UPPSALA UNIVERSITET

Geilo Winter School 2008

Henrik Löf
Uppsala Universitet

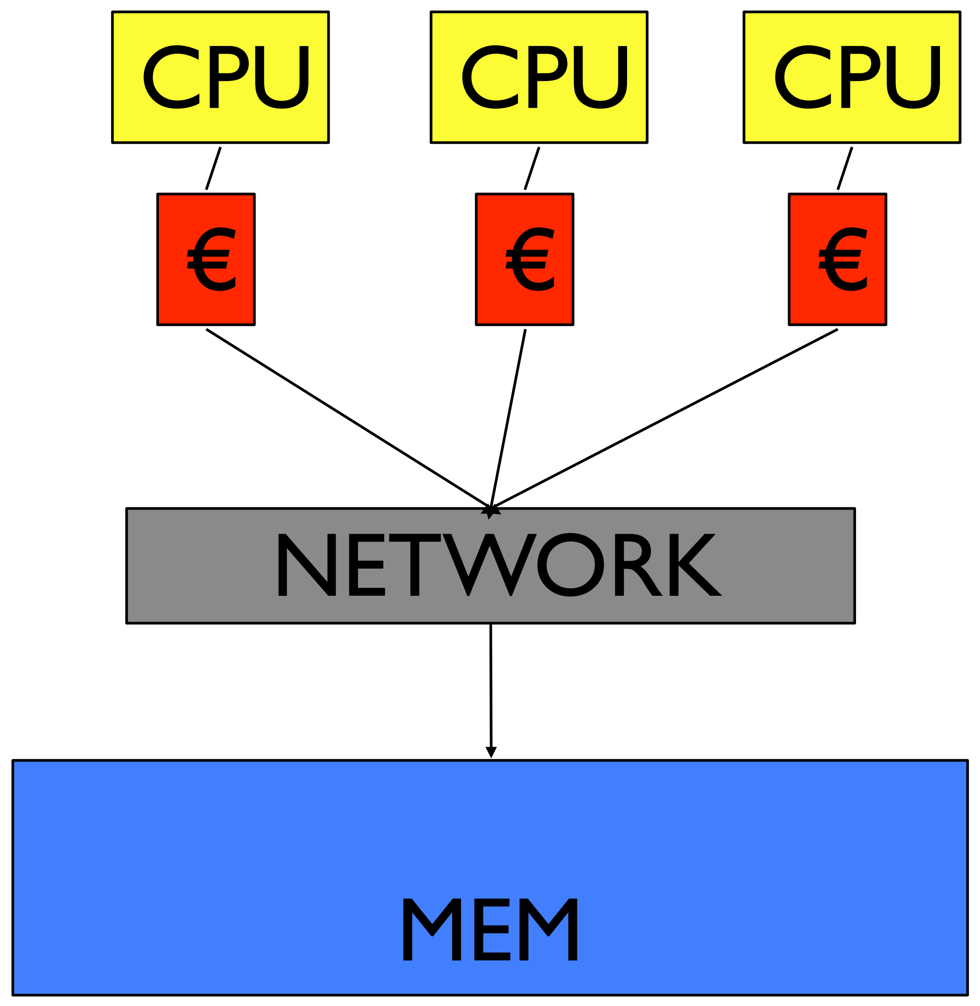
UPPSALA UNIVERSITET

Optimizing parallel applications

- Low latencies (locality of reference)
  - Cache memories
  - Remote Accesses (NUMA)
- Low parallel overhead
  - Minimize communication and synchronization
- Load Balance
  - Partitioning
  - Each thread has an equal amount of work

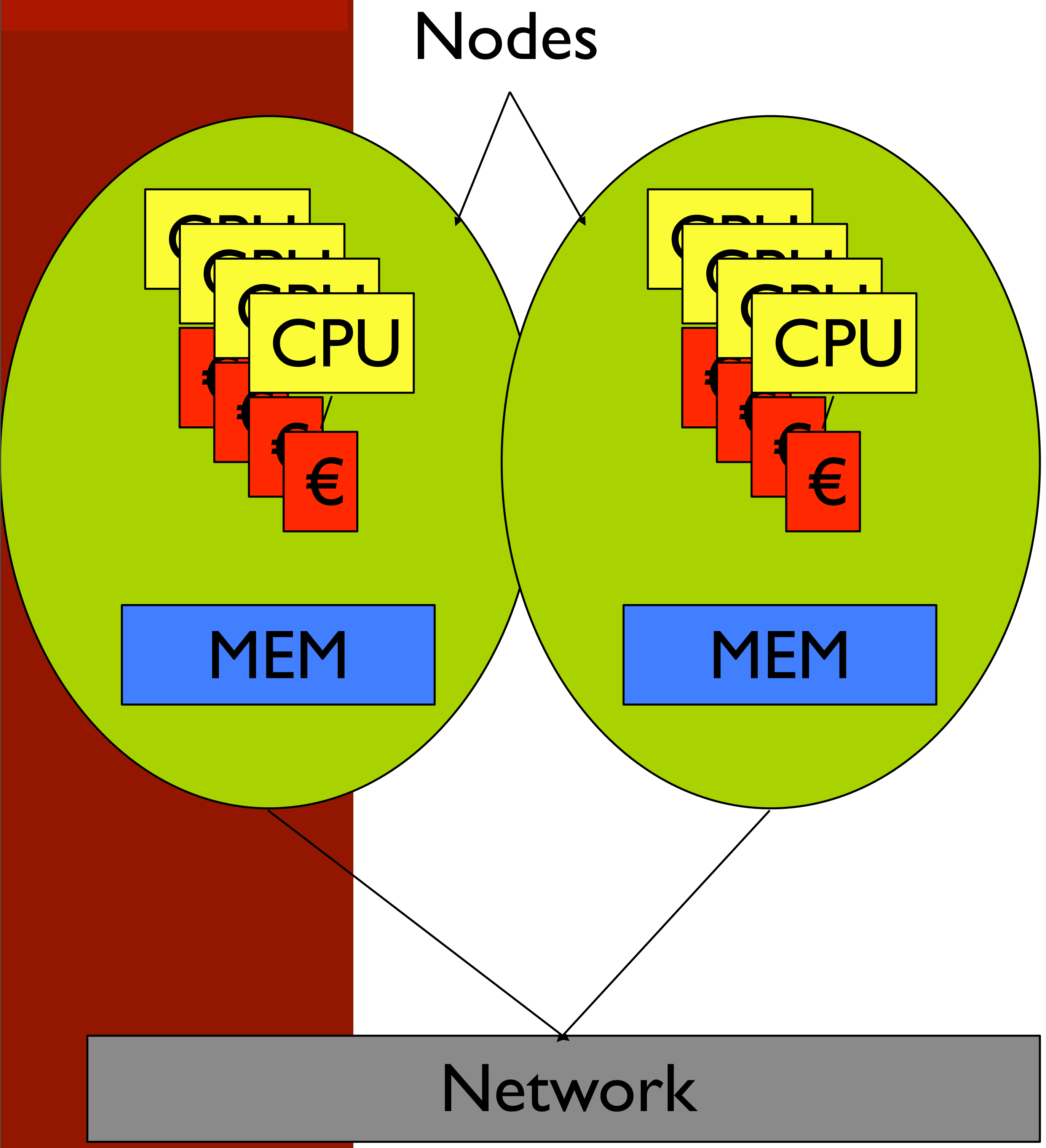
UPPSALA UNIVERSITET

Uniform Memory Access (UMA)



- Server
  - Few dozen CPU/cores
  - Easy to administrate
- Volume product
- Uniform access time
  - "Easy" to understand
- Examples
  - SMP
  - CMP
- Scalability problems
  - Limited interconnect bandwidth

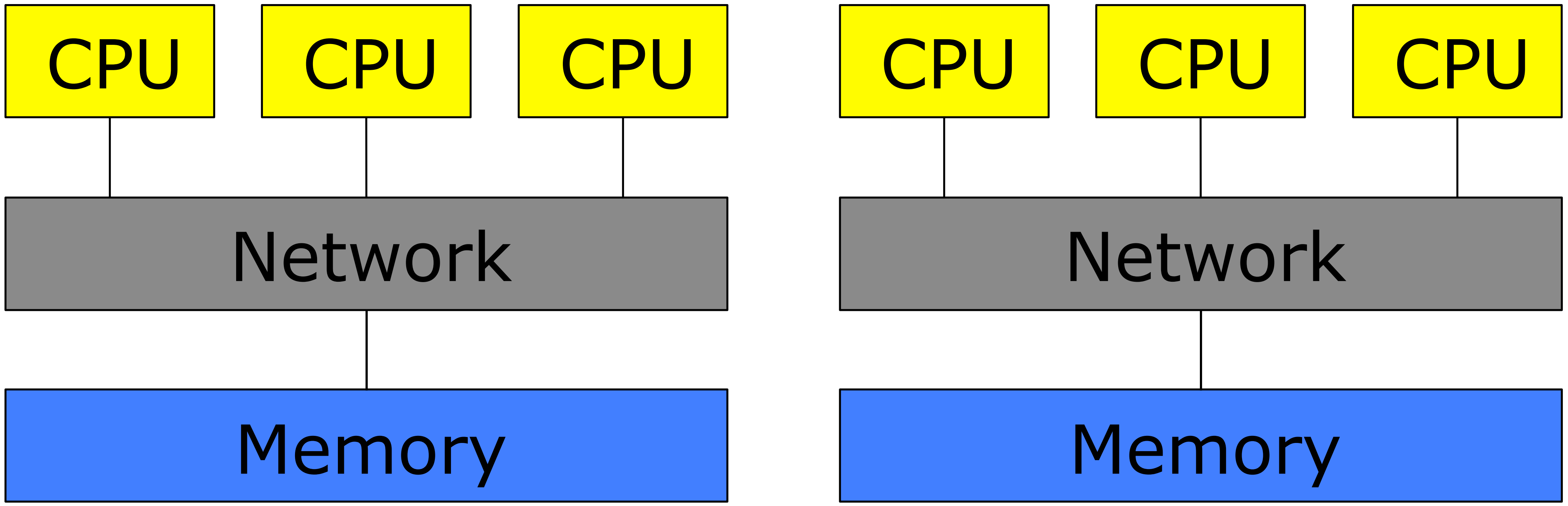
Non-Unifrom Memory Access (NUMA)



- Physically Distributed Memory
  - Shared memory programming
- Cluster of UMA *nodes*
  - CPU boards
  - Multi socket CMP systems
- Better scalability
  - Maintaines simpler shared memory programming model
- Examples
  - Sun Fire 15K, 25K
  - SGI Origin/Altix
  - HyperTransport (AMD)
  - QuickPath (Intel)
- Expensive

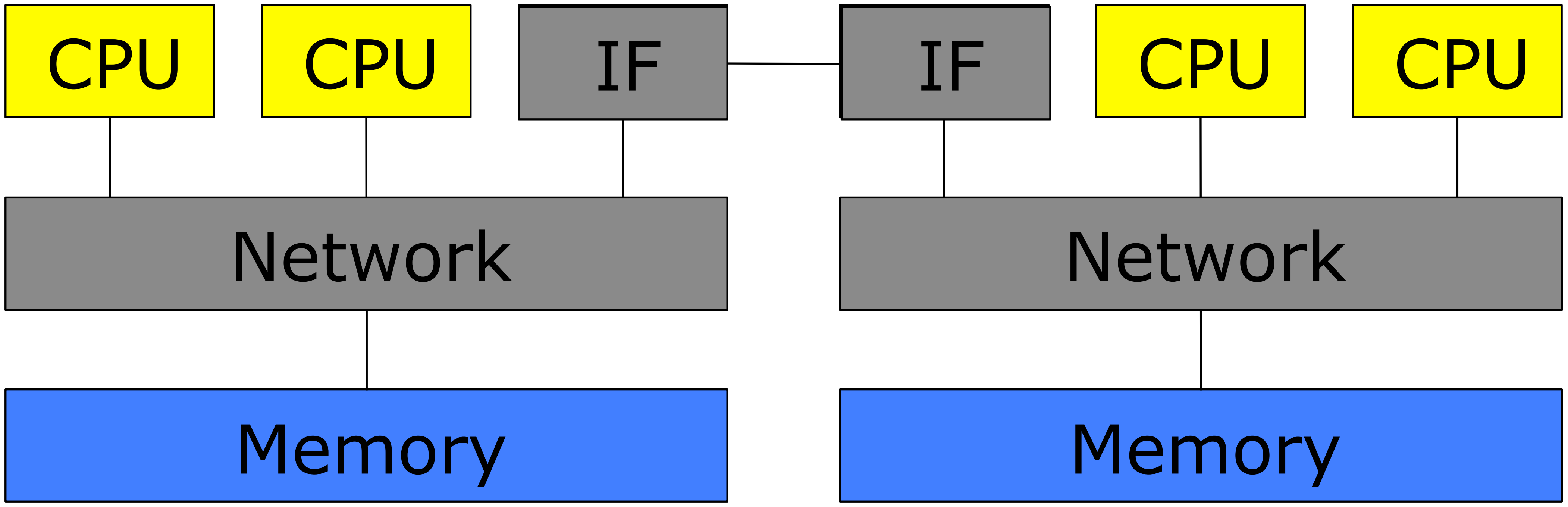
UPPSALA UNIVERSITET

Sun WildFire, a Distributed Shared Memory (DSM) System



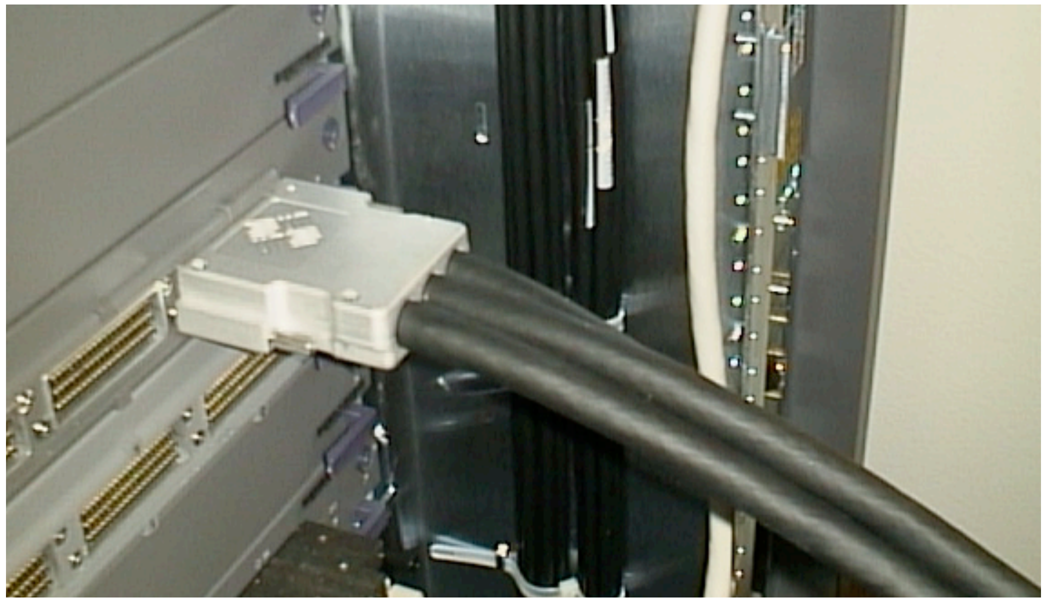
UPPSALA UNIVERSITET

Sun WildFire, a Distributed Shared Memory (DSM) System



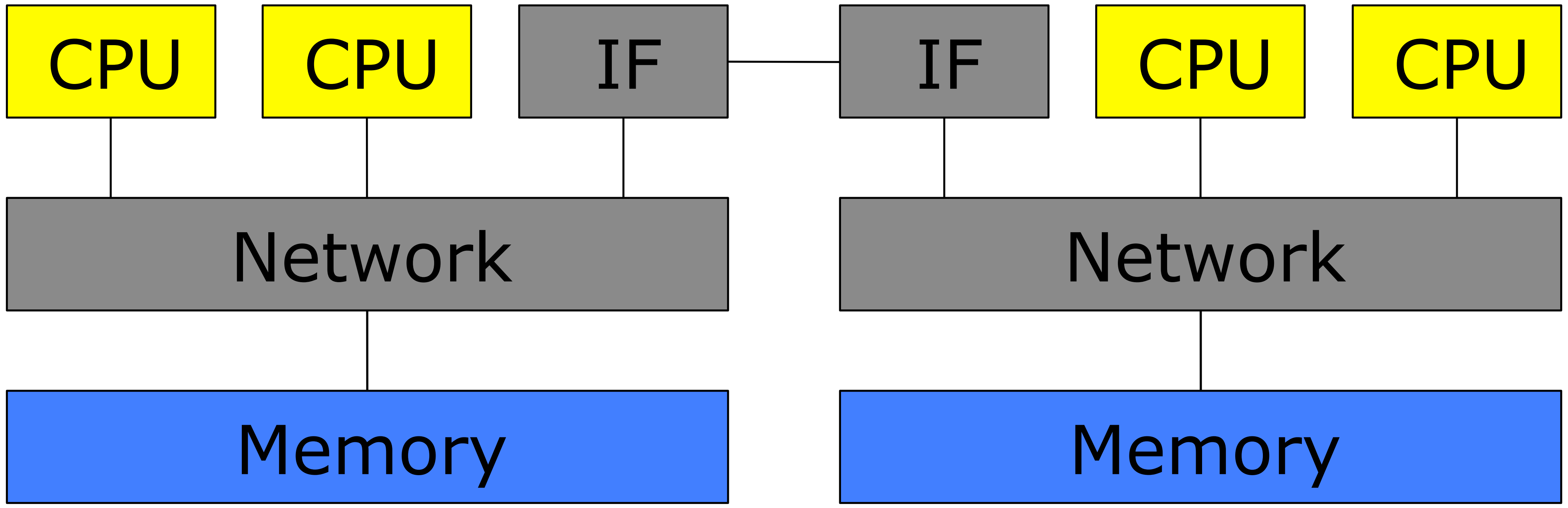
UPPSALA UNIVERSITET

Sun WildFire what it looks like



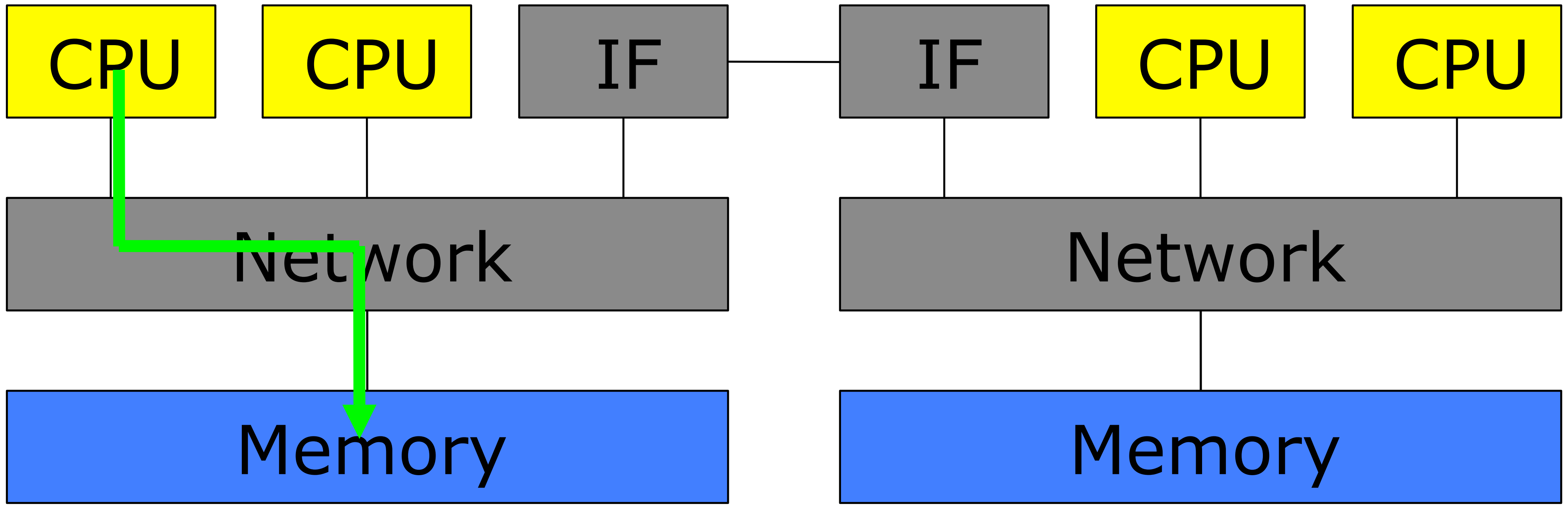
UPPSALA UNIVERSITET

Local access – fast!



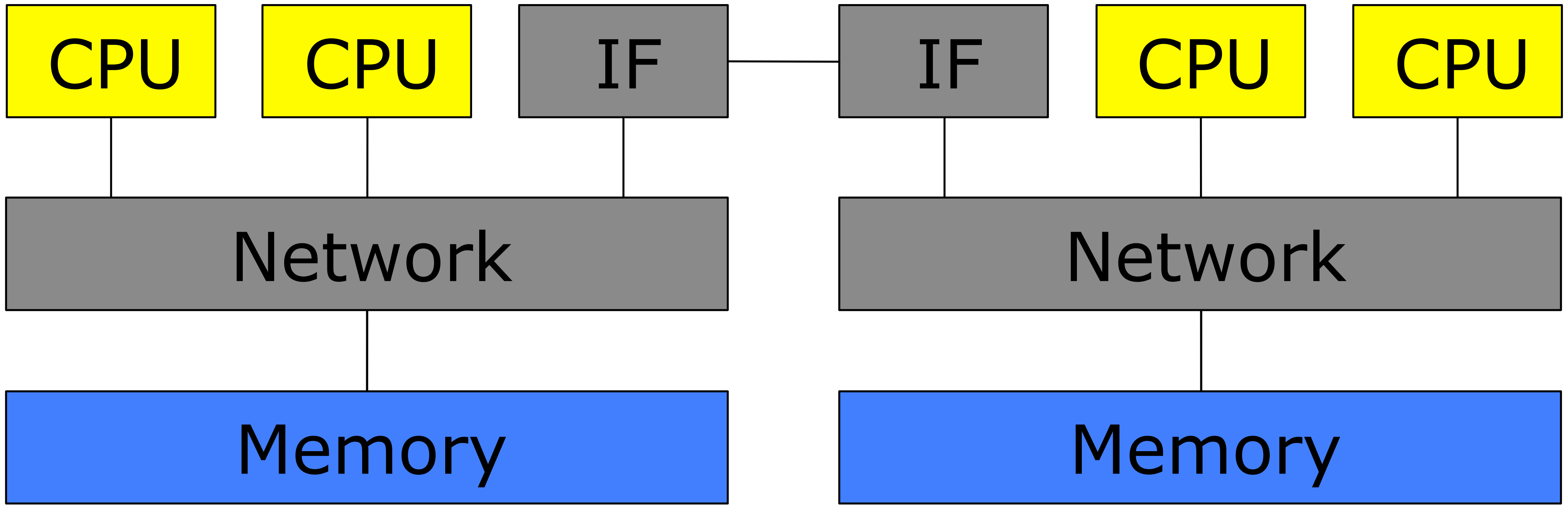
UPPSALA UNIVERSITET

Local access – fast!



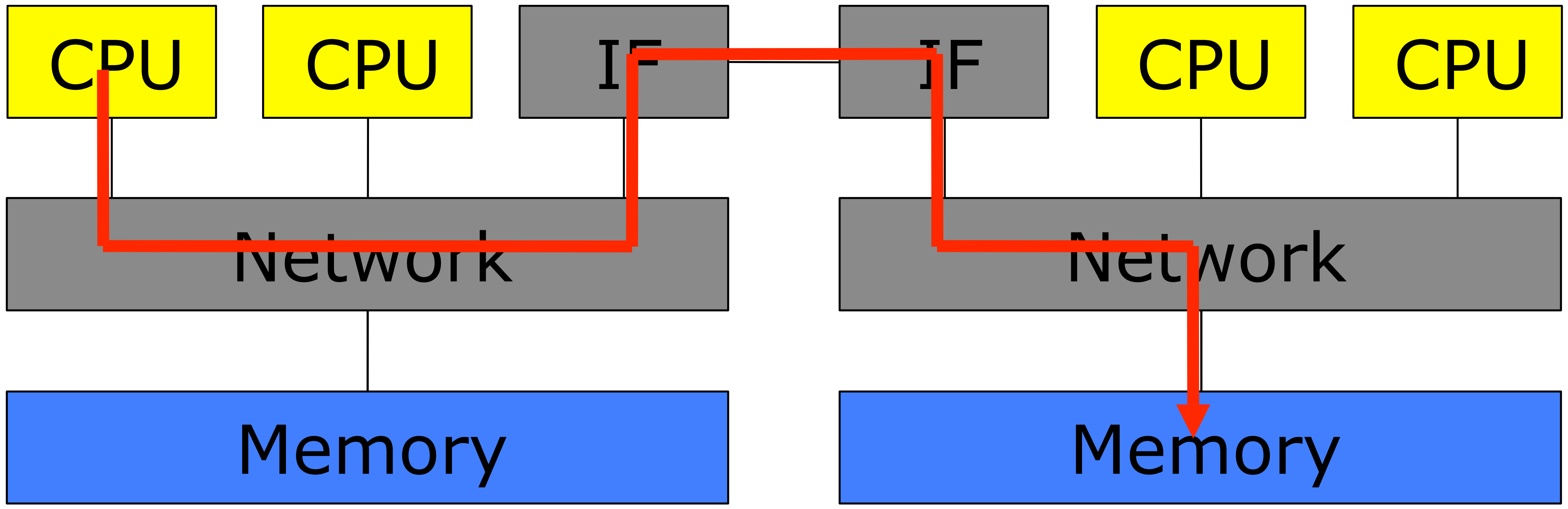
UPPSALA UNIVERSITET

Remote access – slower!



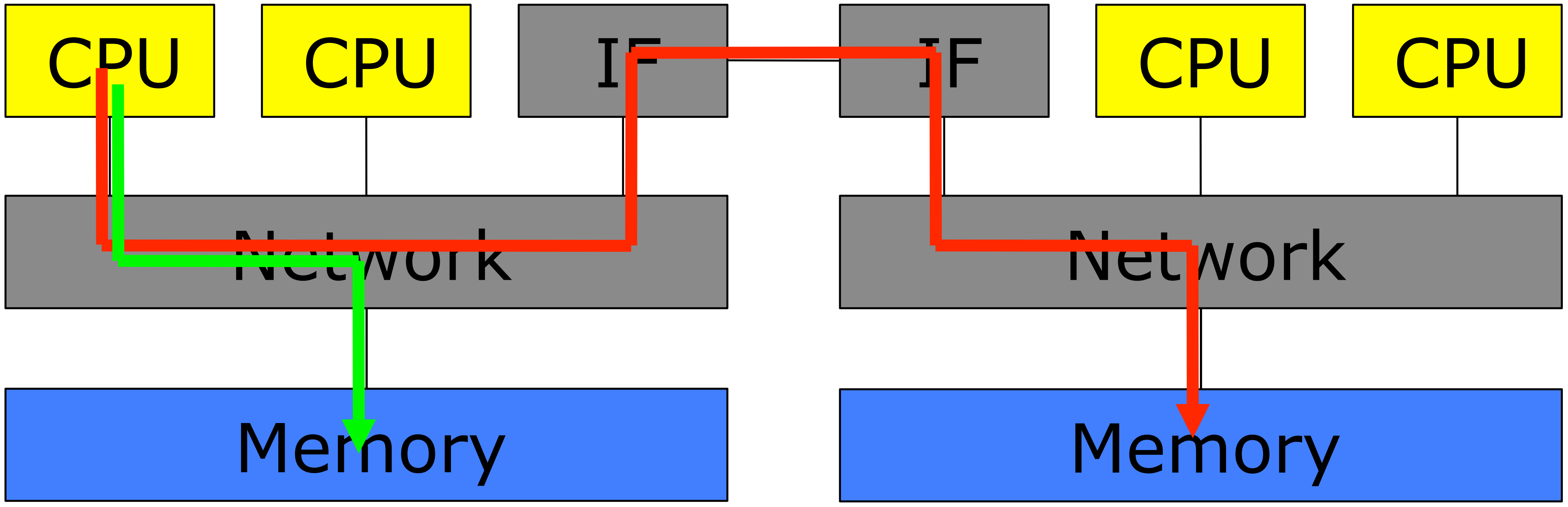
UPPSALA UNIVERSITET

Remote access – slower!



UPPSALA UNIVERSITET

NUMA-ratio



$$NUMA - ratio = \frac{T_{remote}}{T_{local}}$$

UPPSALA UNIVERSITET

Implementing Conjugate Gradients in OpenMP

Analyzing CG

```
Given an initial guess $x_0$,
store $r_0 = b - Ax_0$ and set $p_0 = r_0$.
do $k = 0,1,\ldots$
(1)    Store $Ap_k$
(2)    Store $\langle p_k, Ap_k \rangle$
(3)    $\alpha_k = \frac{\langle r_k, r_k \rangle}{\langle p_k, Ap_k \rangle}$
(4)    $x_{k+1} = x_k + \alpha_k p_k$
(5)    $r_{k+1} = r_k - \alpha_k A p_k$
(6)    Store $\langle r_{k+1}, r_{k+1} \rangle$
(7)    $\beta_k = \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}$
(8)    $p_{k+1} = r_{k+1} + \beta_k p_k$
```

UPPSALA UNIVERSITET

Analyzing CG

```
Given an initial guess $x_0$,
store $r_0 = b - Ax_0$ and set $p_0 = r_0$.
do $k = 0,1,\ldots$
(1)    Store $Ap_k$
(2)    Store $\langle p_k, Ap_k \rangle$
(3)    $\alpha_k = \frac{\langle r_k, r_k \rangle}{\langle p_k, Ap_k \rangle}$
(4)    $x_{k+1} = x_k + \alpha_k p_k$
(5)    $r_{k+1} = r_k - \alpha_k Ap_k$
(6)    Store $\langle r_{k+1}, r_{k+1} \rangle$
(7)    $\beta_k = \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}$
(8)    $p_{k+1} = r_{k+1} + \beta_k p_k$
```

3 Vector Ops.

Analyzing CG

Given an initial guess x_0,

store $r_0 = b - Ax_0$ and set $p_0 = r_0$.

do $k = 0,1,K$

(1) Store Ap_k

(2) Store $\langle p_k, Ap_k \rangle$

(3) $\alpha_k = \dfrac{\langle r_k, r_k \rangle}{\langle p_k, Ap_k \rangle}$

(4) $x_{k+1} = x_k + \alpha_k p_k$

(5) $r_{k+1} = r_k - \alpha_k A p_k$

(6) Store $\langle r_{k+1}, r_{k+1} \rangle$

(7) $\beta_k = \dfrac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}$

(8) $p_{k+1} = r_{k+1} + \beta_k p_k$

2 Inner Products

3 Vector Ops.

UPPSALA UNIVERSITET

Analyzing CG

1 Sparse Matrix-Vector Product (SpMxV)

2 Inner Products

3 Vector Ops.

```
Given an initial guess $x_0$,
store $r_0 = b - Ax_0$ and set $p_0 = r_0$.
do $k = 0,1,\mathrm{K}$
    (1)  Store $Ap_k$
    (2)  Store $\langle p_k, Ap_k \rangle$
    (3)  $\alpha_k = \dfrac{\langle r_k, r_k \rangle}{\langle p_k, Ap_k \rangle}$
    (4)  $x_{k+1} = x_k + \alpha_k p_k$
    (5)  $r_{k+1} = r_k - \alpha_k Ap_k$
    (6)  Store $\langle r_{k+1}, r_{k+1} \rangle$
    (7)  $\beta_k = \dfrac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}$
    (8)  $p_{k+1} = r_{k+1} + \beta_k p_k$
```

UPPSALA UNIVERSITET

Basic parallelization of CG

UPPSALA UNIVERSITET

Basic parallelization of CG

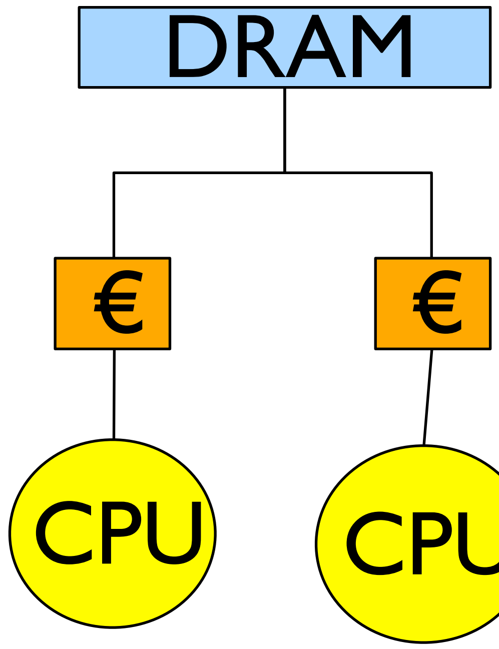
```
 cg_iter_count = 1

do

   call SpMxV(A,p,temp)


   pAp_norm = 0.0_rfp



   do i = 1, matrix_size
      pAp_norm = pAp_norm + p(i)*temp(i)
   end do




   alpha = r_old_norm/pAp_norm



   x(:) = x(:) + alpha * p(:)
   r_new(:) = r_old(:) - alpha * temp(:)

                  .

                  .
```

Basic parallelization of CG

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(cg_iter_count)

   cg_iter_count = 1

  do

      call SpMxV(A,p,temp)

!$OMP SINGLE
      pAp_norm = 0.0_rfp
!$OMP END SINGLE
=====================================
!$OMP DO REDUCTION(+:pAp_norm)
      do i = 1, matrix_size
         pAp_norm = pAp_norm + p(i)*temp(i)
      end do
!$OMP END DO
=====================================

