
Jonathan Eckstein
Rutgers University, Piscataway, NJ, USA
(formerly of Thinking Machines Corporation)
(also consultant for Sandia National Laboratories)
• I am not primarily a computer scientist
• I am not primarily a computer scientist

• I am “user” interested in implementing a particular (large) class of applications
• I am not primarily a computer scientist

• I am “user” interested in implementing a particular (large) class of applications
• I am not primarily a computer scientist

• I am “user” interested in implementing a particular (large) class of applications

• Well, a *relatively* sophisticated user...
Optimization

• Minimize some objective function of many variables

• Subject to constraints, for example
  o Equality constraints (linear or nonlinear)
  o Inequality constraints (linear or nonlinear)
  o General conic constraints (e.g. cone of positive semidefinite matrices)
  o Some or all variables integral of binary

• Applications
  o Engineering and system design
  o Transportation/logistics network planning and operation
  o Machine learning
  o Etc., etc...
**Overgeneralization: Kinds of Optimization Algorithms**

- For “easy” but perhaps very large problems
  - All variables typically continuous
  - Either looking only for local optima, or we know any local optimum is global (convex models)
  - Difficulty may arise extremely large scale

- For “hard” problems
  - Discrete variables, and not in a known “easy” special class like shortest path, assignment, max flow, etc., or...
  - Looking for a provably global optimum of a nonlinear continuous problem with local optima
Algorithms for “Easy” Problems

- Popular standard methods (not exhaustive!) that do not assume a particular block or subsystem structure
  - Active set (for example, simplex)
  - Newton barrier (“interior point”)
  - Augmented Lagrangian
- *Decomposition* methods (many flavors) - exploit some kind of high-level structure
Non-Decomposition Methods: Active Set

- Canonical example: simplex
- Core operation: a pivot
  - Have a usually sparse nonsingular matrix $B$ factored into $LU$
  - Replace one column of $B$ with a different sparse vector
  - Want to update the factors $LU$ to match
- The general sparse case has resisted effective parallelization
- Dense case may be effectively parallelized (E et al. 1995 on CM-2, Elster et al. 2009 for GPU’s)
- Some special cases like just “box” constraints are also fairly readily parallelizable
Non-Decomposition Methods: Newton Barrier

- Avoid combinatorics of constraint intersections
  - Use a barrier function to “smooth” the constraints (often in a “primal-dual” way)
  - Apply one iteration of Newton’s method to the resulting nonlinear system of equations
  - Tighten the smoothing parameter and repeat
- Number of iterations grows weakly with problems size
- Main work: solve a linear system involving
  \[ M = \begin{bmatrix} H & -J^T \\ J & D \end{bmatrix} \]
  - System becomes increasingly ill-conditioned
  - Must be solved to high accuracy
Non-Decomposition Methods: Newton Barrier

- Parallelization of this algorithm class is dominated by linear algebra issues.
- Sparsity pattern and factoring of $M$ is in general more complex than for the component matrices $H$, $J$, etc.
- Many applications generate sparsity patterns with low-diameter adjacency graphs.
  - PDE-oriented domain decomposition approaches may not apply.
- Iterative linear methods can be tricky to apply due to the ill-conditioning and need for high accuracy.
- A number of standard solvers offer SMP parallel options, but speedups tend to be very modest (i.e. 2 or 3).
Non-Decomposition Methods: Augmented Lagrangians

- Smooth constraints with a penalty instead of a barrier; use Lagrange multipliers to “shift” the penalty; do not have to increase penalty level indefinitely

- Creates a series of subproblems with no constraints, or much simpler constraints

- Subproblems are nonlinear optimizations (not linear systems)

- But may be solved to low accuracy

- Parallelization efforts focused on decomposition variants, but the standard, basic approach may be parallelizable
Decomposition Methods

- Assume a problem structure of relatively weakly interacting subsystems
  - This situation is common in large-scale models

- There are many different ways to construct such methods, but there tends to be a common algorithmic pattern:
  - Solve a perturbed, independent optimization problem for each subsystem (potentially in parallel)
  - Perform a coordination step that adjusts the perturbations, and repeat

- Sometimes the coordination step is a non-trivial optimization problem of its own - a potential Amdahl’s law bottleneck

- Generally, “tail convergence” can be poor

- Some successful parallel applications, but highly domain-specific
Branch and bound is the most common algorithmic structure. Integer programming example:

\[
\begin{align*}
\text{min} & \quad c^T x \\
\text{ST} & \quad Ax \leq b \\
& \quad x \in \{0,1\}^n
\end{align*}
\]

- Relax the \( x \in \{0,1\}^n \) constraint to \( 0 \leq x \leq 1 \) and solve as an LP
- If all variables come out integer, we’re done
- Otherwise, divide and conquer: choose \( j \) with \( 0 < x_j < 1 \) and branch

\[
\begin{align*}
x_j = 0 & & \quad \text{node} \\
x_j = 1 & & \quad \text{node}
\end{align*}
\]
Branch and Bound Example Continued

• Loop: pool of *subproblems* with subsets of fixed variables
  o Pick a subproblem out of the pool
  o Solve its LP
  o If the resulting objective is worse than some known solution, throw it away (prune)
  o Otherwise, divide the subproblem by fixing another variable and put the resulting children back in the pool

• The algorithm may be generalized/abstracted to many other settings
  o Including global optimization of continuous problems with local minima
Branch and Bound

- In the worst case, we will enumerate an exponentially large tree with all possible solutions at the leaves.
- Thus, relatively small amounts of data can generate very difficult problems.
- If the bound is “smart” and the branching is “smart”, this class of algorithms can nevertheless be extremely useful and practical.

