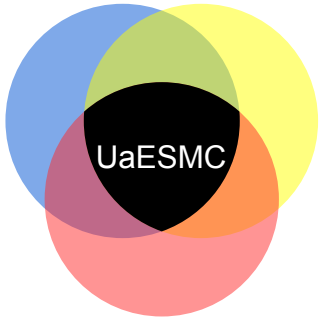


Parallel Oblivious RAM for Secure Multiparty Computation
Private computation of minimum spanning trees

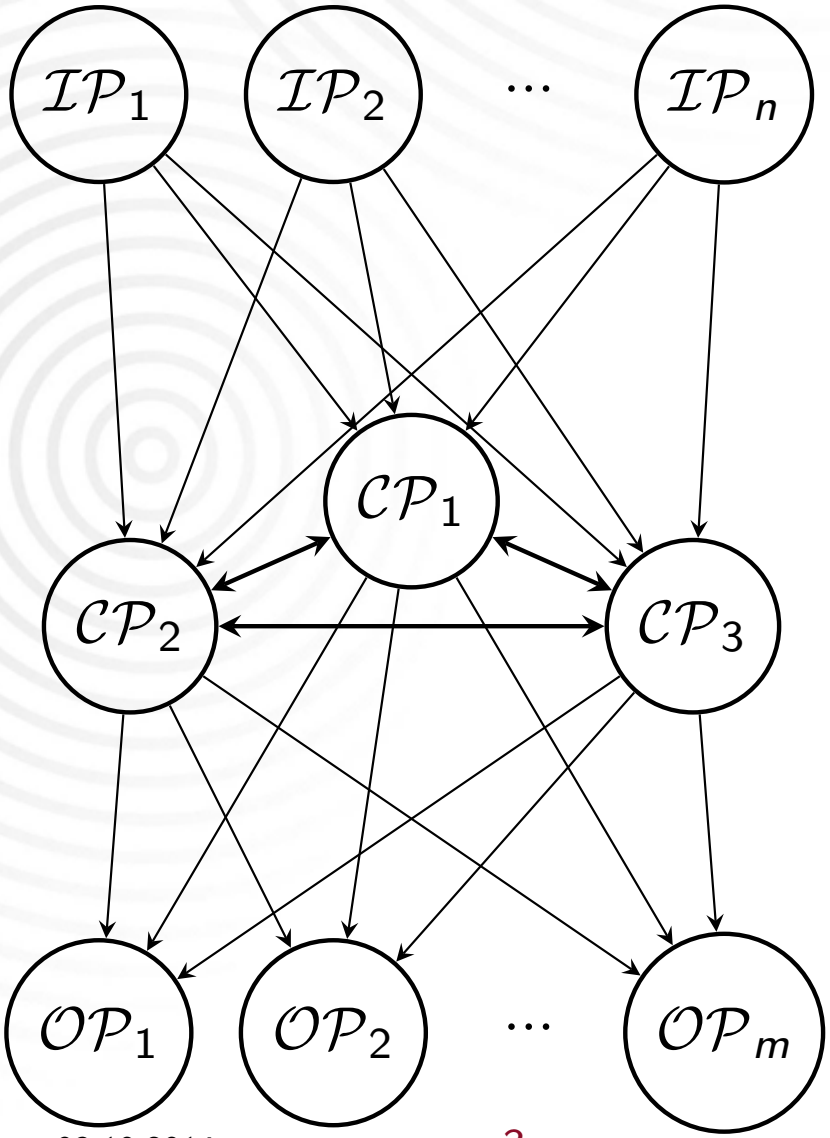
Peeter Laud

03.10.2014

Secure multiparty computation

- n parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, with inputs $x_1, \ldots, x_n$.
- want to compute $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$, where $\mathcal{P}_i$ learns y_i.
- $\mathcal{P}_i$ should not learn anything beyond x_i, y_i
  - More generally, for any $I \subseteq \{1, \ldots, n\}$, $|I| < t$, the coalition $\{\mathcal{P}_i\}_{i \in I}$ should not learn anything beyond $(x_i, y_i)_{i \in I}$.
- $\mathcal{P}_1, \ldots, \mathcal{P}_n$ run a cryptographic protocol for that end

The Sharemind model for Secure Computation



- Three computing parties
- Passive security against one party
- Additive sharing of values:
  - $[\![v]\!] = ([\![v]\!]_1, [\![v]\!]_2, [\![v]\!]_3)$
  - $[\![v]\!]_1 + [\![v]\!]_2 + [\![v]\!]_3 = v$ (in $\mathbb{Z}_N$)
- Some other sharing mechanisms also available or in the works
- $\mathcal{CP}_1, \mathcal{CP}_2, \mathcal{CP}_3$ run protocols to compute sharings of outputs from the sharings of inputs

Available operations with shared values on Sharemind

- Classification, declassification
- Addition, multiplication with public value
- Multiplication of shared values

Available operations with shared values on Sharemind

- Classification, declassification
- Addition, multiplication with public value
- Multiplication of shared values
- Equality, inequality comparisons
  - Result is a value shared over $\mathbb{Z}_2$
- Converting a sharing over $\mathbb{Z}_{2^n}$ to sharing over $\mathbb{Z}_{2^m}$
- Obtaining sharings of bits of shared values
- Division of shared values
- Floating- and fix-point operations
- ...

Sharemind's multiplication protocol

$[\![u]\!]_3$
$[\![v]\!]_3$

$\mathcal{CP}_3$

$w = uv$

$[\![u]\!]_1$
$[\![v]\!]_1$

$\mathcal{CP}_1$

$[\![u]\!]_2$
$[\![v]\!]_2$

$\mathcal{CP}_2$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$[\![u]\!]_3$
$[\![v]\!]_3$

$\mathcal{CP}_3$

$w = uv$

1. Reshare $[\![u]\!]$ and $[\![v]\!]$

$[\![u]\!]_1$
$[\![v]\!]_1$

$\mathcal{CP}_1$

$[\![u]\!]_2$
$[\![v]\!]_2$

$\mathcal{CP}_2$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$[\![u]\!]_3$
$[\![v]\!]_3$

$\mathcal{CP}_3$ $r_{3u}, r_{3v} \xleftarrow{\$} \mathbb{Z}_N$

$w = uv$

1. Reshare $[\![u]\!]$ and $[\![v]\!]$

$[\![u]\!]_1$
$[\![v]\!]_1$

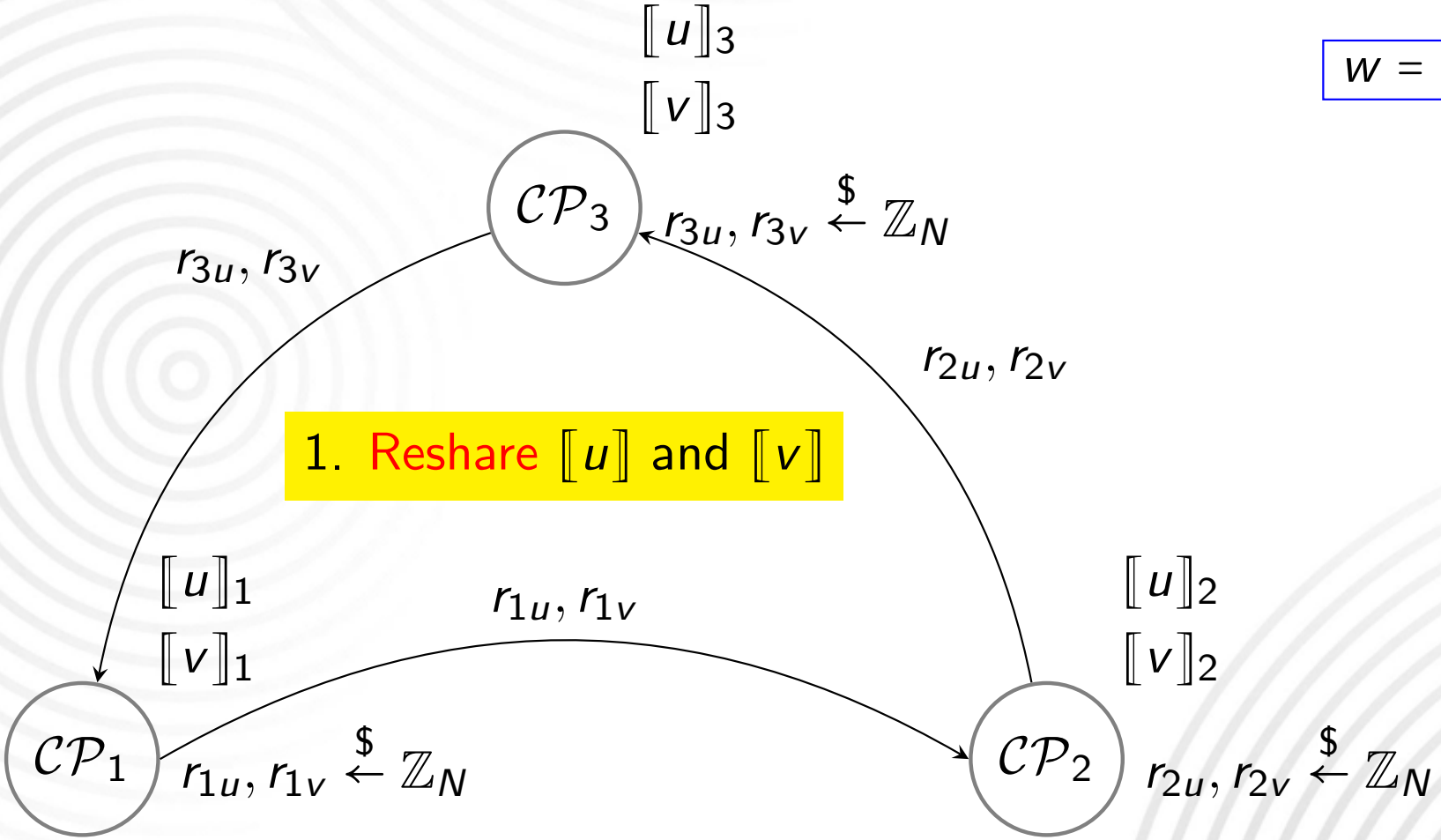
$\mathcal{CP}_1$ $r_{1u}, r_{1v} \xleftarrow{\$} \mathbb{Z}_N$

$[\![u]\!]_2$
$[\![v]\!]_2$

$\mathcal{CP}_2$ $r_{2u}, r_{2v} \xleftarrow{\$} \mathbb{Z}_N$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol



[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$$[\![u]\!]_3 := [\![u]\!]_3 + r_{3u} - r_{2u}$$
$$[\![v]\!]_3 := [\![v]\!]_3 + r_{3v} - r_{2v}$$

$w = uv$

$\mathcal{CP}_3$ $r_{3u}, r_{3v} \xleftarrow{\$} \mathbb{Z}_N$

1. Reshare $[\![u]\!]$ and $[\![v]\!]$

$$[\![u]\!]_1 := [\![u]\!]_1 + r_{1u} - r_{3u}$$
$$[\![v]\!]_1 := [\![v]\!]_1 + r_{1v} - r_{3v}$$

$\mathcal{CP}_1$ $r_{1u}, r_{1v} \xleftarrow{\$} \mathbb{Z}_N$

$$[\![u]\!]_2 := [\![u]\!]_2 + r_{2u} - r_{1u}$$
$$[\![v]\!]_2 := [\![v]\!]_2 + r_{2v} - r_{1v}$$

$\mathcal{CP}_2$ $r_{2u}, r_{2v} \xleftarrow{\$} \mathbb{Z}_N$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$[\![u]\!]_3$
$[\![v]\!]_3$

$\mathcal{CP}_3$

$w = uv$

2. Replicate shares

$[\![u]\!]_1$
$[\![v]\!]_1$

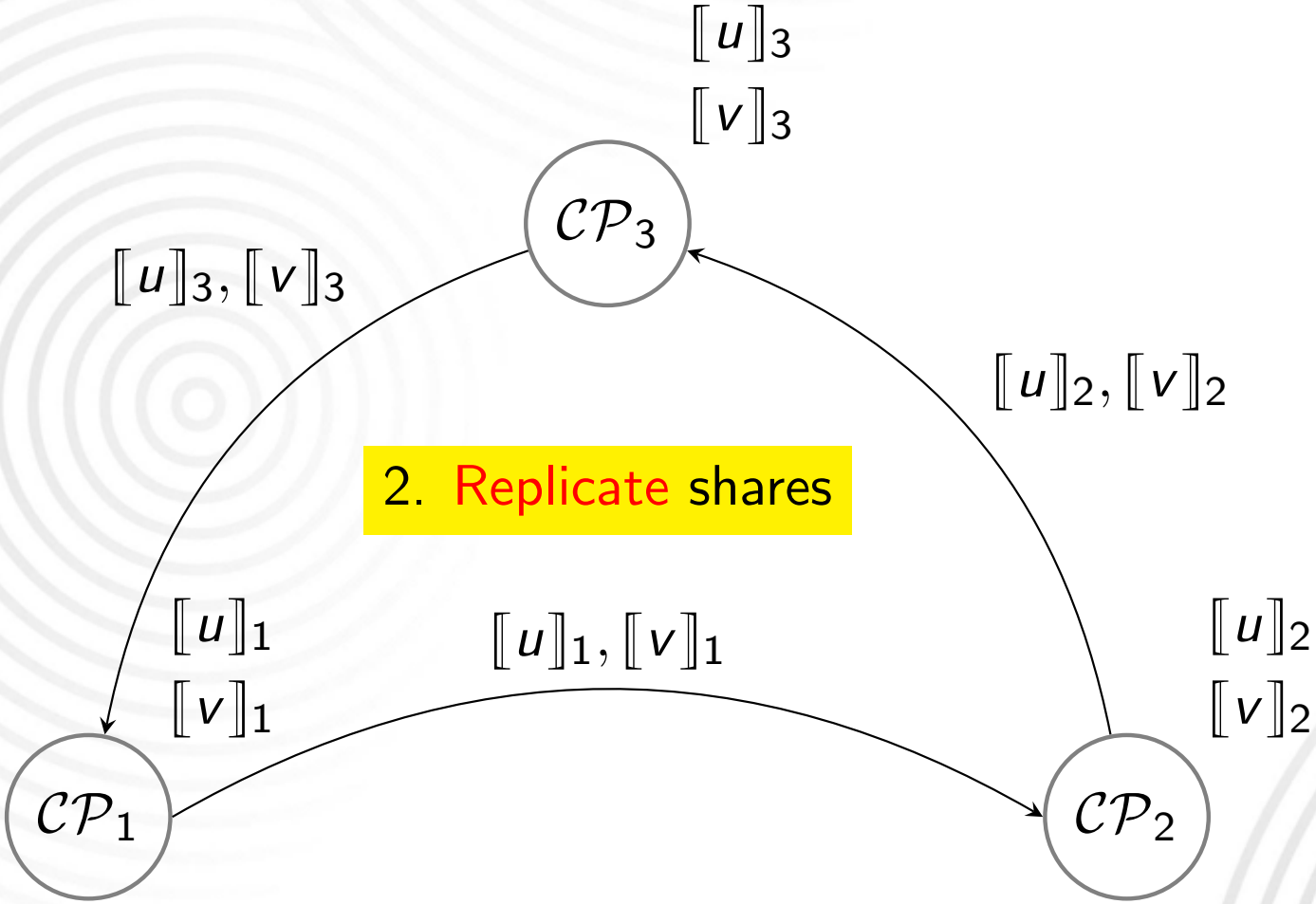
$\mathcal{CP}_1$

$[\![u]\!]_2$
$[\![v]\!]_2$

$\mathcal{CP}_2$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol



[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$[\![u]\!]_3 \quad [\![u]\!]_2$
$[\![v]\!]_3 \quad [\![v]\!]_2$

$\mathcal{CP}_3$

$w = uv$

$[\![u]\!]_1 \quad [\![u]\!]_3$
$[\![v]\!]_1 \quad [\![v]\!]_3$

$\mathcal{CP}_1$

$[\![u]\!]_2 \quad [\![u]\!]_1$
$[\![v]\!]_2 \quad [\![v]\!]_1$

$\mathcal{CP}_2$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$\mathcal{CP}_3$: $[\![u]\!]_3$ $[\![u]\!]_2$ $[\![v]\!]_3$ $[\![v]\!]_2$ $[\![w]\!]_3$

$$w = uv$$

$$w = \sum_{i,j=1,1}^{3,3} u_i v_j$$

3. Compute $[\![w]\!]_i := [\![u]\!]_i[\![v]\!]_i + [\![u]\!]_i[\![v]\!]_{i-1} + [\![u]\!]_{i-1}[\![v]\!]_i$

$\mathcal{CP}_1$: $[\![u]\!]_1$ $[\![u]\!]_3$ $[\![v]\!]_1$ $[\![v]\!]_3$ $[\![w]\!]_1$

$\mathcal{CP}_2$: $[\![u]\!]_2$ $[\![u]\!]_1$ $[\![v]\!]_2$ $[\![v]\!]_1$ $[\![w]\!]_2$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$\mathcal{CP}_3$ $[\![u]\!]_3$ $[\![u]\!]_2$ $[\![v]\!]_3$ $[\![v]\!]_2$ $[\![w]\!]_3$

$w = uv$

4. Reshare $[\![w]\!]$

$\mathcal{CP}_1$ $[\![u]\!]_1$ $[\![u]\!]_3$ $[\![v]\!]_1$ $[\![v]\!]_3$ $[\![w]\!]_1$

$\mathcal{CP}_2$ $[\![u]\!]_2$ $[\![u]\!]_1$ $[\![v]\!]_2$ $[\![v]\!]_1$ $[\![w]\!]_2$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$\mathcal{CP}_3$

$[\![u]\!]_3$ $[\![u]\!]_2$

$[\![v]\!]_3$ $[\![v]\!]_2$

$[\![w]\!]_3$

$r_{3w} \xleftarrow{\$} \mathbb{Z}_N$

$w = uv$

4. Reshare $[\![w]\!]$

$\mathcal{CP}_1$

$[\![u]\!]_1$ $[\![u]\!]_3$

$[\![v]\!]_1$ $[\![v]\!]_3$

$[\![w]\!]_1$

$r_{1w} \xleftarrow{\$} \mathbb{Z}_N$

$\mathcal{CP}_2$

$[\![u]\!]_2$ $[\![u]\!]_1$

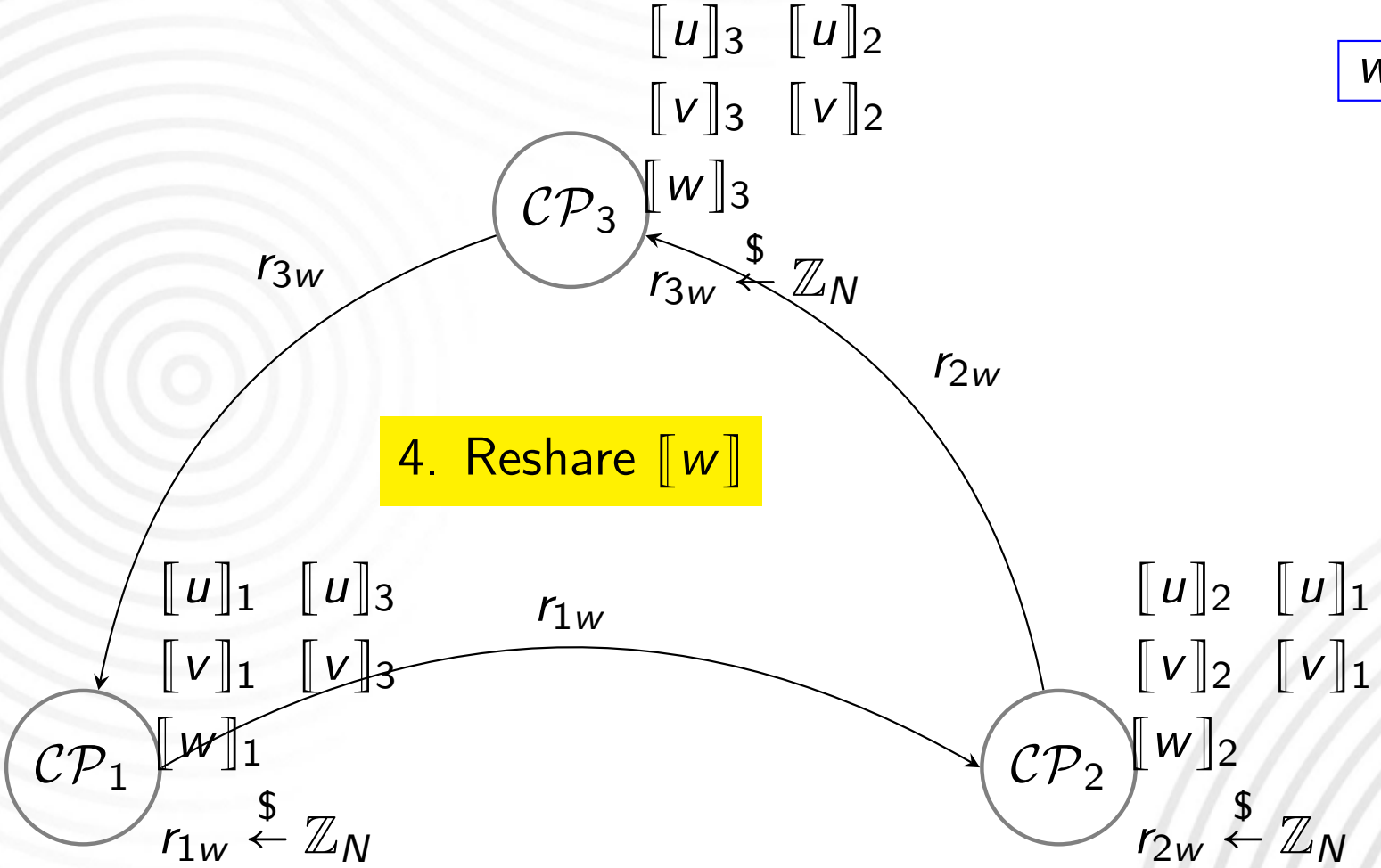
$[\![v]\!]_2$ $[\![v]\!]_1$

$[\![w]\!]_2$

$r_{2w} \xleftarrow{\$} \mathbb{Z}_N$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol



[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

$\mathcal{CP}_3$

$[\![u]\!]_3 \quad [\![u]\!]_2$

$[\![v]\!]_3 \quad [\![v]\!]_2$

$[\![w]\!]_3 := [\![w]\!]_3 + r_{3w} - r_{2w}$

$r_{3w} \xleftarrow{\$} \mathbb{Z}_N$

$w = uv$

4. Reshare $[\![w]\!]$

$\mathcal{CP}_1$

$[\![u]\!]_1 \quad [\![u]\!]_3$

$[\![v]\!]_1 \quad [\![v]\!]_3$

$[\![w]\!]_1 := [\![w]\!]_1 + r_{1w} - r_{3w}$

$r_{1w} \xleftarrow{\$} \mathbb{Z}_N$

$\mathcal{CP}_2$

$[\![u]\!]_2 \quad [\![u]\!]_1$

$[\![v]\!]_2 \quad [\![v]\!]_1$

$[\![w]\!]_2 := [\![w]\!]_2 + r_{2w} - r_{1w}$

$r_{2w} \xleftarrow{\$} \mathbb{Z}_N$

[D. Bogdanov, M. Niitsoo, T. Toft, J. Willemson. High-performance secure multi-party computation for data mining applications. Int. J. of Information Security, 11(6):403–418, 2012]

Sharemind's multiplication protocol

- Let $[\![u]\!]$ denote the following replicated sharing of u:
  - $\mathcal{CP}_i$ knows $[\![u]\!]_i, [\![u]\!]_{i-1}$
  - $[\![u]\!]_1 + [\![u]\!]_2 + [\![u]\!]_3 = u$
- Multiplication protocol used the following operations:
  - Reshare($[\![u]\!]$);
    - Parties require access to pairwise common sources of randomness
  - $[\![u]\!] \mapsto [\![u]\!]$ [requires communication];
  - $([\![u]\!], [\![v]\!]) \mapsto [\![uv]\!]$

What more do we want?

