



Parallel Greedy Matching Algorithms

Fredrik Manne

Department of Informatics
University of Bergen, Norway

Rob Bisseling, University of Utrecht
Md. Mostofa Patwary, University of Bergen

Outline

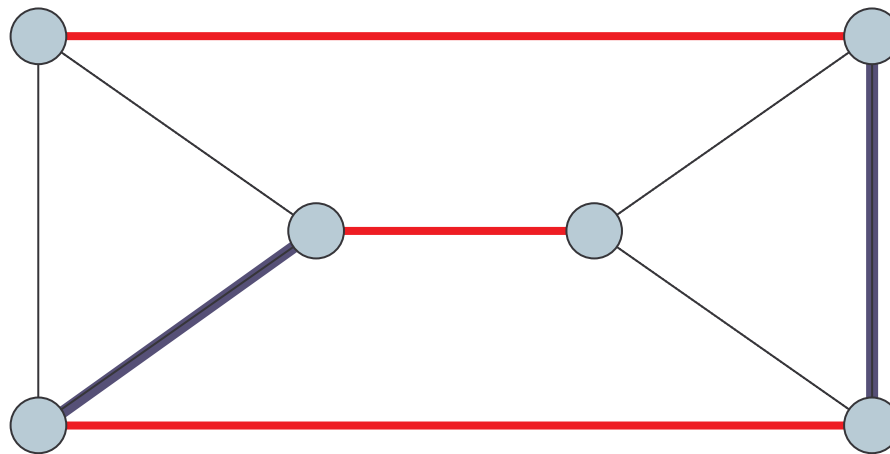
- Background on the matching problem
- Parallel matrix – vector multiplication
- A new parallel greedy matching algorithm
- Experiments
- Conclusion

Combinatorial Scientific Computing

- Study of discrete algorithms in scientific and engineering applications (as opposed to continuous mathematics).
- Graph and geometric algorithms are fundamental tools.
- Has emerged as a separate field within scientific computing.
- Significantly affects performance of scientific computations

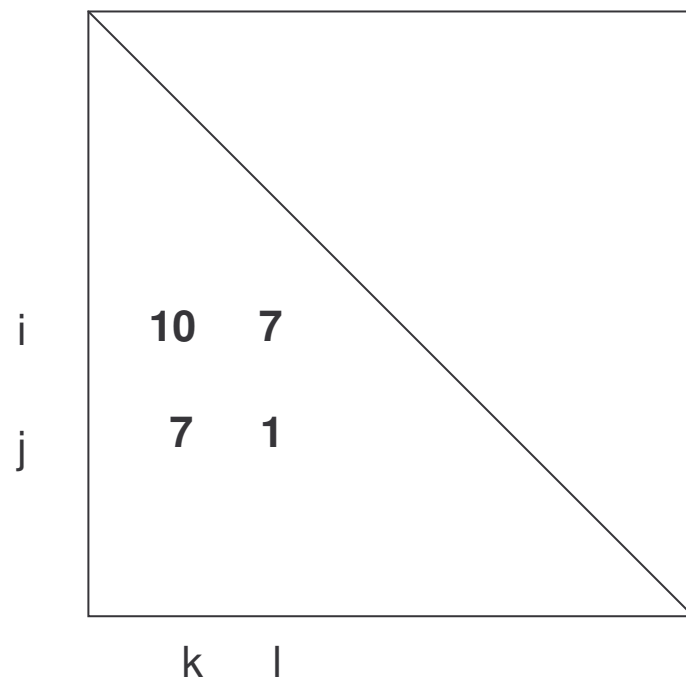
The Matching Problem

- Given graph $G(V,E)$
- Select set of independent edges such that either
 - S has maximum cardinality or
 - Weight of S is maximum



Application in CSC: Weighted Bipartite Matching

- Given n by n matrix A
 Use pivoting to increase weight of diagonal elements
- Increases numerical stability for direct solvers
 - Increases convergence rate for iterative solvers



