Parallel Combinatorial Computing and Sparse Matrices

E. Jason Riedy James Demmel

Computer Science Division
EECS Department
University of California, Berkeley

SIAM Conference on Computational Science and Engineering, 2005

Outline

Fundamental question: What performance metrics are right?

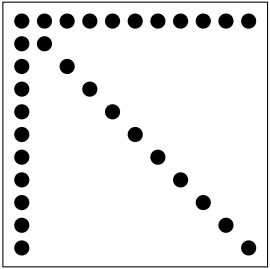
Background

Algorithms
  - Sparse Transpose
  - Weighted Bipartite Matching

Setting Performance Goals

Ordering Ideas

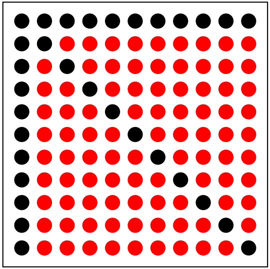
App: Sparse *LU* Factorization



Characteristics:

- Large quantities of numerical work.
- Eats memory and flops.
- Benefits from parallel work.
- And needs combinatorial support.

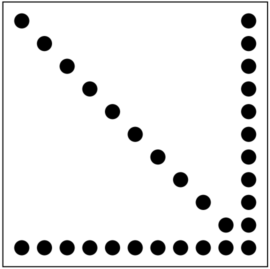
App: Sparse *LU* Factorization



Characteristics:

- Large quantities of numerical work.
- Eats memory and flops.
- Benefits from parallel work.
- And needs combinatorial support.

App: Sparse *LU* Factorization



Combinatorial support:

- Fill-reducing orderings, pivot avoidance, data structures.
- Numerical work is distributed.
- Supporting algorithms need to be distributed.
- Memory may be cheap ($100 GB), moving data is costly.

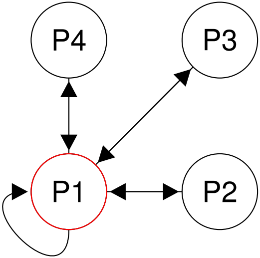
Sparse Transpose

Data structure manipulation

- Dense transpose moves numbers, sparse moves numbers and re-indexes them.
- Sequentially space-efficient "algorithms" exist, but disagree with most processors.
  - Chains of data-dependent loads
  - Unpredictable memory patterns

If the data is already distributed, an unoptimized parallel transpose is better than an optimized sequential one!

Parallel Sparse Transpose



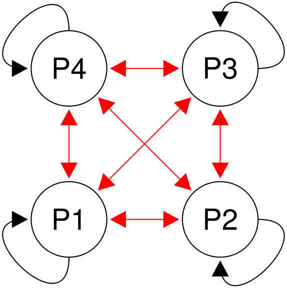
Many, many options:

- Send to root, transpose, distribute.
- Transpose, send pieces to destinations.
- Transpose, then rotate data.
- Replicate the matrix, transpose everywhere.

Communicates most of matrix twice. Node stores whole matrix.

Note: We should compare with this implementation, not purely sequential.

Parallel Sparse Transpose

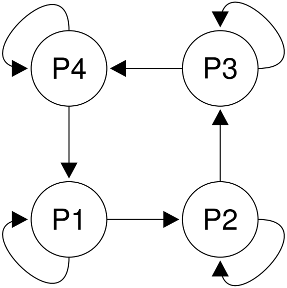


Many, many options:

- Send to root, transpose, distribute.
- Transpose, send pieces to destinations.
- Transpose, then rotate data.
- Replicate the matrix, transpose everywhere.

All-to-all communication. Some parallel work.

Parallel Sparse Transpose

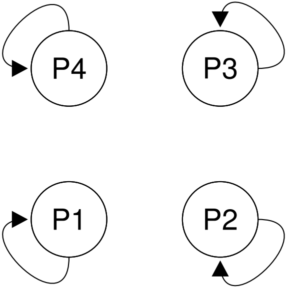


Many, many options:

- Send to root, transpose, distribute.
- Transpose, send pieces to destinations.
- Transpose, then rotate data.
- Replicate the matrix, transpose everywhere.

Serial communication, but may hide latencies.

Parallel Sparse Transpose



Many, many options:

- Send to root, transpose, distribute.
- Transpose, send pieces to destinations.
- Transpose, then rotate data.
- Replicate the matrix, transpose everywhere.

Useful in some circumstances.

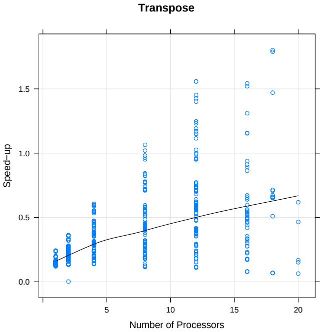
What Data Is Interesting?

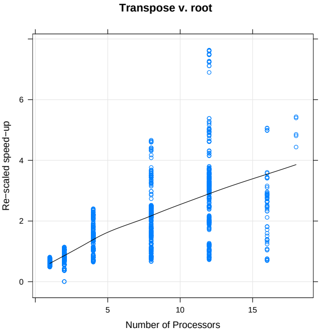
- Time

What Data Is Interesting?

- Time (to solution)
- How much data is communicated.
- Overhead and latency.
- Quantity of data resident on a processor.

Parallel Transpose Performance

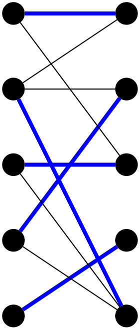




- All-to-all is slower than pure sequential code, but distributed.
- Actual speed-up when the data is already distributed.
- Hard to keep constant size / node when performance varies by problem.