!$OMP SINGLE
      alpha = r_old_norm/pAp_norm
!$OMP END SINGLE
=====================================
!$OMP WORKSHARE
      x(:) = x(:) + alpha * p(:)
      r_new(:) = r_old(:) - alpha * temp(:)
!$OMP END WORKSHARE NOWAIT
```

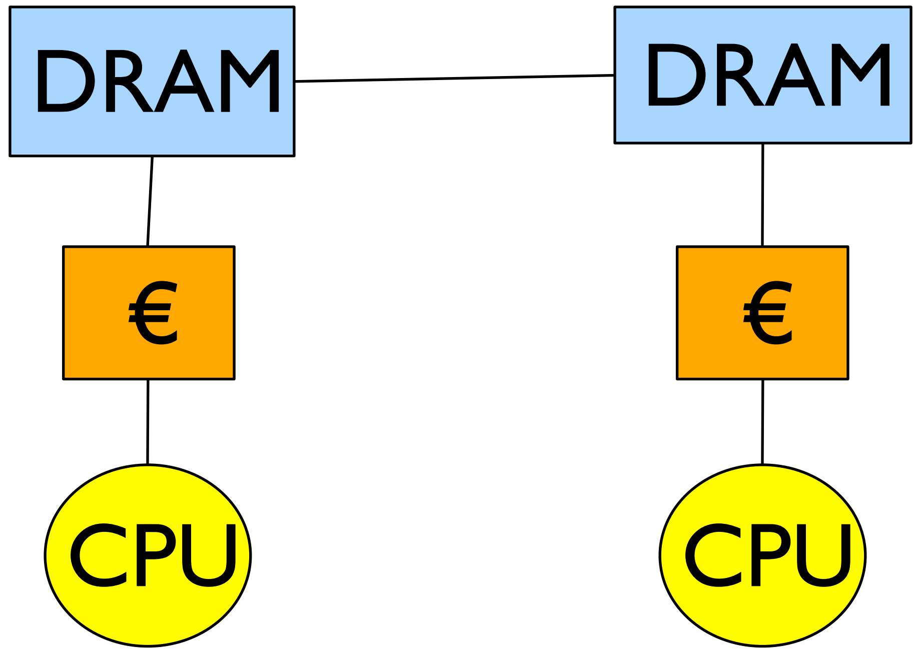
Parallel CG

Sun E10K



400MHz US-II + UMA

Sun SF15K

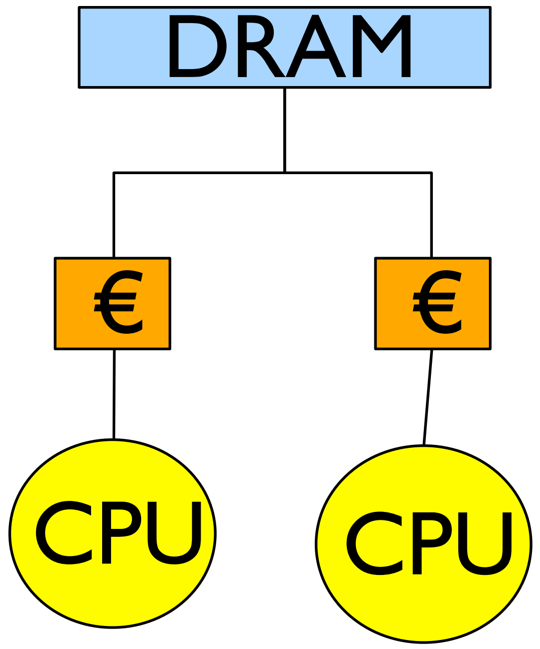


900MHz US-III + NUMA

Parallel CG

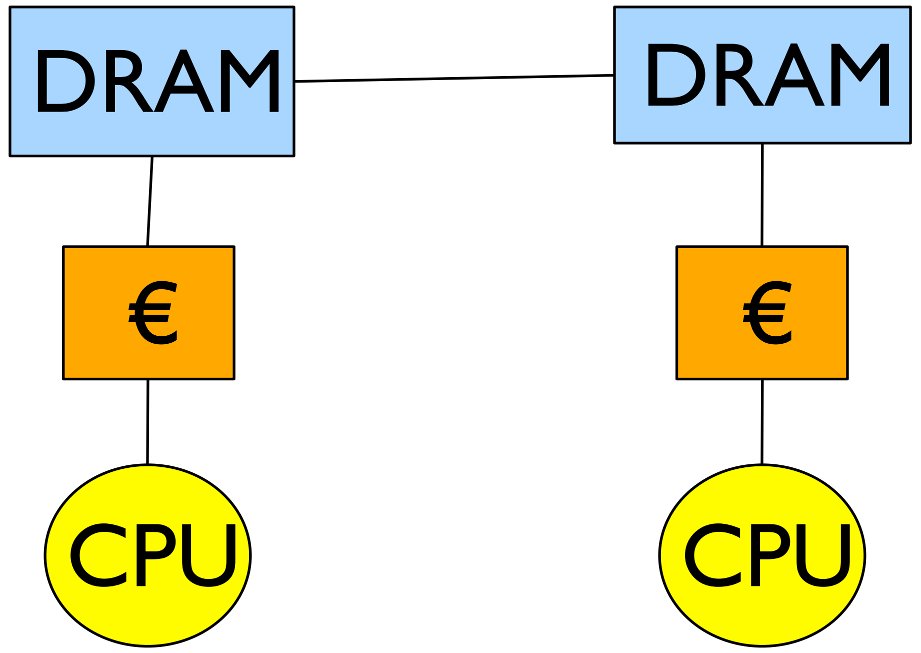
Serial

Sun E10K



400MHz US-II + UMA

Sun SF15K

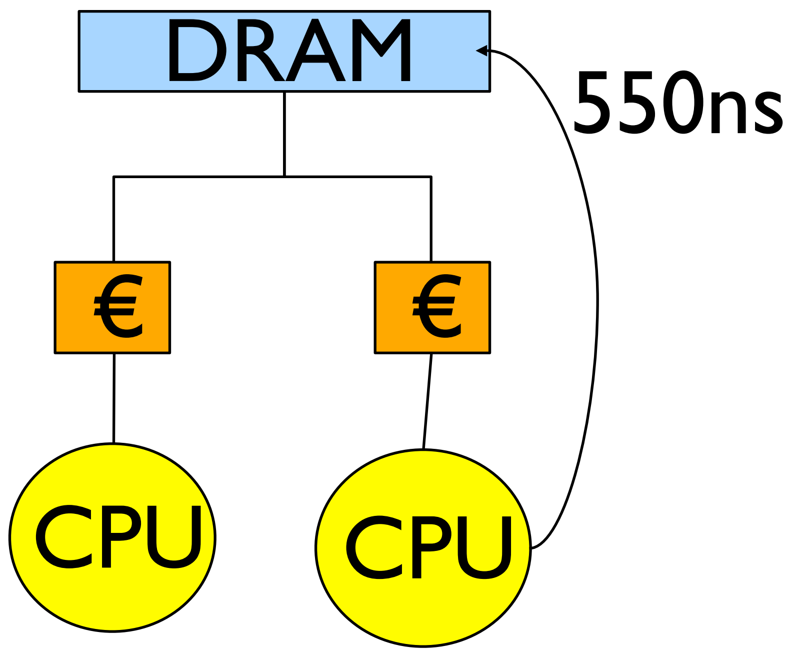


900MHz US-III + NUMA

Parallel CG

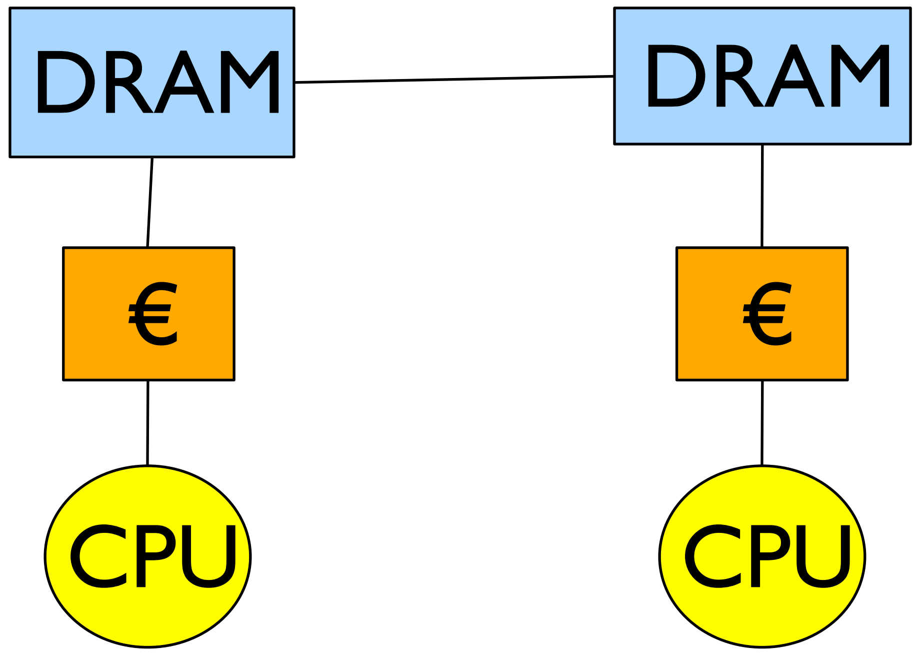
Serial 220s

Sun E10K



400MHz US-II + UMA

Sun SF15K



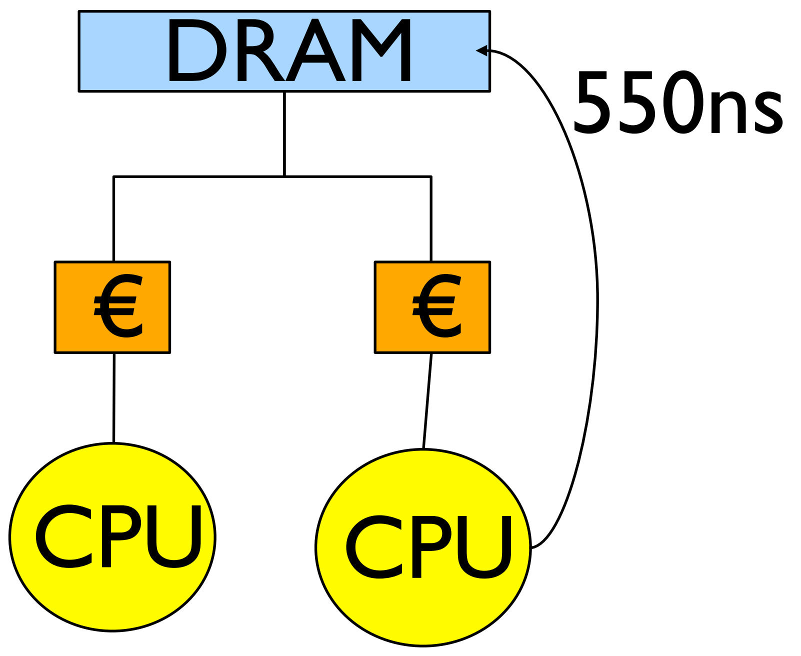
900MHz US-III + NUMA

Parallel CG

Serial 220s

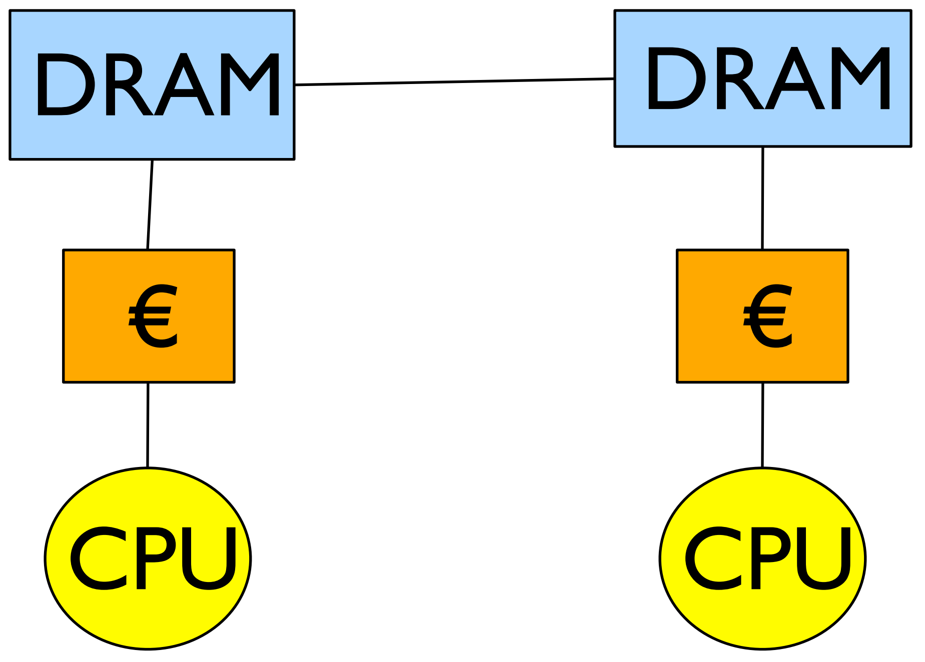
Serial

Sun E10K



400MHz US-II + UMA

Sun SF15K



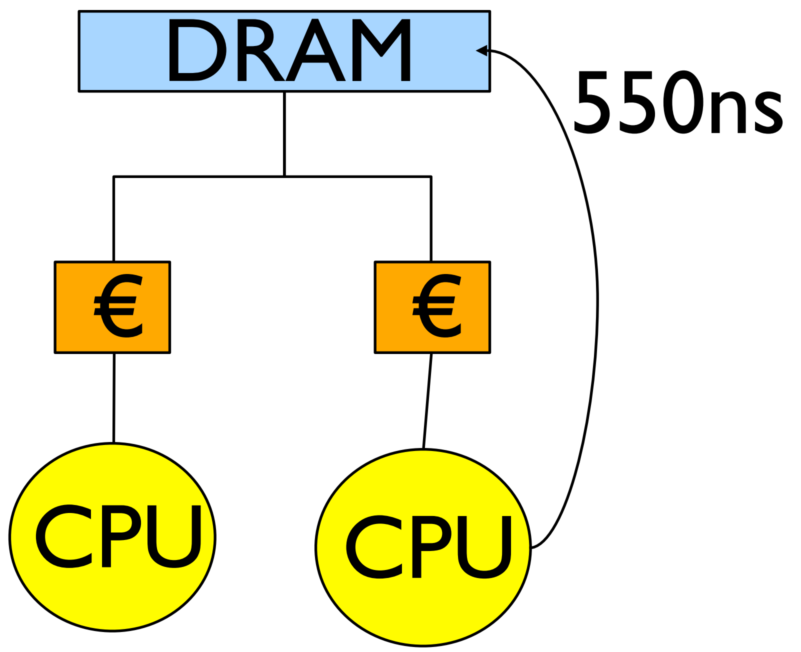
900MHz US-III + NUMA

Parallel CG

Serial 220s

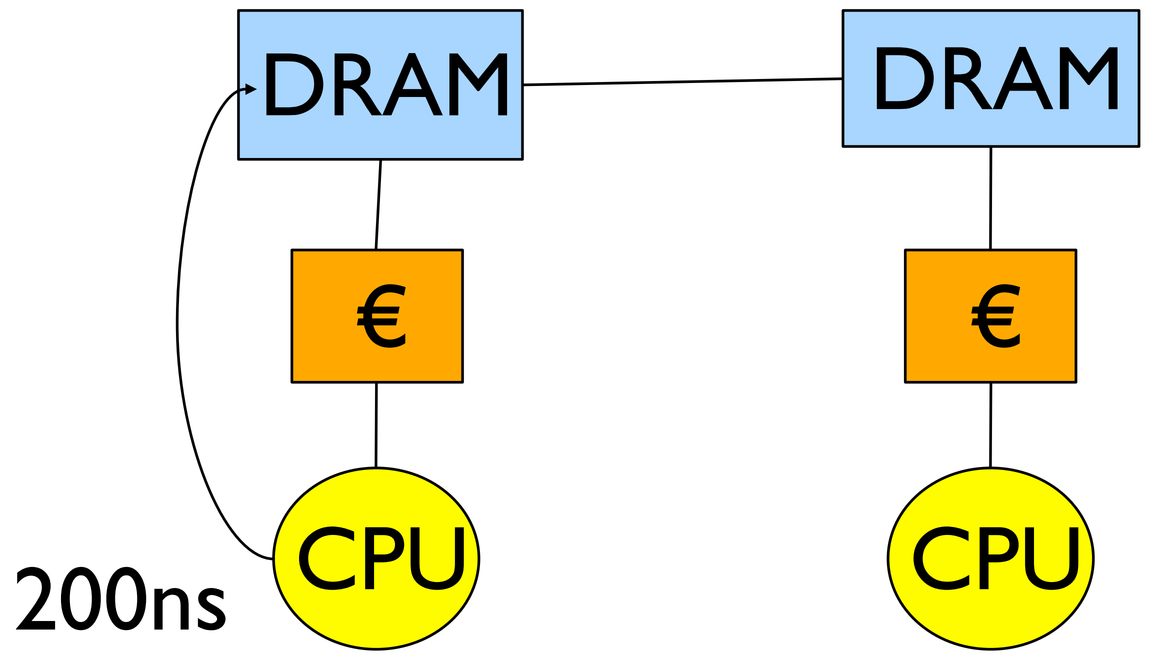
45s Serial

Sun E10K



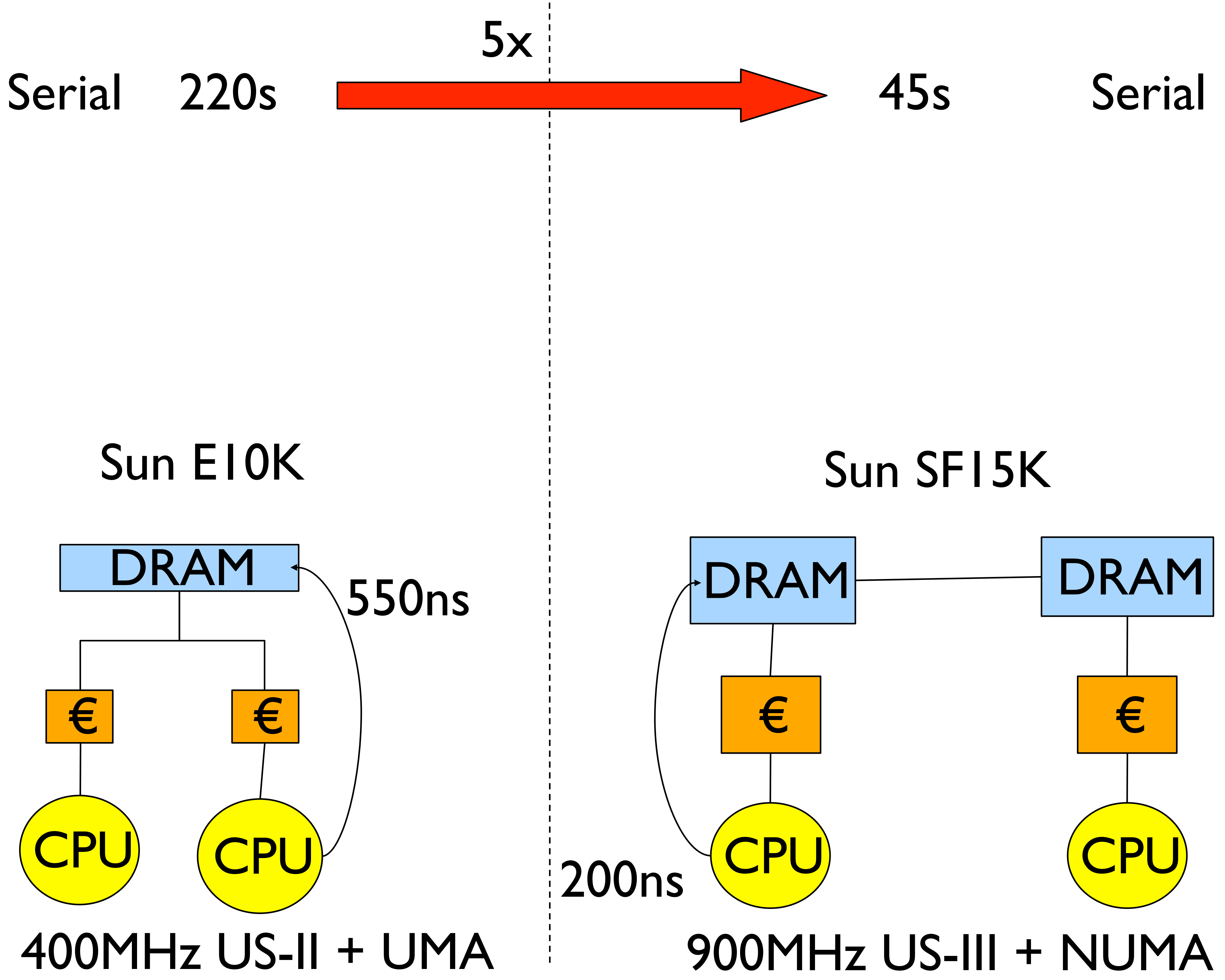
400MHz US-II + UMA

Sun SF15K



900MHz US-III + NUMA

Parallel CG

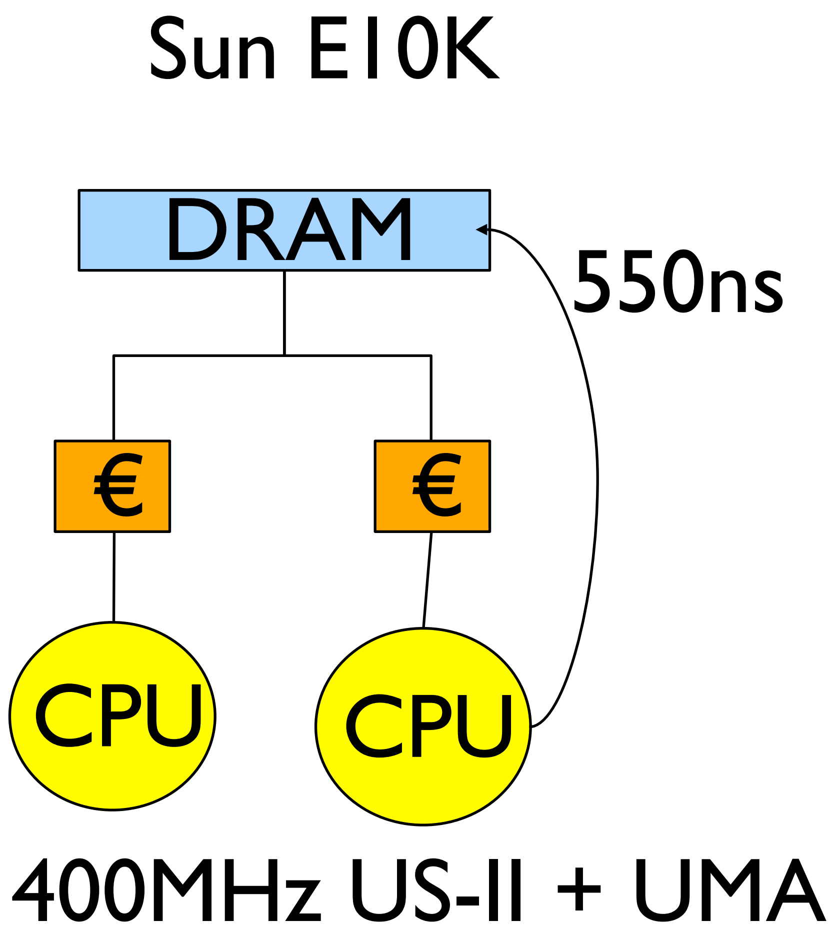


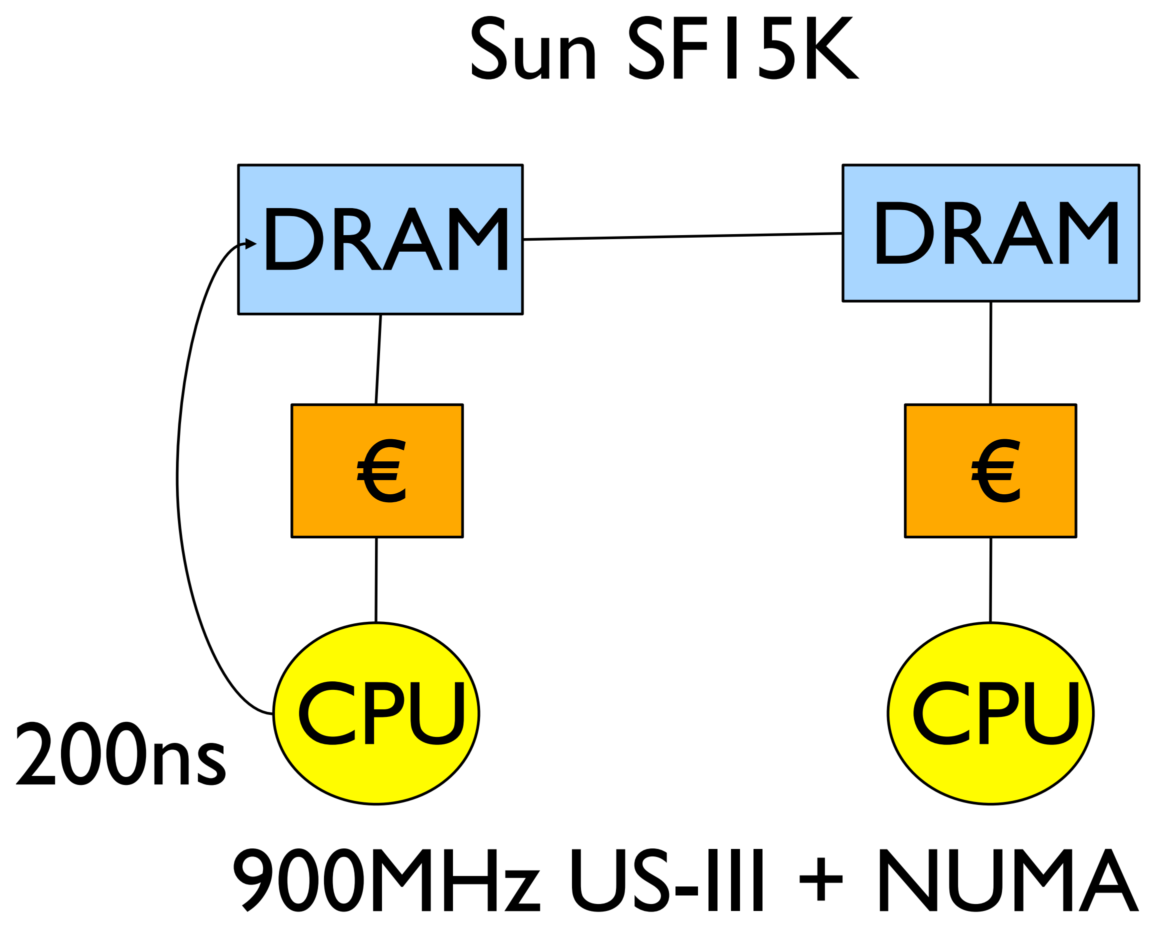
5x
Serial
220s
45s
Serial
Sun E10K
DRAM
550ns
€
€
CPU
CPU
400MHz US-II + UMA
Sun SF15K
DRAM
DRAM
€
€
CPU
CPU
200ns
900MHz US-III + NUMA

Parallel CG

Serial 220s → 5x → 45s Serial

Parallel,
28 threads



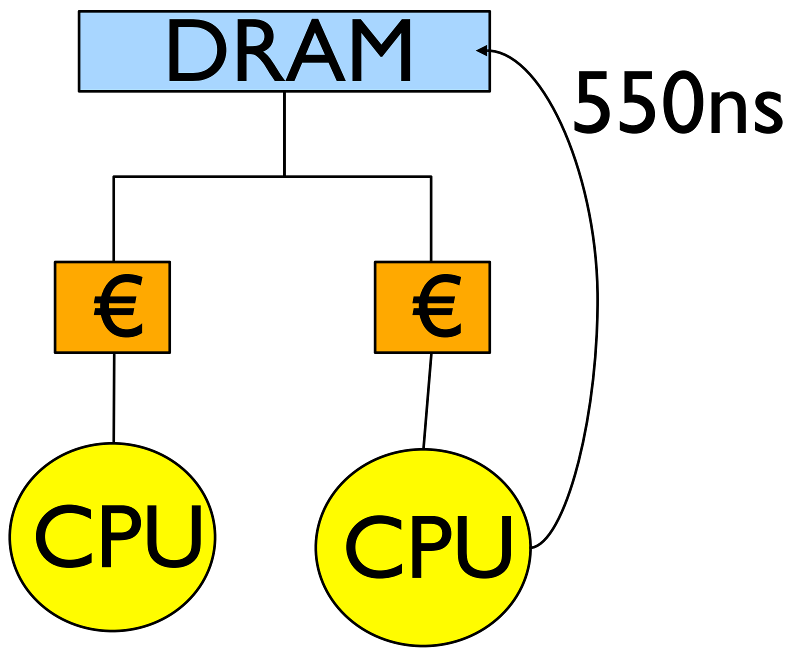


Parallel CG

Serial 220s → 5x → 45s Serial

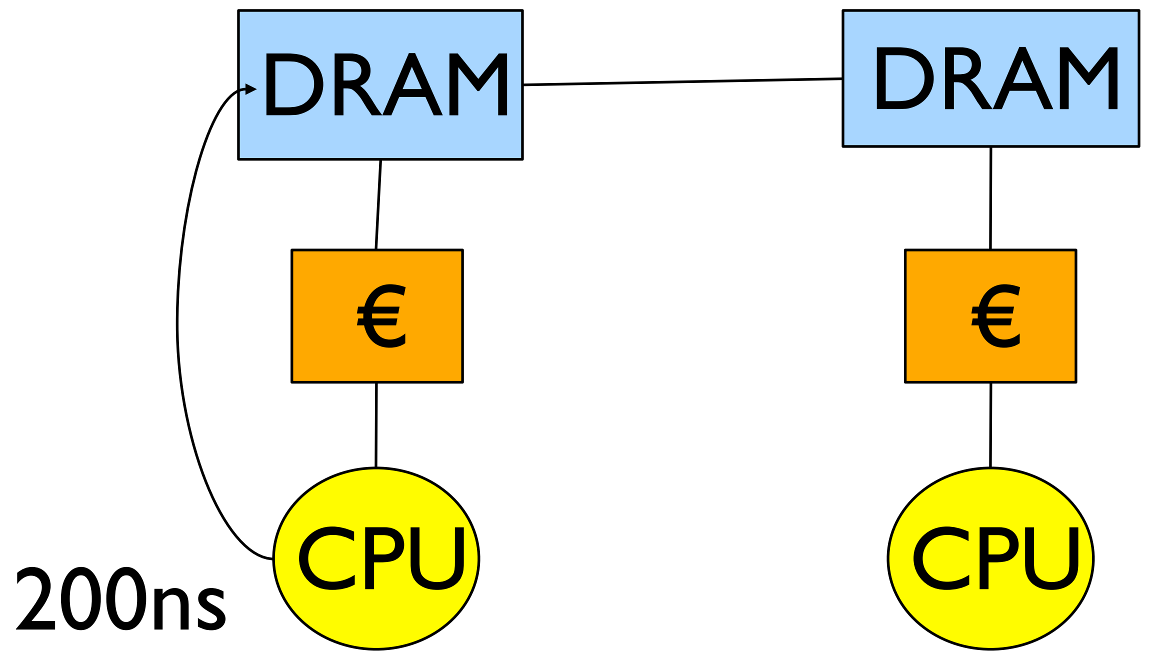
Parallel, 28 threads 8s

Sun E10K



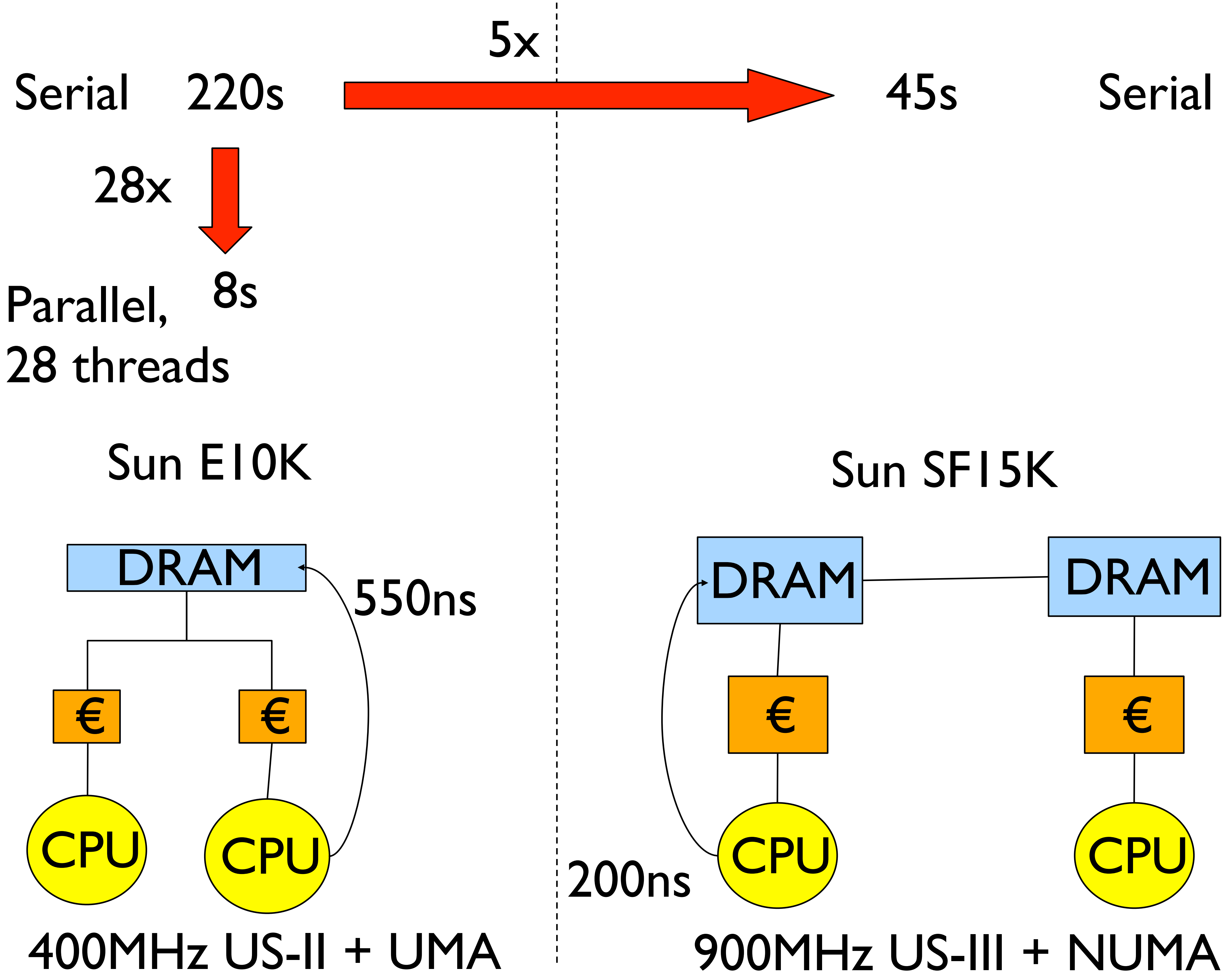
400MHz US-II + UMA

Sun SF15K



900MHz US-III + NUMA

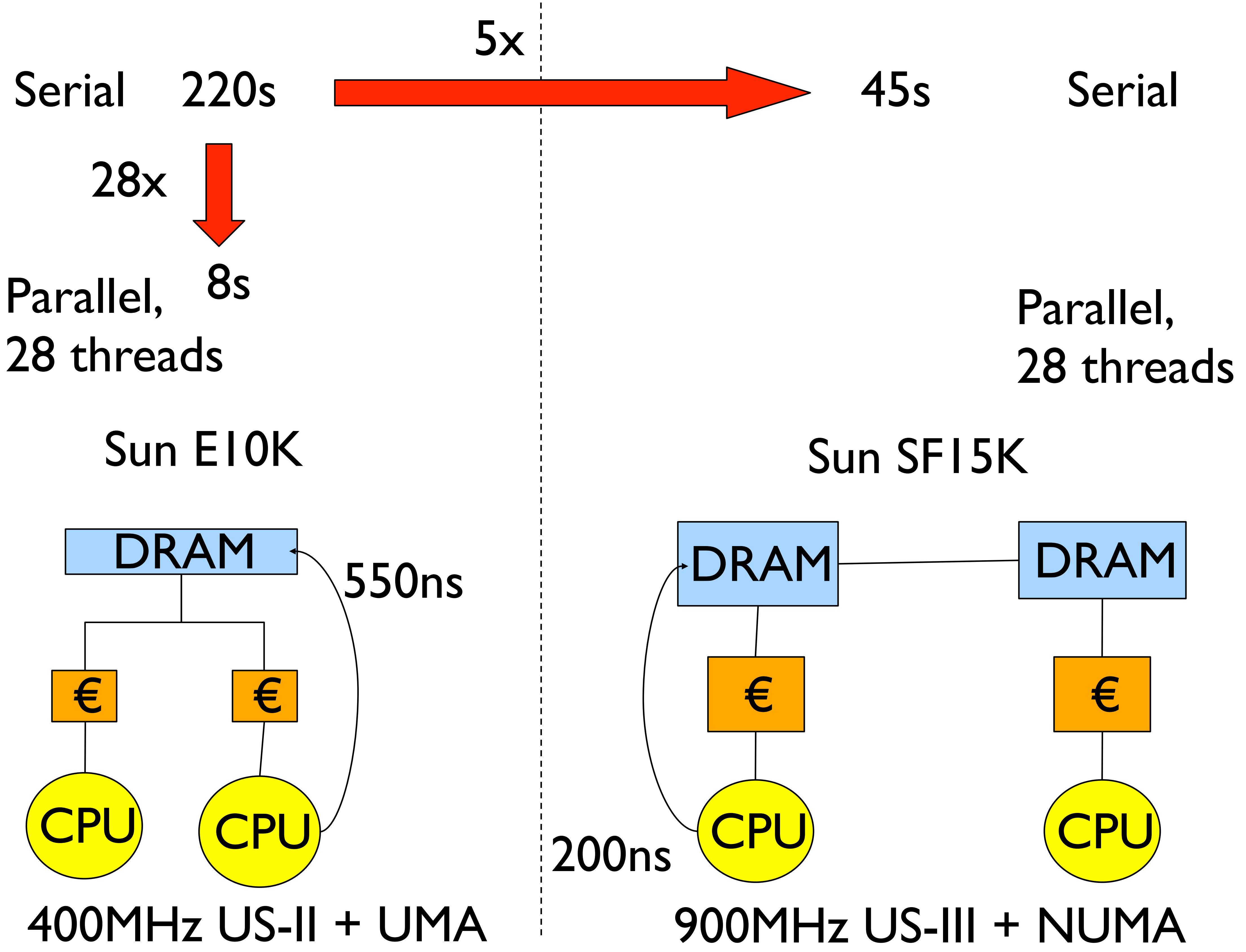
Parallel CG



5x
Serial 220s
45s Serial
28x
8s
Parallel,
28 threads
Sun E10K
DRAM
550ns
€
€
CPU
CPU
400MHz US-II + UMA
Sun SF15K
DRAM
DRAM
€
€
200ns
CPU
CPU
900MHz US-III + NUMA

UPPSALA UNIVERSITET

Parallel CG



UPPSALA UNIVERSITET

Parallel CG

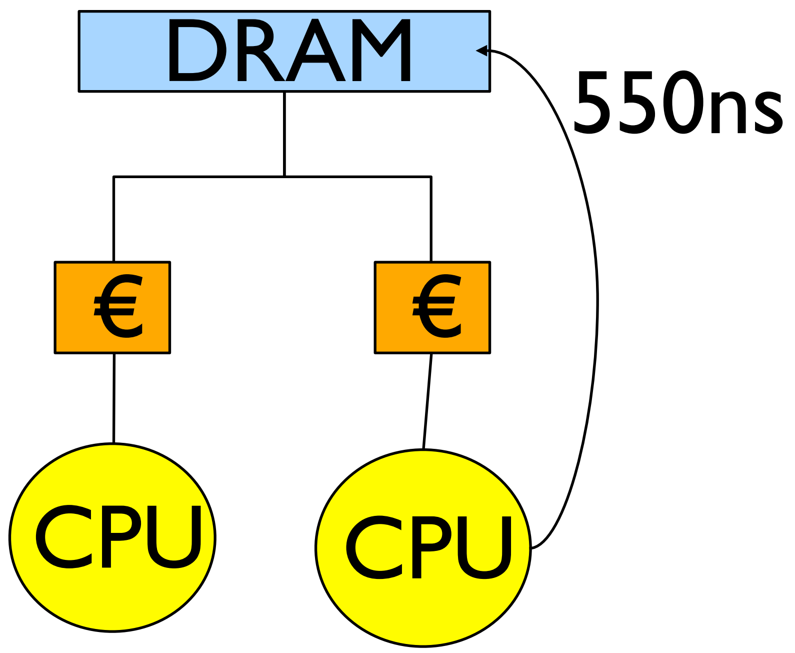
Serial 220s $\xrightarrow{5\times}$ 45s Serial

28x ↓

Parallel, 28 threads 8s

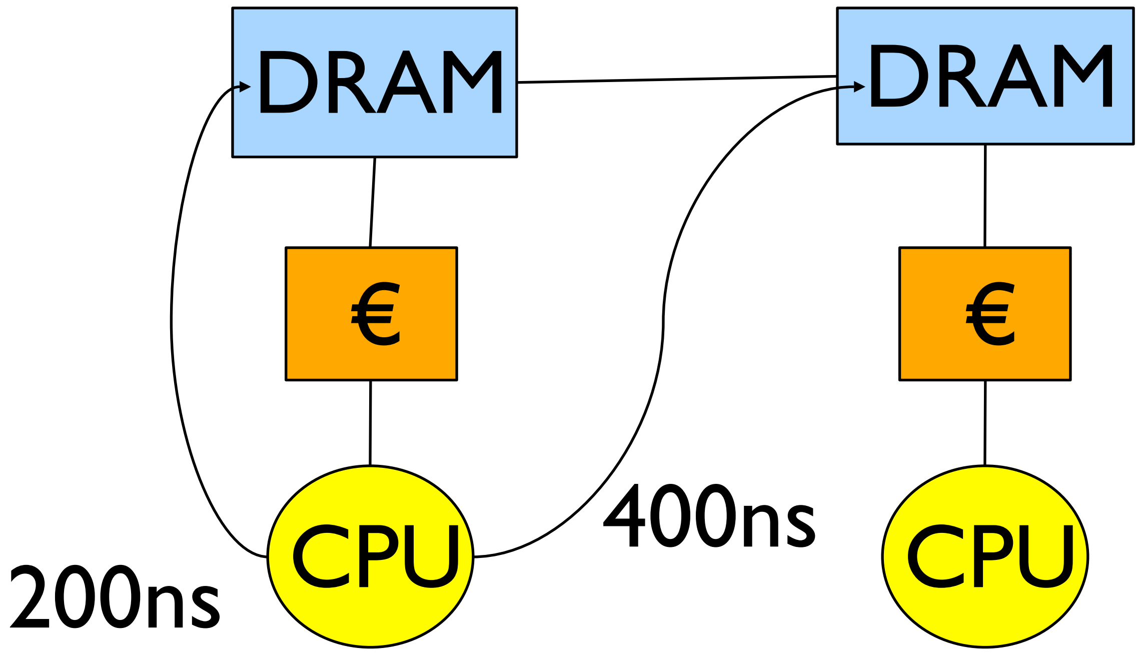
Parallel, 28 threads

Sun E10K



400MHz US-II + UMA

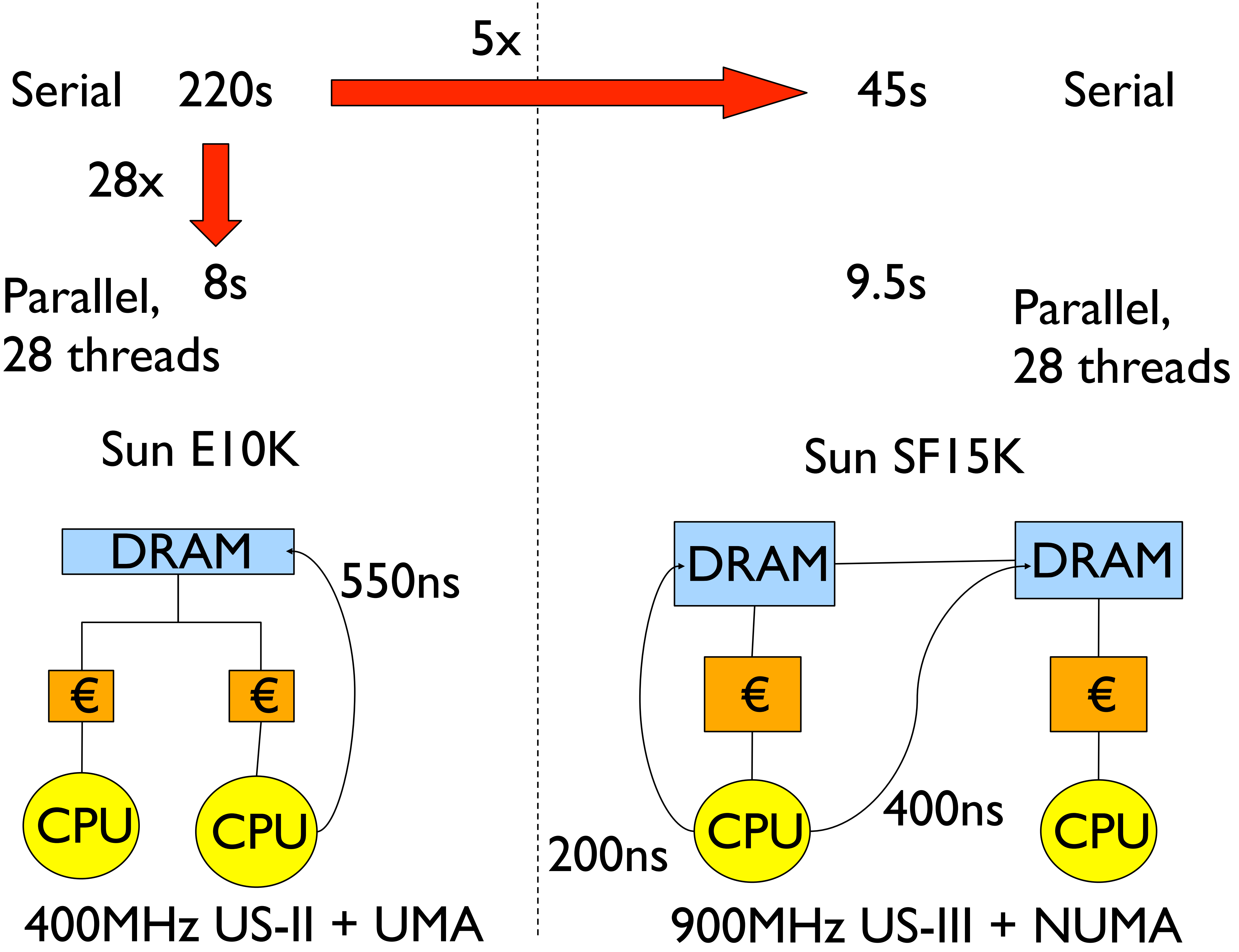
Sun SF15K



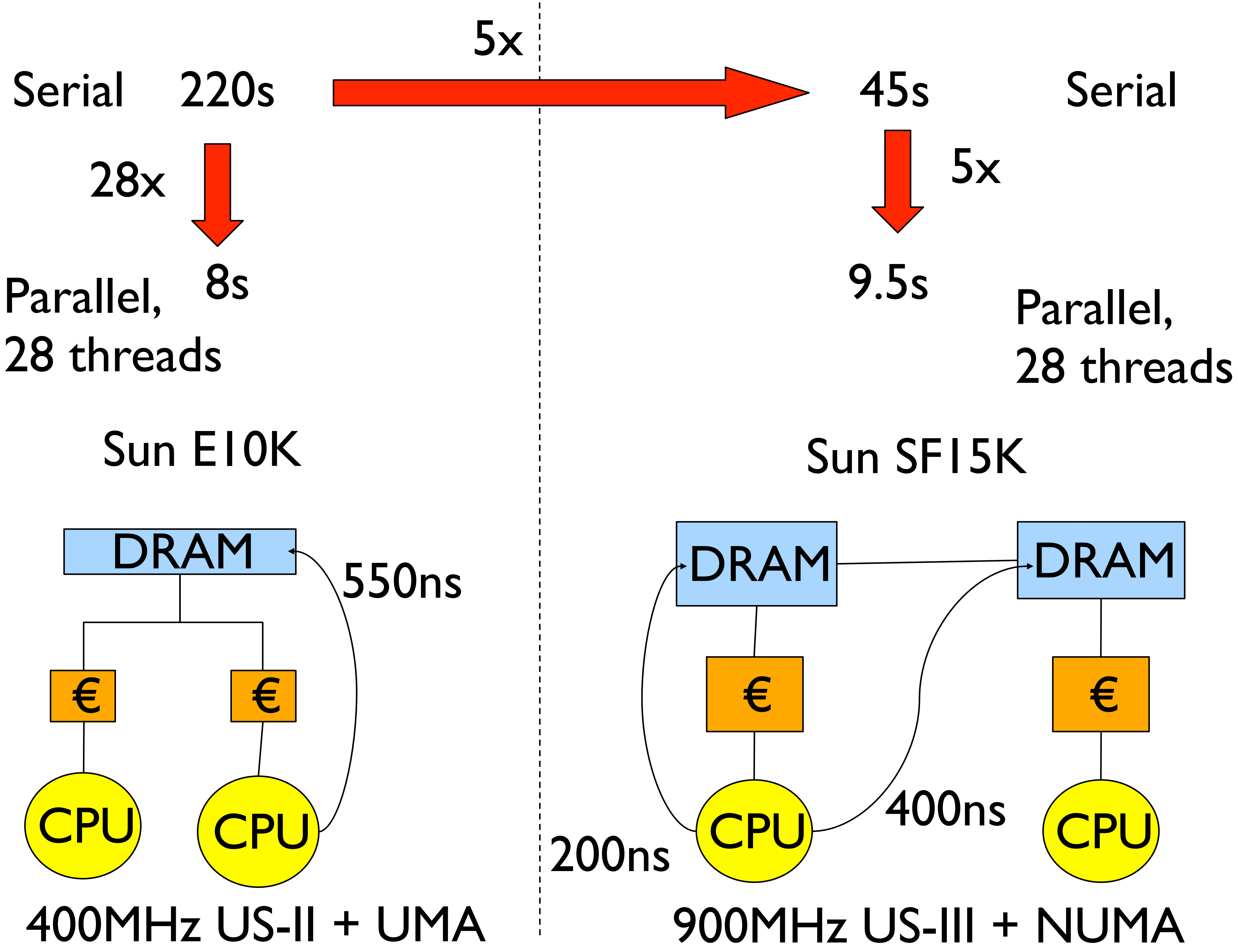
900MHz US-III + NUMA

UPPSALA UNIVERSITET

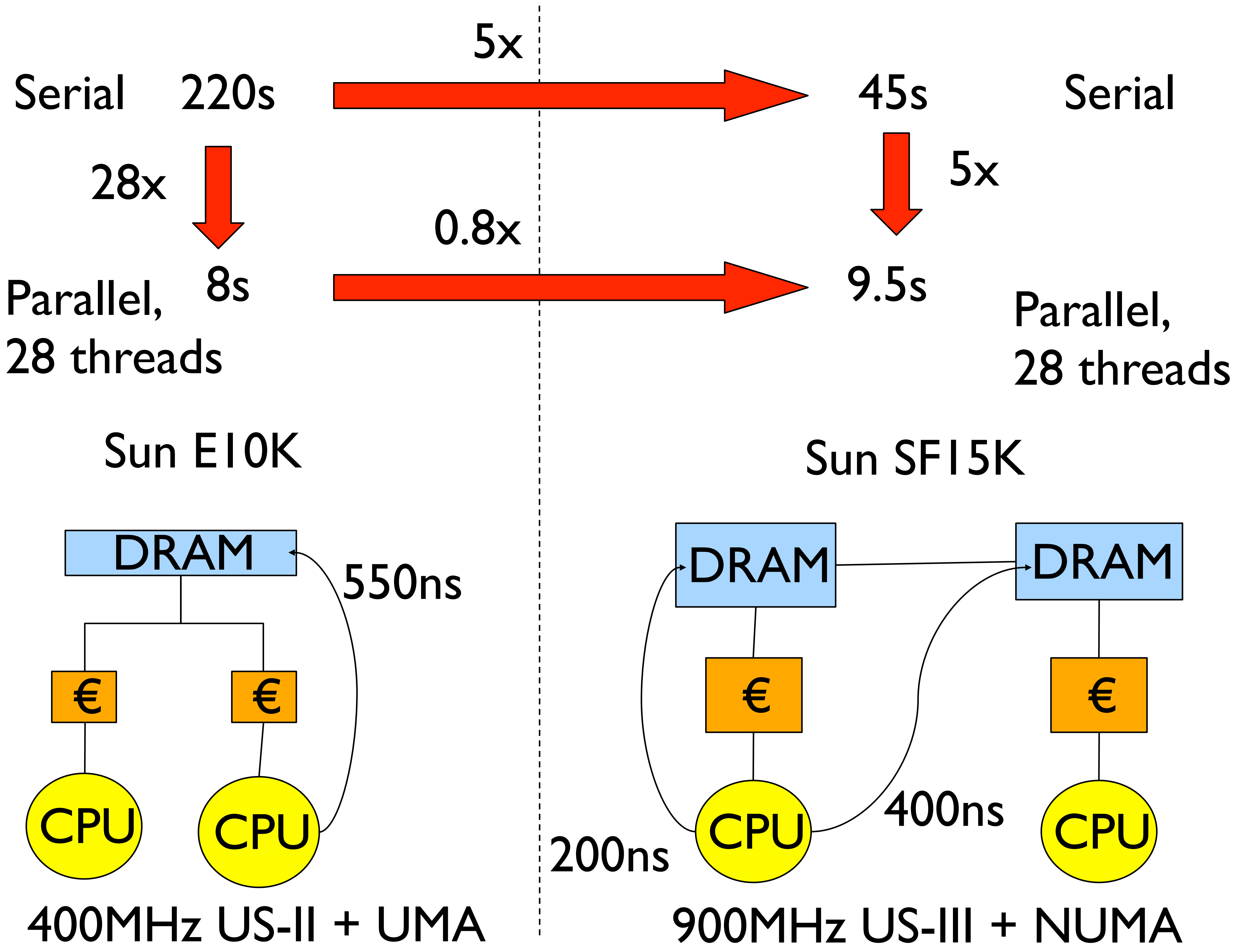
Parallel CG



Parallel CG


5x
Serial 220s
45s Serial
28x
5x
8s
Parallel, 28 threads
9.5s
Parallel, 28 threads
Sun E10K
Sun SF15K
DRAM
550ns
€
€
CPU
CPU
DRAM
DRAM
€
€
CPU
CPU
200ns
400ns
400MHz US-II + UMA
900MHz US-III + NUMA

Parallel CG


5x
Serial
220s
45s
Serial
28x
5x
0.8x
Parallel, 28 threads
8s
9.5s
Parallel, 28 threads
Sun E10K
Sun SF15K
DRAM
550ns
€
€
CPU
CPU
DRAM
DRAM
€
€
CPU
CPU
200ns
400ns
400MHz US-II + UMA
900MHz US-III + NUMA

UPPSALA UNIVERSITET

Geographical Locality

- Property of an application
  - ”How many memory accesses are node-local?”
  - Communication
- Dependent on many things
  - Data distribution (source code/OS)
  - Thread scheduling (OS)
- Thread-data affinity: *Minimize the amount of remote accesses by co-location of threads and data*

