Parallel Sort-Based Matching for Data Distribution Management on Shared-Memory Multiprocessors

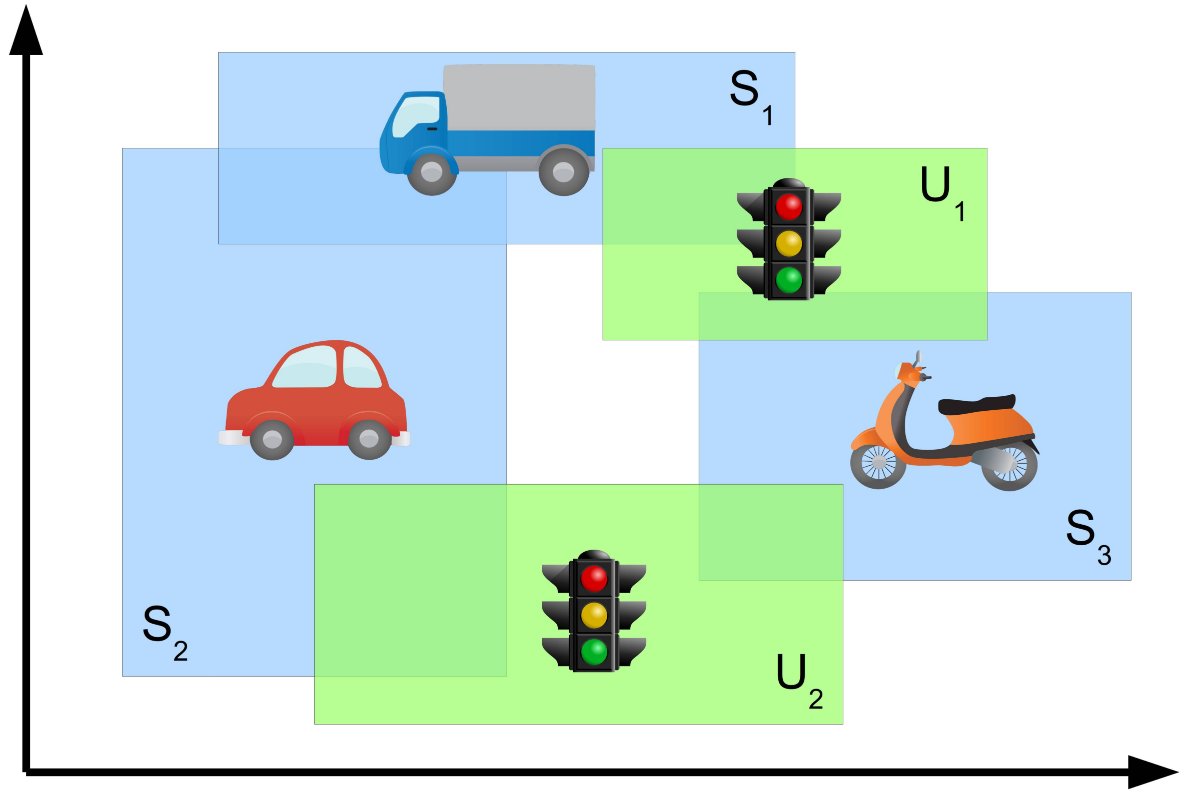
Moreno Marzolla (moreno.marzolla@unibo.it)
Gabriele D'Angelo (g.dangelo@unibo.it)

Dept. of Computer Science and Engineering
University of Bologna

Data Distribution Management

- DDM services are part of the IEEE 1516 “High Level Architecture” (HLA) specification
- Given
  - Sets of subscription and update regions in a *d*-dimensional space
  - Update regions (*extents*) generate events
  - Subscription regions must receive events generated by overlapping update regions
- Goal
  - Find find all update/subscription pairs that overlap

Example in d = 2 dimensions



Intersections:

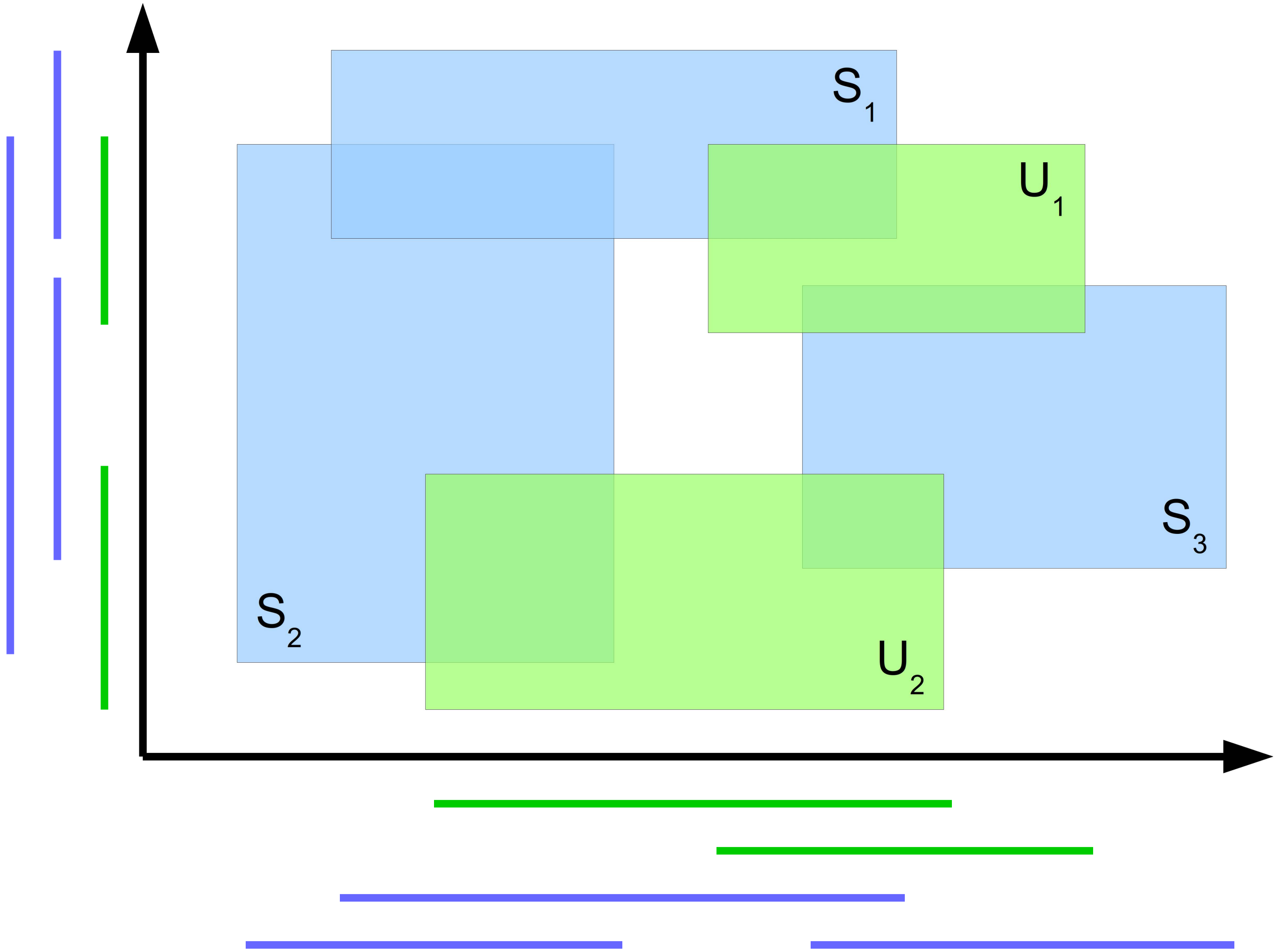
- (S_1, U_1), (S_2, U_2), (S_3, U_1), (S_3, U_2)

The Region Matching Problem

- Can be solved using spatial data structures and related algorithms
  - e.g., *k*-*d*-trees, *R*-trees, Quad-trees, …
- However, simpler algorithms are generally preferred for DDM implementations
  - Brute-Force
  - Grid-Based [Boukerche and Dzermajko 2001]
  - Sort-Based [Raczy, Tan and Yu 2005]
  - Interval-Tree [Marzolla, D'Angelo and Mandrioli 2013]

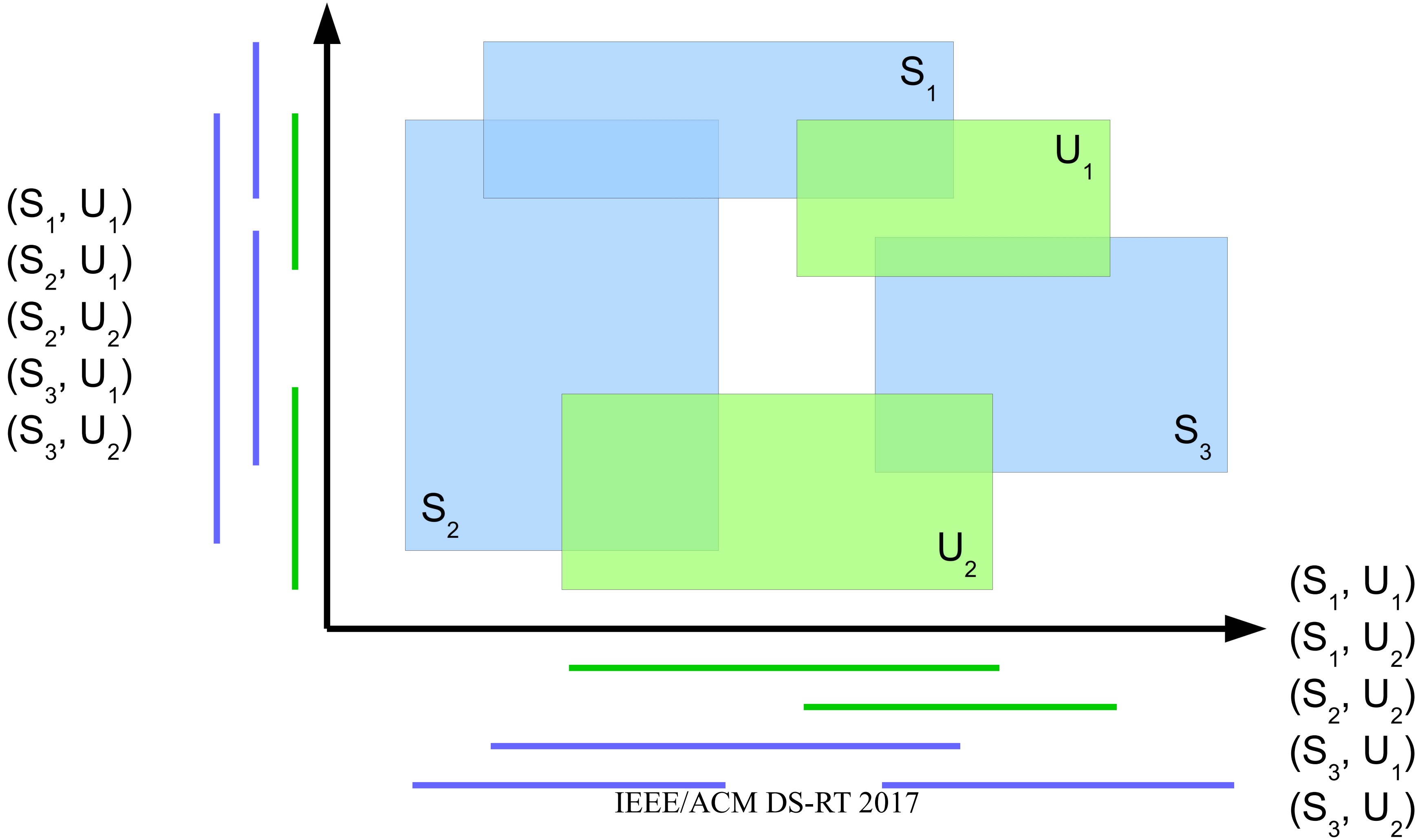
The Region Matching Problem

- The Region Matching Problem in *d* > 1 dimensions can be reduced to *d* instances on 1D intervals



