Object-Parallel Solution of Large-Scale Lasso Problems

Jonathan Eckstein
Rutgers University, NJ, USA

joint work with

György Mátyásfalvi (doctoral student)
Rutgers University, NJ, USA

RUTGERS

RUTGERS
Rutgers Business School
Newark and New Brunswick

Funded in part by US National Science Foundation Grant CCF-1115638

Why “Object-Parallel”?

- Current trend: growth in computer power is mainly through increased parallelism
    - Speed of executing a single thread is not improving much
- To solve huge problem instances, including from “big data”, we need scalably parallel algorithms and implementations
- The best ways to exploit parallelism are often dictated by problem instance data structure and are thus highly application-dependent

Suggests we need optimization environments in which

- The application guides the use of parallelism as flexibly as possible
- The underlying algorithms can adapt easily to very different data representations (across distributed memory)

An “Object-Parallel” Approach to Optimization

- An “inside-out” approach:

- User determines the most appropriate data layout for a given application
  - How variables are divided (and perhaps replicated) among processors
  - And similarly for constraints
  - User provides (presumably efficient) calculations of objectives, constraints, and their derivatives using these layouts

- The optimization code is an algorithmic “template” that is applied to the user’s data representation

What Kind of Optimization Algorithms Seem Suitable to Such a Context?

- Methods that use only first derivatives
  - Structure of Hessians may be too complex for users to contemplate
- Methods that do not directly solve linear systems:
  - Structure of factors of matrices like

$$\begin{bmatrix} H & J^{\mathsf{T}} \\ J & 0 \end{bmatrix}$$

  may be too much for users to think about, or may defeat simple parallelism within calculating gradients/Jacobians

- Restrictive, but not impossible:
  - These are largely the same restrictions that apply to most approaches to solving very large-scale problem instances

A Possible General NLP Approach (Digression): Relative-Error First-Order Approximate Augmented Lagrangians

- Quadratic augmented Lagrangian “outer loop”
- Subproblems created within this outer loop are either unconstrained or box-constrained
- Solve the subproblems using an efficient box-constrained first-order method
- Use an approximation criterion so that relatively little effort can be expended on each subproblem…
  - …maybe just a handful of gradient / CG steps

- E & Silva (2013) suggest an algorithm of this form
  - Show global convergence in convex case
  - Encouraging empirical performance results (nonconvex)

Relative-Error Augmented Lagrangians: Background

$$\boxed{\begin{array}{lll} \min & f(x) & \\ \mathrm{ST} & g_i(x) \le 0 & i = 1,\ldots,m_1 \\ & h_i(x) = 0 & i = 1,\ldots,m_2 \\ & l \le x \le u & \end{array}}$$

or just

$$\boxed{\begin{array}{ll} \min & f(x) \\ \mathrm{ST} & g(x) \le 0 \\ & h(x) = 0 \\ & l \le x \le u \end{array}}$$

Basic algorithm:

$$x^k \in \mathop{\arg\min}_{x \in \Re^n,\, l \le x \le u} \left\{ f(x) + \tfrac{1}{2c_k}\left(\sum_{i=1}^{m} \max\left\{0, p_i^{k-1} + c_k g_i(x)\right\}^2 + \left\| q^{k-1} + c_k h(x) \right\|^2 \right) \right\}$$

$$p_i^k = \max\left\{0, p_i^{k-1} + c_k g_i(x^k)\right\} \qquad i = 1,\ldots,m$$

$$q^k = q^{k-1} + c_k h(x^k)$$

- But don’t want to invest too much effort in subproblem minimization when multiplier estimates are poor
- Let $L_c(x,p,q) = f(x) + \tfrac{1}{2c}\left(\sum_{i=1}^{m} \max\left\{0, p_i + c g_i(x)\right\}^2 + \left\| q + c h(x) \right\|^2 \right)$

Relative Error Criterion (E & Silva 2012)

Violation of KKT

$$y^k \in \partial_x L_{c_k}(x^k, p^{k-1}, q^{k-1})$$

$$\frac{2}{c_k}\left|\left\langle w^{k-1} - x^k, y^k \right\rangle\right| + \left\|y^k\right\|^2 \le \sigma\left(\left\|\min\left\{\tfrac{1}{c_k} p^{k-1}, -g(x^k)\right\}\right\|^2 + \left\|h(x^k)\right\|^2\right)$$

Scalar $\sigma \in [0,1)$

$$p_i^k = \max\left\{0, p^{k-1} + c_k g_i(x^k)\right\} \qquad i = 1, \ldots, m$$

$$q^k = q^{k-1} + c_k h(x^k)$$

Weird auxiliary sequence

$$w^k = w^{k-1} - c_k y^k$$

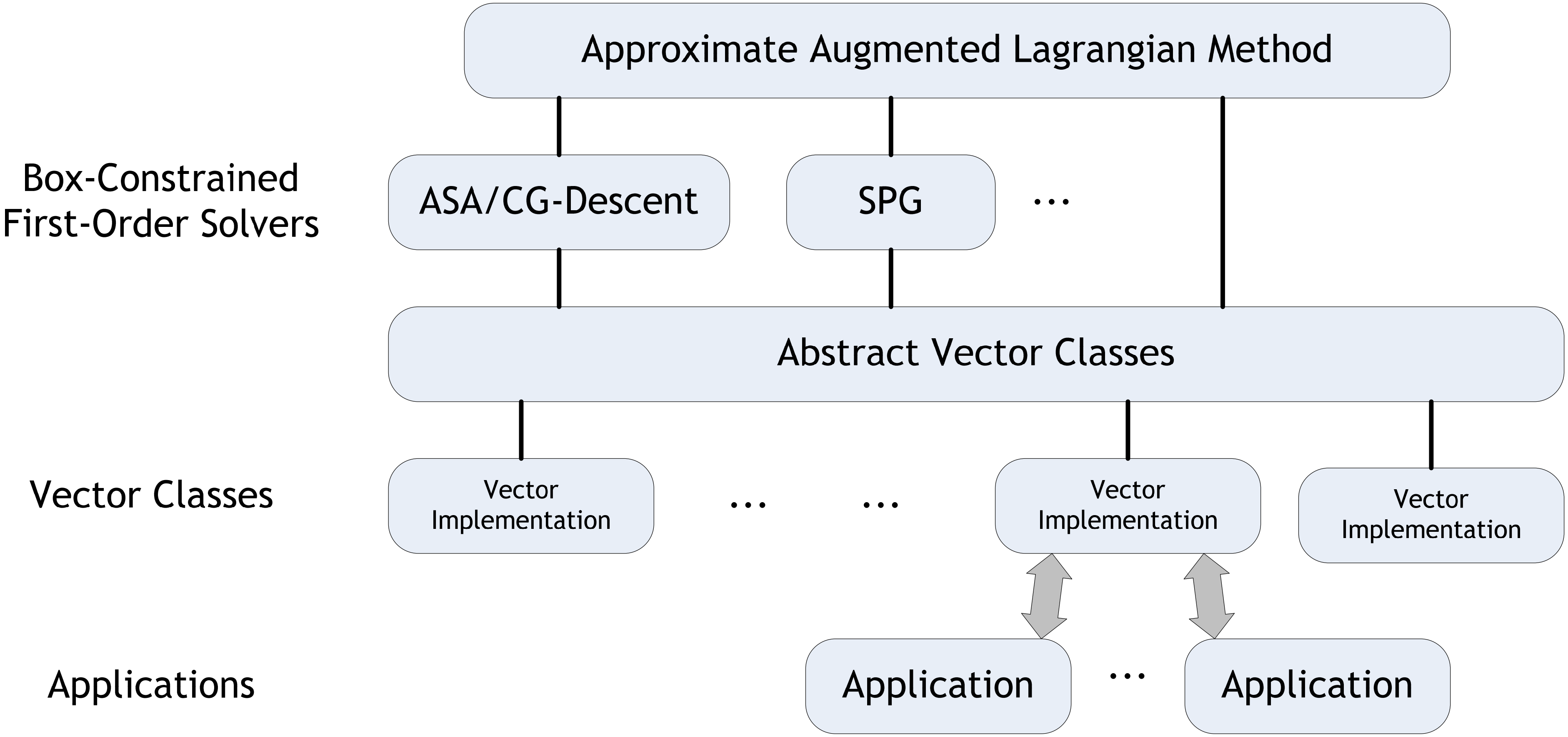
The “FOAAL” Project

- The relative-error augmented Lagrangian approach seems like a reasonably good algorithm in serial — that is, likely to be competitive for at least some problem classes

- So, try to build this approach into object-parallel software:

- FOAAL: First-Order Approximate Augmented Lagrangian

FOAAL Software Architecture (C++)



FOAAL Software Architecture Rationale

- Vector classes encapsulate various vector representations
  - Serial representations of various kinds
  - …including encapsulating various forms of BLAS
  - …including parallel, such as Epetra (Heroux 2004, 2006)

- Abstract vector classes provide operator overloading so that the core optimization algorithms may be expressed with (almost) MATLAB-like simplicity
  - Unlike most high-performance optimization codes, in which the underlying algorithm is obscured by details
  - Without sacrificing much performance
  - Flexibly with respect underlying vector representation
  - Possibly in parallel (upper-level code running synchronously on all processors)

Operator Overloading

- When at least one of **`a`** and **`b`** has a user-defined type, C++ allows the definition of functions like **`operator+`** so that an expression like **`a + b`** implicitly calls **`a.operator+(b)`** etc.

