Parallel Algorithms for Graph Similarity and Matching

Advanced Algorithms and Parallel Programming Course Project

Marco Bacis, Antonio Di Bello

Outline

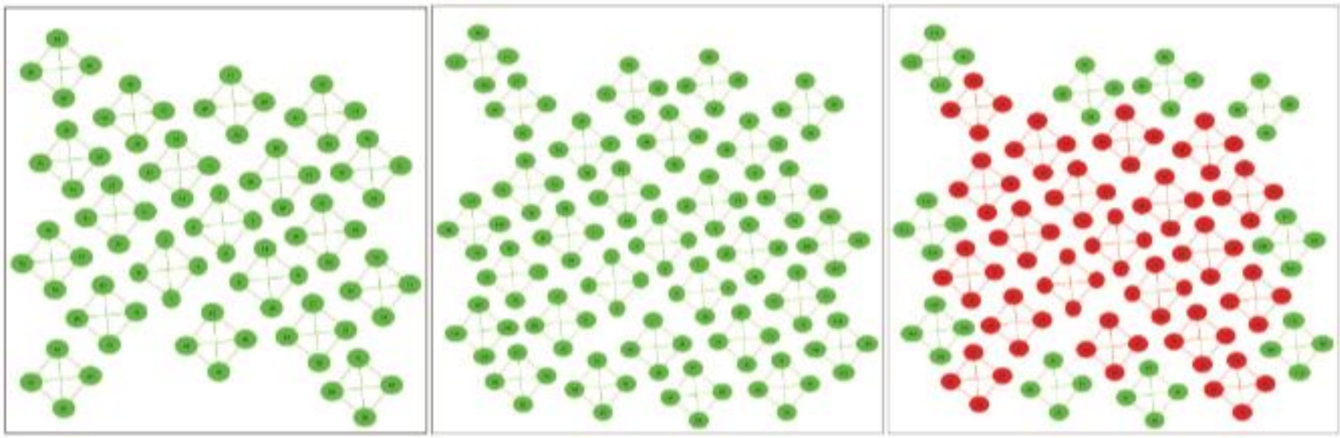
- Problem Overview
- Network Similarity Decomposition
- Auction-based matching
- Experimental Set and Results

Outline

- Problem Overview
- Network Similarity Decomposition
- Auction-based matching
- Experimental Set and Results

Problem Definition

- Given two graphs:
  - How similar is each vertex in the first graph to each vertex in the second?
  - What is the best match for each vertex in the first graph to a vertex in the second graph?



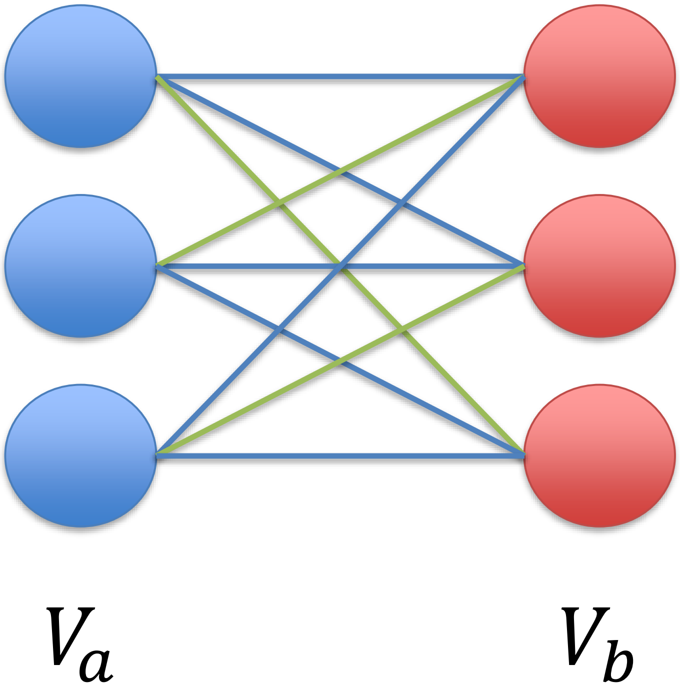
Graph Similarity

- Two main categories:
  - Single/ Global similarity score (scalar)
  - Vertex-wise similarity score (matrix)
- Approaches:
  - GRAAL family, the “seed and extend” idea
  - IsoRank, vertex similarity scores using vertex attributes and topological similarities
  - NSD, low-rank decompositions of the matrix to decouple its construction process