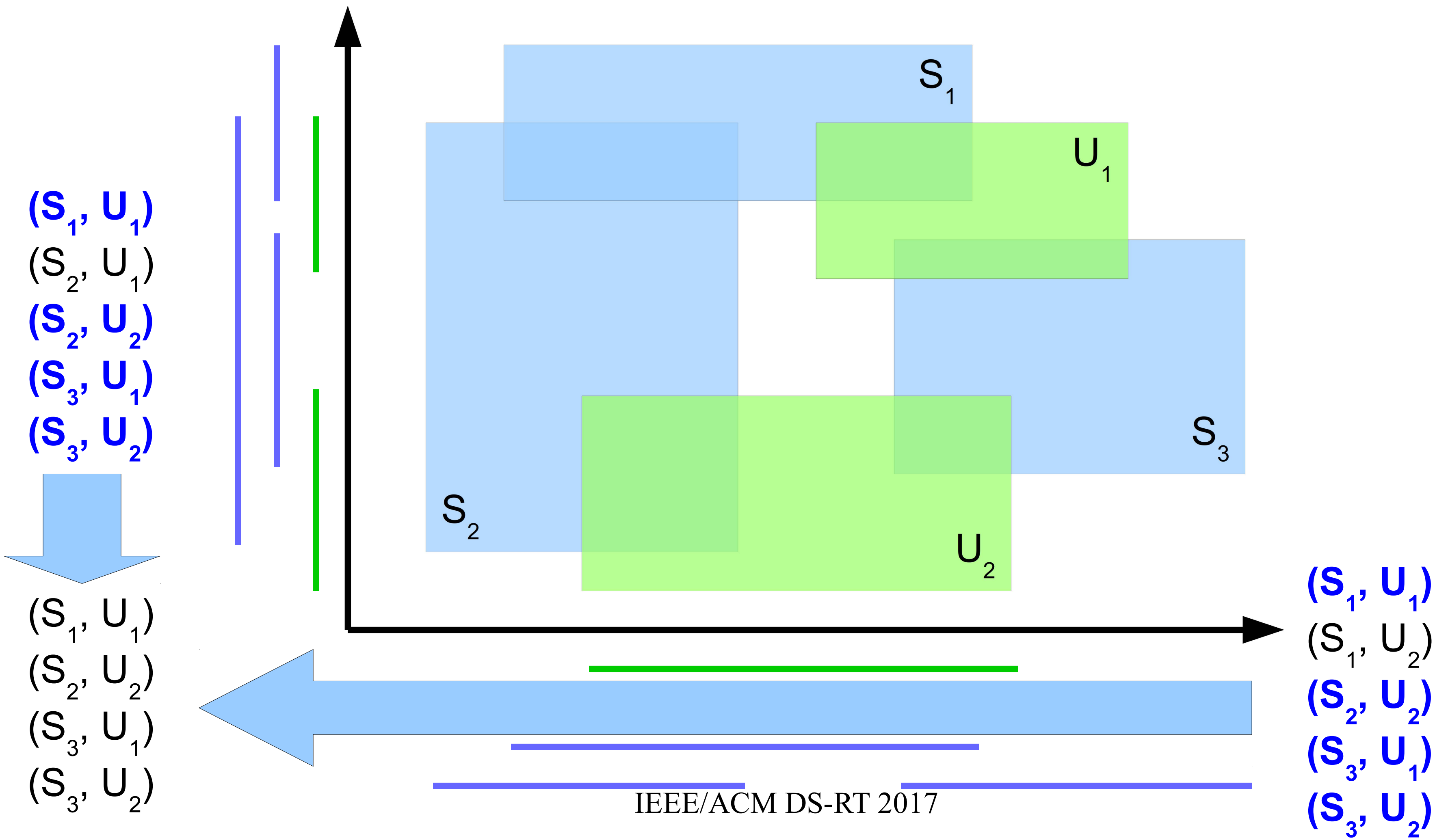
The Region Matching Problem

- The Region Matching Problem in *d* > 1 dimensions can be reduced to *d* instances on 1D intervals



The Region Matching Problem

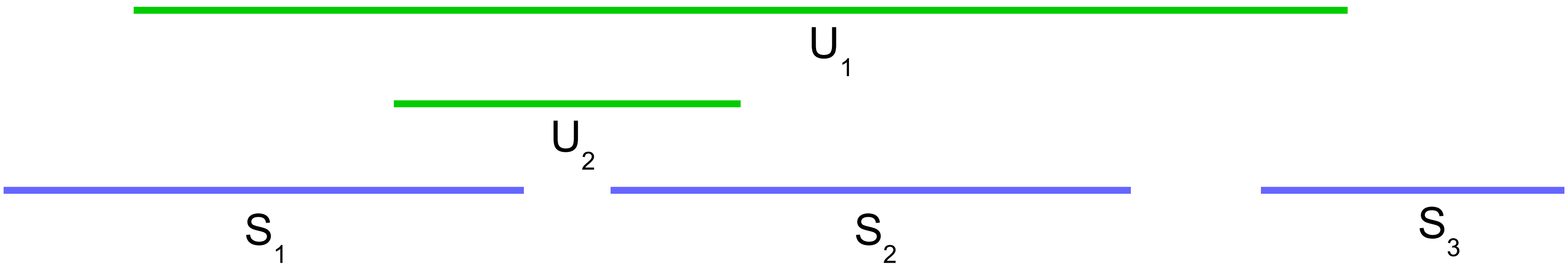
- The Region Matching Problem in $d > 1$ dimensions can be reduced to d instances on 1D intervals



Sort-Based Matching

- Sort endpoints
- Scan endpoints in sorted order
  - Let *SubSet* and *UpdSet* be the sets of currently active subscription and update intervals, resp.
  - For each endpoint *t*
    - If *t* marks the beginning of a subs/upd interval *X*, then
      - add *X* to *SubSet* or *UpdSet*
    - Else
      - remove *X* from *SubSet* or *UpdSet*
      - *X* overlaps with all intervals currently in *UpdSet* (if *X* is a subscription extent) or *SubSet* (if *X* is an update extent)