Reading from a vector

$\mathrm{read}([\![a_1]\!], \ldots, [\![a_n]\!]; [\![i]\!]) \mapsto [\![a_i]\!]$

Writing to a vector

$$\mathrm{write}([\![a_1]\!], \ldots, [\![a_n]\!]; [\![i]\!], [\![v]\!]) \mapsto \\ ([\![a_1]\!], \ldots, [\![a_{i-1}]\!], [\![v]\!], [\![a_{i+1}]\!], \ldots, [\![a_n]\!])$$

Desired complexity

- $O(\log^c n)$ for small c
  - strongly desire $c = 1$
- Constant hidden in O should be reasonable, too

Parallel execution

- Each non-trivial computation step requires communication
- Latency adds up fast
- Hence we try to parallelize our privacy-preserving applications as much as possible
  - SIMD paradigm is strong for us
- If we want to read/write from/to a vector, we probably want to do it many times

Parallel reads and writes

Parallel read from a vector

$$\mathrm{read}([\![a_1]\!], \ldots, [\![a_n]\!]; [\![i_1]\!], \ldots, [\![i_m]\!]) \mapsto ([\![a_{i_1}]\!], \ldots, [\![a_{i_m}]\!])$$

Parallel write to a vector

$$\begin{aligned}\mathrm{write}(\ & [\![a_1]\!], \ldots, [\![a_n]\!]; \\ & [\![i_1]\!], \ldots, [\![i_m]\!]; \\ & [\![v_1]\!], \ldots, [\![v_m]\!]) \\ & \mapsto ([\![b_1]\!], \ldots, [\![b_n]\!]),\end{aligned}$$

where $[\![\vec{b}]\!]$ is $[\![\vec{a}]\!]$ after the writing:

$$b_j = \begin{cases} a_j, & \text{if } j \notin \{i_1, \ldots, i_m\} \\ v_k, & \text{if } i_k = j \end{cases}$$

Parallel reads and writes

Parallel read from a vector

$$\mathrm{read}(\llbracket a_1 \rrbracket, \ldots, \llbracket a_n \rrbracket; \llbracket i_1 \rrbracket, \ldots, \llbracket i_m \rrbracket) \mapsto (\llbracket a_{i_1} \rrbracket, \ldots, \llbracket a_{i_m} \rrbracket)$$

Parallel write to a vector

$$\begin{aligned}\mathrm{write}(\ & \llbracket a_1 \rrbracket, \ldots, \llbracket a_n \rrbracket; \\ & \llbracket i_1 \rrbracket, \ldots, \llbracket i_m \rrbracket; \\ & \llbracket v_1 \rrbracket, \ldots, \llbracket v_m \rrbracket\ ; \\ & \llbracket p_1 \rrbracket, \ldots, \llbracket p_m \rrbracket) \mapsto (\llbracket b_1 \rrbracket, \ldots, \llbracket b_n \rrbracket),\end{aligned}$$

where $\llbracket \vec{b} \rrbracket$ is $\llbracket \vec{a} \rrbracket$ after the writing:

$$b_j = \begin{cases} a_j, & \text{if } j \notin \{i_1, \ldots, i_m\} \\ v_k, & \text{if } i_k = j \text{ and } p_k = \min\{p_\ell \mid i_\ell = j\} \end{cases}$$

On complexity

- Vector of length n; perform m operations
- Desired complexity: $O(m \log n)$
- We can achieve $O((m+n)\log(m+n))$

Parallel Random Access Machines (PRAM)

- PRAM — a theoretical model for parallel computation
- Many processors, single memory bank
- Each processor executes its own program
- Processors run in synchrony
- Different ways to resolve concurrency in memory cell accesses
  - EREW, CREW, common-CRCW, arbitrary-CRCW, priority-CRCW
- A PRAM algorithm is efficient if it works
  - with $O(n^c)$ processors,
  - in $O(\log^d n)$ time.
- PRAM algorithms have been actively researched
  - For many tasks, there are algorithms with $d \leq 1$.

Asymptotic overhead of our protocols

- $O((\hat{m}+\hat{n})\log(\hat{m}+\hat{n}))$ for $\hat{m}$ accesses to vector of length $\hat{n}$
  - typically, $\hat{m}$ is $O(\hat{n})$
- "No overhead" would be $O(\hat{m})$

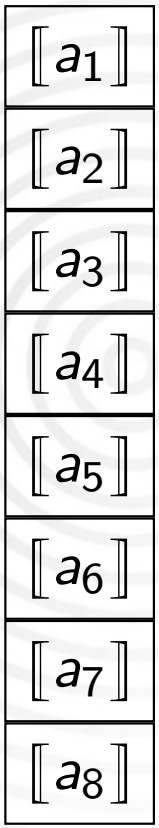
Asymptotic overhead of our protocols

- $O((\hat{m}+\hat{n})\log(\hat{m}+\hat{n}))$ for $\hat{m}$ accesses to vector of length $\hat{n}$
  - typically, $\hat{m}$ is $O(\hat{n})$
- "No overhead" would be $O(\hat{m})$
- Let algorithm $\mathcal{A}$ work with P processors in T time steps
- $\hat{n} \leq PT$ and $\hat{m} \geq \hat{n}/T$ (on average)
  - in many algorithms, $\hat{m}$ is actually $\Omega(\hat{n})$
  - assume: T is small wrt. P, $\hat{n}$ or $\hat{m}$
- $O((\hat{m}+\hat{n})\log(\hat{m}+\hat{n}))$ is $O(\hat{m}(1+T)(\log\hat{m}+\log(1+T)))$
  - ...which is $O(\hat{m}T\log\hat{m})$.

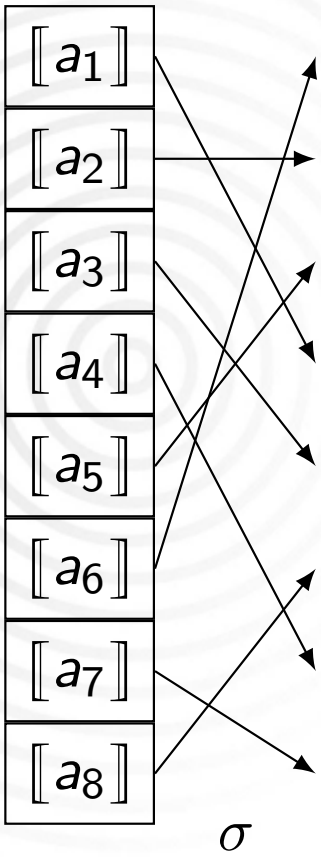
Asymptotic overhead of our protocols

- $O((\hat{m}+\hat{n})\log(\hat{m}+\hat{n}))$ for $\hat{m}$ accesses to vector of length $\hat{n}$
  - typically, $\hat{m}$ is $O(\hat{n})$
- "No overhead" would be $O(\hat{m})$
- Let algorithm $\mathcal{A}$ work with P processors in T time steps
- $\hat{n} \leq PT$ and $\hat{m} \geq \hat{n}/T$ (on average)
  - in many algorithms, $\hat{m}$ is actually $\Omega(\hat{n})$
  - assume: T is small wrt. P, $\hat{n}$ or $\hat{m}$
- $O((\hat{m}+\hat{n})\log(\hat{m}+\hat{n}))$ is $O(\hat{m}(1+T)(\log\hat{m}+\log(1+T)))$
  - ... which is $O(\hat{m}T\log\hat{m})$.
- Let $P = O(n^c)$ and $T = O(\log^d n)$ for the task size n
- Asymptotic overhead of our protocols is $O(\log^{d+1} n)$
  - If $\hat{m}$ is $\Omega(\hat{n})$ then the overhead is only $O(\log n)$

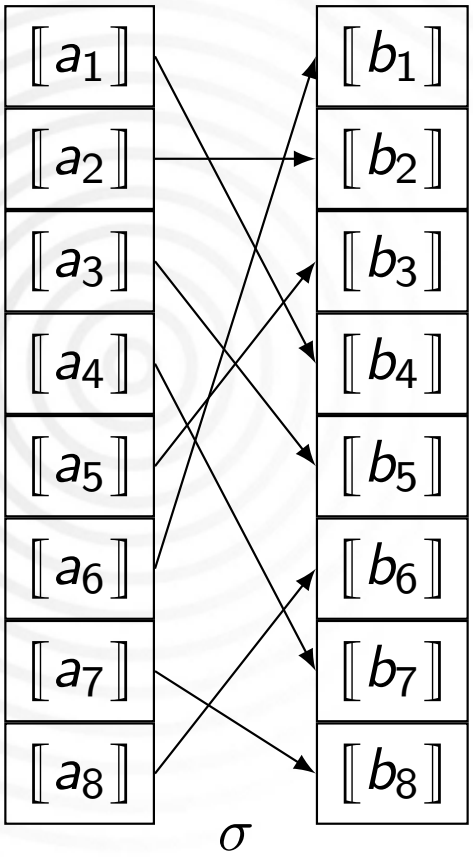
Private shuffle



Private shuffle

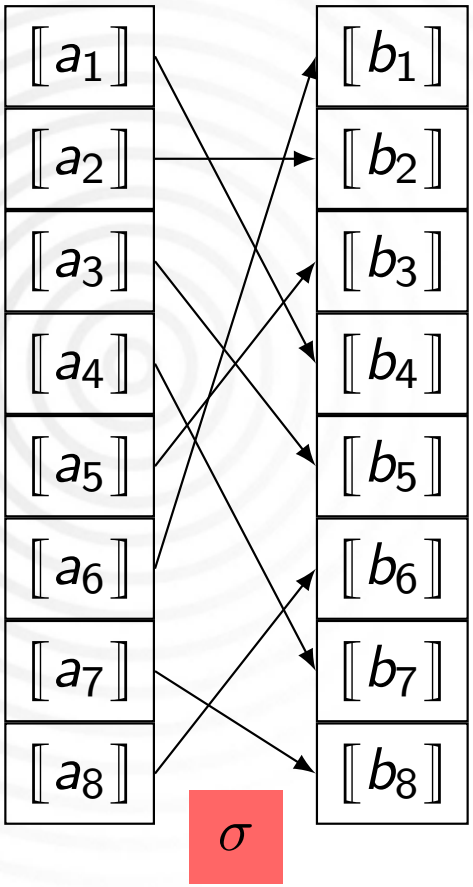


Private shuffle



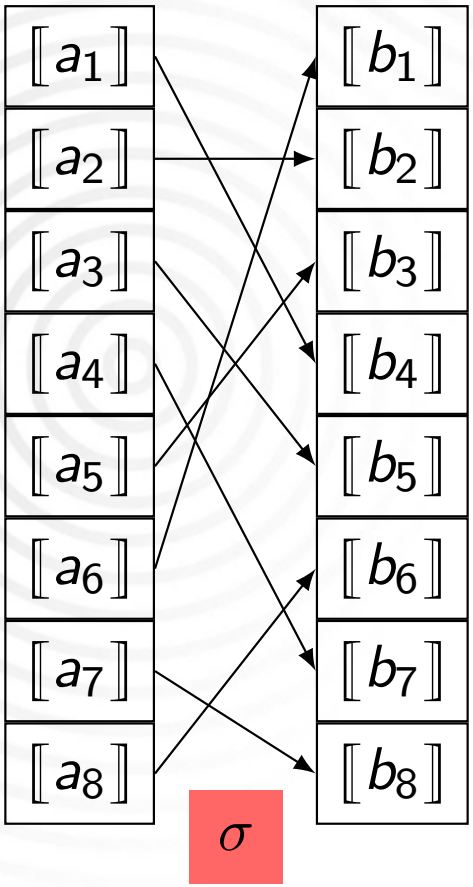
- $b_i = a_{\sigma(i)}$ for all $i \in \{1, \ldots, n\}$

Private shuffle



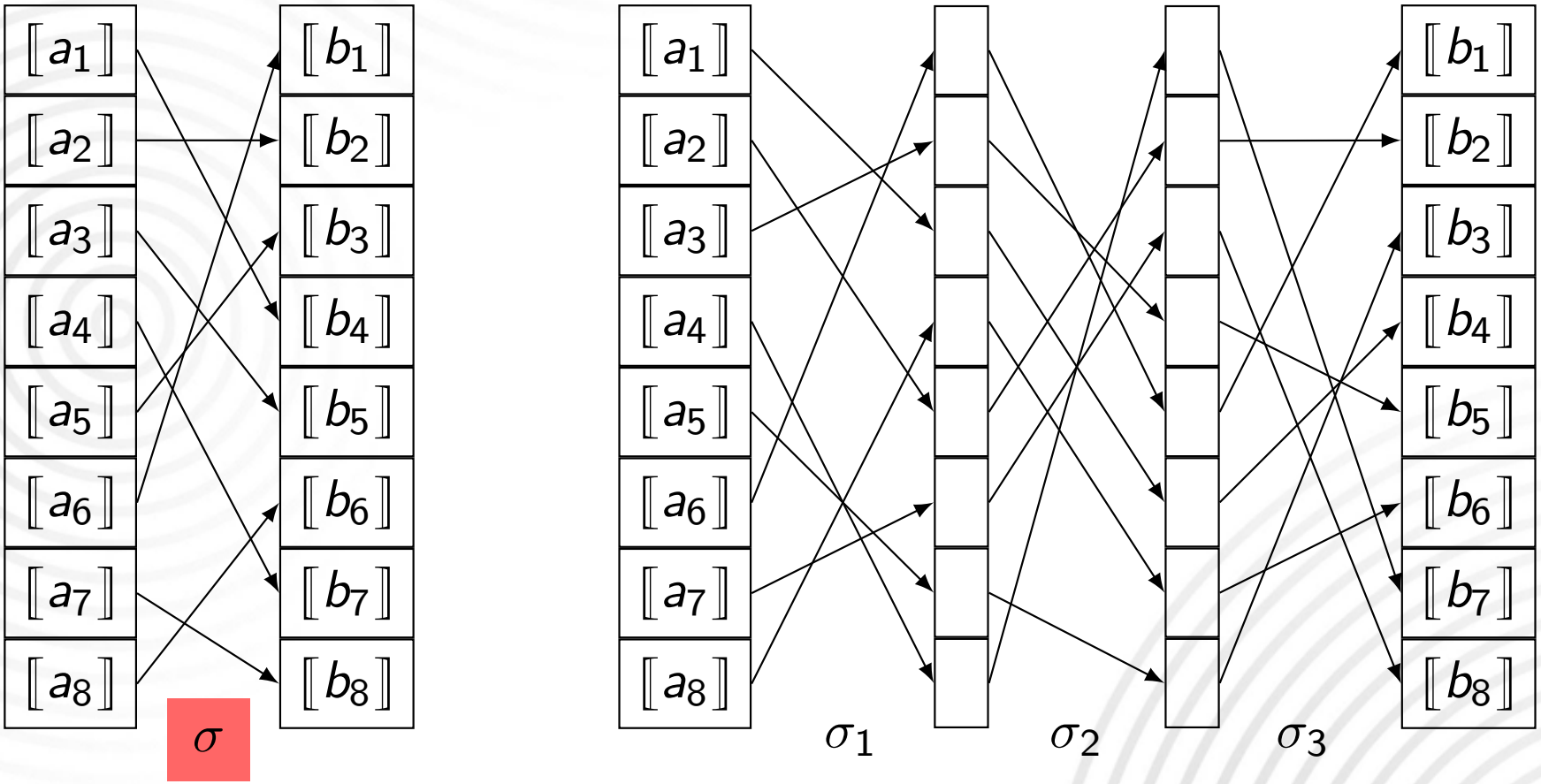
- $b_i = a_{\sigma(i)}$ for all $i \in \{1, \ldots, n\}$
- $\sigma \in S_n$ is provided by an input party
- How to represent σ and do the shuffle if σ itself is private?

Private shuffle

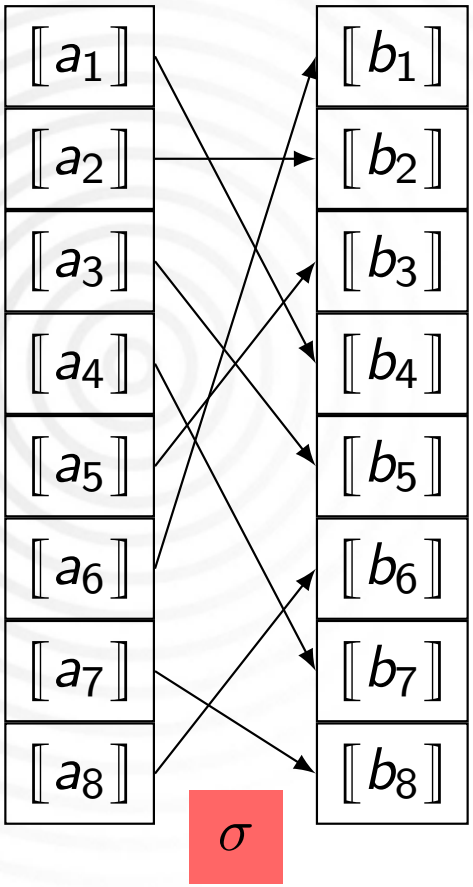


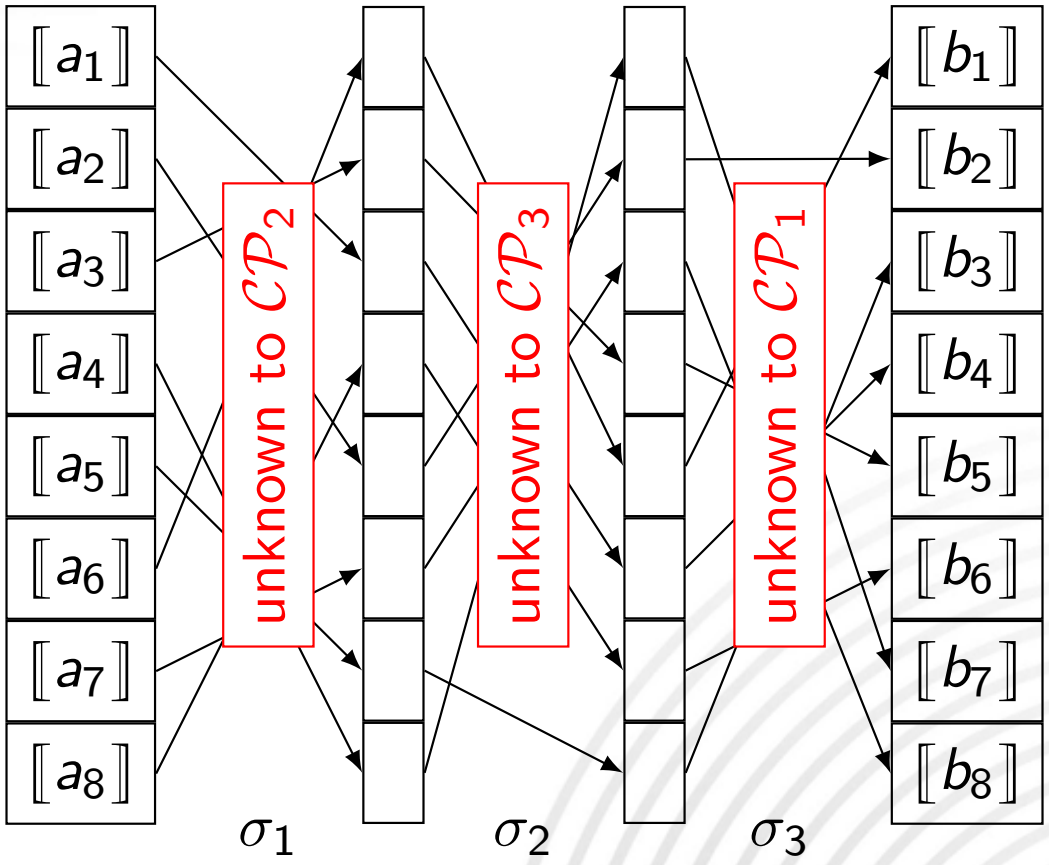
- $b_i = a_{\sigma(i)}$ for all $i \in \{1, \ldots, n\}$
- $\sigma \in S_n$ is provided by an input party
- How to represent σ and do the shuffle if σ itself is private?
- $\langle\sigma\rangle = ((\sigma_1, \sigma_2), (\sigma_2, \sigma_3), (\sigma_3, \sigma_1))$
  - $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3$;
  - $\sigma_1, \sigma_2, \sigma_3$ are random elements of S_n.

