The cutting-plane algorithm: synchronous and asynchronous master-worker parallelism in Julia

Miles Lubin
MIT Operations Research Center

Julia IAP Tutorial, January 15-16, 2013

1 The cutting plane algorithm

This section is meant to be a self-contained presentation of the cutting-plane algorithm for convex optimization. It assumes only a background in calculus. We present the one-dimensional case for simplicity. Most of this section can be skipped or skimmed; the fun starts in the next section. The essential parts of this section are the statement of Algorithm 1 and the paragraph that follows it.

Definition. A function $f : \mathbb{R} \to \mathbb{R}$ is *subdifferentiable* if for all $x' \in \mathbb{R}$, there exists a scalar $g_{x'}$ such that for all $x \in \mathbb{R}$,

$$f(x) \geq g_{x'}(x - x') + f(x') = g_{x'}x + (f(x') - g_{x'}x').$$

Any such $g_{x'}$ is called a *subderivative* of f at x'.

The property of subdifferentiability is nearly equivalent to *convexity*. The above definition means that at every point in the domain of f, we can draw a line tangent to the graph of f which will remain under the graph at all points. If f is also differentiable, then $g_{x'}$ is exactly the derivative $f'(x)$, but the definition holds for functions which may not be differentiable everywhere. An example is the absolute value function $f(x) = |x|$. For fun, you may want to find the set of valid $g_{x'}$ for $f(x) = |x|$ at $x' = 0$.

We move on immediately to the field of convex optimization. Let $f_1, f_2, \ldots, f_n$ be given subdifferentiable functions. The problem we are interested in solving is

$$\text{minimize}_{x\in\mathbb{R}} \sum_{i=1}^{n} f_i(x), \tag{1}$$

that is, find a value of $x \in \mathbb{R}$ that minimizes the function $f(x) := \sum_{i=1}^{n} f_i(x)$.

Suppose we have a black box procedure for each f_i that, given any point x, returns a tuple $(f_i(x), g_{i,x})$; that is, it computes both the value of f at x and a corresponding subderivative. Now suppose that we've called this procedure at a sequence of points $x^1, x^2, \ldots, x^k$ for each i. Using the set of subderivatives, we build a *model* of each f_i, which we define as

$$m_i^k(x) := \max\{g_{i,x^1}(x-x^1)+f_i(x^1), g_{i,x^2}(x-x^2)+f_i(x^2), \ldots, g_{i,x^k}(x-x^k)+f_i(x^k)\}. \tag{2}$$

This model is piecewise linear and is a lower estimate of f_i. The idea of the *cutting-plane* algorithm is to use the minimum of the model, something which is easy to compute, as a guess for the minimum of f. In particular, the next trial point x^{k+1} is set to the minimizer of $\sum_{i=1}^{n} m_i^k(x)$, the black boxes are then called again, and we repeat until we're satisfied with our solution. An iteration of the algorithm is illustrated in Figure 1, and the algorithm is formally stated in Algorithm 1.

Algorithm 1 Cuting plane algorithm – Serial

Input: Subdifferentiable functions $f_1, \ldots, f_n$. Starting point x^1.

1: $k \leftarrow 1$
2: **repeat**
3: **for** $i = 1$ **to** n **do**
4: Compute function value and subderivative $(f_i(x^k), g_{i,x^k})$ at x^k.
5: **end for**
6: $x^{k+1} \leftarrow \text{argmin}_x \sum_{i=1}^{n} m_i^k(x)$, with $m_i^k(x)$ defined by (2)
7: $k \leftarrow k + 1$
8: **until** convergence

An important observation is that the computation of the function values and subderivatives may be performed independently for each i in the **for** loop. In many applications this computation is expensive, hence there is a real potential for speedup if we parallelize this loop. We will now investigate implementing and parallelizing this algorithm in Julia.