Example

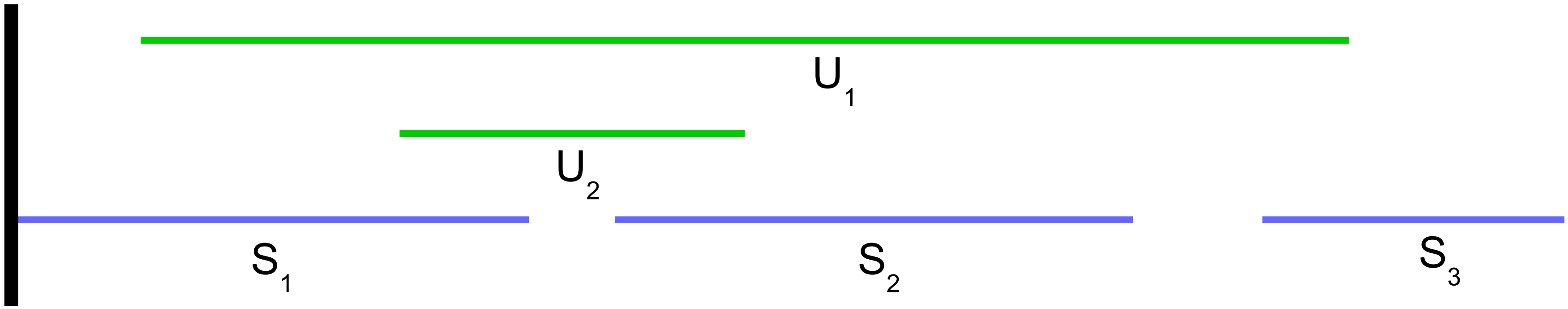


UpdSet = { }

SubSet = { }

Intersections = { }

Example

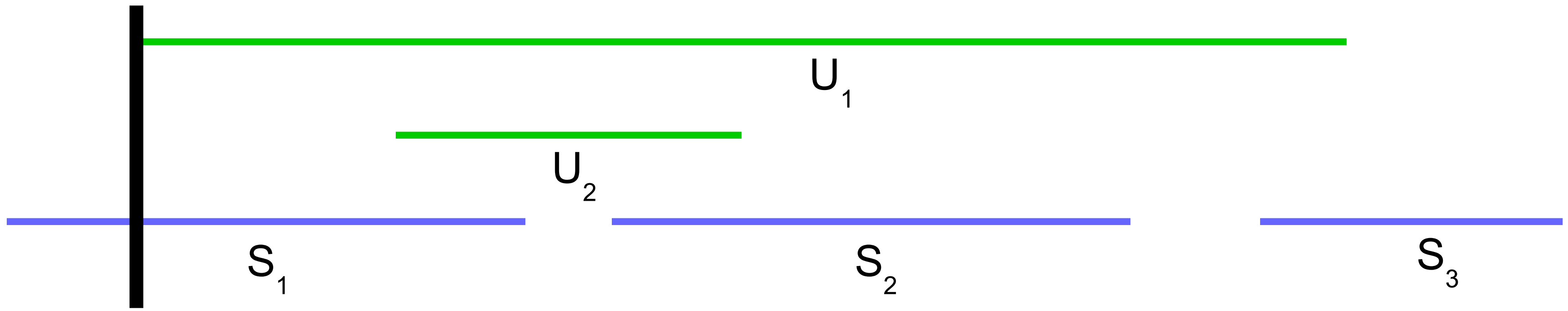


UpdSet = { }

SubSet = { S_1 }

Intersections = { }

Example

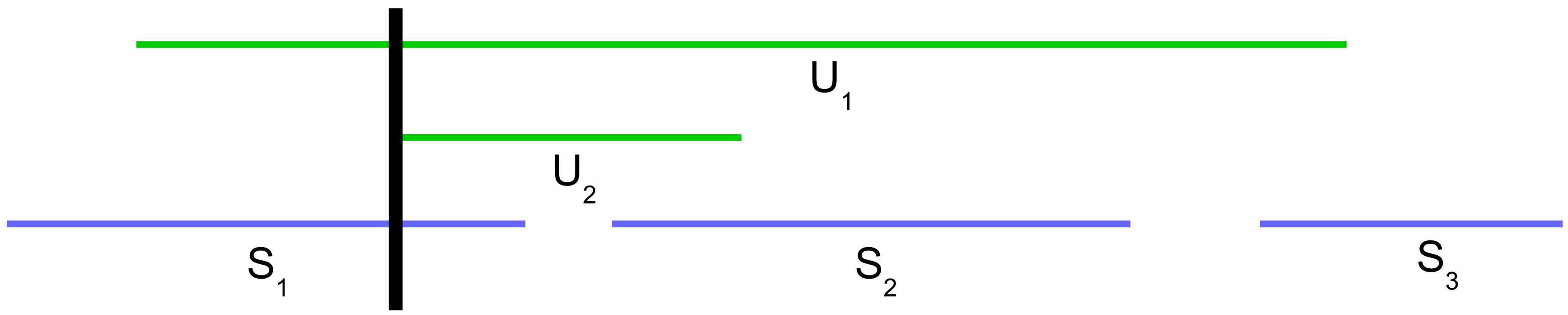


UpdSet = { U_1 }

SubSet = { S_1 }

Intersections = { }

Example

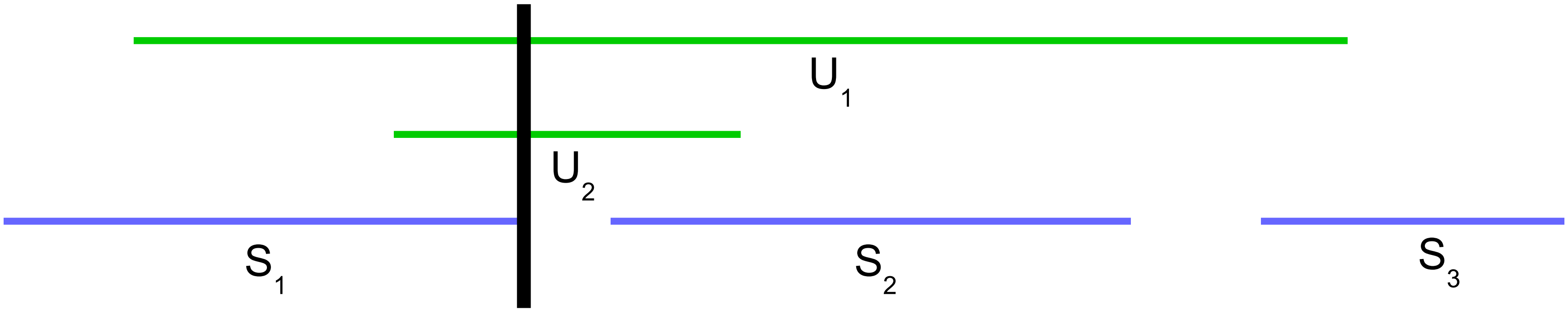


UpdSet = { U_1 , U_2 }

SubSet = { S_1 }

Intersections = { }

Example

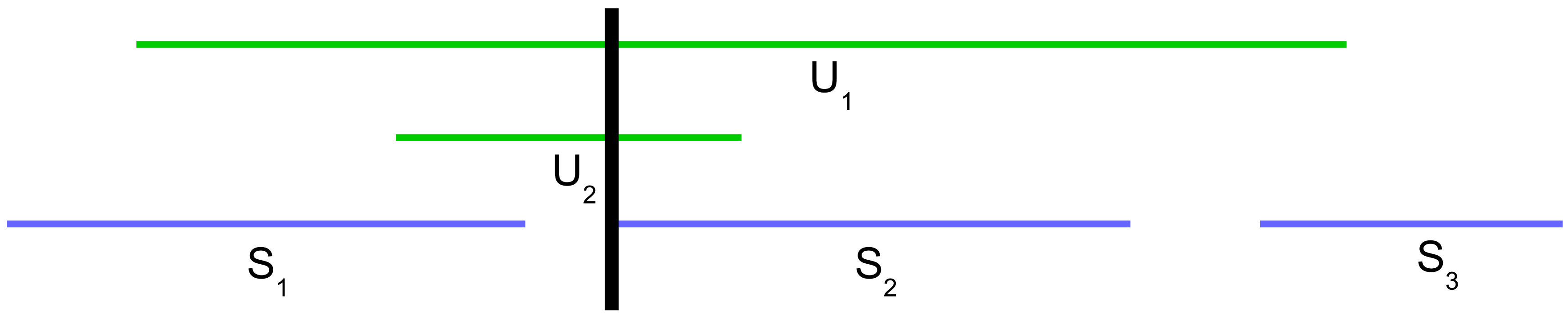


UpdSet = { U_1 , U_2 }

SubSet = { }

Intersections = { (S_1, U_1), (S_1, U_2) }

Example

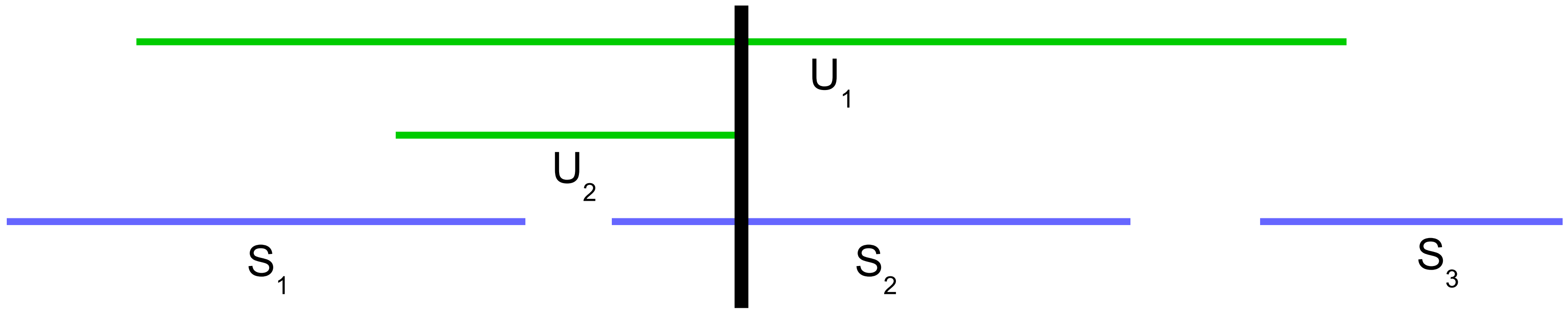


UpdSet = { U_1 , U_2 }

SubSet = { S_2 }

Intersections = { (S_1, U_1), (S_1, U_2) }

Example

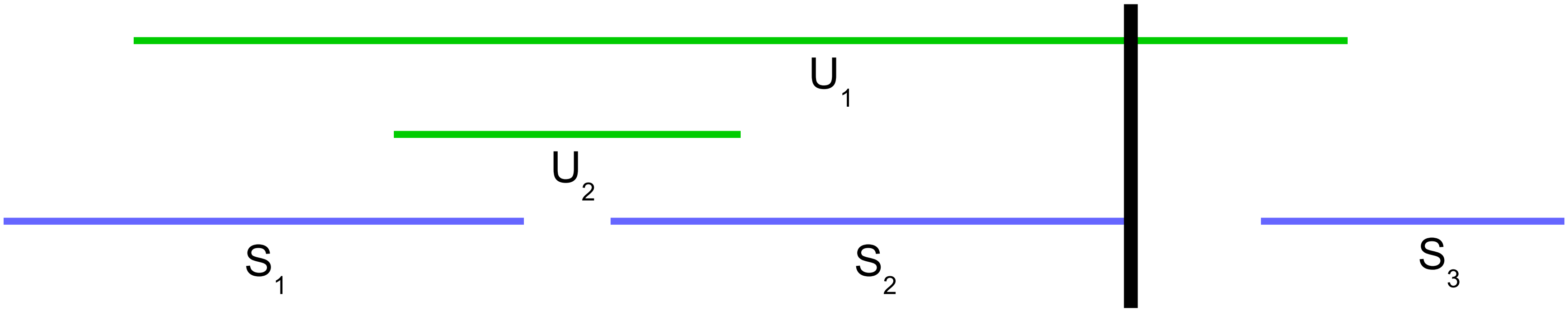


UpdSet = { U_1 }

SubSet = { S_2 }

Intersections = { (S_1, U_1), (S_1, U_2), (S_2, U_2) }

Example

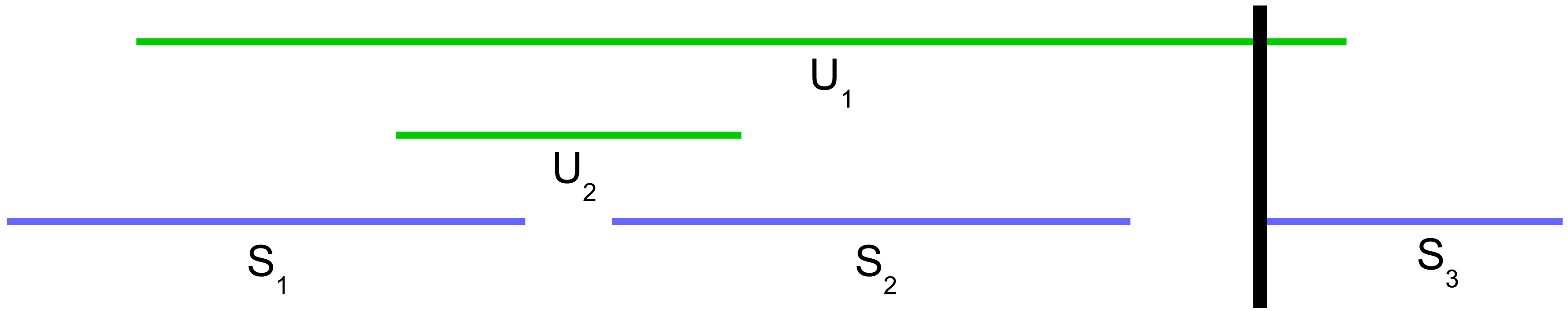


$UpdSet = \{ U_1 \}$

$SubSet = \{ \}$

$Intersections = \{ (S_1, U_1), (S_1, U_2), (S_2, U_2),$
$(S_2, U_1) \}$

Example

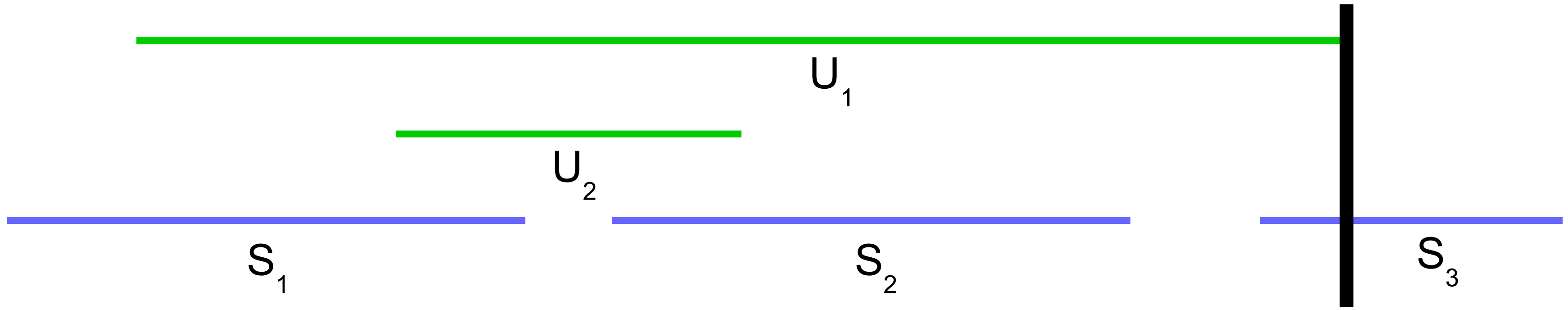


UpdSet = { U_1 }

SubSet = { S_3 }

Intersections = { (S_1, U_1), (S_1, U_2), (S_2, U_2), (S_2, U_1) }

Example

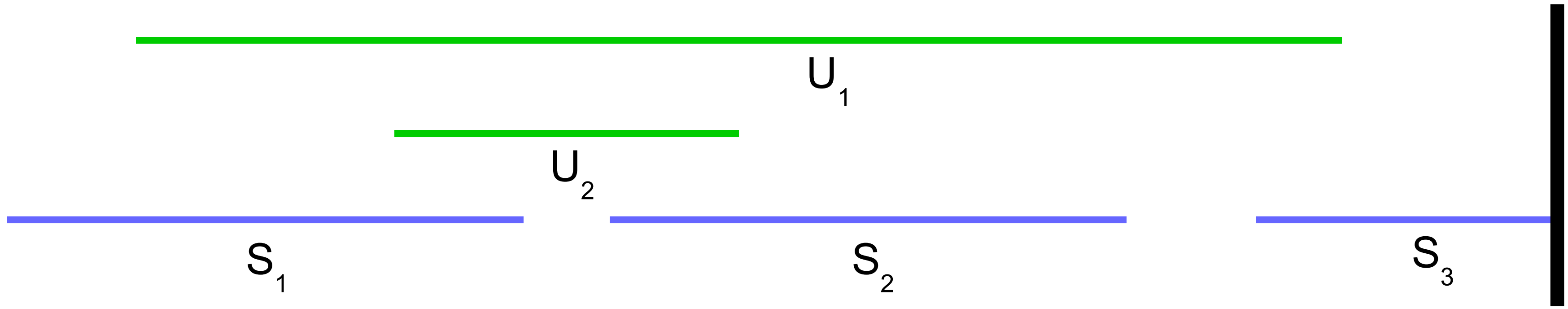


UpdSet = { }

SubSet = { S_3 }

Intersections = { (S_1, U_1), (S_1, U_2), (S_2, U_2), (S_2, U_1), (S_3, U_1) }

Example



UpdSet = { }

SubSet = { }

Intersections = { (S_1, U_1), (S_1, U_2), (S_2, U_2), (S_2, U_1), (S_3, U_1) }

Parallel Sort-Based Matching on Shared-Memory Systems

- Sort endpoints
- Scan endpoints
  - Let *SubSet* and *UpdSet* be the sets of currently active subscription and update intervals, resp.
  - For each endpoint *t*
    - If *t* marks the beginning of a subs/upd interval *X*, then
      - add *X* to *SubSet* or *UpdSet*
    - Else
      - remove *X* from *SubSet* or *UpdSet*
      - *X* overlaps with all intervals currently in *UpdSet* (if *X* is a subscription extent) or *SubSet* (if *X* is an update extent)