Private shuffle

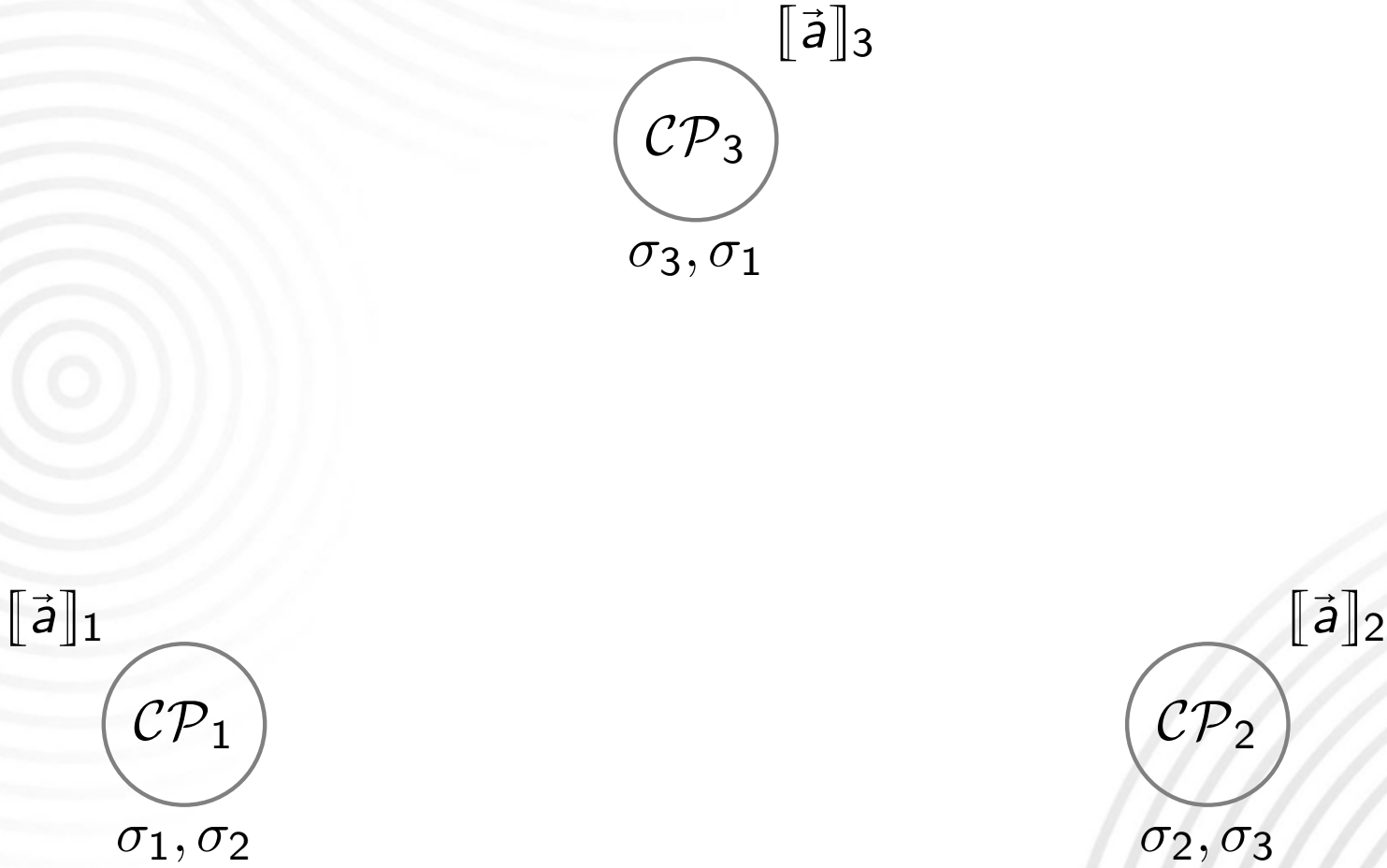


Private shuffle



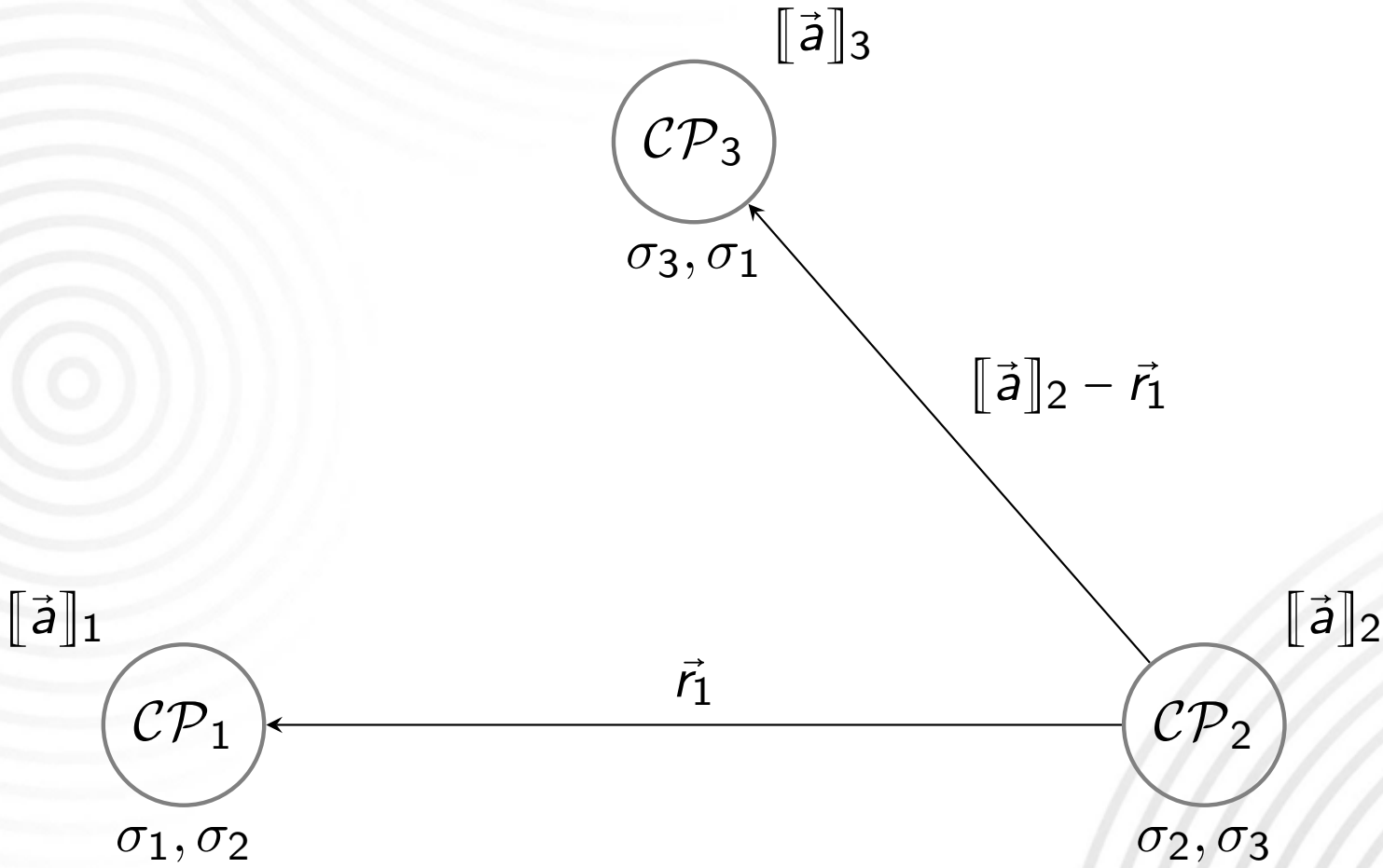


Shuffling protocol



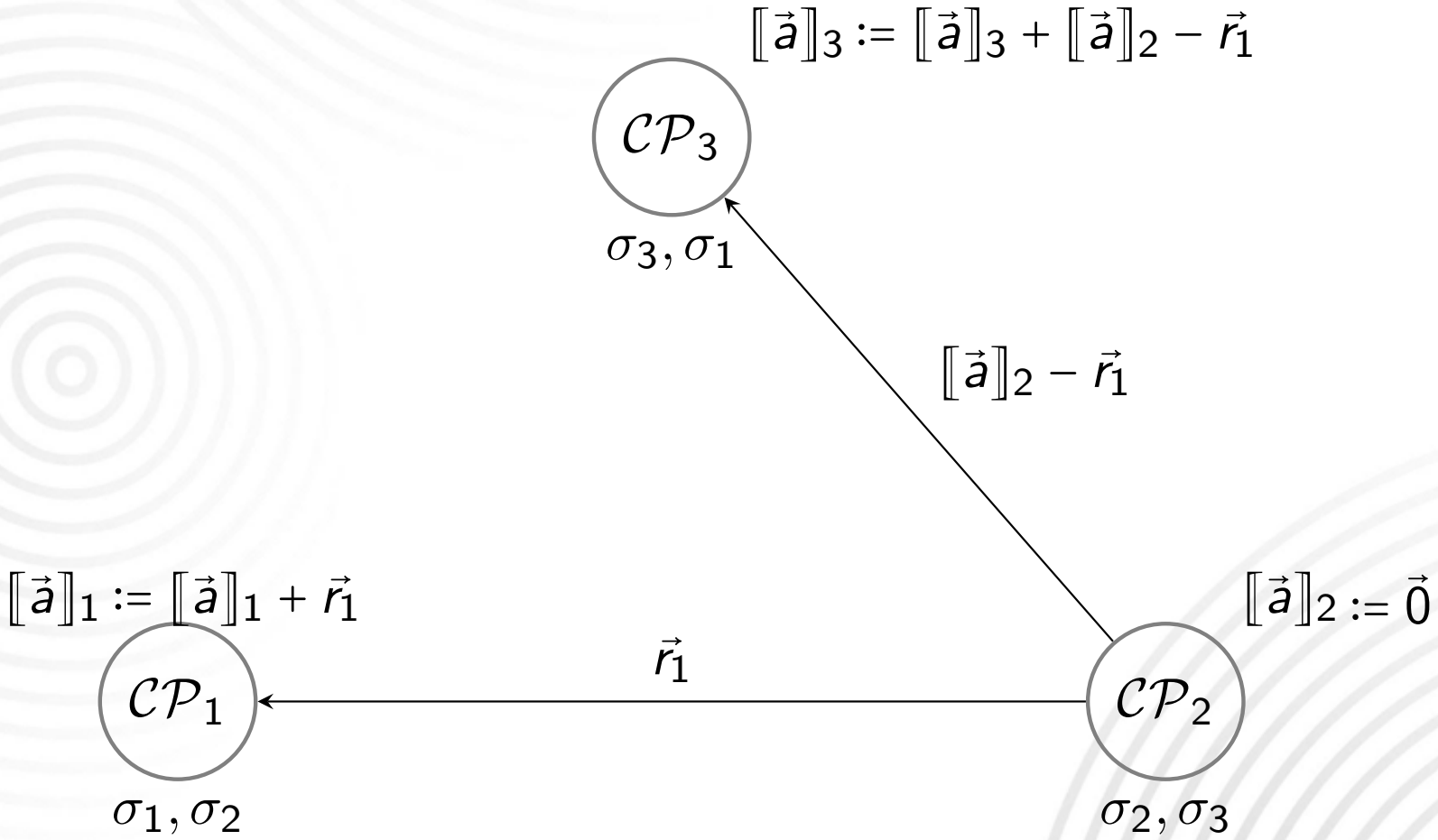
[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

Shuffling protocol



[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

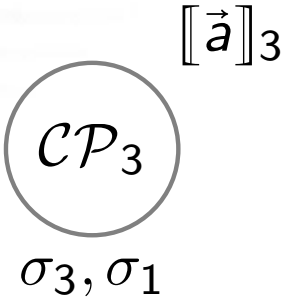
Shuffling protocol



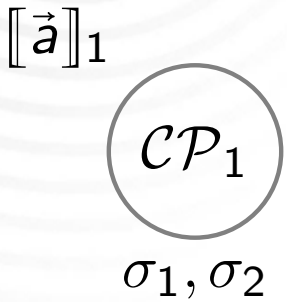
[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

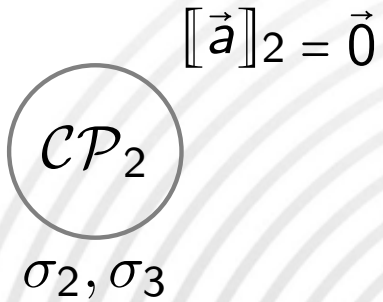
CYBERNETICA

Shuffling protocol



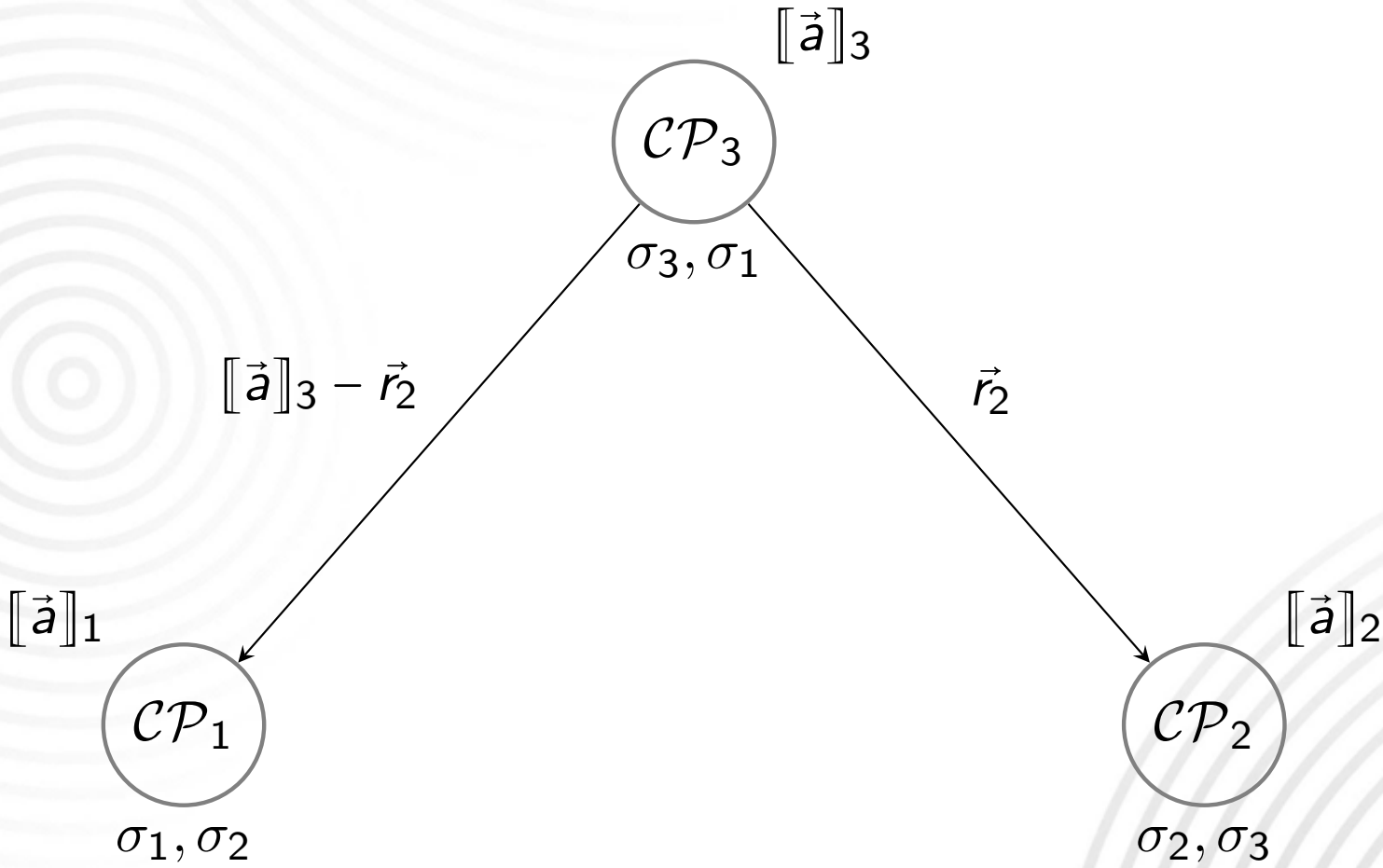
Party $\mathcal{CP}_i$ shuffles $[\![\vec{a}]\!]_i$ using σ_1





[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

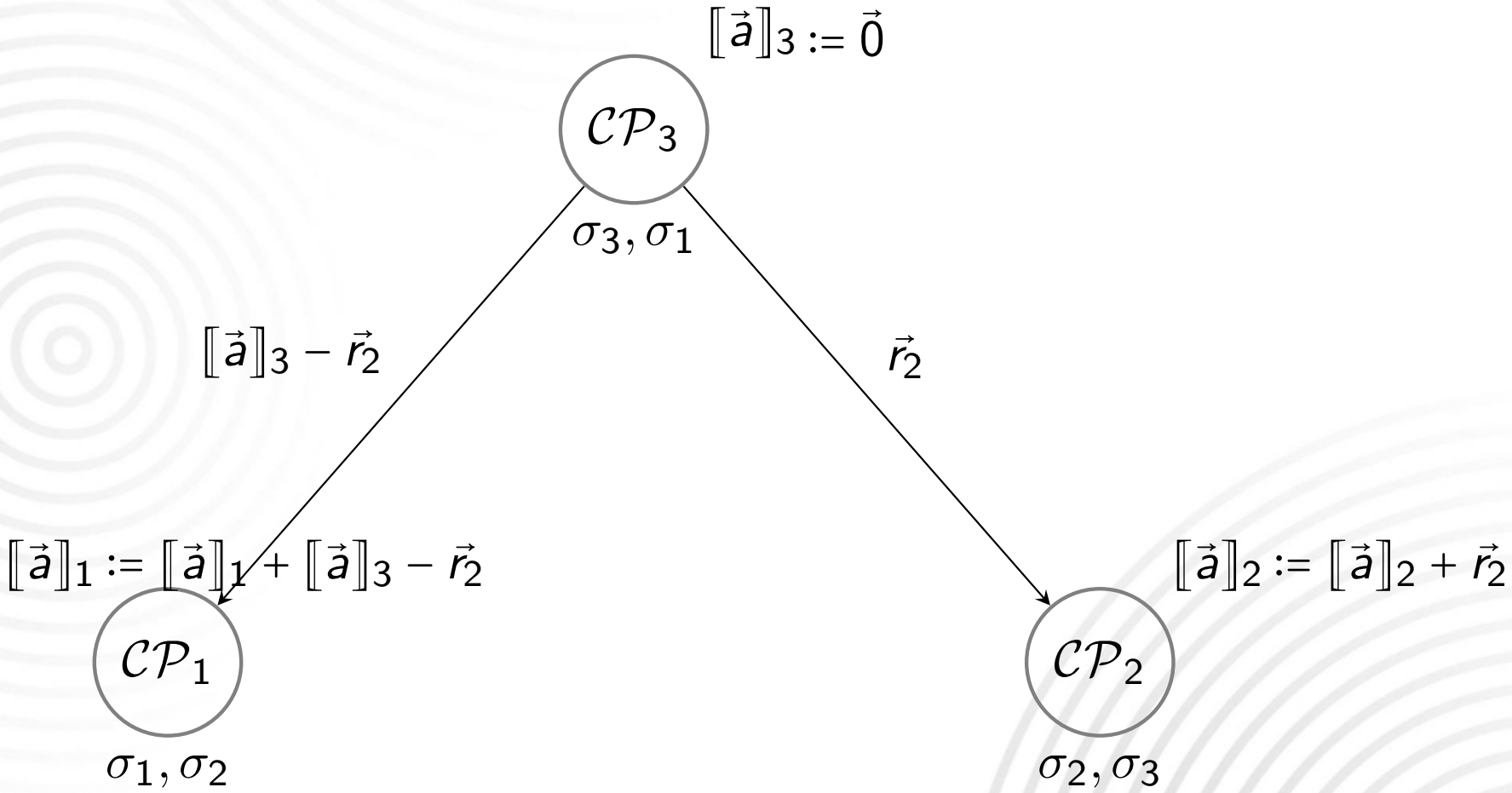
Shuffling protocol



[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

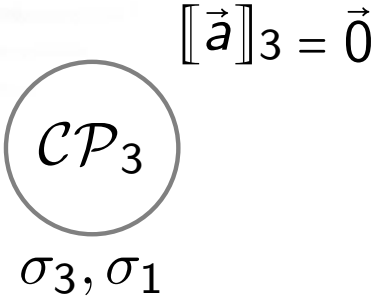
CYBERNETICA

Shuffling protocol

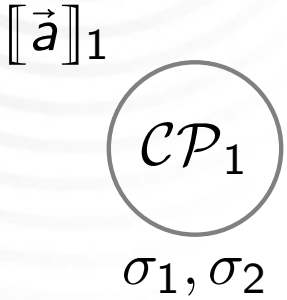


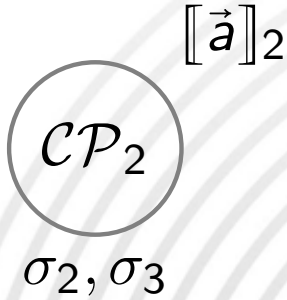
[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

Shuffling protocol



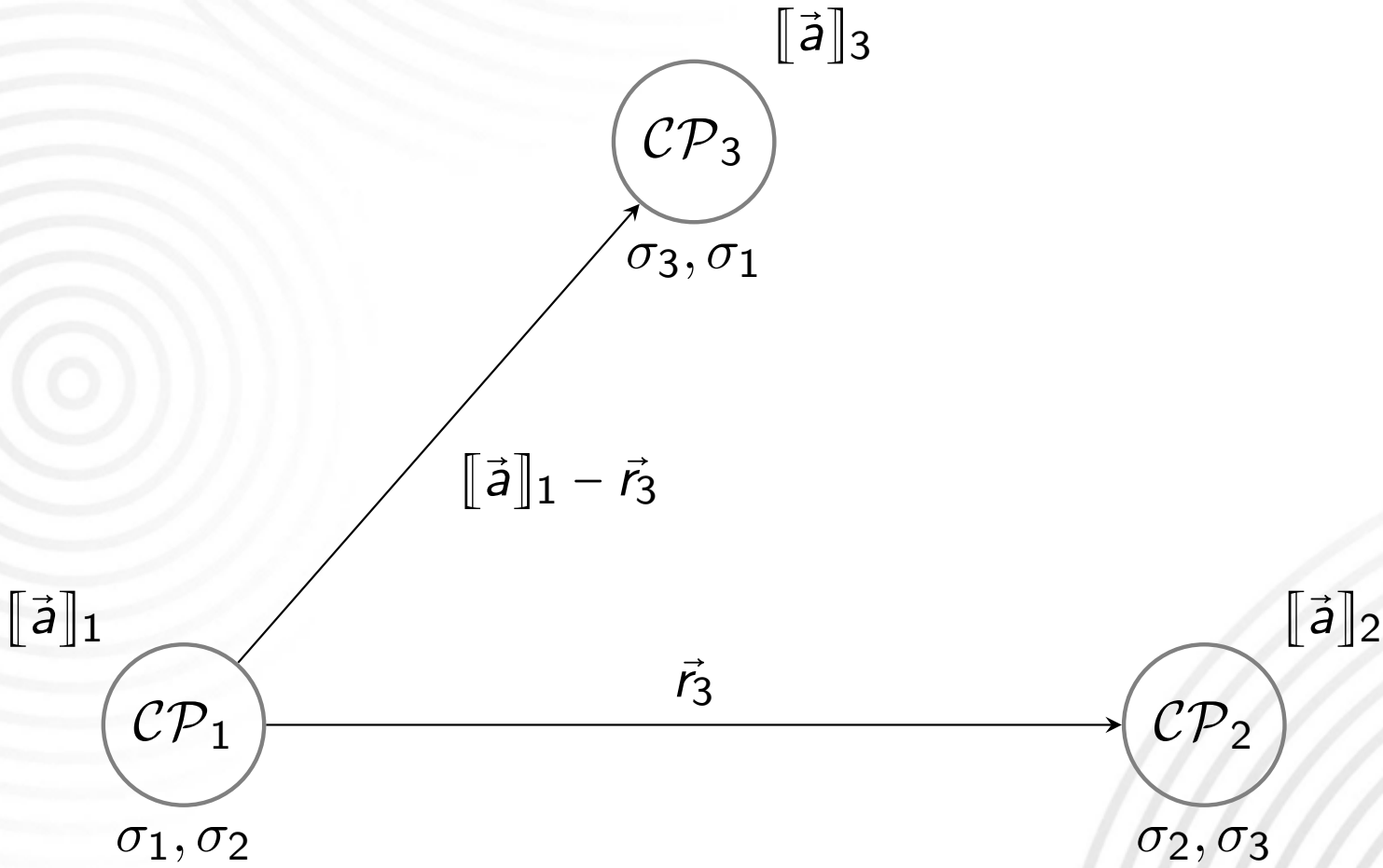
Party $\mathcal{CP}_i$ shuffles $[\![\vec{a}]\!]_i$ using σ_2