- We use this feature to allow us to write simple MATLAB-like code for addition, subtraction, and scaling of vectors

- If implemented in the most straightforward way, could be extremely inefficient for large-scale (and parallel) vectors
  - Frequent creation and destruction of temporaries
    - Including parallel layout information
  - Unnecessary copying of memory
  - Excessive load and store operations

“Symbolic Temporaries” Technique

- Suppose **`a`** is a **`double`** and **`w`**, **`x`**, **`y`**, **`z`** are instances of our **`VectorObject`** class
- **`VectorObject`** encapsulates an **`AbstractVector`** pointer
- Consider an assignment statement like

```
w = x + a*y – z;
```

- At run time, this expression builds a relatively compact **`LinearExpression`** object expressing (1,&**`x`**), (a,&**`y`**), (–1,&**`z`**)
- Overloading of the **`=`** operator invokes code to efficiently calculate the result and overwrite **`w`**
- Temporaries are still created, but they are compact and symbolic — their storage is independent of the vector length
- Overhead should be insignificant for large-scale applications (we have confirmed this)

“Symbolic Temporaries” Technique: More Details

`w = x + a*y – z;`

- A **`LinearExpression`** object is a list of pairs of the form (*scalar coefficient*, *pointer to vector*)
- Overload of **`operator*(double&,VectorObject&)`** produces a **`LinearExpression`** of the form (a,&**`y`**)
- Overload of **`VectorObject.operator+(LinearExpression&)`** appends (1,&**`x`**) to the **`LinearExpression`** list
- Overload of **`LinearExpression.operator-(VectorObject&)`** appends (–1,&**`z`**) to the **`LinearExpression`** list
- Overload of **`VectorObject.operator=(LinearExpression&)`** triggers actual calculation with minimum use of temporary memory (deferred evaluation technique)

The `LinearExpression` Class is Enough

First-order optimization methods typically require only a limited range of vector expressions

- Addition, scaling, (subtraction)
- Inner products (just produce a double)
- Projection on simple sets

- Function and gradient calculations:
    - We encapsulate these through a separate **`Problem`** class

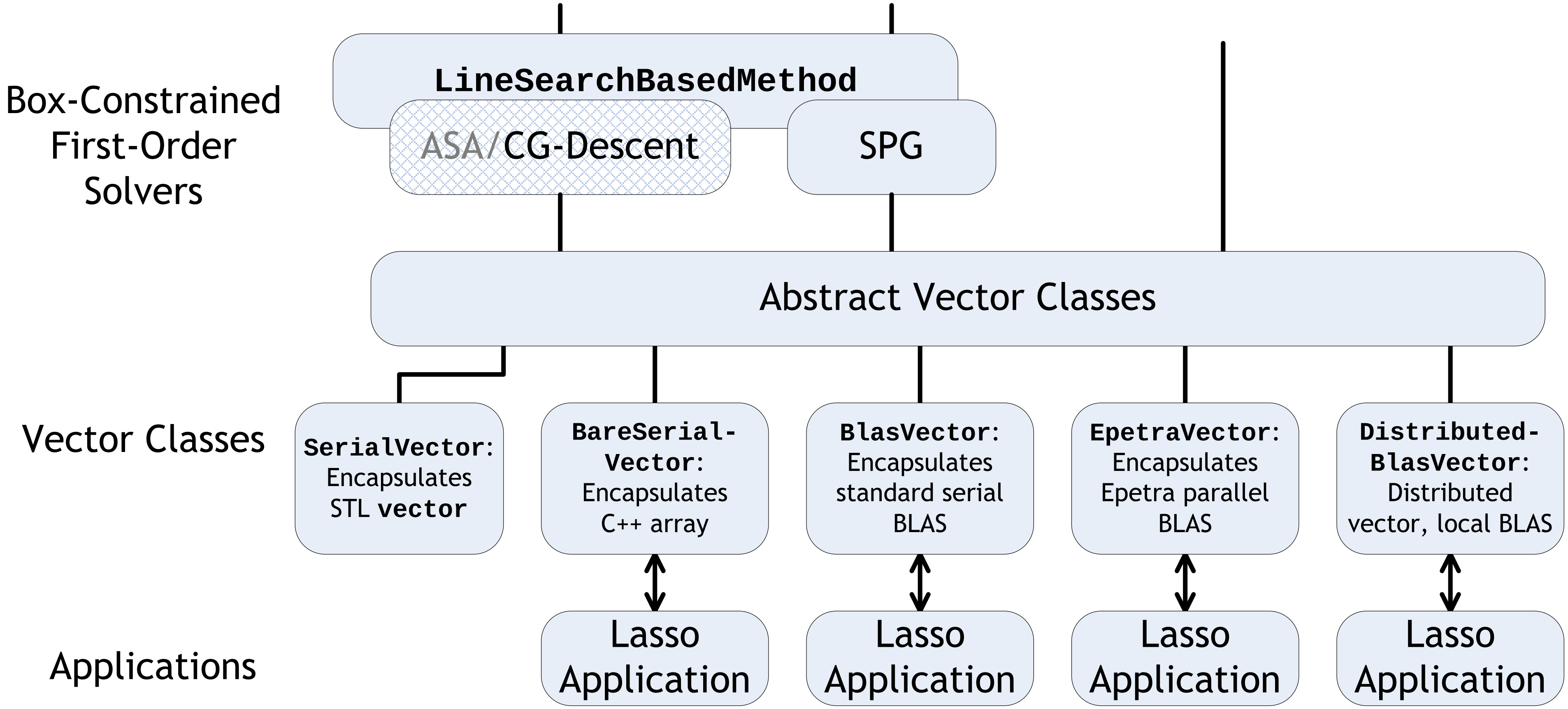
Other operations occur, but are typically on scalars

Support for Line-Search Methods

- We have also built some specialized support for line search
- Automatically cache function values and gradients to avoid both
    - Multiple evaluations at the same point, and…
    - Associated “clutter” in main algorithm code

What We Have Built So Far

- No top-level augmented Lagrangian yet
- Partial implementation of box solvers



Box-Constrained First-Order Solvers

LineSearchBasedMethod

ASA/CG-Descent

SPG

Abstract Vector Classes

Vector Classes

SerialVector: Encapsulates STL **vector**

BareSerial-Vector: Encapsulates C++ array

BlasVector: Encapsulates standard serial BLAS

EpetraVector: Encapsulates Epetra parallel BLAS

Distributed-BlasVector: Distributed vector, local BLAS

Applications

Lasso Application

Lasso Application

Lasso Application

Lasso Application

- Lasso is a convenient (and potentially large-scale) test application for components already built

The Code for SPG Algorithm (with Nonmonotone Line Search)

```
VectorObject& min() {
 while(pGrad.norm_2() > tolerance && iter < maxIter) {
   iter++;
   xPlus = x + lambda*d;
   objValxPlus = Pr->objValGrad(xPlus, gk);
   objValMax = *(max_element(objValArray, objValArray + M));
   if(objValxPlus < objValMax + gamma*lambda*d.inner(g)) {
     descentSteps++;
     objValArray[descentSteps%M] = objValxPlus;
     xk = xPlus;
     b = inner(xk - x, gk - g);
     if(b <= 0) {
       stepSize = stepSizeMax;
     }
     else {
       a = norm_2_sq(xk-x);
       stepSize = smin(stepSizeMax, max(stepSizeMin, a/b));
     }
     x = xk;
     g = gk;
     xk = x - g;
     Pr->projectOnBounds(xk, pxk);
     pGrad = pxk - x;
     xk = x - stepSize*g;
     Pr->projectOnBounds(xk, pxk);
     d = pxk - x;
     lambda = 1;
   }
   else {
     objValx = Pr->objective(x);
     lambda = (-g.inner(d) * lambda*lambda )
       / (2*( objValxPlus - lambda*g.inner(d) - objValx ));
     if (lambda < 0.1 || lambda > 0.9*lambda)
      lambda = lambda/2;
   }
 }
```

Simple Test Application: Lasso

$$\boxed{\min\left\{\tfrac{1}{2}\left\|Ax-b\right\|^2+\nu\left\|x\right\|_1\right\}}$$ (unconstrained, nonsmooth)