Greedy solution: **Optimal:**

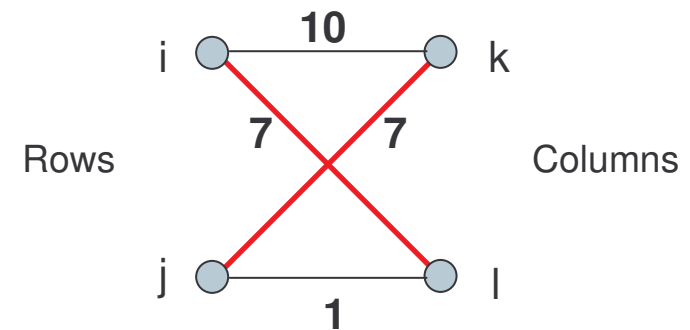
i	10	7
j	7	1
	k	l

Weight = 11

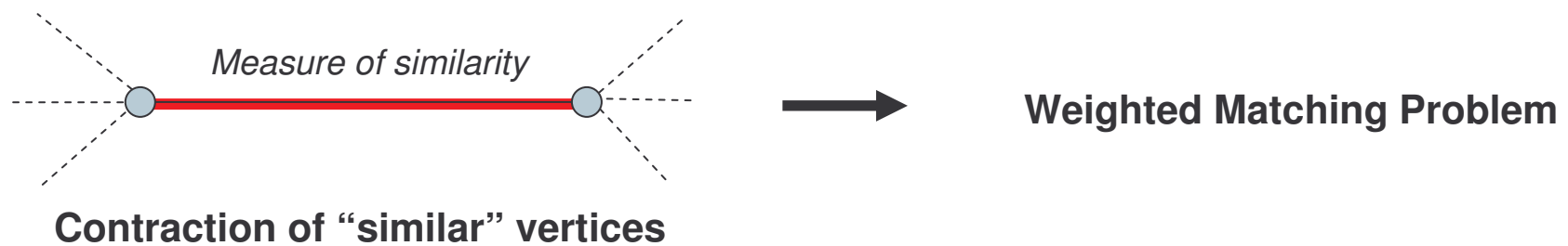
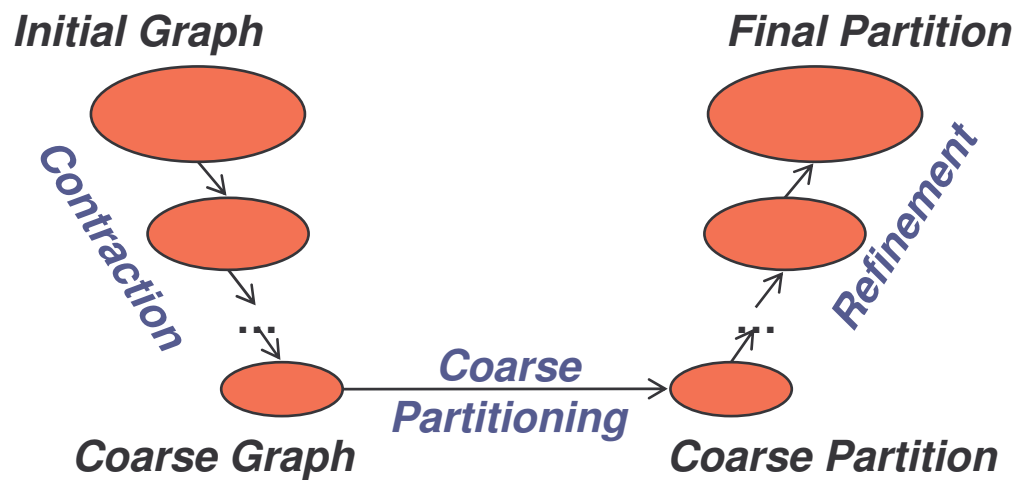
i	7	10
j	1	7
	k	l

Weight = 14

As a matching problem:



Application in CSC: Graph Partitioning





The Greedy Approach

While there are edges left

Add “best” remaining edge (v,w) to S

Remove (v,w) and all edges incident on v and w .

Several variations depending on how “best” is defined.

Examples:

Unweighted case: Best = edge with lowest degree endpoint

Weighted case: Best = edge with maximum weight.