UPPSALA
UNIVERSITET

Problems of initialization

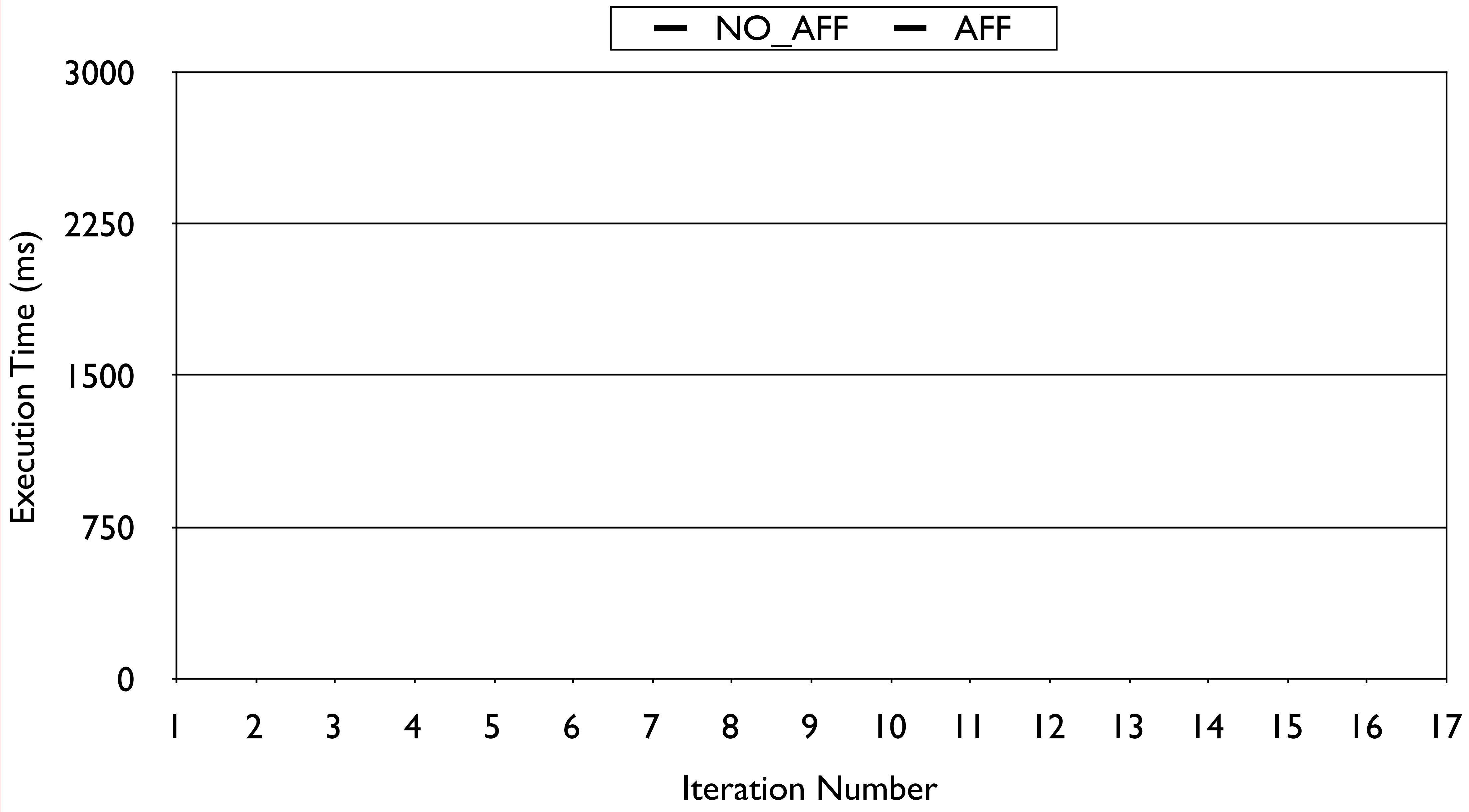
- FEM assembly process serial
- First-touch strategy
  - All data allocated on a single node
- Access pattern stored in the data
  - Compressed Sparse Row (CSR)
  - Efficient parallel initialization not possible
- We need ways of redistributing data during runtime

Code modifications

- Inserted affinity-on-next-touch call before the first CG iteration
- Access pattern is static cross the iterations
  - Only need to redistribute once
- Used hardware counters to quantify effect

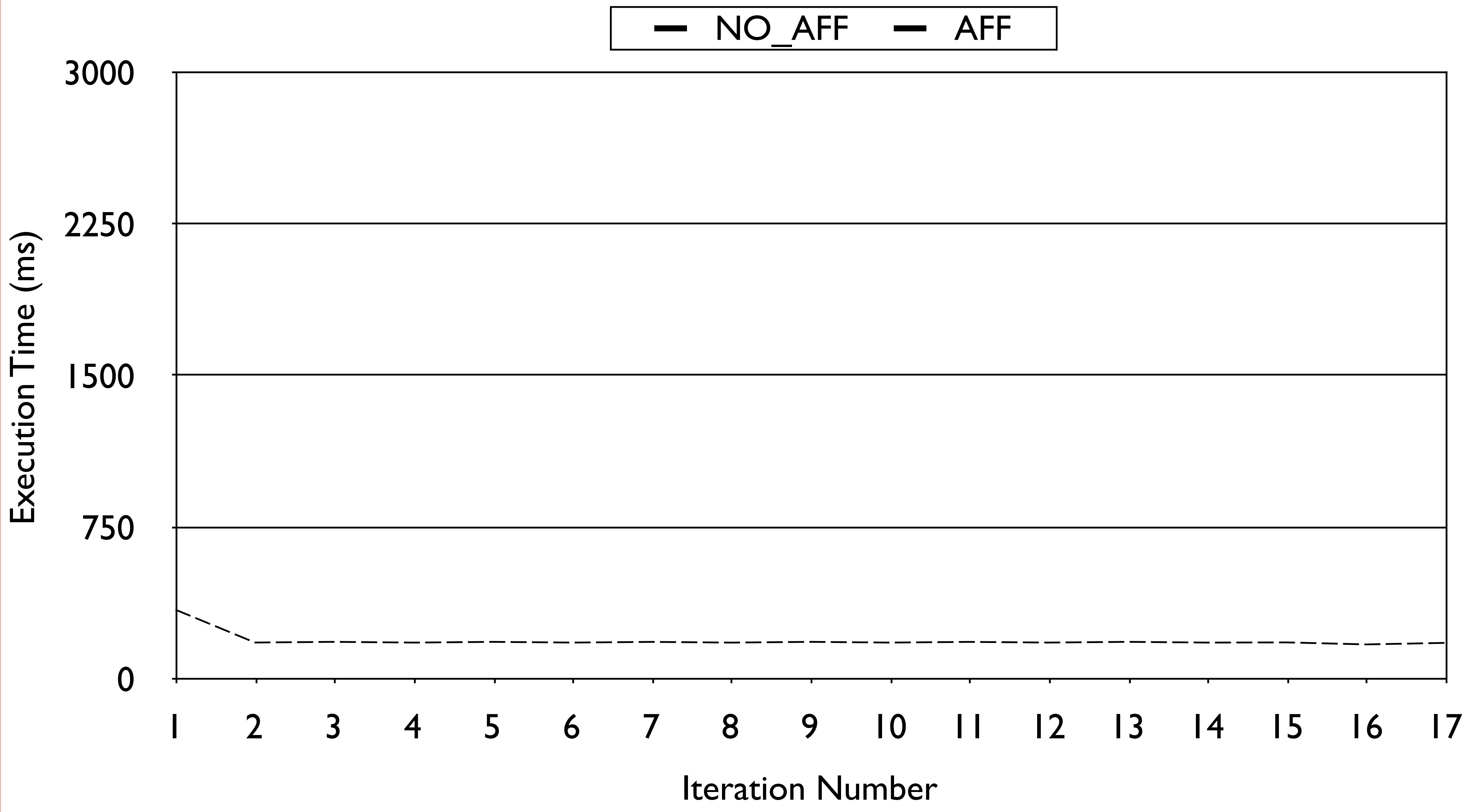
UPPSALA UNIVERSITET

Effect of affinity-on-next-touch



16 threads

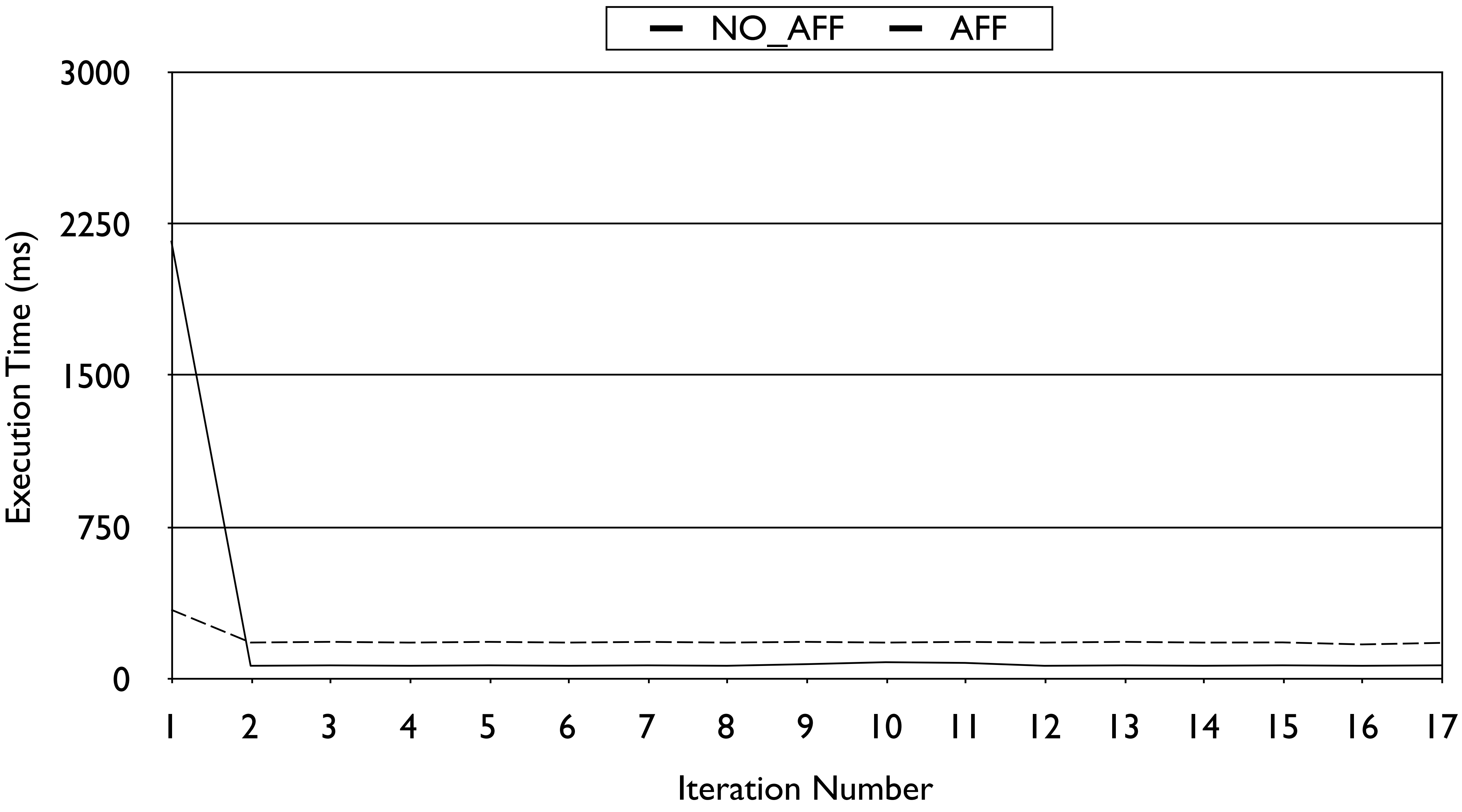
Effect of affinity-on-next-touch



16 threads

UPPSALA UNIVERSITET

Effect of affinity-on-next-touch

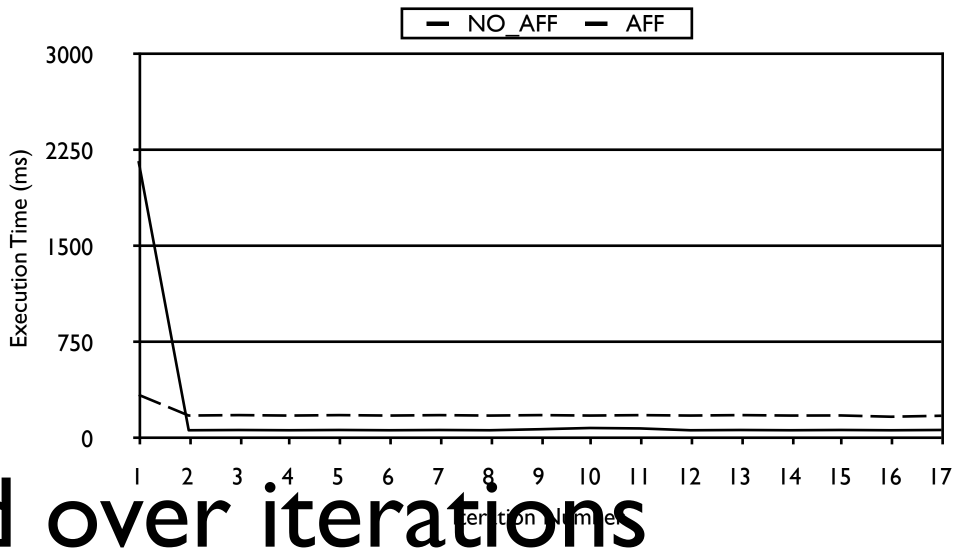


16 threads

UPPSALA UNIVERSITET

Analysis, affinity-on-next-touch

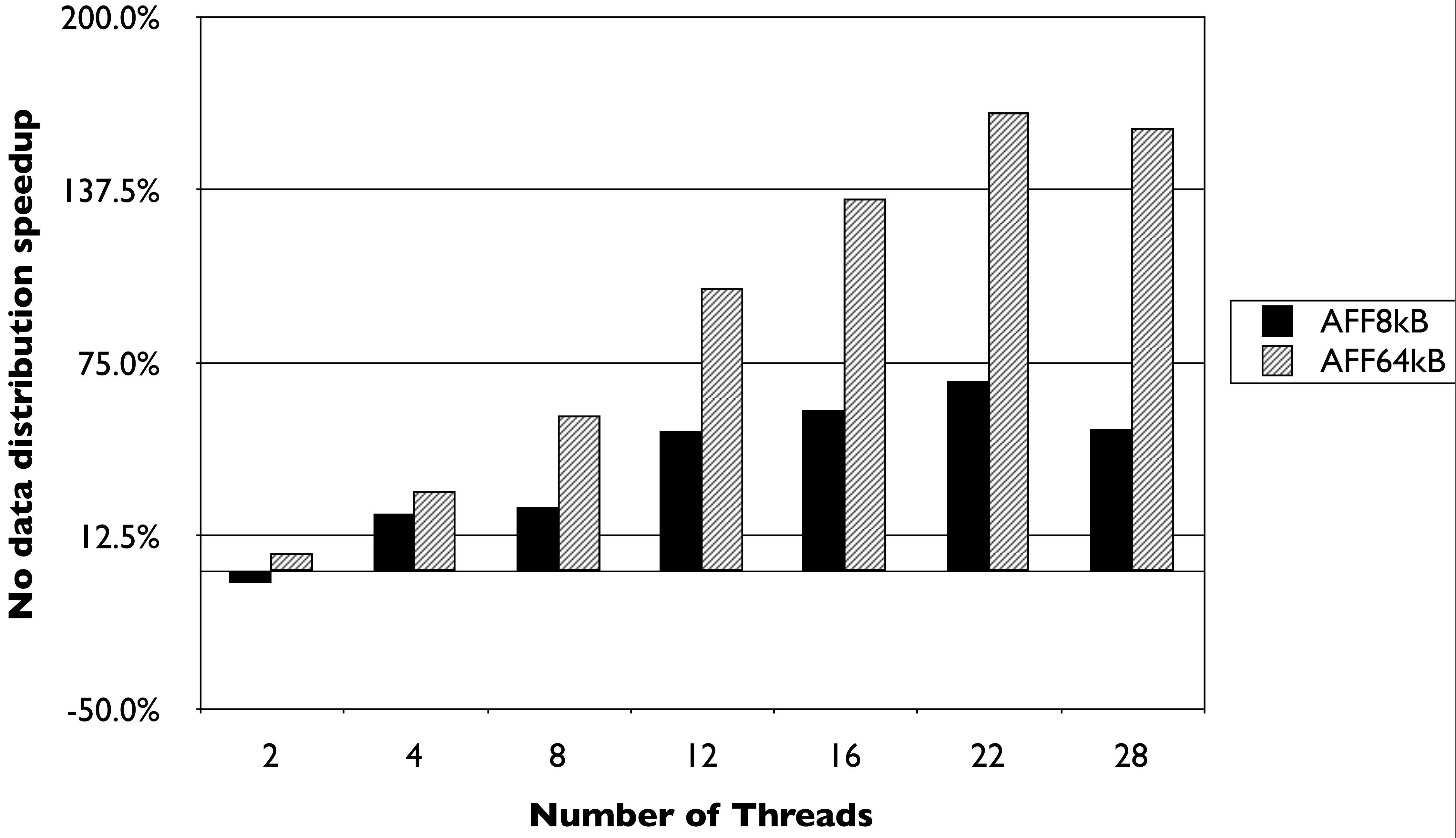
- Proper data distribution important
  - Every iteration 2.61 times faster
- Overhead
  - Cold start effects
  - Cost of migration



- Overhead is amortized over iterations
- Scalability better, still poor
  - Speedup 9 on 28 threads

UPPSALA UNIVERSITET

Overall effect of affinity-on-next-touch



UPPSALA UNIVERSITET

Two generations, again

$$\begin{tikzcd}
\text{Serial} & 220\text{s} \arrow[r, "5\text{x}"] \arrow[d, "28\text{x}"'] & 45\text{s} \arrow[d, "5\text{x}"] & \text{Serial} \\
\text{Parallel} & 8\text{s} \arrow[r, "0.8\text{x}"] & 9.5\text{s} & \text{Parallel}
\end{tikzcd}$$

UPPSALA UNIVERSITET

Two generations, again

$$\begin{tikzcd}
\text{Serial} & 220\text{s} \arrow[r, "5\text{x}"] \arrow[d, "28\text{x}"'] & 45\text{s} \arrow[d, "5\text{x}"] & \text{Serial} \\
\text{Parallel} & 8\text{s} \arrow[r, "0.8\text{x}"] & 9.5\text{s} & \text{Parallel}
\end{tikzcd}$$

affinity-on-next-touch + 64Kb

Two generations, again

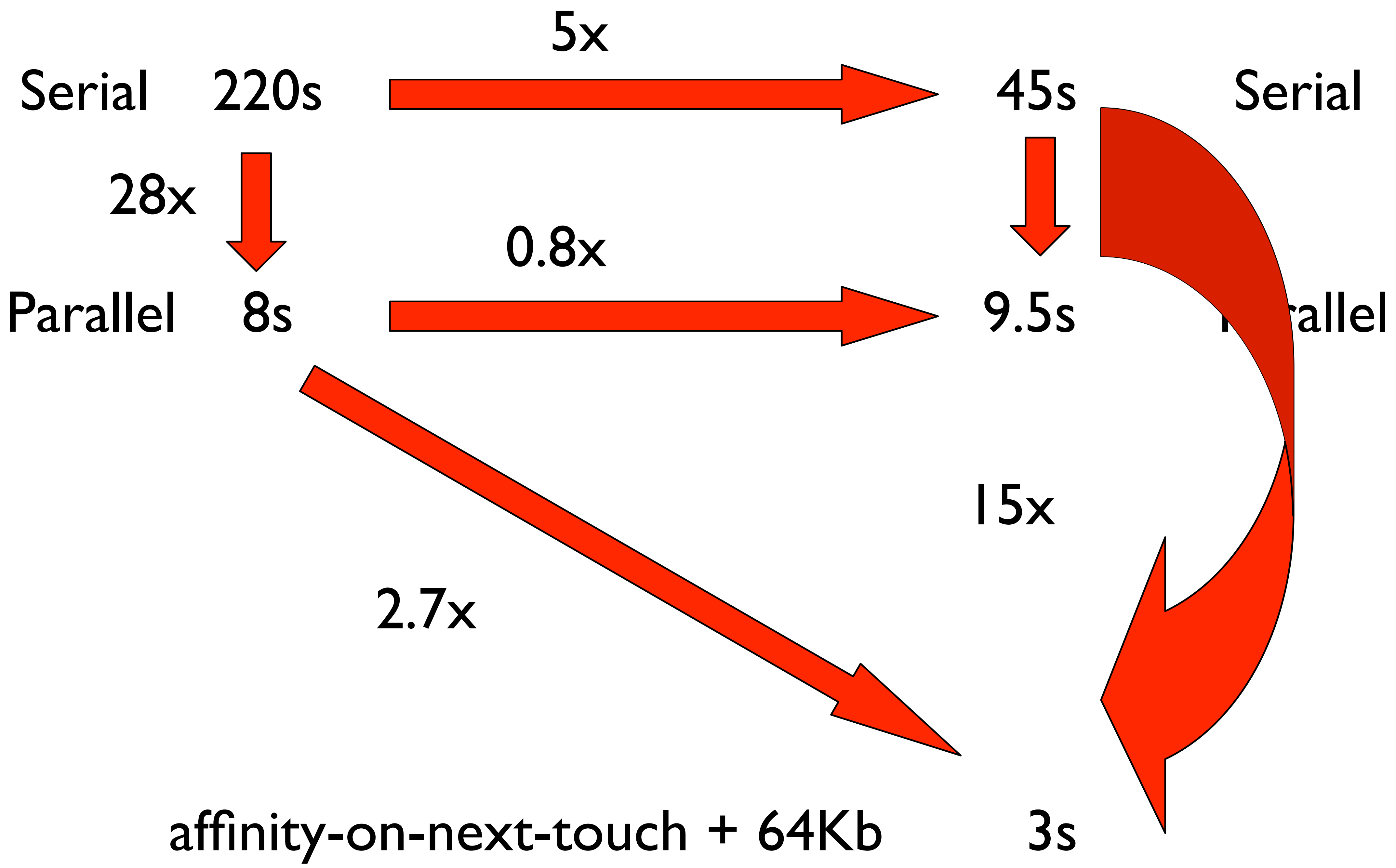
$$\begin{tikzcd}
\text{Serial} & 220\text{s} \arrow[rr, "5\text{x}"] \arrow[d, "28\text{x}"'] & & 45\text{s} \arrow[d, "5\text{x}"] & \text{Serial} \\
\text{Parallel} & 8\text{s} \arrow[rr, "0.8\text{x}"] & & 9.5\text{s} & \text{Parallel}
\end{tikzcd}$$

affinity-on-next-touch + 64Kb 3s

Two generations, again

$$\begin{tikzcd}
\text{Serial} & \text{220s} \arrow[r, "\text{5x}"] \arrow[d, "\text{28x}"'] & \text{45s} \arrow[d] \arrow[dd, bend left, "\text{15x}"'] & \text{Serial} \\
\text{Parallel} & \text{8s} \arrow[r, "\text{0.8x}"] & \text{9.5s} & \text{Parallel} \\
\text{affinity-on-next-touch + 64Kb} & & \text{3s} &
\end{tikzcd}$$

Two generations, again



Algorithmic Optimizations (CG)

- Bandwidth minimization
  - Increases cache utilization
- Load Balance
  - Graph partitioning (MeTiS)
- Removing barriers
  - Reduces parallel overhead
- How do they interact with data distributions?

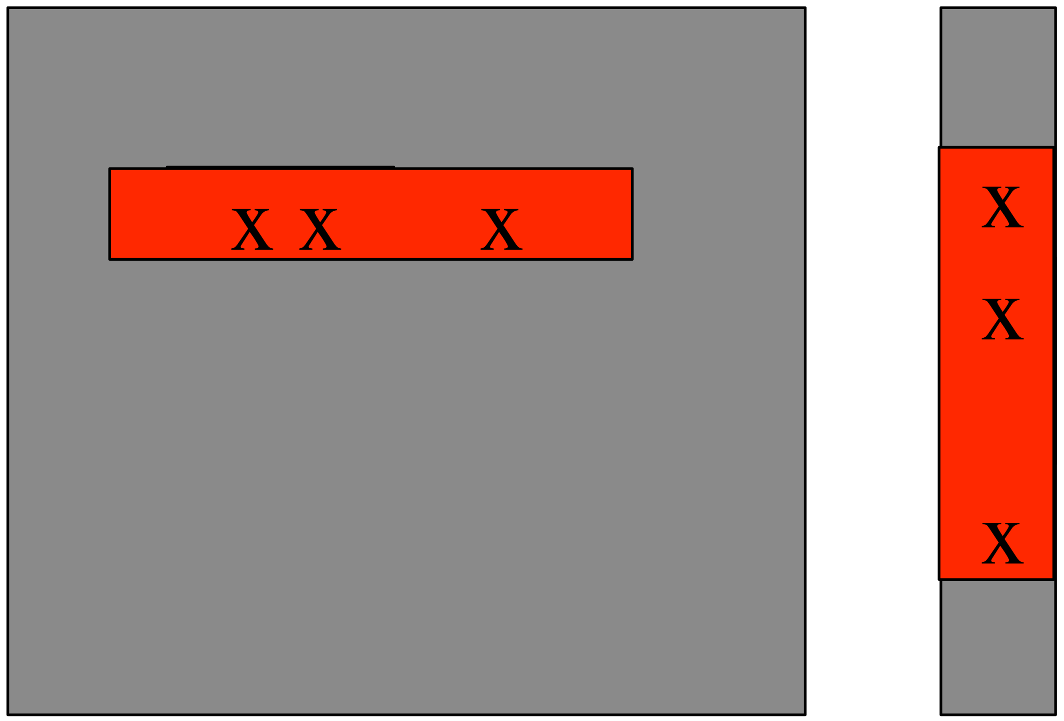
UPPSALA UNIVERSITET

Removing Barriers

- Standard implementation uses 7 barriers
- 4 barriers can be removed
  - Privatizing scalars
  - Reording initializations of reduction variables