[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

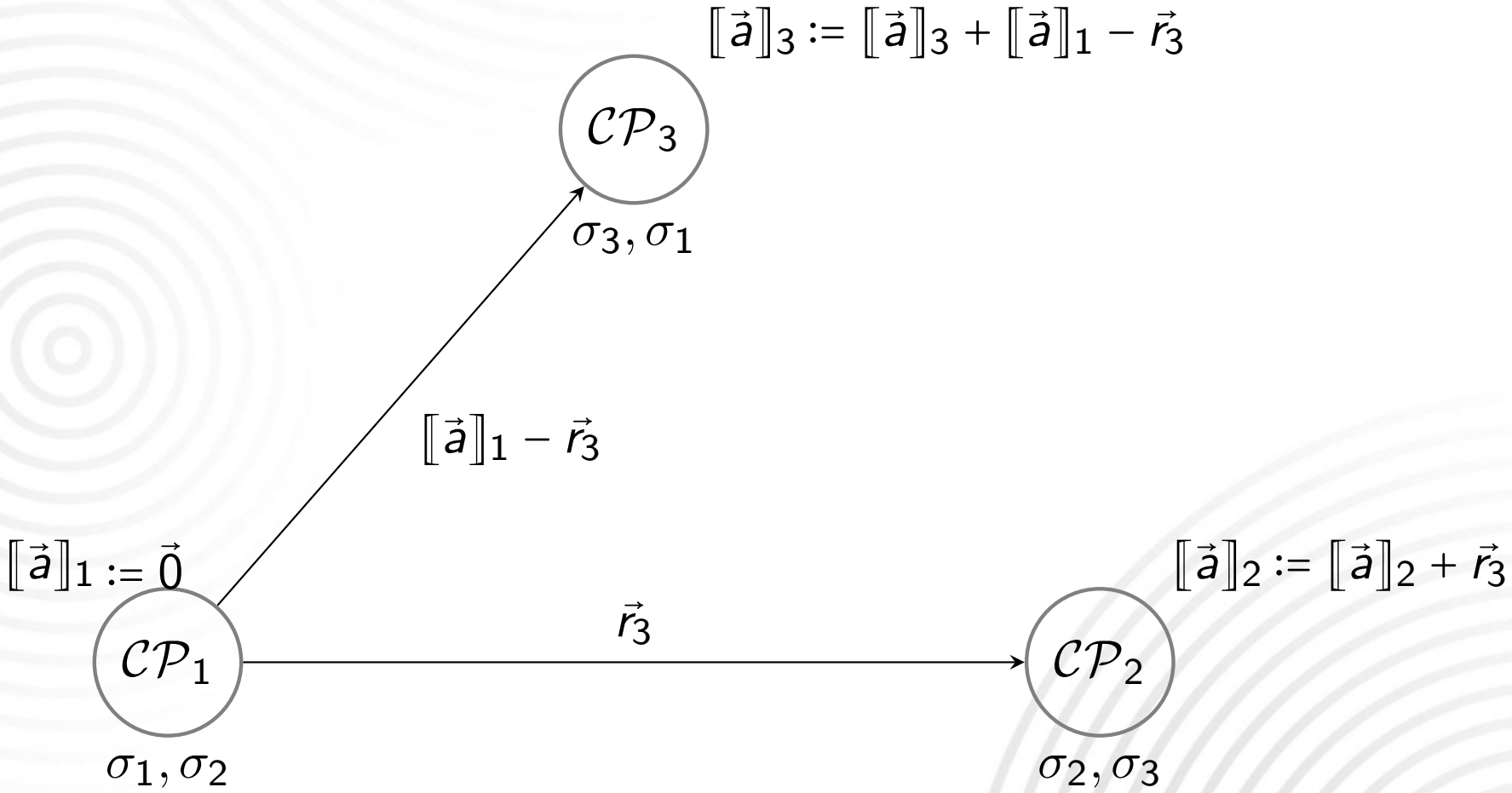
Shuffling protocol



[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

CYBERNETICA

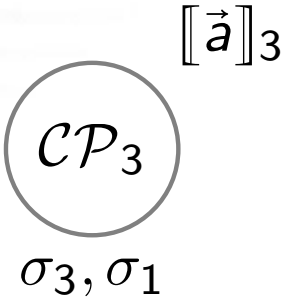
Shuffling protocol



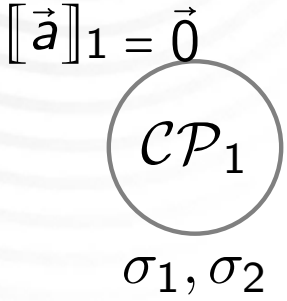
[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

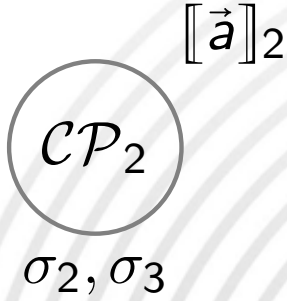
CYBERNETICA

Shuffling protocol



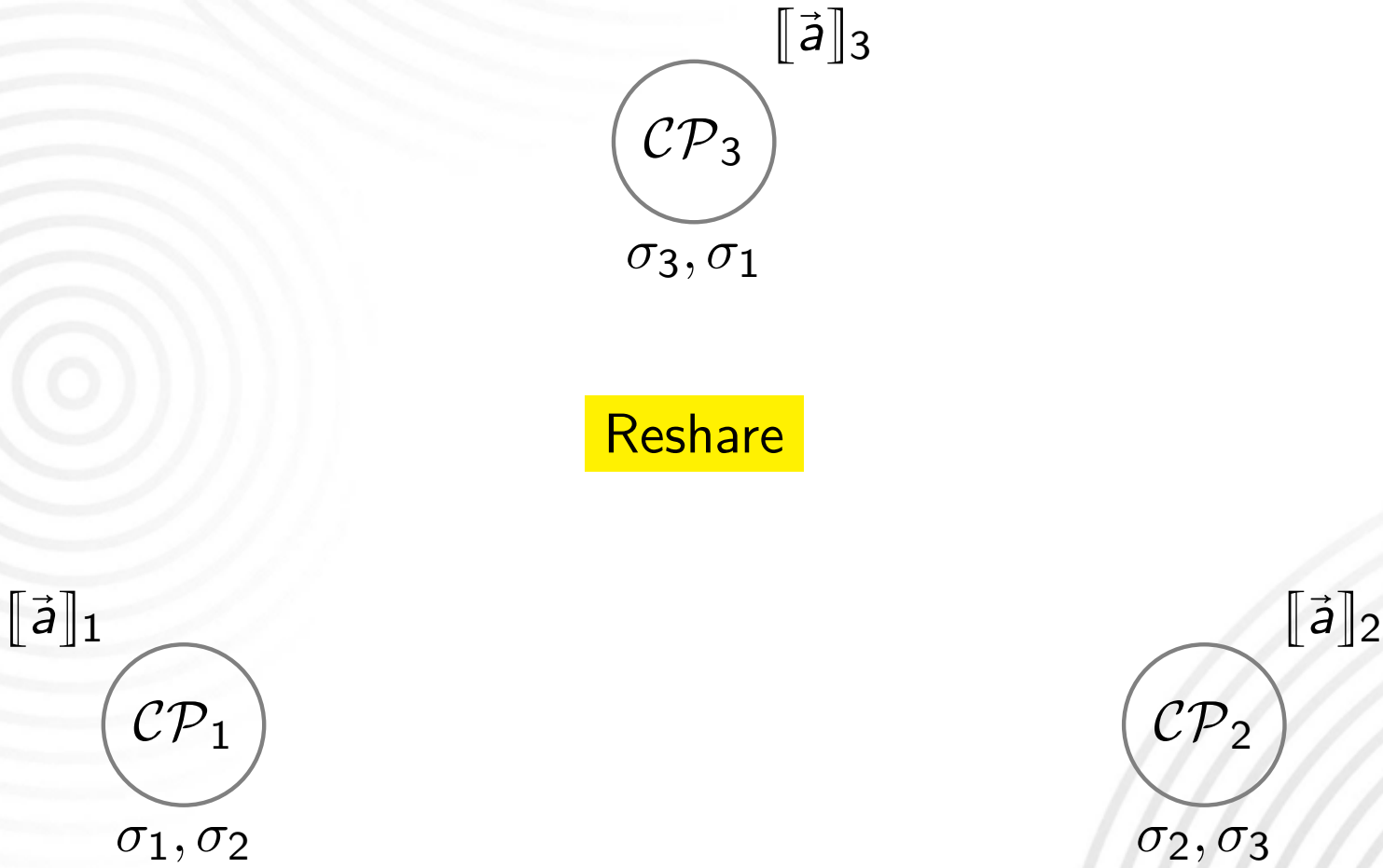
Party $\mathcal{CP}_i$ shuffles $[\![\vec{a}]\!]_i$ using σ_3





[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

Shuffling protocol



[S. Laur, J. Willemson, B. Zhang. Round-Efficient Oblivious Database Manipulation. Information Security Conference, ISC 2011]

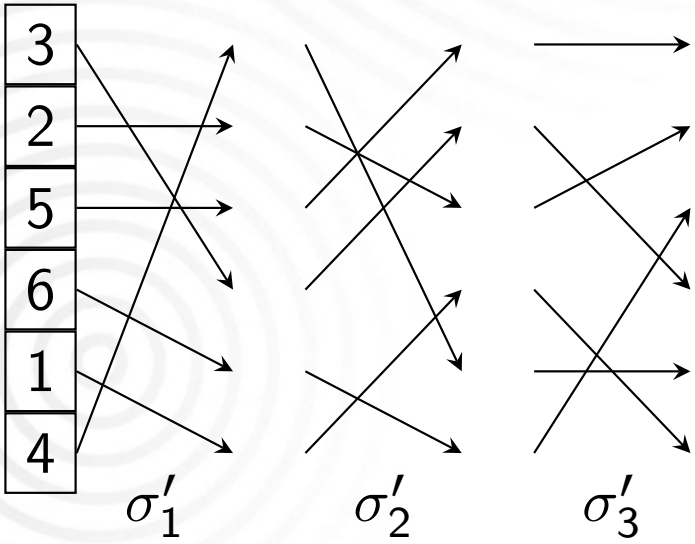
Use for sorting

- Computing parties can generate a sharing $\langle\sigma\rangle$ of a random σ.
  - $\mathcal{CP}_i$ constructs a random $\sigma_i \in S_n$ and sends it to $\mathcal{CP}_{i-1}$.
- After randomly shuffling an array, the comparison results between its elements may be made public
  - If all elements of the array are different
- After shuffling, we can use any sorting method to sort a private array.
  - No need to use data-oblivious methods, e.g. sorting networks

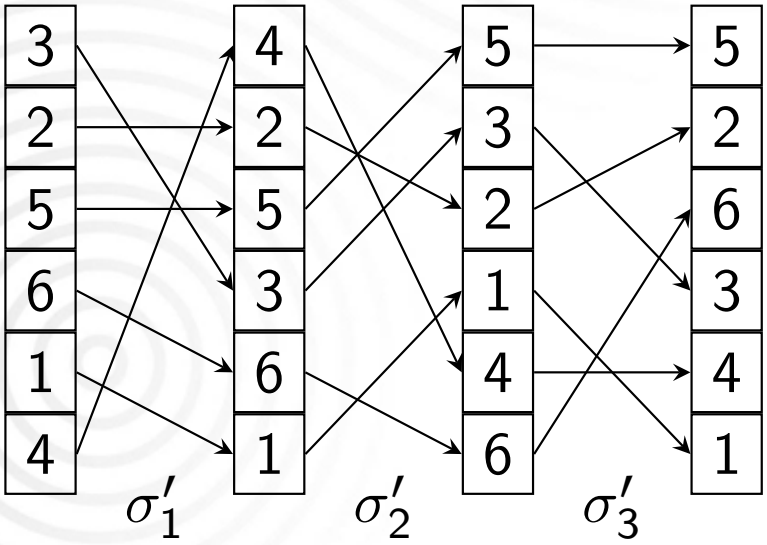
[K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, K. Takahashi. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. Int. Conf on Inf. Security and Cryptology, ICISC 2012]

Remembering the sorting permutation

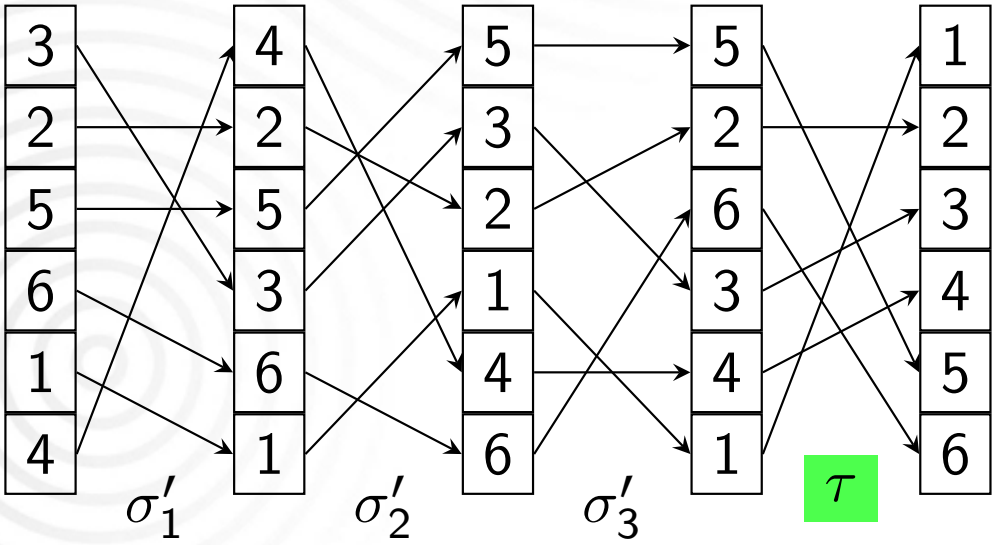
Remembering the sorting permutation



Remembering the sorting permutation

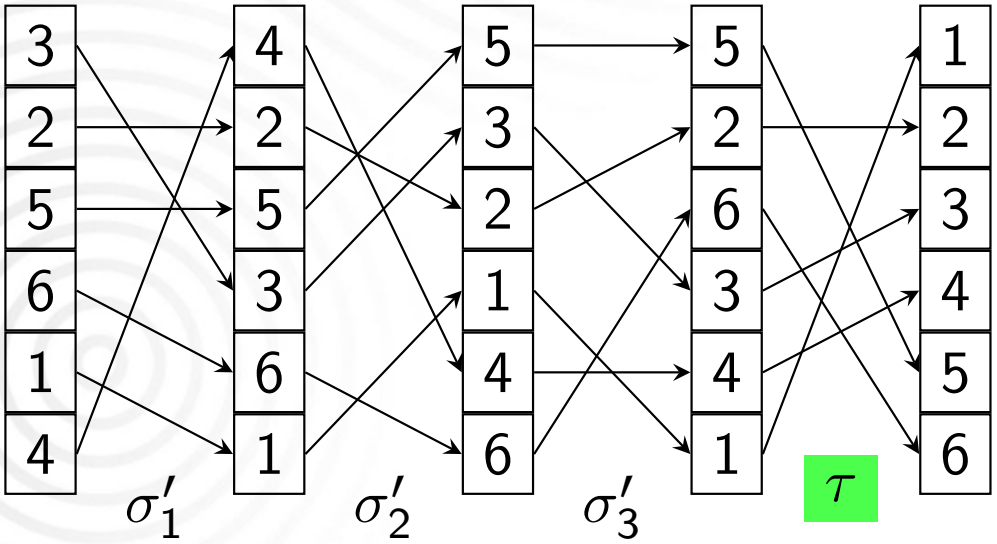


Remembering the sorting permutation



CYBERNETICA

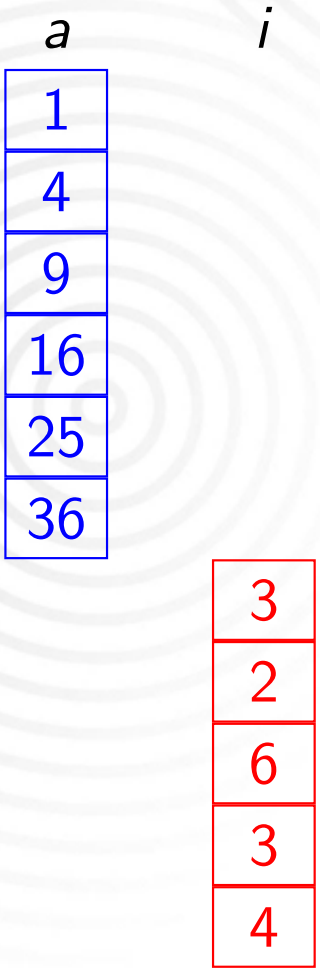
Remembering the sorting permutation



- $\sigma_1 := \sigma'_1;$ $\sigma_2 := \sigma'_2;$ $\sigma_3 := \sigma'_3 \circ \tau$
- σ'_i is generated by those $\mathcal{CP}_{j_1}$ and $\mathcal{CP}_{j_2}$ that are supposed to know σ_i afterwards