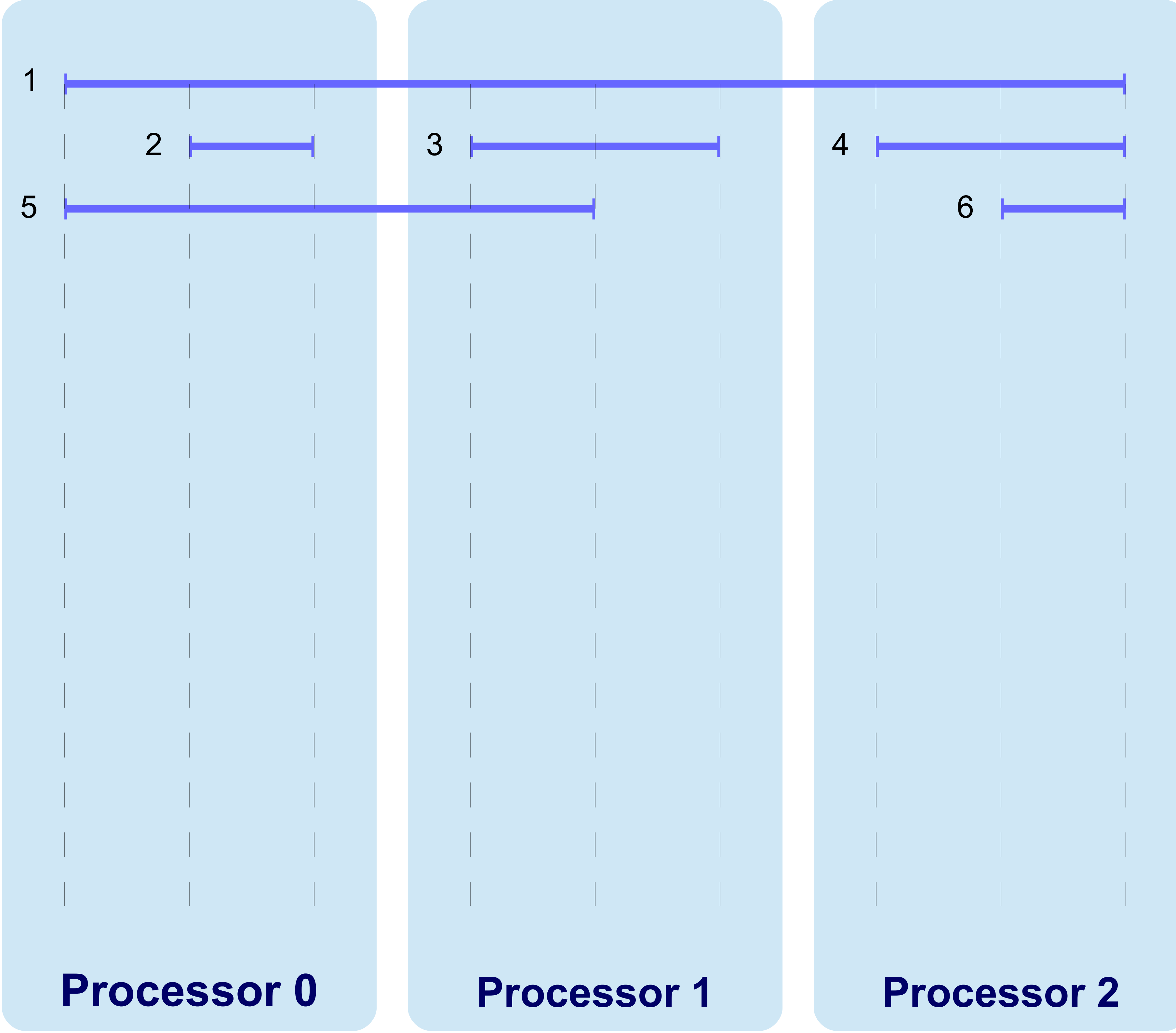
Parallel Sort-Based Matching on Shared-Memory Systems

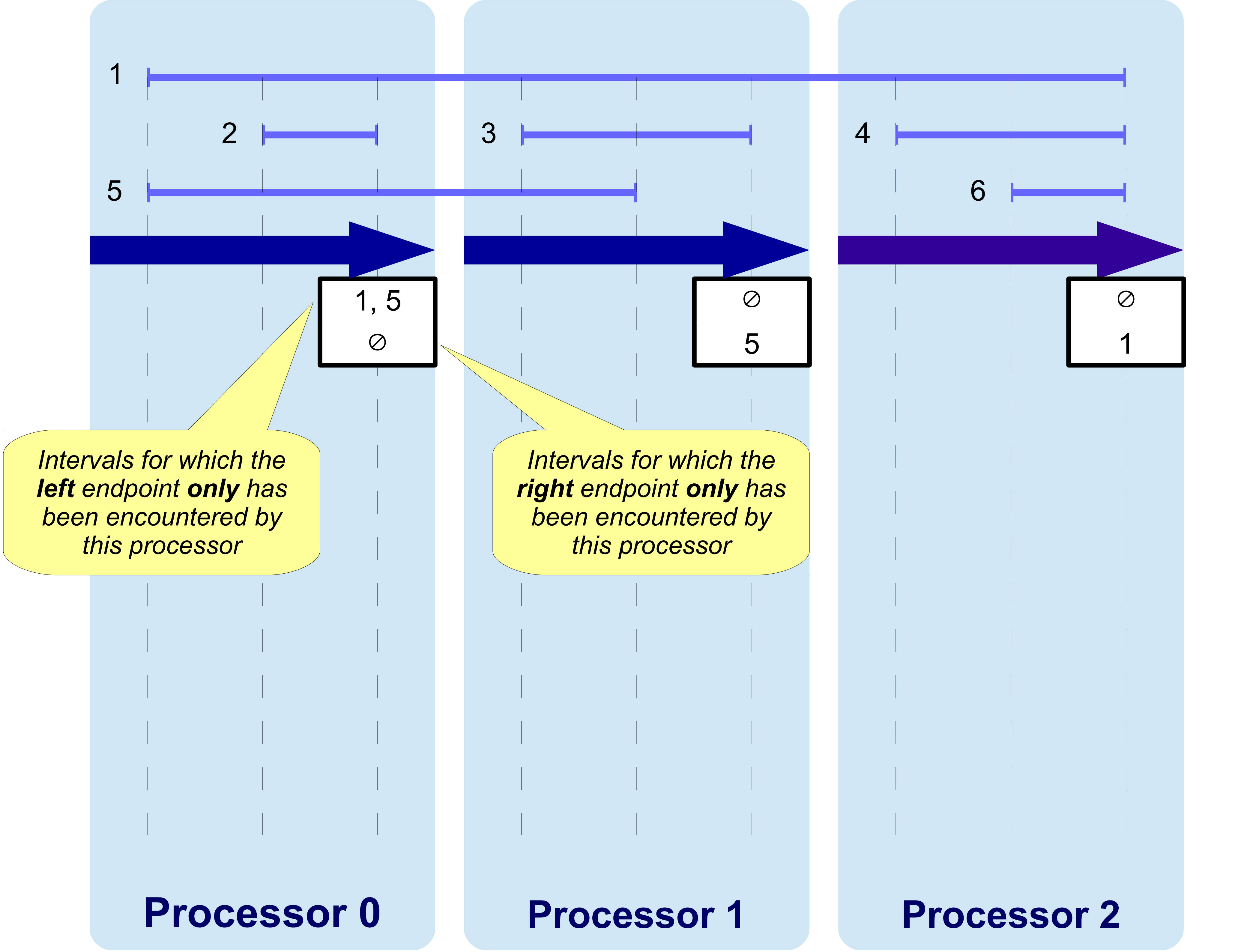
- Sort endpoints in parallel
- Scan endpoints
  - Let *SubSet* and *UpdSet* be the sets of currently active subscription and update intervals, resp.
  - For each endpoint *t*
    - If *t* marks the beginning of a subs/upd interval *X*, then
      - add *X* to *SubSet* or *UpdSet*
    - Else
      - remove *X* from *SubSet* or *UpdSet*
      - *X* overlaps with all intervals currently in *UpdSet* (if *X* is a subscription extent) or *SubSet* (if *X* is an update extent)

Parallel Sort-Based Matching on Shared-Memory Systems

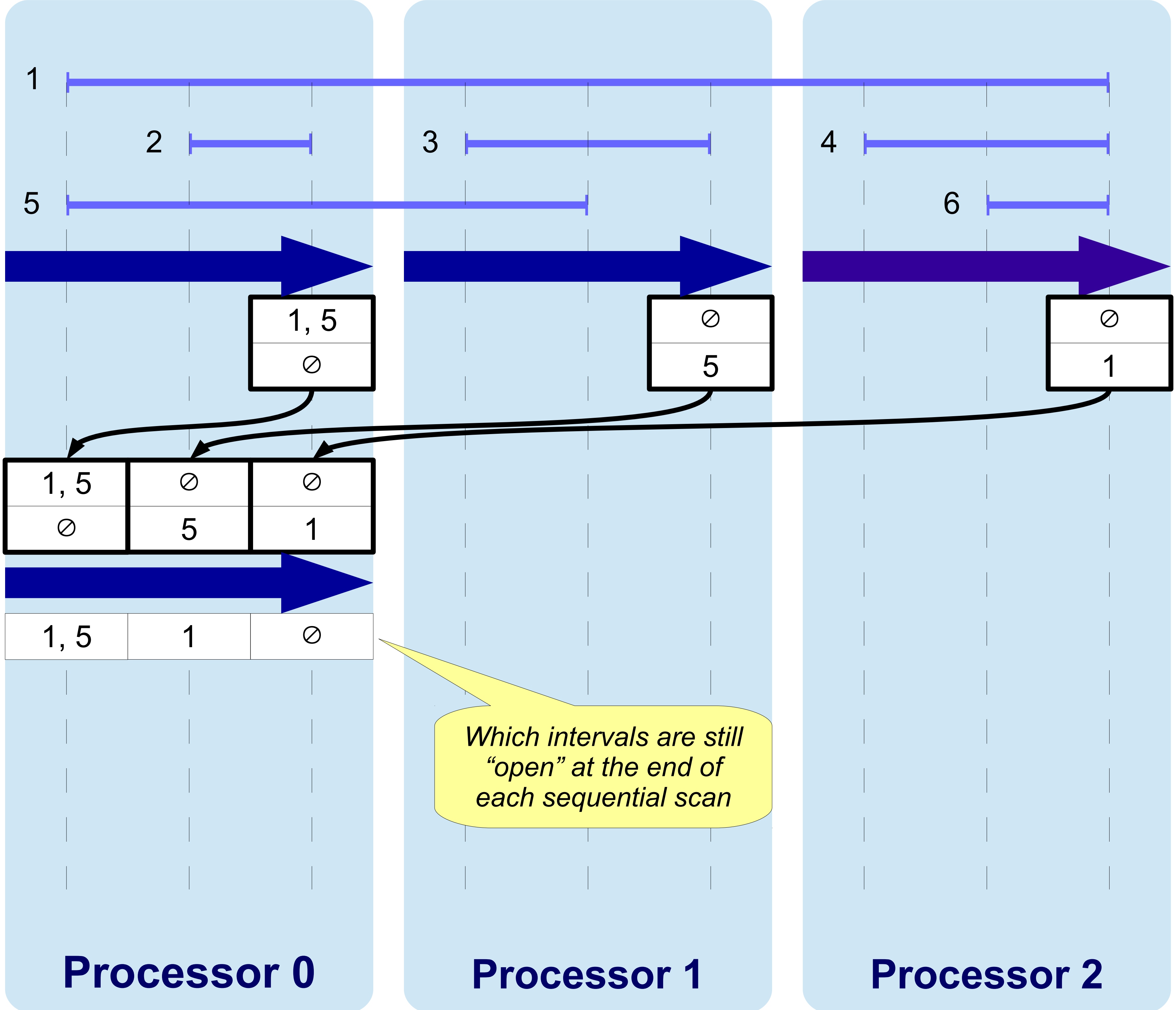
- Sort endpoints in parallel
- Scan endpoints in parallel???
  - Let *SubSet* and *UpdSet* be the sets of currently active subscription and update intervals, resp.
  - For each endpoint *t*
    - If *t* marks the beginning of a subs/upd interval *X*, then
      - add *X* to *SubSet* or *UpdSet*
    - Else
      - remove *X* from *SubSet* or *UpdSet*
      - *X* overlaps with all intervals currently in *UpdSet* (if *X* is a subscription extent) or *SubSet* (if *X* is an update extent)