  - 3 global barriers in total
- S-step methods (Chronopoulos/Gear)
  - Introduces another vector to calculate two iterations simultaneously
  - Only one reduction necessary
  - 2 global barriers in total
  - No numerical problems

Bandwidth Minimization

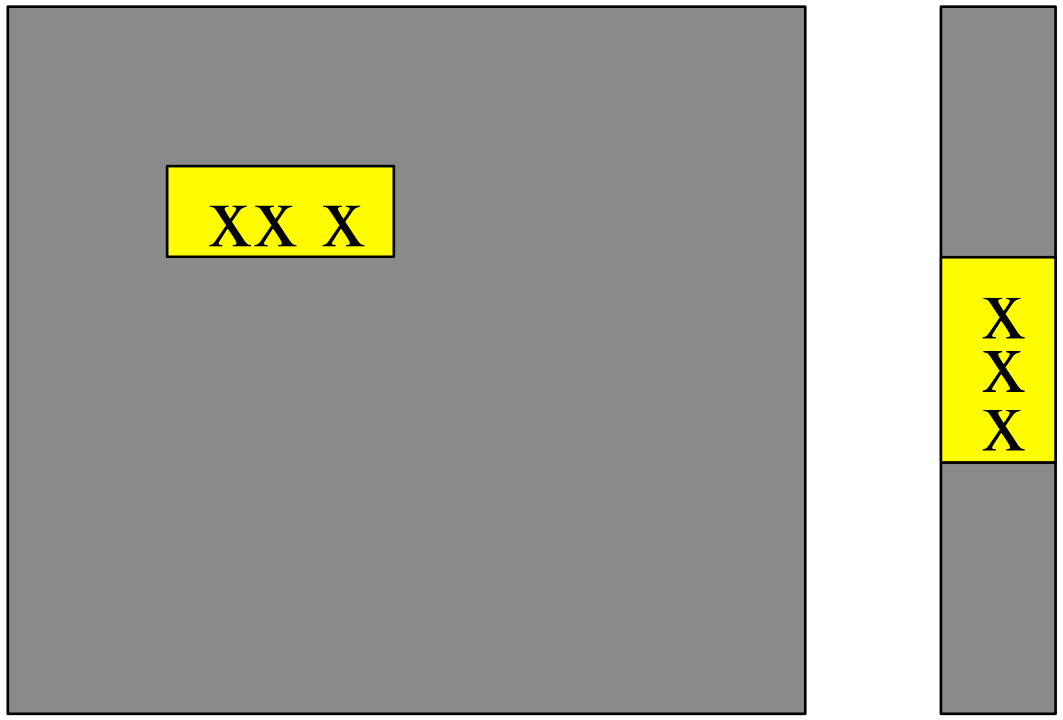
- Graph theoretical method
  - Reverse Cuthill-McKee
  - Gibbs-Poole-Stockmayer (GPS)
- Permutes matrix to minimize bandwidth
- Small bandwidth increases locality
  - Larger part of RHS cache blocks utilized



UPPSALA UNIVERSITET

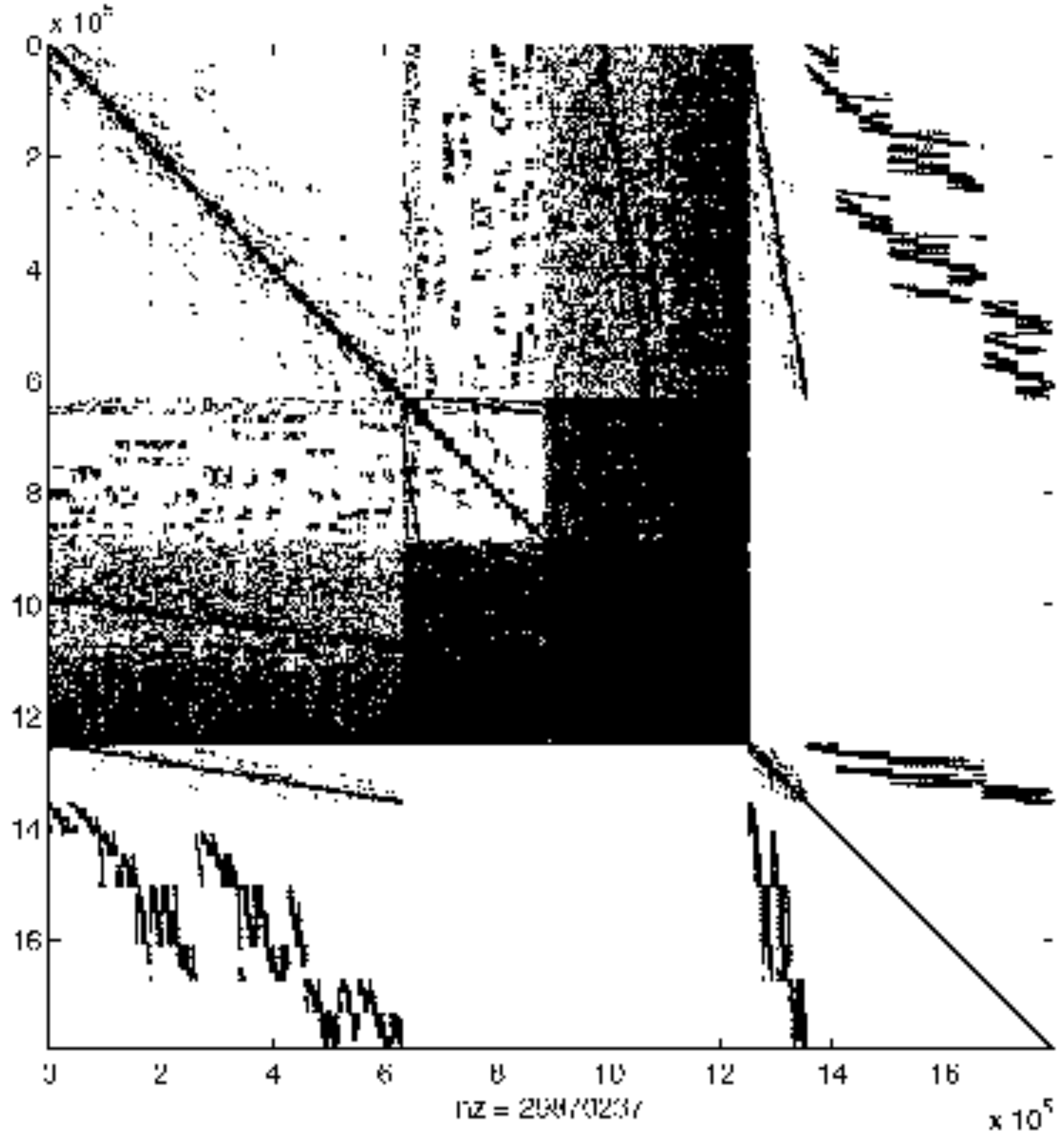
Bandwidth Minimization

- Graph theoretical method
  - Reverse Cuthill-McKee
  - Gibbs-Poole-Stockmayer (GPS)
- Permutes matrix to minimize bandwidth
- Small bandwidth increases locality
  - Larger part of RHS cache blocks utilized

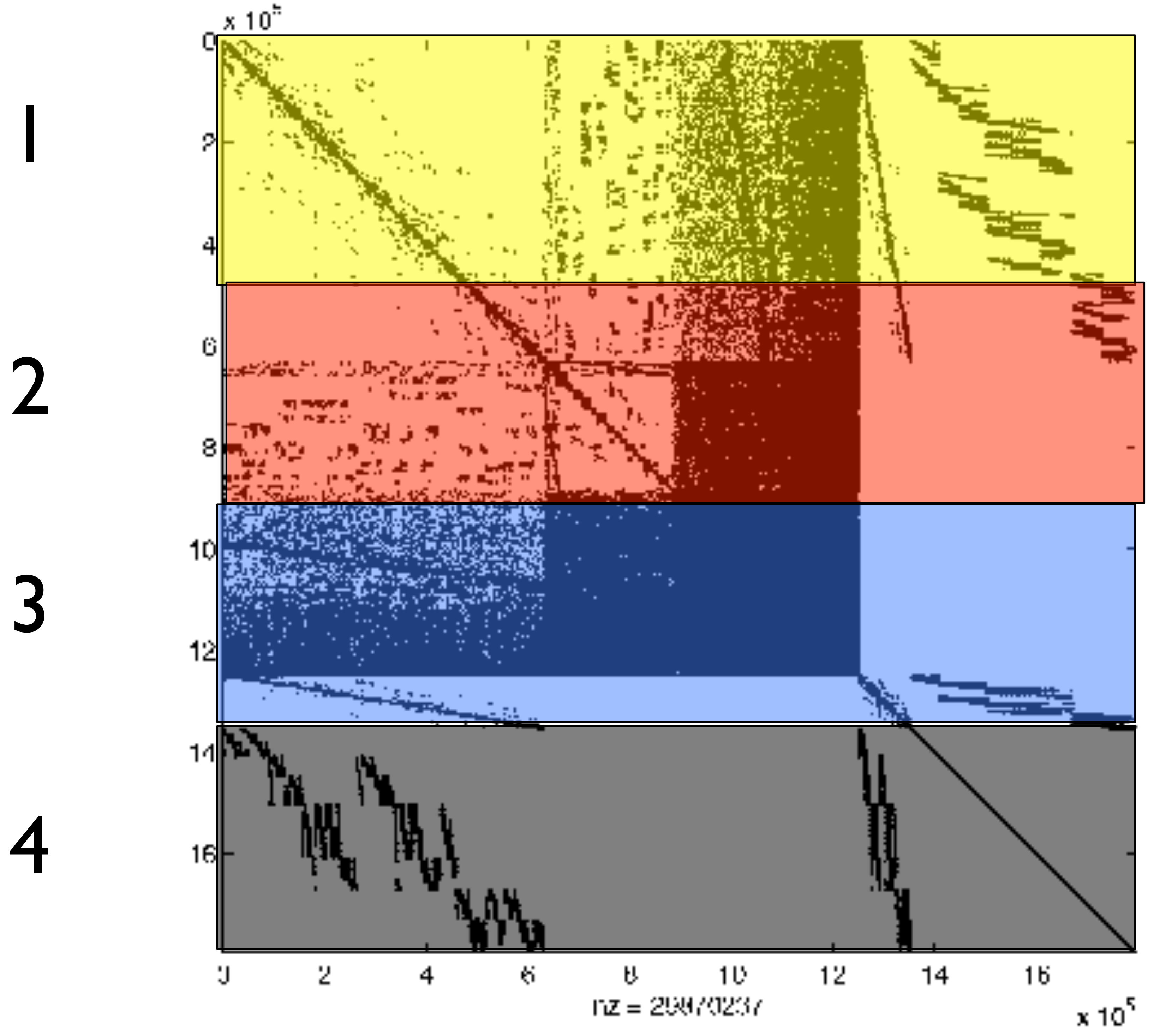


UPPSALA UNIVERSITET

Load Balance, example



Load Balance, example

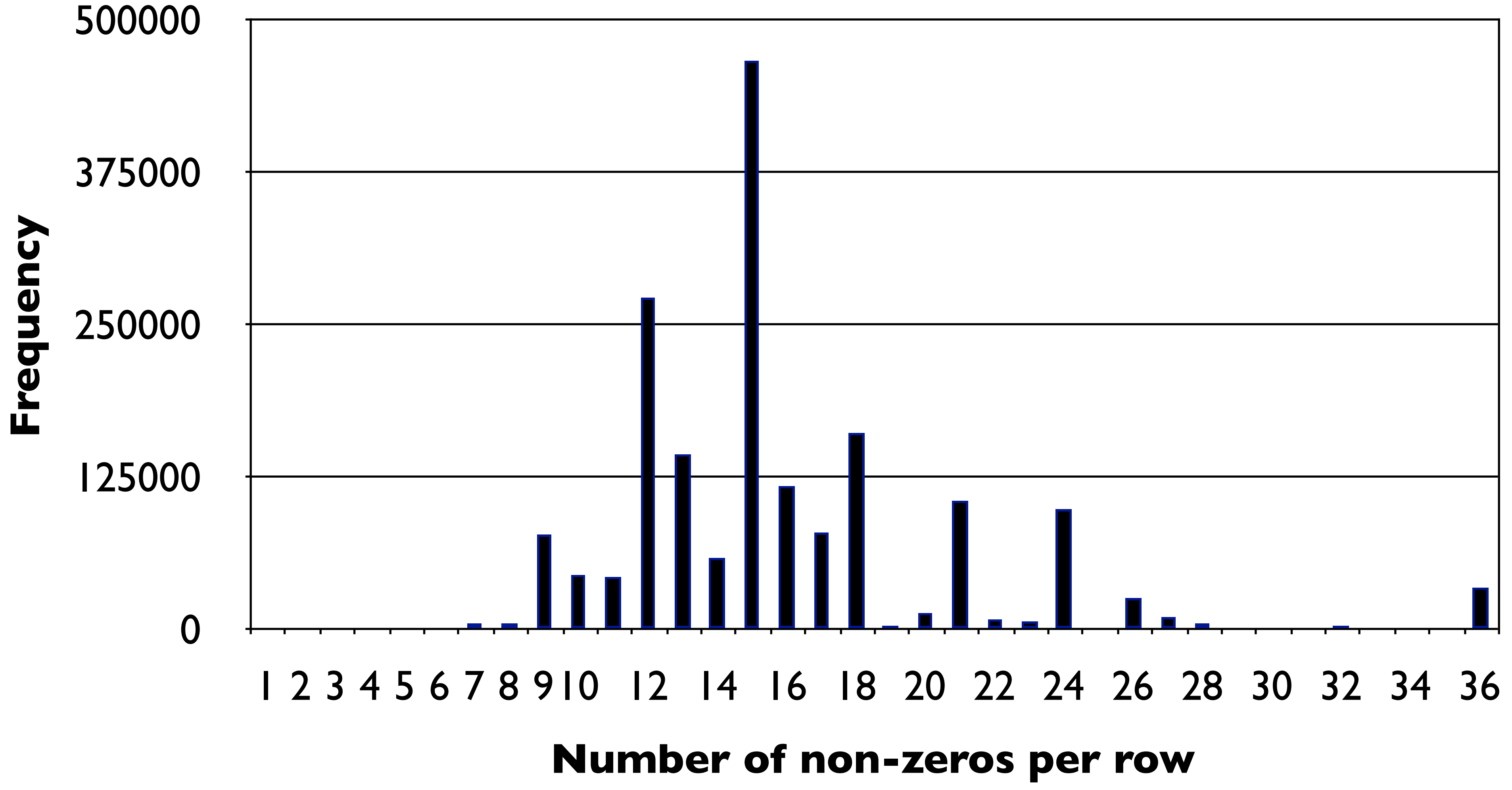


Load Balance and Partitioning

- In OpenMP partitions are linear
  - A chunk of rows (indeces) is associated with a thread
  - No care is taken to the number of non-zeros
- Graph partitioning partitions the non-zeros more evenly
  - Used in message-passing applications
- Results:
  - Graph partitioning increased matrix bandwidth which led to poor locality
  - Bandwidth minimization toghether with standard OpenMP produced very good partitions

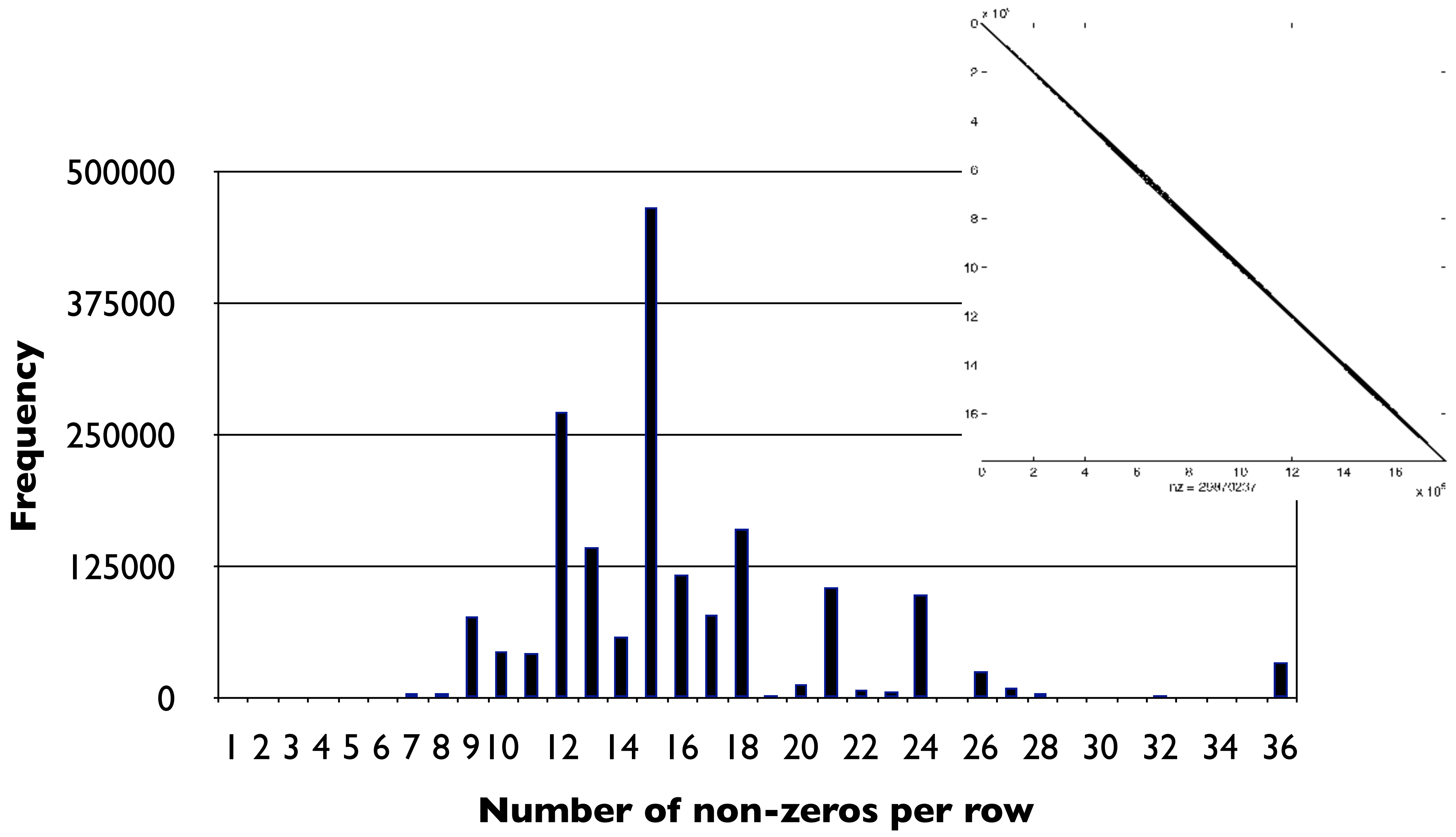
UPPSALA UNIVERSITET

Load balance of SpMxV



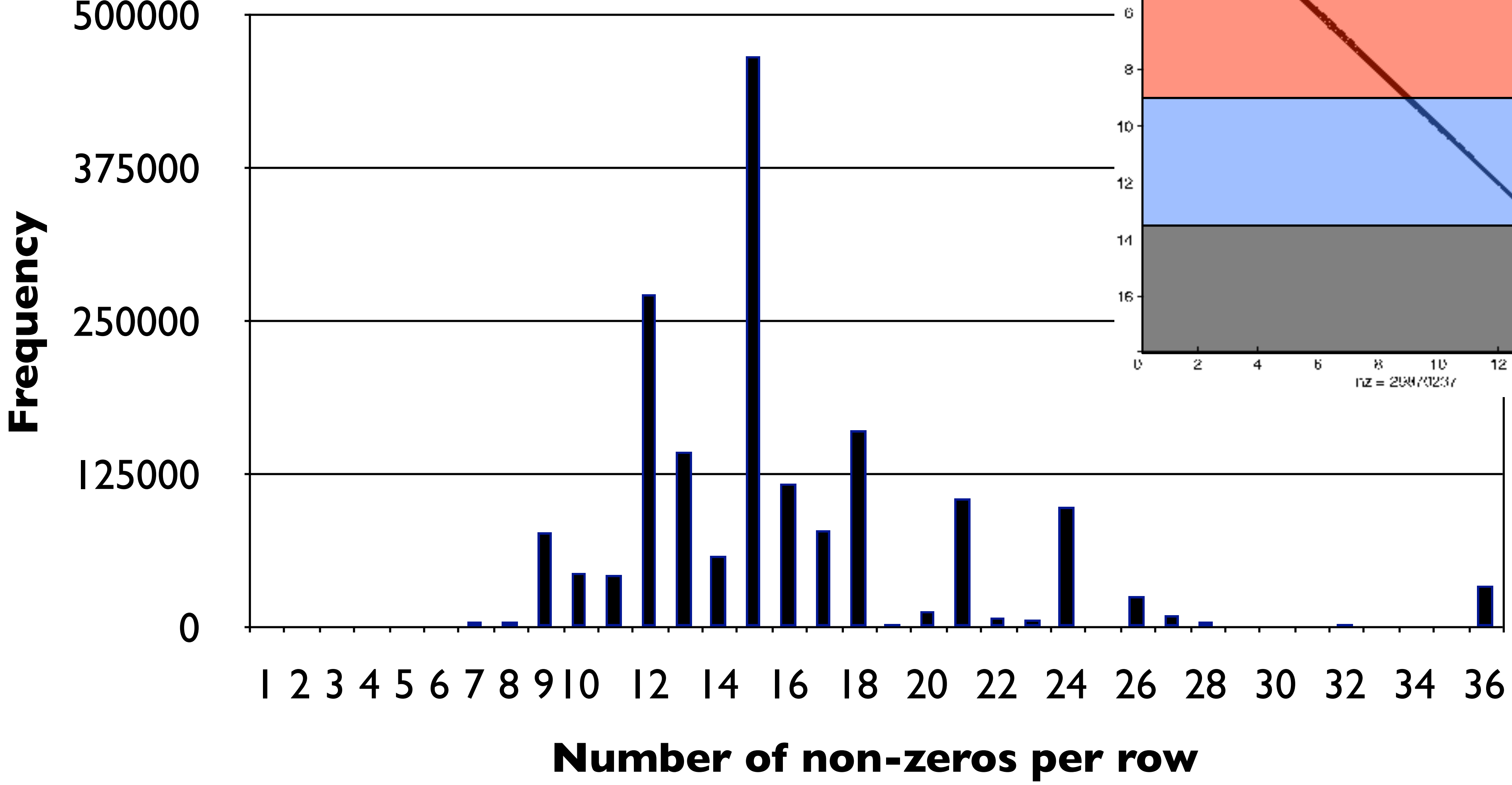
16 partitions	Base	GPS	MeTiS
Max(non_zeros)/Avg(non_zeros)	1.24	1.01	1.15

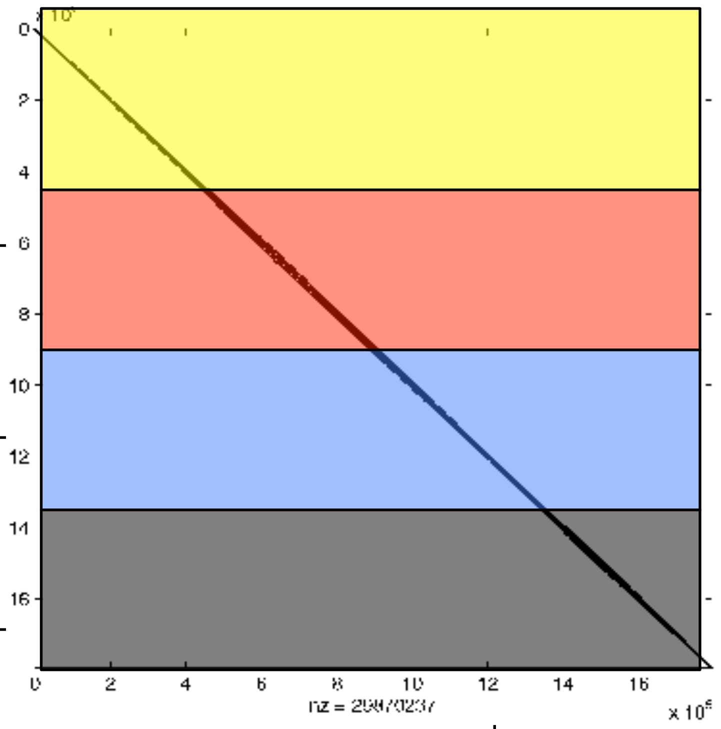
Load balance of SpMxV



16 partitions	Base	GPS	MeTiS
Max(non_zeros)/Avg(non_zeros)	1.24	1.01	1.15

Load balance of SpMxV

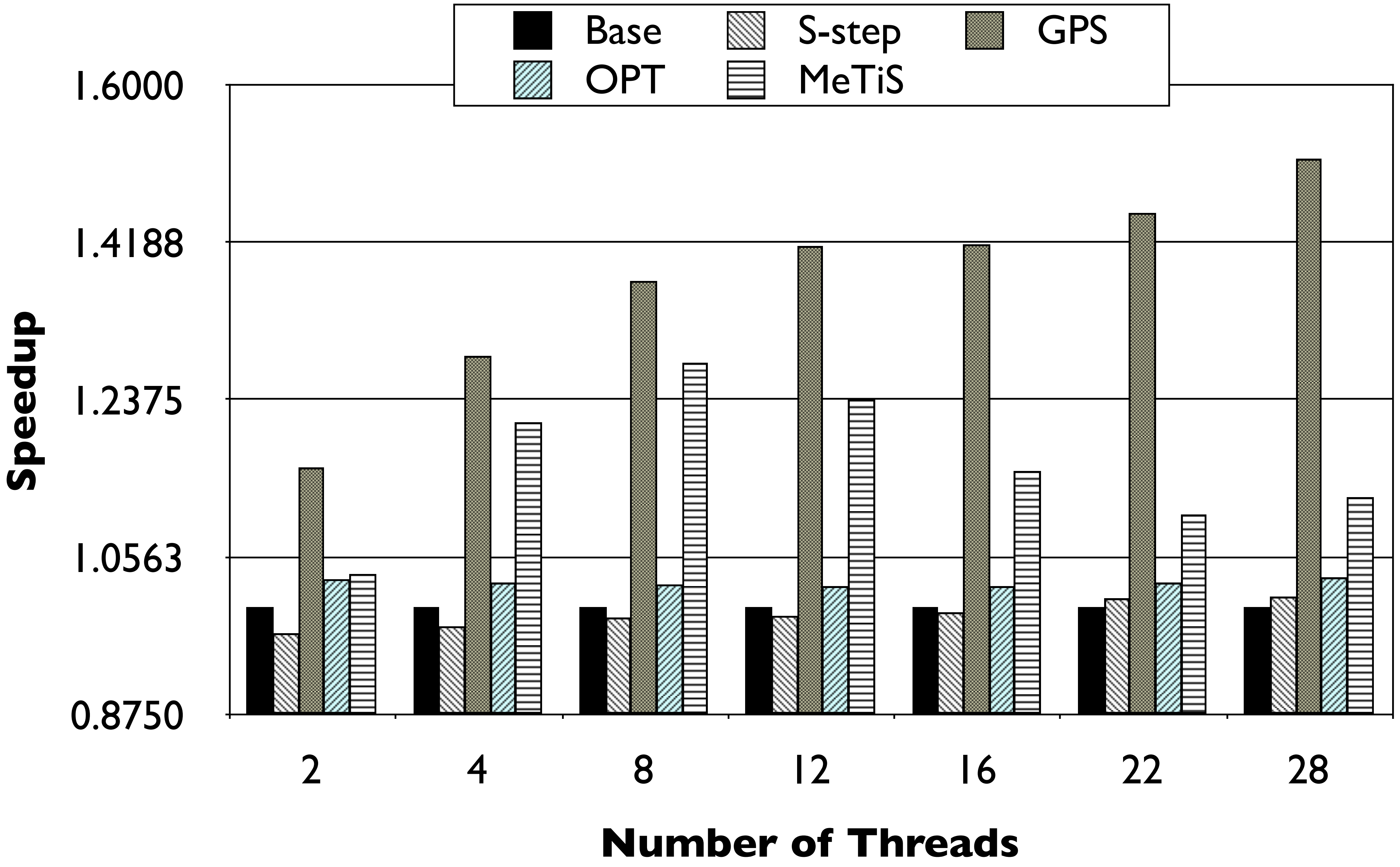




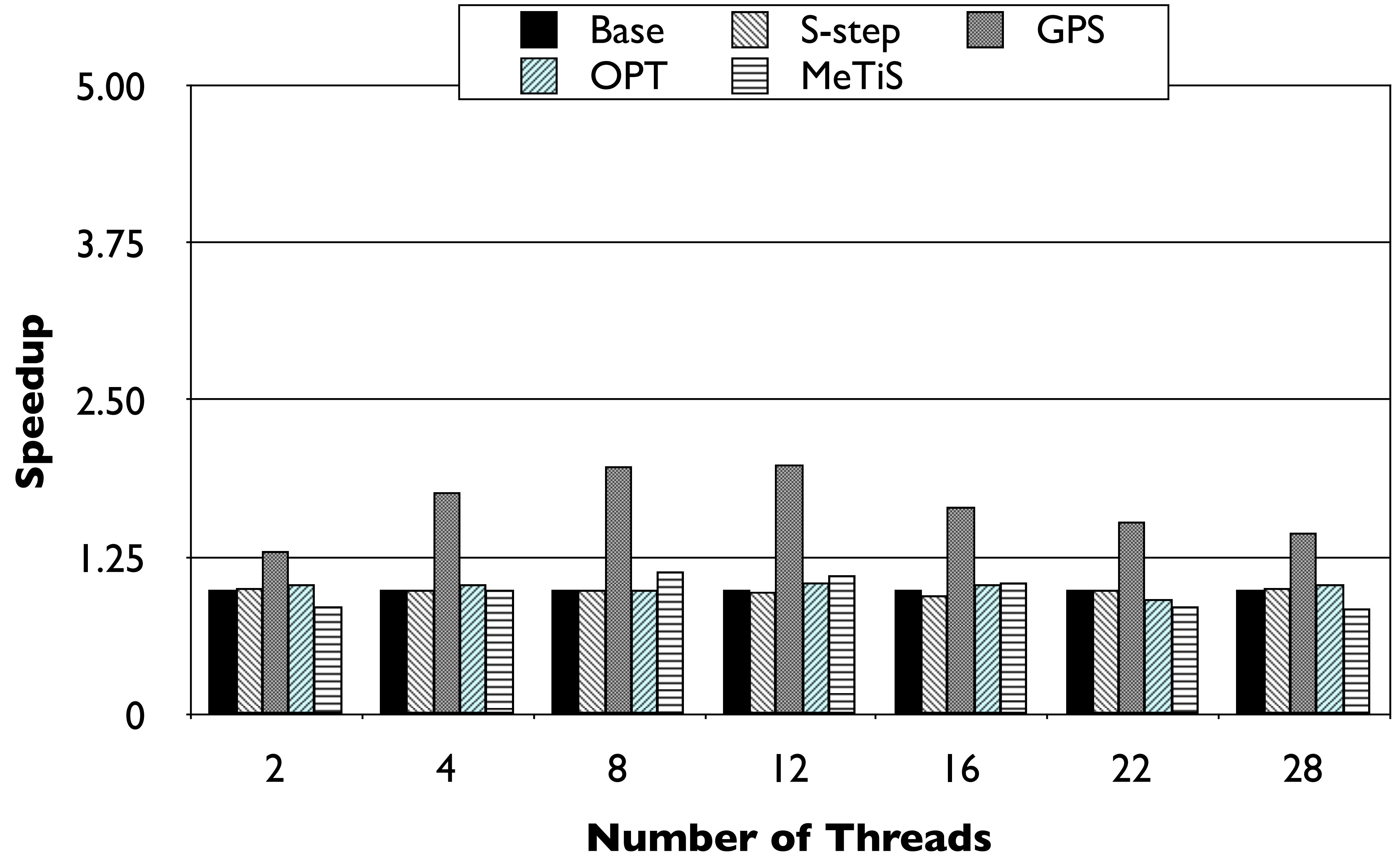
16 partitions	Base	GPS	MeTiS
Max(non_zeros)/Avg(non_zeros)	1.24	1.01	1.15

UPPSALA UNIVERSITET

Uniform system, E10K



Non-uniform system (SF15K) with data distribution



Conclusions, algorithmical optimizations

- Locality most important
  - Bandwidth minimization
- Bandwidth minimization also produced load balanced partitions
- Graph partitioning increased the amount of remote accesses
- Load balance and synchronization overheads are of secondary importance

UPPSALA
UNIVERSITET

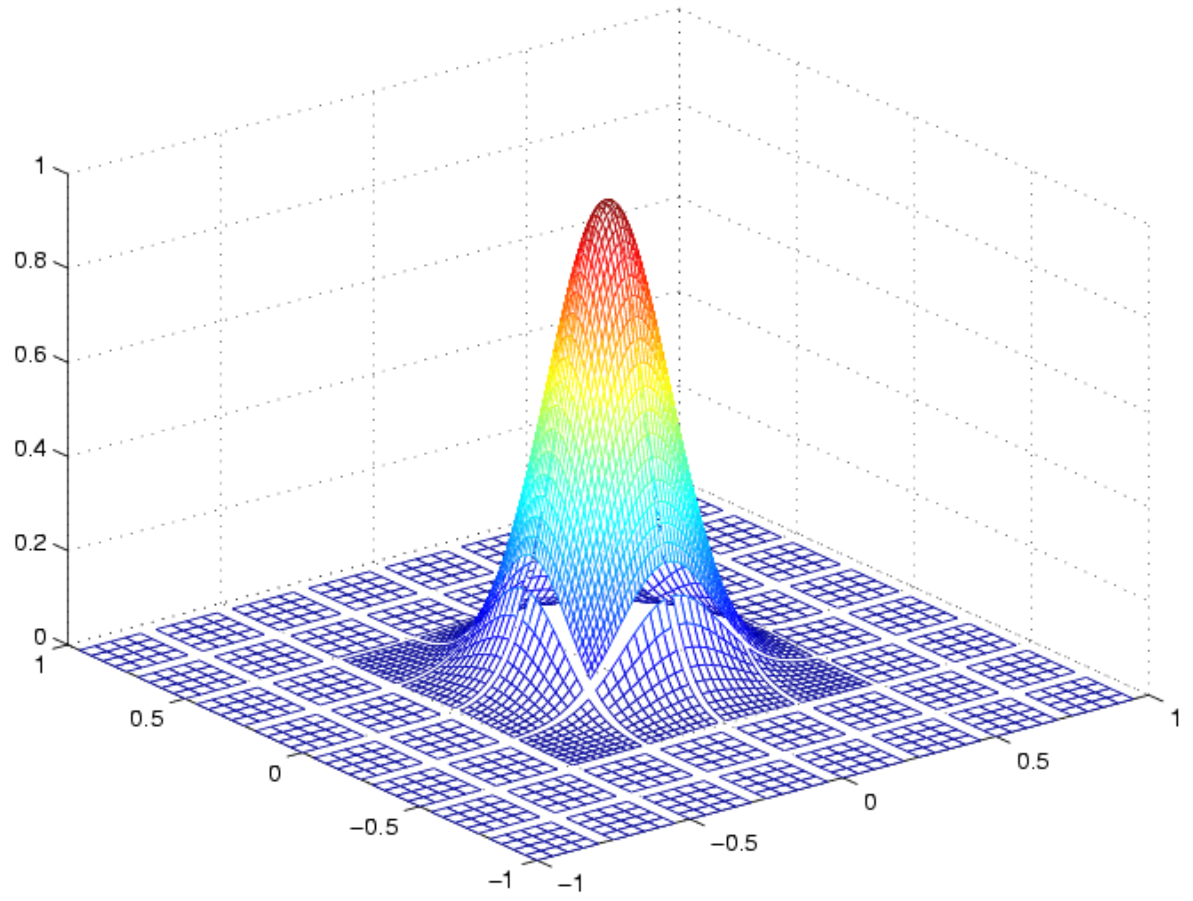
Dynamic access patterns

- Many applications exhibit dynamic and/or unstructured access patterns
  - Commercial Software
  - Adaptive Mesh Refinement (AMR)
- For these applications, clever initializations will not be optimal
  - We need some kind of migration/replication mechanism

UPPSALA UNIVERSITET

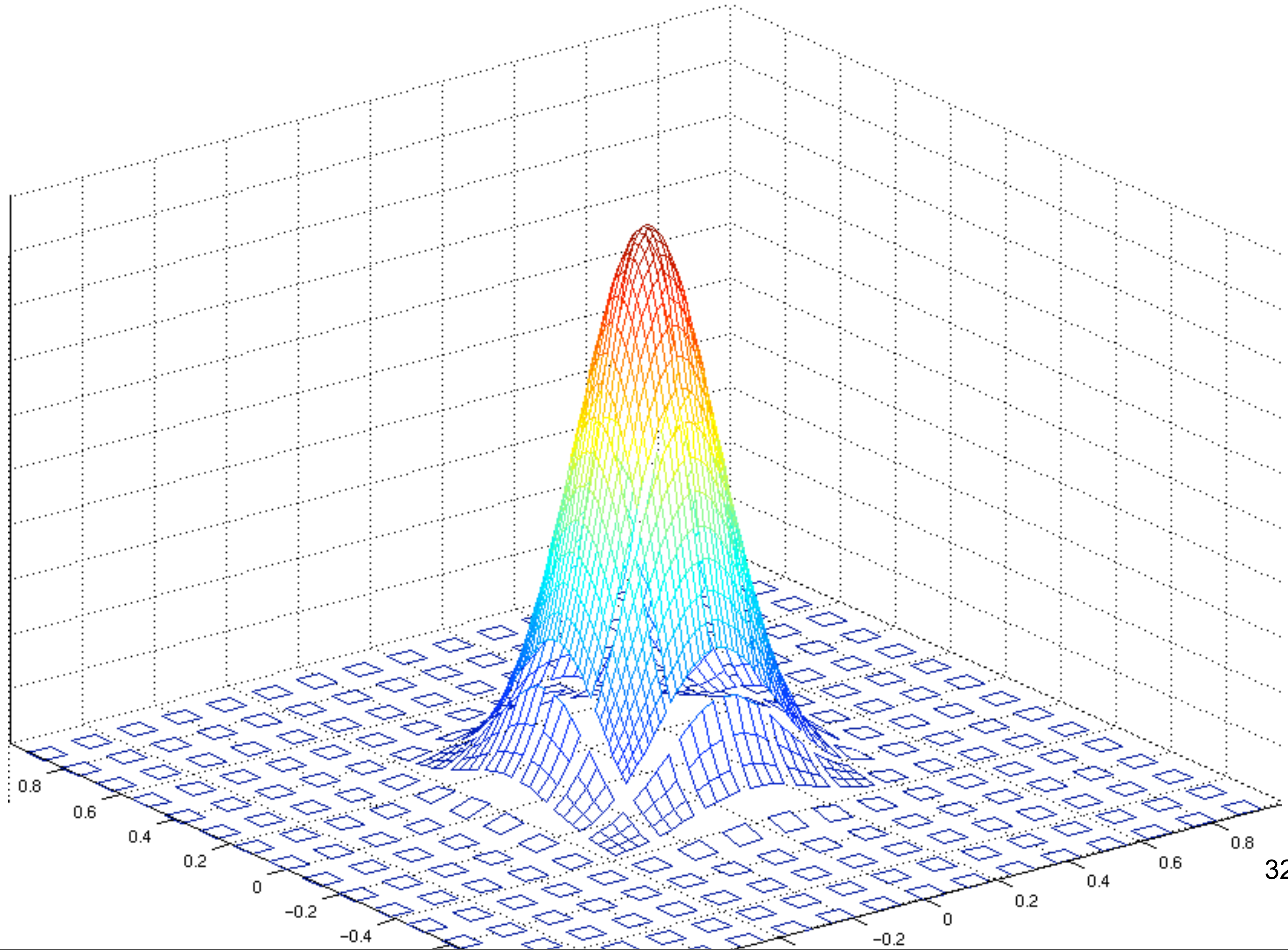
Model problem

- $$\frac{\partial u}{\partial t} = a\frac{\partial u}{\partial x} + b\frac{\partial u}{\partial y}$$
- Periodic boundaries
- Finite differences
  - – 2nd order in space
  - – 4th order R-K in time
- Blockwise AMR
- Written in Fortran 90/95
- Parallelized using OpenMP



UPPSALA UNIVERSITET

AMR movie



UPPSALA UNIVERSITET

Solver Algorithm