Parallel oblivious reading

Parallel oblivious reading



Parallel oblivious reading

a	w	i
1	1	3
4	3	2
9	5	6
16	7	3
25	9	4
36	11	

$$w = \text{prefixsum}^{-1}(a)$$

Parallel oblivious reading

a	w	i
1	1	1
4	3	2
9	5	3
16	7	4
25	9	5
36	11	6
		3
		2
		6
		3
		4

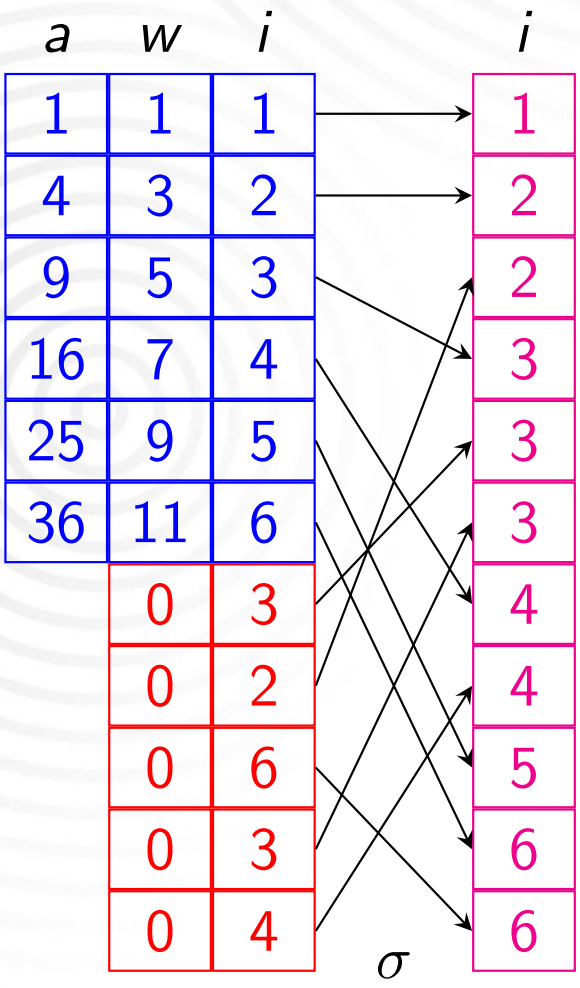
$$w = \text{prefixsum}^{-1}(a)$$

Parallel oblivious reading

a	w	i
1	1	1
4	3	2
9	5	3
16	7	4
25	9	5
36	11	6
	0	3
	0	2
	0	6
	0	3
	0	4

$$w = \text{prefixsum}^{-1}(a)$$

Parallel oblivious reading

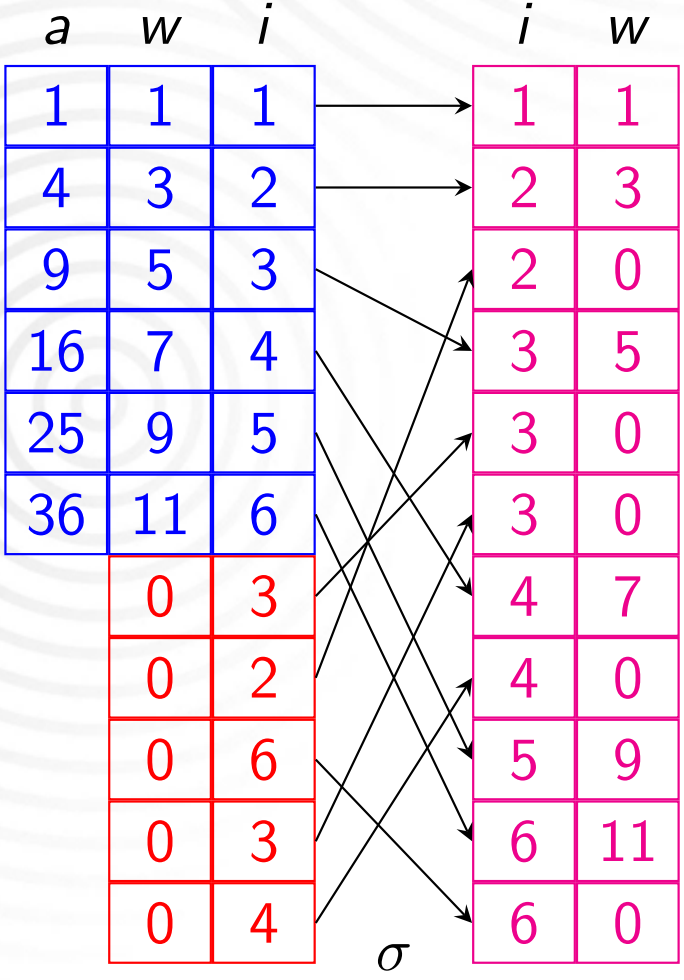


$$w = \text{prefixsum}^{-1}(a)$$

$$\sigma = \text{sort}(i)$$

CYBERNETICA

Parallel oblivious reading



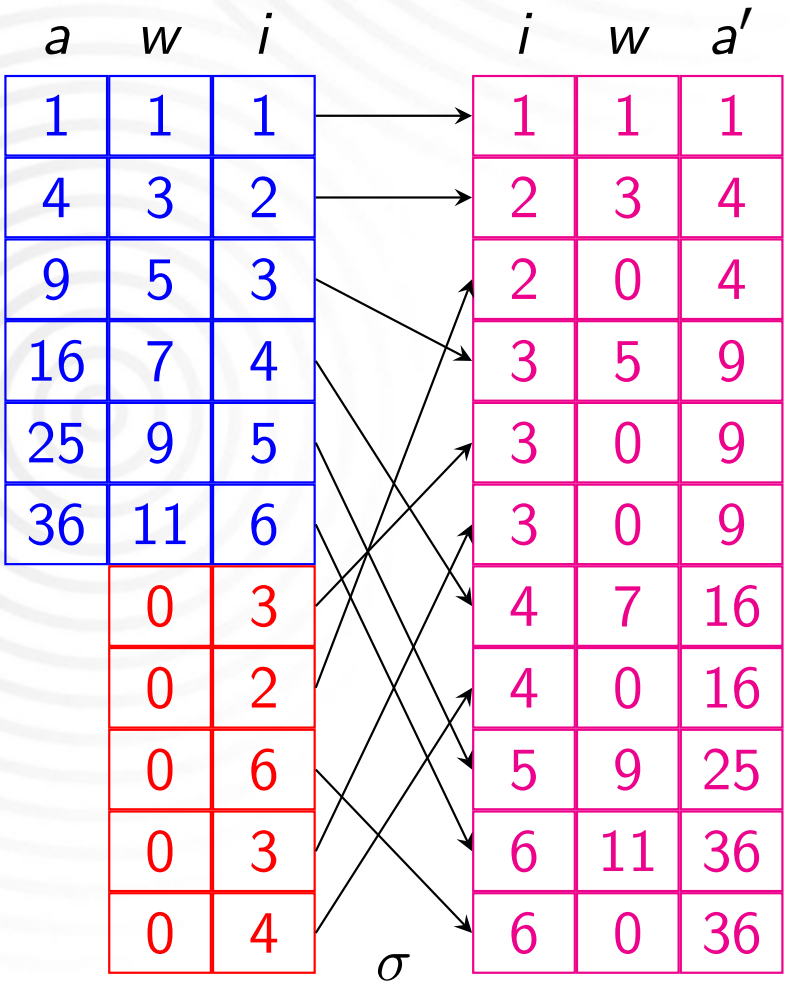
$$w = \text{prefixsum}^{-1}(a)$$

$$\sigma = \text{sort}(i)$$

apply σ to w

CYBERNETICA

Parallel oblivious reading



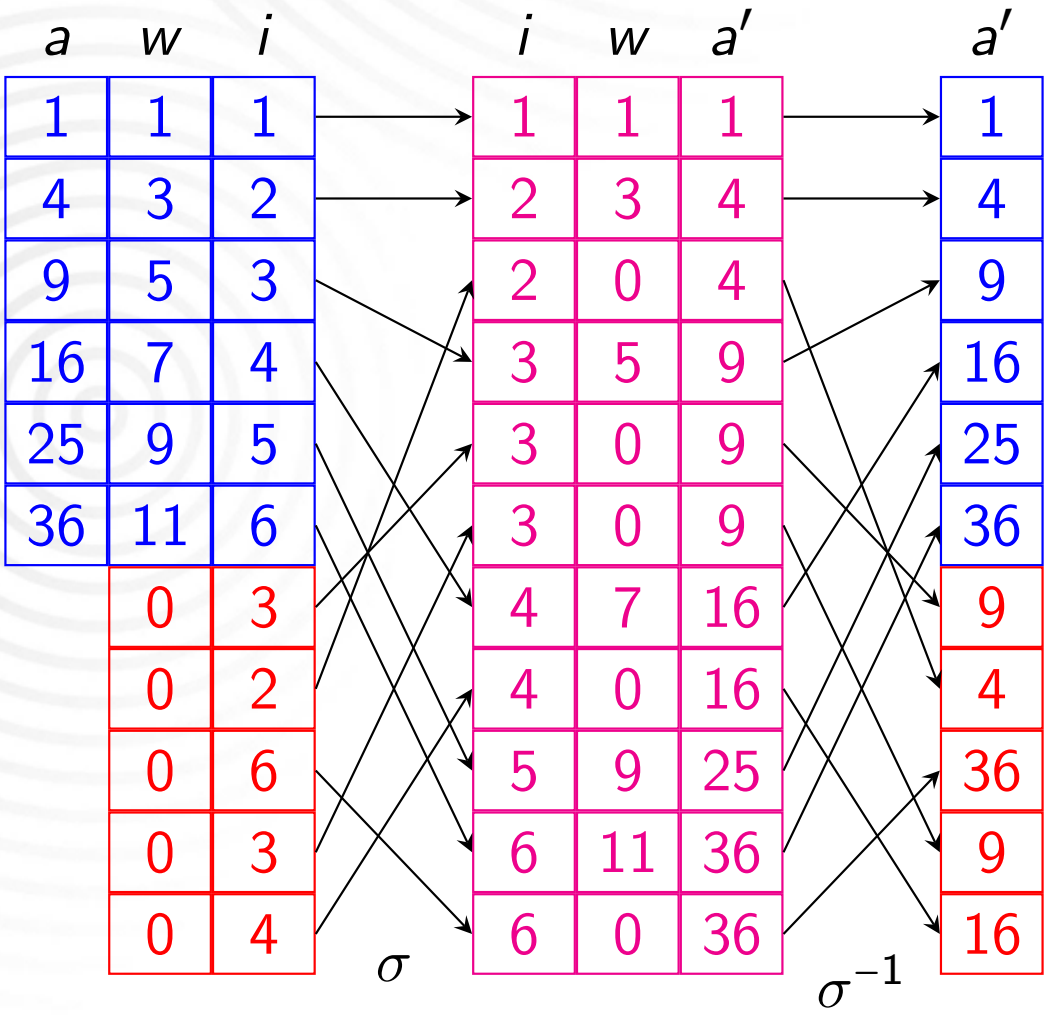
$$w = \text{prefixsum}^{-1}(a)$$

$$\sigma = \text{sort}(i)$$

apply σ to w

$$a' = \text{prefixsum}(w)$$

Parallel oblivious reading



$w = \text{prefixsum}^{-1}(a)$

$\sigma = \text{sort}(i)$

apply σ to w

$a' = \text{prefixsum}(w)$

apply σ^{-1} to a'

Parallel oblivious writing

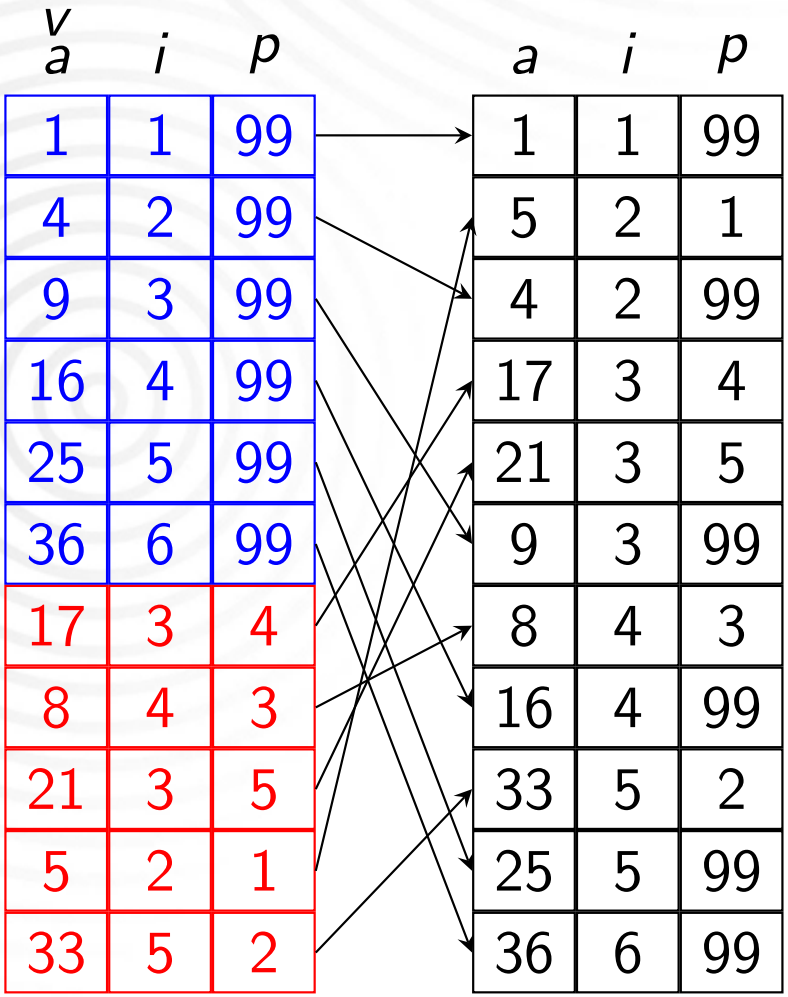
Parallel oblivious writing

v_a	i	p
1		
4		
9		
16		
25		
36		
17	3	4
8	4	3
21	3	5
5	2	1
33	5	2

Parallel oblivious writing

v_a	i	p
1	1	99
4	2	99
9	3	99
16	4	99
25	5	99
36	6	99
17	3	4
8	4	3
21	3	5
5	2	1
33	5	2

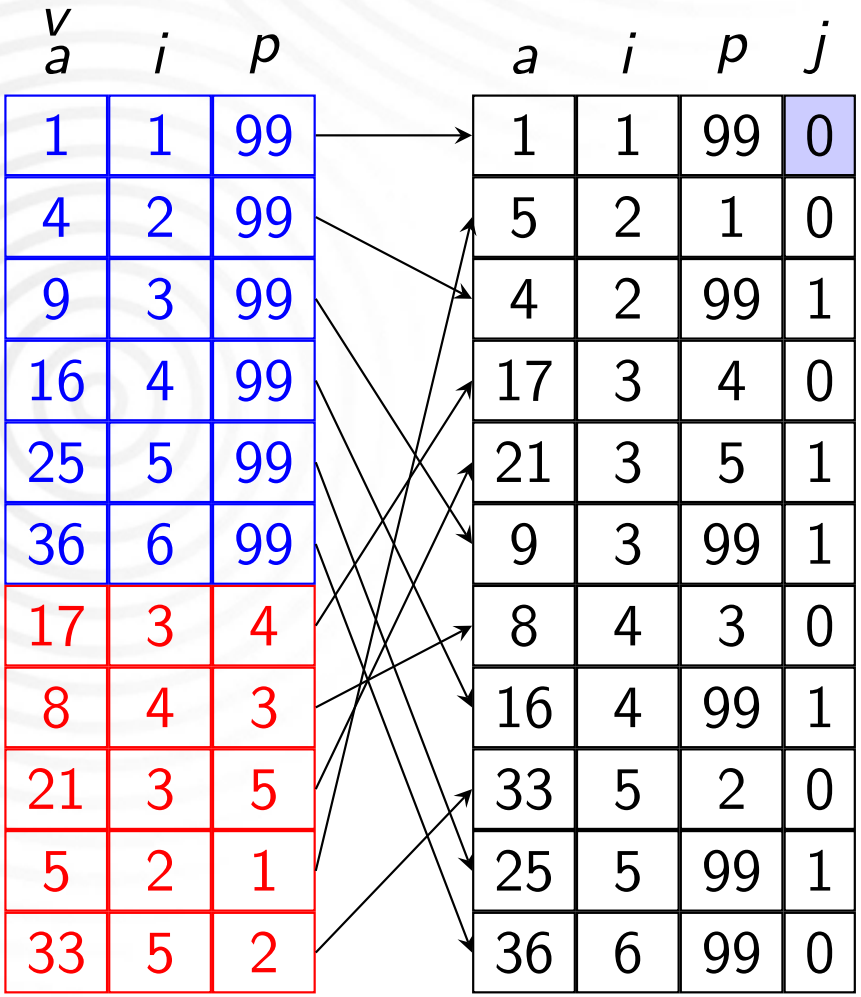
Parallel oblivious writing



sort by $i; p$

CYBERNETICA

Parallel oblivious writing



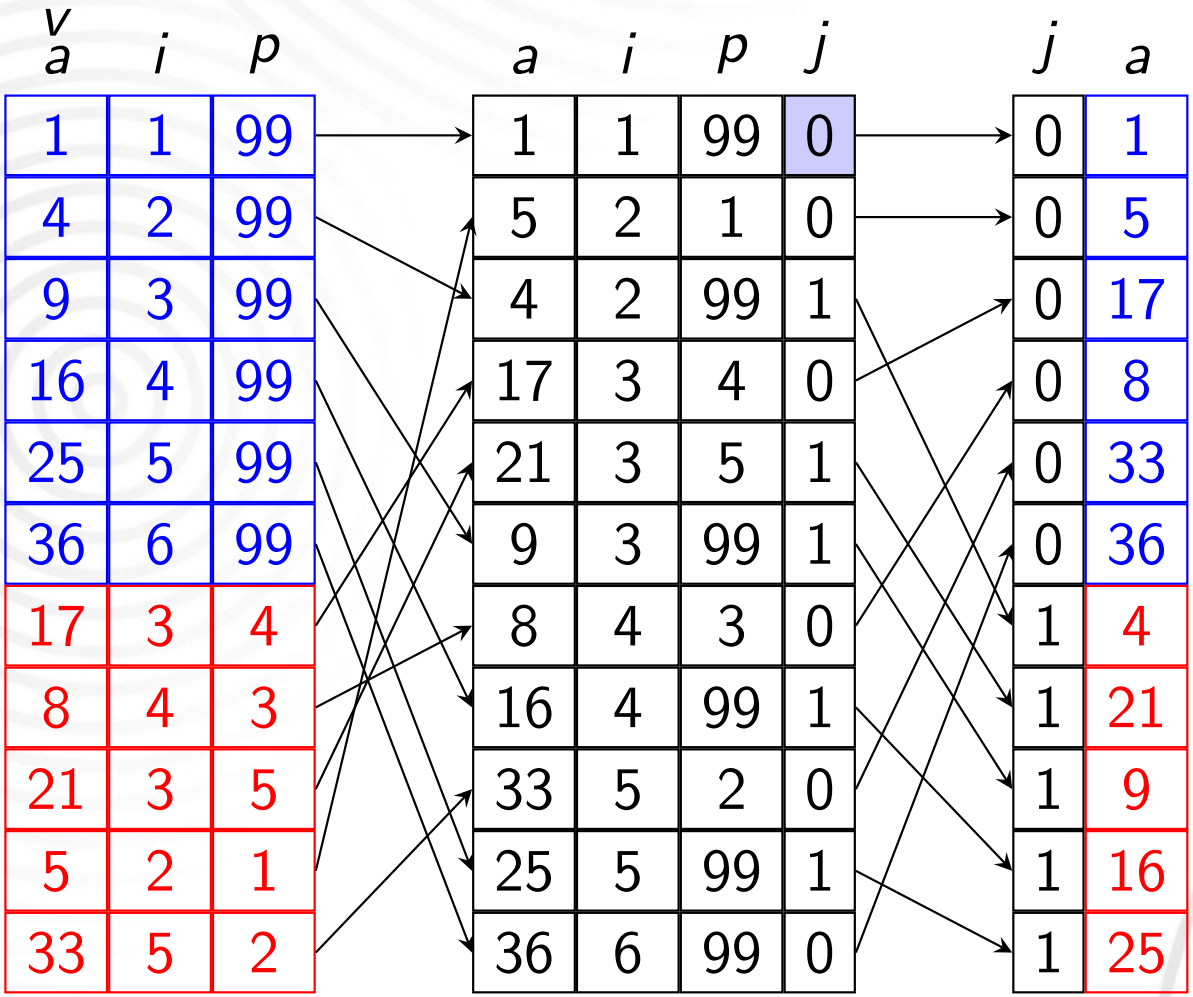
$v.a$	i	p
1	1	99
4	2	99
9	3	99
16	4	99
25	5	99
36	6	99
17	3	4
8	4	3
21	3	5
5	2	1
33	5	2

a	i	p	j
1	1	99	0
5	2	1	0
4	2	99	1
17	3	4	0
21	3	5	1
9	3	99	1
8	4	3	0
16	4	99	1
33	5	2	0
25	5	99	1
36	6	99	0