$$W(S) \geq \frac{1}{2}W(\text{opt})$$

Parallelization: Difficult since each decision depends on previous choices

The Karp – Sipser algorithm

While there are edges left in G

If there exists a vertex v of degree one

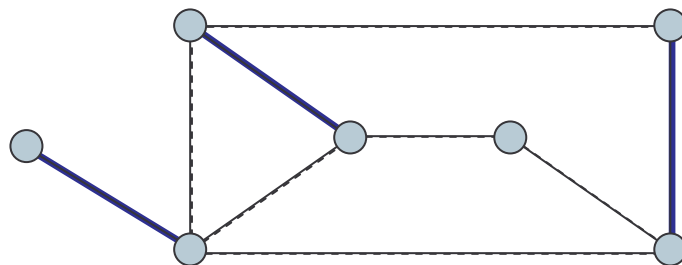
$e = (v, \text{neighbor of } v)$

else

$e = \text{random remaining edge}$

$S = S \cup \{e\}$

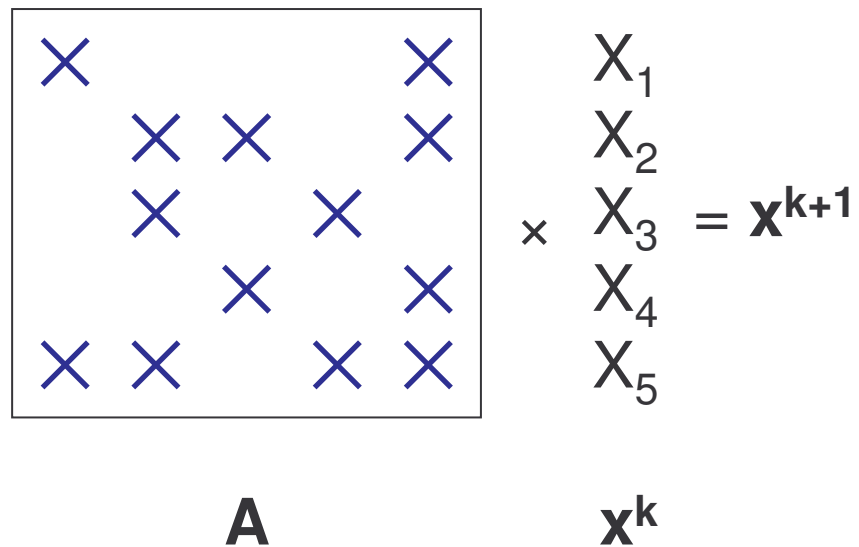
Remove e and all edges incident on e from G



Observation: Works well in practice

Sparse Matrix-vector multiplication

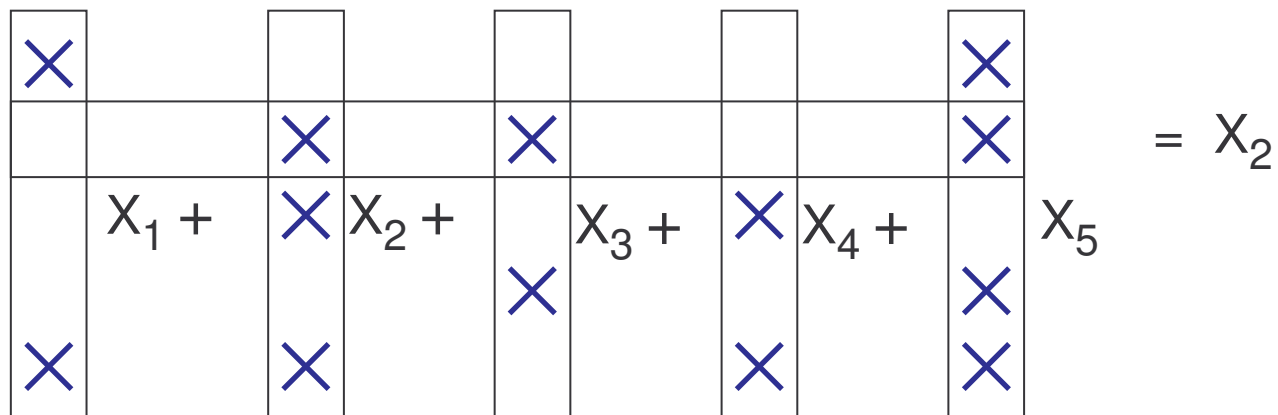
- Repeatedly compute $\mathbf{x}^{k+1} = \mathbf{A}\mathbf{x}^k$
- \mathbf{A} is large, sparse and symmetric


$$\begin{array}{ccccc} \times & & & & \times \\ & \times & \times & & \times \\ & \times & & \times & \\ & & \times & & \times \\ \times & \times & & \times & \times \end{array} \times \begin{array}{c} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{array} = \mathbf{x}^{k+1}$$