Loop-carried dependencies

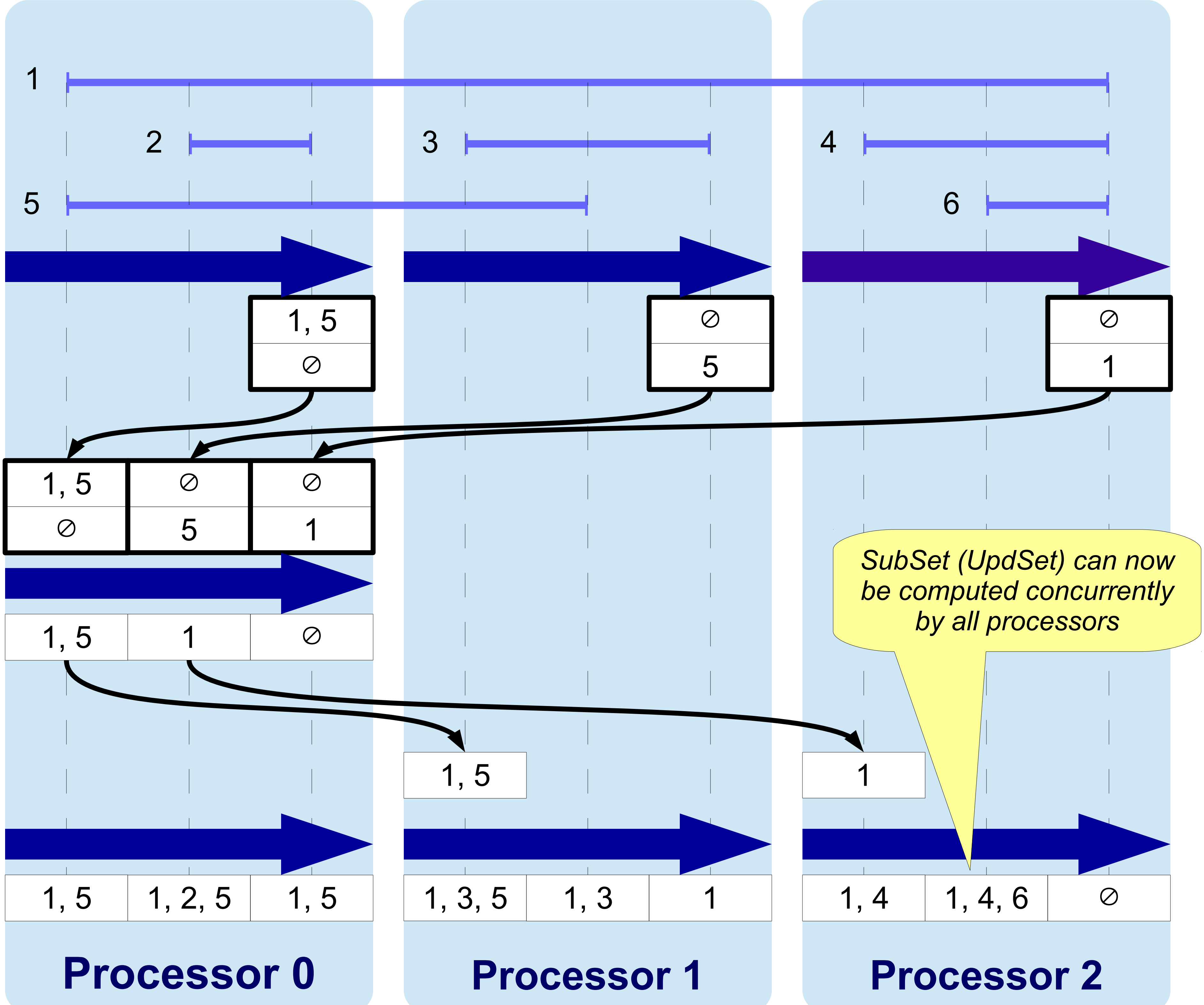

1
2
3
4
5
6
Processor 0
Processor 1
Processor 2



1
2
3
4
5
6
1, 5
∅
∅
5
∅
1
Intervals for which the *left* endpoint *only* has been encountered by this processor
Intervals for which the *right* endpoint *only* has been encountered by this processor
Processor 0
Processor 1
Processor 2



1
2
3
4
5
6
1, 5
∅
∅
5
∅
1
1, 5
∅
∅
5
∅
1
1, 5
1
∅
Which intervals are still "open" at the end of each sequential scan
Processor 0
Processor 1
Processor 2



1
2
3
4
5
6
1, 5
∅
∅
5
∅
1
1, 5
∅
∅
5
∅
1
1, 5
1
∅
SubSet (UpdSet) can now be computed concurrently by all processors
1, 5
1
1, 5
1, 2, 5
1, 5
1, 3, 5
1, 3
1
1, 4
1, 4, 6
∅
Processor 0
Processor 1
Processor 2

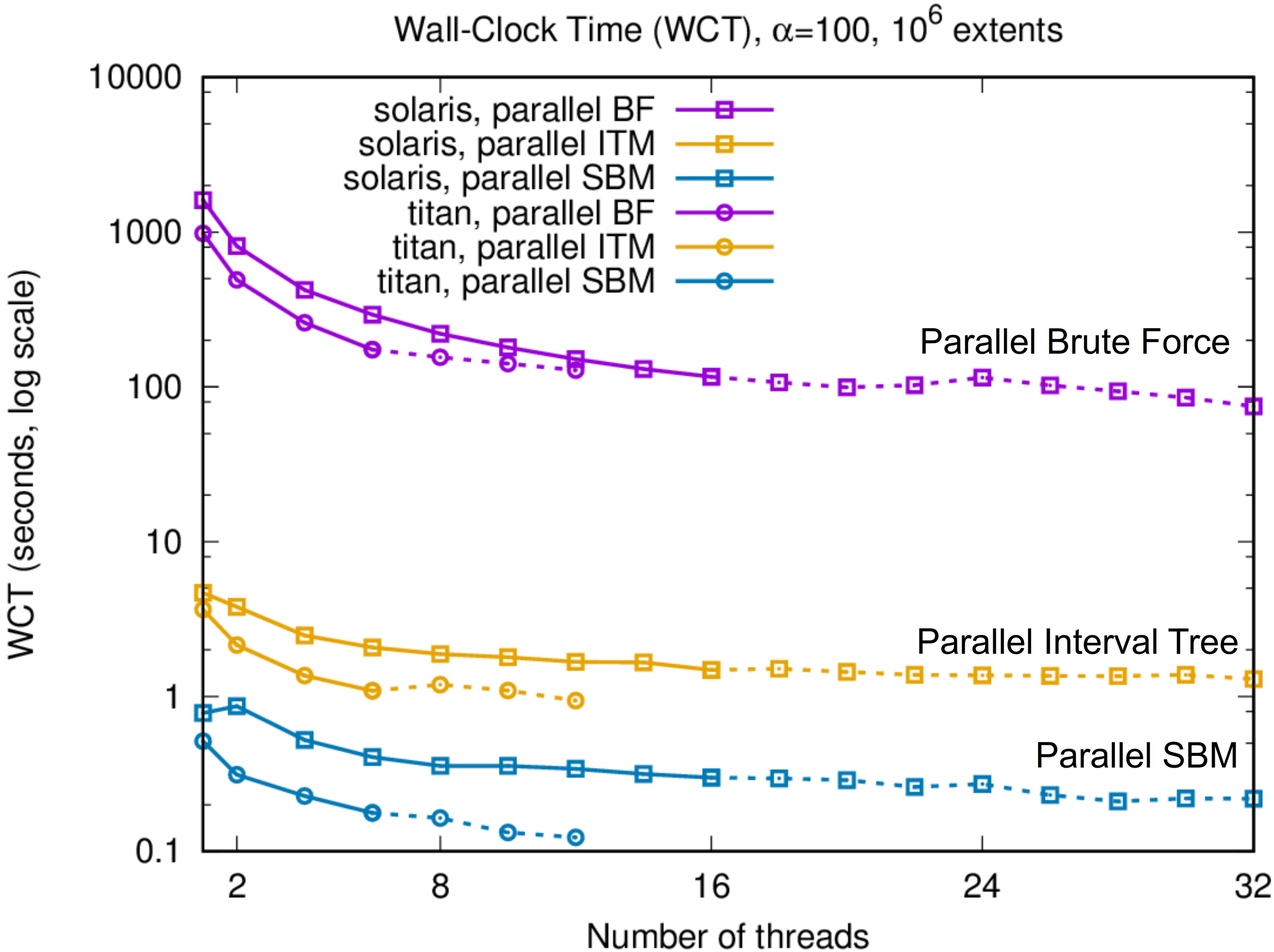
Performance Evaluation

- Parallel SBM implemented in C++/OpenMP
- Testing according to the methodology used in [Raczy et al. 2005]
- Instances with a single dimension
- Parameters:
  - N = number of intervals
  - α = overlapping degree = $\frac{\sum \text{Area of intervals}}{\text{Total area of the routing space}}$