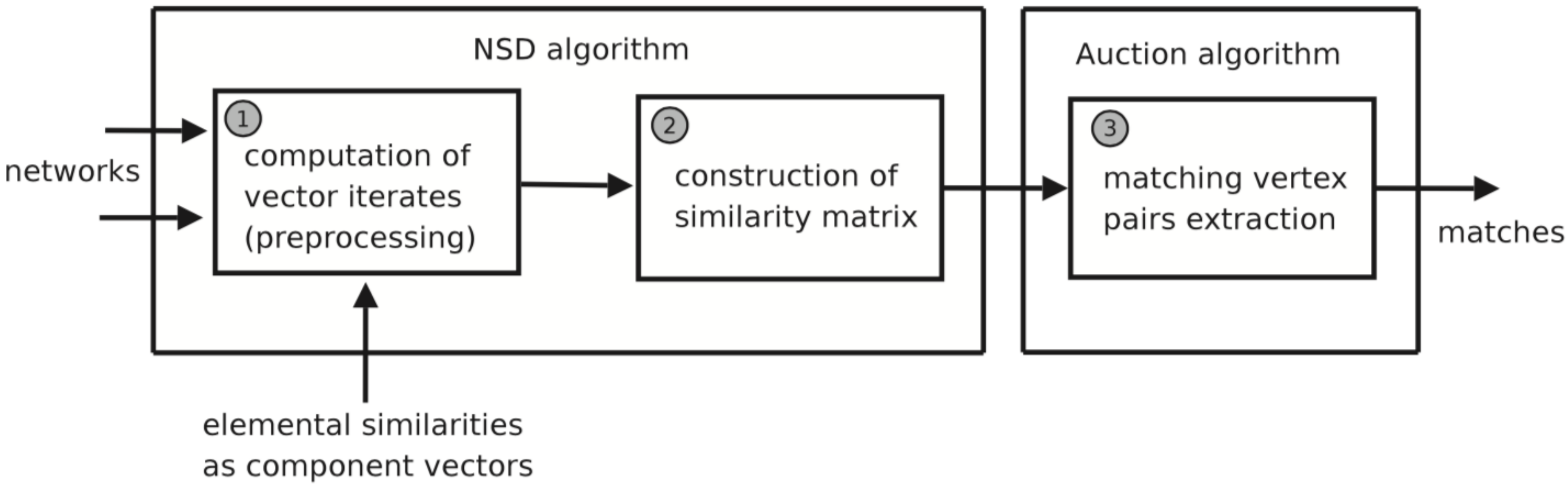
Bipartite Graph Matching

- Matching M of vertices over weighted edge :
  - a vertex is an endpoint of at most one matching edge
  - sum over the matched edges is maximized

- Implementations:
  - Augmenting path
  - Hungarian method
  - $\boxed{\text{Auction-based algorithm}}$



Overview



- Network Similarity Decomposition (NSD) matrix computation
- Auction-Based matching using the similarity matrix
- Integrated approach (NSD + Auction on the same processes)

Outline

- Problem Overview
- Network Similarity Decomposition
- Auction-based matching
- Experimental Set and Results

Network Similarity Decomposition (NSD)

- Iterative definition of X:

$$X \leftarrow \alpha \tilde{B} X \tilde{A}^T + (1 - \alpha) H$$

where:

- X = Resulting similarity matrix
- $\tilde{A}, \tilde{B}$ = Normalized/Trasposed Adjacency matrices
- α = scale factor
- H = a-priori vertex similarity matrix

Network Similarity Decomposition (NSD)

NSD relies on low-rank representations of the H matrix, into a sum of outer products of vectors.

- Singular Value Decomposition (SVD) decompose H into eigenvalue and pair of eigenvector .
- Other possible methods:
  - Non-negative Matrix Factorization (NMF)
  - or other decomposition method

This enables further possibilities of parallellization.

$$H = \sum_{i=1}^{r} \sigma_i u_i {v_i}^T$$

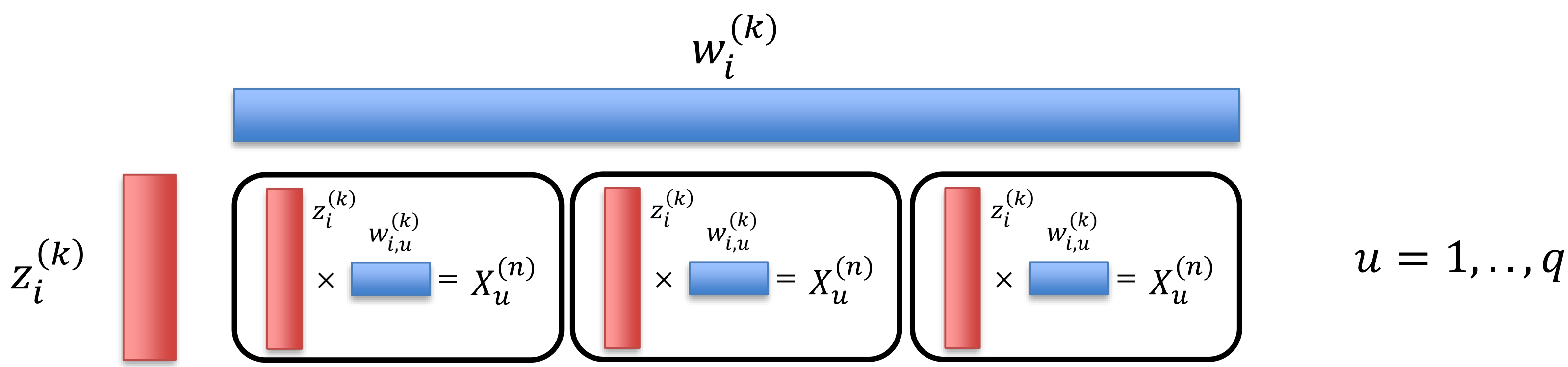
NSD Algorithm Pseudocode

Algorithm 1 NSD: Calculate $X^{(n)}$ given A, B, $\{w_i, z_i | i = 1, \ldots, s\}$, α and n

compute $\tilde{A}$, $\tilde{B}$
for $i = 1$ to s **do**
  $w_i^{(0)} \leftarrow w_i$
  $z_i^{(0)} \leftarrow z_i$
  for $k = 1$ to n **do**
    $w_i^{(k)} \leftarrow \tilde{B} w_i^{(k-1)}$
    $z_i^{(k)} \leftarrow \tilde{A} z_i^{(k-1)}$
  end for
  zero $X_i^{(n)}$
  for $k = 0$ to $n - 1$ **do**
    $X_i^{(n)} \leftarrow X_i^{(n)} + \alpha^k w_i^{(k)} {z_i^{(k)}}^T$
  end for
  $X_i^{(n)} \leftarrow (1 - \alpha) X_i^{(n)} + \alpha^n w_i^{(n)} {z_i^{(n)}}^T$
end for
$X^{(n)} \leftarrow \sum_{i=1}^{s} X_i^{(n)}$