\mathbf{A} \mathbf{x}^k

Parallel Sparse Matrix-vector multiplication

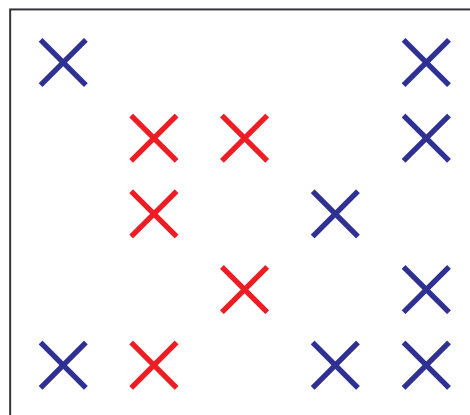
Each x_i is initially stored on some processor $P(i)$



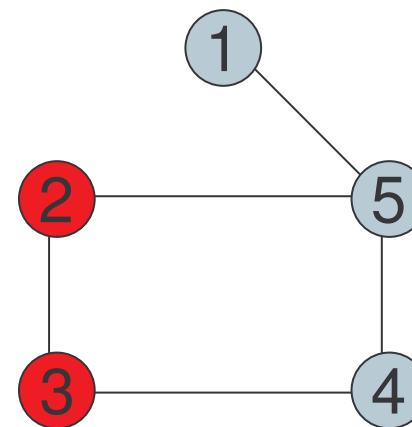
- x_i must be sent to wherever there is an element from column A_i
- Every element in row j of A must be reduced to processor $P(j)$

Data distribution

- Objectives
 - Split every row and column of A on as few processors as possible
 - Assign an equal number of non-zeros to each processor.
- Can be done using graph partitioning on the adjacency graph of A :