sort by $i; p$

$j_n = (i_n \stackrel{?}{=} i_{n-1})$

Parallel oblivious writing



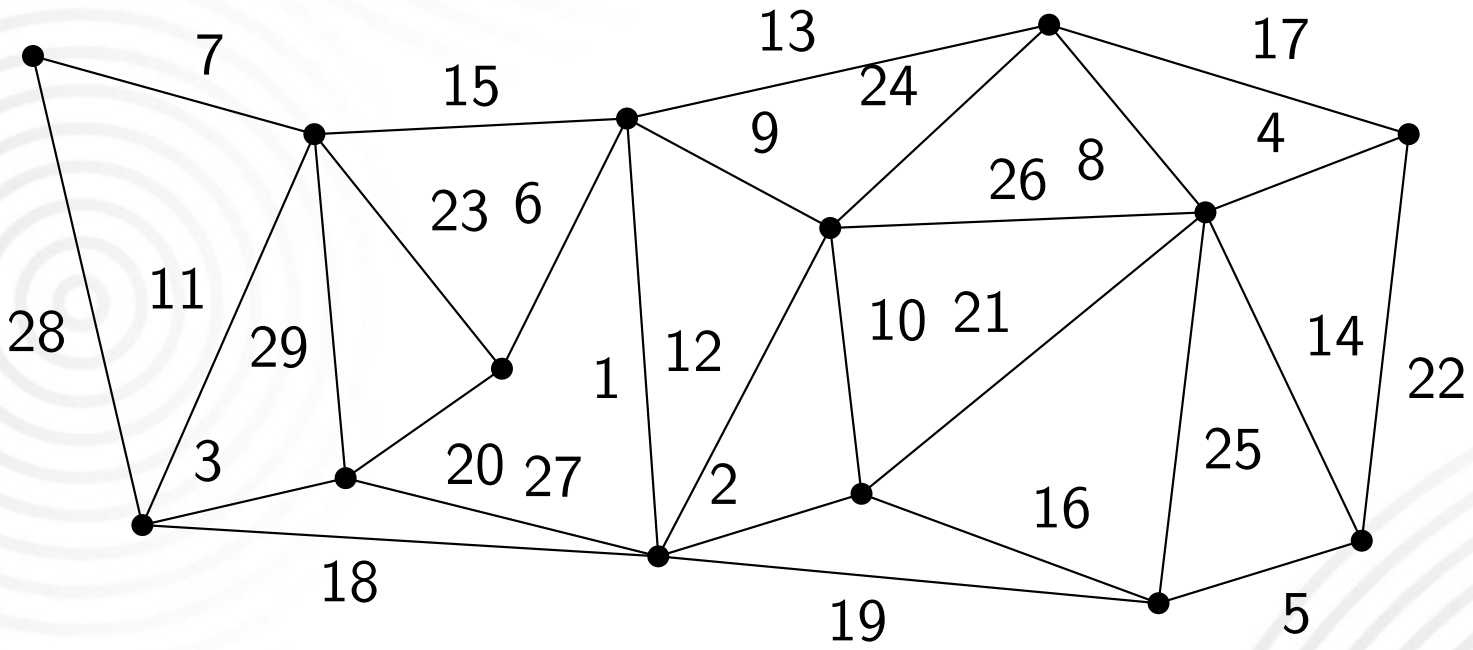
sort by $i; p$

$j_n = (i_n \stackrel{?}{=} i_{n-1})$

sort by j

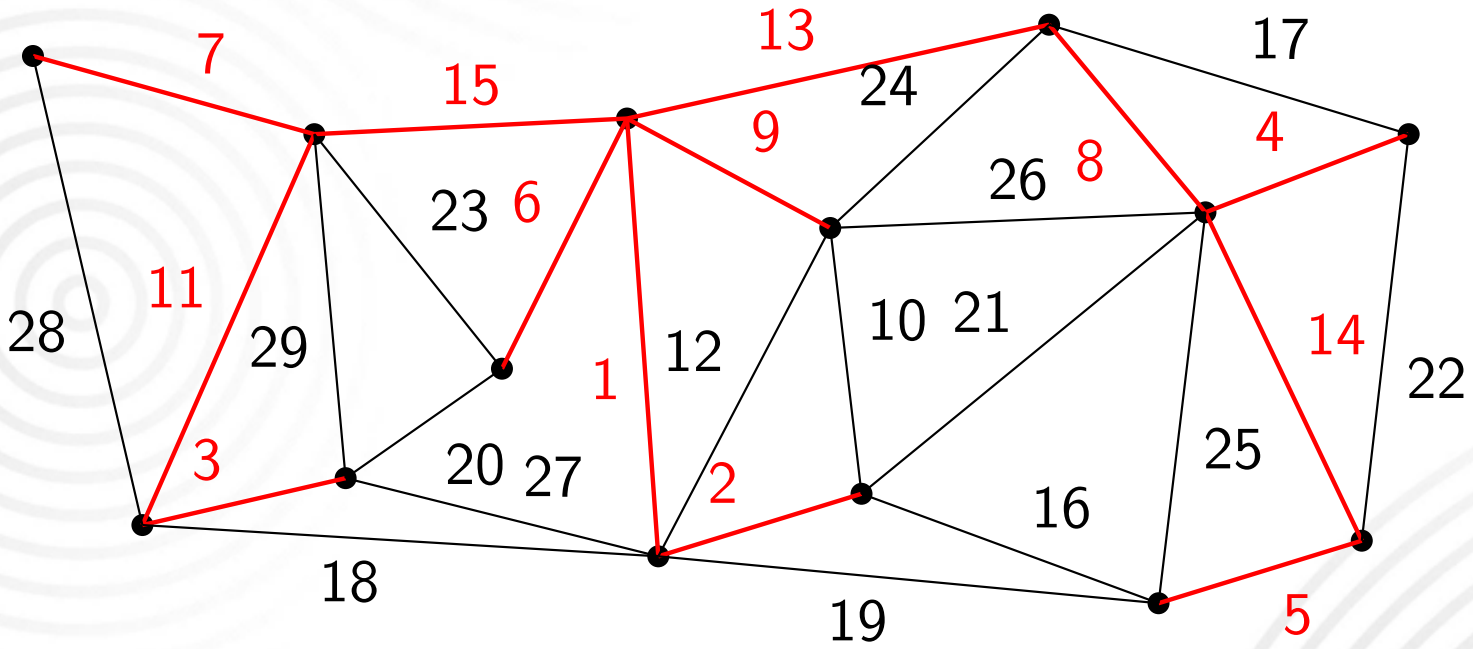
CYBERNETICA

Minimum spanning tree



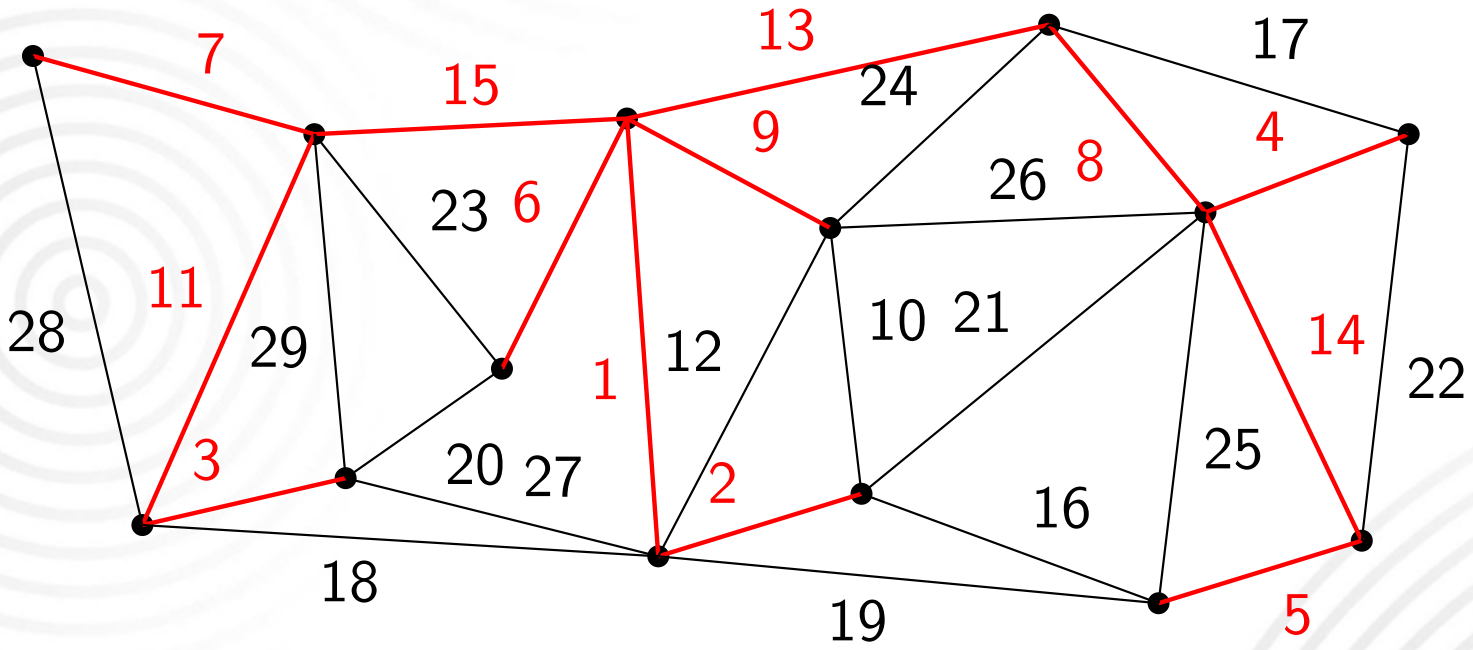
CYBERNETICA

Minimum spanning tree



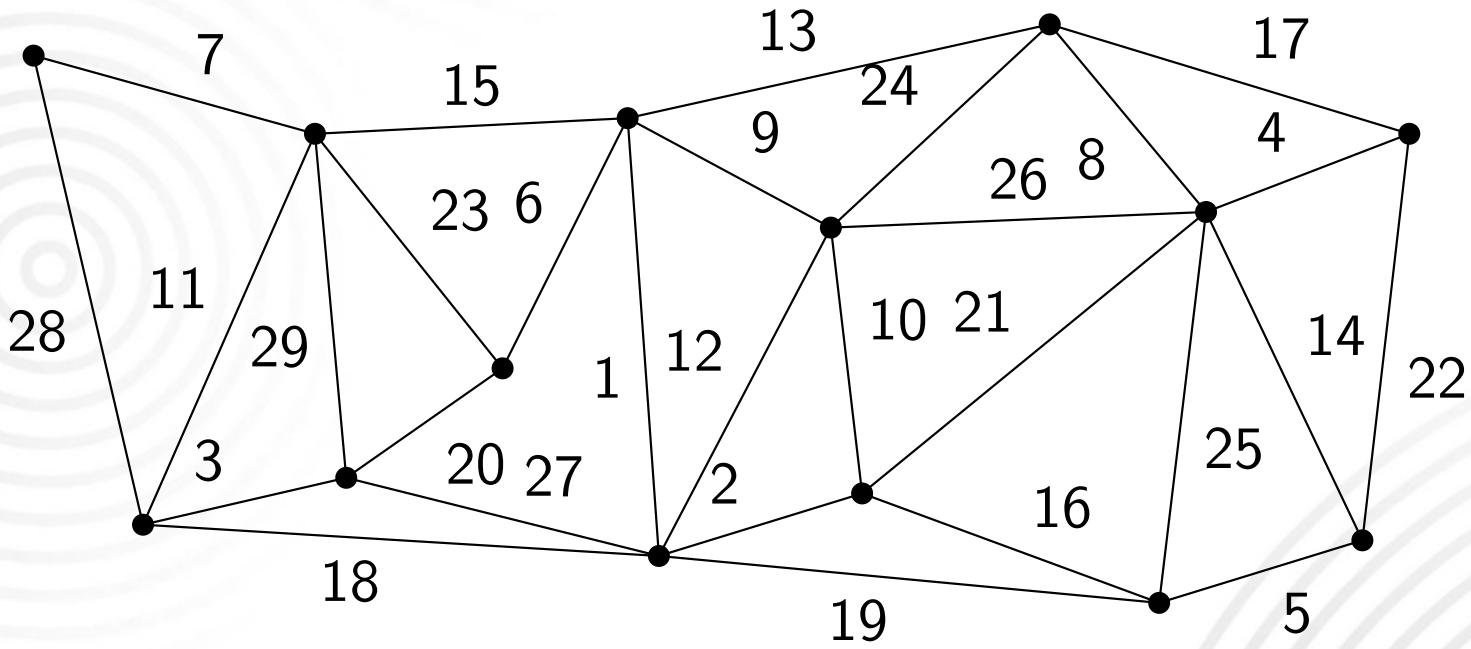
CYBERNETICA

Minimum spanning tree



Kruskal's and Prim's algorithms — inherently sequential

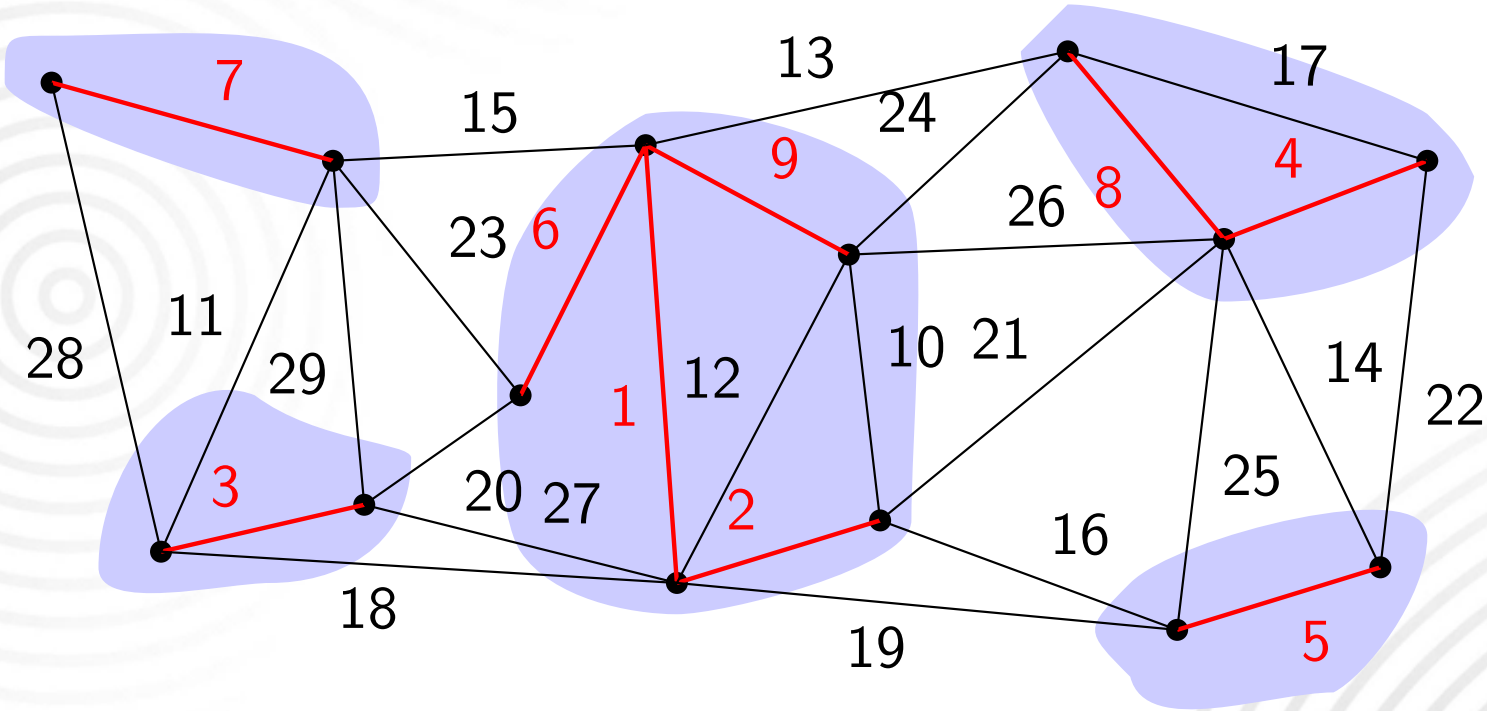
Borůvka's algorithm



http://en.wikipedia.org/wiki/Borůvka's_algorithm

CYBERNETICA

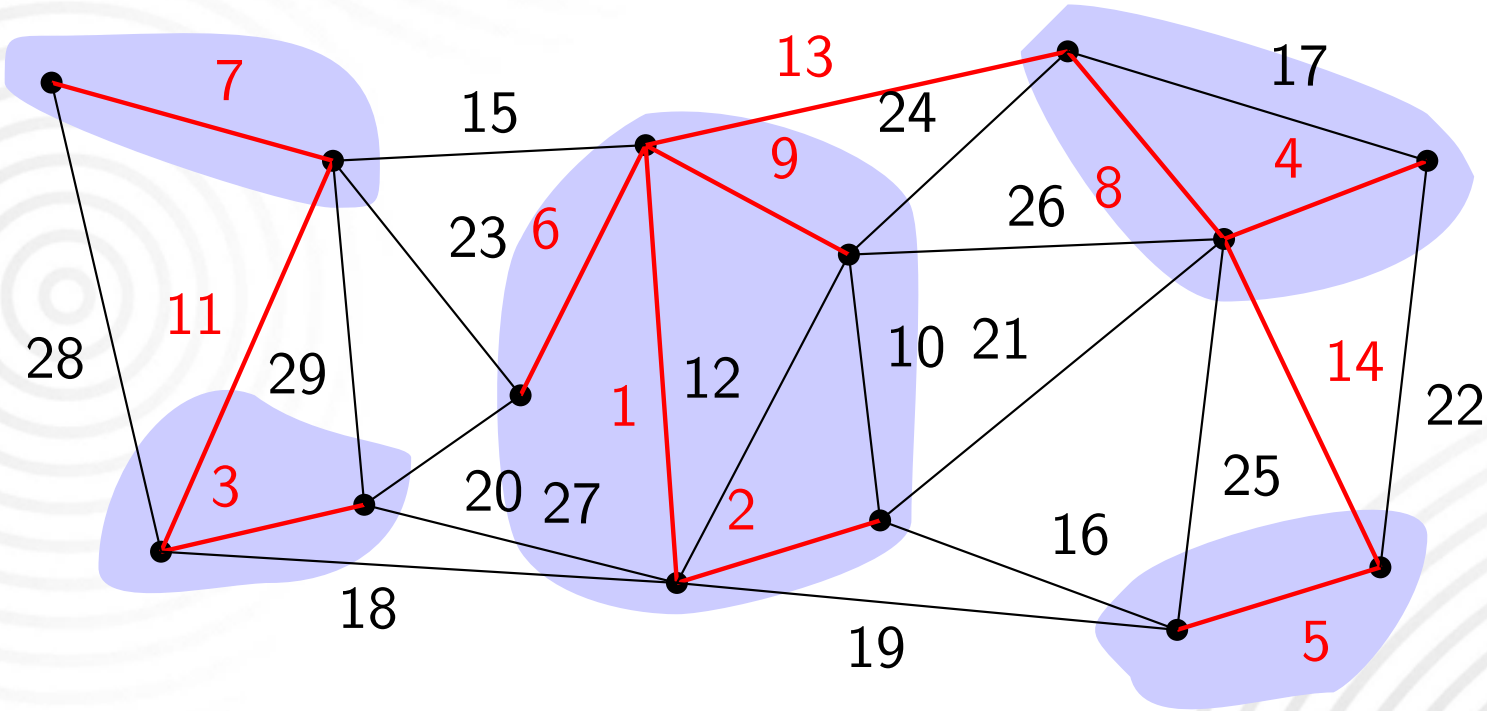
Borůvka's algorithm



http://en.wikipedia.org/wiki/Borůvka's_algorithm

CYBERNETICA

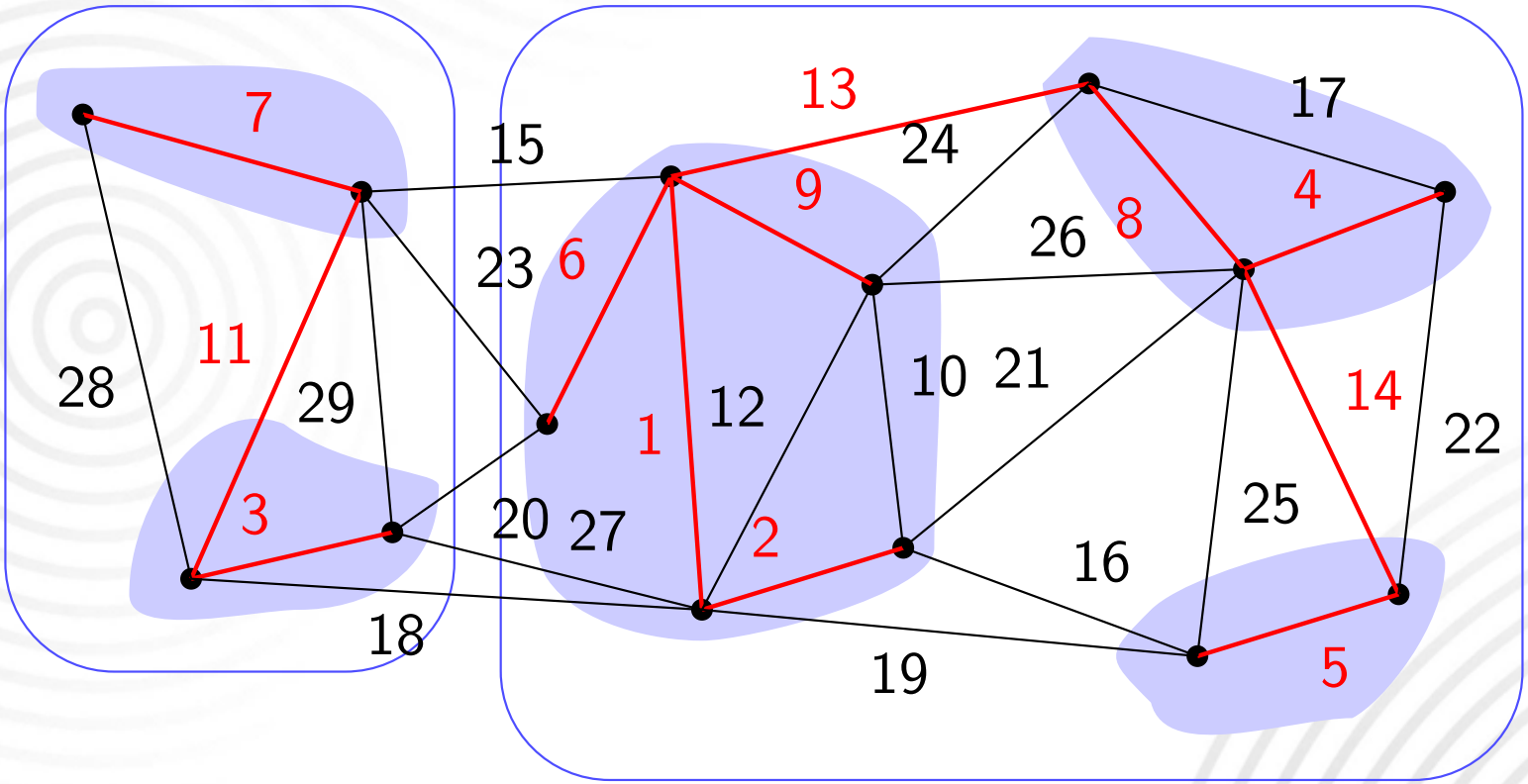
Borůvka's algorithm



http://en.wikipedia.org/wiki/Borůvka's_algorithm

CYBERNETICA

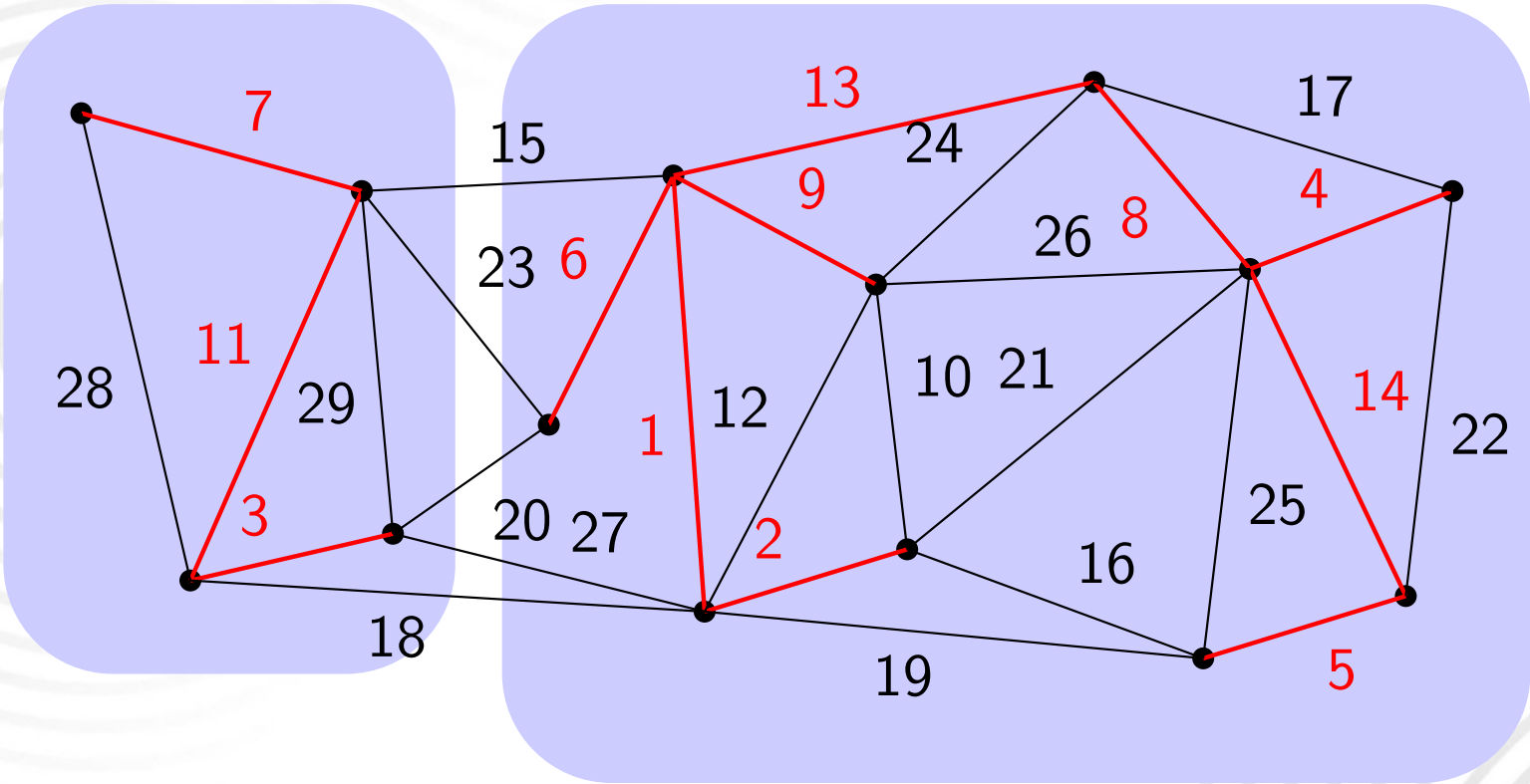
Borůvka's algorithm



http://en.wikipedia.org/wiki/Borůvka's_algorithm

CYBERNETICA

Borůvka's algorithm



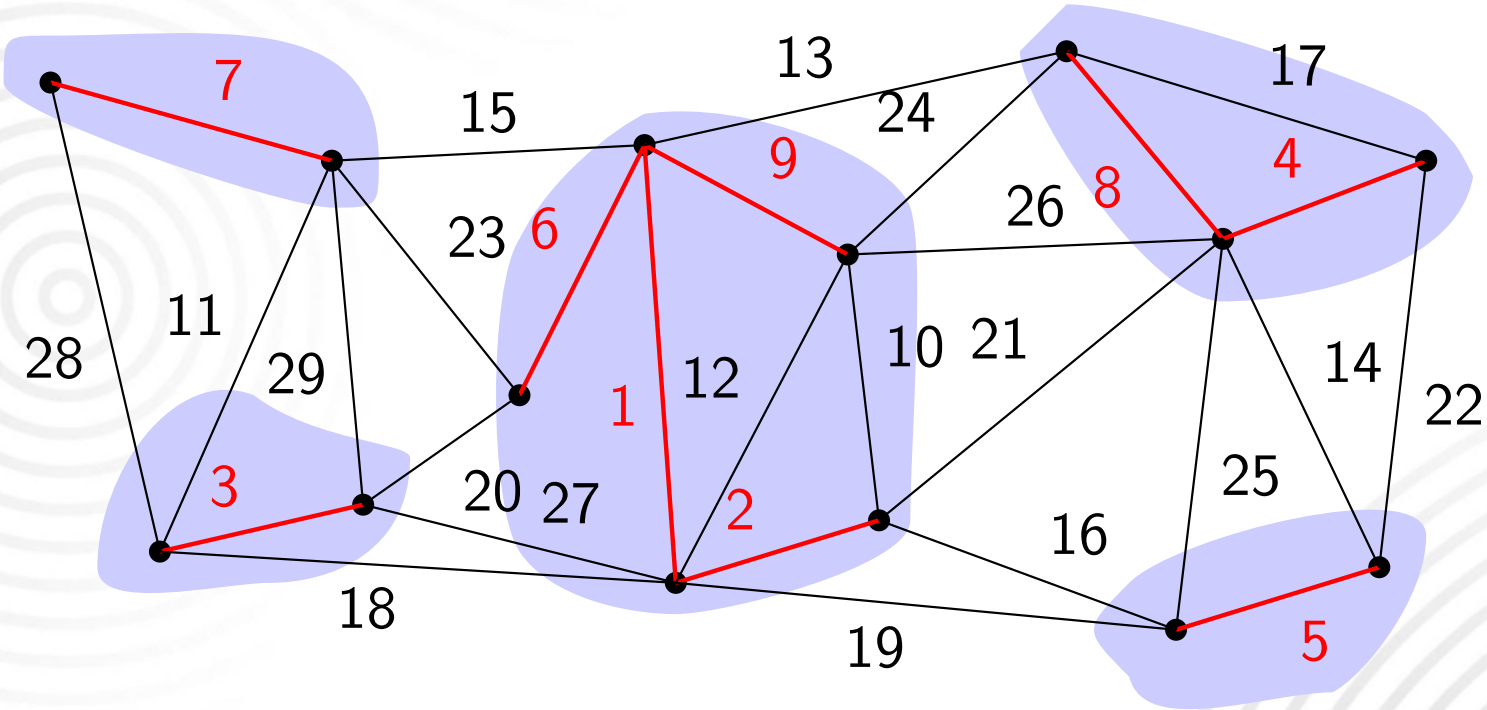
http://en.wikipedia.org/wiki/Borůvka's_algorithm