NSD Parallel Algorithm

- Centralized (parallel) computation of w, z
- Distributed X computation
  - w split over a group of processes
  - Partial computation on each node/process



Outline

- Problem Overview
- Network Similarity Decomposition
- Auction-based matching
- Experimental Results

Auction-based graph matching

- V_a and V_b represent the set of buyers and objects where $n_a \leq n_b$

The algorithm consists of three phases:

- the initialization phase
- the bidding phase
- the assignment phase

Auction-based graph matching

Algorithm 2 Sequential Auction Algorithm for Maximum Weighted Matching

Input: Bipartite graph $G = (V_A, V_B, E, w)$
Output: Matching M

$M \leftarrow \emptyset$ ▷ *current matching*
$I \leftarrow \{i : 1 \leq i \leq n_A\}$ ▷ *set of unassigned buyers*
$p_j \leftarrow 0$ for $j = 1, \ldots, n_B$ ▷ *initialize prices for objects*
initialize(ε) ▷ *initialize* ε
while $I \neq \emptyset$ **do** ▷ *auction iteration*
  $j_i \leftarrow \arg\max_j \{x_{ij} - p_j\}$ ▷ *find best object of buyer* i
  $u_i \leftarrow x_{ij_i} - p_{j_i}$ ▷ *store profit of the most valuable object*
  $v_i \leftarrow \max_{j \neq j_i} \{x_{ij} - p_j\}$ ▷ *store second-best profit*
  $p_{j_i} \leftarrow p_{j_i} + u_i - v_i + \varepsilon$ ▷ *update price with the bid* $u_i - v_i$ *and* ε
  $M \leftarrow M \cup \{i, j_i\}$; $I \leftarrow I \setminus \{i\}$ ▷ *assign buyer to the desired object*
  $M \leftarrow M \setminus \{k, j_i\}$; $I \leftarrow I \cup \{k\}$ ▷ *free previous owner* k *if available*
  update(ε) ▷ *increment/decrement* ε
end while

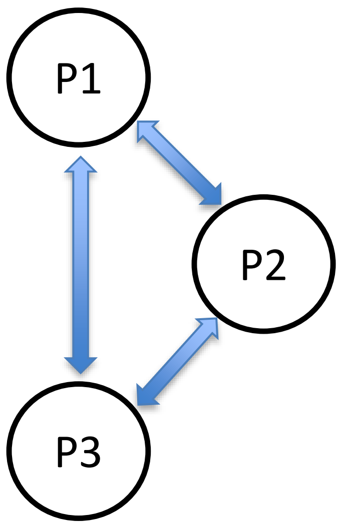
Auction-based graph matching

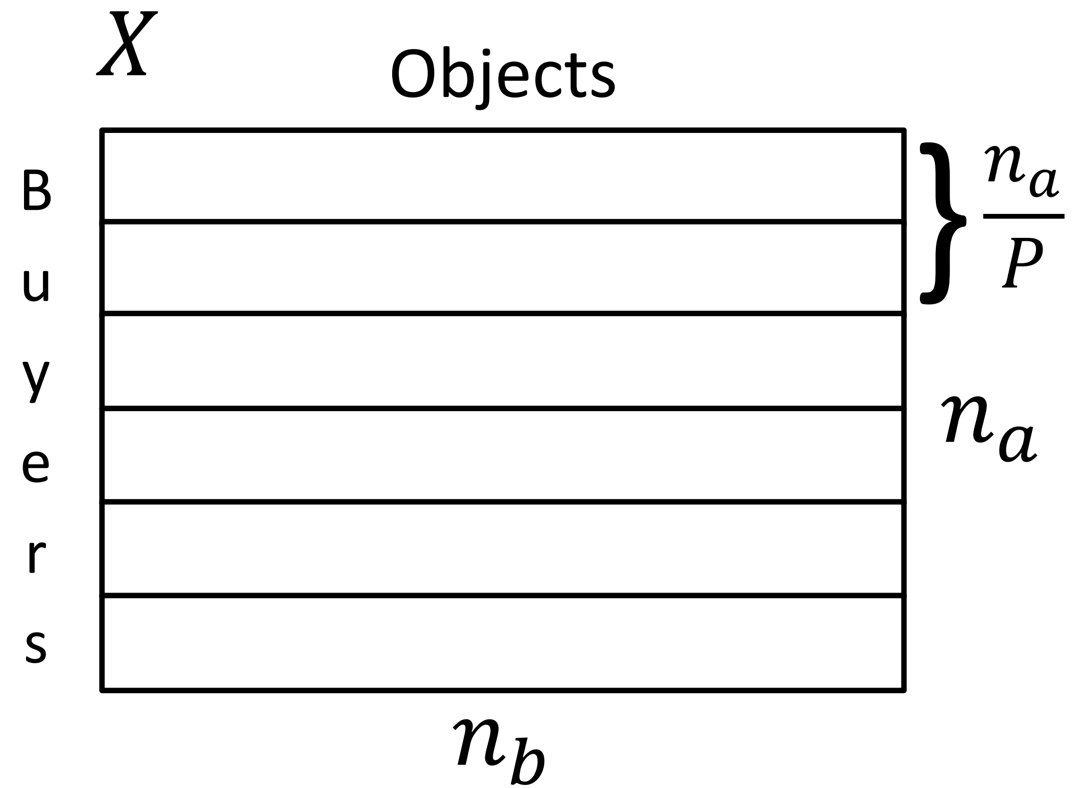
Detected parallelism:

- Bids of free buyers simultaneously computed:
  - Each free buyer computes a bid for the most-valuable object according to the current price
  - The prices of the objects are updated according to the highest bids
  - Exchanging through messages only locally altered prices

- 1D row-wise distribution of the similitary matrix to facilitate buyers partitioning

Auction-based graph matching





- Local auction on each process (local free buyers)
- Global check and price/free buyers update
- Convergence when no free buyers left

Outline

- Problem Overview
- Network Similarity Decomposition
- Auction-based matching
- Experimental Set and Results

Implementation

- **Hybrid** parallel programming model
  - **OpenMP** for MVP, similarity matrix and local auction
  - **MPI** for problem partitioning (each task works on a subset of the matrix/auction
- Tested on a single dual-socket machine
  - Ideally, one mpi task for each socket
  - Maximize number of openmp threads on each socket

Complete Algorithm

Algorithm 6 NSD-based Parallel Graph Matching

$\square$ = *root process, no labels = all processes r*
$\square$ load adjacency matrices A, B and component vectors w_i, z_i;
$\square$ compute $\tilde{A}$, $\tilde{B}$;
`broadcast` $\tilde{A}$, w_i, z_i;
distribute $\tilde{B}$ by row blocks ▷ each process r gets its $\tilde{B}_r$ part;
for all components i and steps k ($z_i^{(0)} = z_i$, $w_i^{(0)} = w_i$)
compute vector iterates $z_i^{(k)} \leftarrow \tilde{A} z_i^{(k-1)}$
compute vector iterates $w_{i,r}^{(k)} \leftarrow \tilde{B}_r w_i^{(k-1)}$, `gather` $w_i^{(k)}$ (`// matvec`);
compute *row-wise* the local similarity matrix X_r (*embarrassingly //*)
▷ `NSD`-based, *sparsify* if needed (sort row entries, keep largest ones);
compute weighted matchings by `// auction`
▷ matching permutation lands on root;
$\square$ compute number of conserved edges, similarity rate;

Complete Algorithm

Algorithm 6 NSD-based Parallel Graph Matching

1: ☐ = *root process, no labels = all processes r*
2: ☐ load adjacency matrices A, B and component vectors w_i, z_i;
3: ☐ compute $\tilde{A}$, $\tilde{B}$;

- Sparse representation of $\tilde{A}$, $\tilde{B}$
- Fast transpose/normalization thanks to low number of nonzero values

Complete Algorithm

broadcast $\tilde{A}$, w_i, z_i;
distribute $\tilde{B}$ by row blocks $\triangleright$ each process r gets its $\tilde{B}_r$ part;
for all components i and steps k ($z_i^{(0)} = z_i$, $w_i^{(0)} = w_i$)
compute vector iterates $z_i^{(k)} \leftarrow \tilde{A} z_i^{(k-1)}$
compute vector iterates $w_{i,r}^{(k)} \leftarrow \tilde{B}_r w_i^{(k-1)}$, **gather** $w_i^{(k)}$ (`// matvec`);
compute *row-wise* the local similarity matrix X_r (*embarrassingly //*)

- Sparse matrix broadcast/allgather
- OpenMP for Matrix-Vector product and for X iterations

X Iterations

```
    for (int i = 1; i < n; i++) {
        vector_t W_gathered;

        //std::cout << "Worker " << rank << " iterate " << i << std::endl;
        if (i == 1) W_gathered = W;
        else W_gathered = allgather_vector(W_i[i-1]);

        W_i[i] = matvect_prod(B, W_gathered);
        Z_i[i] = matvect_prod(A, Z_i[i-1]);
    }

    #pragma omp parallel for shared(X, W_i, Z_i)
    for(unsigned int y = 0; y < X.size1(); y++) {
        float alpha_pow = alpha;
        //if (y%1000 == 0) std::cout << "Worker " << rank << " row " << y << std::endl;
        //#pragma omp parallel for collapse(2) private(i,x,alpha_pow)
        for(int i = 0; i < n-1; i++) {
            for(unsigned int x = 0; x < X.size2(); x++) {
                X(y,x) += alpha_pow * W_i[i][y] * Z_i[i][x];
            }
            alpha_pow *= alpha;
        }
        //last step (n)
        //#pragma omp parallel for
        for(unsigned int x = 0; x < X.size2(); x++)
            X(y,x) = (1-alpha) * X(y,x) + alpha_pow * W_i[n-1][y] * Z_i[n-1][x];
    }
```

Complete Algorithm

compute weighted matchings by `// auction`
▷ matching permutation lands on root;

- Parallel argmax (OpenMP custom reduction)
- Randomized epsilon scaling
- Allgather for local auctions
  results (price and assignement) merge

Parallel argmax

```
struct BidResult res = {.obj=-1, .buyer=-1, .maxP=-FLT_MAX, .secondP=-FLT_MAX};

#pragma omp parallel
{
#pragma omp for collapse(2) reduction(bidReduce:res)
for (unsigned int i = 0; i < freeBuyer.size(); i++) {
    for (int j=0; j<nb; j++) {
        auto it = freeBuyer.begin();
        advance(it,i);
        if (X(*it,j) - price[j] > res.maxP) {
            if (j != res.obj) {
                res.secondP = res.maxP;
            }
            res.obj = j;
            res.maxP = X(*it,j) - price[j];
            res.buyer = *it;
        } else if (j!=res.obj && X(*it,j) - price[j] > res.secondP) {
            res.secondP = X(*it,j) - price[j];
        }
    }
}
}
```