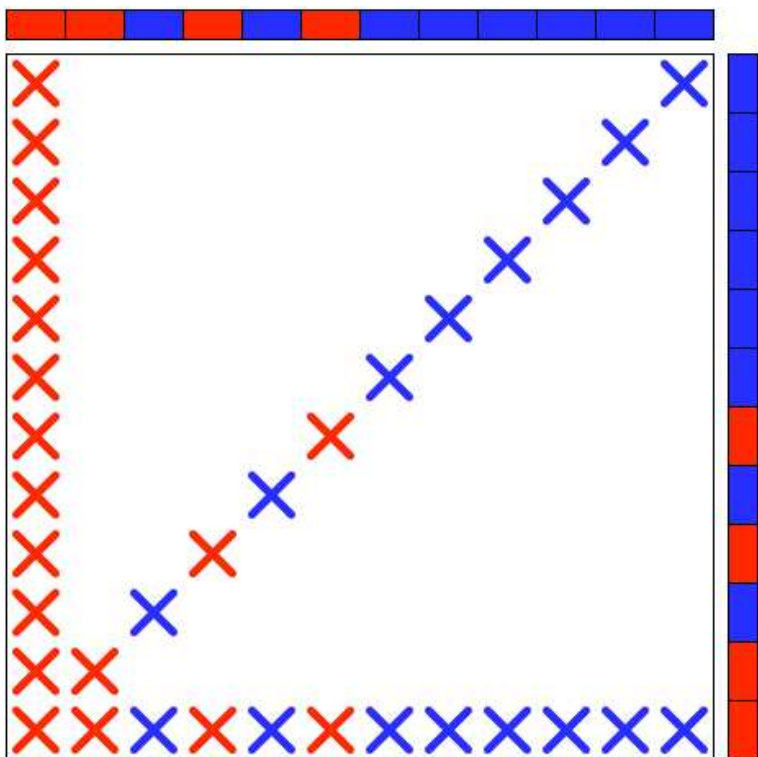


A



$G(A)$

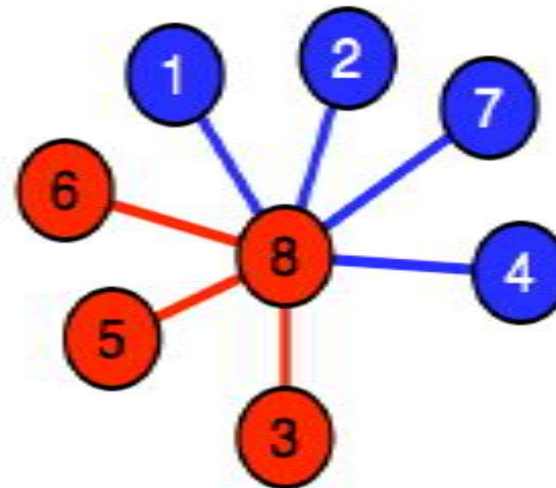
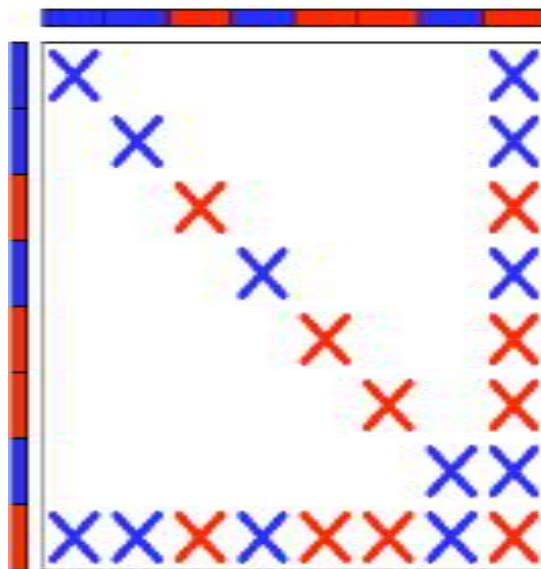
But things can go bad..



- For any 1-D bisection of $n \times n$ arrowhead matrix:
 - $\text{nnz} = 3n - 2$
 - Volume $\approx (3/4)n$

2d Hypergraph based partitioning

Any non-zero can be mapped to any processor



Note a_{ij} and a_{ji} are mapped to the same processor

Partitioning software

- Several high quality software packages:
 - Mondriaan, University of Utrecht
 - PaToH, Ohio State
 - Zoltan, Sandia Laboratories
 - Scotch, INRIA
 - hMetis, University of Minnesota

Parallelizing the Karp-Sipser algorithm

- Algorithm uses Bulk Synchronous Processing.
- Uses 2d partitioning of edges, with adjacency matrix of G partitioned using Mondriaan such that a_{ij} and a_{ji} are mapped to the same processor
- Each vertex "owned" by one processor which is responsible for matching it
- A processor tries to match s of its vertices in each round before communicating
 - First, match degree one vertices and then random matches.
- Local matches (v,w) can be executed immediately
- If w is non-local P_i must send a matching request to owner of w .

Algorithm executed on P_i

While there are edges left in G

Process incoming messages

For $i = 1, s$

If there exists unmatched vertex v of degree one owned by P_i
 $e = (v, w)$ where w is the neighbor of v