CYBERNETICA

Awerbuch-Shiloach PRAM algorithm

- Variation of Borůvka's algorithm
- Uses a union-find structure to keep track of parts
  - Each vertex points to its "parent"
  - Independent of graph edges
- Consider only the external edges of stars
  - star — a union-find tree of height ≤ 1
    1. If $parent(v) \neq u = parent(parent(v))$, then v and u are not in a star.
    2. If $parent(v)$ is not in a star, then v also isn't.
- Shortcut the paths in union-find trees
  - grandparent becomes the new parent
- The number of iterations grows from $\log_2 |V|$ to $\log_{3/2} |V|$

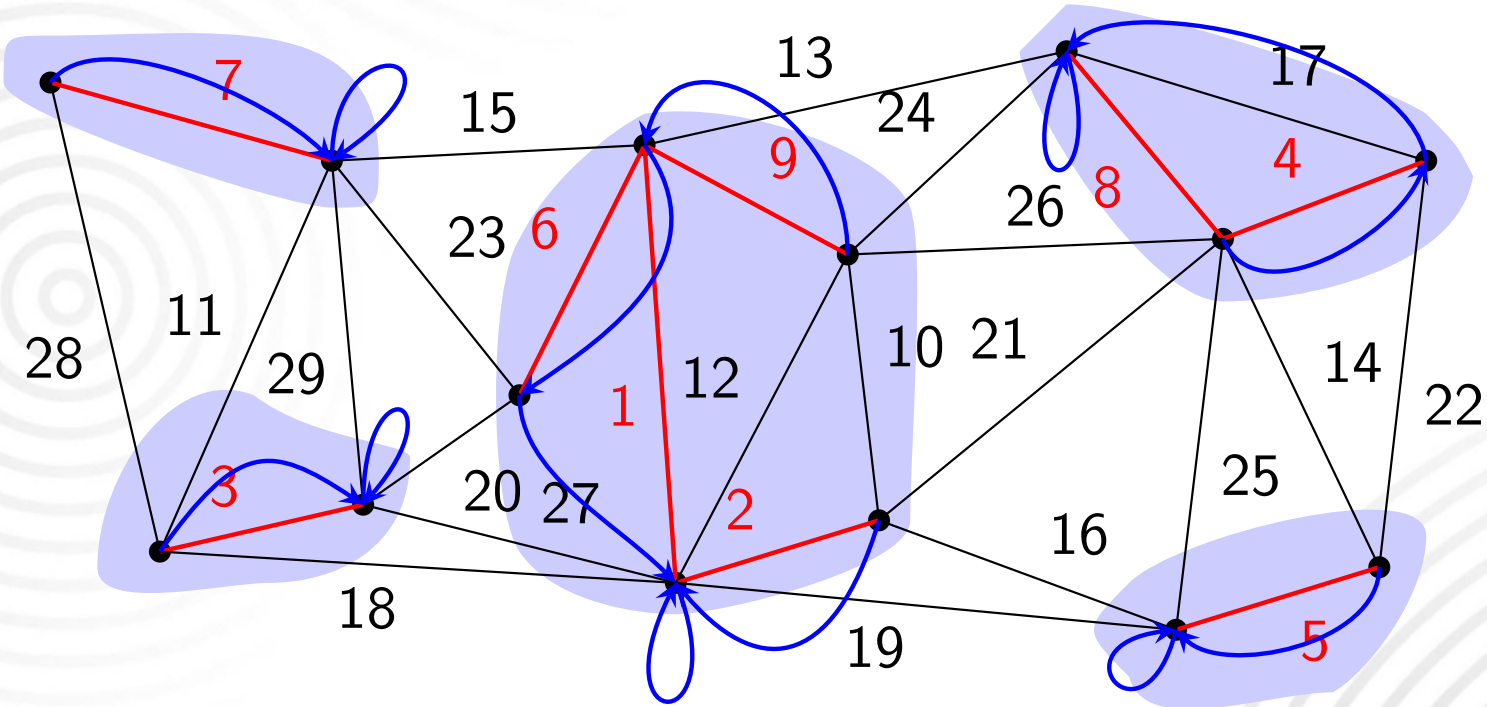
Awerbuch-Shiloach algorithm



[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

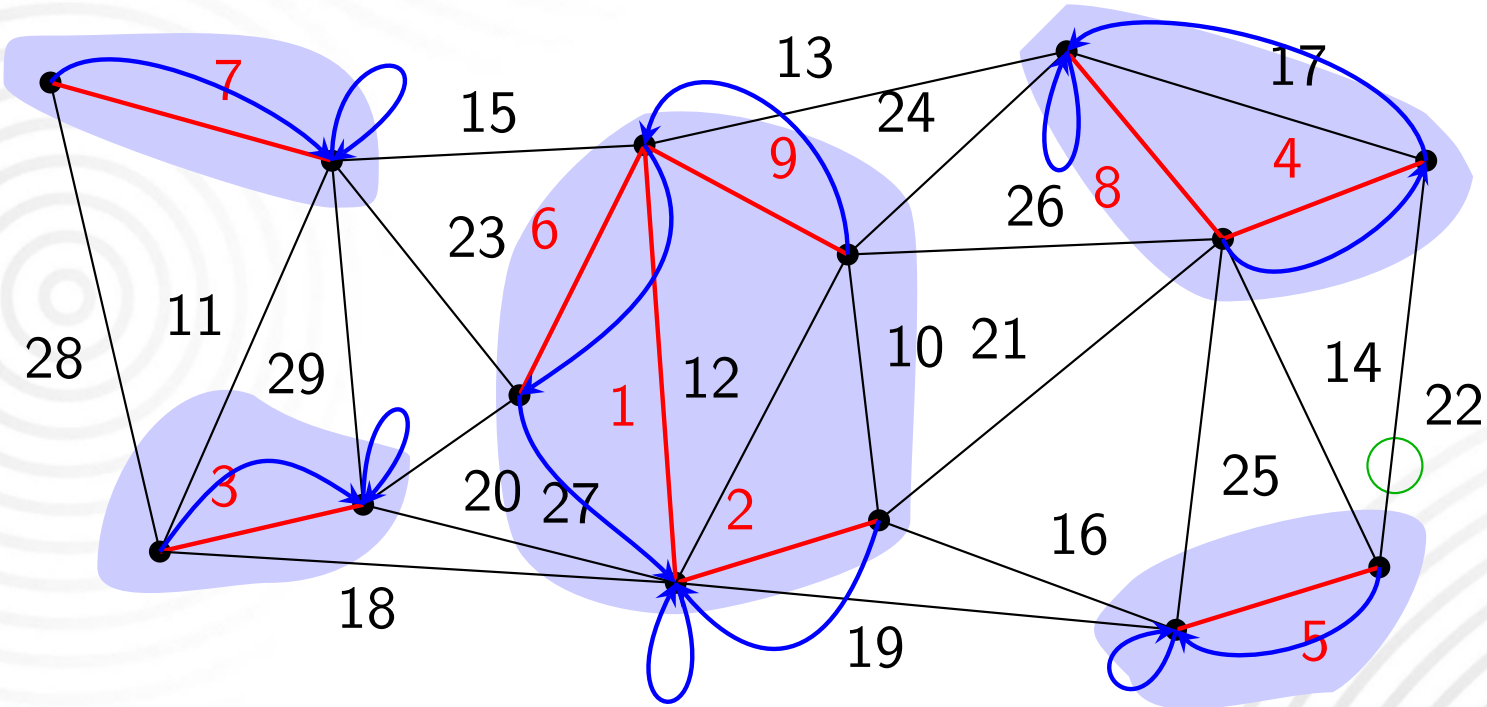
CYBERNETICA

Awerbuch-Shiloach algorithm



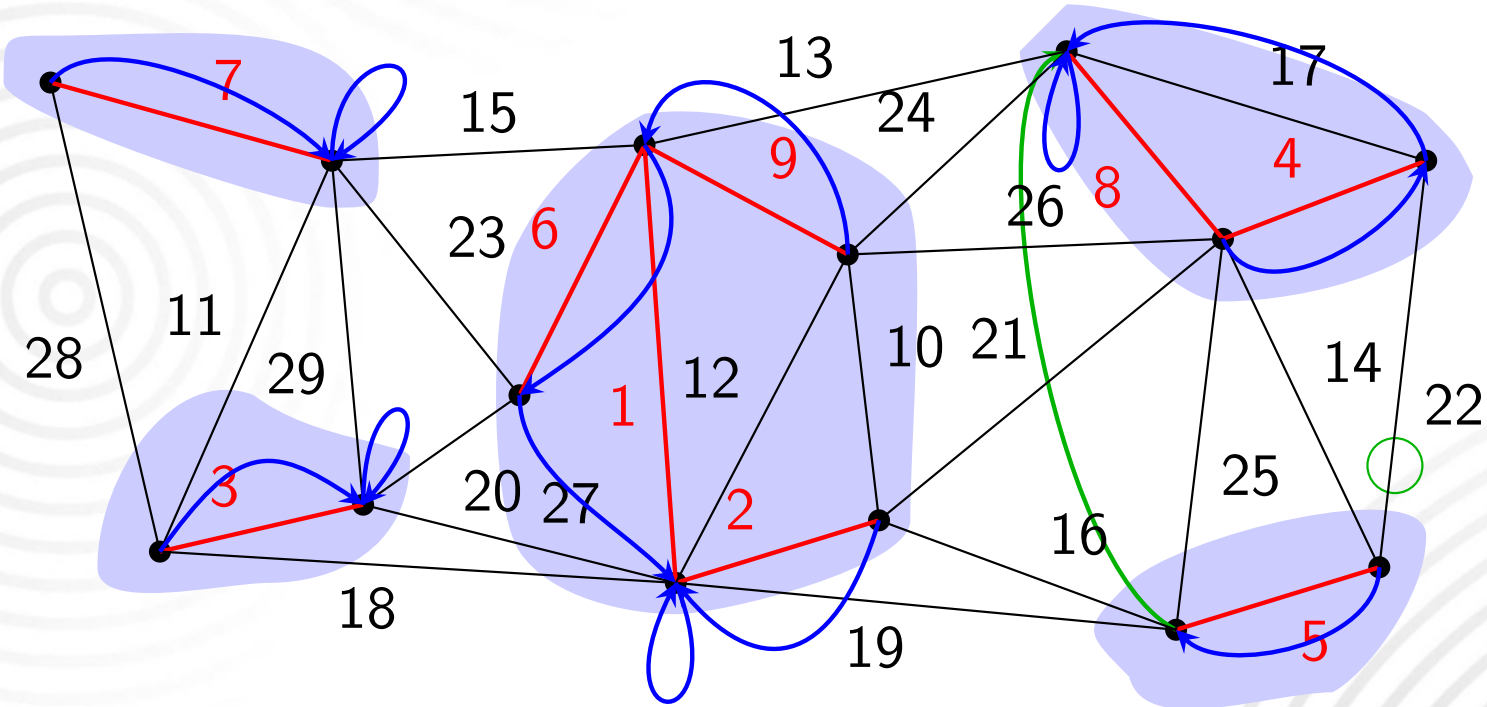
[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



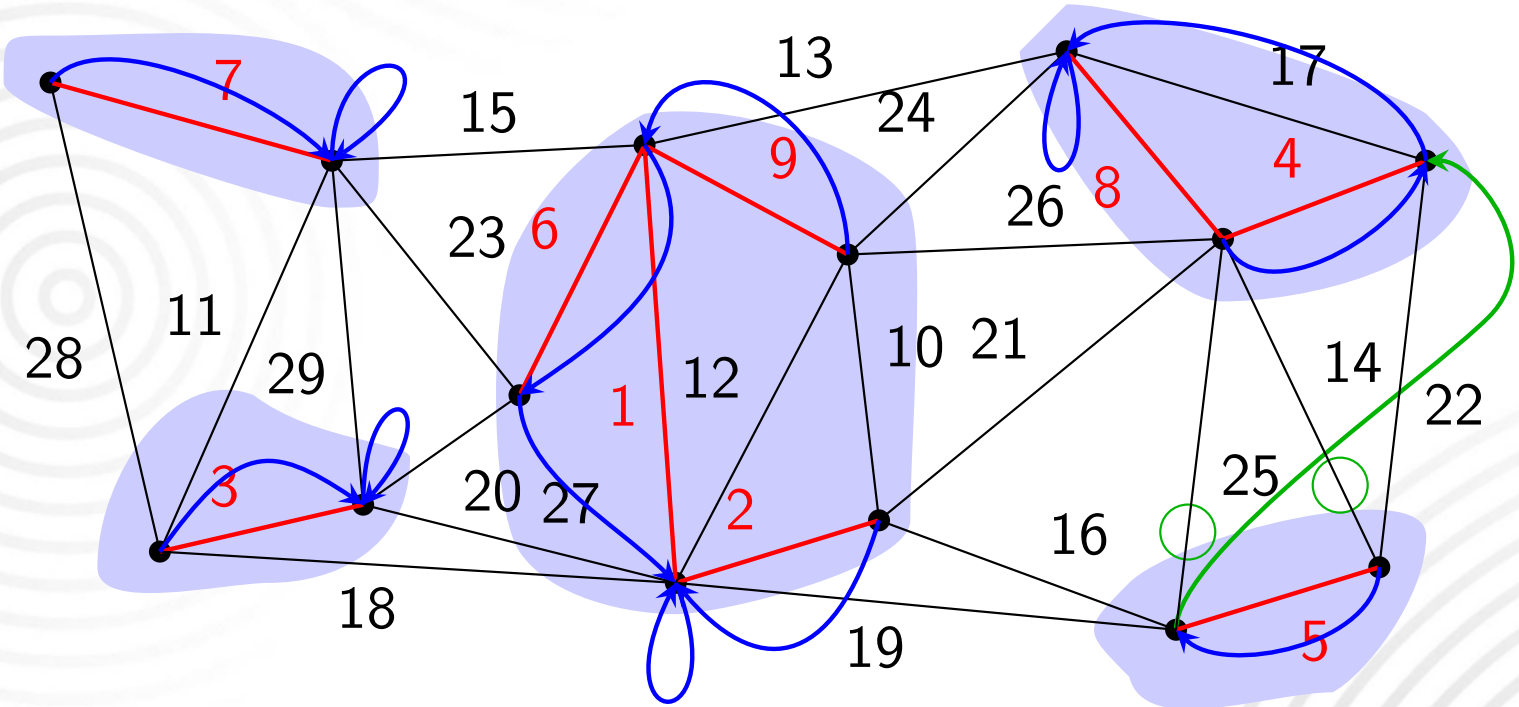
[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



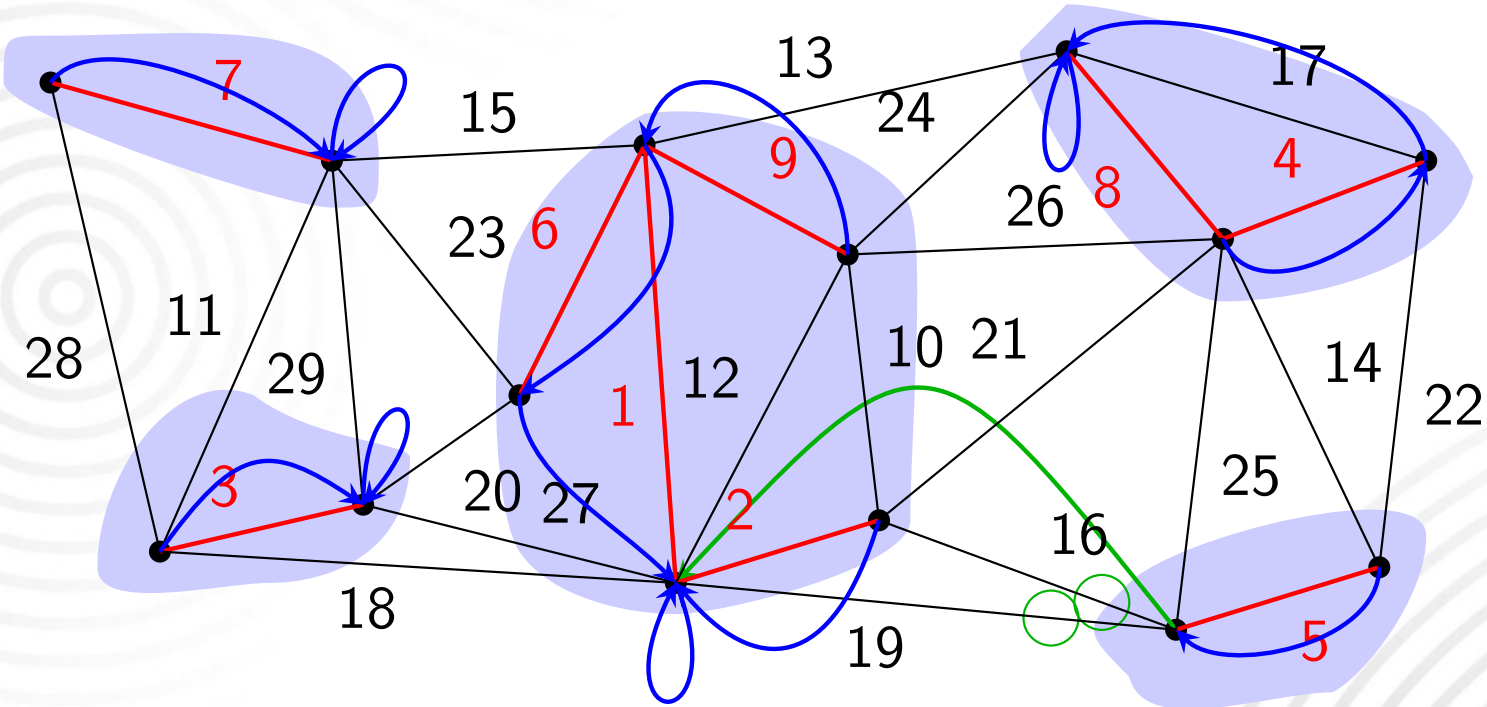
[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



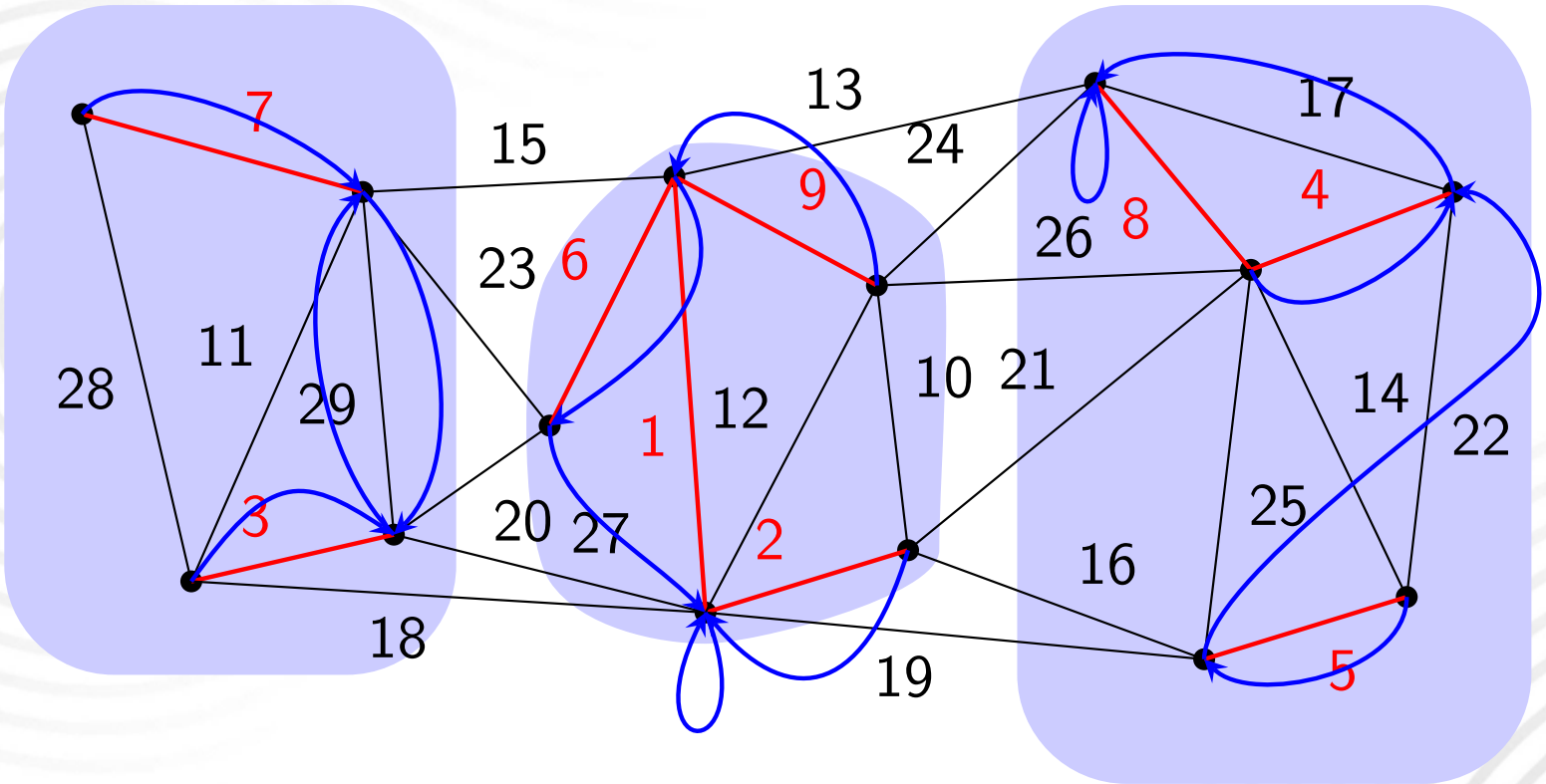
[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