Results exchange

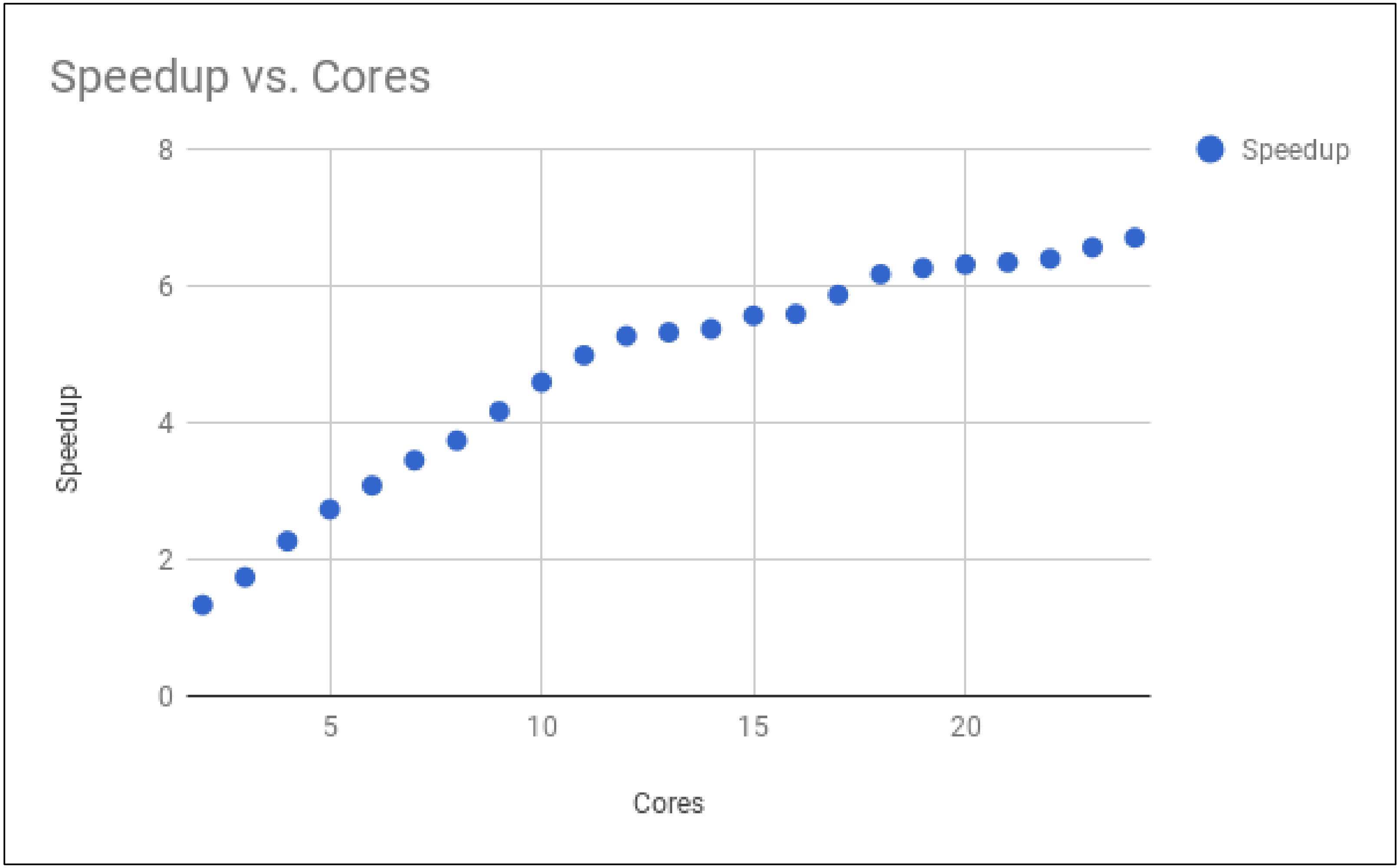
```
/* Gather changed prices */
sendBuyer.clear();
sendPrice[0] = localPrice; //price
sendPrice[1] = res.obj; // object
MPI_Allgather(sendPrice,2,MPI_FLOAT,receivedPrices,2,MPI_FLOAT,MPI_COMM_WORLD);
for (int i=0; i<worldSize*2; i+=2) {
    if (price[receivedPrices[i+1]] < receivedPrices[i]) {
        price[receivedPrices[i+1]] = receivedPrices[i];
        //se qualcuno dei miei local aveva comprato quell'oggetto ora lo perde.
        if (assigned[receivedPrices[i+1]] != -1) {

            if (debug)
                cout << worldRank <<  " : localbuyer " << assigned[receivedPrices[i+1]] << " lose obj " << receivedPrices[i+1]

            sendBuyer.push_back(assigned[receivedPrices[i+1]] + (worldRank*(na/worldSize)) + 1);

            match[assigned[receivedPrices[i+1]]] = -1;
            freeBuyer.insert(assigned[receivedPrices[i+1]]);
            assigned[receivedPrices[i+1]] = -1;


        }
    } else if (price[receivedPrices[i+1]] == receivedPrices[i] && receivedPrices[i+1] == res.obj) {
        //se qualcuno ha offerto lo stesso prezzo del mio vincitore local e il suo indice globale è minore del mio, perdo l'ogg
        localPrice--;
                }
}
```