```
do t=1,Nt
   if (t mod adaptInterval=0) then
       Estimate error per block.
       Adapt blocks with inappropriate resolution.
       Repartition the grid.
       Migrate blocks (if migration is activated).
   end if
   F1=Diff(u);
   F2=Diff(u+k/2*F1)
   F3=Diff(u+k/2*F2)
   F4=Diff(u+k*F3)
   u=u+k/6*F1+k/3*F2+k/3*F3+k/6*F4
end do
```

UPPSALA UNIVERSITET

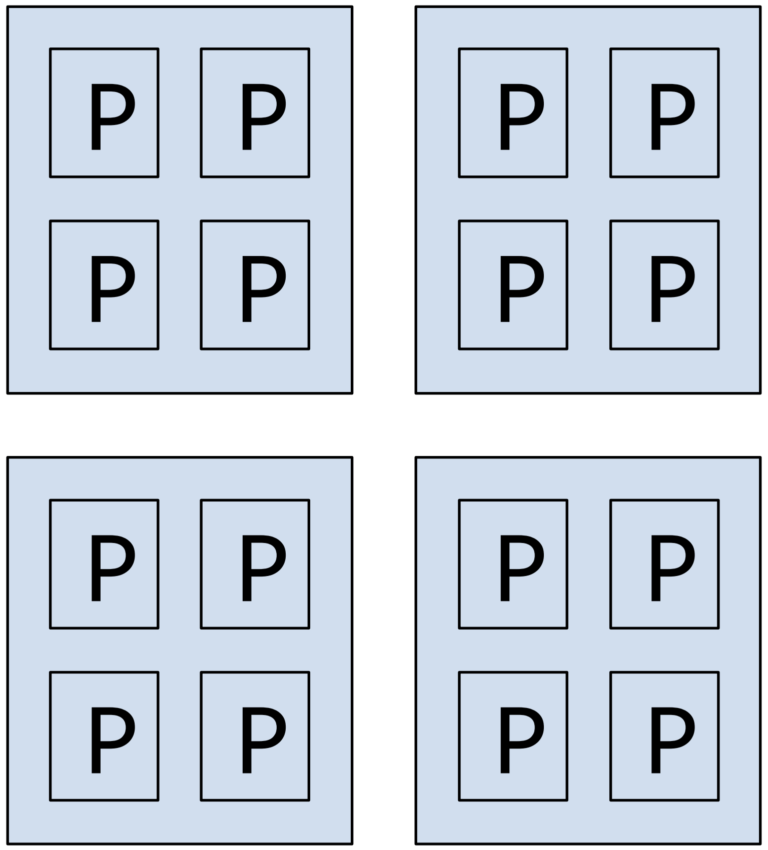
Partitioning and locality

- The blocks needs to be assigned to threads in a way that balance the work (partitioning)
  - Domain based partitioning won't do the job since the pulse moves across the domain
  - We used the Jostle diffusion partitioner
- Our problem:
  - *We need to make sure that the data in each partition is physically allocated on the node where the corresponding thread executes*

UPPSALA UNIVERSITET

Experimental Setup

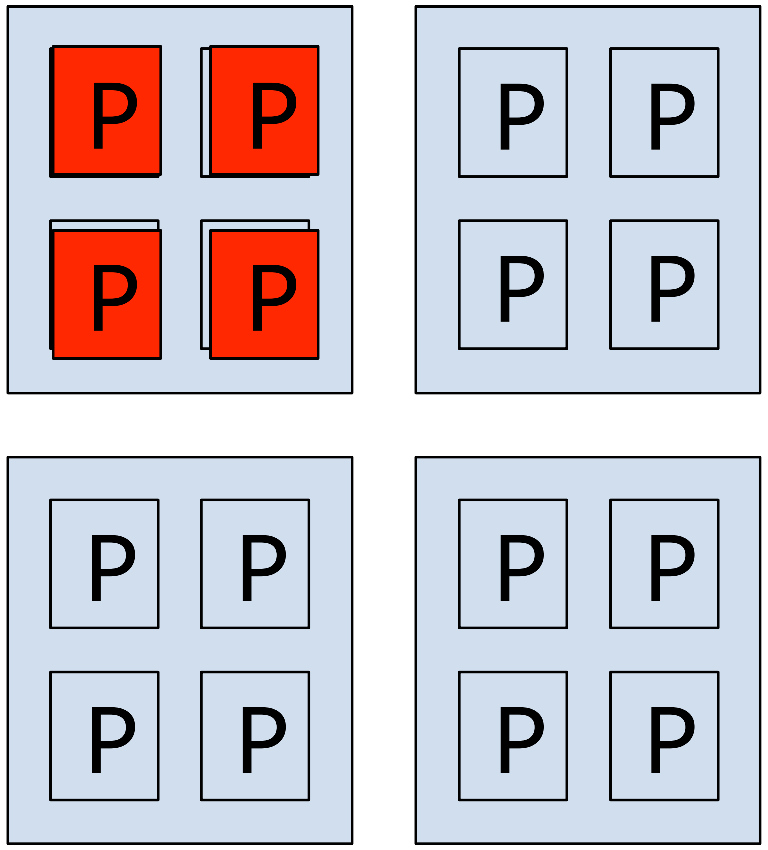
- Four nodes
  - Four CPUs per node
- Bound threads
  1. UMA
  2. NUMA
  3. NUMA-MIG
- The amount of remote accesses quantifed using CPU hardware counters



UPPSALA UNIVERSITET

Experimental Setup

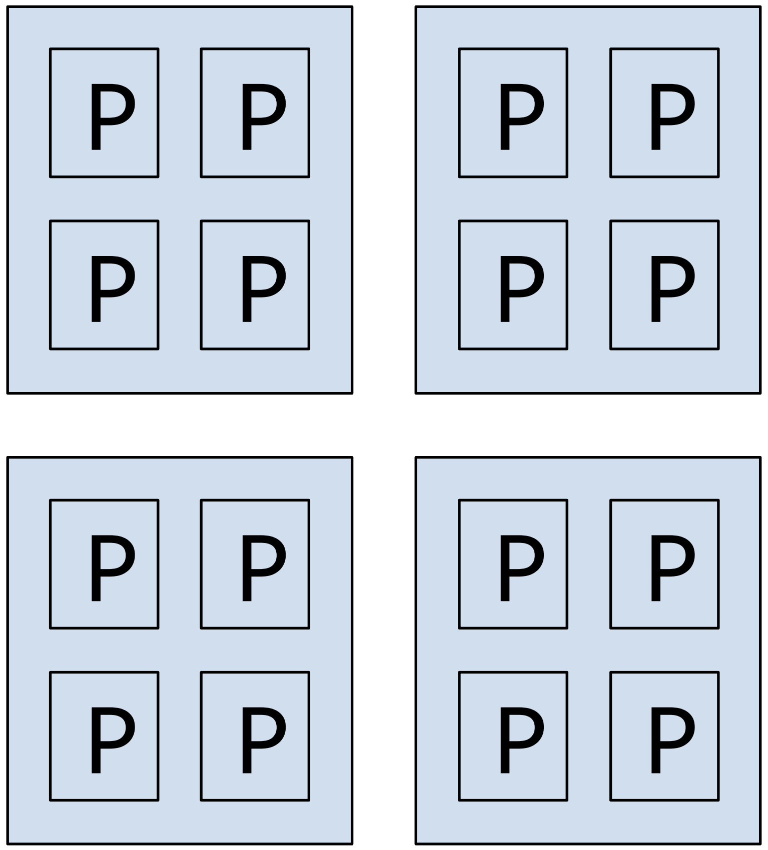
- Four nodes
  - Four CPUs per node
- Bound threads
  1. UMA
  2. NUMA
  3. NUMA-MIG
- The amount of remote accesses quantifed using CPU hardware counters



UPPSALA UNIVERSITET

Experimental Setup

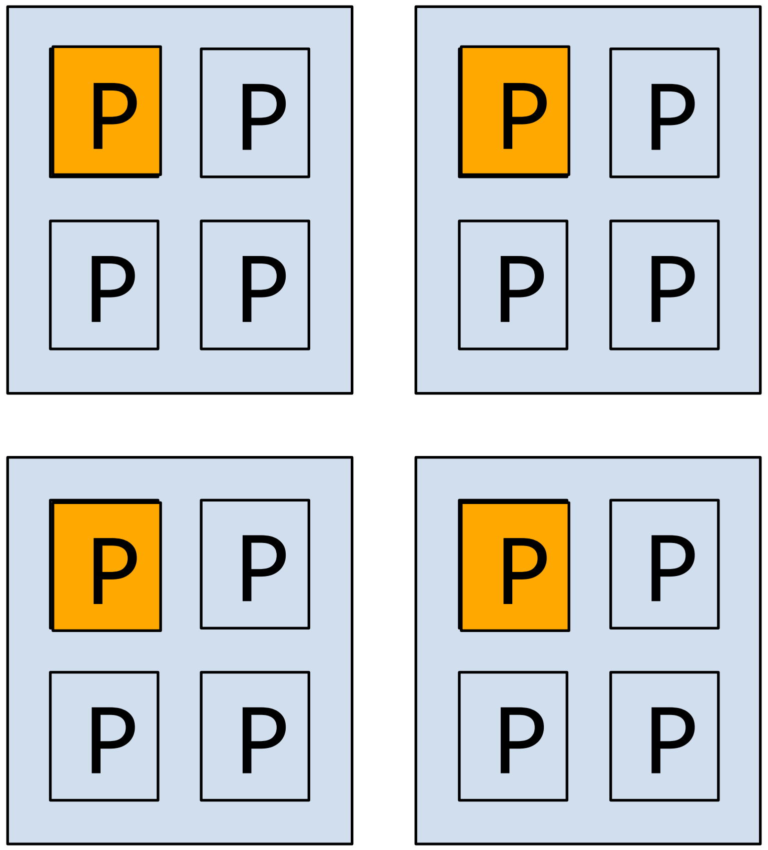
- Four nodes
  - Four CPUs per node
- Bound threads
  1. UMA
  2. NUMA
  3. NUMA-MIG
- The amount of remote accesses quantifed using CPU hardware counters



UPPSALA UNIVERSITET

Experimental Setup

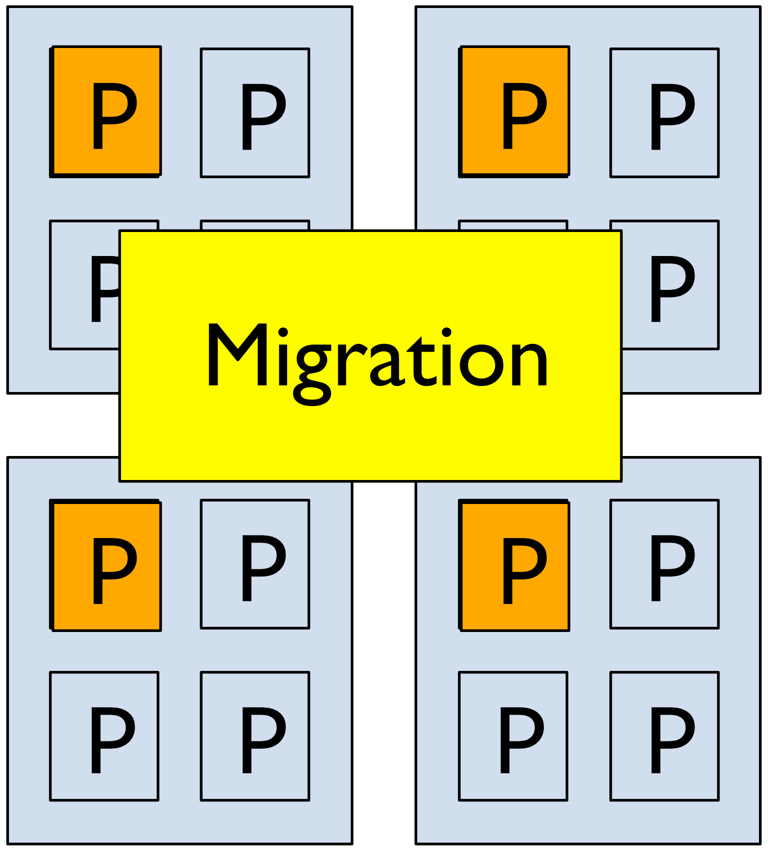
- Four nodes
  - Four CPUs per node
- Bound threads
  1. UMA
  2. NUMA
  3. NUMA-MIG
- The amount of remote accesses quantifed using CPU hardware counters



UPPSALA UNIVERSITET

Experimental Setup

- Four nodes
  - Four CPUs per node
- Bound threads
  1. UMA
  2. NUMA
  3. NUMA-MIG
- The amount of remote accesses quantifed using CPU hardware counters



UPPSALA
UNIVERSITET

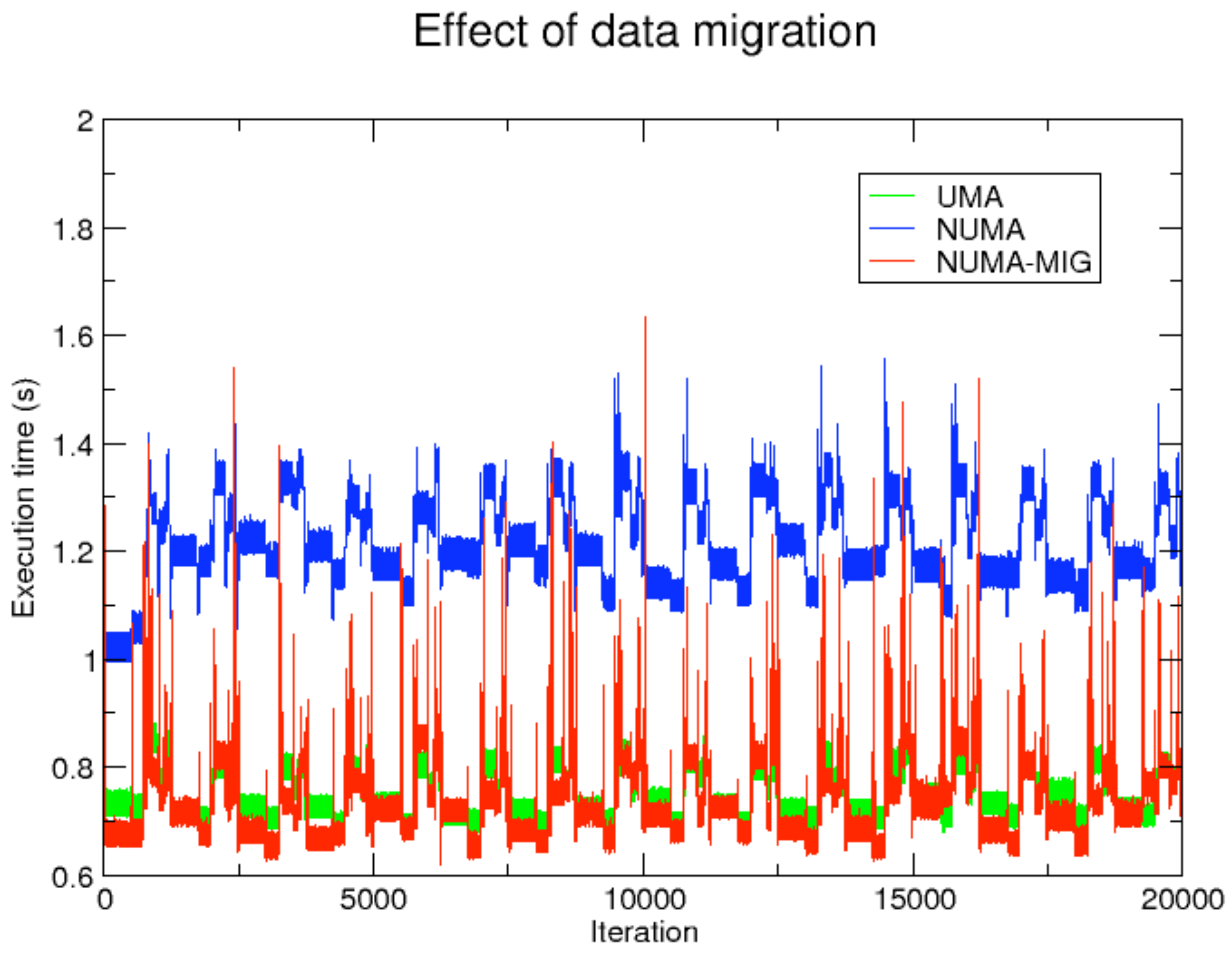
Results

	UMA	NUMA	NUMA-MIG
Total time	4.09 h	6.64 h	3.99 h
L2 miss rate	4.3%	3.9%	4.2%
Remote accesses	0.2%	62.9%	8.1%

- Geographical locality has a significant effect!
- Dynamic page migration works!

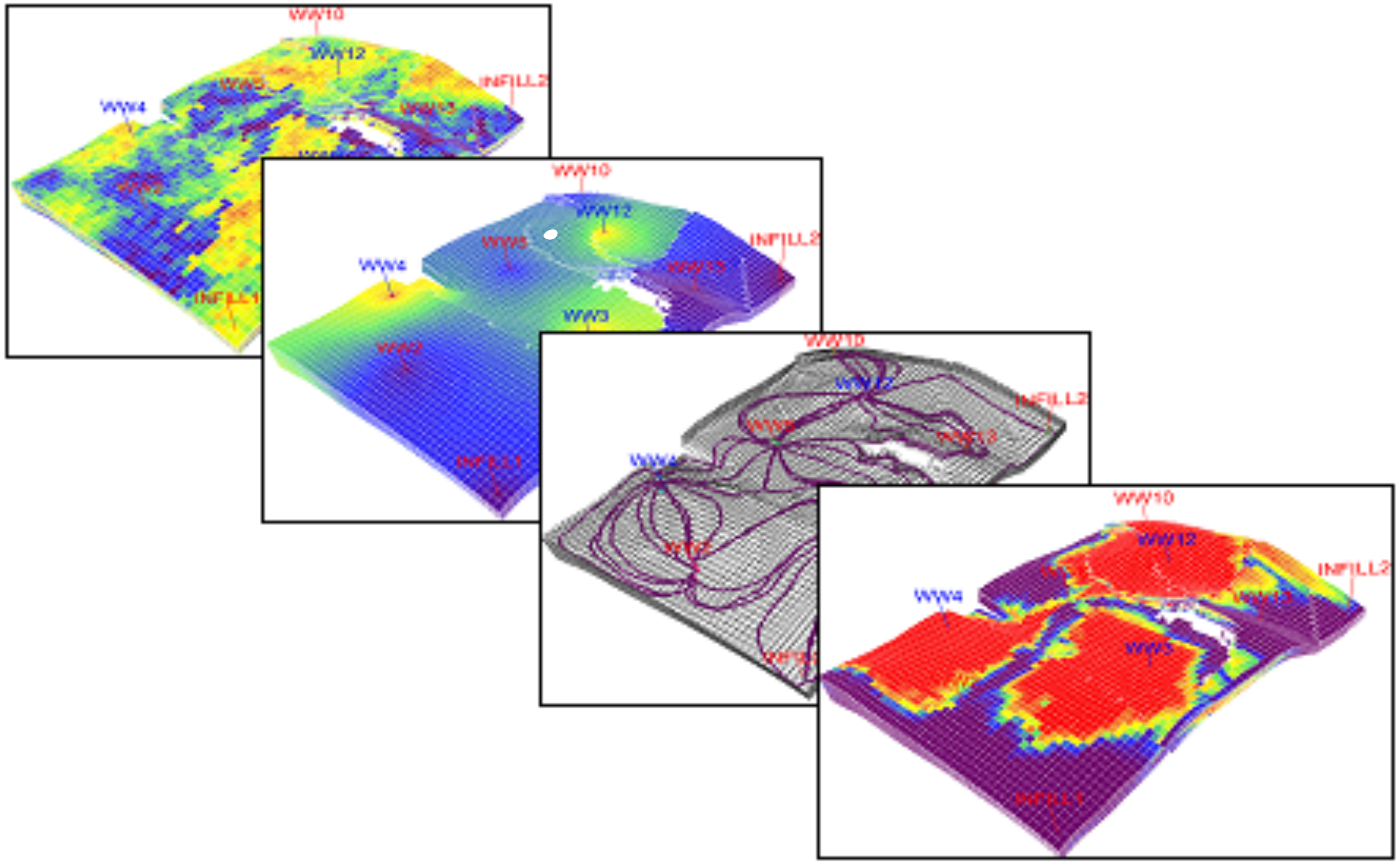
UPPSALA UNIVERSITET

Execution Time



UPPSALA
UNIVERSITET

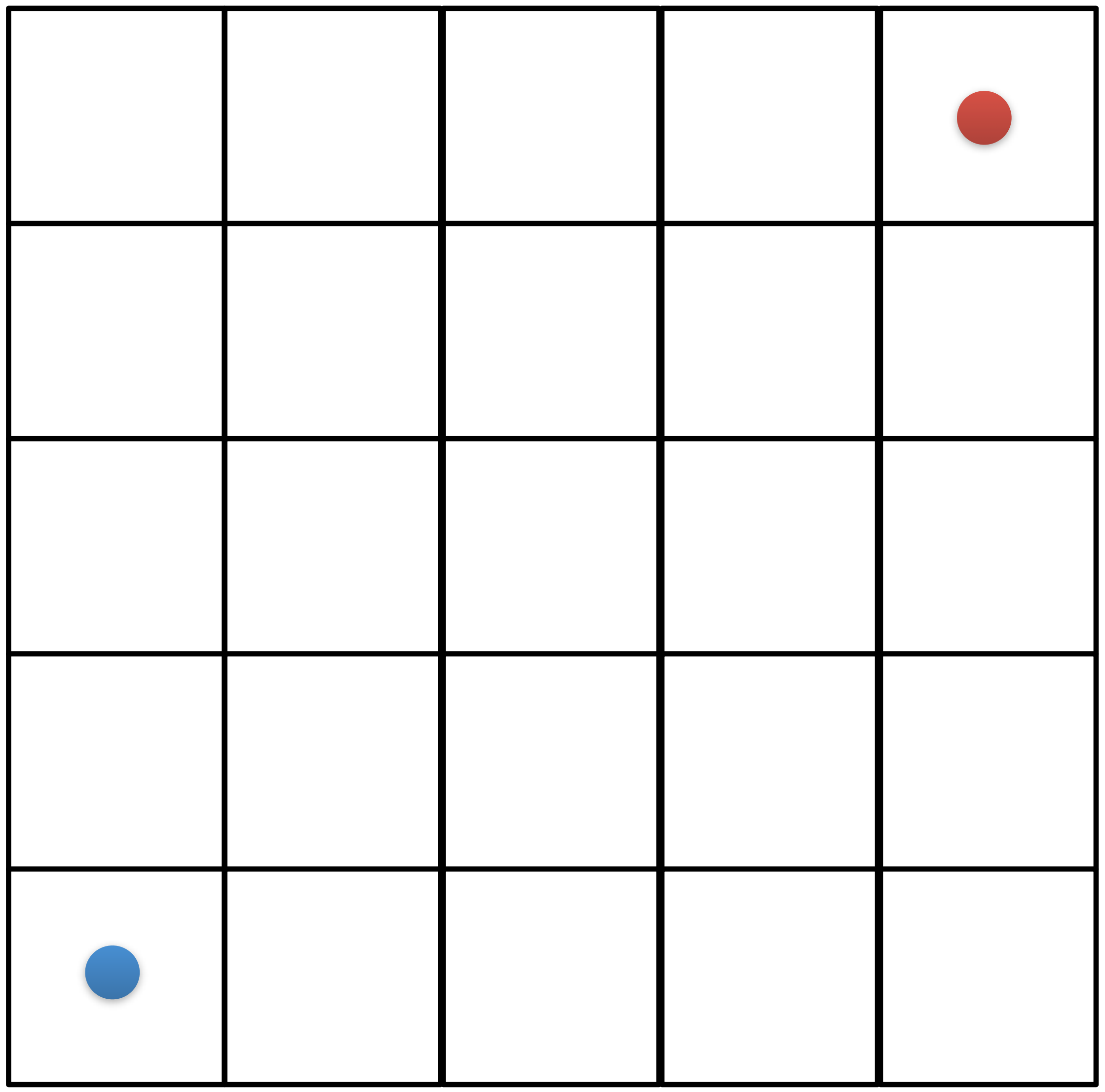
Streamline Simulation



UPPSALA UNIVERSITET

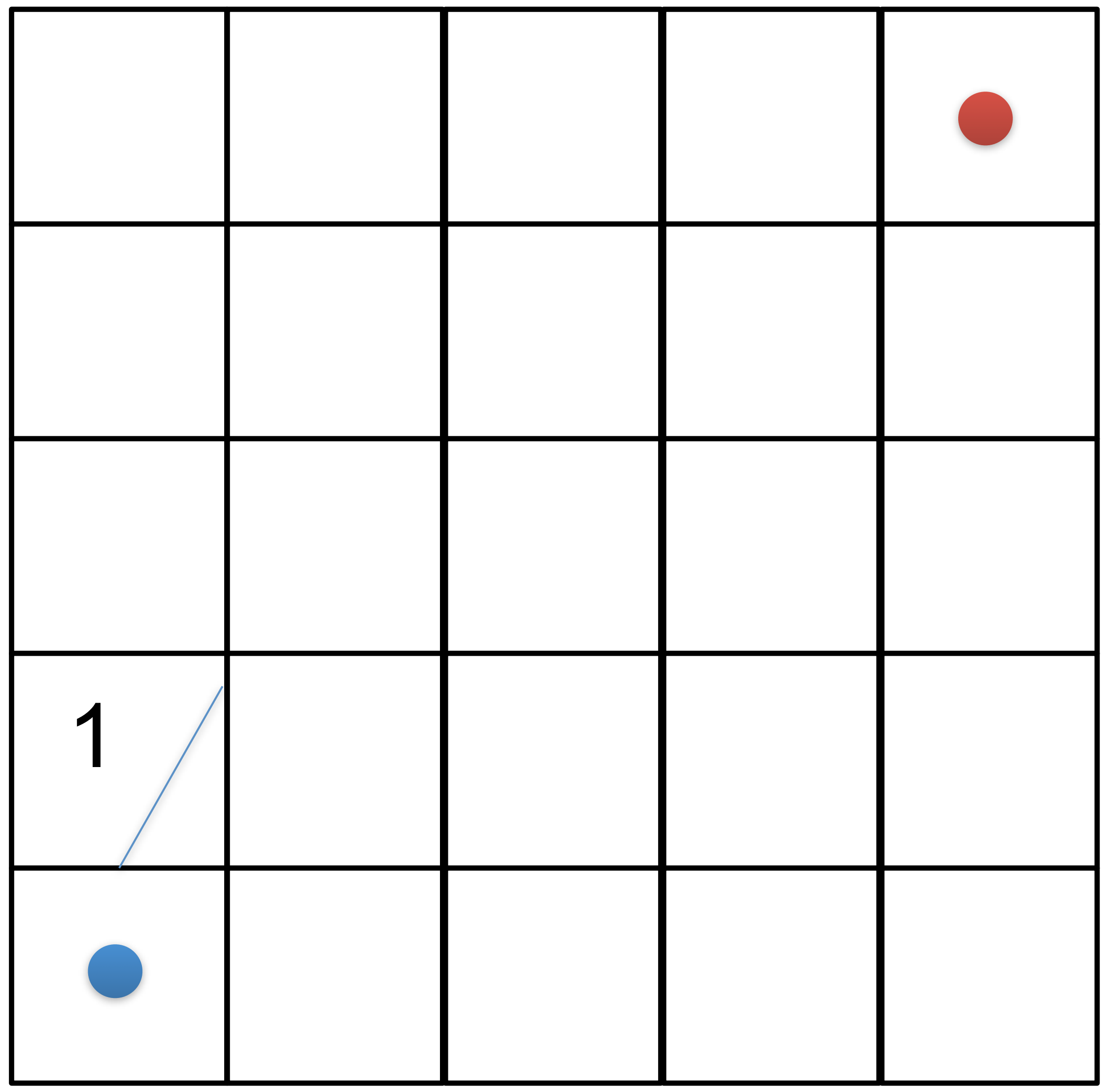
Streamline Tracing

Segments



UPPSALA UNIVERSITET

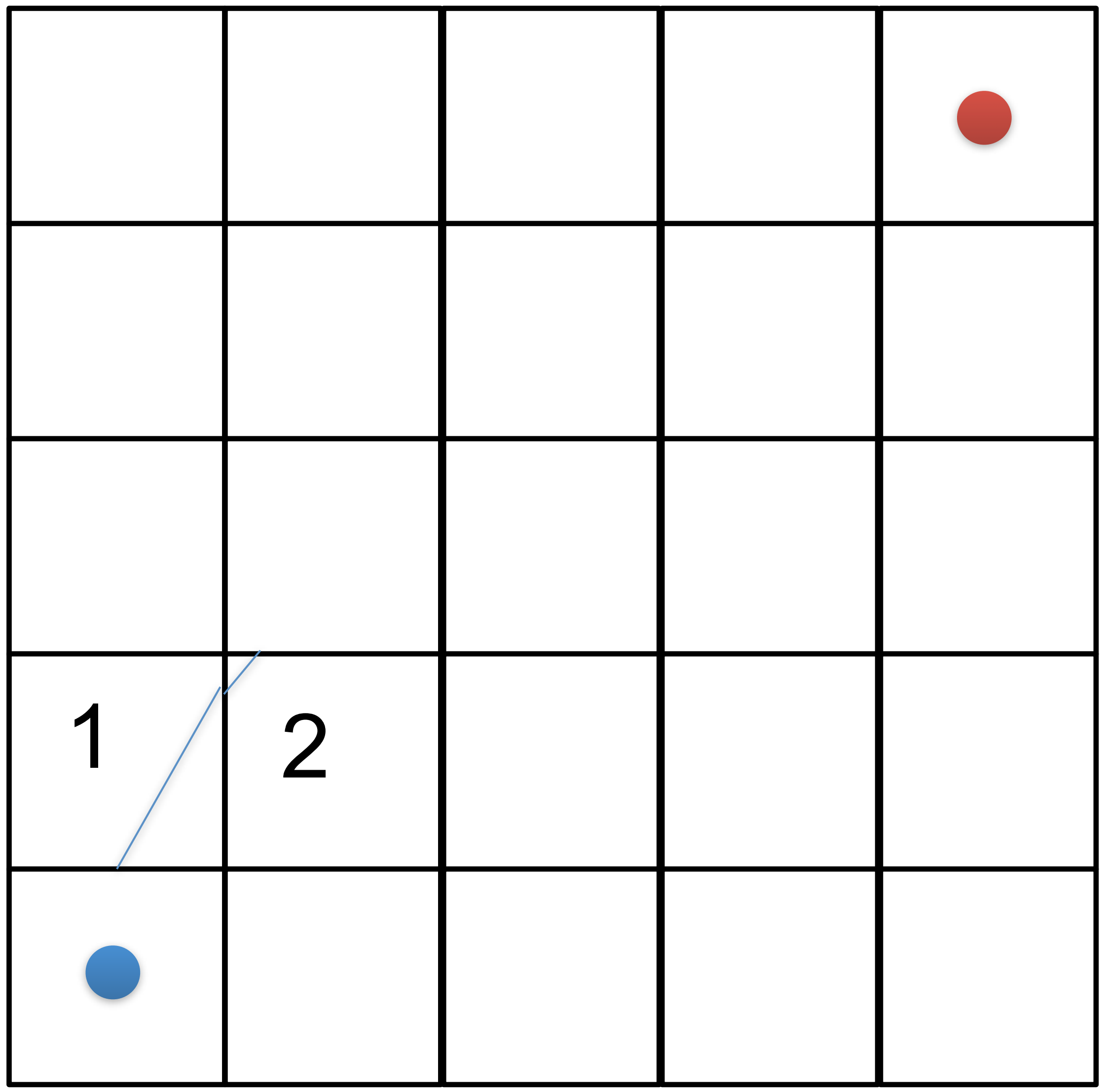
Streamline Tracing



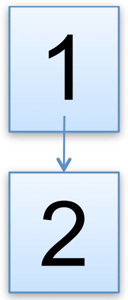
Segments

UPPSALA UNIVERSITET

Streamline Tracing

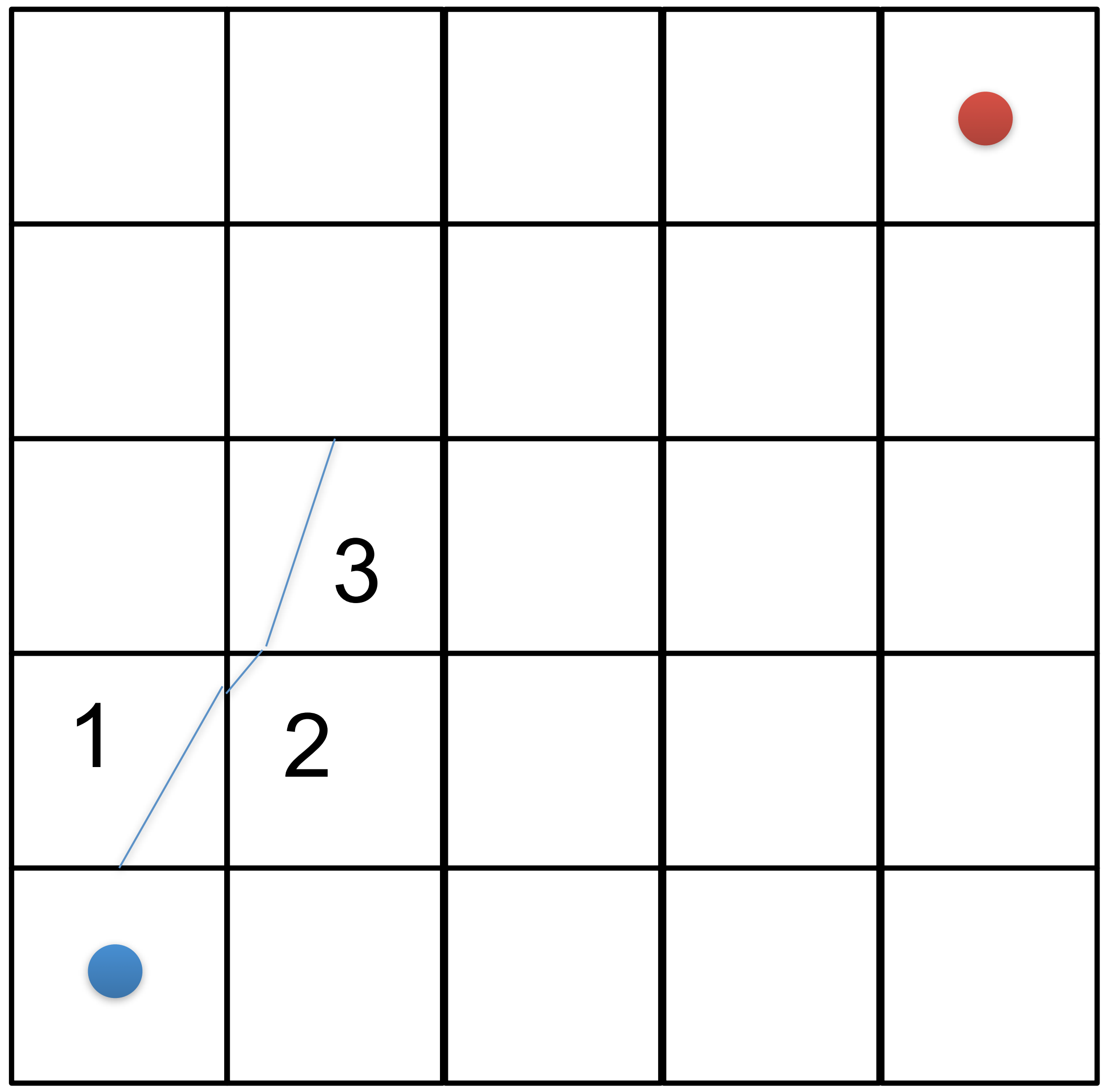


Segments

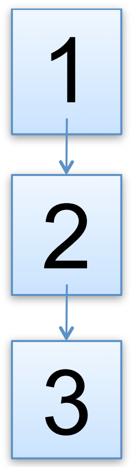


UPPSALA UNIVERSITET