Reformulated as

$$\boxed{\begin{array}{ll}\min & \tfrac{1}{2}\left\|A(x-y)-b\right\|^2+\nu\cdot 1^{\mathsf{T}}(x+y)\\ \mathrm{ST} & x,y\geq 0\end{array}}$$

- Becomes a box-constrained problem with a smooth objective
- In this form, does not test augmented Lagrangian level, but allows other software components to be tested
- Rest of talk will be about our experiences with (large-scale) Lasso

Overall Setup

- Fundamentally, using spectral projected gradient (SPG) on

$$\boxed{\begin{array}{ll} \min & \tfrac{1}{2}\left\|A(x-y)-b\right\|^2 + \nu\cdot\mathbf{1}^\mathsf{T}(x+y) \\ \mathrm{ST} & x, y \geq 0 \end{array}}$$

- SPG implemented with our operator overloading technique
- Standard serial algorithm
- Parallelism through underlying linear algebra operations
  - o Partly automatic in SPG / parallel vector classes
  - o Partly application-specific (function and gradient evals)
- Main work per iteration is performing matrix multiplications of the form Ax and $A^\mathsf{T}u$:
  - o Objective gradient is $\nabla h(x,y) = \begin{bmatrix} A^\mathsf{T}\big(A(x-y)-b\big)+\nu\mathbf{1} \\ -A^\mathsf{T}\big(A(x-y)-b\big)+\nu\mathbf{1} \end{bmatrix}$

Data Distribution

- For now, assume A is large but has many more columns than rows ($n \gg m$) – fairly typical in Lasso
- Simple idea, sufficient for dense problems and some sparse problems: partition columns among processors, with matching partition of n-vectors:

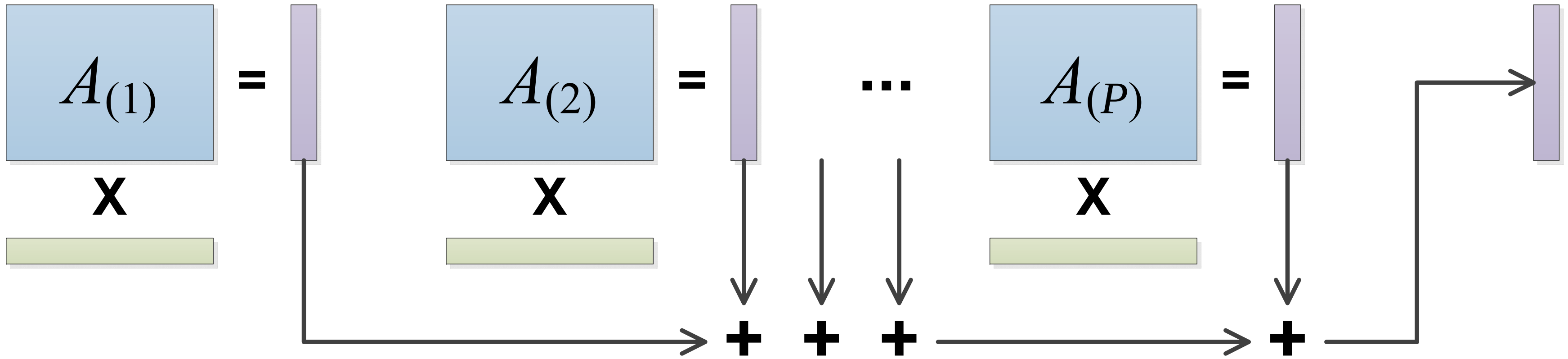
$A_{(1)}$	$\cdots$	$A_{(j)}$	$\cdots$	$A_{(P)}$
$x_{(1)}$ $y_{(1)}$	$\cdots$ $\cdots$	$x_{(j)}$ $y_{(j)}$	$\cdots$ $\cdots$	$x_{(P)}$ $y_{(P)}$

- x and y are segments of a **`DistributedBlasVector`**
- Vectors of length m (like b) are replicated in every processor
  - o Impact small because $n \gg m$

Matrix Multiplications

To multiply $z \in \mathbb{R}^n$ by A: $r = Az$

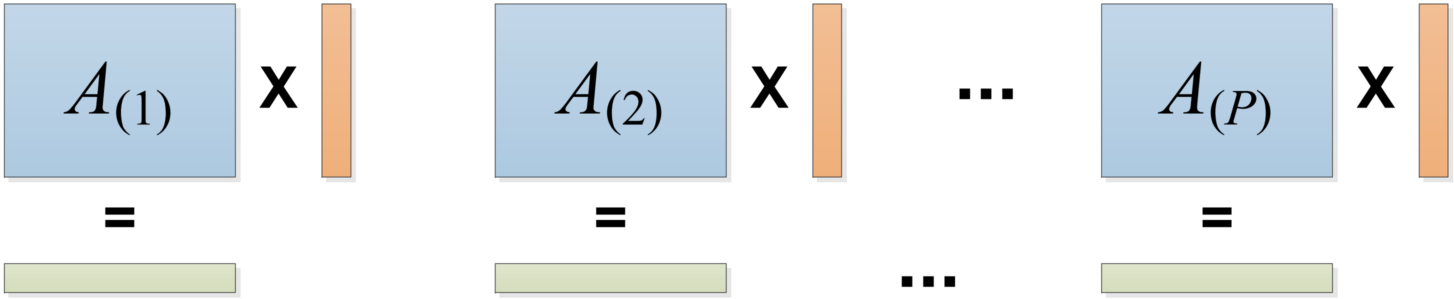
- Each processor performs a local multiplication $r_{(j)} = A_{(j)} z_{(j)}$
  - o For Lasso, first compute $z_{(j)} = x_{(j)} + y_{(j)}$ (in parallel)
- Then do **`Allreduce`** operation (MPI primitive) to compute and broadcast $r = \sum_{j=1}^{P} r_{(j)}$
  - o Requires $\mathrm{O}(\log P)$ rounds of messages, time $\mathrm{O}(m \log P)$



Transpose Matrix Multiplications

To multiply $p \in \mathbb{R}^m$ by A^T: $g = A^\mathsf{T} p$

- Remember, m-vectors are replicated in each processor, so…
- Each processor performs a local multiplication $g_{(j)} = A_{(j)}^\mathsf{T} p$
- Result is g, properly distributed



Balancing Workload

- Key is for each of $A_{(1)},\ldots,A_{(P)}$ to have approximately the same number of nonzero elements
- Then the amount of work in computing $r_{(j)} = A_{(j)}z_{(j)},\ g_{(j)} = A_{(j)}^{\mathsf{T}}p$ will be roughly the same across processors, and the dominant portion of the computation will exhibit near-linear speedup
- Very easy for dense problems: just partition columns as equally as possible and relative imbalance will be $\mathrm{O}(P/n)$
- One approach for sparse problems: heuristically partition the columns in order to balance the number of nonzero elements in each processor