Evaluation

- Graphs tested
  - Yeast protein: 2361 nodes
  - Wikivote graph: 8297 nodes
- Matching with subgraphs (and self)
  - Yeast -> 500/1000 nodes subgraph
  - Wikivote -> 1000 nodes subgraph

Results

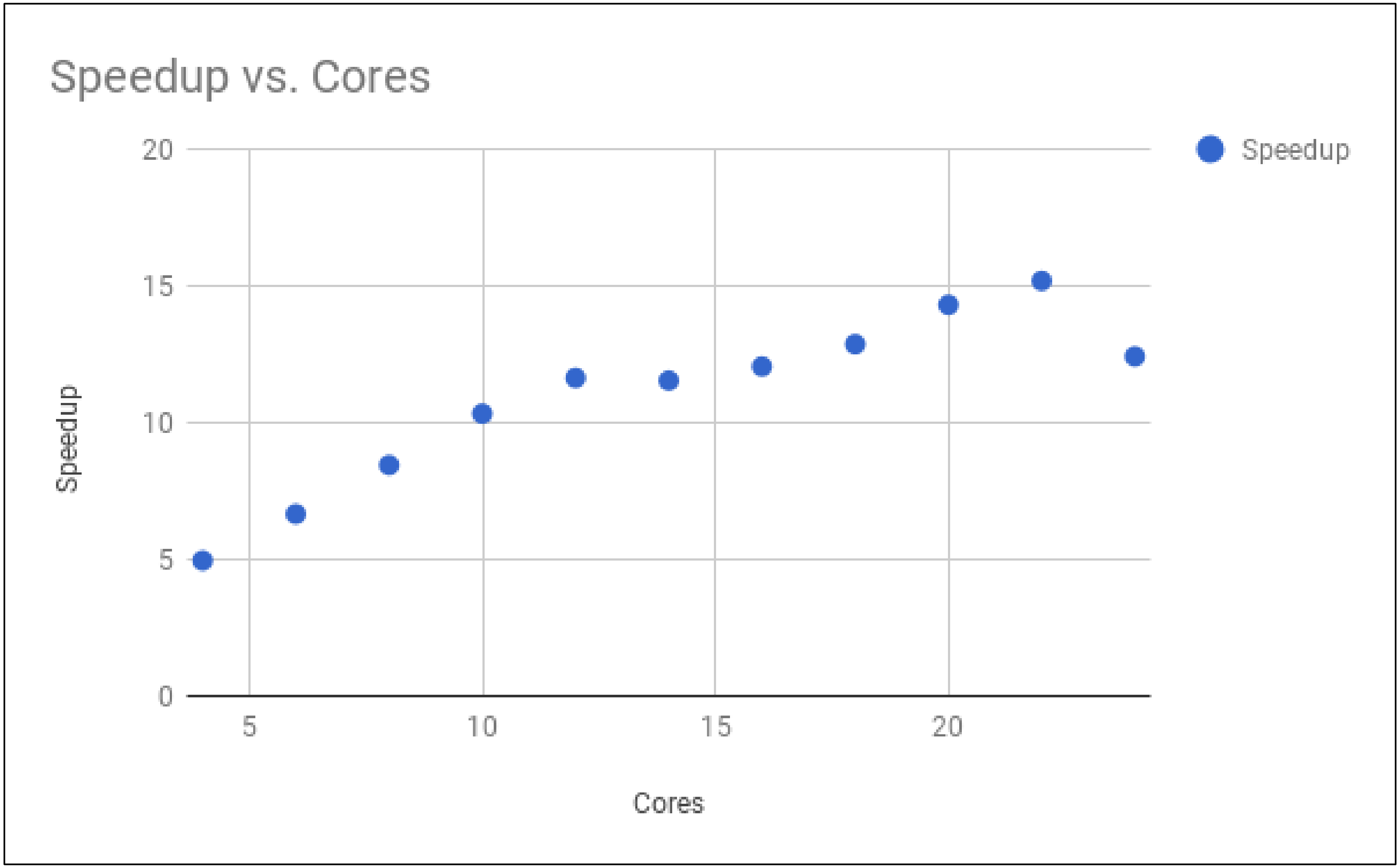
Yeast (complete) vs Yeast subgraph (500 nodes)