else

$e =$ random remaining edge (v, w) where P_i owns v

If w is owned by P_i

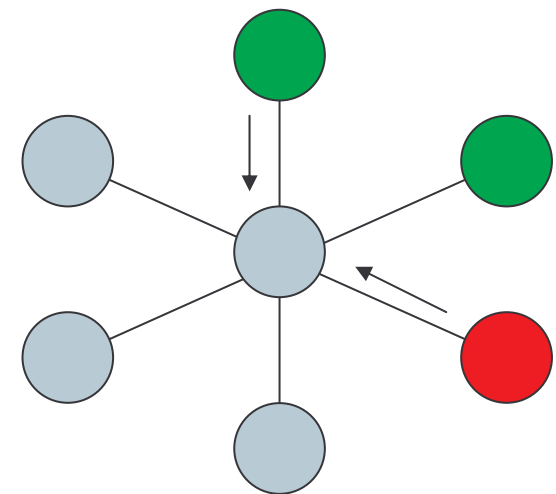
Add (v, w) to solution

Initiate removal of edges incident on v and w from G

Exchange messages with other processors

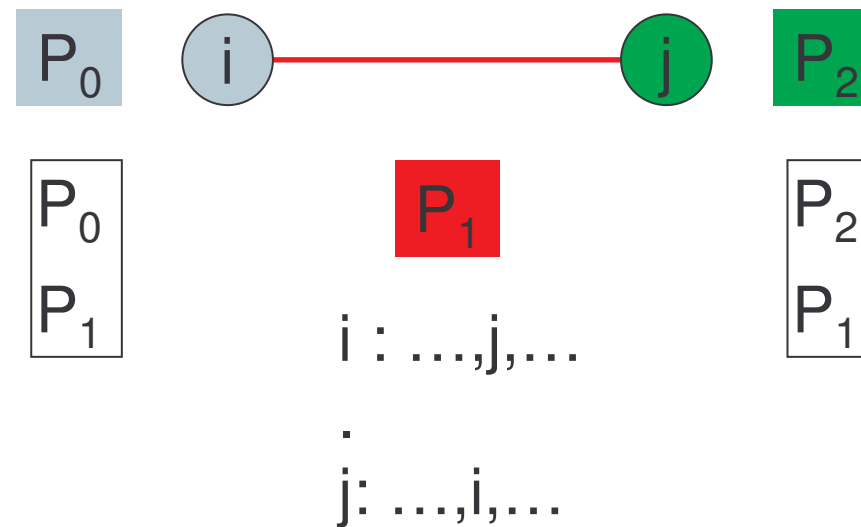
Message processing

- Main message types:
 - Matching request
 - Vertex has been removed
 - Matching succeeded (Failures are handled implicitly)
- A vertex can receive multiple matching request.
 - Only one succeeds, the rest will fail.
 - A processor will never try to match to the same vertex twice
- Total number of messages sent is
 $0.25 \text{ SpMV}(A) \leq \text{Vol}(\text{Matching}) \leq 1.5 \text{ SpMV}(A)$



Data structures

- The owner of a node i has a list of the processors that owns edges incident on vertex i
- Each owner of an edge (i,j) knows which processor owns i .