Alternative Approach for Sparse Problems: Trust Epetra

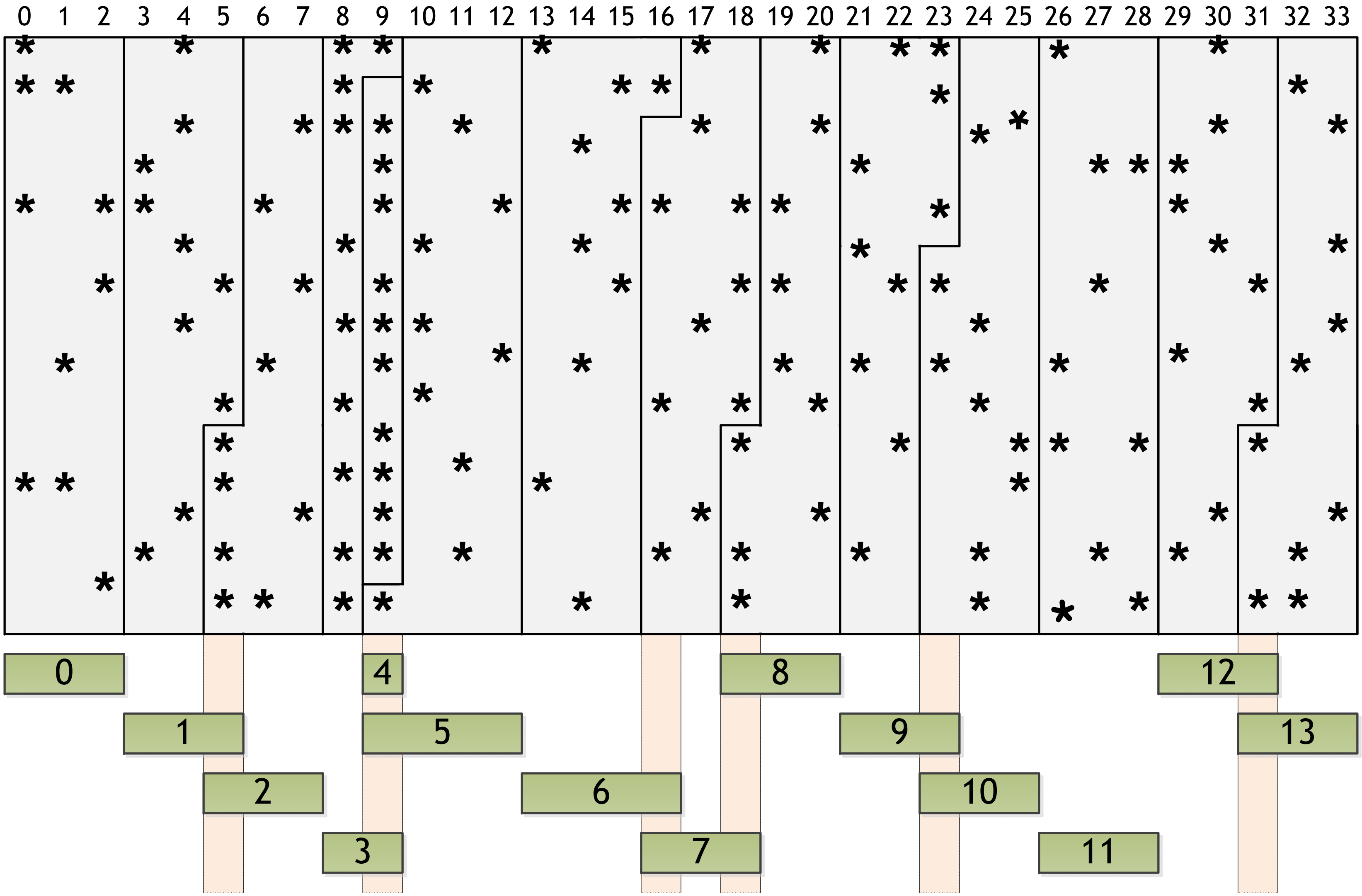
- Represent n-vectors using **`EpetraVector`**
- Partition matrices using Epetra's "Isorropia" sub-package
    - o Can partition by rows **and**/or columns
    - o Could be helpful for matrices that are closer to square
- Use Epetra parallel sparse matrix multiplication

Difficulties with Row/Column Partitioning

- We use matrices derived from real-world datasets
- Derived from UCI repository
- Many are sparse overall, but have a small fraction of relatively dense columns containing a significant fraction of the total nonzeroes
- Rectangular partitions can work poorly such cases

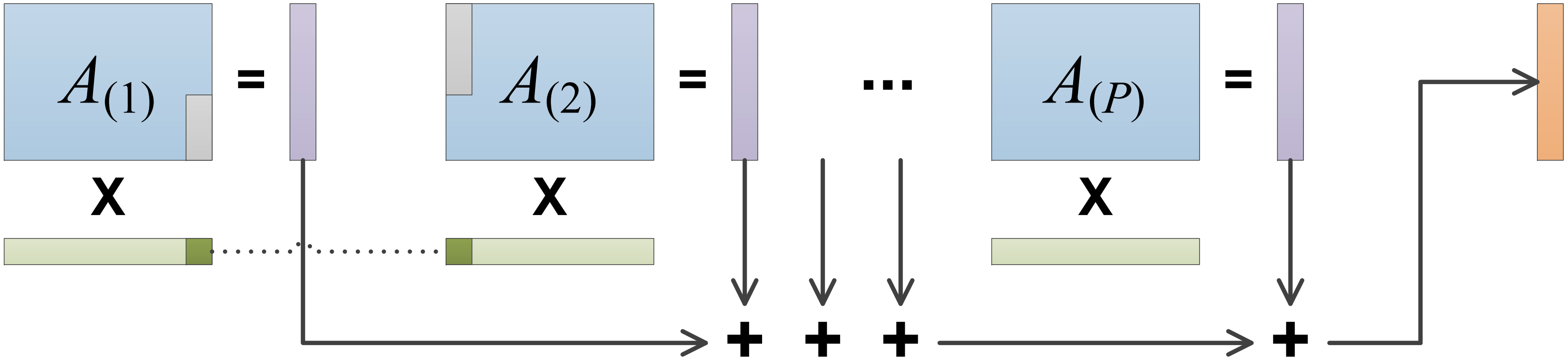
Third Alternative for Sparse Problems

- Sort sparse matrix entries (*row*, *column*, *value*) by *column*
- Partition nonzeros evenly between processors, using this order



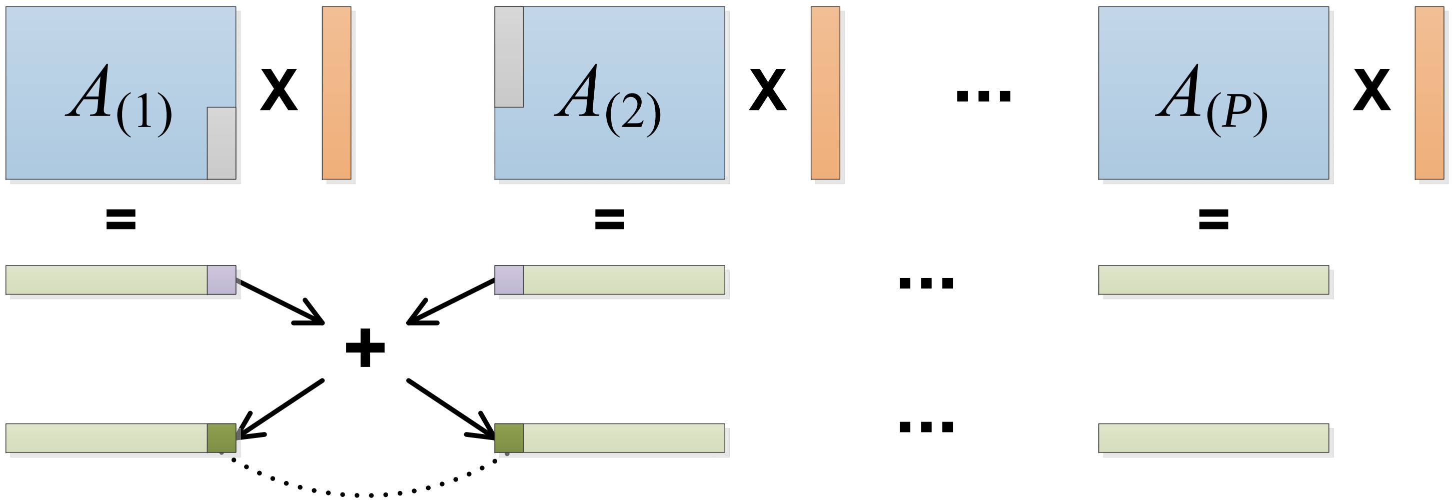
More on the Third Sparse Approach

- For n-vectors, each processor owns all the columns for which it has (*row*, *column*, *value*) triplets
    - o The first and/or last elements might overlap with adjacent processors (with replicated values)
- m-vectors are replicated on each processor as before
- Let $A_{(j)}$ denote the sparse matrix of locally stored triplets, filled out to rectangular shape by zeros
- The multiplication $r = Az$ works exactly the same as before



More on the Third Sparse Approach

- The multiplication $g = A^{\mathsf{T}} p$ consists of the same local multiplication, followed by a scalar sum-reduction and broadcast within each “overlap zone”



Setting Things Up

- For implementation in MPI, we need to set up a “communicator” for each overlap zone
  - Once this is accomplished, implementation is easy
  - Reduction takes $\mathrm{O}(\log V)$ steps, where $V \le P$ is the maximum number of processors in an overlap zone
- Tricky to get communicators set up without introducing an $\Theta(P)$ operation, but can be done in $\mathrm{O}(\log P)$

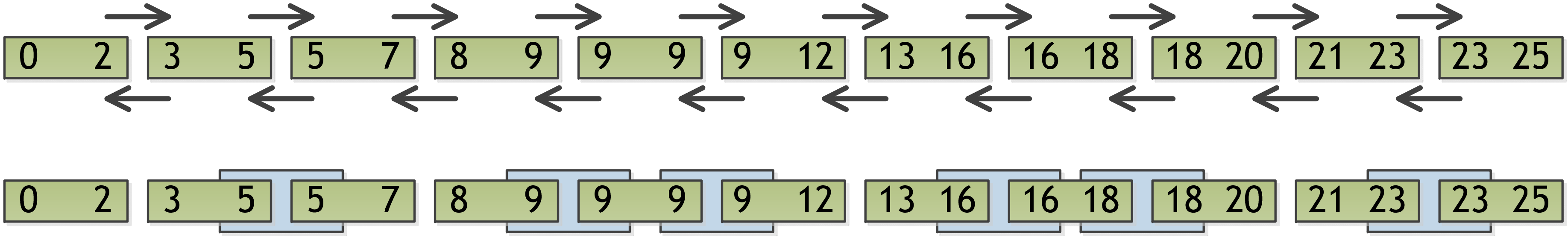
Digression: Parallel Scans

$$\begin{array}{|c|c|c|c|c|c} \hline 1.0 & 2.3 & 4.0 & 2.1 & 0.1 & \cdots \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|c|c|c} \hline 1.0 & 3.3 & 7.3 & 9.4 & 9.5 & \cdots \\ \hline \end{array}$$