Only 1 MPI Task

Results

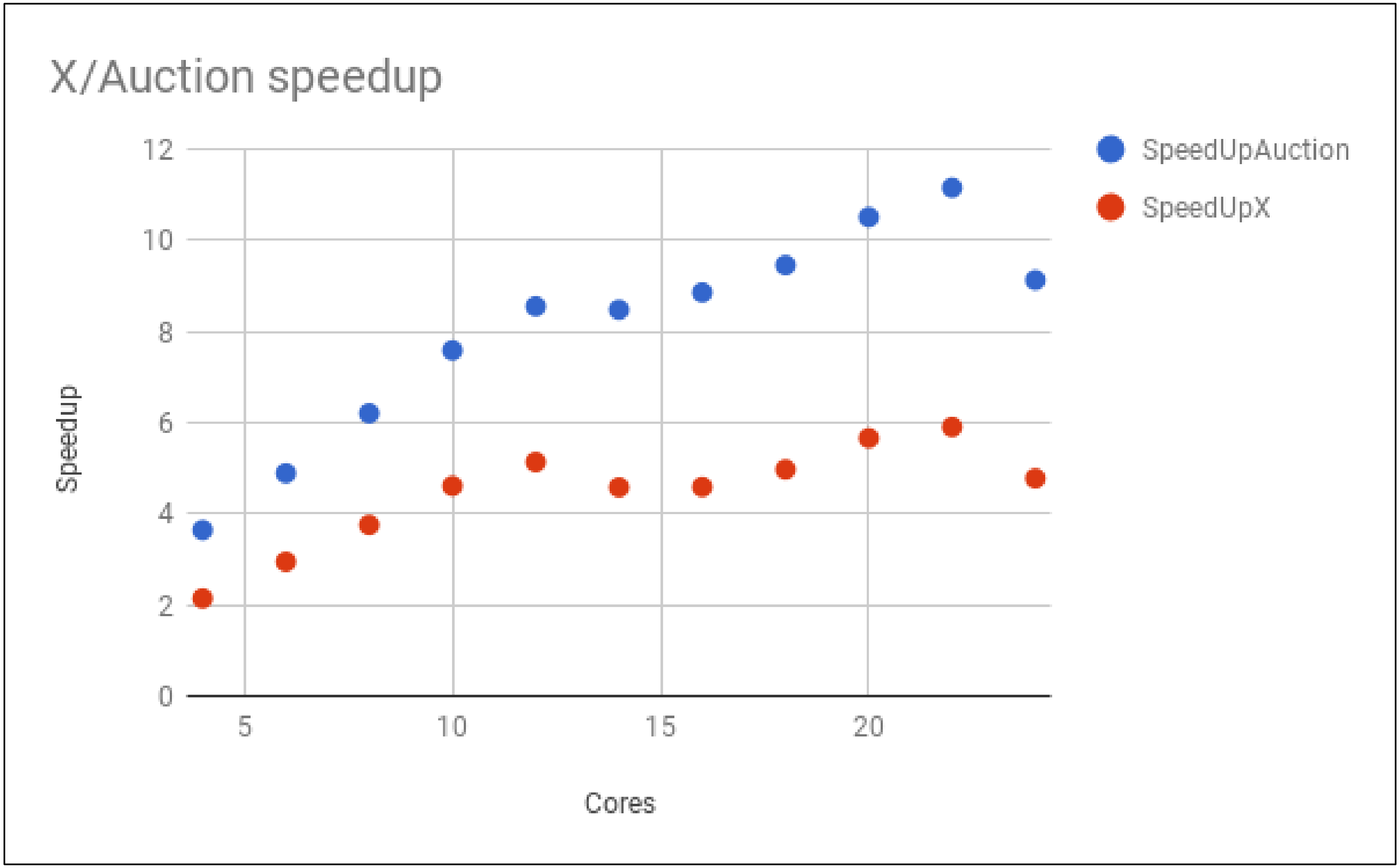
Yeast (complete) vs Yeast subgraph (500 nodes)



2 MPI Tasks

Results

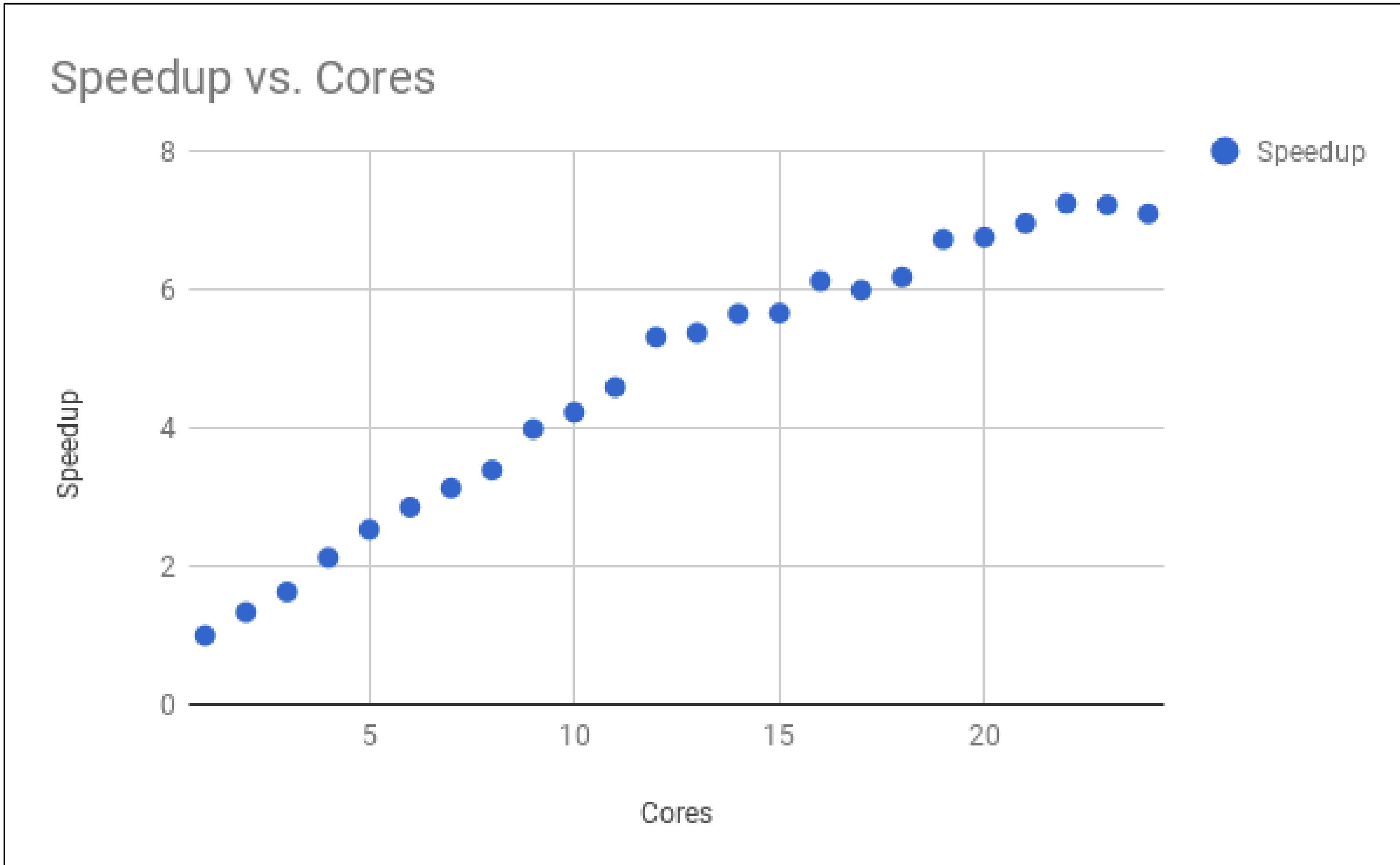
Yeast (complete) vs Yeast subgraph (500 nodes)