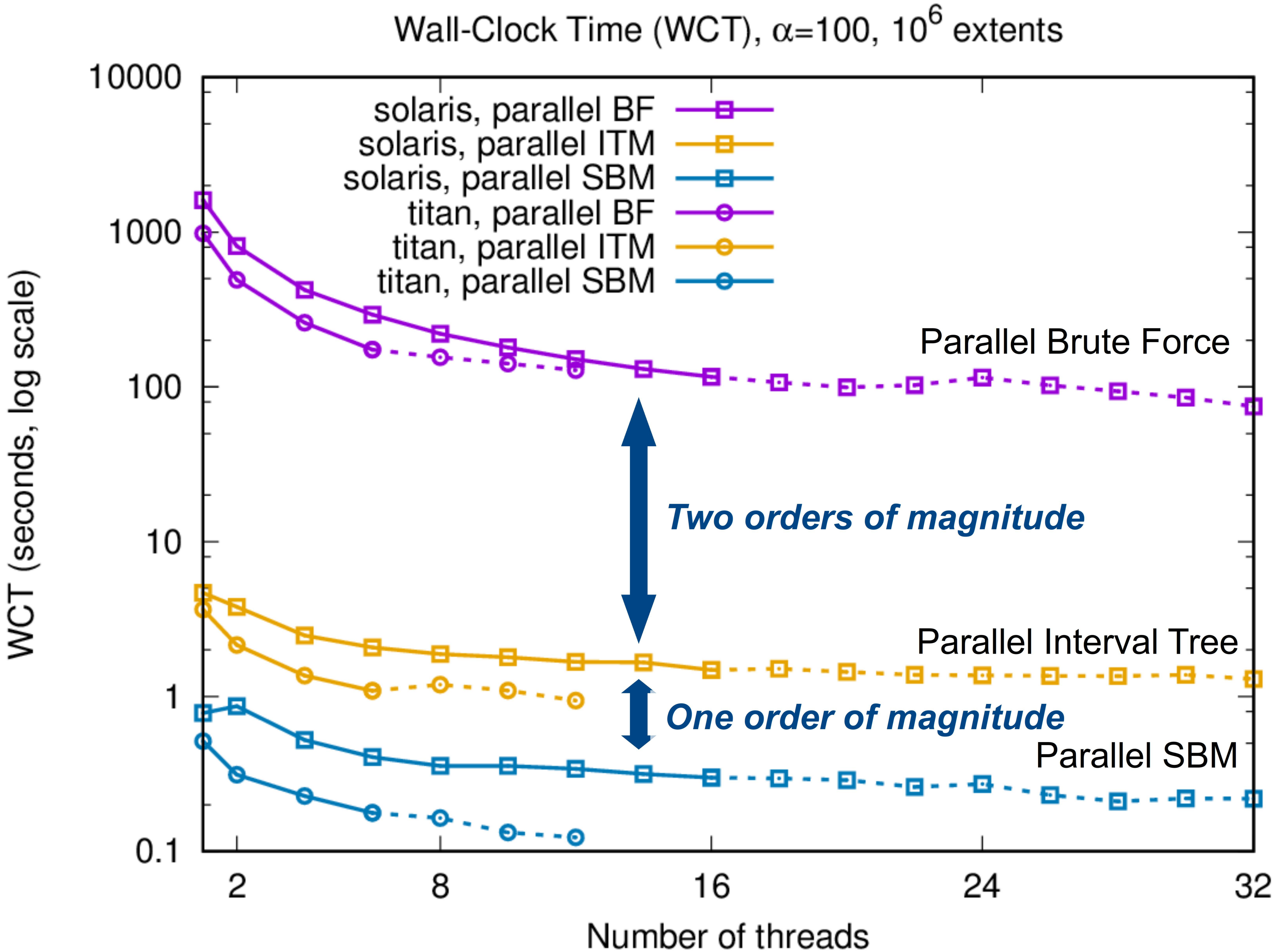
Execution platforms

	Solaris	**Titan**
CPU	Intel Xeon E5-2640	Intel Core I7-5820K
Clock freq.	2 GHz	3.3 GHz
Processors	2	1
Tot n. of cores	16	6
Hyperthreading?	Yes	Yes
RAM	128GB	64GB