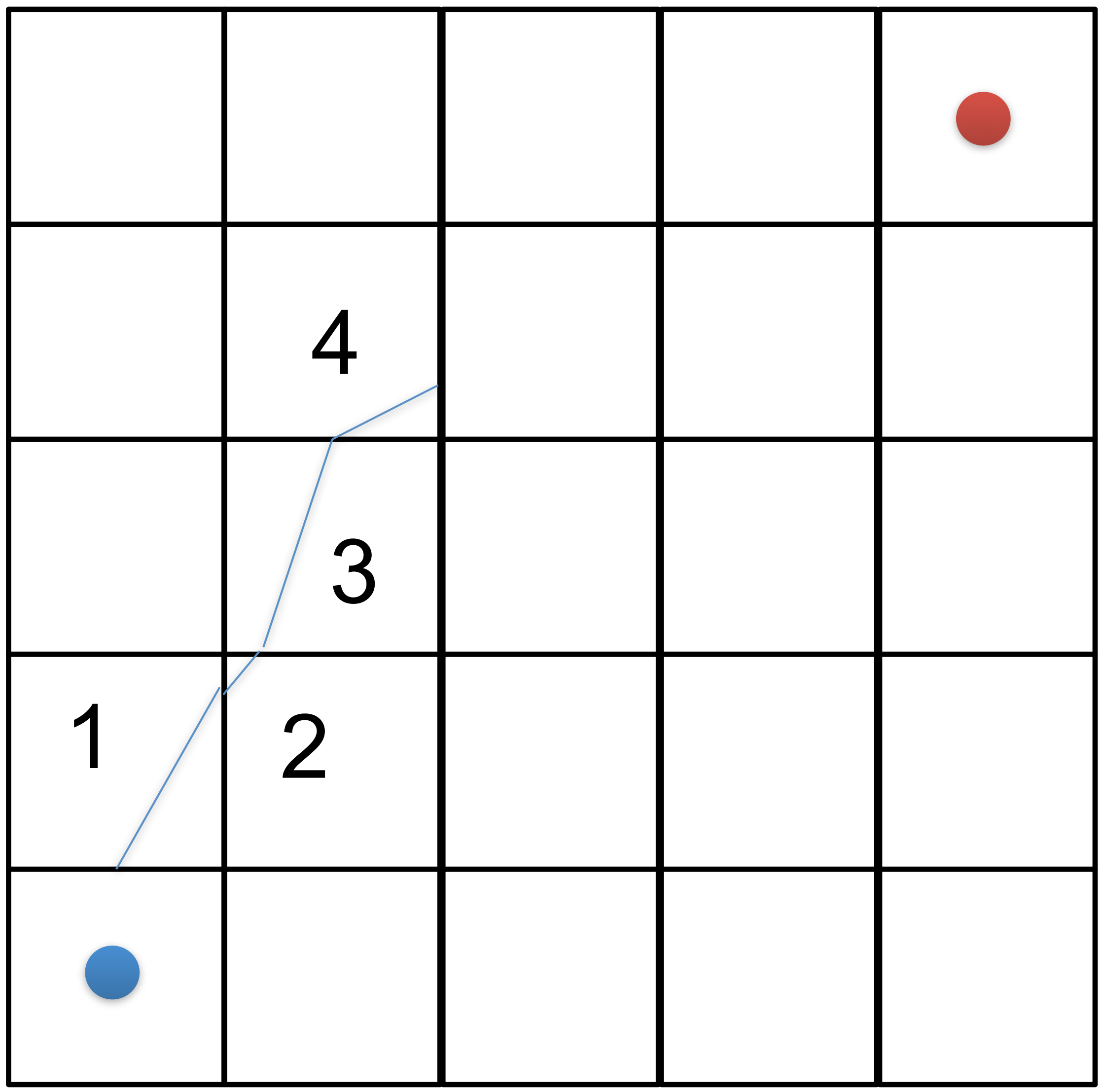
Streamline Tracing



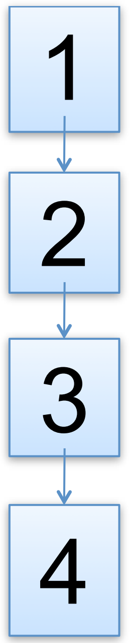
Segments



Streamline Tracing

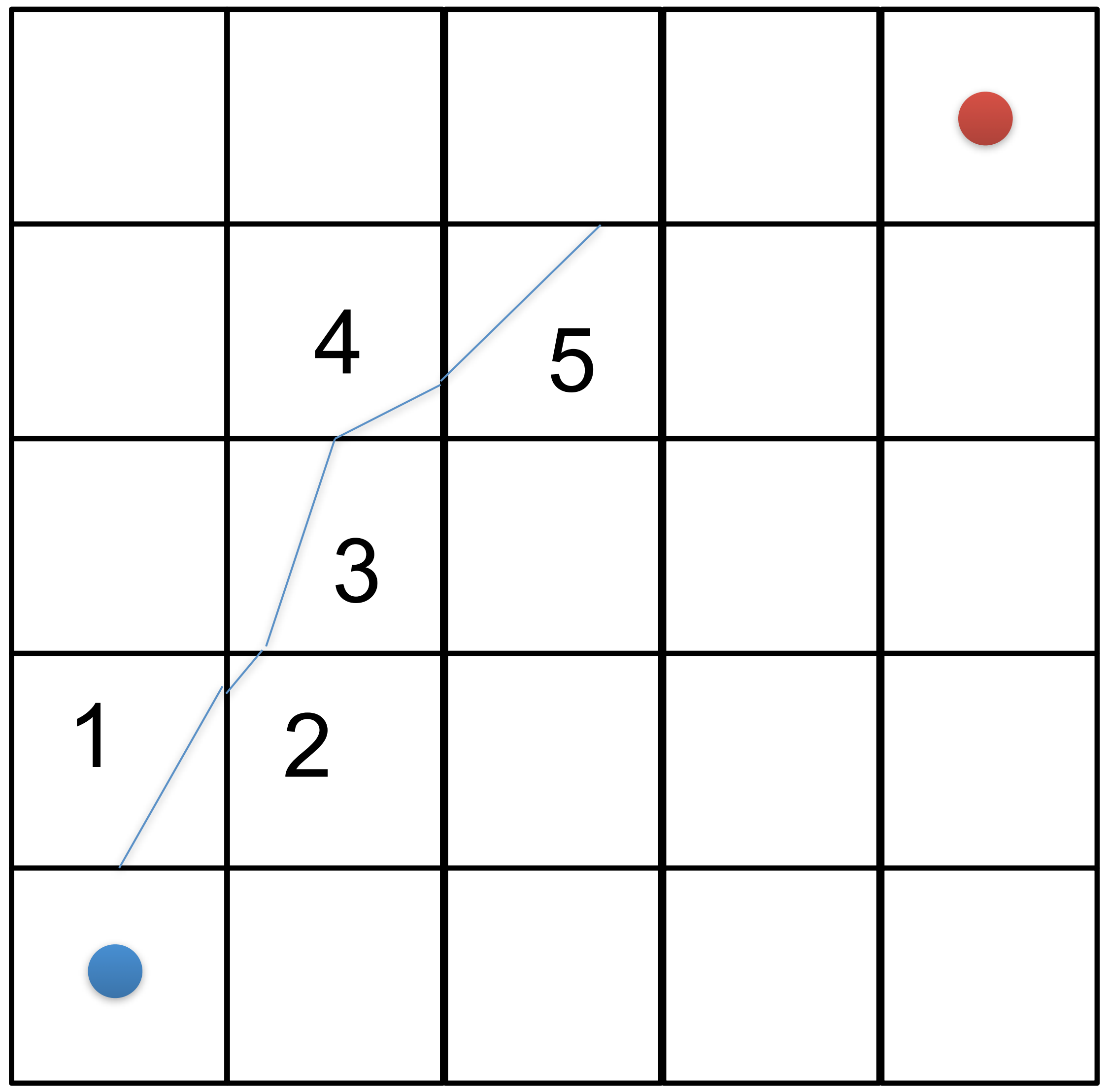


Segments

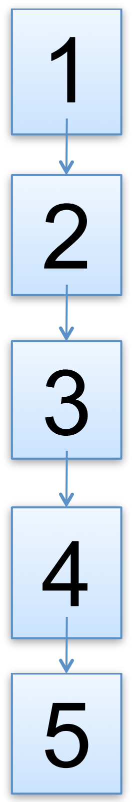


UPPSALA UNIVERSITET

Streamline Tracing

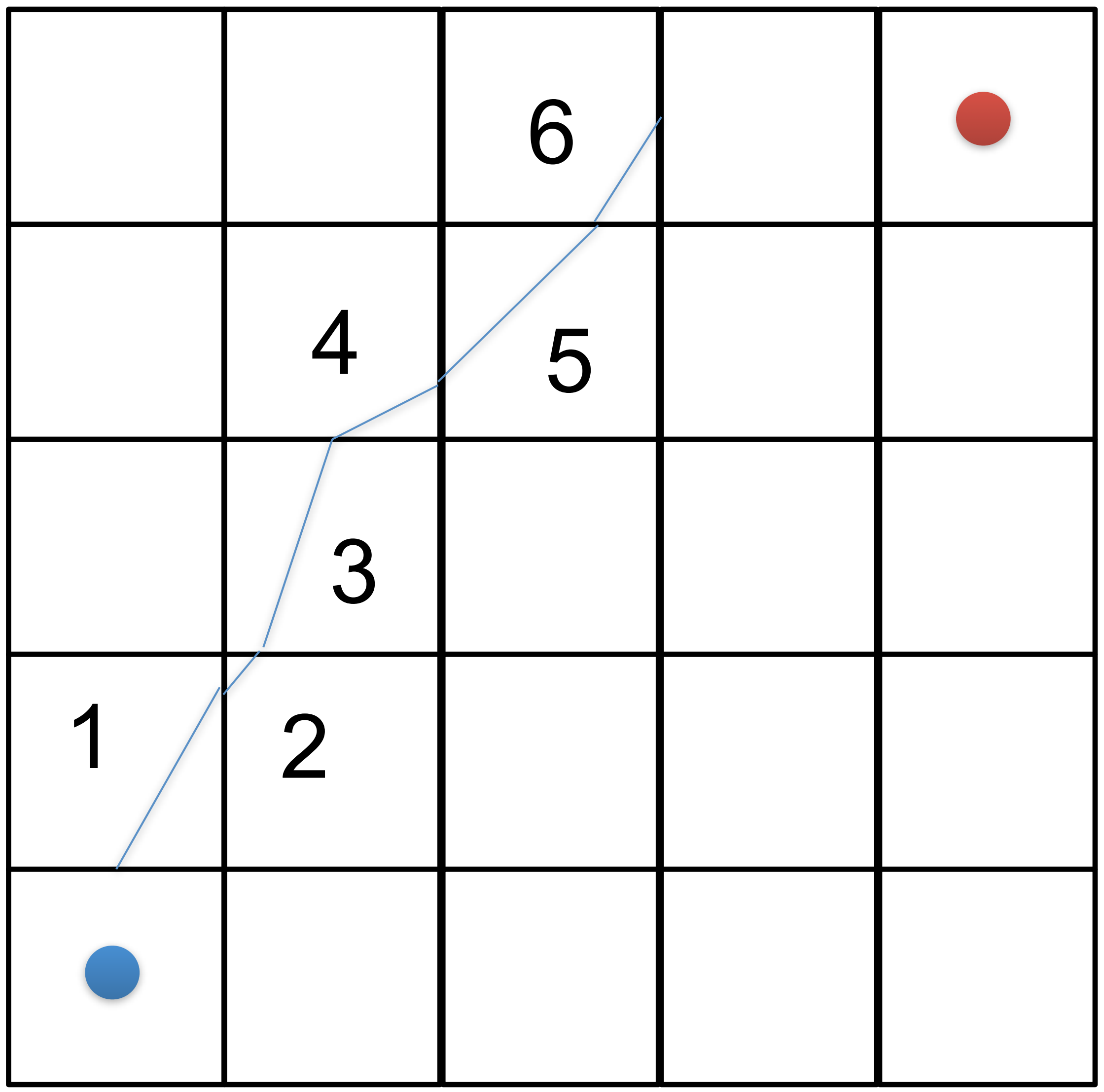


Segments

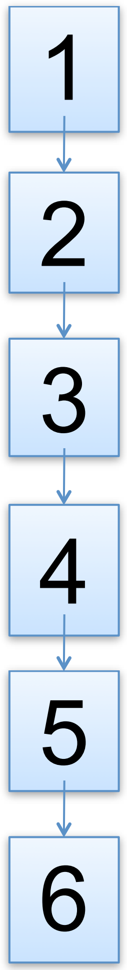


UPPSALA UNIVERSITET

Streamline Tracing

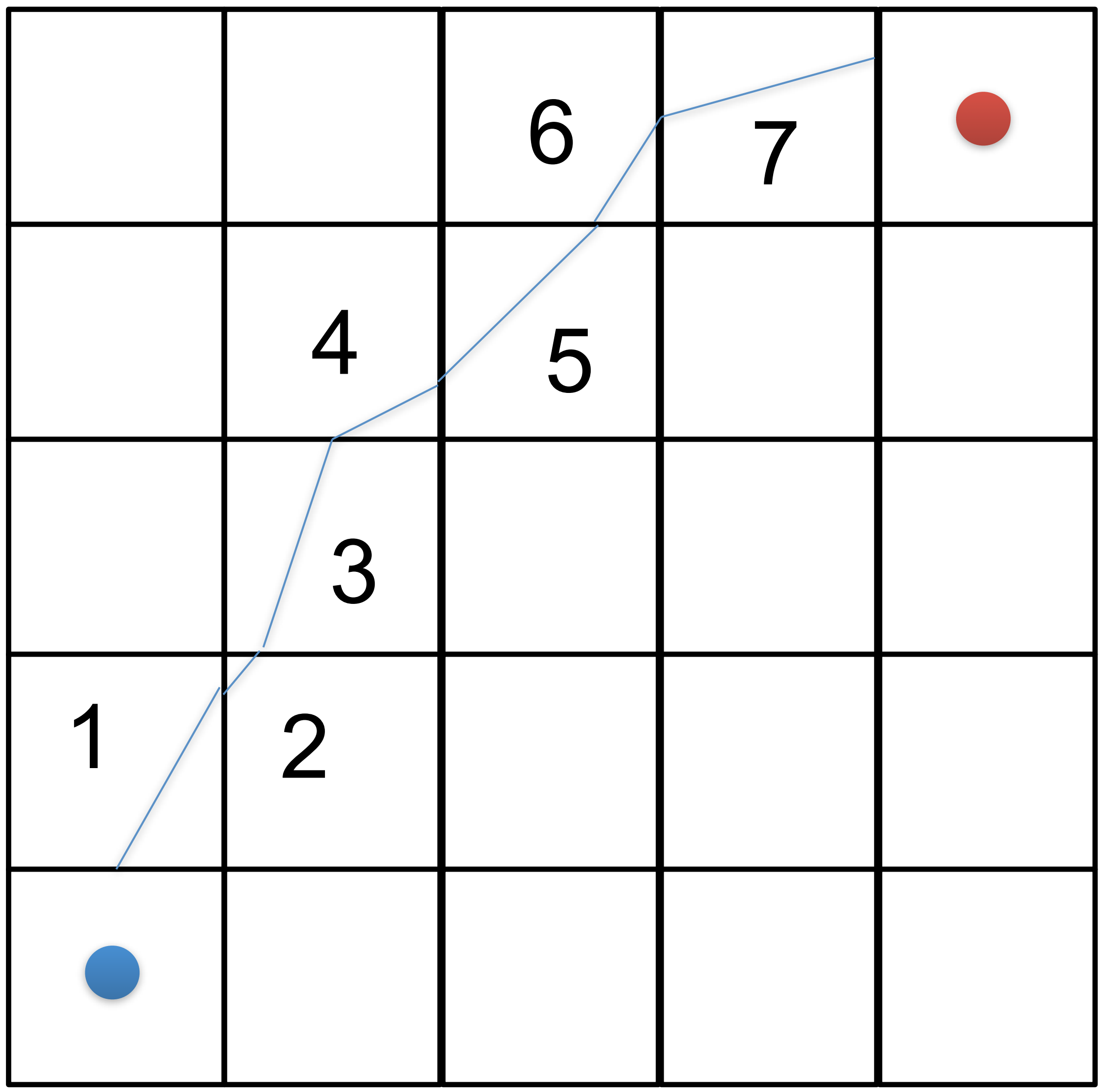


Segments

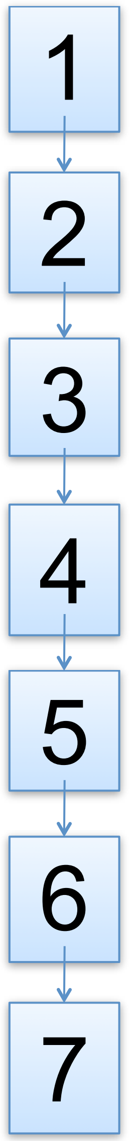


UPPSALA UNIVERSITET

Streamline Tracing



Segments

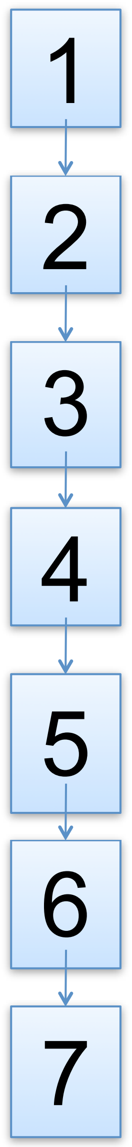


UPPSALA
UNIVERSITET

Streamline Storage

- The geometry of the streamlines is stored as a sequence of coordinates
- To store the streamlines we have several options

1. Allocate huge array and hope for the best
2. Use a dynamic array or a linked list
3. Trace once to count the segments, allocate an array and trace again

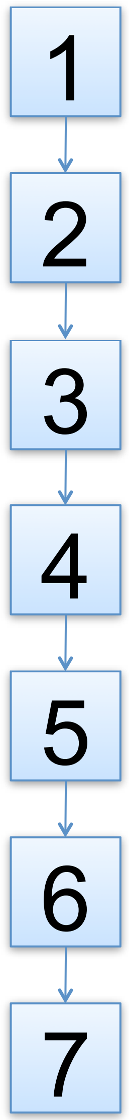


1-D Solves. Illustration

UPPSALA
UNIVERSITET

1-D Solves. Illustration

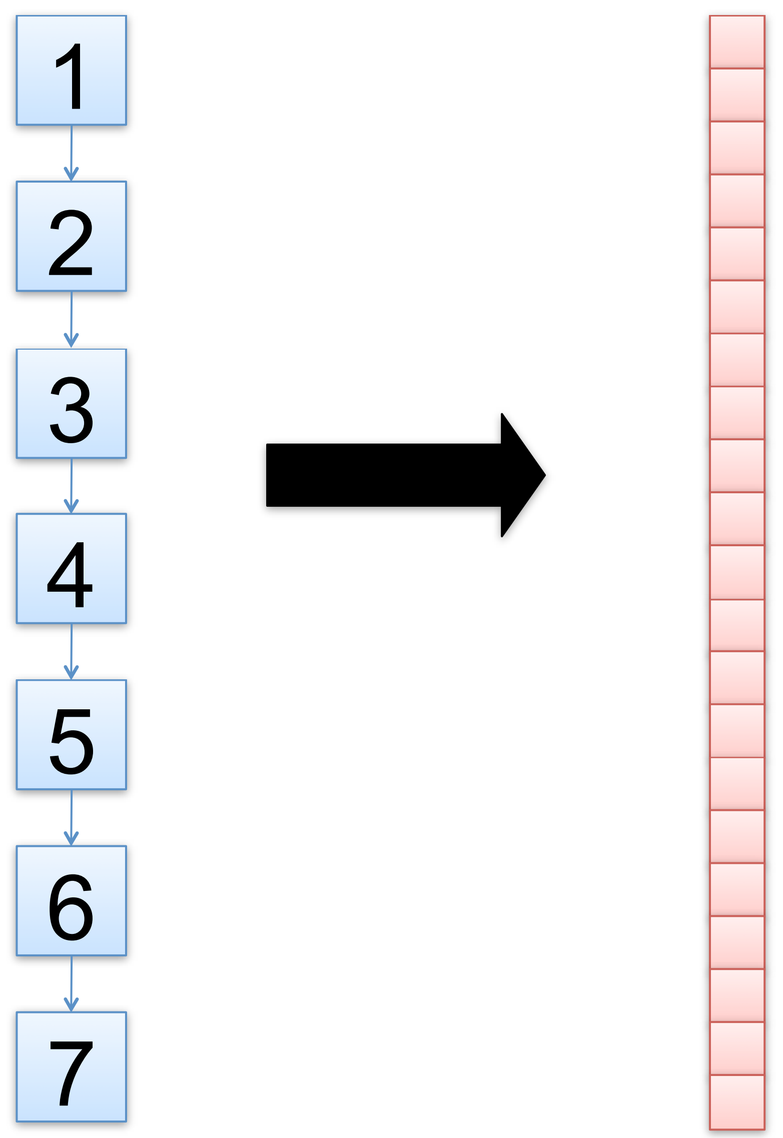
Segments



UPPSALA
UNIVERSITET

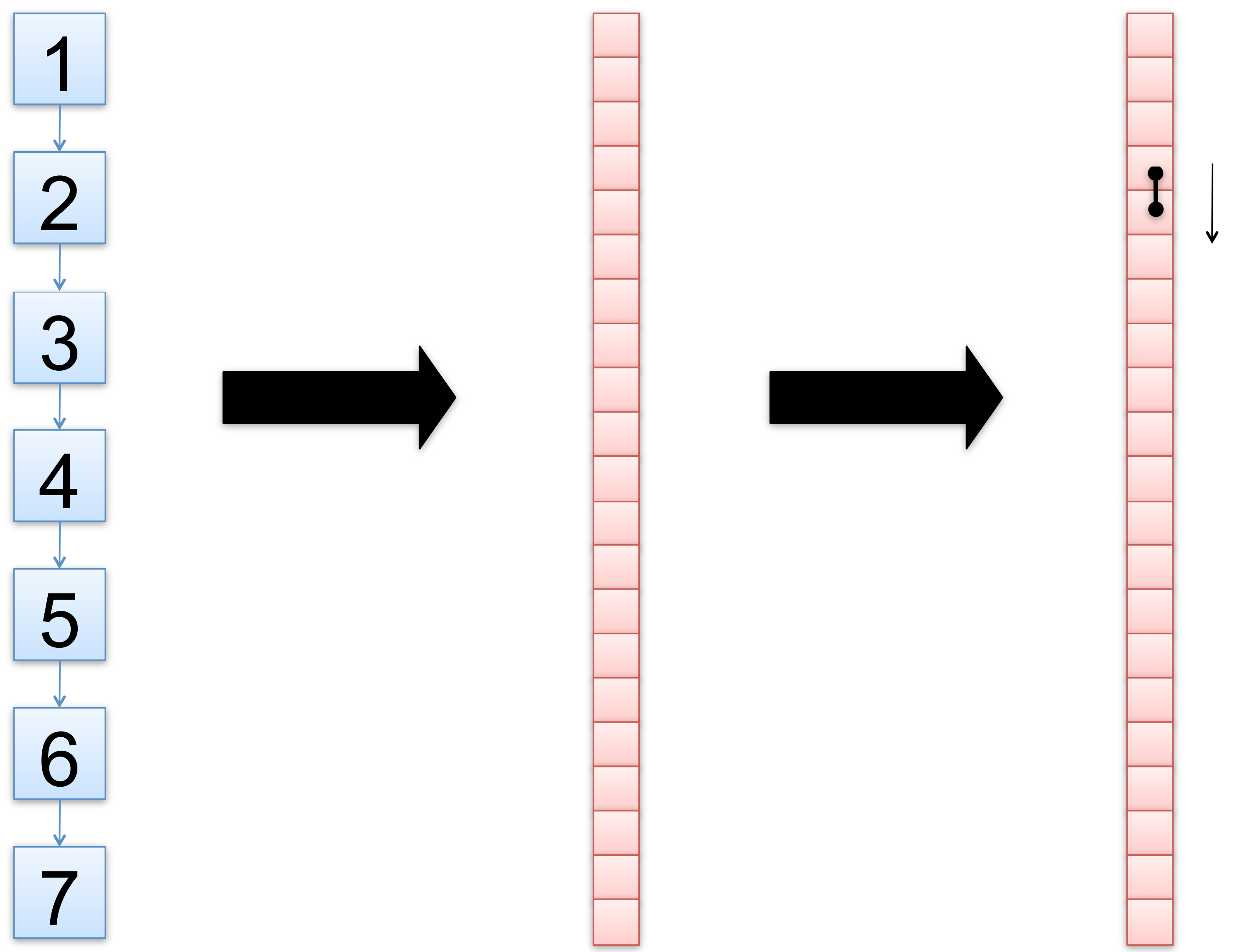
1-D Solves. Illustration

Segments Regularized Grid



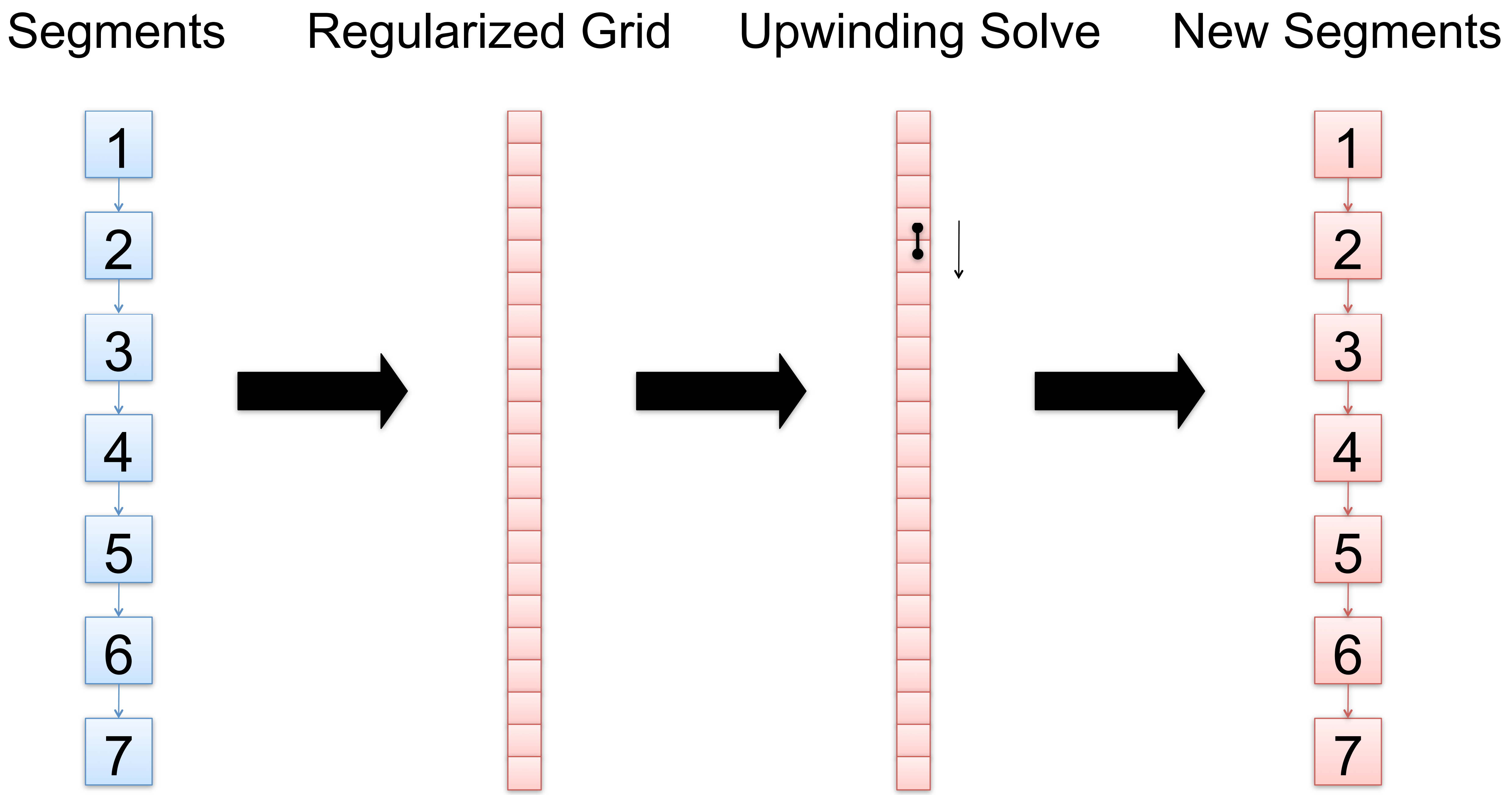
1-D Solves. Illustration

Segments Regularized Grid Upwinding Solve



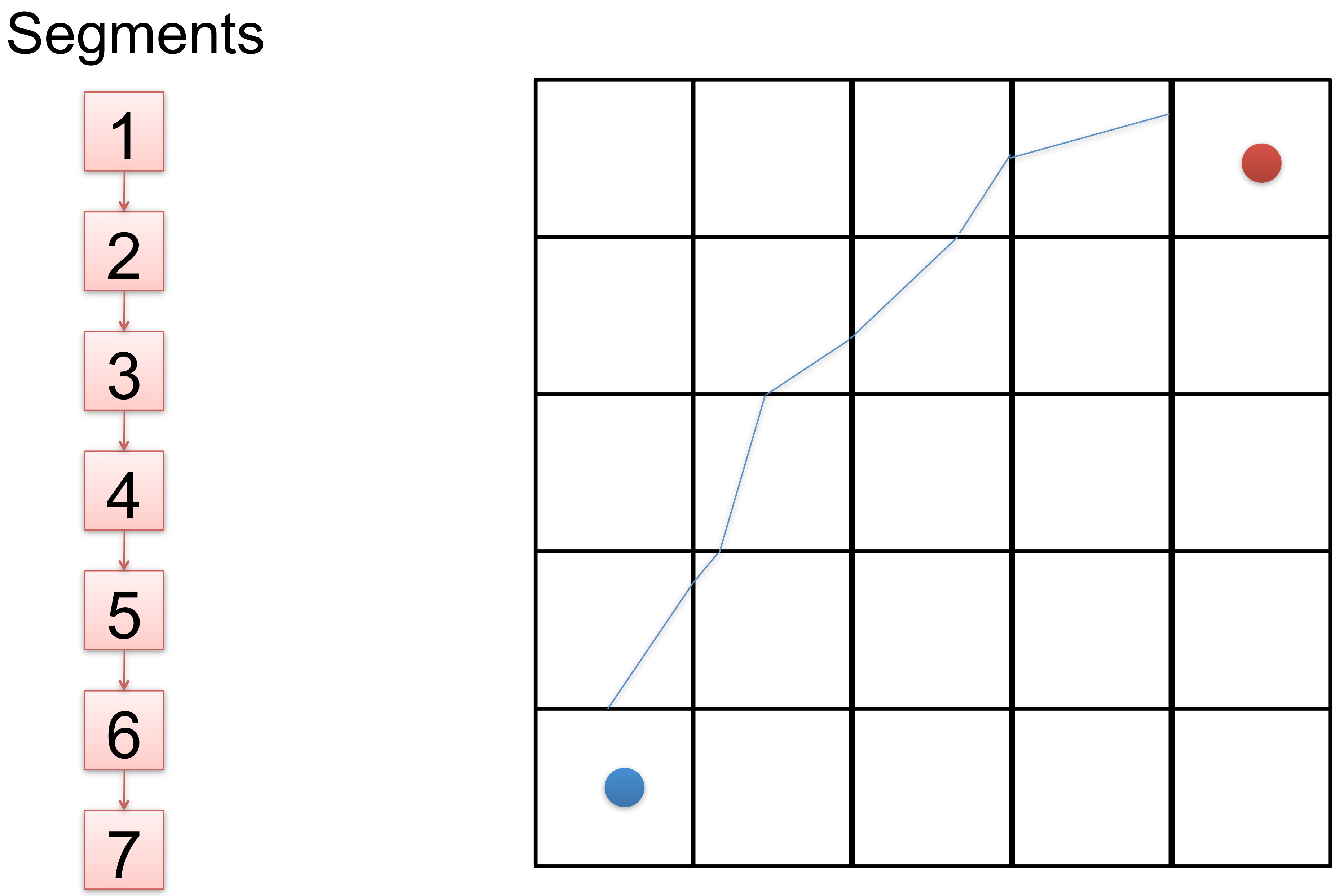
UPPSALA UNIVERSITET

1-D Solves. Illustration



UPPSALA UNIVERSITET

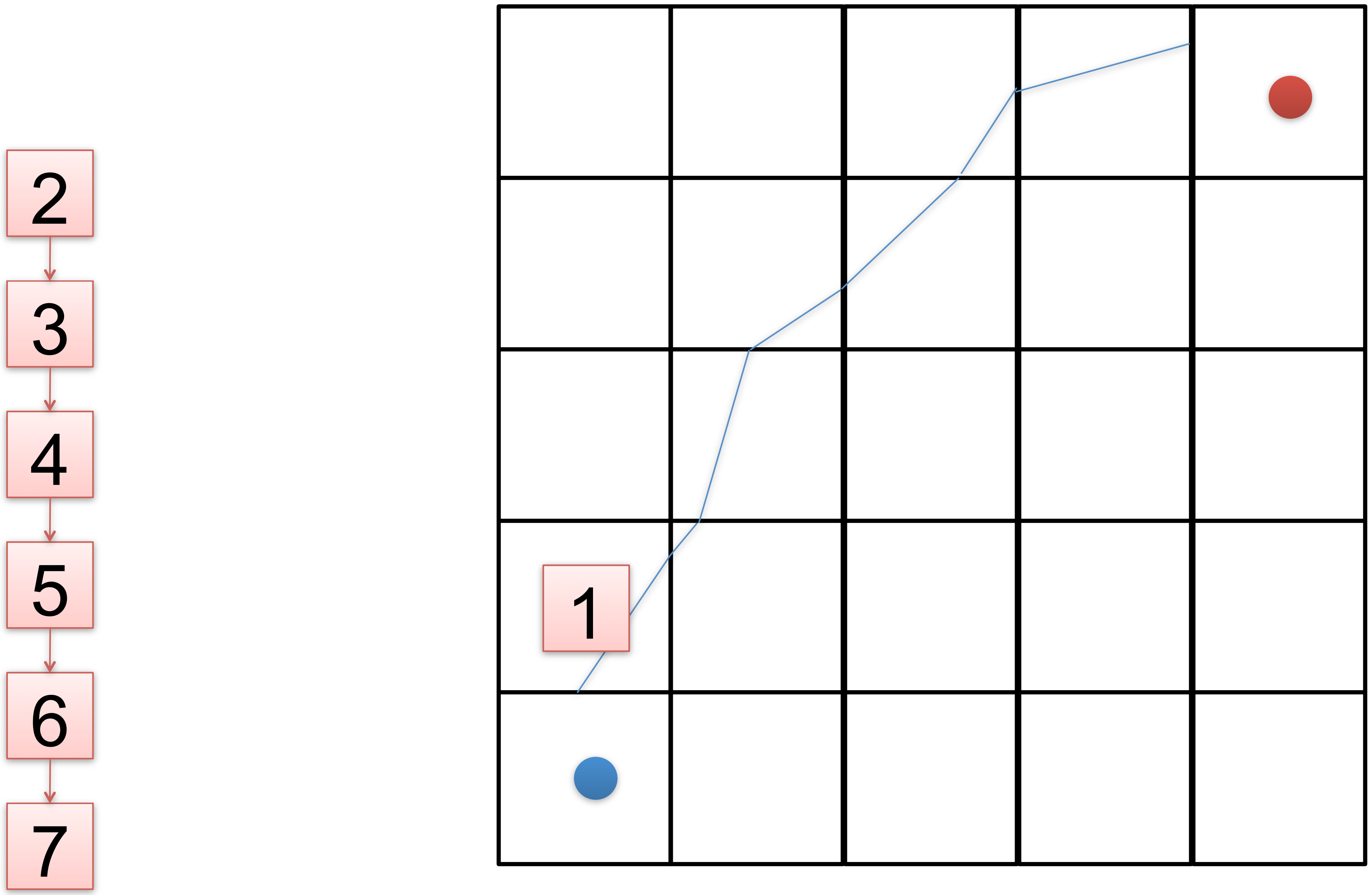
Segment Mapping, illustration



UPPSALA UNIVERSITET

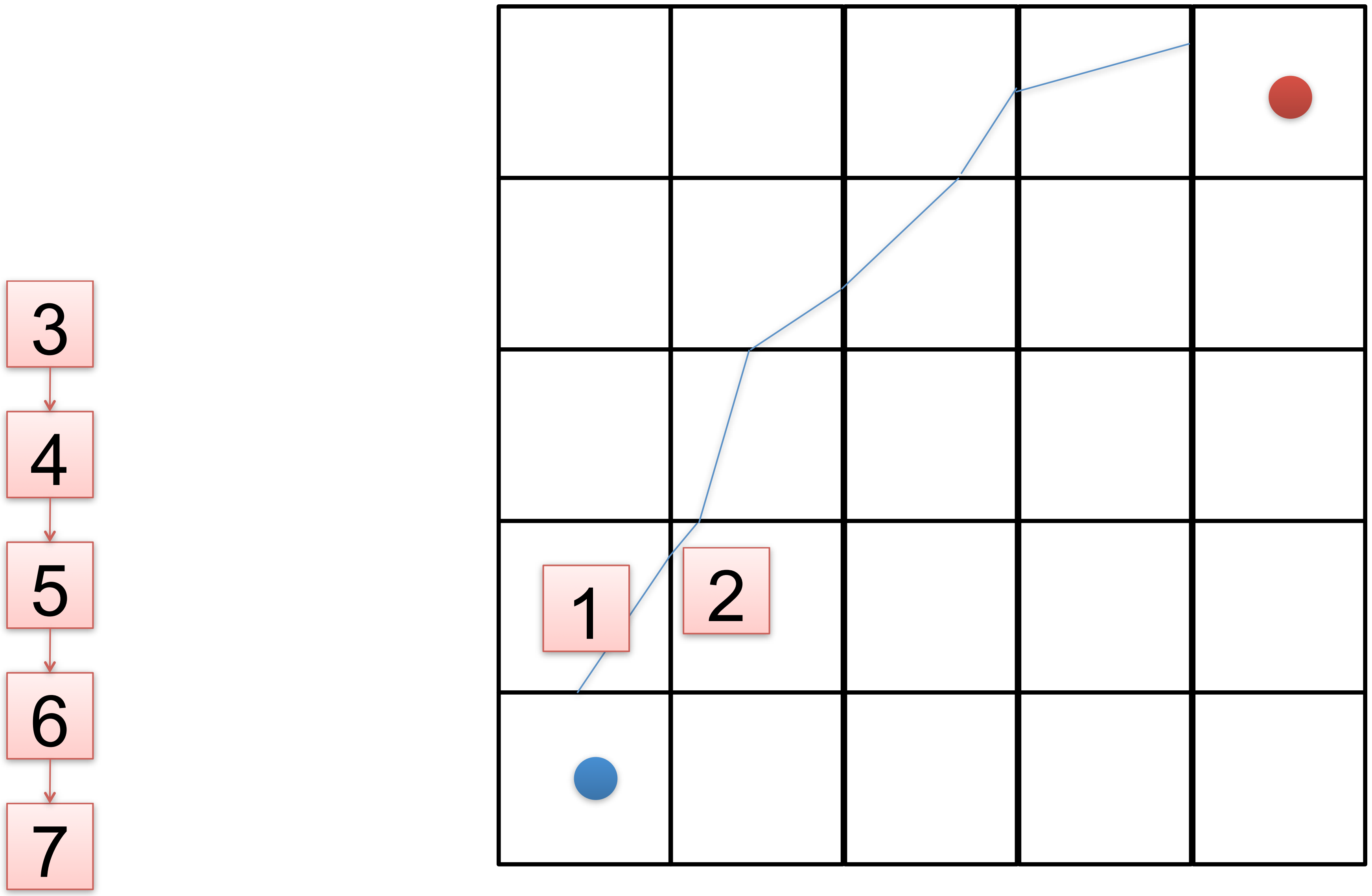
Segment Mapping, illustration

Segments



Segment Mapping, illustration

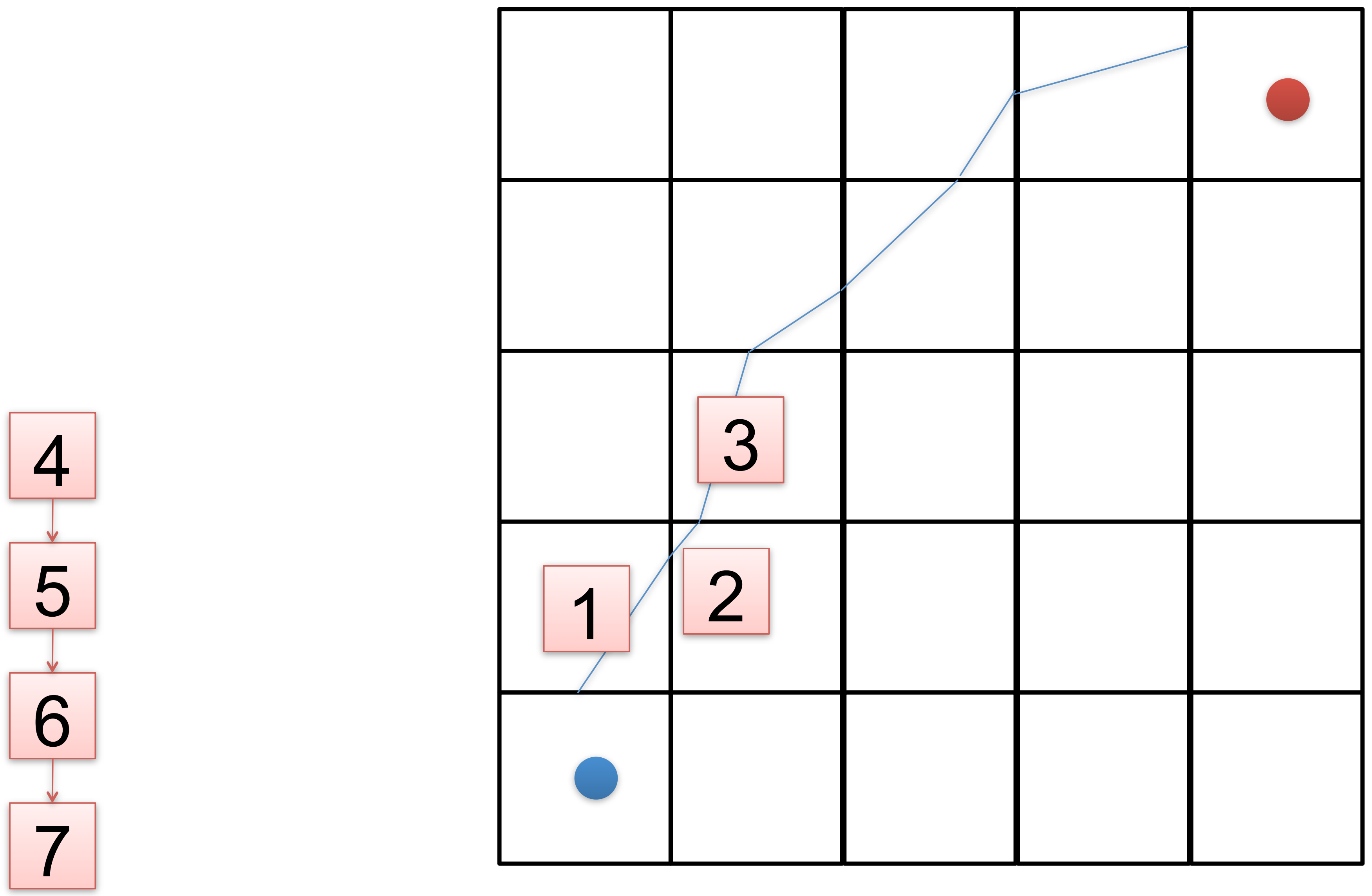
Segments



UPPSALA UNIVERSITET

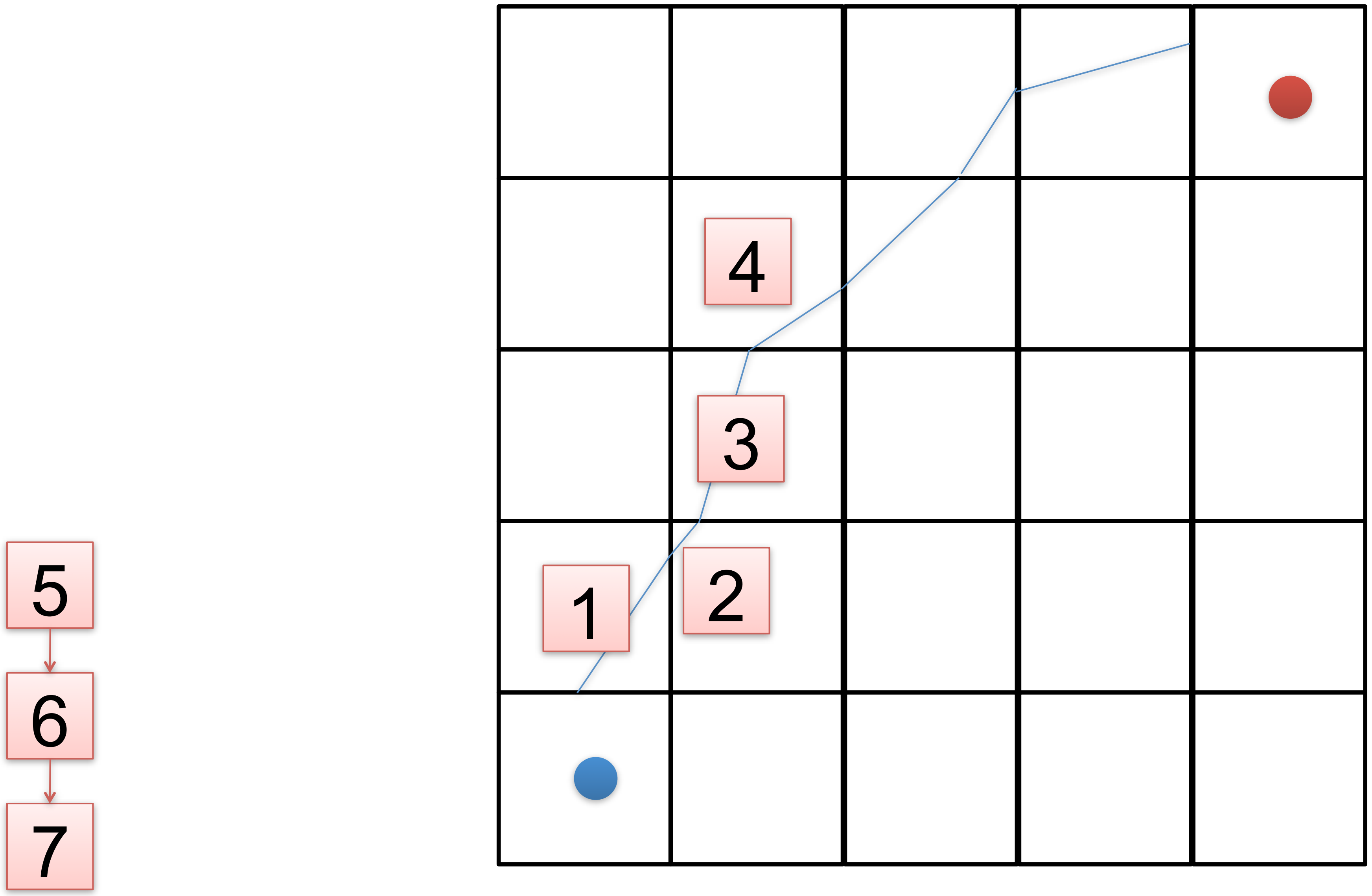
Segment Mapping, illustration

Segments



Segment Mapping, illustration

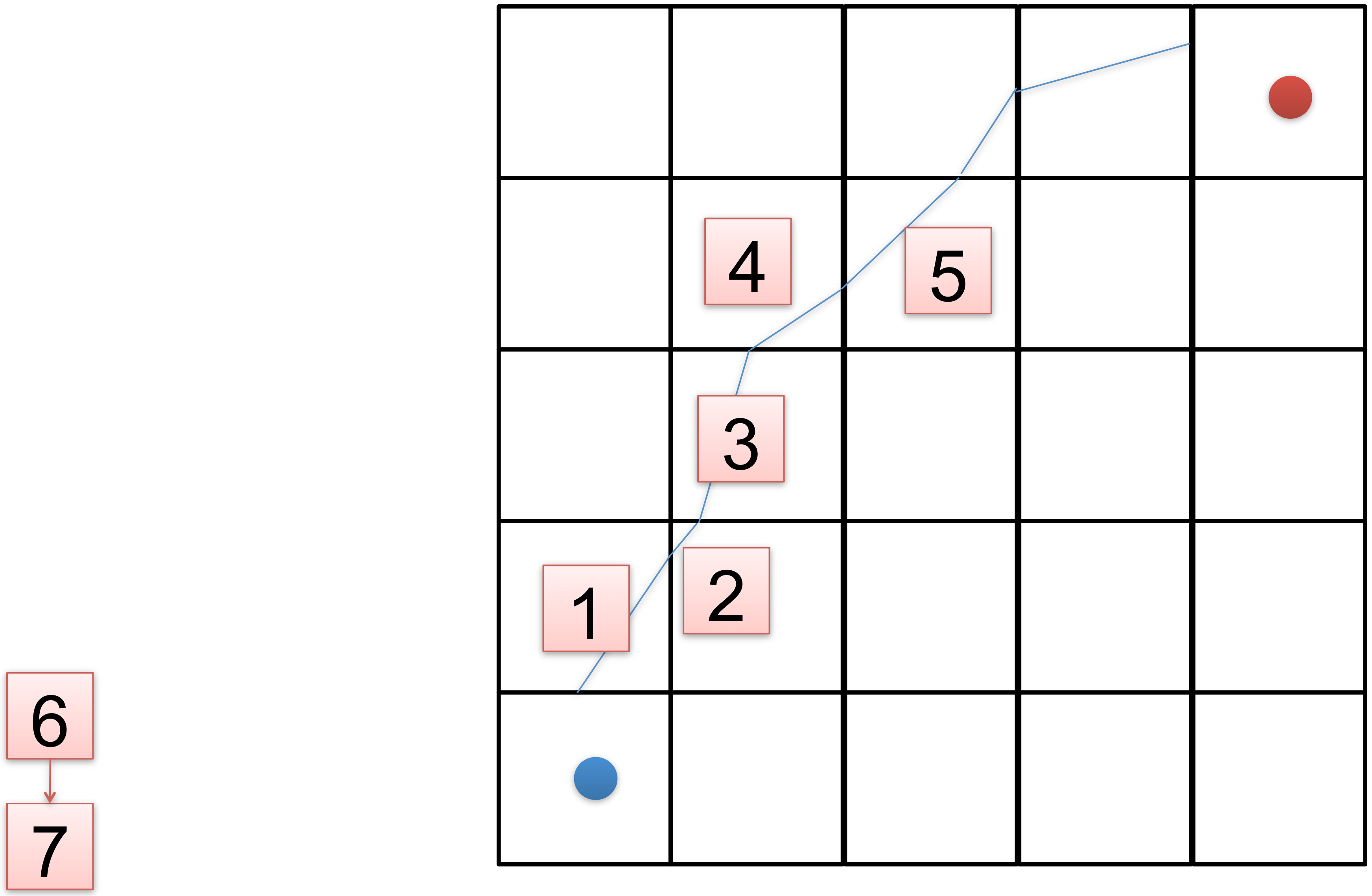
Segments



UPPSALA
UNIVERSITET

Segment Mapping, illustration

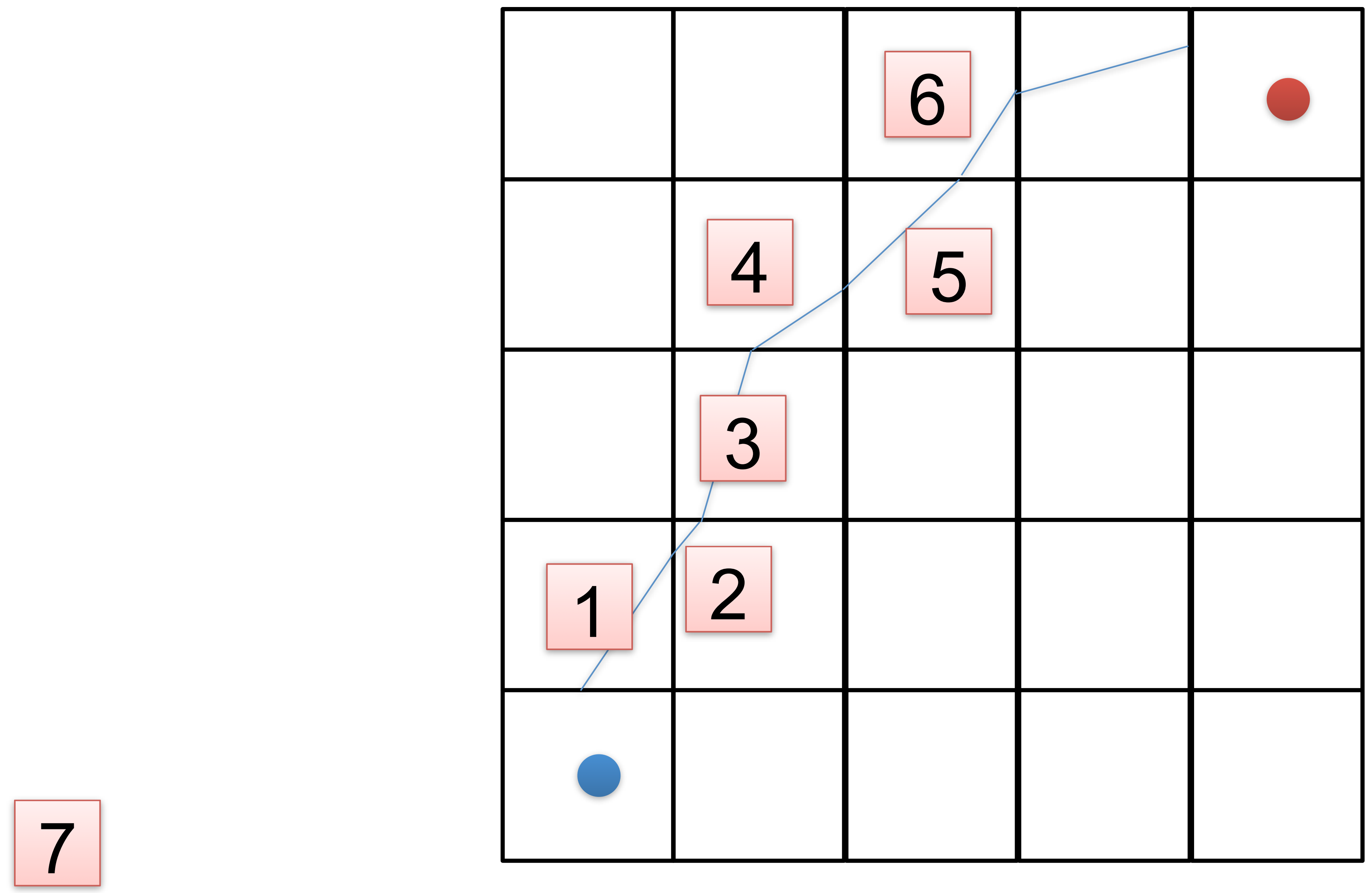
Segments



UPPSALA UNIVERSITET

Segment Mapping, illustration

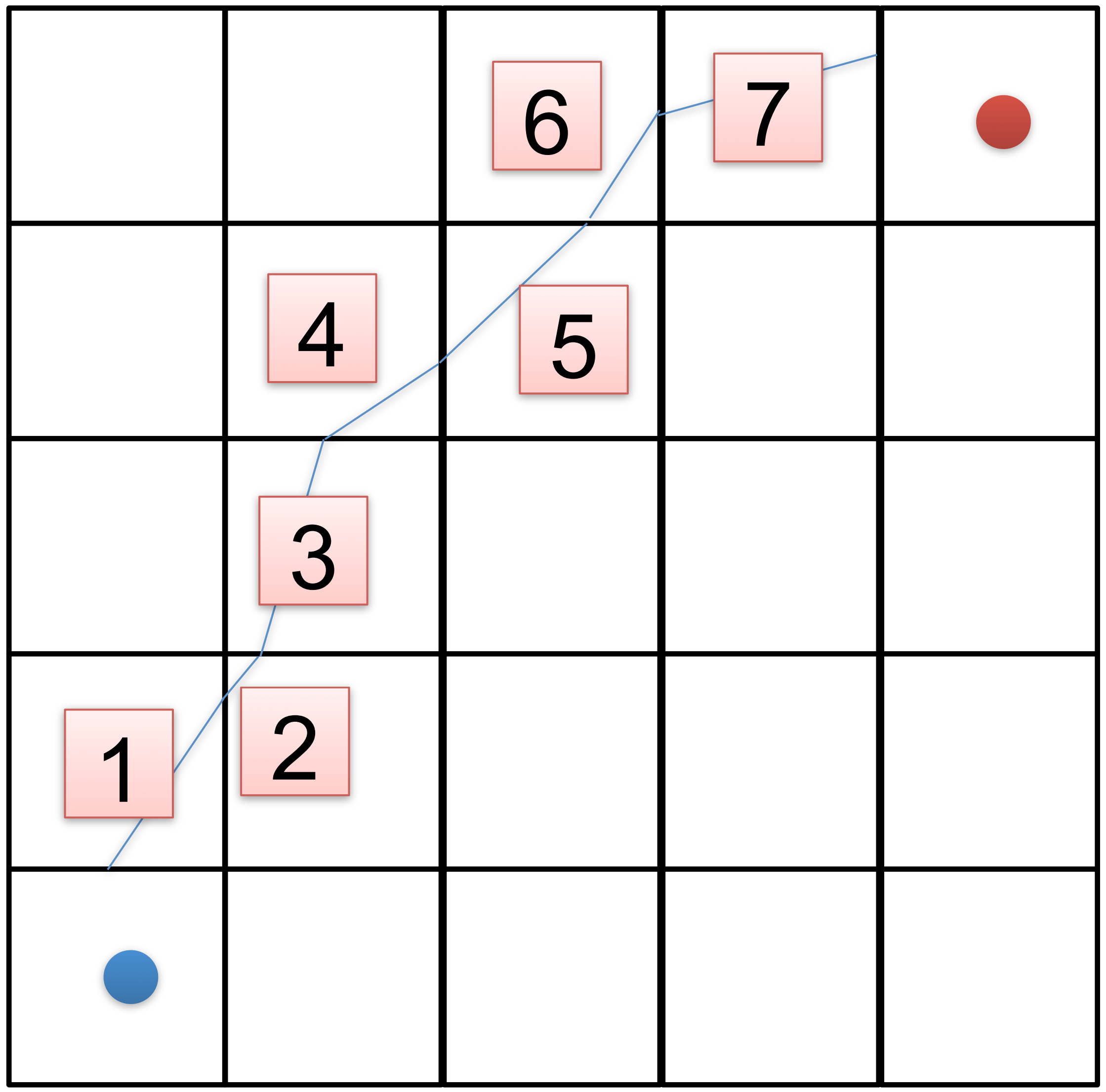
Segments



UPPSALA UNIVERSITET

Segment Mapping, illustration

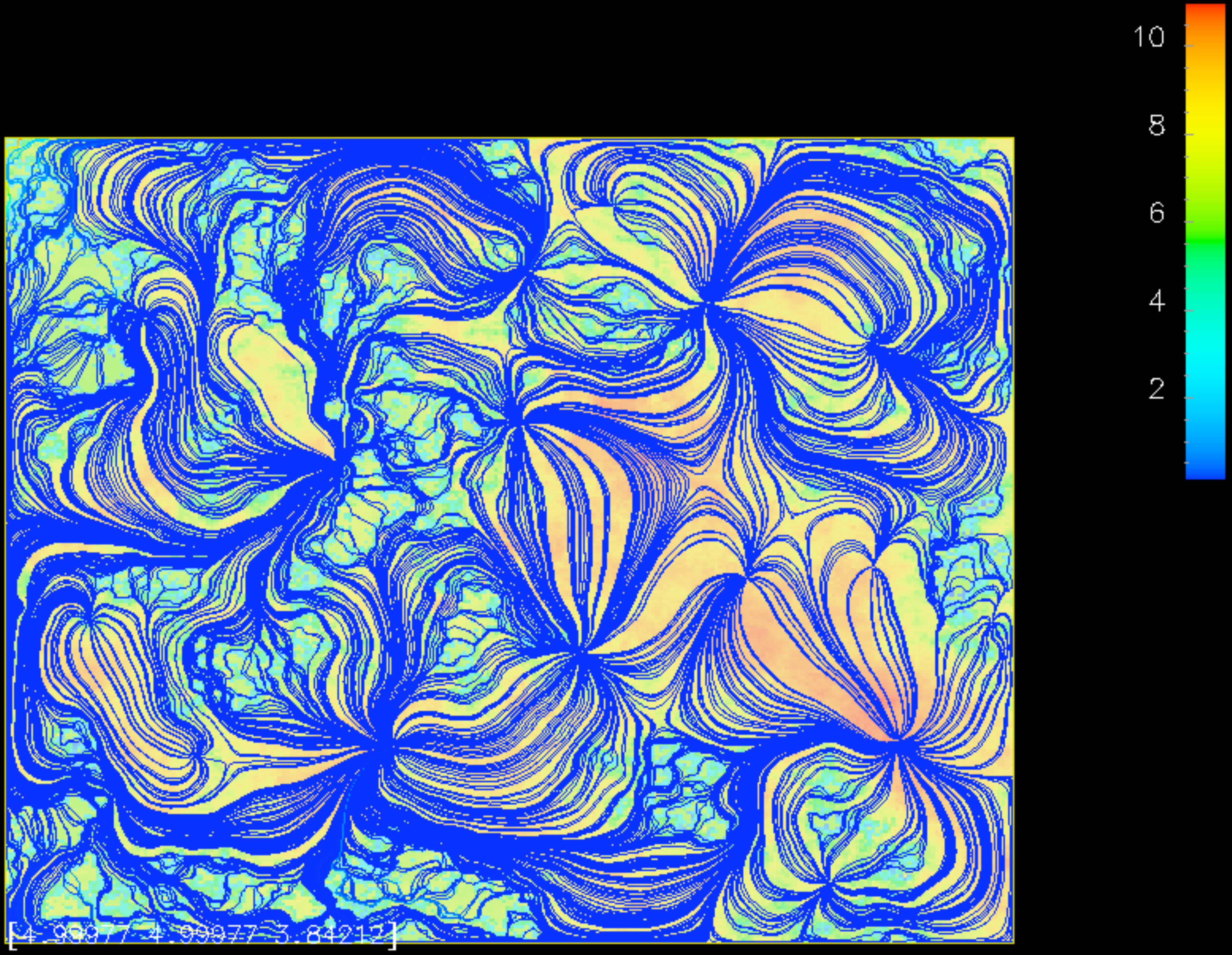
Segments



UPPSALA UNIVERSITET

Streamlines and Shared Memory

- To extract parallelism we trace, solve and map for several streamlines concurrently
  - You can extract fine-grained parallelism from a single streamline
- We call a set of streamlines a **streamline bundle** or simply **bundle**
- We seek a parallel algorithm that minimizes parallel overhead
  - Communication
  - Load imbalance
  - Synchronization


10
8
6
4
2
[4.99877 4.99877 3.84212]

UPPSALA UNIVERSITET

Parallel Algorithm

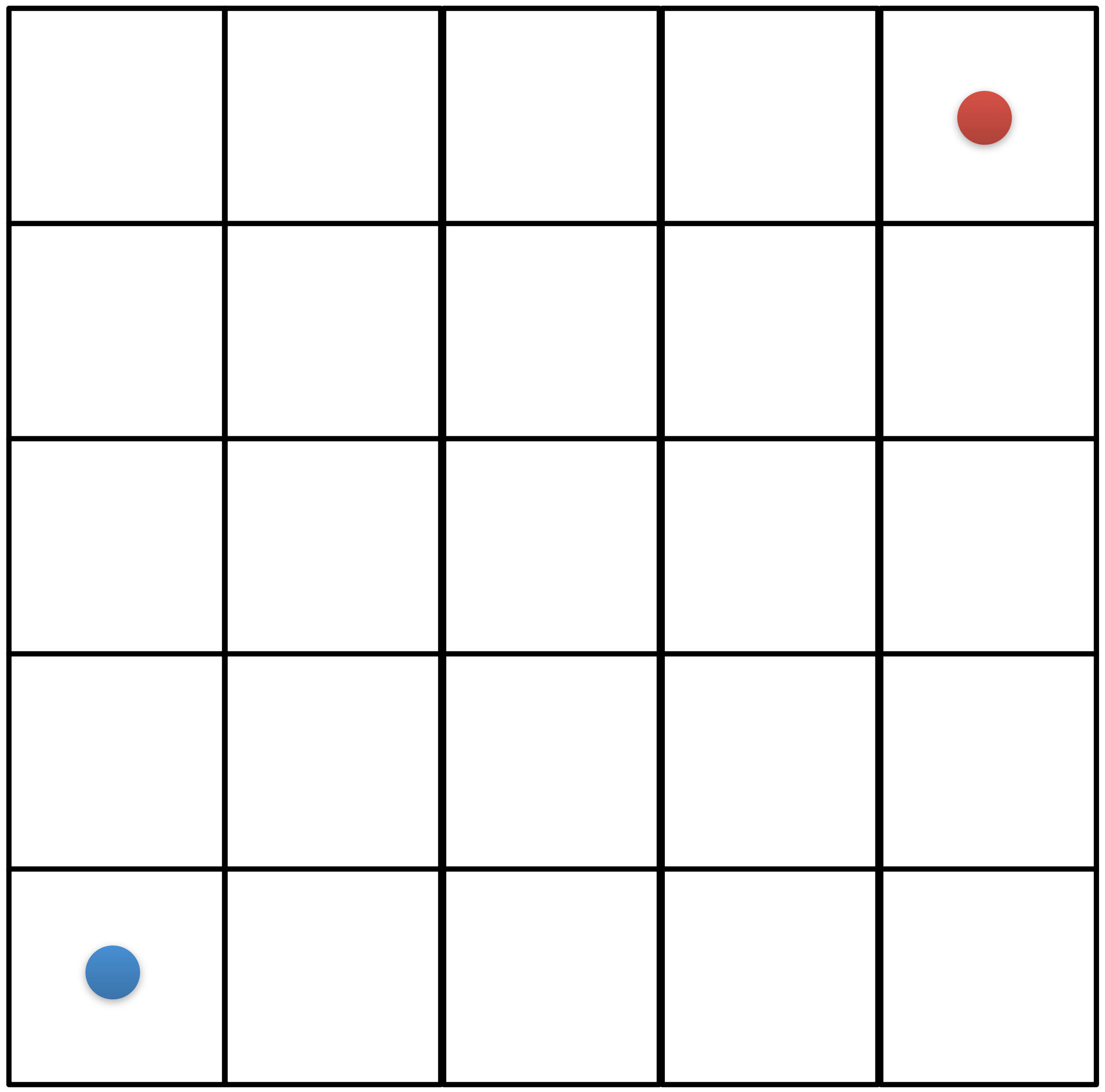
Algorithm 1 Parallel Streamline Simulation