  - For the example problem above, the LP bound may be greatly strengthened by using polyhedral combinatorics - adding additional linear constraints implied by combining $x \in \{0,1\}^n$ and $Ax \leq b$.

  - Clever choices of branching variable or different ways of branching have enormous value.
Parallelizing Branch and Bound

- Branch and bound is a “forgiving” algorithm to parallelize
  - Idea: work on multiple parts of the tree at the same time
  - But trees may be highly unbalanced and their shape is not predictable
    - A variety of load-balancing approaches can work very well
- A number object-oriented parallel branch-and-bound frameworks/libraries exist, including
  - PEBBL/PICO (E et al.)
  - ALPS/BiCePS/BLIS (Ralphs et al.)
  - BOB (Lecun et al.)
  - OOBB (Gendron et al.)
- Most production integer programming solvers have an SMP parallel option: CPLEX, XPRESS-MP, GuRoBi, CBC
Effectiveness of Parallel Branch and Bound

• I have seen examples with near-linear speedup through hundreds of processors, and it should scale up further

• Sometimes there are even apparently superlinear speedup anomalies (for which there are reasonable explanations)

• I have also seen disappointing speedups. Why?
  o Non-scalable load balancing techniques
    ▪ Central pool for SMPs or master-slave
  o Task granularity not matched to platform
    ▪ Too fine ⇒ excessive overhead
    ▪ Too coarse ⇒ too hard to balance load
  o Ramp-up/ramp-down issues
  o Synchronization penalties from requiring determinism
Big Picture: Where We Are (Both “Hard” and “Easy” Problems)

• Most numerical optimization is done by large, encapsulated solvers / callable libraries which encapsulate the expertise of numerical optimization experts

• Models are often passed to these libraries using specialized modeling languages
  - Leading example: AMPL
  - Digression - challenge to merge these optimization model description languages with a usable procedural language
Monolithic Solvers and Callable Libraries

• These libraries / solvers have some parameters (often poorly understood by our users), but are otherwise fairly monolithic

• Results
  o Minimal or no speedups on LP and other continuous problems
  o Moderate speedups on hard integer problems
  o Usually available only on SMP platforms

• Why?
  o “Hard” problems: we need to assemble the right teams
  o “Easy” problems: we need a different approach
“Hard” Problems

- For branch-and-bound-related algorithms, the monolithic approach can take us much farther than we are today.
- Today’s parallel implementations are somewhat weak, but the right combination of domain knowledge and implementation knowledge should yield monolithic solvers that could exploit parallelism far better.

“Easy” (But Huge) Problems

- The monolithic approach will not get us much farther.
- Fully analyzing the structure of a gigantic problem and picking the optimal problem partitioning & solution algorithm is a tall order.
  - To work effectively, a monolithic parallel solver must analyze the input model much more deeply than a serial one.
New Approaches for Large “Easy” Problems

1. Better decomposition algorithms - but results will probably be application-specific

2. A “toolkit” approach for non-decomposition algorithms
   - Provide high-quality, rigorous fundamental optimization algorithms
     - Avoid user *ad hoc* approaches and “reinventing the wheel” for basic optimization algorithms
   - But give users control over data layout and function / gradient evaluation to best suit their application
   - Somewhat similar in spirit to CMSSL
   - Could still plug this framework to a monolithic solver that attempts to analyze problem structure and find good decomposition strategies
A Particular Approach I’m Working On

- “Outer loop”: augmented Lagrangian with a relative error criterion (E + Silva 2010)
  - Generates a sequence of nonlinear box-constrained subproblems solved to gradually increasing accuracy
- “Inner loop”: CG-DESCENT/ASA (Hager and Zhang 2005/2006), with minor modifications for parallelism

User provides
- “Primal layout”: assignment of variables to processors (some may be replicated on multiple processors)
- “Dual layout”: assignment of constraints to processors (some may be replicated on multiple processors)
- Function / gradient evaluators adapted to these layouts

Asking for parallelization help from user, ...
  - but in a natural application domain (not matrix factoring)
Programming Environments

- What framework should we implement this in?
- What framework should we ask our users to employ for the function / gradient evaluator?
- What approach would make applications as portable as possible?

- C++ / MPI ? (what I do most of my current work in)
- CUDA ?
- OpenCL ?
- Yecch...
Programming Environments

• These environments are the assembly languages of parallelism

• Literally:
  o CUDA and OpenCL resemble C/PARIS, the assembly language of the CM-2

• Conceptually:
  o Low level of abstraction
  o Lots of clutter
  o Will only work (well) on certain families of platforms
Wish List

• We need a “C of parallelism”
  o Something that allows reasonably low level control and is built for performance
  o But also supports a proper level of abstraction
  o ... and is not heavily platform dependent


• Note:
  o The #1 linear programming code of the 60’s-80’s (MPSX) was written in IBM/360 assembler
  o Competitors were in FORTRAN
  o In the 80’s, they were swept aside by fast C codes

• If the right tools are there, they will get used
Wish List Continued

- Ideally, should be a superset of a recognizable standard language
  - We’ll need users to code modules for us
  - Otherwise, it should interface easily to standard languages

- Aggregate operation support
  - Witness popularity of MATLAB, despite its many flaws
  - Also SciPy

- But also some kind of task / nested parallelism
  - More than just data parallelism and aggregate operations

- “Locality” support
  - Must express more than a flat global address space