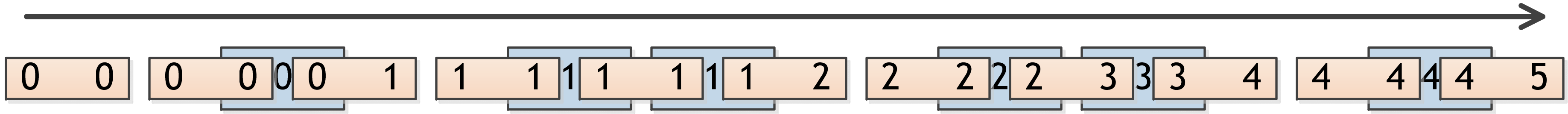
- For any associative operator, this operation can be performed in $\mathrm{O}(\log P)$ communication, $\mathrm{O}((n/P)\log P)$ time
- Can design the operator to restart accumulation “from scratch” at certain points (implementing “segments”)

Digression: Details of Setting up Communicators

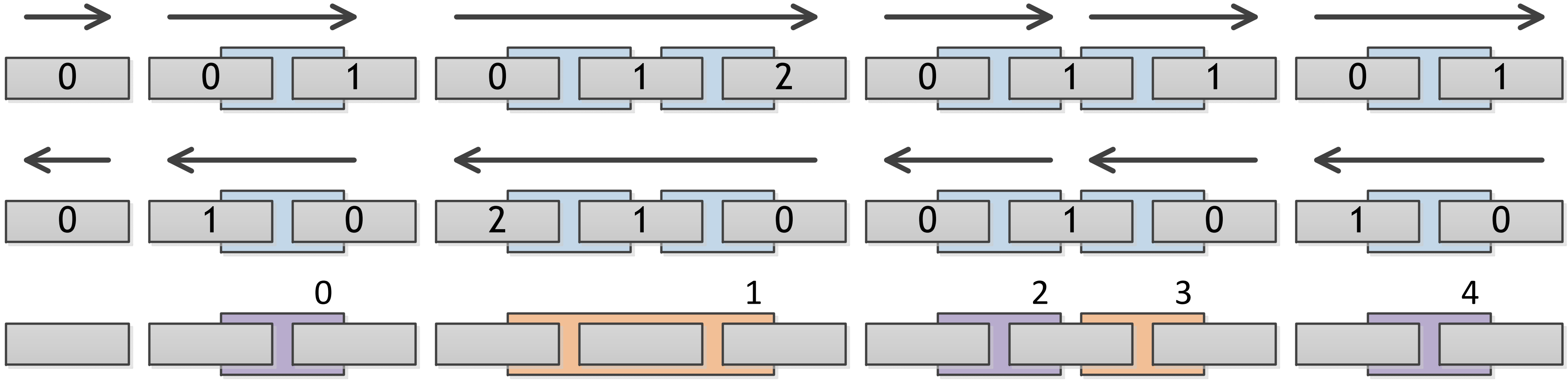
1. One message left, one message right to identify neighbors



2. Sum-scan to assign a unique number to each overlap group

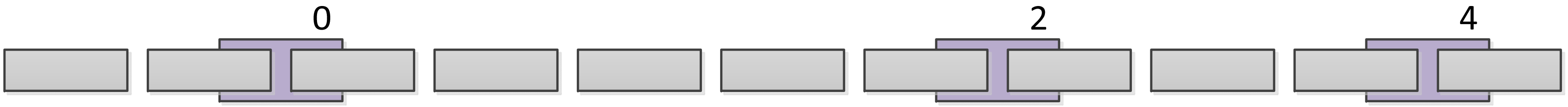


3. Scan forward and back with a customized operator so each processor knows full extent of left and right group

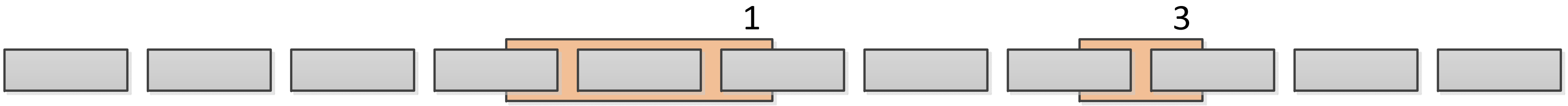


Continued Digression: Detail of Setting Up Communicators

4. Call **`MPI_Comm_create`** to make all even-numbered groups



5. Call **`MPI_Comm_create`** to make all odd-numbered groups



- Each call subdivides an existing communicator
- Each processor in at most one communicator
- All processors call **`MPI_Comm_create`** synchronously
- Those with nothing to do create and destroy a trivial communicator

This procedure avoids using tempting routines like **`MPI_Comm_split`**, which perform gather / scatter operations that can introduce $O(P)$ communication, hurting scalability

Computational Experiments

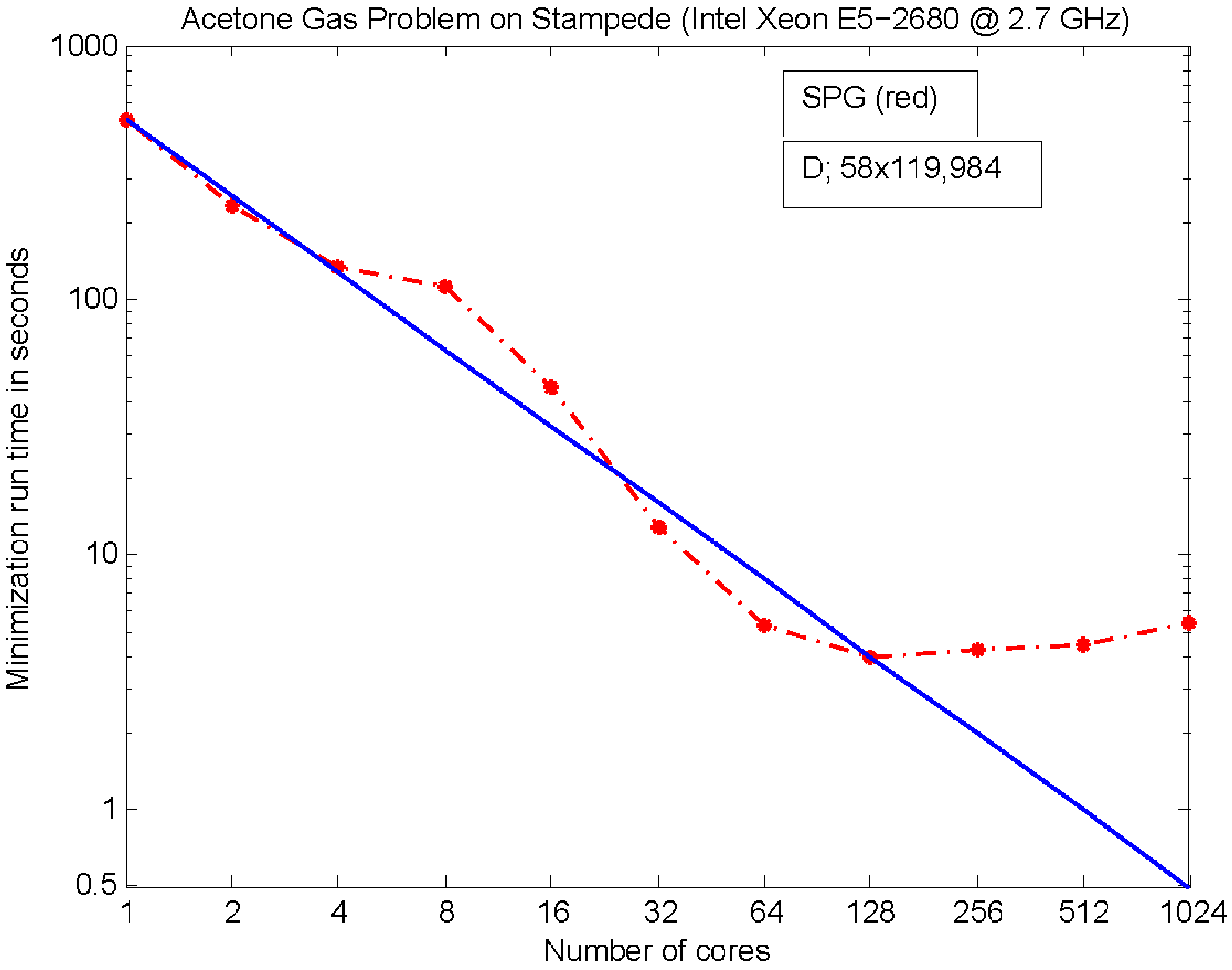
- SPG is a numerically sensitive algorithm
  - o Minor changes (even version of BLAS) can cause some variation (not gigantic) in the number of iterations