```
Require: $T_{end} \leftarrow$ simulation end time
Require: $dt \leftarrow$ global timestep
Require: $T \leftarrow dt$
 1: repeat
 2:     Solve pressure equation
 3:     Calculate velocity field
 4:     repeat
 5:         Select launch points for streamlines and build bundle
 6:         Assign launch points in bundle to threads
 7:         for all streamlines in bundle do
 8:             1:st trace, count segments
 9:             2:nd trace, pick up pressure grid data, store the segments
10:             Build streamline grids from segment and pressure grid data
11:             Solve the corresponding 1-D transport equations
12:         end for
13:         Map new values of the transport variables to the pressure grid
14:     until Domain is sufficiently covered
15:     $T \leftarrow T + dt$
16: until $T \geq T_{end}$
```

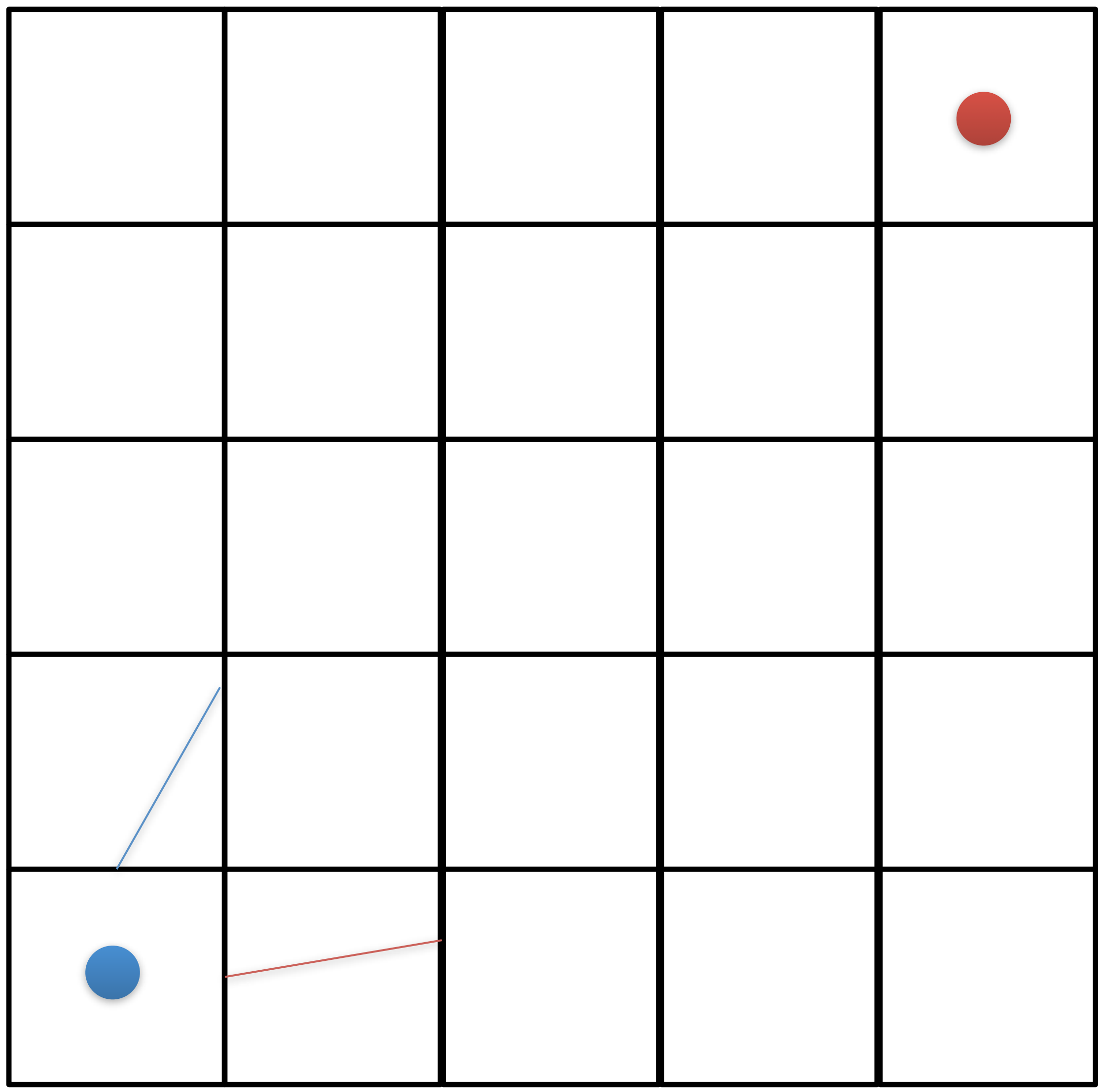
UPPSALA UNIVERSITET

Static Assignment

Thread 1 Thread 2



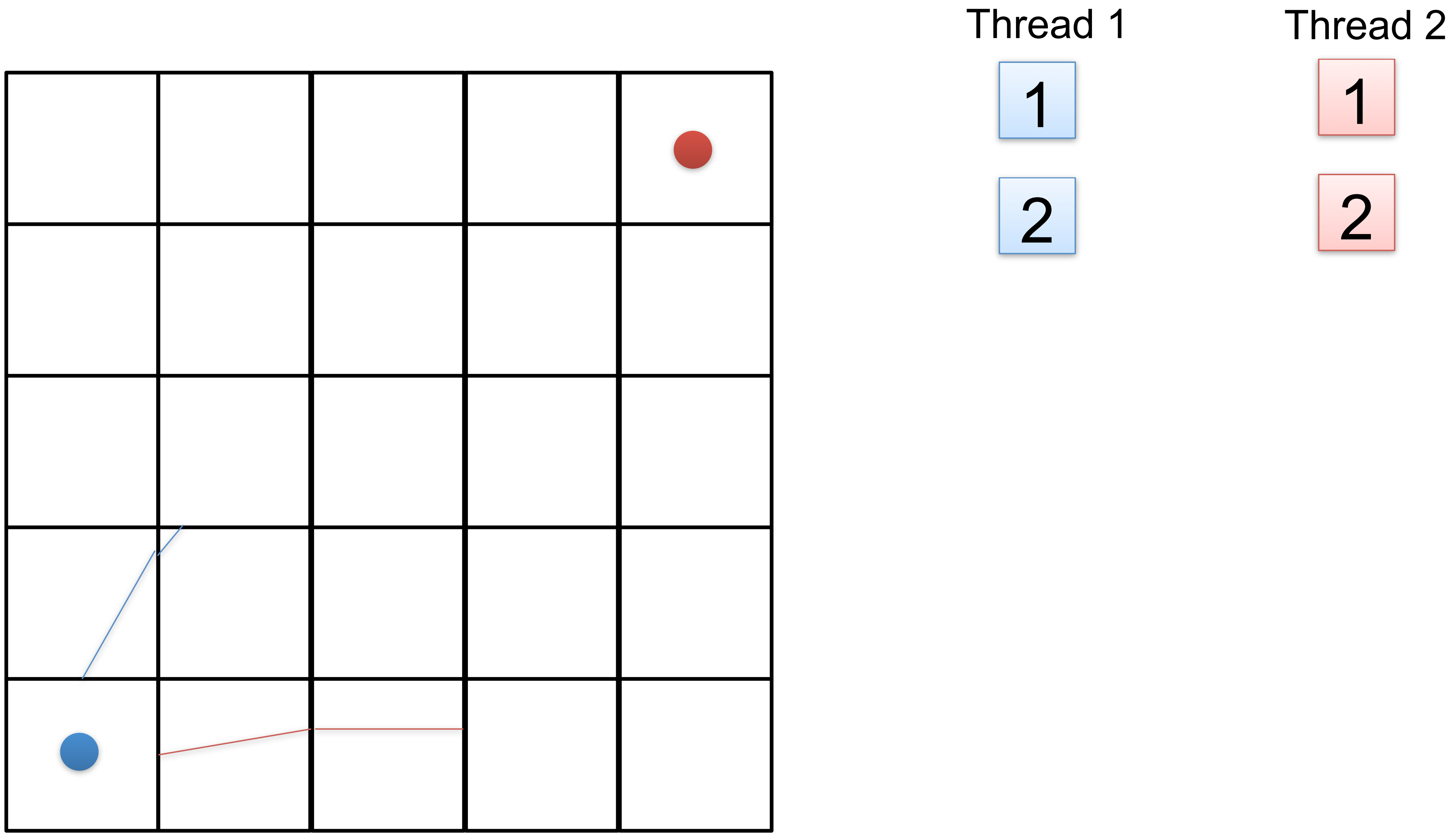
Static Assignment



Thread 1
1
Thread 2
1

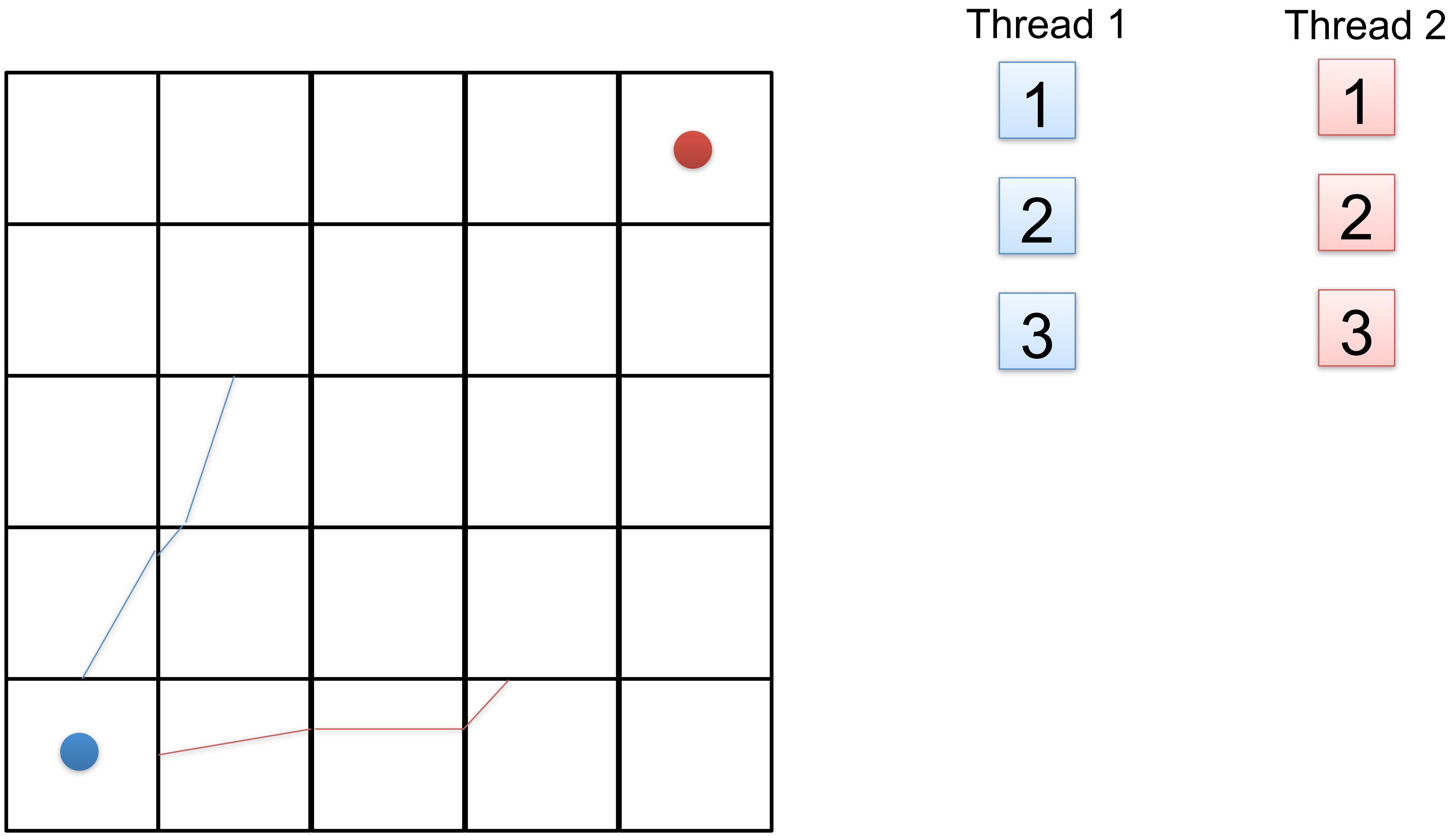
UPPSALA UNIVERSITET

Static Assignment



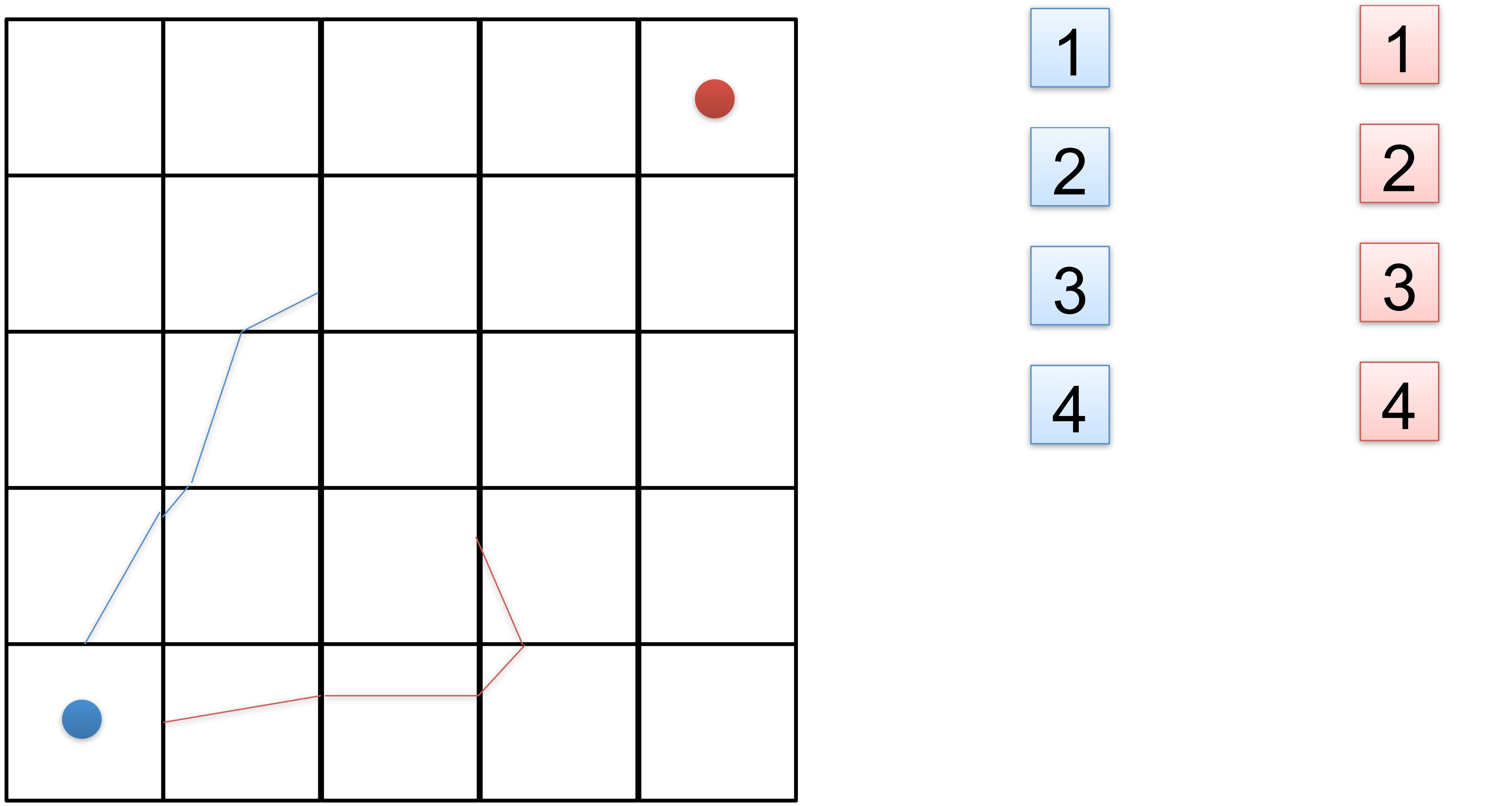
UPPSALA UNIVERSITET

Static Assignment



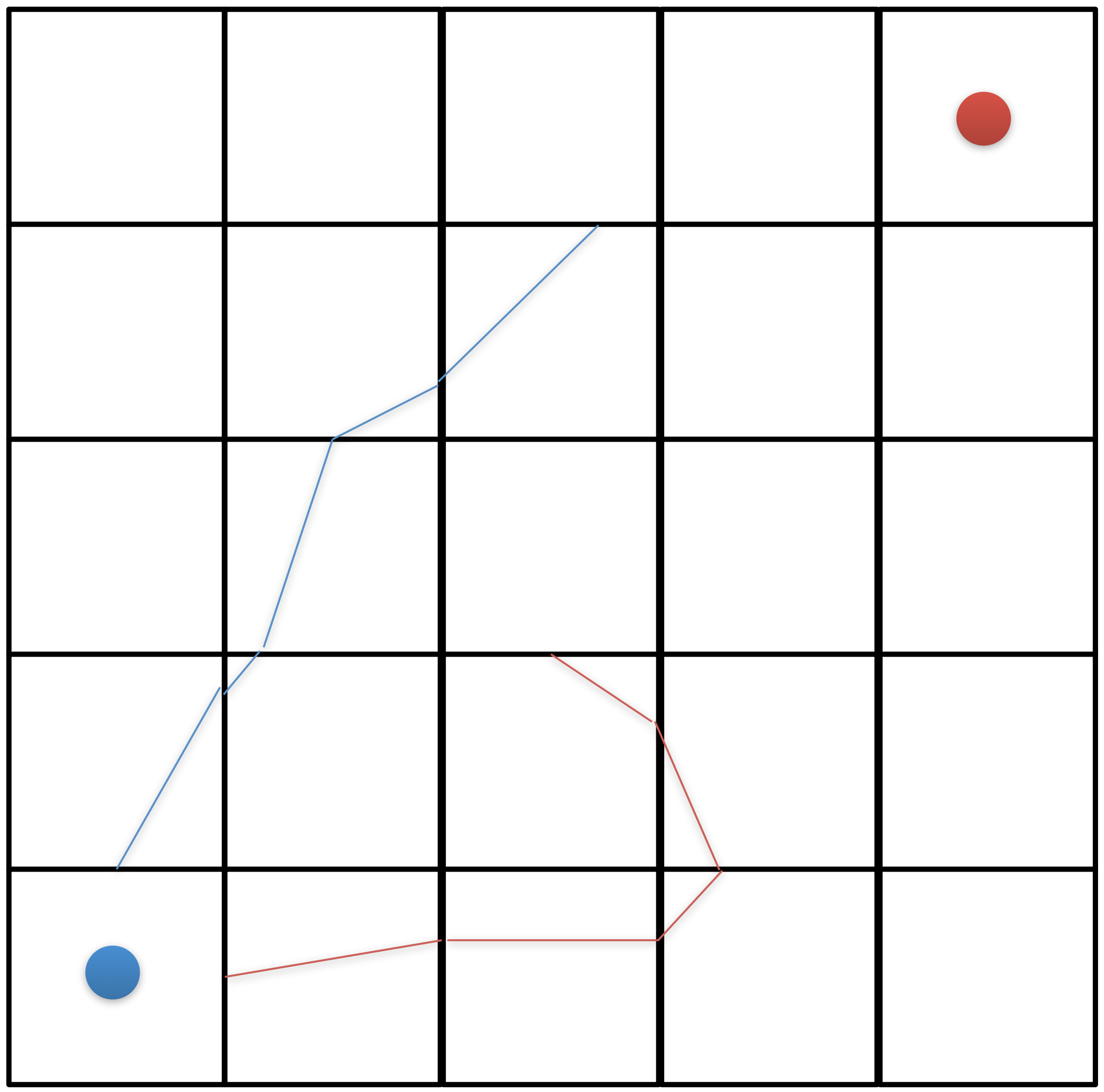
UPPSALA UNIVERSITET

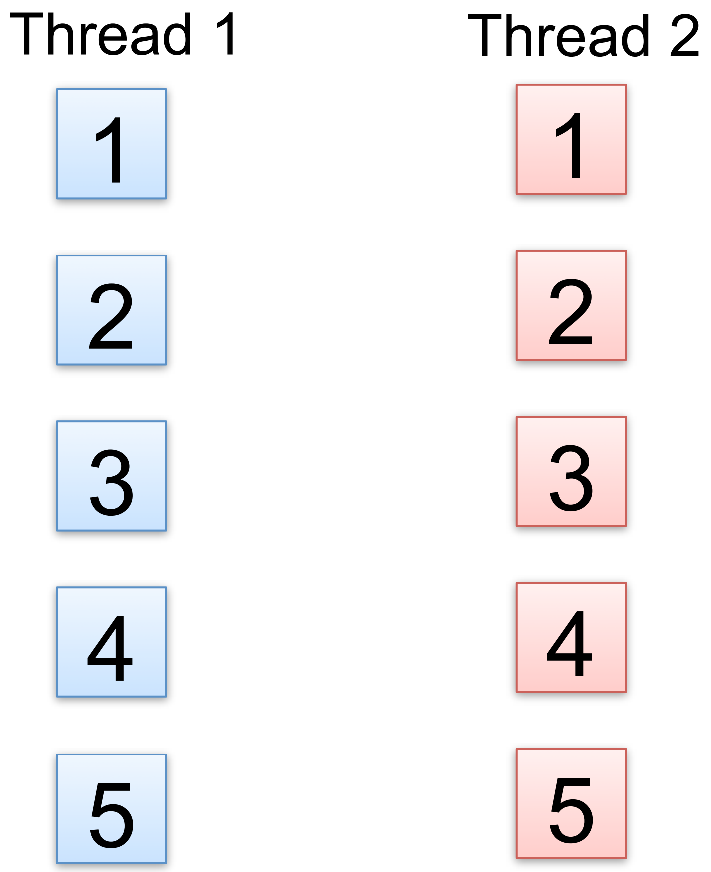
Static Assignment



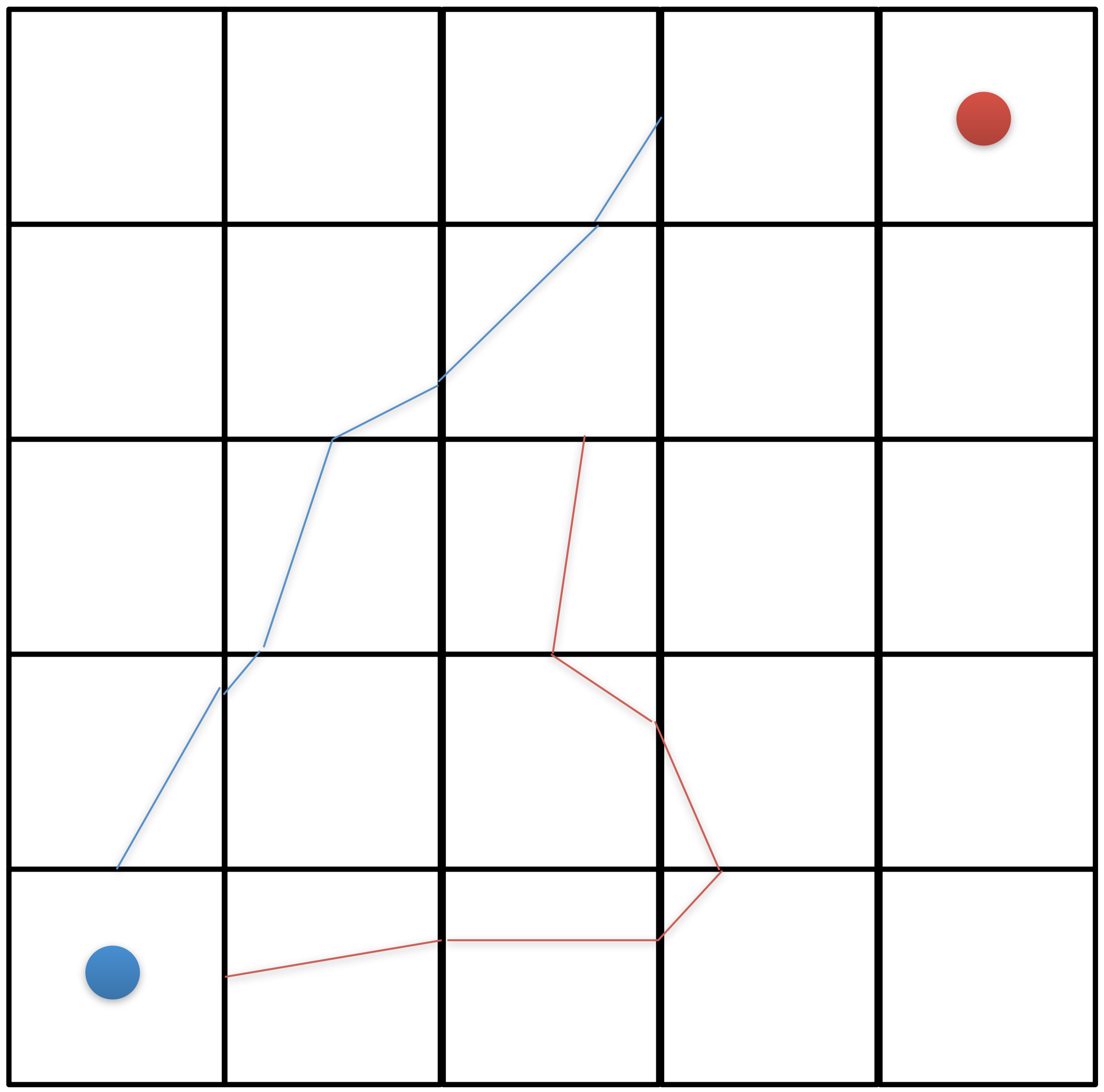
UPPSALA UNIVERSITET

Static Assignment





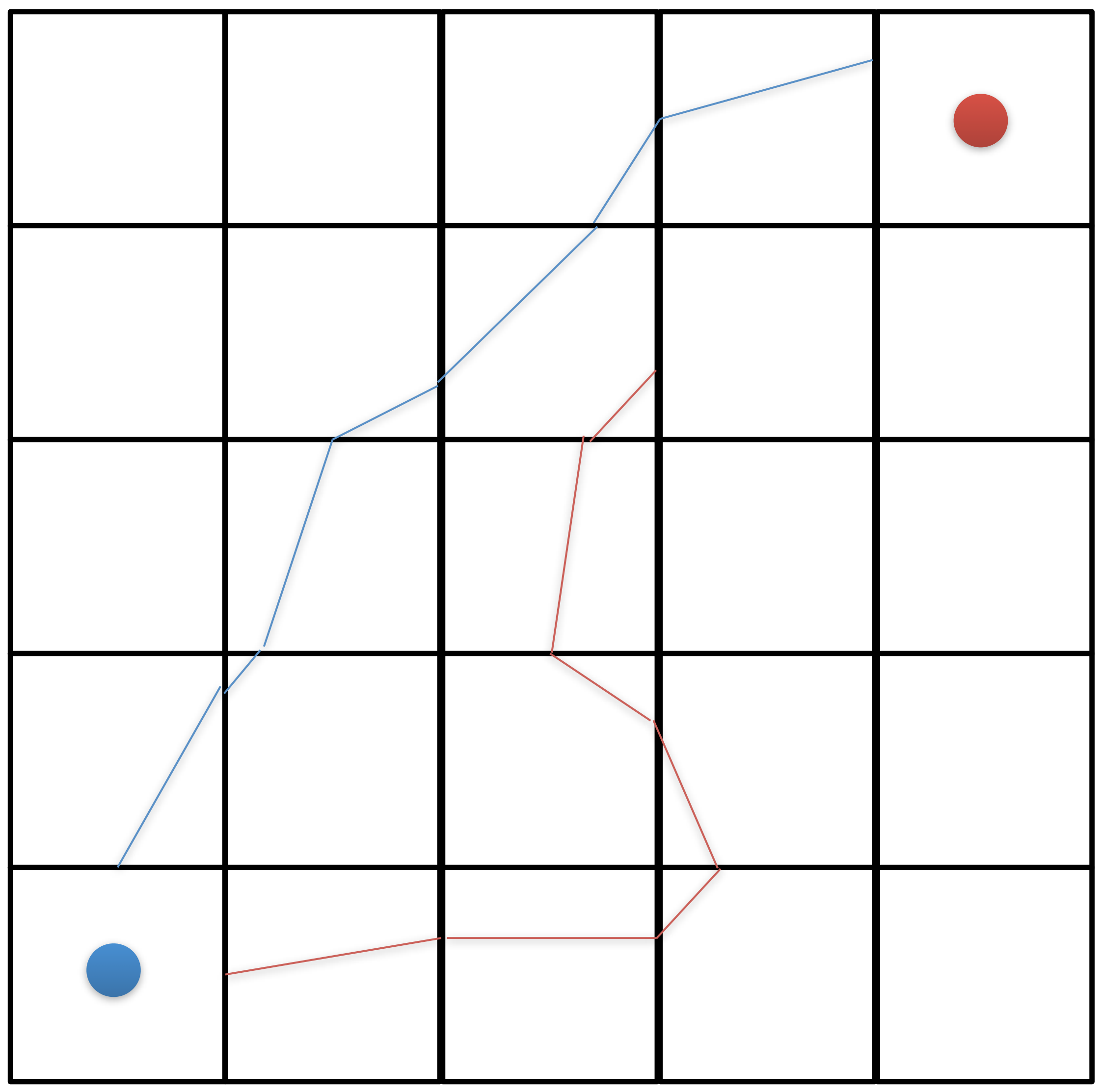
Static Assignment

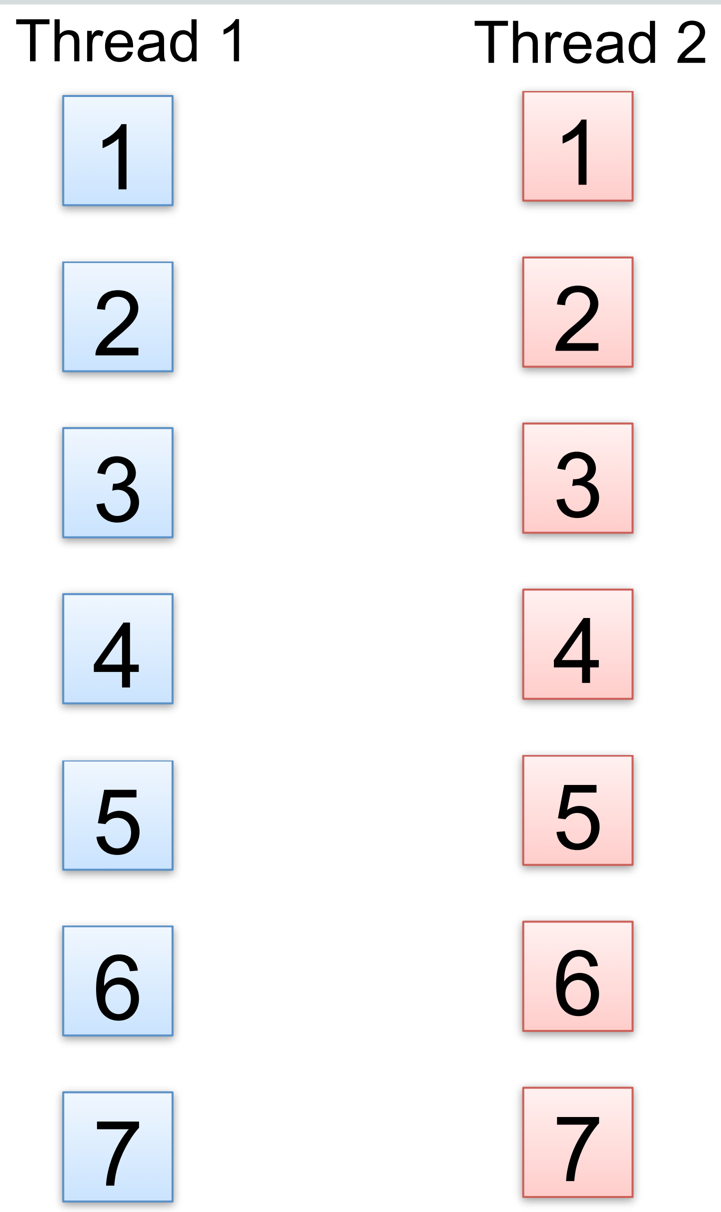


Thread 1	Thread 2
1	1
2	2
3	3
4	4
5	5
6	6

UPPSALA UNIVERSITET

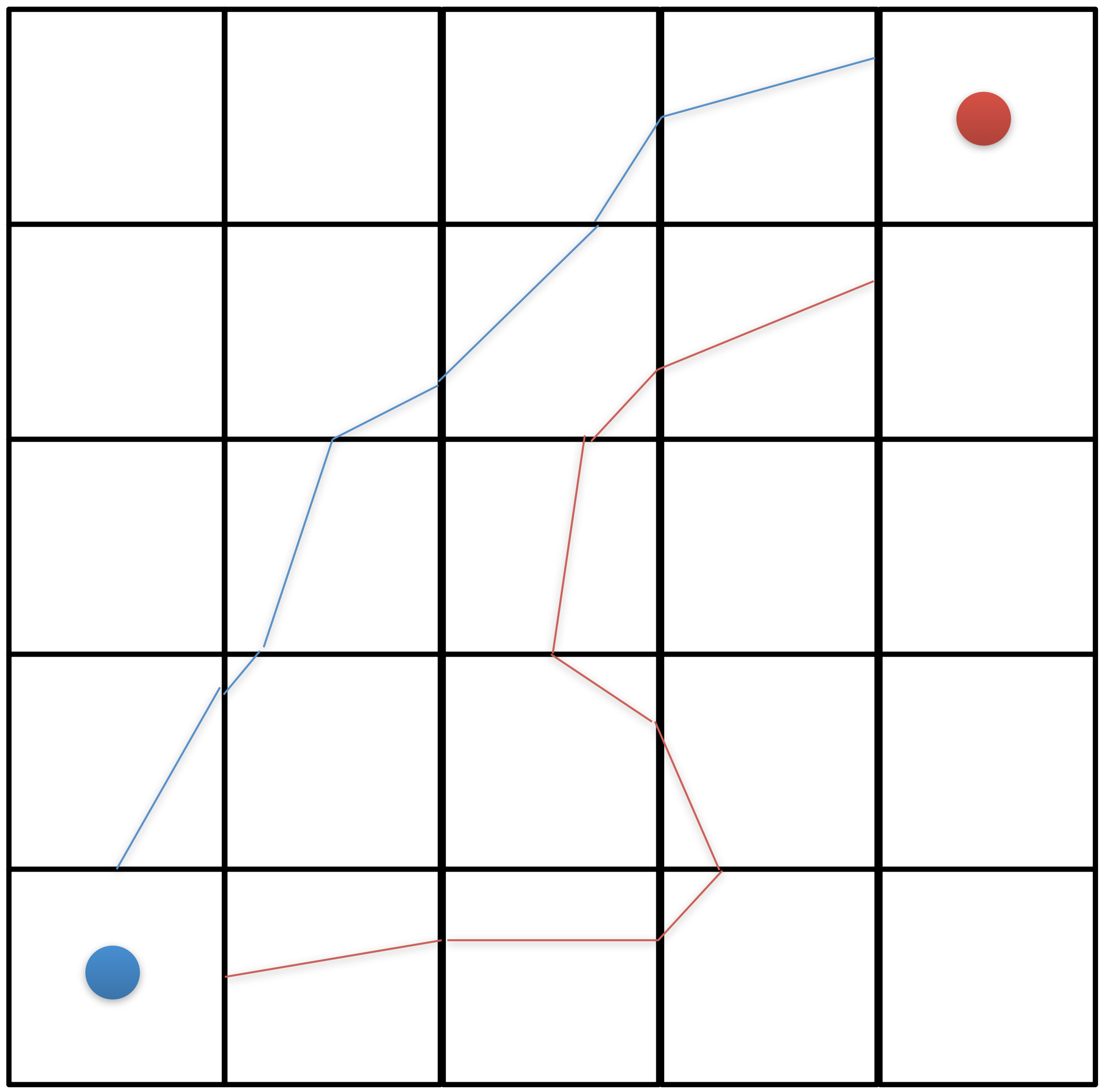
Static Assignment

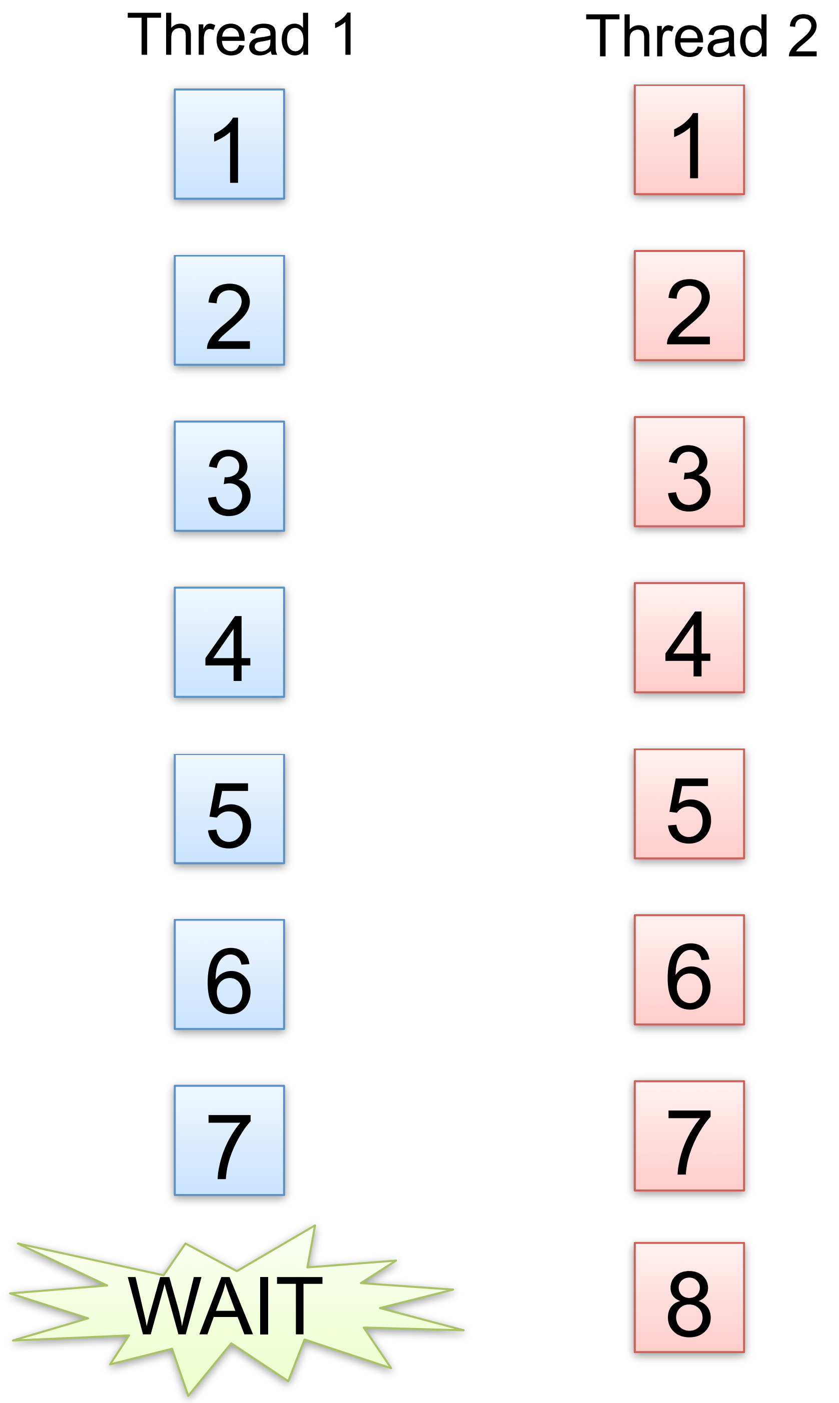




UPPSALA UNIVERSITET

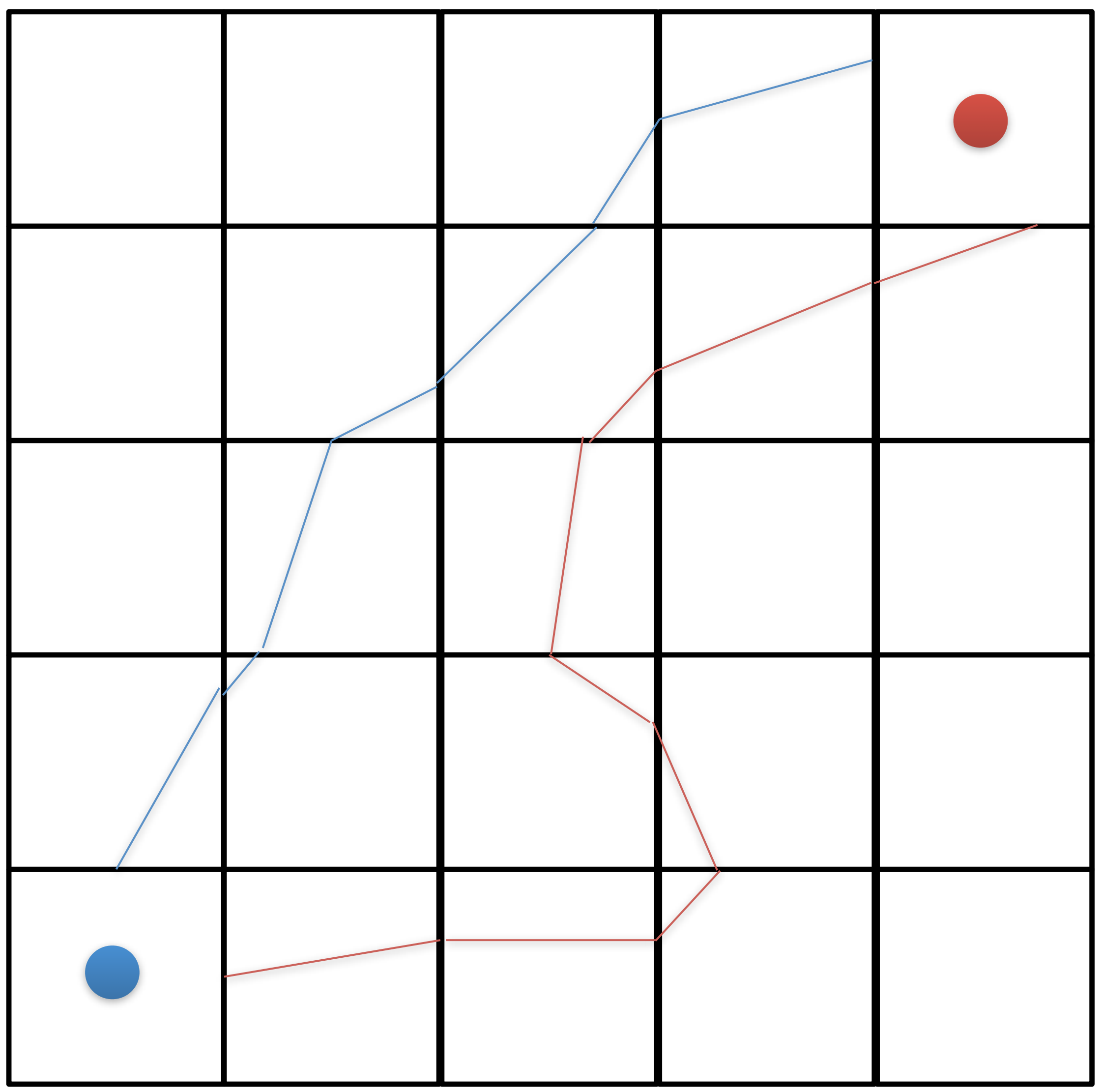
Static Assignment





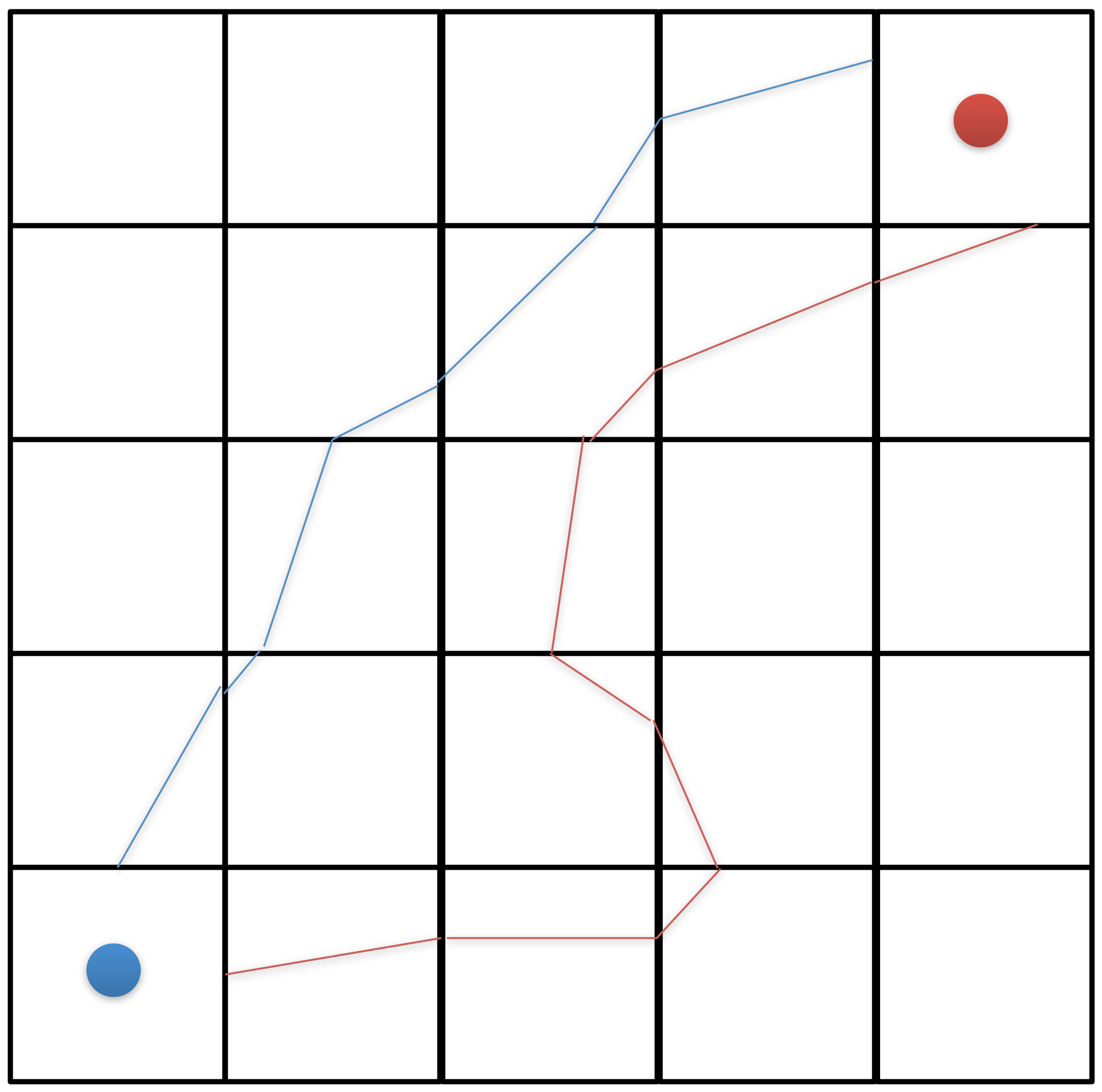
UPPSALA UNIVERSITET

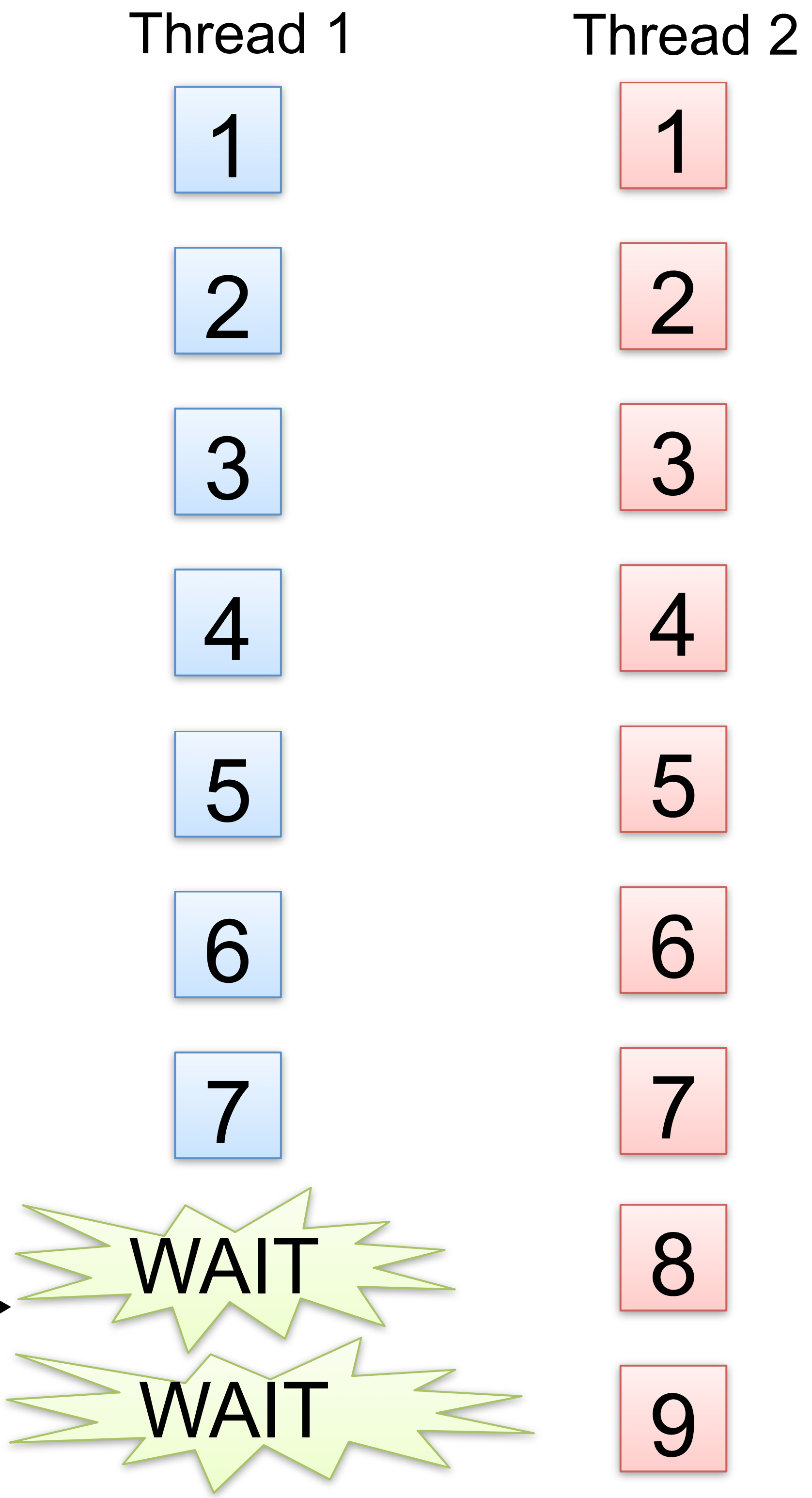
Static Assignment



