

ACM/CS 114

Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

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 }
```

Threaded Jacobi: the master thread

```
52 void Jacobi::_solve(Problem & problem) {
53     Grid & current = problem.solution();
54
55     // create and initialize temporary storage
56     Grid next(current.size());
57     problem.initialize(next);
58
59     // shared thread info
60     Task task(_workers, current, next);
61     // per-thread information
62     Context context[_workers];
63
64     // let's get going
65     std::cout << "jacobi: tolerance=" << _tolerance << std::endl;
66
67     // put an upper bound on the number of iterations
68     const size_t max_iterations = (size_t) 1.0e4;
```

Threaded Jacobi: the master thread, part 2

```
69 for (size_t iterations = 0; iterations < max_iterations; iterations++) {
70     if (iterations % 100 == 0) {
71         std::cout << " " << iterations << std::endl;
72     }
73     // reset the maximum deviation
74     task.maxDeviation = 0.0;
75     // spawn the threads
76     for (size_t tid=0; tid < _workers; tid++) {
77         context[tid].id = tid;
78         context[tid].task = &task;
79
80         int status = pthread_create(&context[tid].descriptor, 0, _update, &context[tid]);
81         if (status) {
82             throw ("error in pthread_create");
83         }
84     }
85     // harvest the threads
86     for (size_t tid = 0; tid < _workers; tid++) {
87         pthread_join(context[tid].descriptor, 0);
88     }
89
90     // swap the blocks between the two grids
91     Grid::swapBlocks(current, next);
92     // check convergence
93     if (task.maxDeviation < _tolerance) {
94         std::cout << " ### convergence in " << iterations << " iterations!" << std::endl;
95         break;
96     }
97 }
98 std::cout << " --- done." << std::endl;
99
100 return;
101 }
```

Threaded Jacobi: update in the worker threads

```
102 void * Jacobi::_update(void * arg) {
103     Context * context = static_cast<Context *>(arg);
104
105     size_t id = context->id;
106     Task * task = context->task;
107
108     size_t workers = task->workers;
109     Grid & current = task->current;
110     Grid & next = task->next;
111     pthread_mutex_t lock = task->lock;
112
113     double max_dev = 0.0;
114     // do an iteration step
115     // leave the boundary alone
116     // iterate over the interior of the grid
117     for (size_t j=id+1; j < current.size()-1; j+=workers) {
118         for (size_t i=1; i < current.size()-1; i++) {
119             next(i,j) = 0.25*(current(i+1,j)+current(i-1,j)+current(i,j+1)+current(i,j-1));
120             // compute the deviation from the last generation
121             double dev = std::abs(next(i,j) - current(i,j));
122             // and update the maximum deviation
123             if (dev > max_dev) {
124                 max_dev = dev;
125             }
126         }
127     }
128
129     // grab the lock and update the global maximum deviation
130     pthread_mutex_lock(&lock);
131     if (task->maxDeviation < max_dev) {
132         task->maxDeviation = max_dev;
133     }
134     pthread_mutex_unlock(&lock);
135
136     return 0;
137 }
```


Assessing the threaded implementation

- ▶ the implemented synchronization scheme is very simple
 - ▶ each grid update step spawns some number of workers to update a subset of the cells
 - ▶ the workers are harvested after the grid is updated
 - ▶ the main thread checks for convergence
 - ▶ if another iteration is required, a new set of workers is spawned
- ▶ the simplicity of this strategy comes at a cost
 - ▶ *scalability* suffers when the overhead of creating and harvesting threads is comparable to amount of work done by each thread
 - ▶ for low thread counts, it is still an overall win, since the time to solution decreases and the machine utilization is better
 - ▶ but as the number of threads increases, the program becomes *slower*
 - ▶ timing a 100×100 grid to convergence on a recent MacPro

| threads | 1 | 2 | 4 | 8 | 16 |
|---------|-------|-------|-------|-------|-------|
| time(s) | 4.367 | 2.517 | 1.918 | 1.937 | 3.537 |

- ▶ and 10,000 iterations of a 1000×1000 grid

| threads | 1 | 2 | 4 | 8 | 16 |
|---------|---------|---------|---------|--------|--------|
| time(s) | 413.306 | 211.050 | 109.509 | 98.279 | 74.087 |

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 }
```