- Convergence condition: $\left\| \left[(x,y) - \nabla h(x,y) \right]_+ - (x,y) \right\|_\infty \le 10^{-6}$

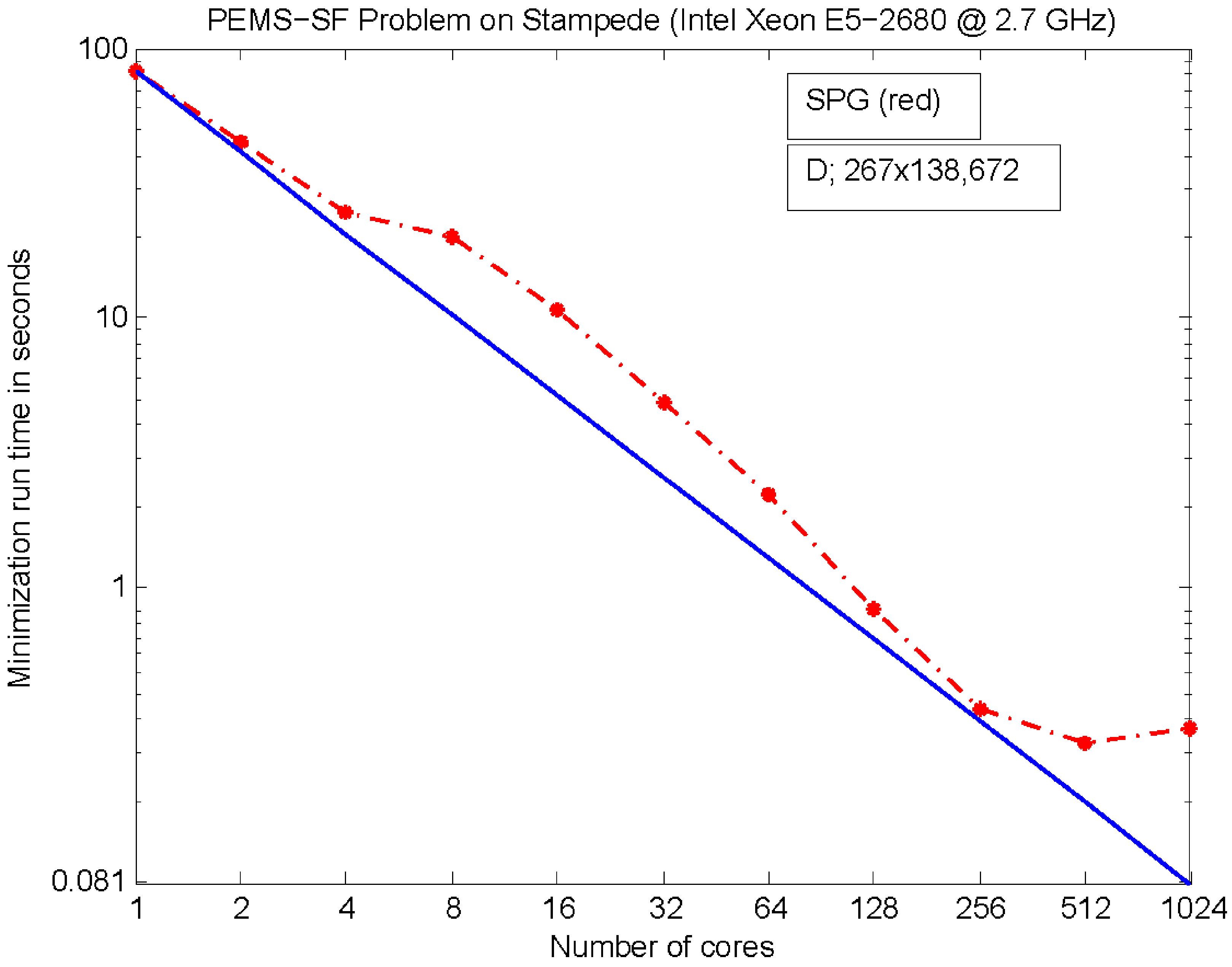
- Some data adapted from UCI machine learning Repository
- Other data randomly generated

- Computer system: TACC Stampede
  - o Xeon E5 cores, 2.7 GHz clock
  - o Xeon Phi accelerators, but not used by our code
  - o Infiniband fat-tree interconnect, MPI

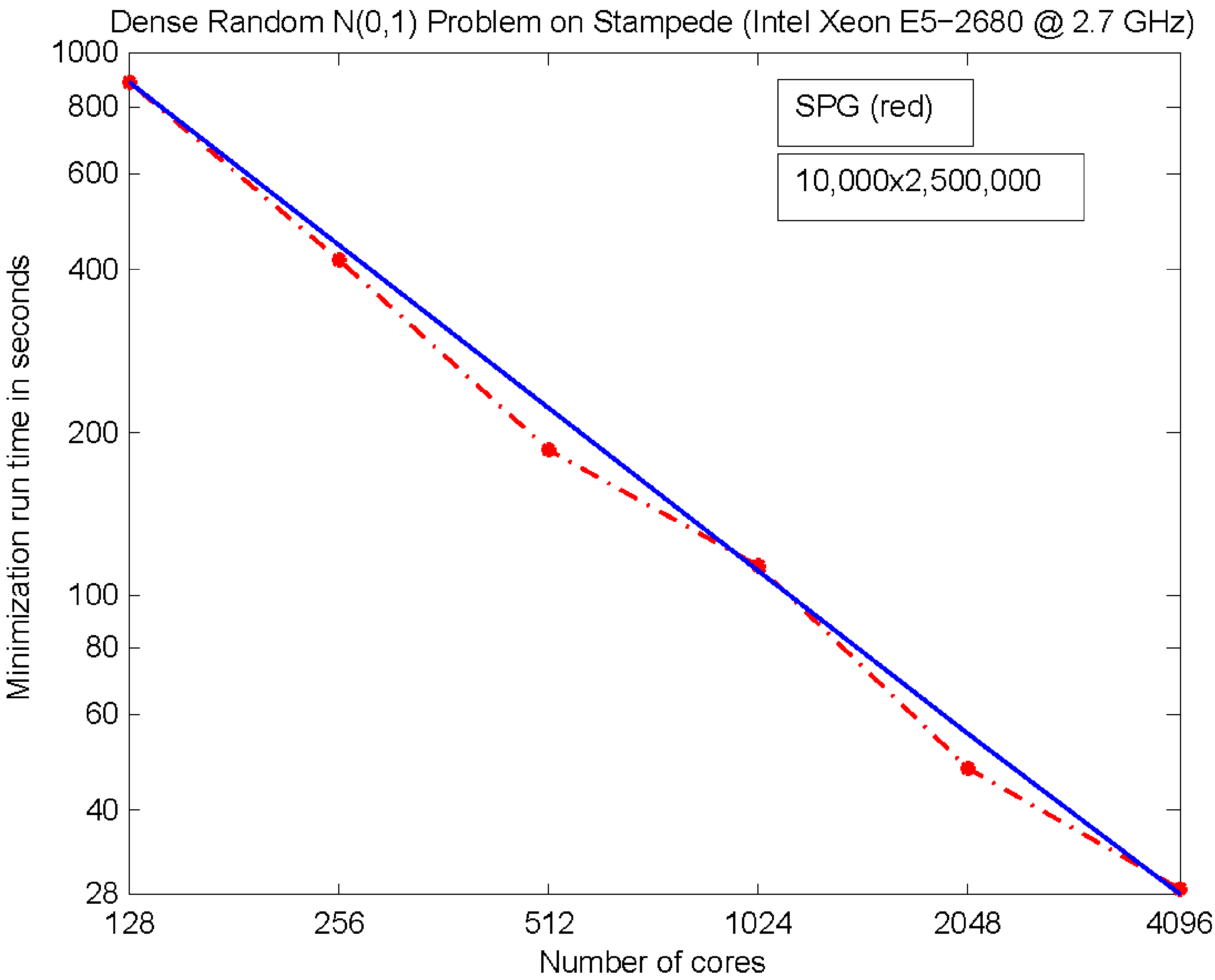
Dense: Detecting Acetone in Gas Samples (from UCI Repository)