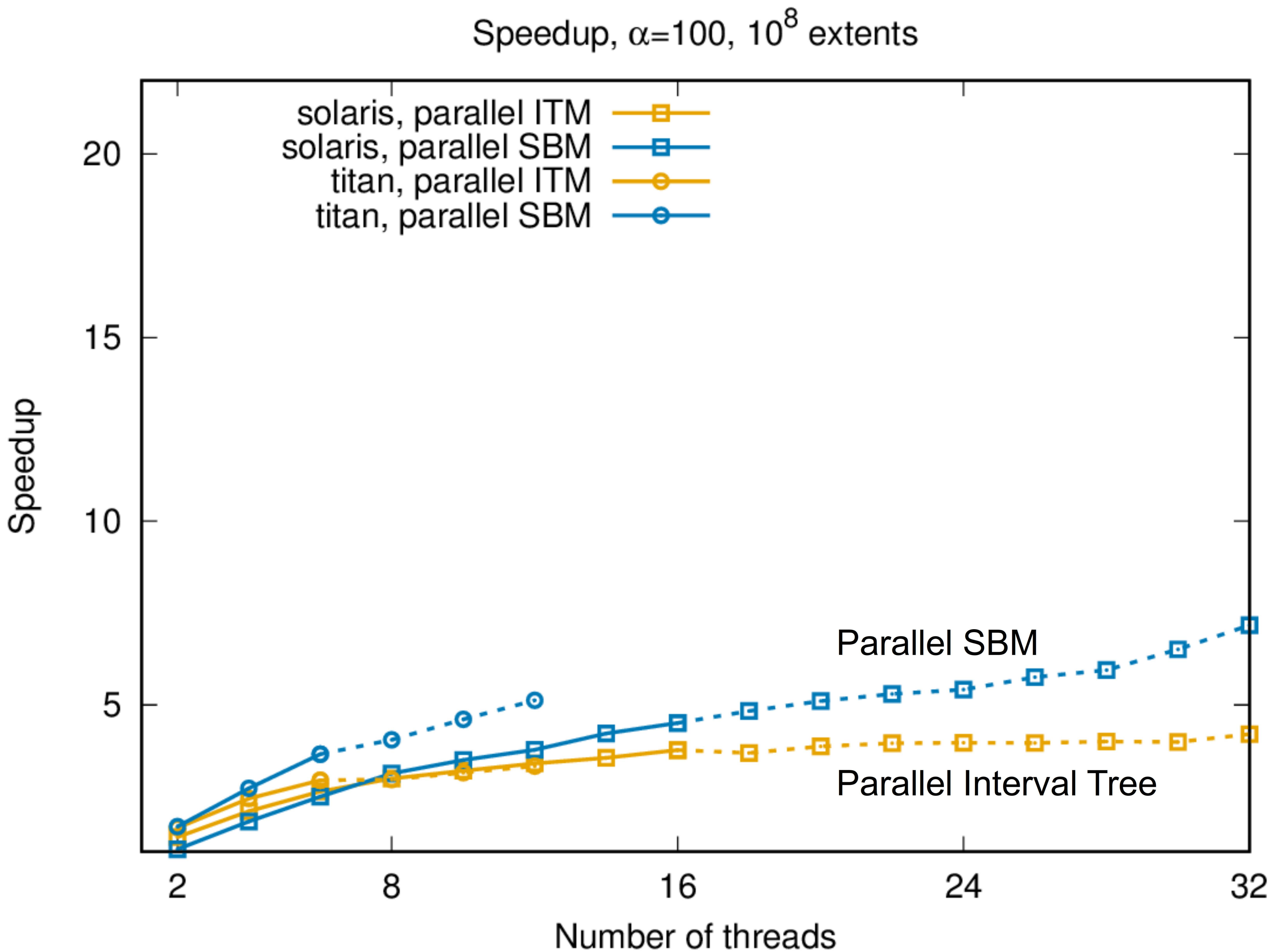
Wall-Clock Time



Wall-Clock Time

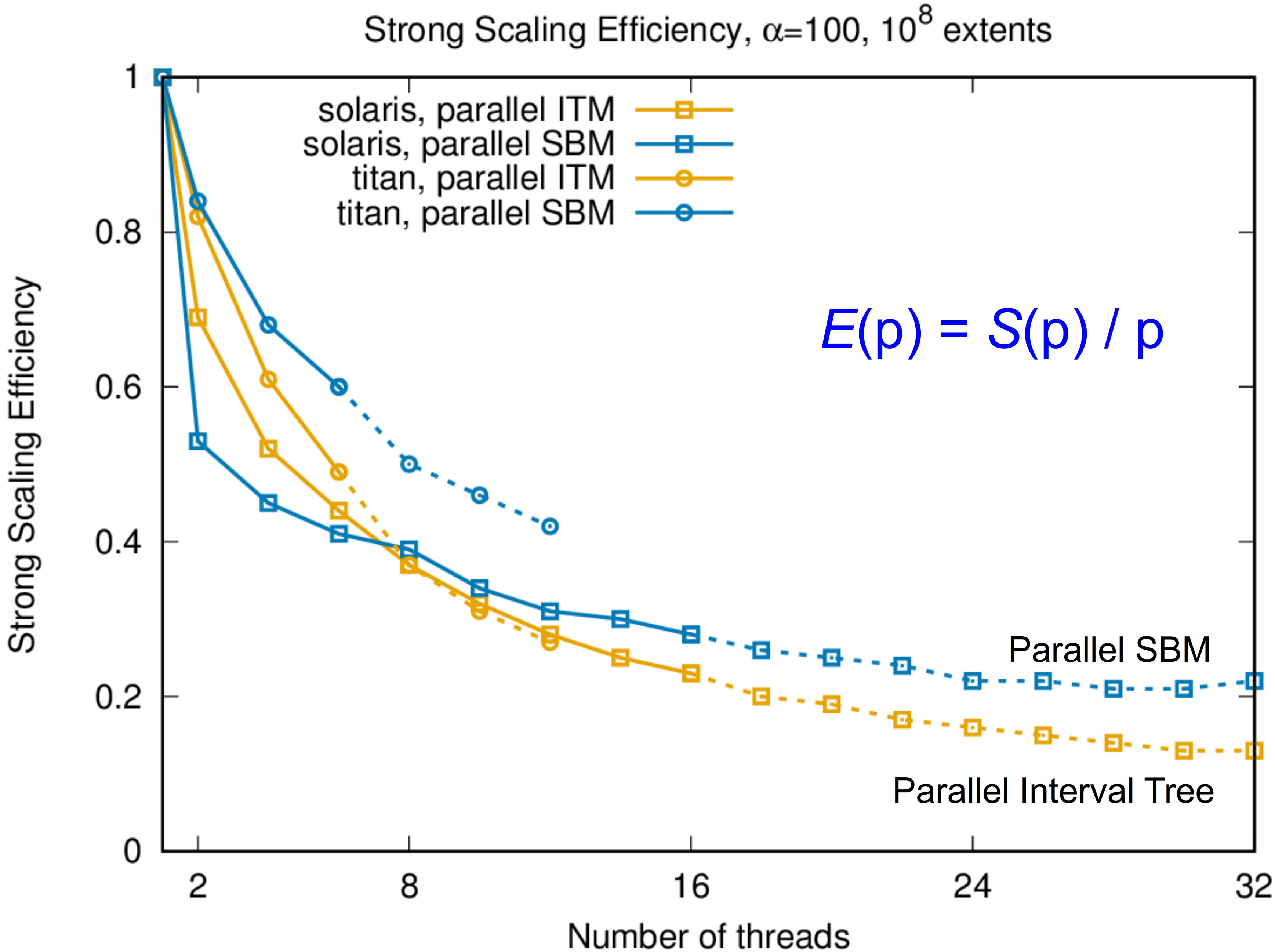


Speedup

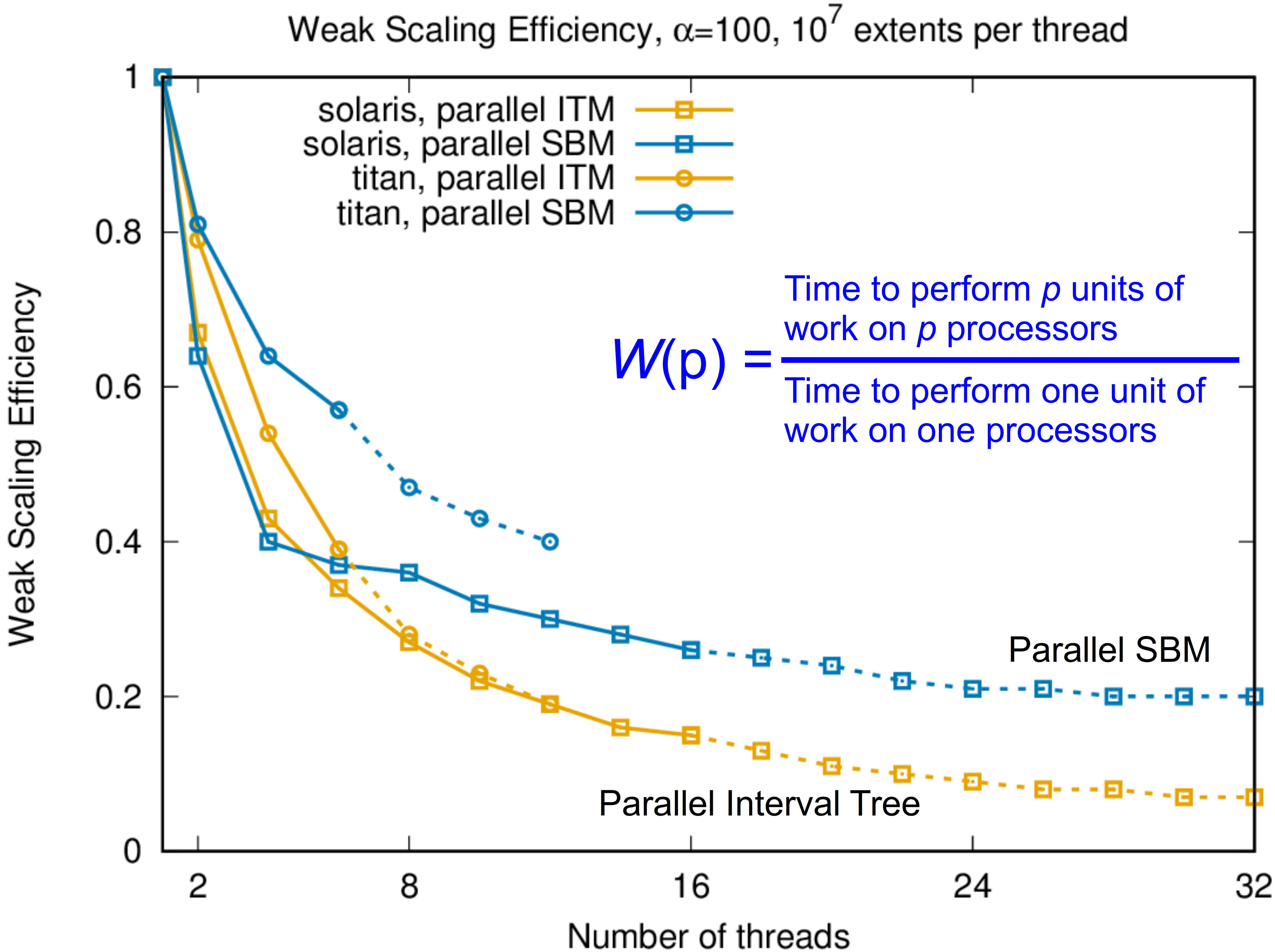


Speedup, α=100, 10^8 extents
solaris, parallel ITM
solaris, parallel SBM
titan, parallel ITM
titan, parallel SBM
Speedup
20
15
10
5
2
8
16
24
32
Number of threads
Parallel SBM
Parallel Interval Tree

Strong Scaling Efficiency



Weak Scaling Efficiency



$$W(p) = \frac{\text{Time to perform } p \text{ units of work on } p \text{ processors}}{\text{Time to perform one unit of work on one processors}}$$

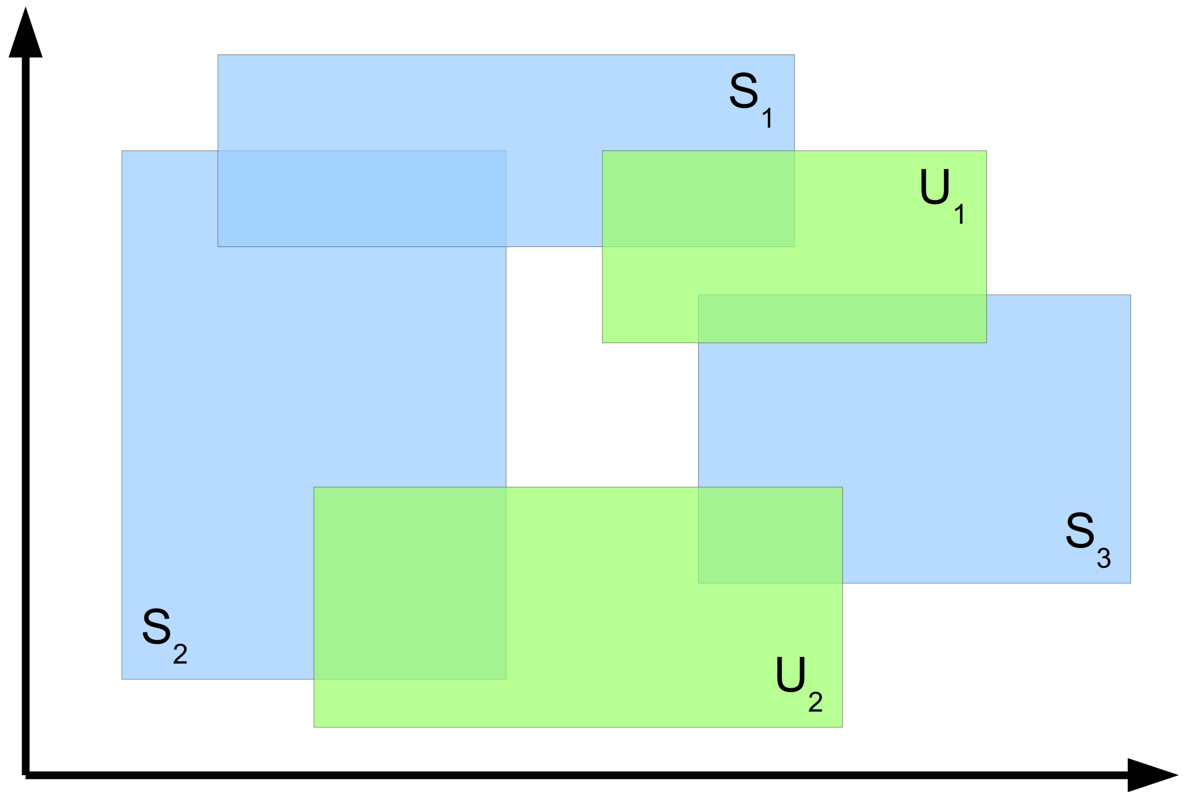
Conclusions

- Parallel SBM improves the already fast SBM algorithm
  - Can take advantage of modern multicore processors
- The speedup is limited by several factors
  - The parallel sorting phase
  - Intrinsecally serial regions
  - The baseline is very fast!
- Future works
  - Improve scaling efficiency
  - Extend the parallel SBM algorithm to cope with moving regions
  - Implement parallel SBM on the GPU (???)

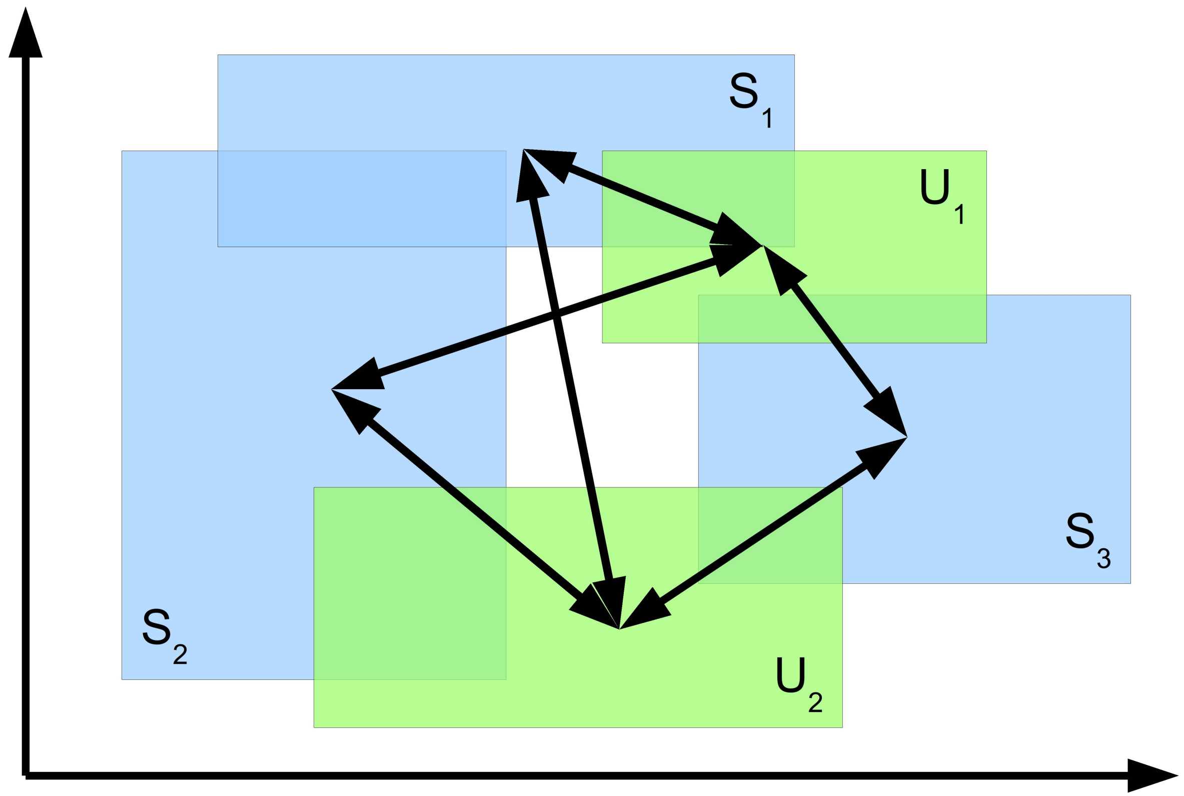
Thanks for your attention

Questions?