2 MPI Tasks

Results

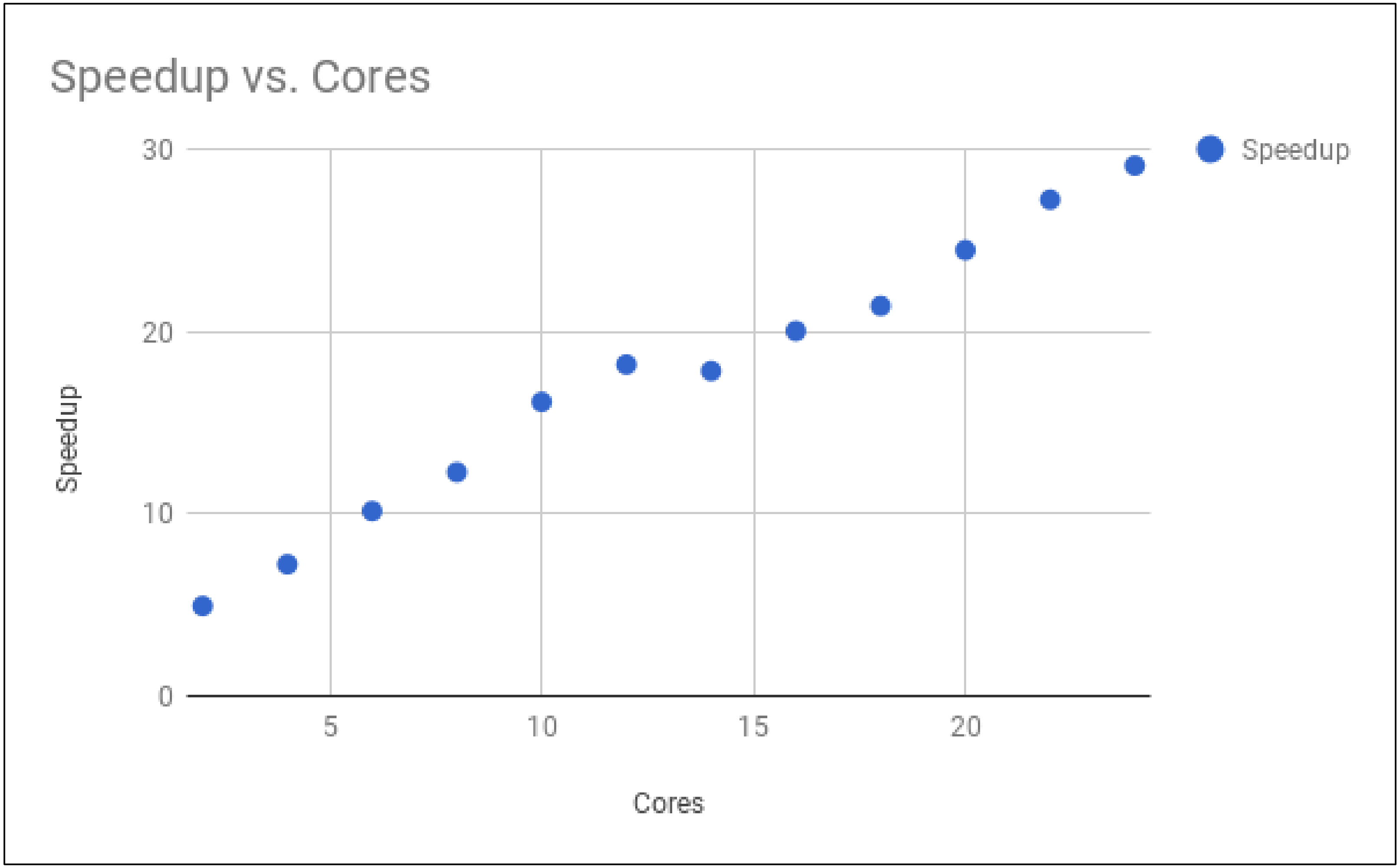
Wikivote (complete) vs Wikivote subgraph (1000 nodes)



1 MPI Task

Results

Wikivote (complete) vs Wikivote subgraph (1000 nodes)



2 MPI Tasks

Conclusions

- 2 dimensions scaling (MPI tasks – OMP threads)
- Linear speedup w.r.t number of cores used
- Slow auction convergence due to price wars
- Using 2 MPI tasks leads to best results (as expected)

Question