
64    for (size_t cell=0; cell < g.size(); cell++) {
65       sendbuf[cell] = g(cell, g.size()-2);
66    }
67    // do the shift
68    MPI_Sendrecv(
69              sendbuf, g.size(), MPI_DOUBLE, top, 17,
70              recvbuf, g.size(), MPI_DOUBLE, bottom, 17,
71              _cartesian, MPI_STATUS_IGNORE
72              );
73    if (bottom == MPI_PROC_NULL) {
74       // if i am on the boundary, paint the dirichlet conditions
75       for (size_t cell=0; cell < g.size(); cell++) {
76          g(cell, 0) = std::sin((_x0 + cell*_delta)*pi);
77       }
78    } else {
79       // fill my BOTTOM FRINGE with the received data
80       for (size_t cell=0; cell < g.size(); cell++) {
81          g(cell, 0) = recvbuf[cell];
82       }
83    }
```

# Boundary conditions and data exchanges, part 6

```
84    // shift down
85    // fill my sendbuf with my BOTTOM DATA BORDER
86    for (size_t cell=0; cell < g.size(); cell++) {
87       sendbuf[cell] = g(cell, 1);
88    }
89    // do the shift
90    MPI_Sendrecv(
91              sendbuf, g.size(), MPI_DOUBLE, bottom, 17,
92              recvbuf, g.size(), MPI_DOUBLE, top, 17,
93              _cartesian, MPI_STATUS_IGNORE
94              );
95    if (top == MPI_PROC_NULL) {
96       // if i am on the boundary, paint the dirichlet conditions
97       for (size_t cell=0; cell < g.size(); cell++) {
98          g(cell, g.size()-1) =
99             std::sin((_x0 + cell*_delta)*pi) * std::exp(-pi);
100      }
101   } else {
102      // fill my TOP FRINGE with the received data
103      for (size_t cell=0; cell < g.size(); cell++) {
104         g(cell, g.size()-1) = recvbuf[cell];
105      }
106   }
107
108   return;
109}
```