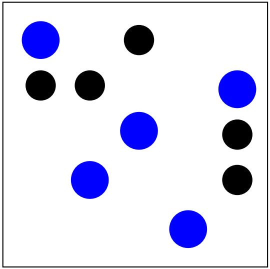
Data from CITRIS cluster, Itanium2s with Myrinet.

Weighted Bipartite Matching



- Not moving data around but finding where it should go.
- Find the “best” edges in a bipartite graph.
- Corresponds to picking the “best” diagonal.
- Used for static pivoting in factorization.
- Also in travelling salesman problems, *etc*.

Weighted Bipartite Matching



- Not moving data around but finding where it should go.
- Find the “best” edges in a bipartite graph.
- Corresponds to picking the “best” diagonal.
- Used for static pivoting in factorization.
- Also in travelling salesman problems, *etc*.

Algorithms

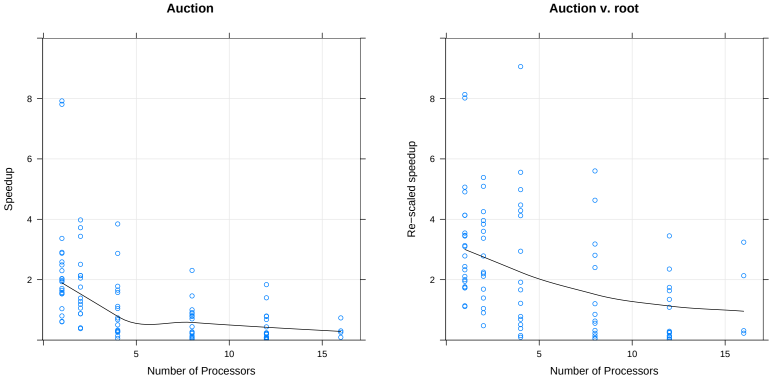
Depth-first search

- Reliable performance, code available (MC64)
- Requires A and A^T.
- Difficult to compute on distributed data.

Interior point

- Performance varies wildly; many tuning parameters.
- Full generality: Solve larger sparse system.
- Auction algorithms replace solve with iterative bidding.
- Easy to distribute.

Parallel Auction Performance



Compare with running an auction on the root, a parallel auction achieves slight speed-up.

Proposed Performance Goals

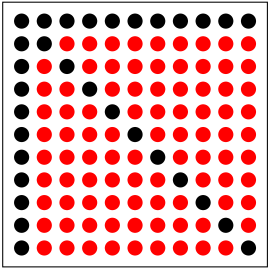
When is a distributed combinatorial algorithm (or code) successful?

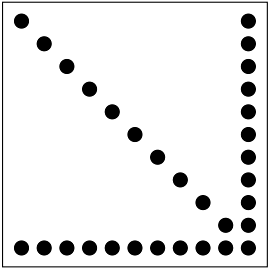
- Does not redistribute the data excessively.
- Keeps the data distributed.
- No process sees more than the whole.
- Performance is competitive with the on-root option.

Pure speed-up is a great goal, but not always reasonable.

Distributed Matrix Ordering

Finding a permutation of columns and rows to reduce fill.





NP-hard to solve, difficult to approximate.

Sequential Orderings

Bottom-up

- Pick columns (and possibly rows) in sequence.
- Heuristic choices:
  - Minimum degree, deficiency, approximations
- Maintain symbolic factorization

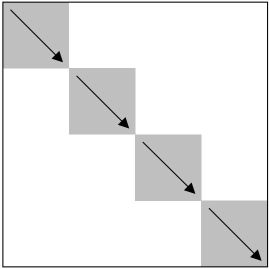
Top-down

- Separate the matrix into multiple sections.
  - Graph partitioning: $A + A^T$, $A^T \cdot A$, A
- Needs vertex separators: Difficult.

Top-down Hybrid

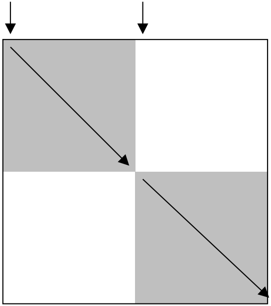
- Dissect until small, then order.

Parallel Bottom-up Hybrid Order



1. Separate the graph into chunks.
    - Needs an edge separator,
    - and knowledge of the pivot.
2. Order each chunk separately.
    - Forms local partial orders.
3. Merge the orders.
    - What needs communicated?

Merging Partial Orders



Respecting partial orders

- Local, symbolic factorization done once.
- Only need to communicate quotient graph.
  - Quotient graph: Implicit edges for Schur complements.
- No node will communicate more than the whole matrix.

Preliminary Quality Results

Merging heuristic

- Pairwise merge.
- Pick the head pivot with least worst-case fill (Markowitz cost).

Small (tiny) matrices: performance not reliable.

Matrix	Method	NNZ increase
west2021	AMD $(A + A^T)$	1.51×
	merging	1.68×
orani678	AMD	2.37×
	merging	6.11×

Increasing the numerical work drastically spends any savings from computing a distributed order. Need better heuristics?

Summary

- Meeting classical expectations of scaling is difficult.
  - Relatively small amounts of computation for much communication.
  - Problem-dependent performance makes equi-size scaling hard.
- But consolidation costs when data is already distributed.

In a supporting role, don't sweat the speed-up.
Keep the problem distributed.

Open topics

- Any new ideas for parallel ordering?