More Dense: Highway Lane Occupancy (UCI Repository)



Dense: Randomly Generated Data



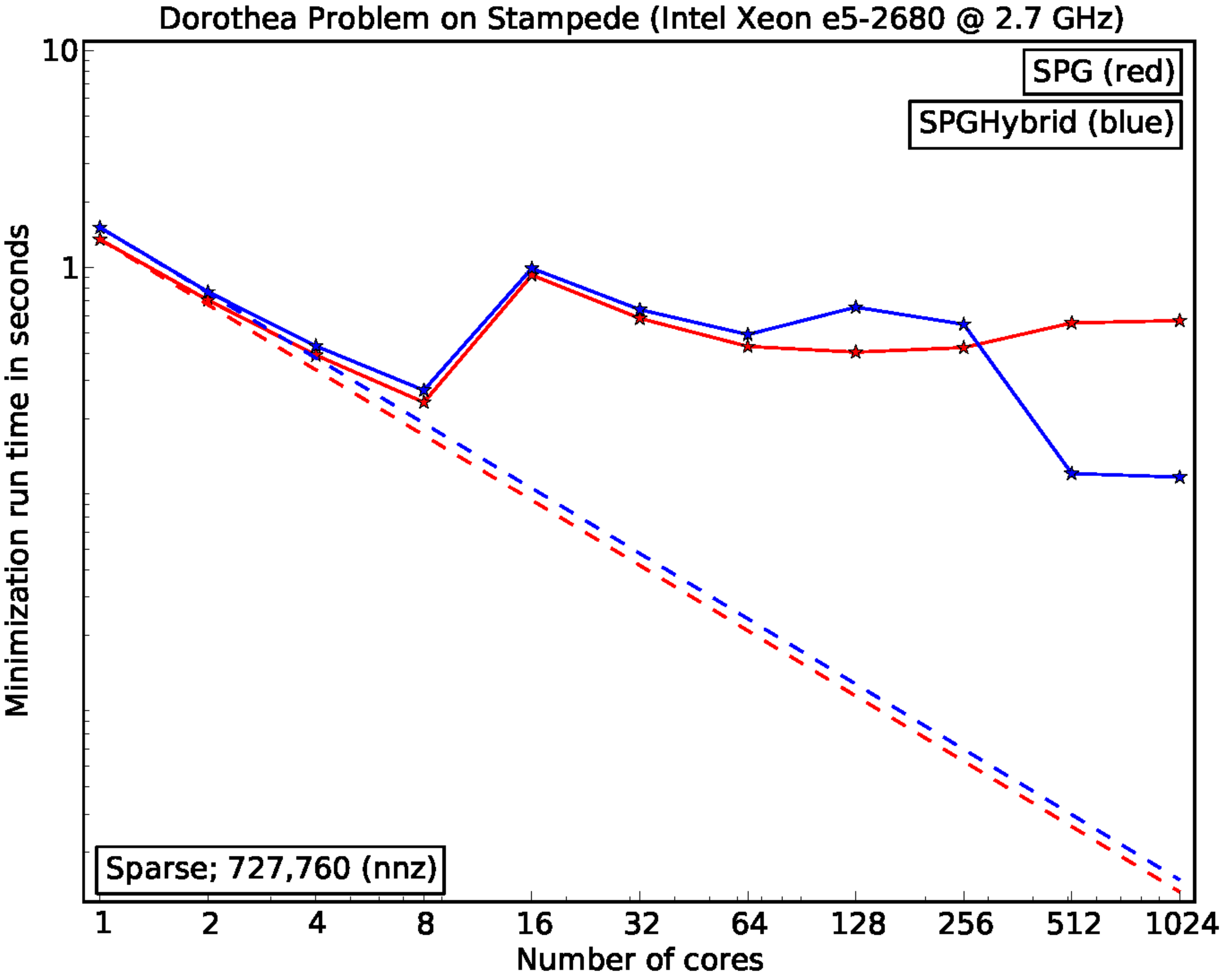
Sparse Problems

- “SPG” : our method with balanced nonzero partition
- “SPGHybrid” : our method with heuristic column partition
- “SPGEpetra” : our method with Epetra partitioning of matrix and vectors
- “ACDC” : Richtárik / Takáč random parallel coordinate descent
    - With small setting of proximal parameter – far below threshold required by (probabilistic) convergence theory
    - Does not have its own termination test; we terminate when it is close to the optimal value computed by SPG

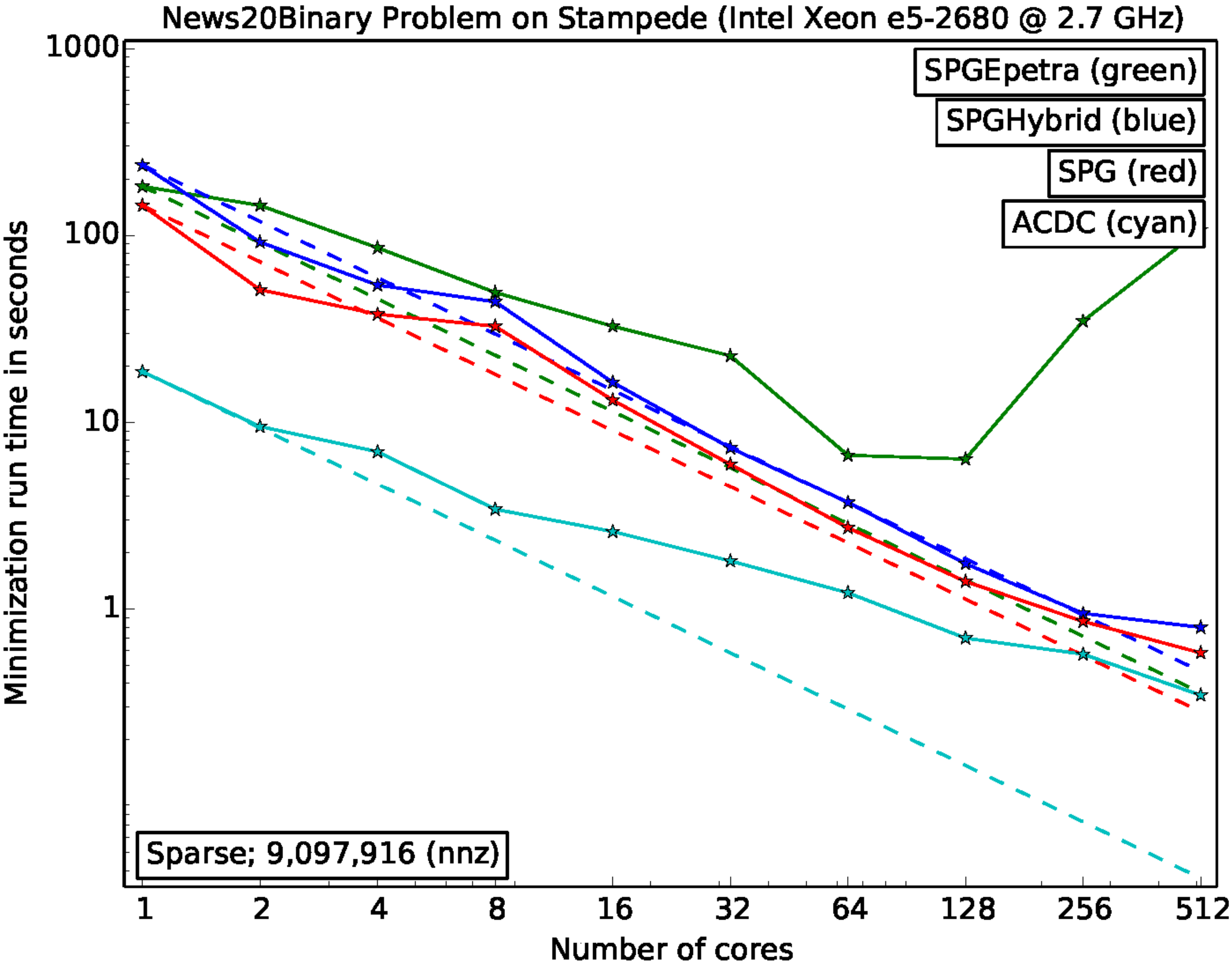
Current datasets: (working on getting more)

- Drug discovery: 1,950 × 100,000, 728K nonzeroes
- News articles: 19,996 × 1,355,191, 9.1M nonzeroes
- Randomly generated: 10,000 × 2,500,000, 250M nonzeroes