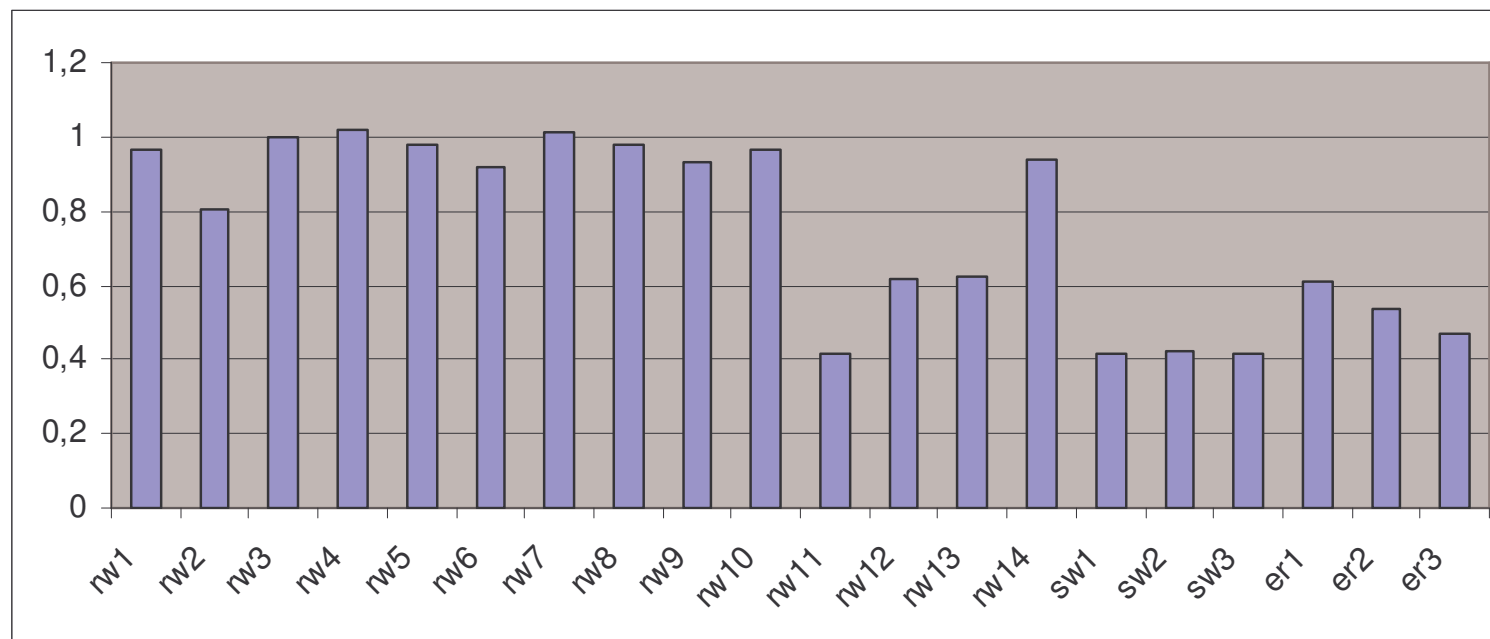


These data structures are dynamically updated as the algorithm progresses

Experiments

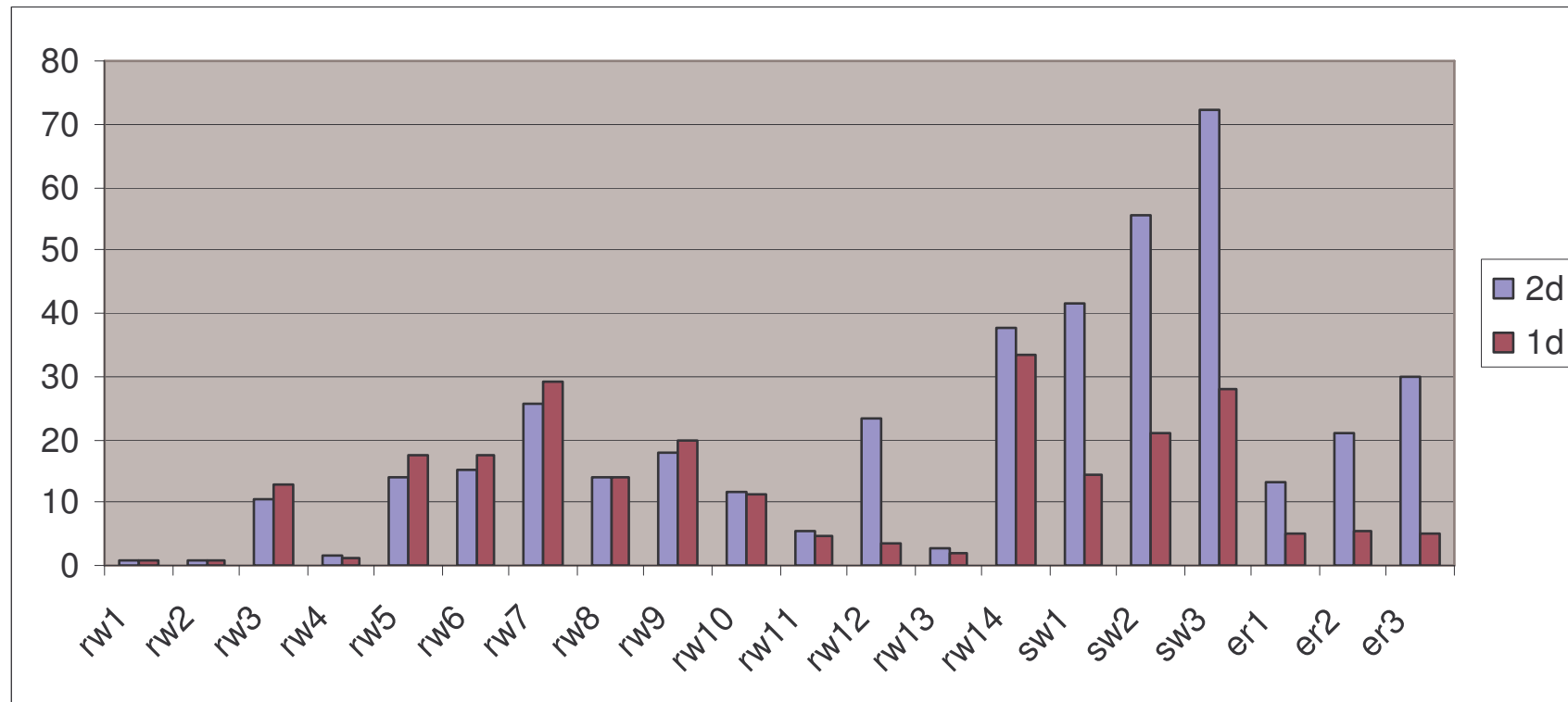
- Test sets from real-world matrices, small world matrices and Erdős-Renyi random matrices
- Performed on IBM pSeries 575 with 104 nodes
- Implemented in C++ using BSPonMPI

Relative communication volume for SpMV when using 2D partitioning vs 1D on 32 processors



Numbers for communication volume in matching are similar

Speed up of matching code when using 64 processors



Observations

- 2d outperforms 1d partitioning on complex problems but 1d almost never outperforms 2d.
- Quality of matching tends to degrade with
 - Increasing number of processors
 - Increasing s-value
 - Complexity of problems
- It should be possible to expand the 2d approach to other greedy algorithms such as for weighted matching and graph coloring.