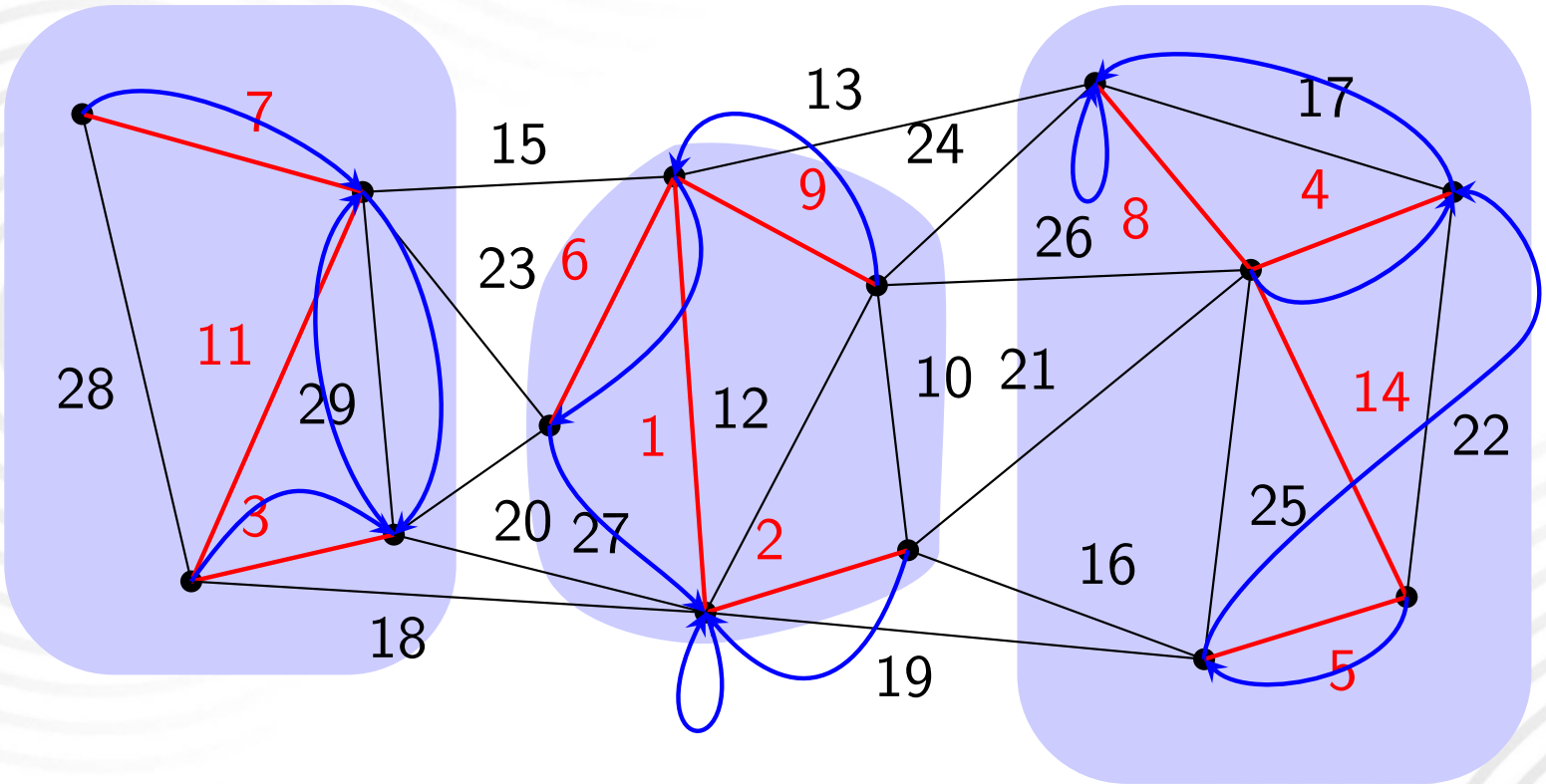
[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



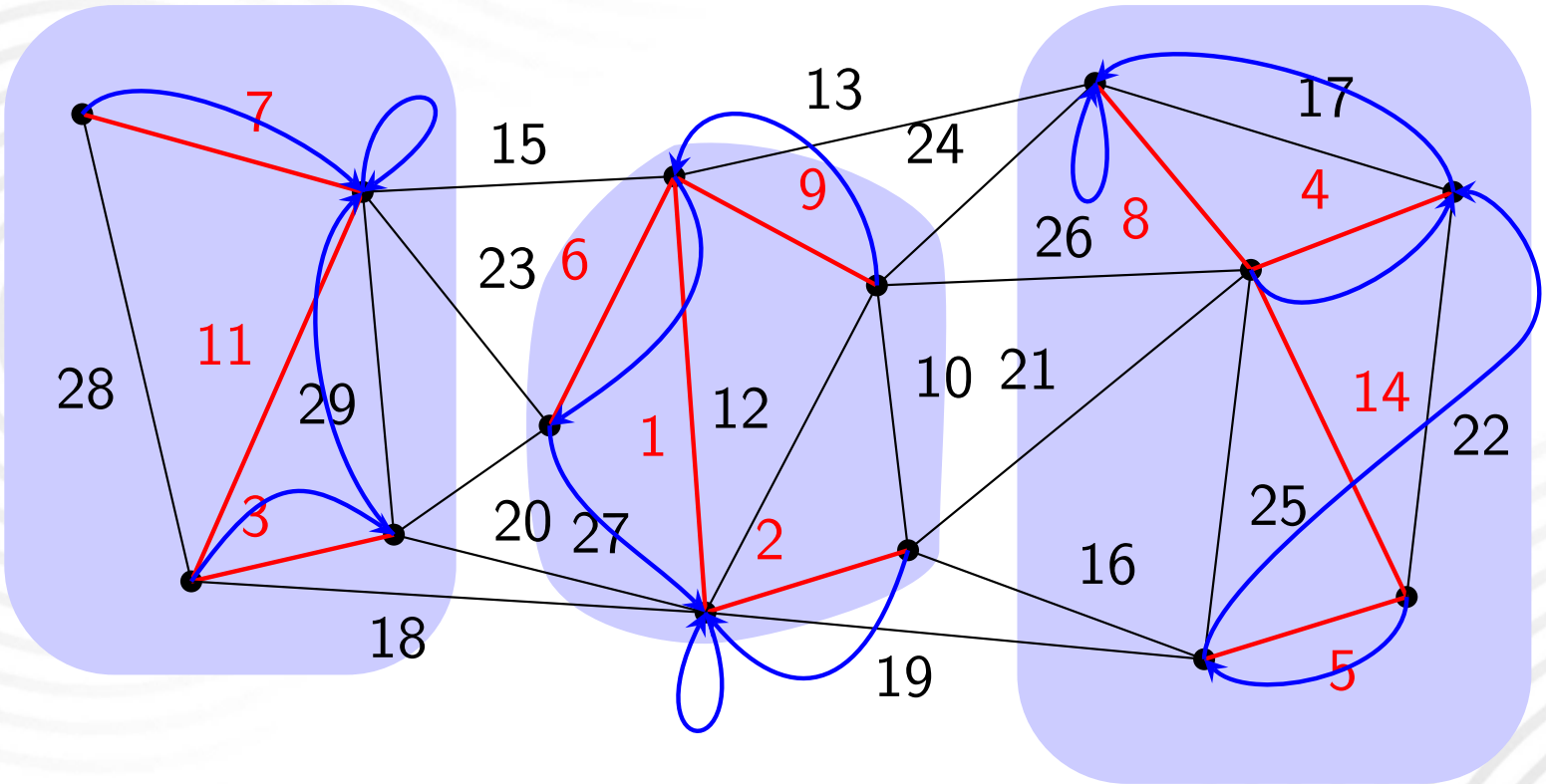
[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



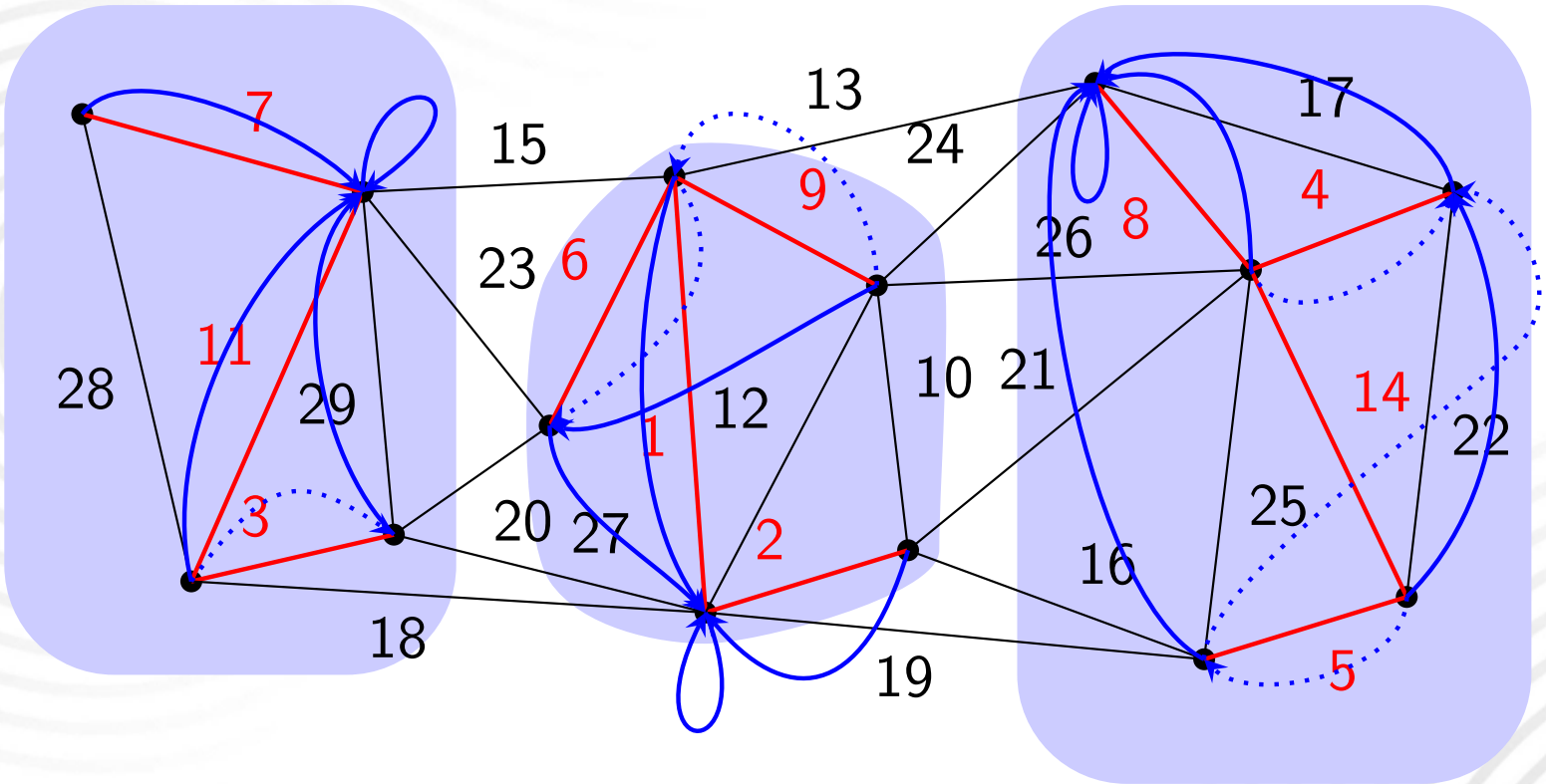
[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



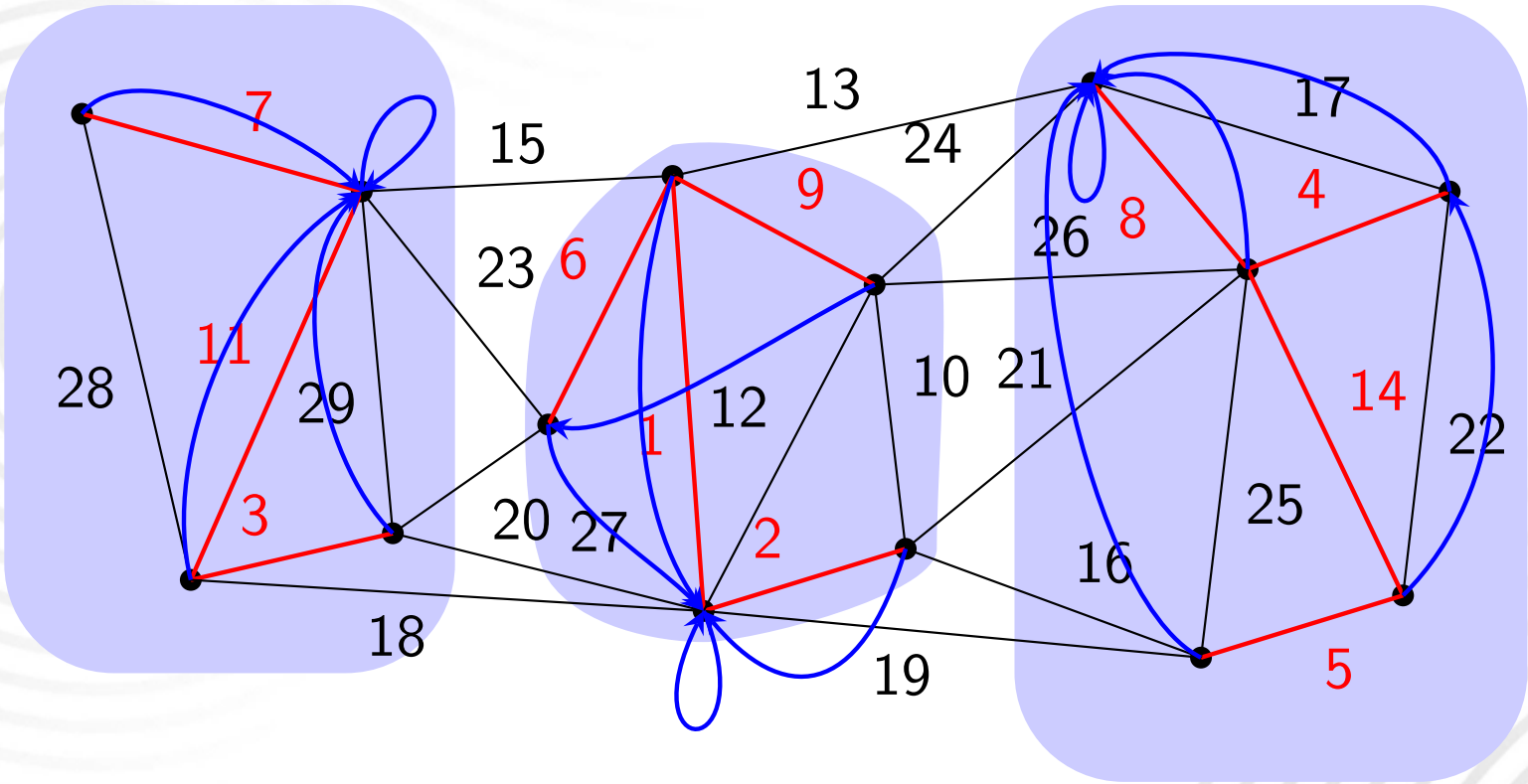
[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

Awerbuch-Shiloach algorithm



[B. Awerbuch, Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. on Computers 36(10):1258–1263, 1987]

On the implementation in Sharemind

Public inputs: $n = |V|$, $m = |E|$

Shared (private) inputs

- For each edge: indices of its end vertices $\in \{1, \ldots, n\}$
- For each edge: its weight $\in \mathbb{N}$

On the implementation in Sharemind

Public inputs: $n = |V|$, $m = |E|$

Shared (private) inputs

- For each edge: indices of its end vertices $\in \{1, \ldots, n\}$
- For each edge: its weight $\in \mathbb{N}$

Major steps in each iteration ($\log_{3/2} n$ in total)

- Sort a vector (of pairs of numbers) (5x)
  - Length: $2n$ (3x), $m + n$, $2m + n$
- Apply an oblivious shuffle to a vector of values (10x)
  - Length: $2n$ (5x), $m + n$ (3x), $2m + n$ (2x)
- Perform $O(m + n)$ arithmetic and logic operations

Some interesting details

Lack of certain dependencies in reading and writing

- Both reading and writing split into two parts:
  1. Depends on vector length, indices, priorities
     - sorts — complexity $O((m+n)\log(m+n))$
  2. Also depends on actual values in vector(s)
     - only applies shuffles — complexity $O(m+n)$

Some interesting details

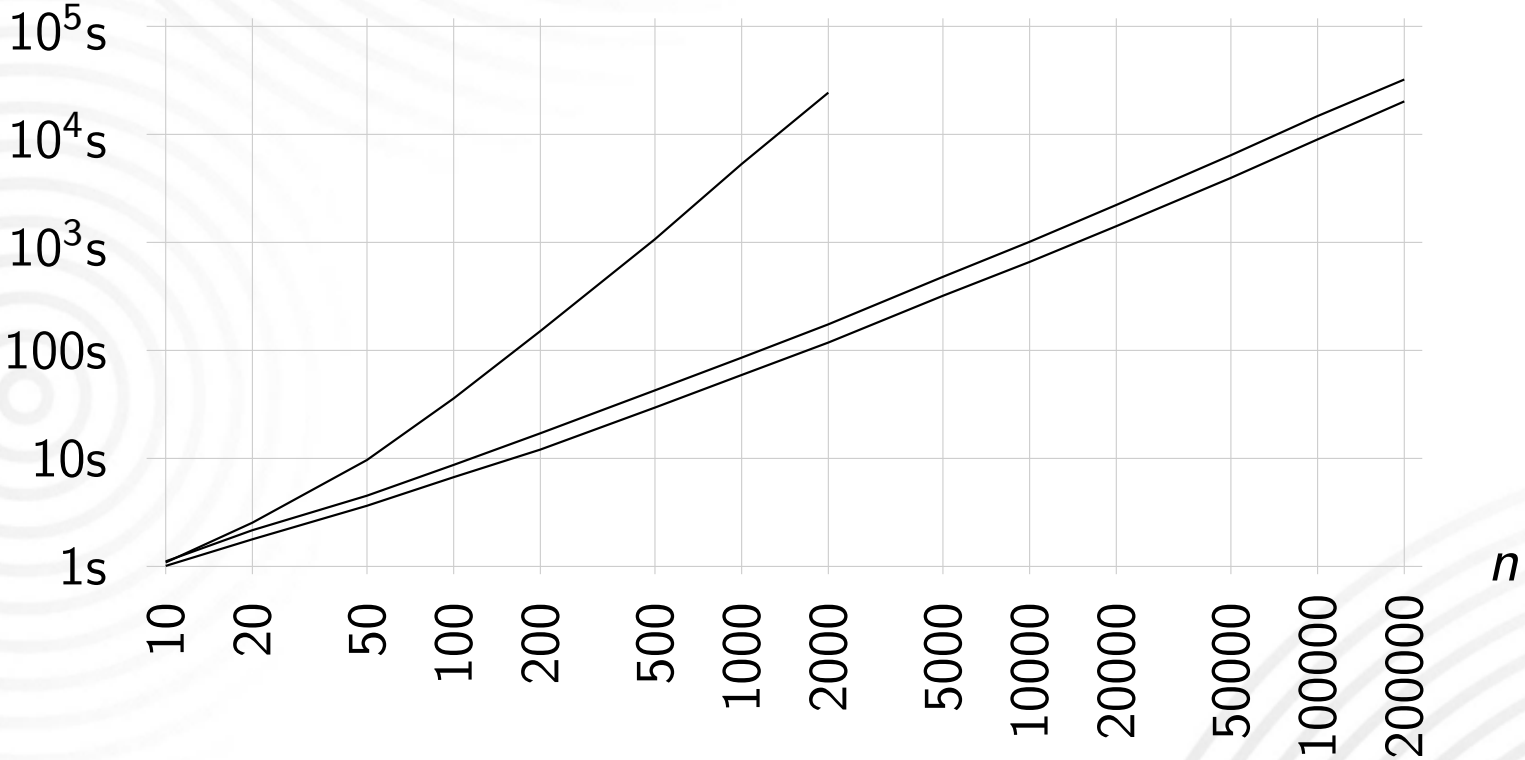
Lack of certain dependencies in reading and writing

- Both reading and writing split into two parts:
  1. Depends on vector length, indices, priorities
     - sorts — complexity $O((m+n)\log(m+n))$
  2. Also depends on actual values in vector(s)
     - only applies shuffles — complexity $O(m+n)$

Combining cycle-breaking and shortcutting

- After these operations, v's parent will be either v, its current parent or grandparent
- Choice depends on v, its parent, grandparent and great-grandparent
  - Grandparent and great-grandparent found by reading same indices

Running time for privacy-preserving minimum spanning tree



- Running times for $m = 3n$; $m = 6n$; $m = n(n-1)/2$
- On 1Gbps LAN: $n = 2 \cdot 10^5$, $m = 1.2 \cdot 10^6$, time < 9 hours