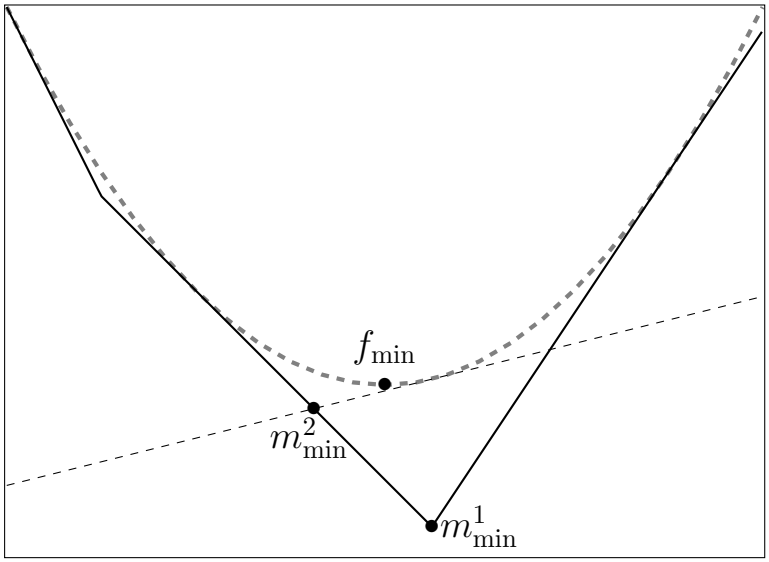

Figure 1: An iteration of the cutting-plane algorithm applied to $f = x^2$ ($n = 1$). The point $m^1_{\min}$ is the minimizer of the current model. We then re-evaluate f and add a *cut* (dashed line) to the model corresponding to the new (sub)derivative. The point $m^2_{\min}$ minimizes the new model. The point $f_{\min}$ is the true minimizer.

2 Julia implementation

In the previous section we presented the mathematical algorithm. Here we'll work with an actual implementation on a small toy problem and experiment with parallel computing in Julia. First, retrieve the code from here. The `Tutorial.jl` file contains the code we will be referring to, while `ExampleSolution.jl` contains potential solutions to some of the exercises.

Our toy problem will be based on the functions $f_i(x) = \frac{1}{2}(x - i)^2$. With $f(x) = \sum_{i=1}^n f_i(x)$ as before, we have $f'(x) = \sum_{i=1}^n (x - i)$. Using basic calculus, we know already that the minimizer of f is at $x = \frac{1}{n}\sum_{i=1}^n i = (n + 1)/2$. Before running, install the Optim package, `Pkg.add("Optim")`. We can now run the the code by `julia Tutorial.jl`. The output should appear as follows:

```
Solving model with n = 10, initial solution: [0.5]
Optimal solutuon should be 5.5
Model minimizer: [10.0]
Model minimizer: [5.25]
Model minimizer: [7.625]
...
Model minimizer: [5.49995]
Converged in 15 iterations
elapsed time: 78.94296097755432 seconds
```

Let's open up the code. The function `f` implements the black-box calculation of the subderivatives. Notice the line with `sleep`, which we use to simulate a variably expensive black-box. The `evalmodel` function evaluates $\sum_{i=1}^{n} m_i^k(x)$ at `x` given the stored `subderivatives`.

The `cpserial` function implements Algorithm 1 in serial. The parameter N is our cursive n. The line `results = map(f,[(cur[1],i) for i in 1:N])` corresponds to loop of lines 3–5 of the algorithm. The call to `Optim.optimize` corresponds to line 6. Now for the fun stuff.

Exercise 1. Insert code to plot $\sum_{i=1}^{n} m_i^k(x)$ versus $\sum_{i=1}^{n} f_i(x)$ at each iteration. You may use Winston or any other package.