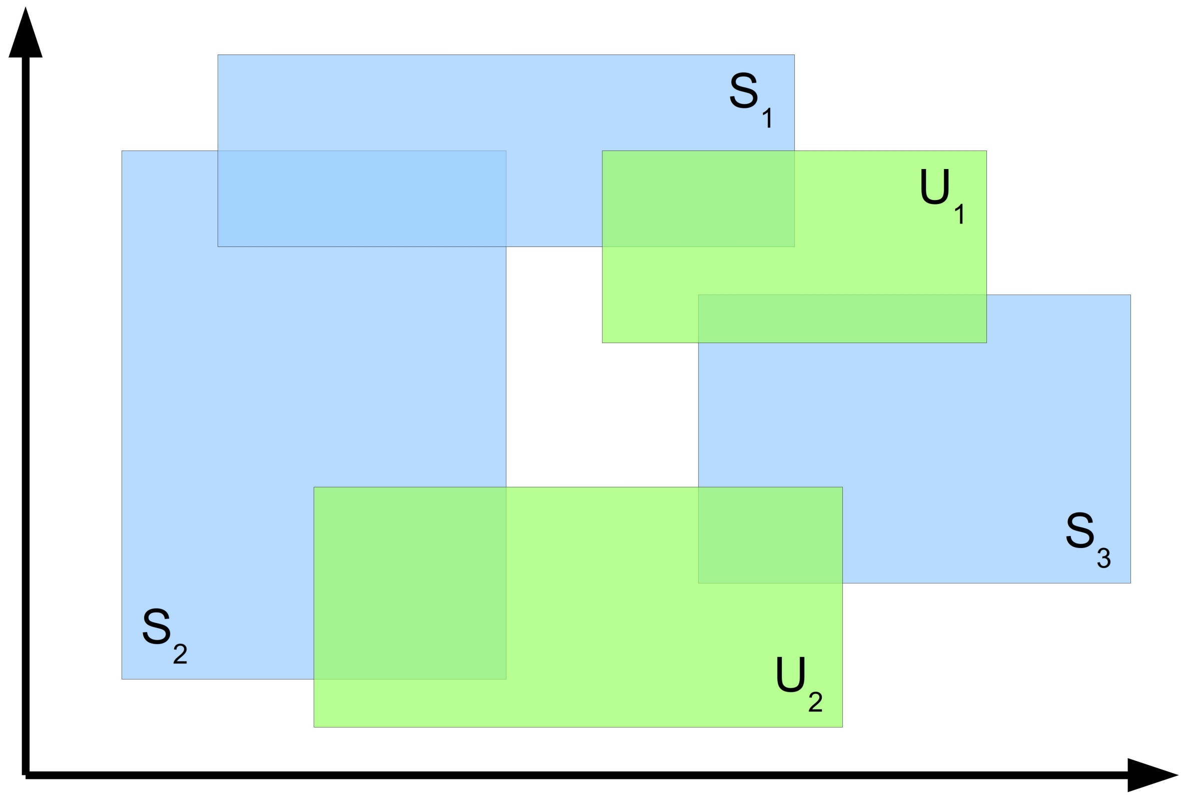
Brute-Force Matching



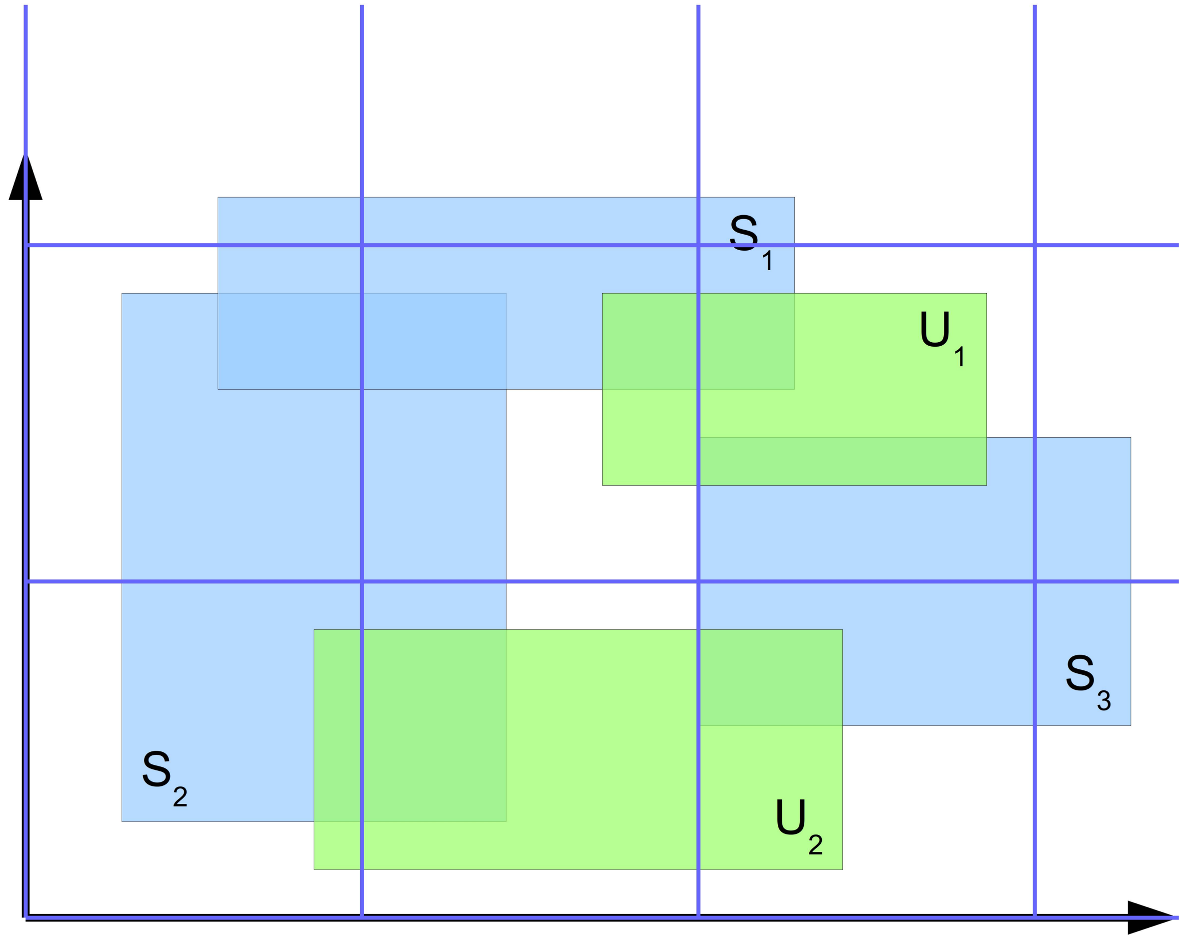
Brute-Force Matching



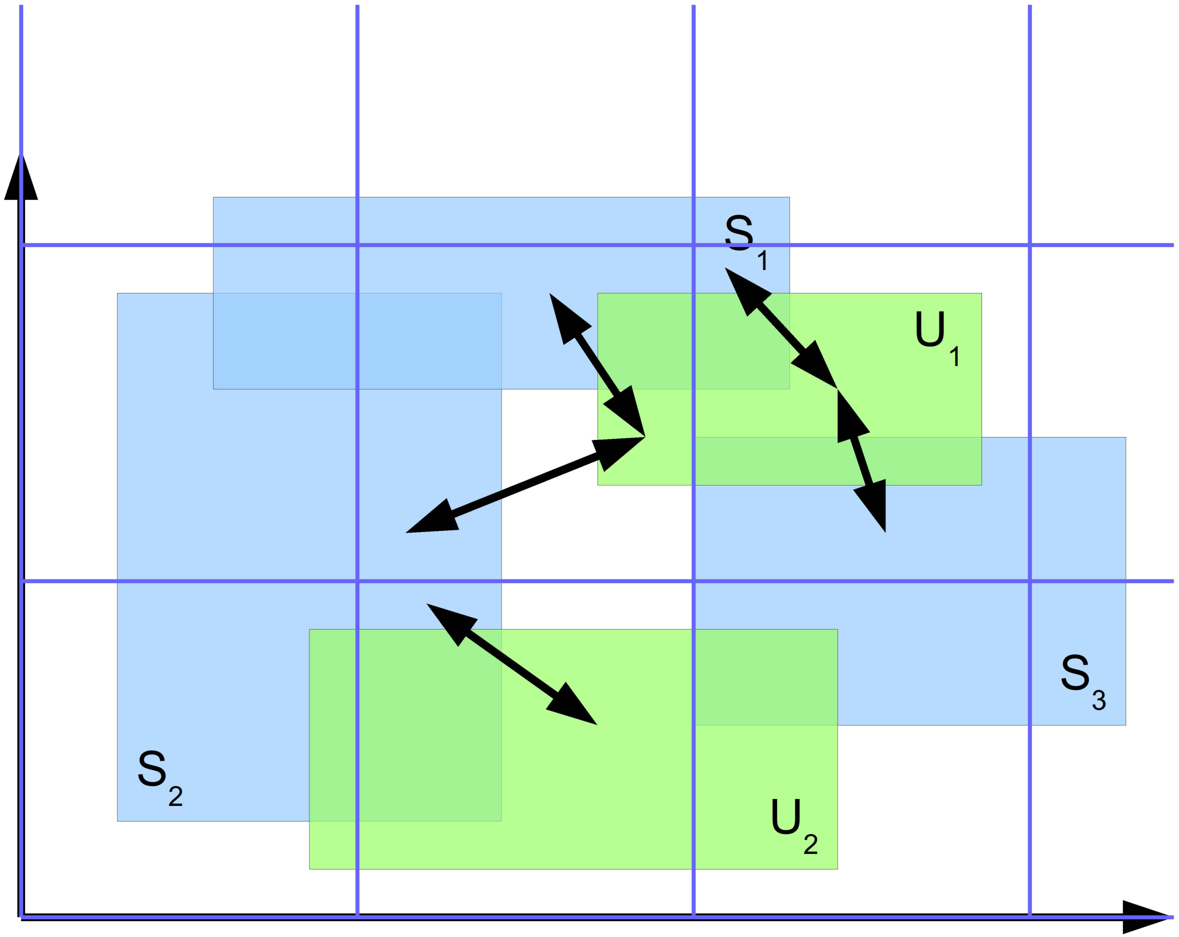
Grid-Based Matching



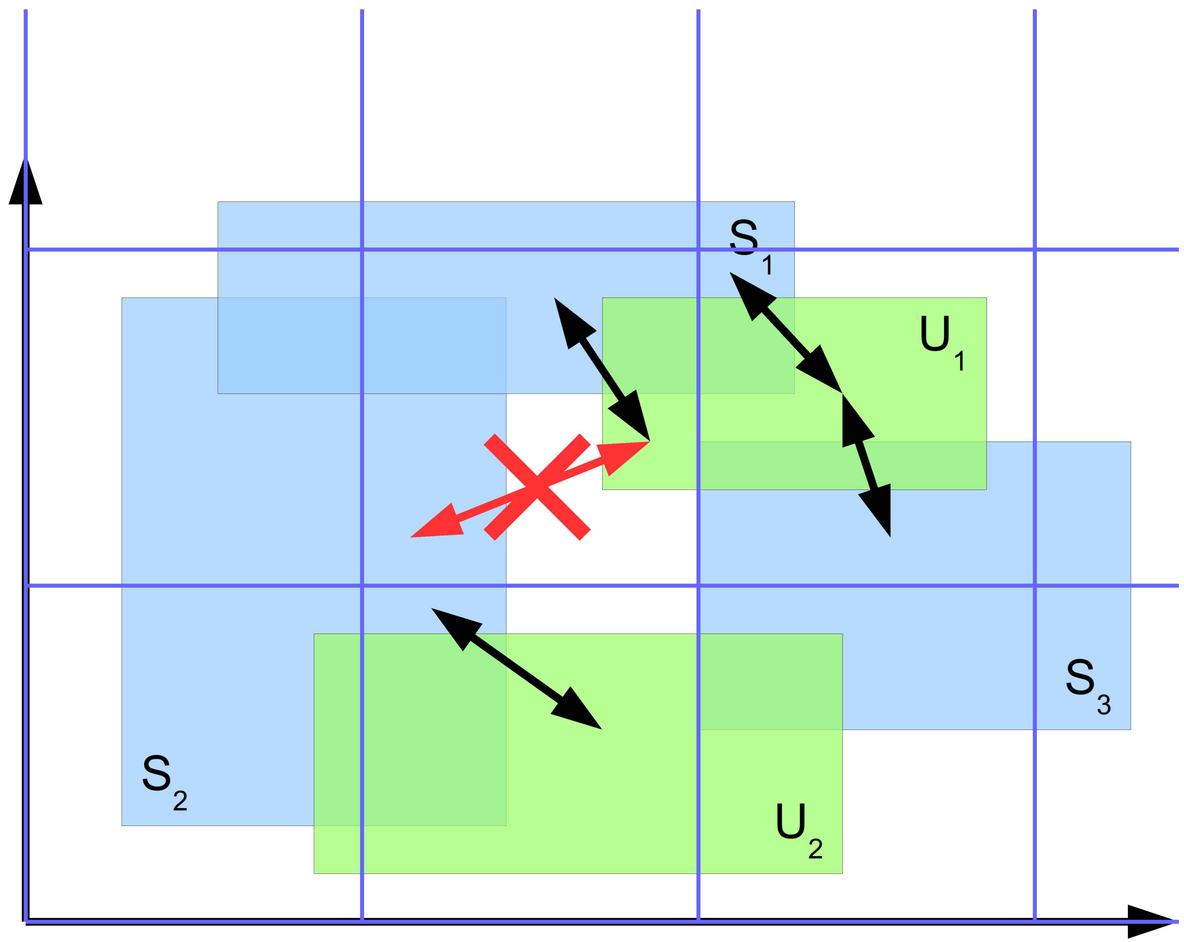
Grid-Based Matching



Grid-Based Matching



Grid-Based Matching



Interval-Tree Matching

- Based on the Interval Tree data structure
- Solves the 1D matching problem
- Subscription (Update) intervals are stored in the leaves of an Interval Tree
  - Balanced Search Tree
  - Internal nodes are augmented with auxiliary data used to steer queries towards overlapping intervals
- Intersections can be identified with a tree visit for each Update (Subscription) interval

Moreno Marzolla, Gabriele D'Angelo, Marco Mandrioli, *A Parallel Data Distribution Management Algorithm*, proc. DS-RT 2013, http://dx.doi.org/10.1109/DS-RT.2013.23

Interval Tree