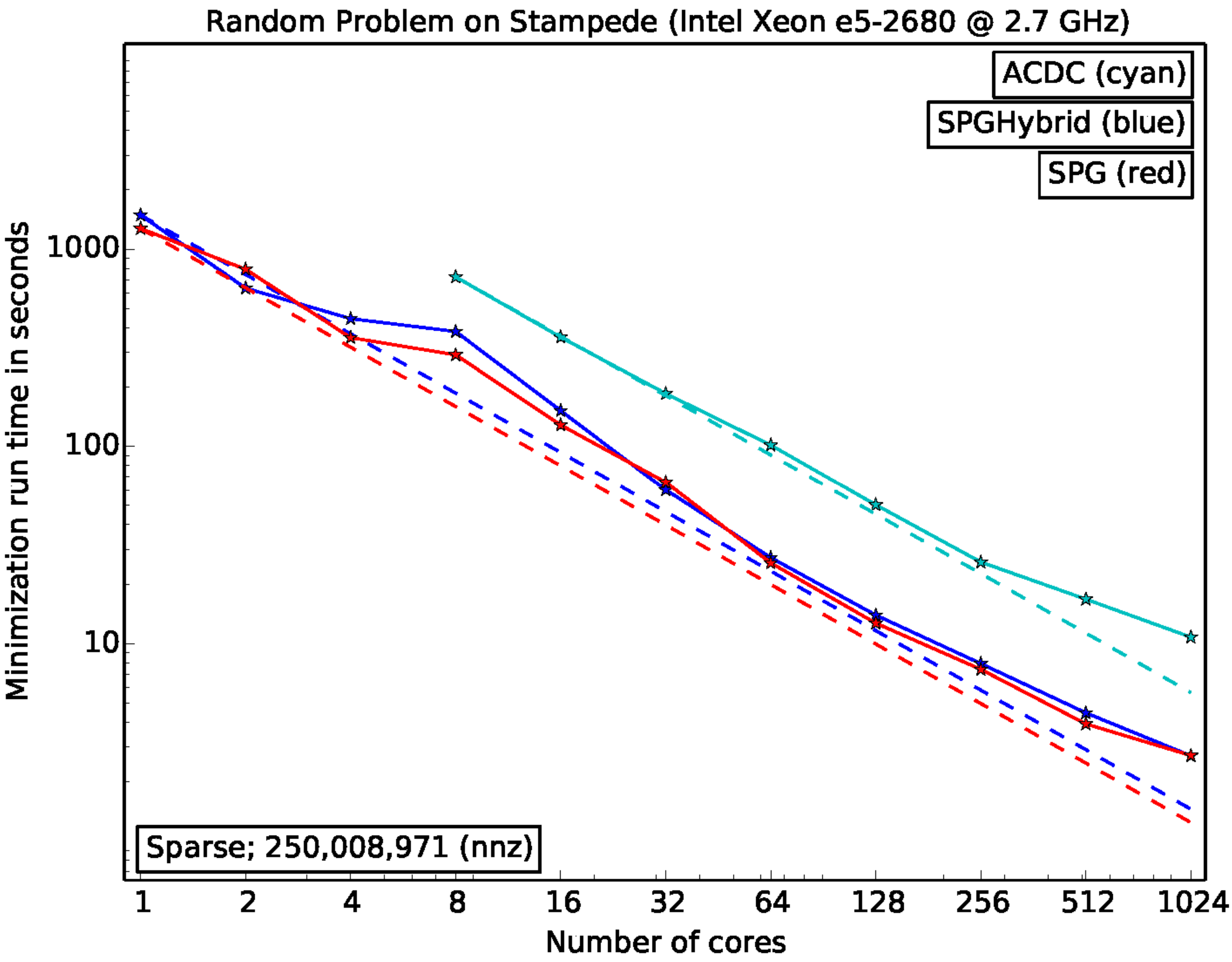
Sparse: Drug Discovery (Small)



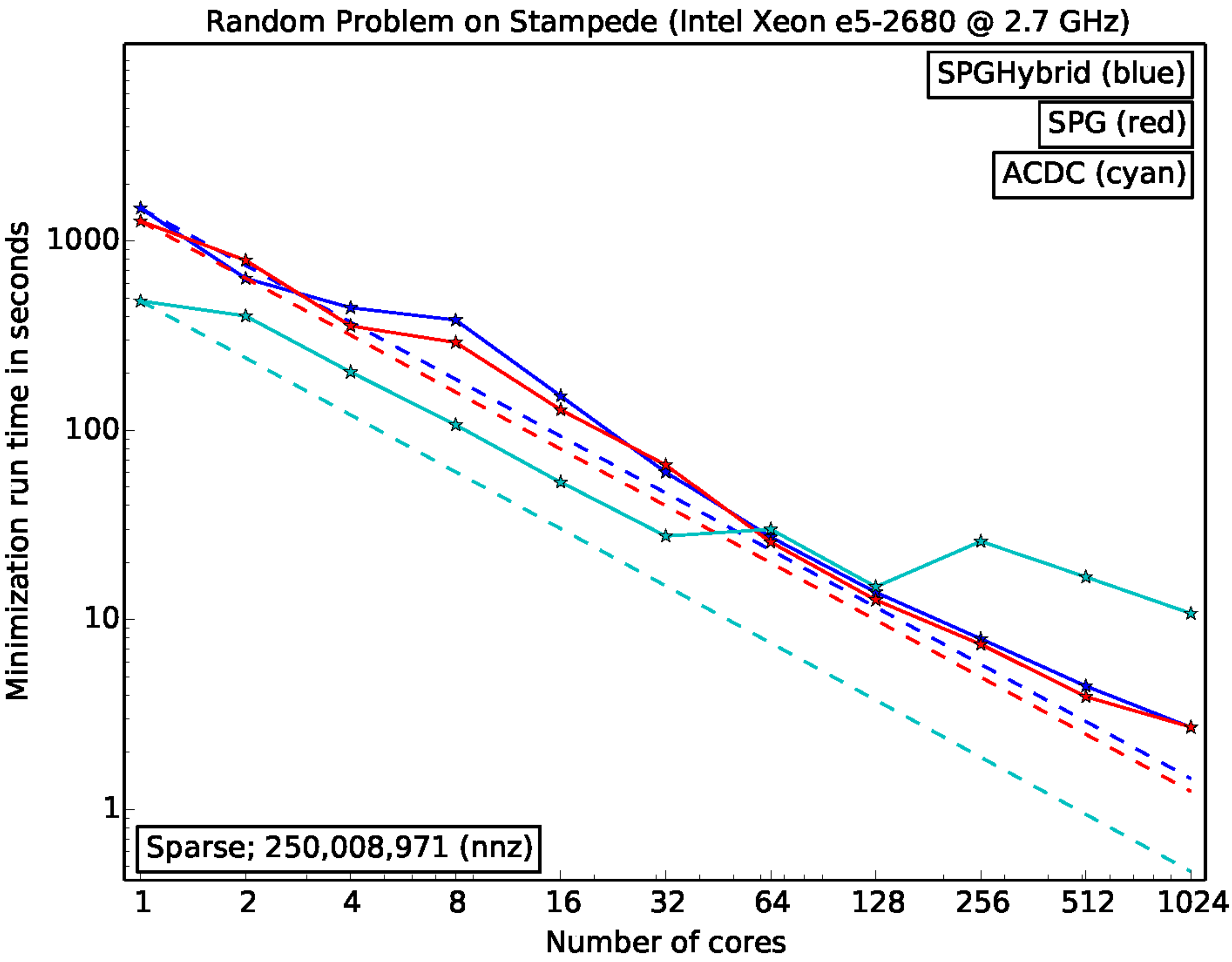
Sparse: News Articles



Sparse: Randomly Generated (Fairly Large)



Sparse: Randomly Generated (Varying Parameters for ACDC)



Discussion of Experimental Results

- Dense code scales well

- Balanced nonzero distribution seems to work best and scales quite well

- But only beats heuristic column balancing by a small amount (will this remain true for larger real datasets?)
  - However, it also has lower setup time (not shown here), so it's preferable to heuristic column balancing

- Epetra does not scale well on these datasets ☹

- Not dominated by ACDC in a parallel setting
  - ACDC specifically designed for Lasso-like problems
  - Our fundamental method is generic…
  - … yet reasonably competitive in parallel

The “Object-Parallel” Approach…

… allowed application development to focus on efficiency of objective/gradient evaluations

- But not more complicated things like structure of matrix factors
- We could organize the data to optimize efficiency of the most compute-intensive operations
- We tried four different data representations and two different data layouts for x and y …
- … but the SPG code remained identical
- Parallelism applied to underlying operations of serial method: iteration count largely independent of number of processors.

Same principles could apply to problems with general constraints, through approximate augmented Lagrangian method

- But also need to consider application constraint structure

The “Object-Parallel” Approach: Summary

Approach to parallel optimization by combining several well-established principles:

- Efficient first-order methods (continuing to develop)
- Object-oriented programming

- And eventually, augmented Lagrangians (for problems with general constraints)
    - … with new “loose” approximation criteria