Exercise 2. Review the Julia documentation on parallel computing. Modify the code so that the subderivatives are computed in parallel. *(Hint: it only requires a small change.)*

Now try running `julia -p 2 Tutorial.jl`. If you get an error like

```
From worker 2:        exception on 2: f not defined
```

this means that `f` was only defined on process 1, but you're trying to call it on process 2. Check the documentation or `ExampleSolution.jl` for a fix.

Exercise 3. Change n to 20 and run `julia -p nproc Tutorial.jl` for nproc = 1, 2, 3, 5, 10 on a machine with sufficiently many cores. Because of the explicit randomness in computing times, you may want to repeat and take average execution times. Do you observe speedup? Compute the parallel efficiency (observed speedup divided by perfect speedup) from 1 to 10 processes.

Exercise 4. With `nproc` = 1, record and print out the total time spent in the *serial bottleneck* of calling `Optim.optimize`. Use Amdahl's law to compute the theoretical maximum possible speedup. Was this achieved in the previous exercise?

Advanced Exercise. Replace `Optim.optimize` with a faster approach for minimizing the model function. Hint: this problem can be solved efficiently by using Linear Programming. *(Note: you should complete the rest of the tutorial before attempting this.)*

Exercise 5. Change the random `sleep` time inside `f` to a deterministic time, say 0.6 seconds. Repeat Exercise 3. Does the parallel efficiency change?

You should have observed that the imbalance in computation time does have a significant effect on the parallel speedups observed. Let's visualize this effect.

Exercise 6. Modify the code so that each process records the intervals during which it spends inside the function `f`. Plot these intervals in some reasonable form, using Figure 2 as an example (source code in `ExampleSolution.jl`). This figure could be improved by plotting the time spent inside `Optim.optimize` as well.

Optional Exercise. Try using `Base.pmap_static` (take a look at its definition in `julia/base/multi.jl`). How does this affect performance and the previous discussion?

3 Asynchronous algorithm

The asynchronous variant of the cutting-plane algorithm aims to reduce the idle time of the worker processes by eliminating the bottleneck of resolving the model. While previously the parallelism was hidden by high-level Julia functions, now we will need to explicitly consider the master process and the set of workers.

The idea of the asynchronous algorithm is to minimize the model function using *incomplete information* in order to generate new tasks to feed to workers. That is, instead of waiting for all subderivatives to be computed at a given candidate solution, we generate a new candidate solution (and a

Figure 2: Plots of process activity over time running on 5 processes (4 worker processes). Variable subproblem evaluation time on left, deterministic on right. Black lines denote intervals of activity, white space indicates inactivity.

full set of new subgradient evaluation tasks) once some proportion σ of the subderivatives have been computed. These tasks may be immediately fed to workers, hence workers will spend less time waiting for new tasks. Further description is beyond the scope of this tutorial; see [2].

The asynchronous algorithm is implemented as `asyncversion` in `Tutorial.jl`. It is worth spending a few minutes trying to understand the code. The use of Julia tasks with `@spawnlocal` and `@sync` is not intuitively obvious (at least to me). As a starting point, compare with the implementation of `pmap` in `julia/base/multi.jl`.

Exercise 7. Run both the original parallel code and the asynchronous code with $n = 100$ and nprocs $= 5, 10, 20$, under the original random computation time model (`max(0.1,0.2+1*randn())`). Is the asynchronous code faster? Note that because less information is used to generate the iterates, the asynchronous version typically requires more iterations. Compare both the total execution time and the average rate of subproblems solved per second.

Exercise 8. Experiment with different random models for computation time.

What happens if you increase or decrease the variance (the coefficient of `randn()`)?

Advanced Exercise. Currently `Optim.optimize` is called by the same process that manages the workers. Modify the code so that new tasks can be distributed in the middle of calls to `Optim.optimize`.

References

[1] J. B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms*, volume I-II. Springer-Verlag, Germany, 1993.

[2] Jeff Linderoth and Stephen Wright. Decomposition algorithms for stochastic programming on a computational grid. *Computational Optimization and Applications*, 24(2):207–250, 2003.