Thread 1	Thread 2
1	1
2	2
3	3
4	4
5	5
6	6
7	7
WAIT	8
WAIT	9

Static Assignment



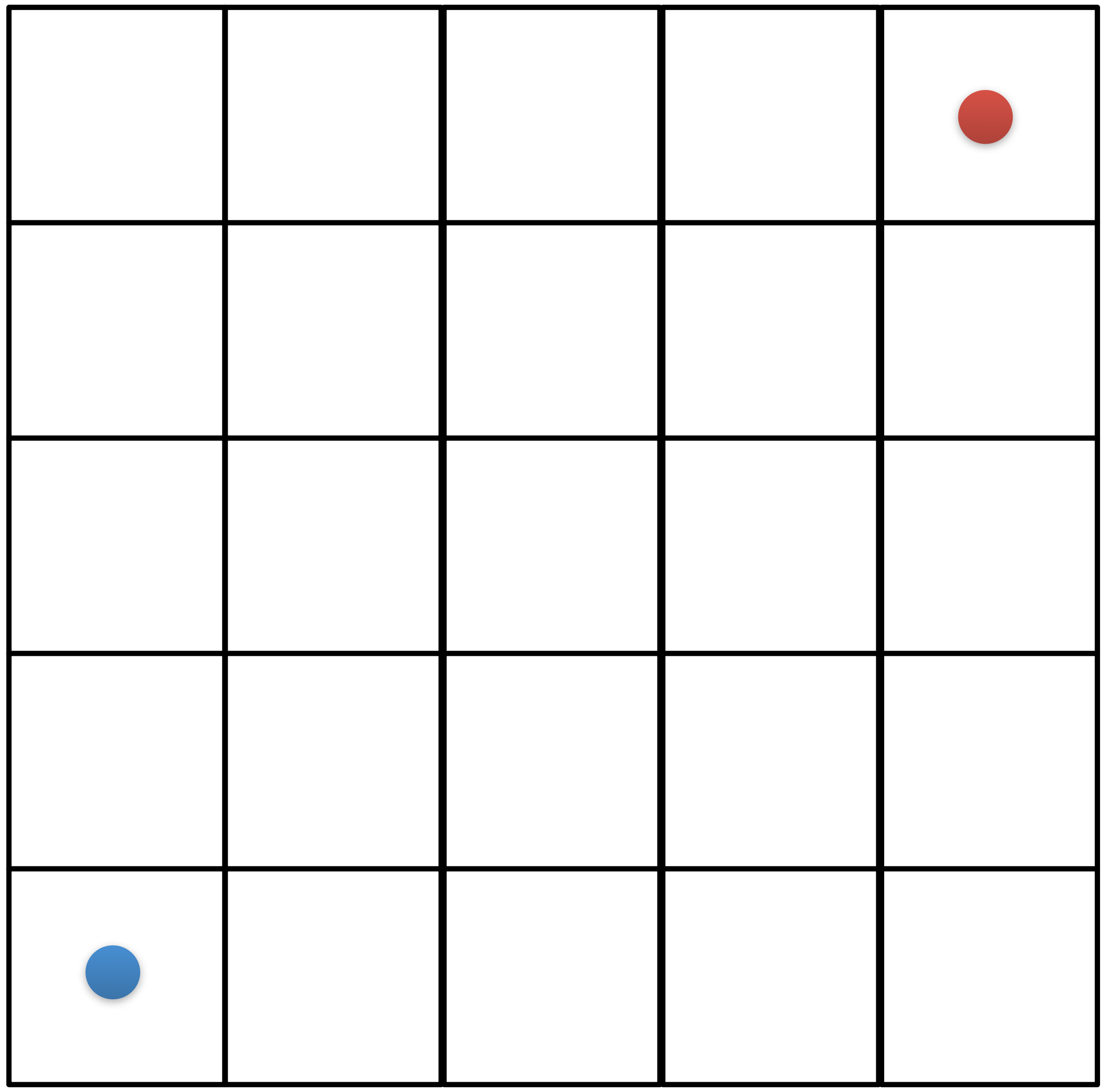


Waiting will limit scalablility

UPPSALA UNIVERSITET

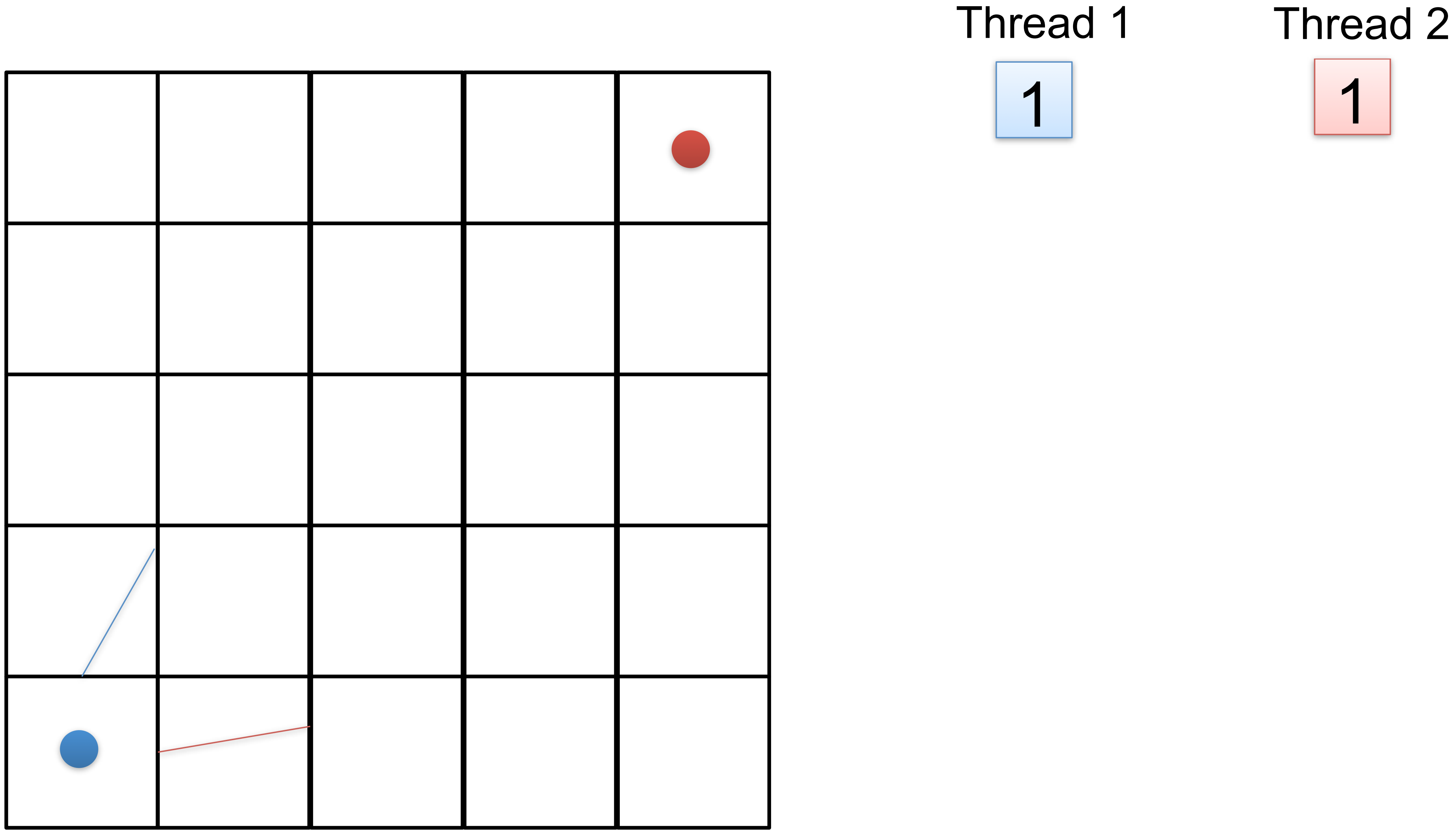
Dynamic Assignment

Thread 1 Thread 2



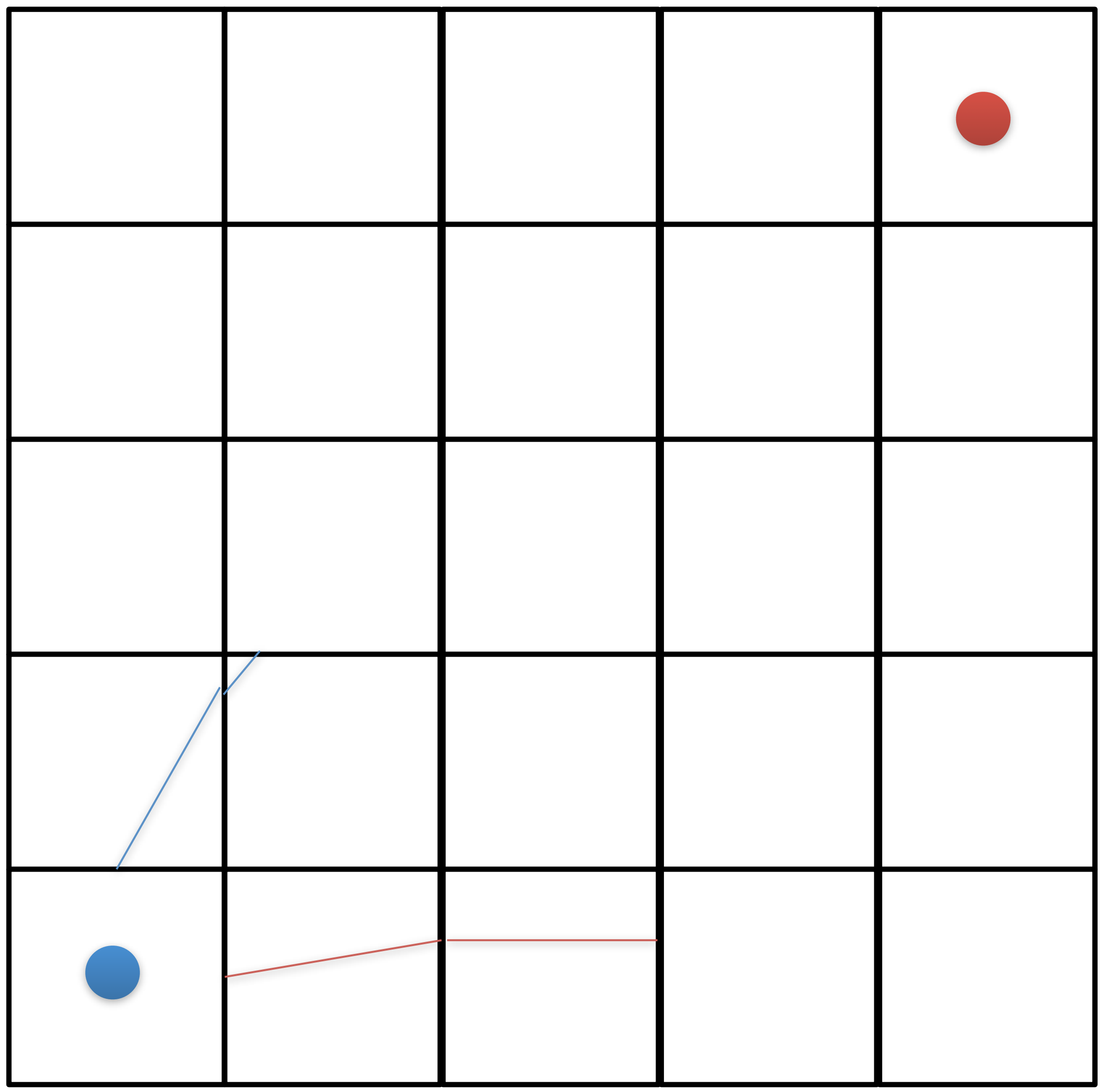
UPPSALA UNIVERSITET

Dynamic Assignment



UPPSALA UNIVERSITET

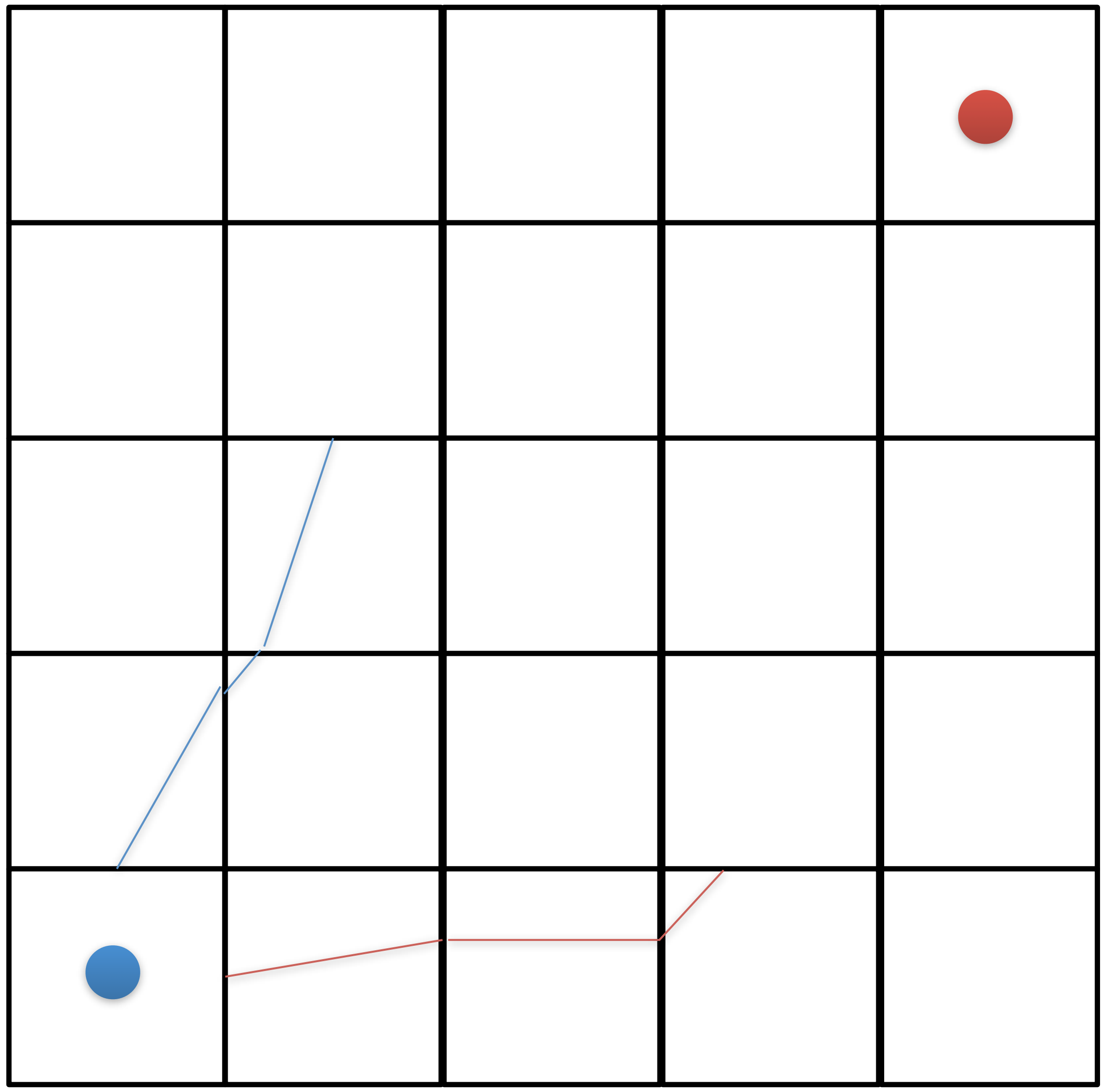
Dynamic Assignment

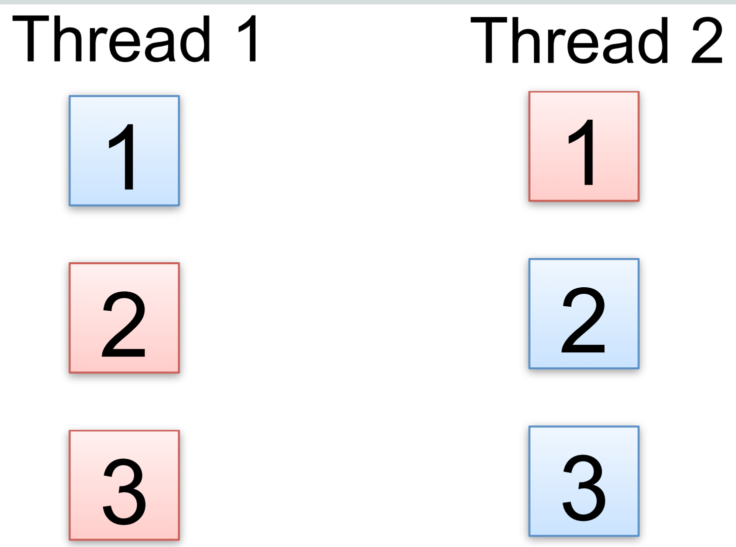


Thread 1	Thread 2
1	1
2	2

UPPSALA UNIVERSITET

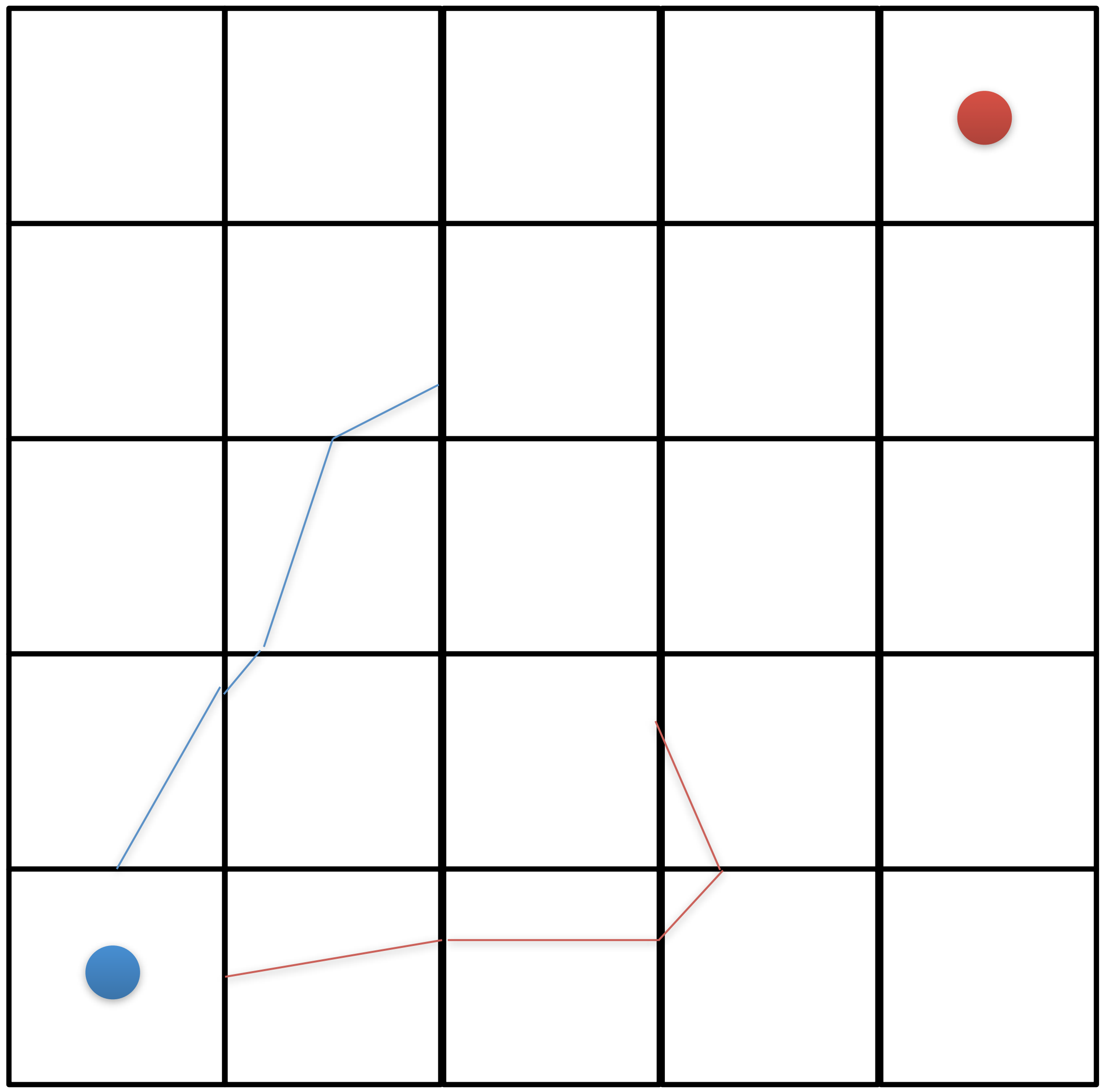
Dynamic Assignment

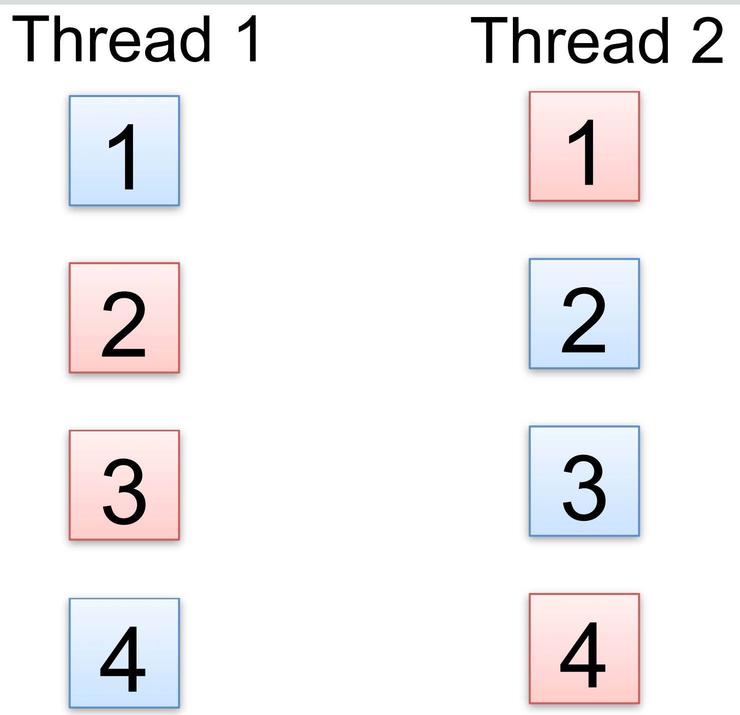




UPPSALA UNIVERSITET

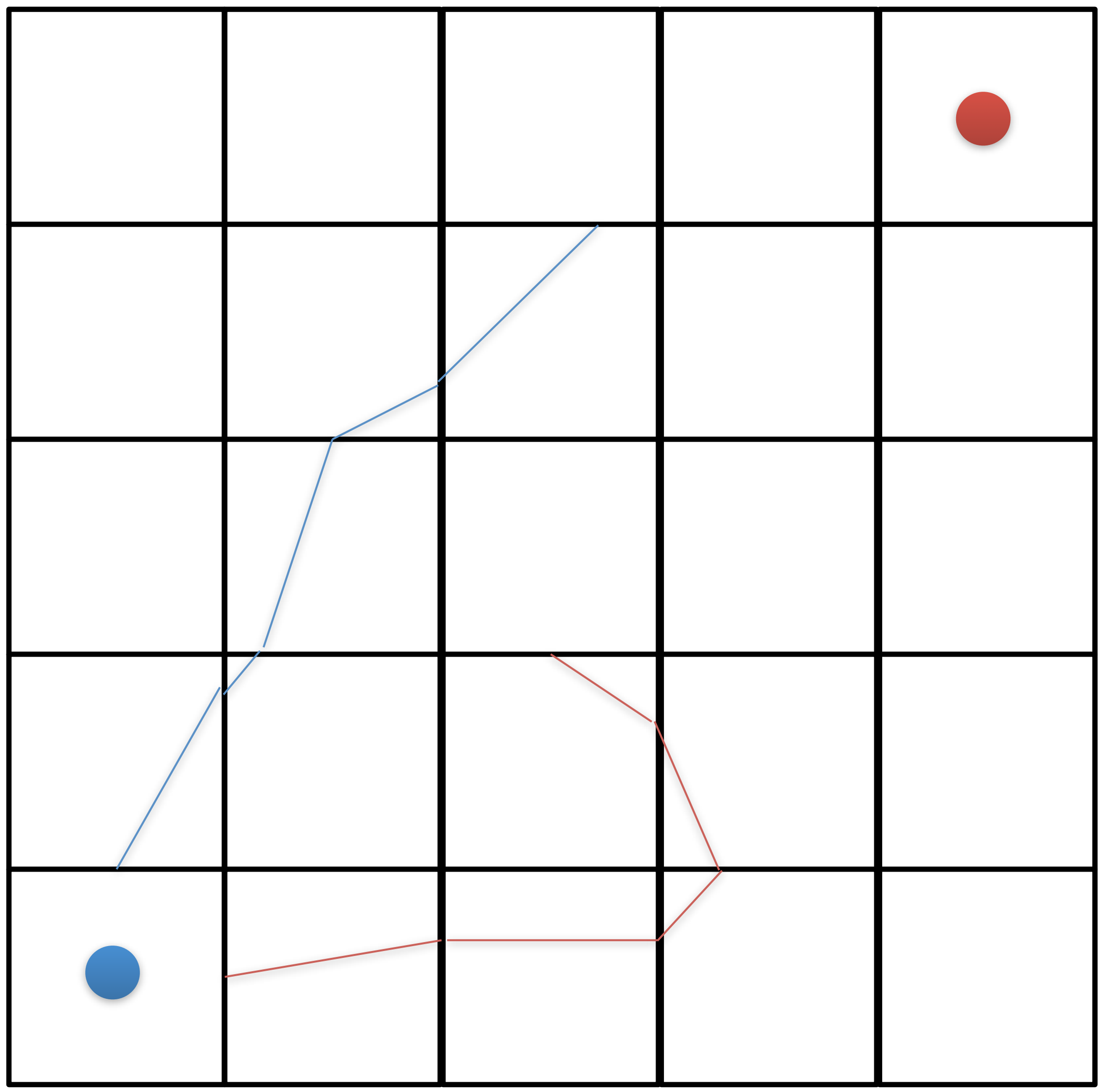
Dynamic Assignment





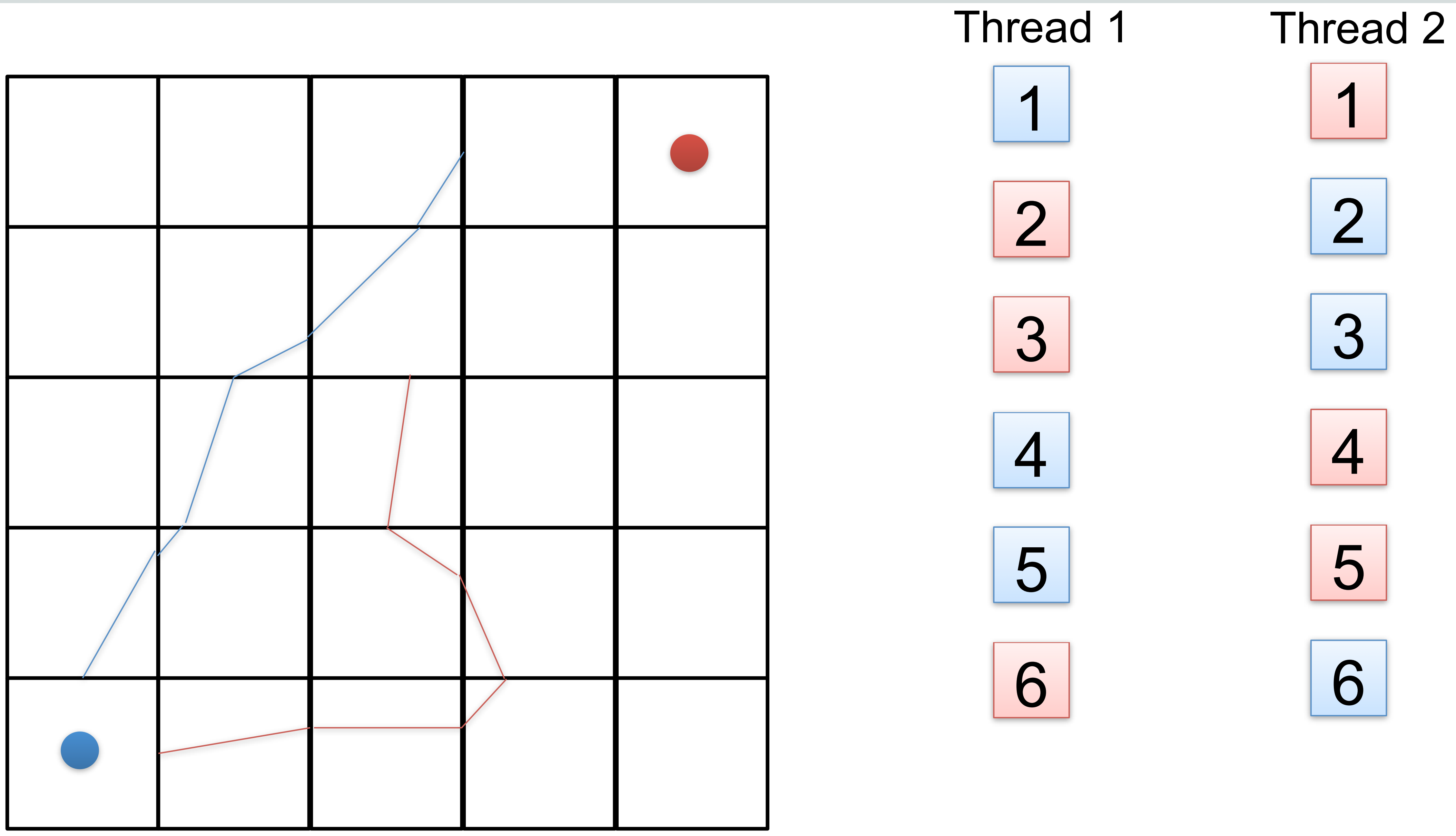
UPPSALA UNIVERSITET

Dynamic Assignment

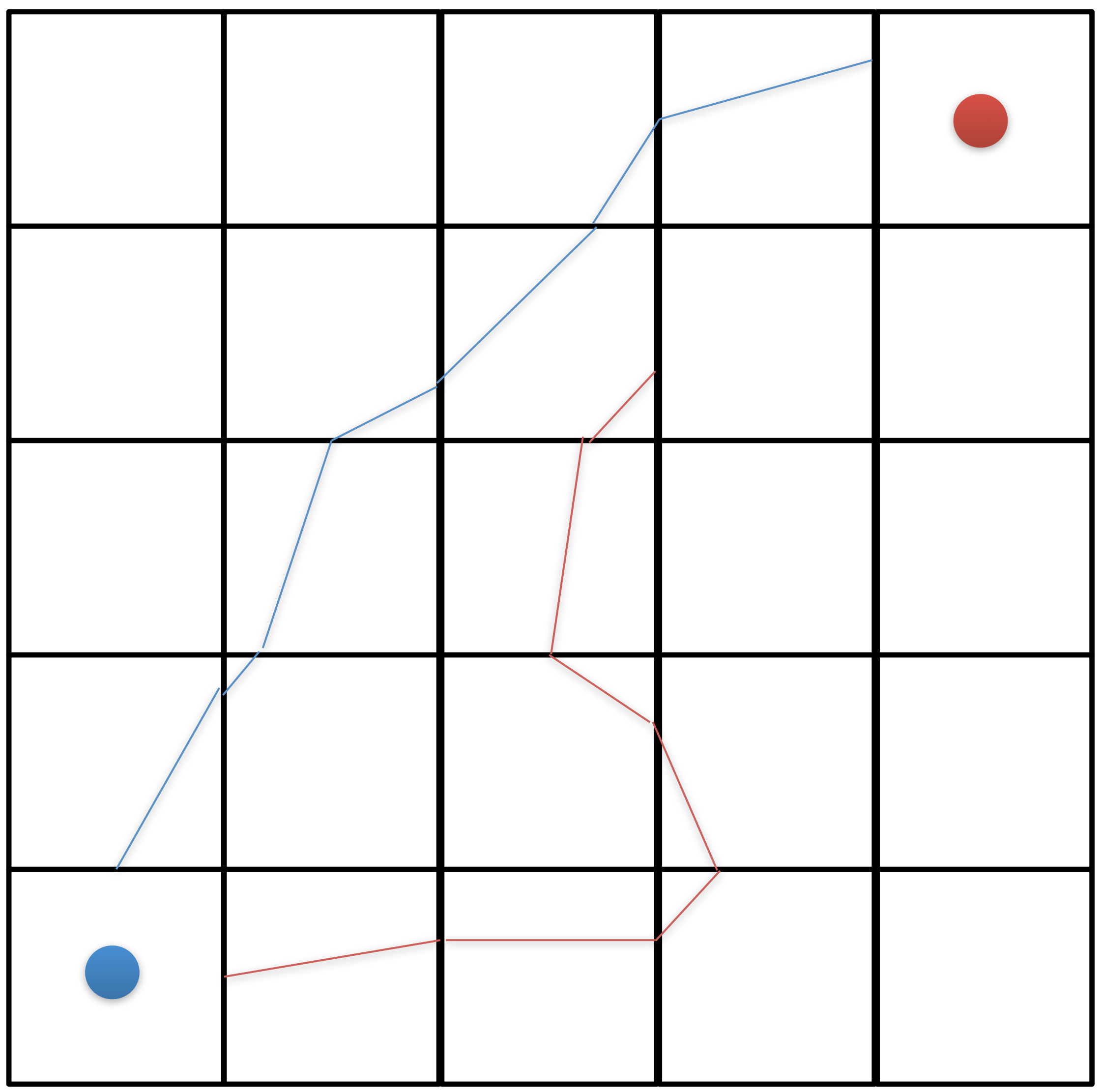


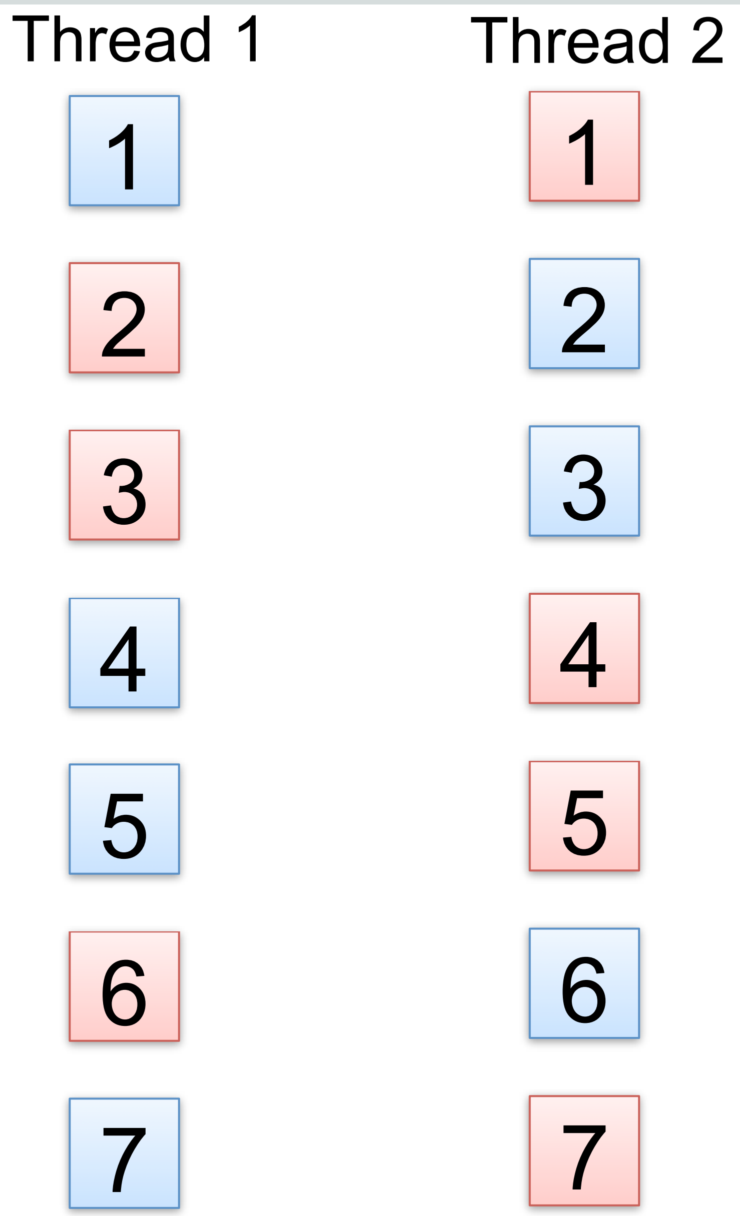
Thread 1	Thread 2
🟦 1	🟥 1
🟥 2	🟦 2
🟥 3	🟦 3
🟦 4	🟥 4
🟦 5	🟥 5

Dynamic Assignment

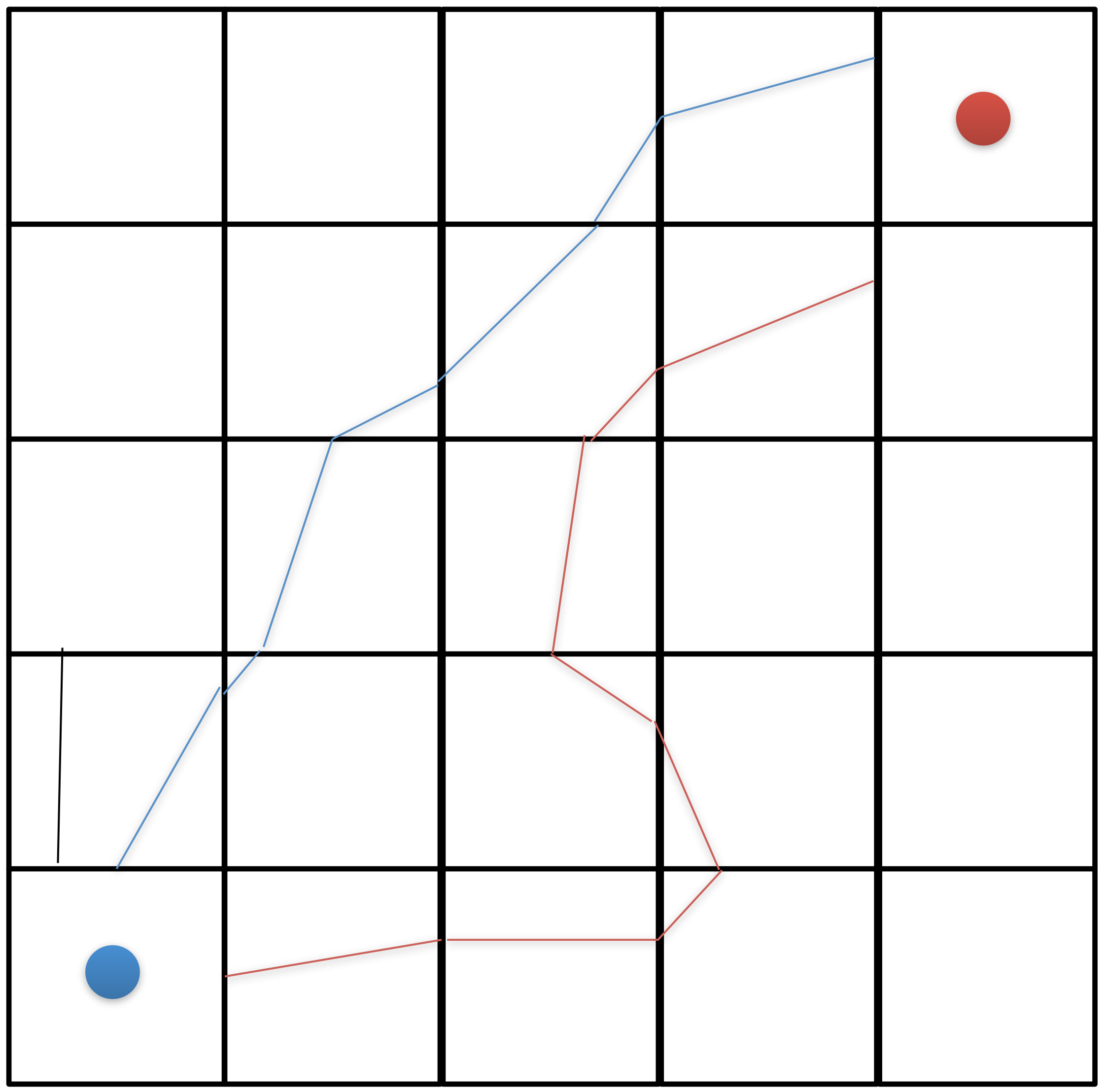


Dynamic Assignment



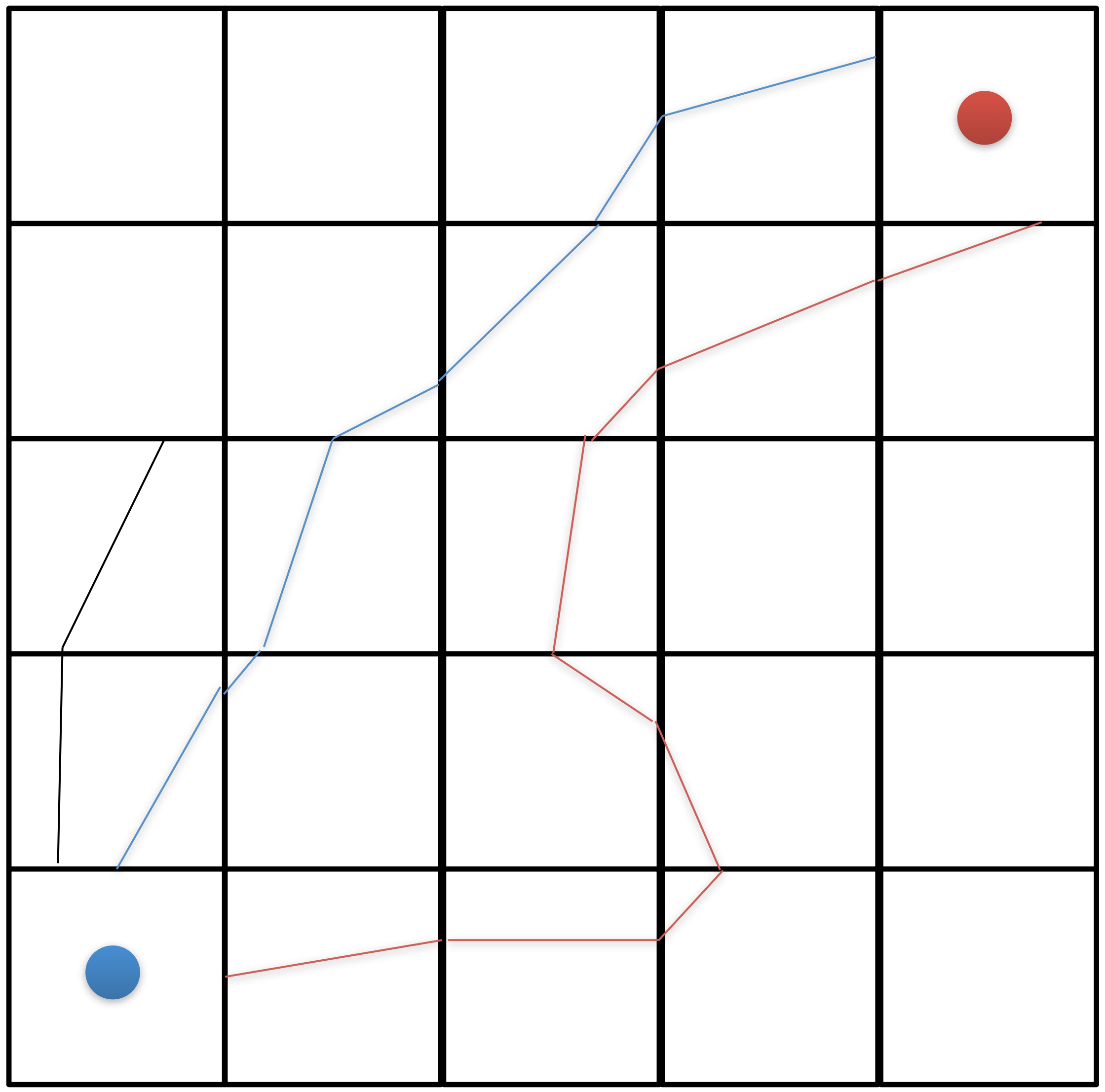


Dynamic Assignment



Thread 1	Thread 2
1	1
2	2
3	3
4	4
5	5
6	6
7	7
1	8

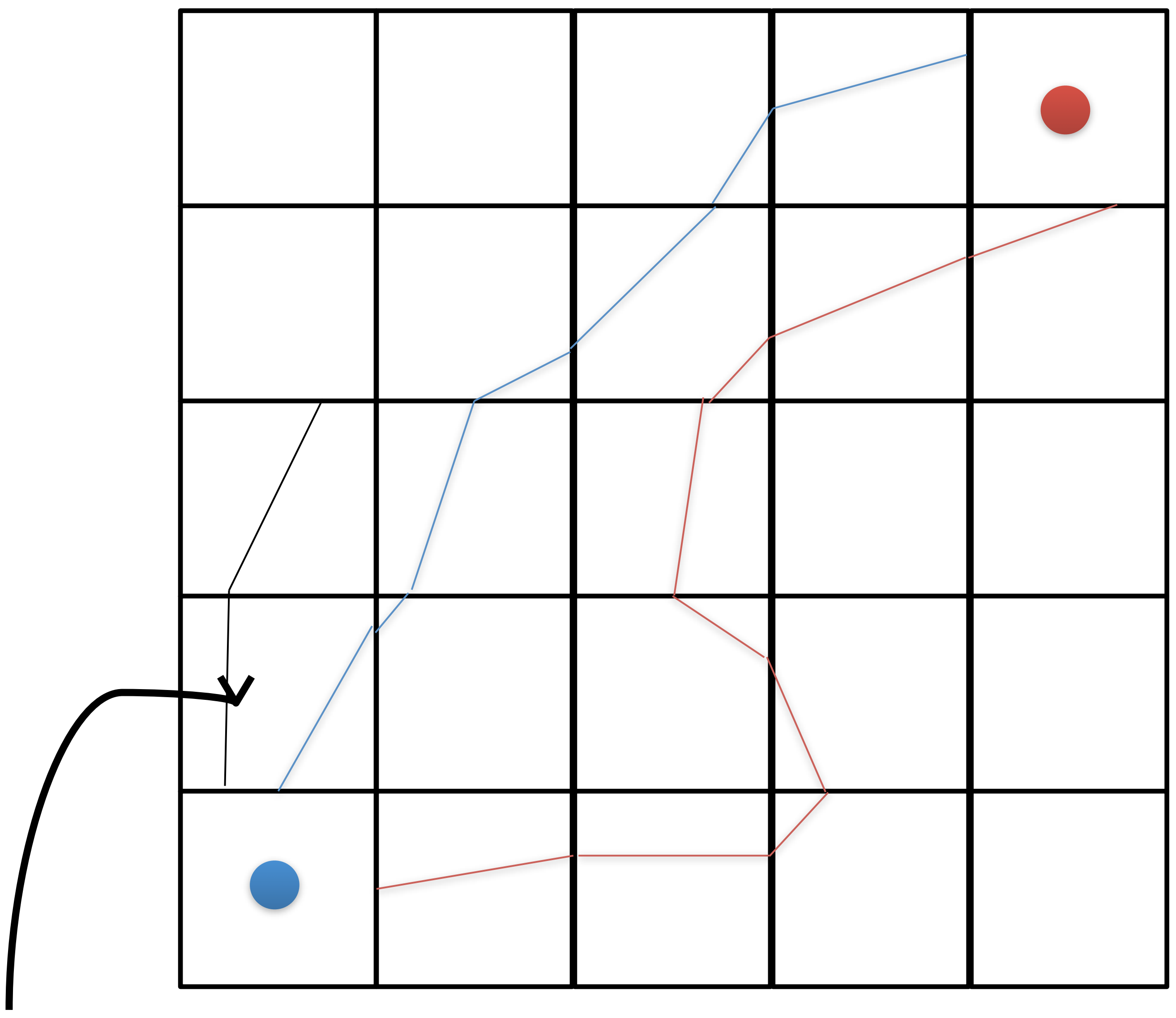
Dynamic Assignment



Thread 1	Thread 2
1	1
2	2
3	3
4	4
5	5
6	6
7	7
1	8
2	9

UPPSALA UNIVERSITET

Dynamic Assignment

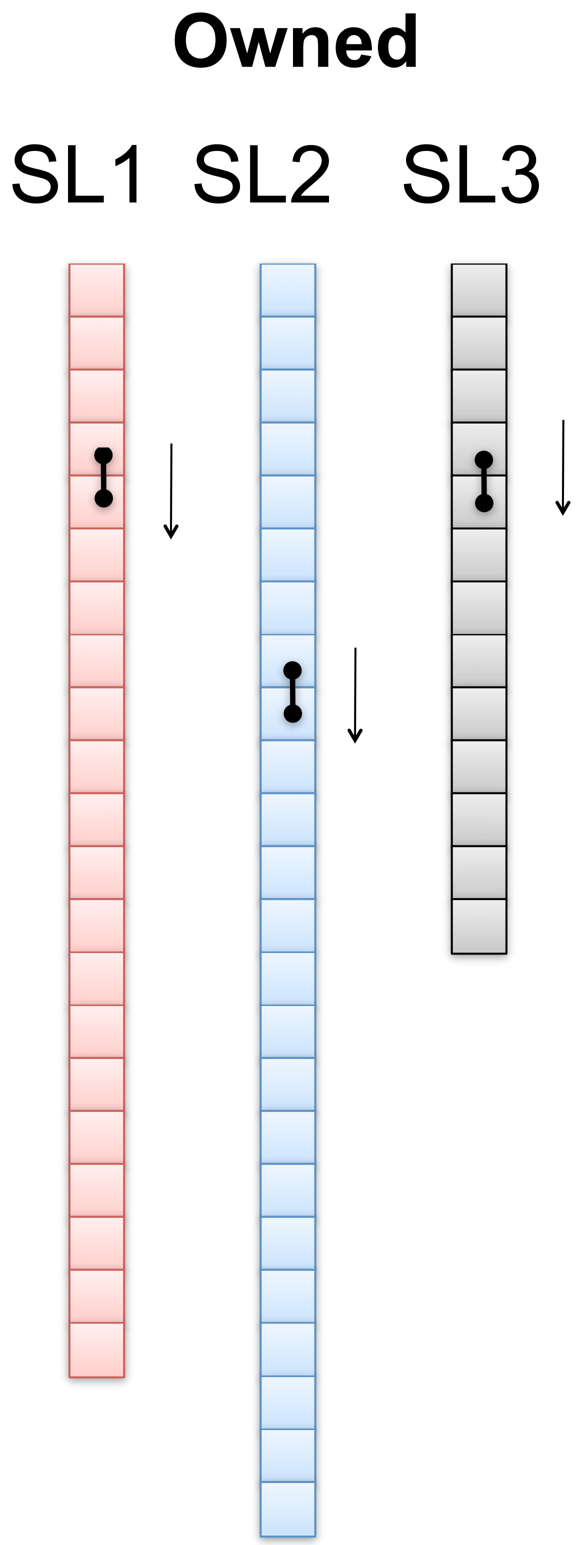


Thread 1	Thread 2
1	1
2	2
3	3
4	4
5	5
6	6
7	7
1	8
2	9

No waiting. We can start on the next SL directly

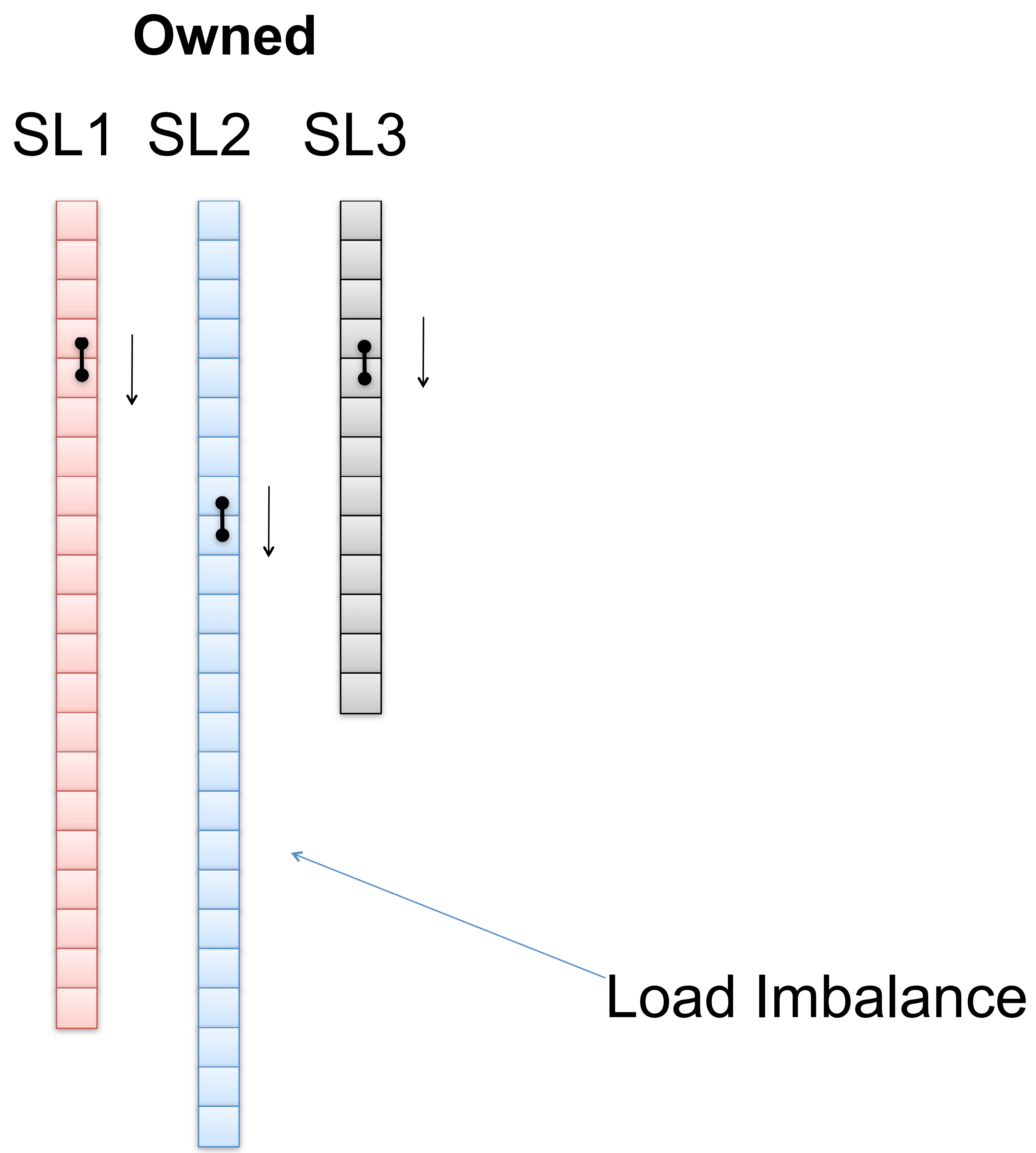
UPPSALA UNIVERSITET

Parallel 1-D Solves, illustration

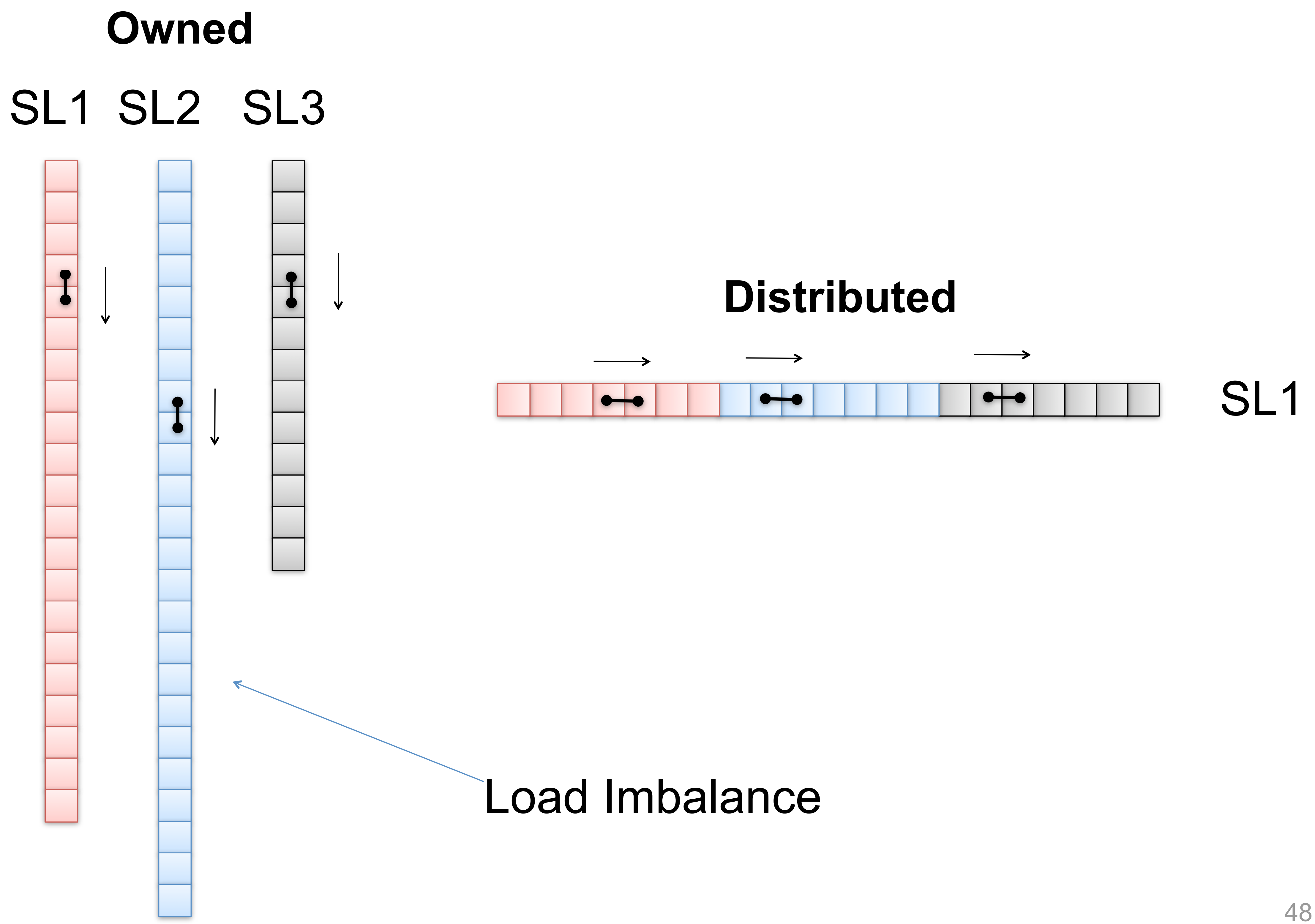


UPPSALA UNIVERSITET

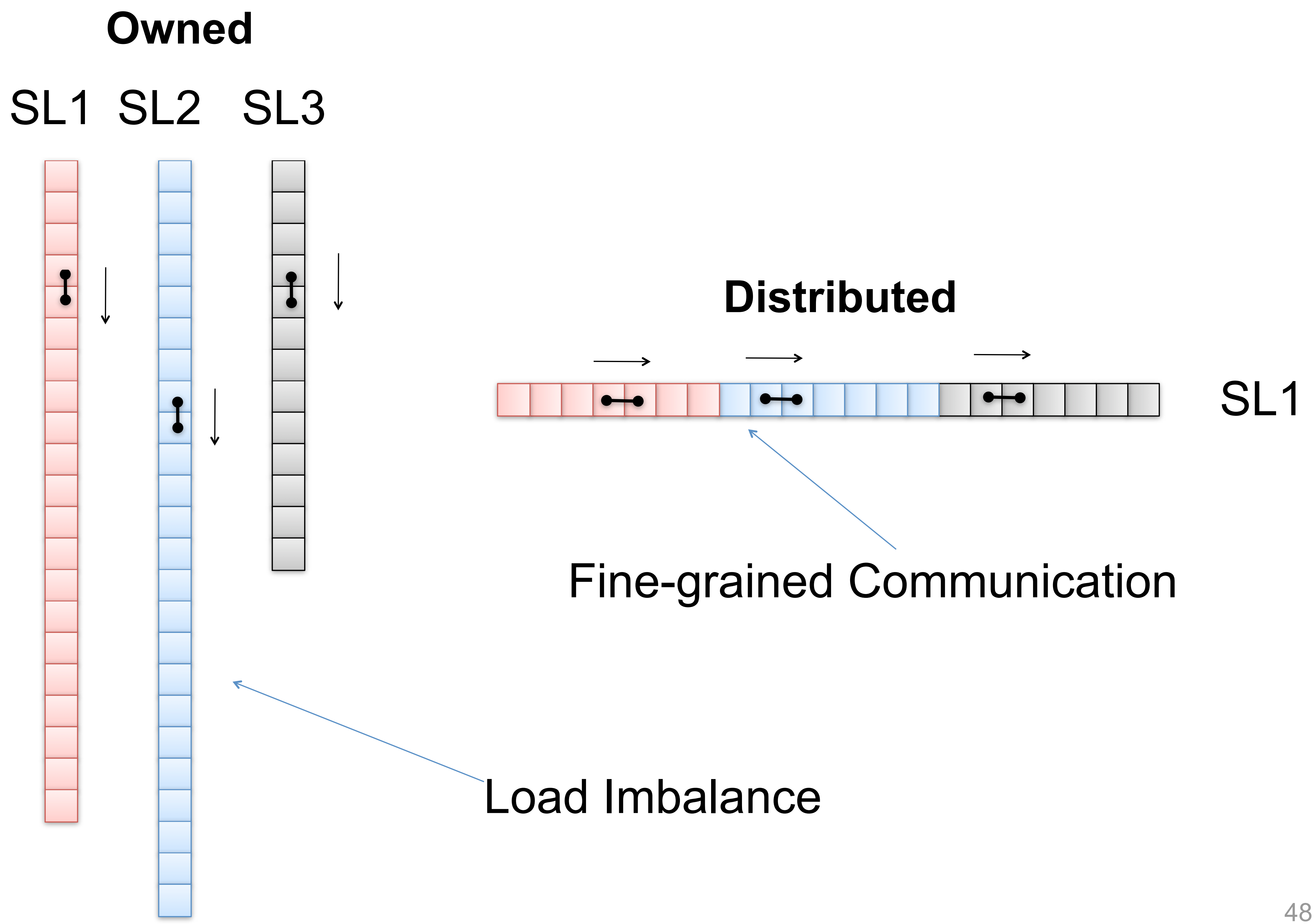
Parallel 1-D Solves, illustration



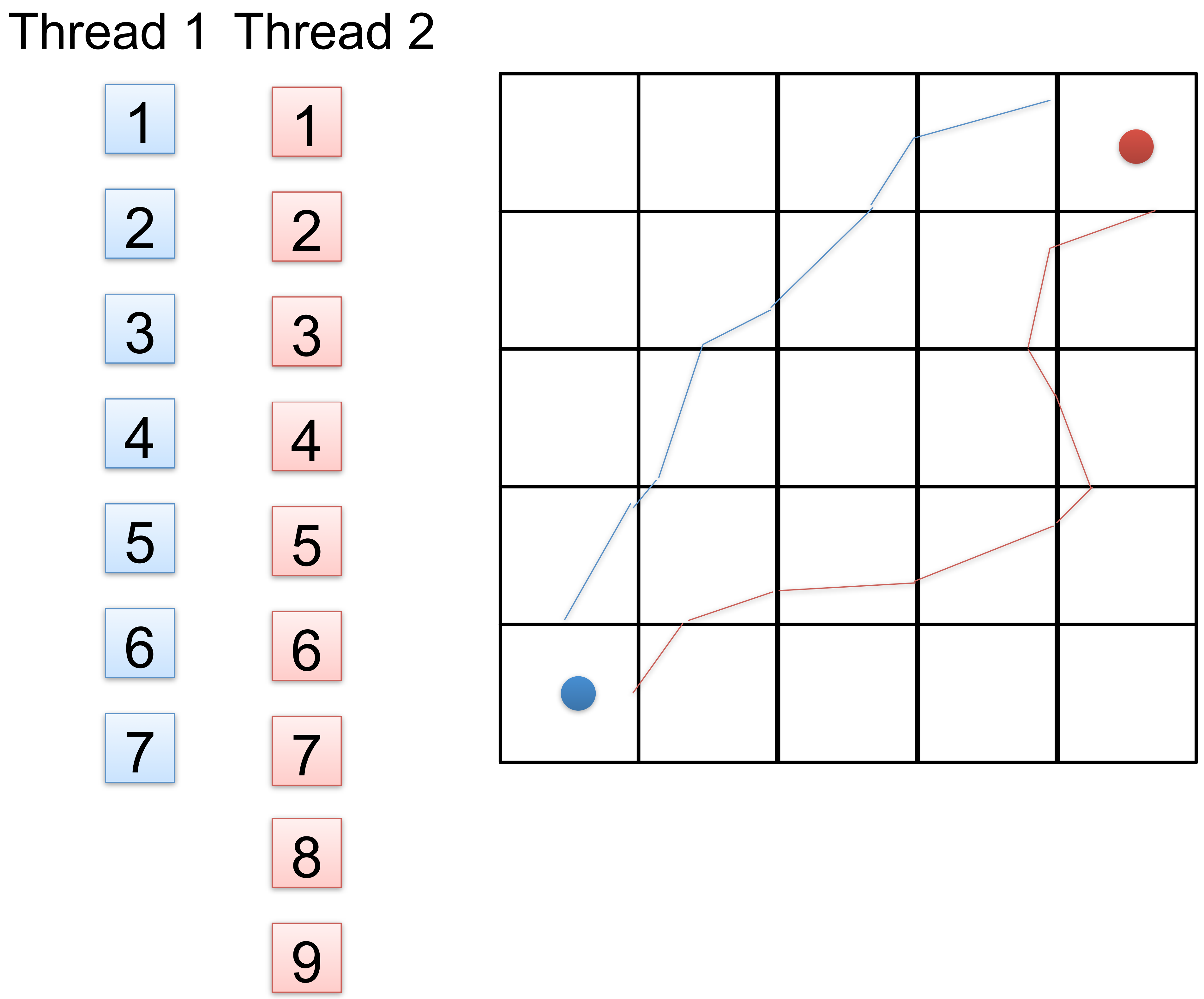
Parallel 1-D Solves, illustration



Parallel 1-D Solves, illustration

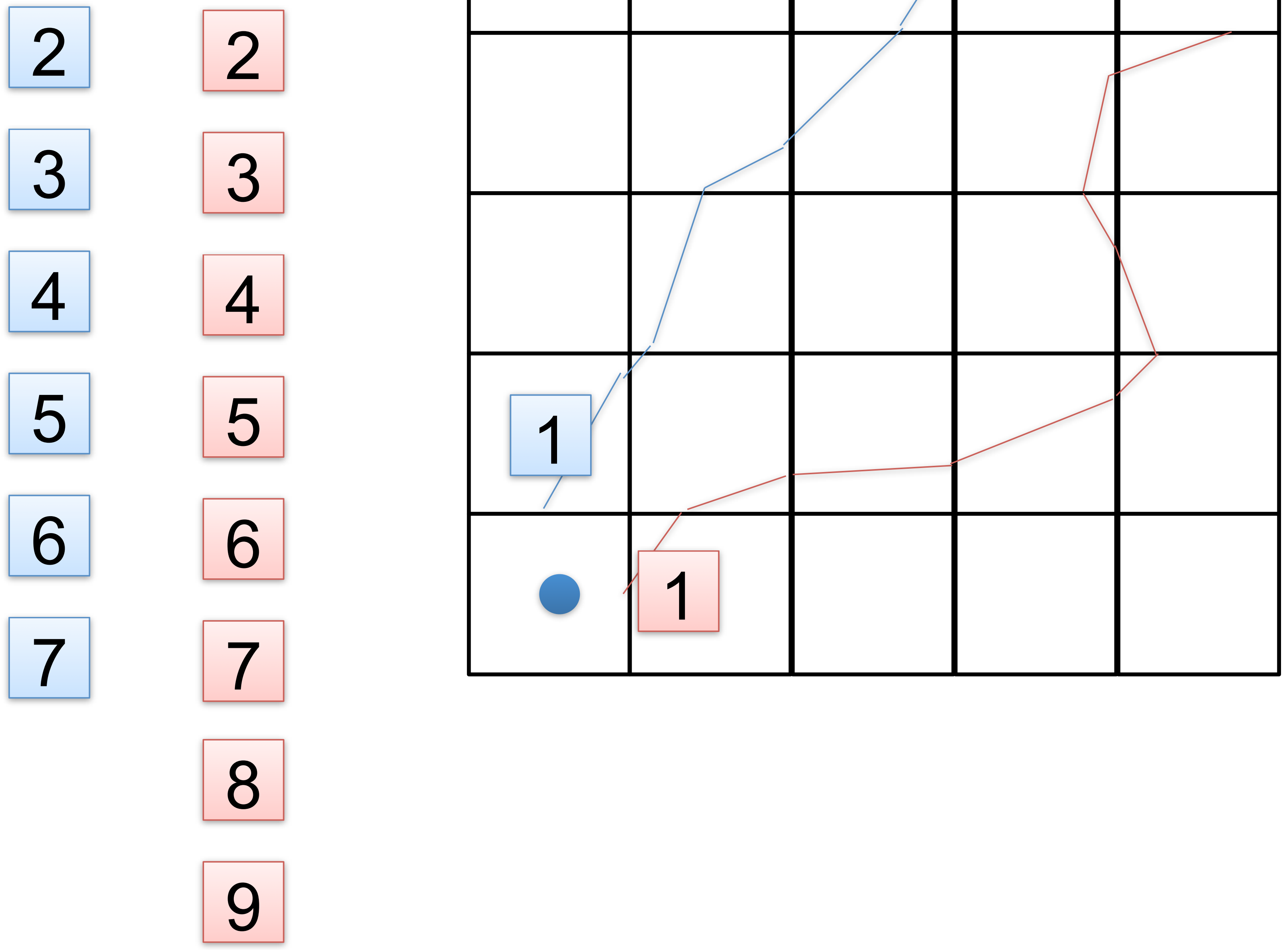


Parallel Mapping, illustration



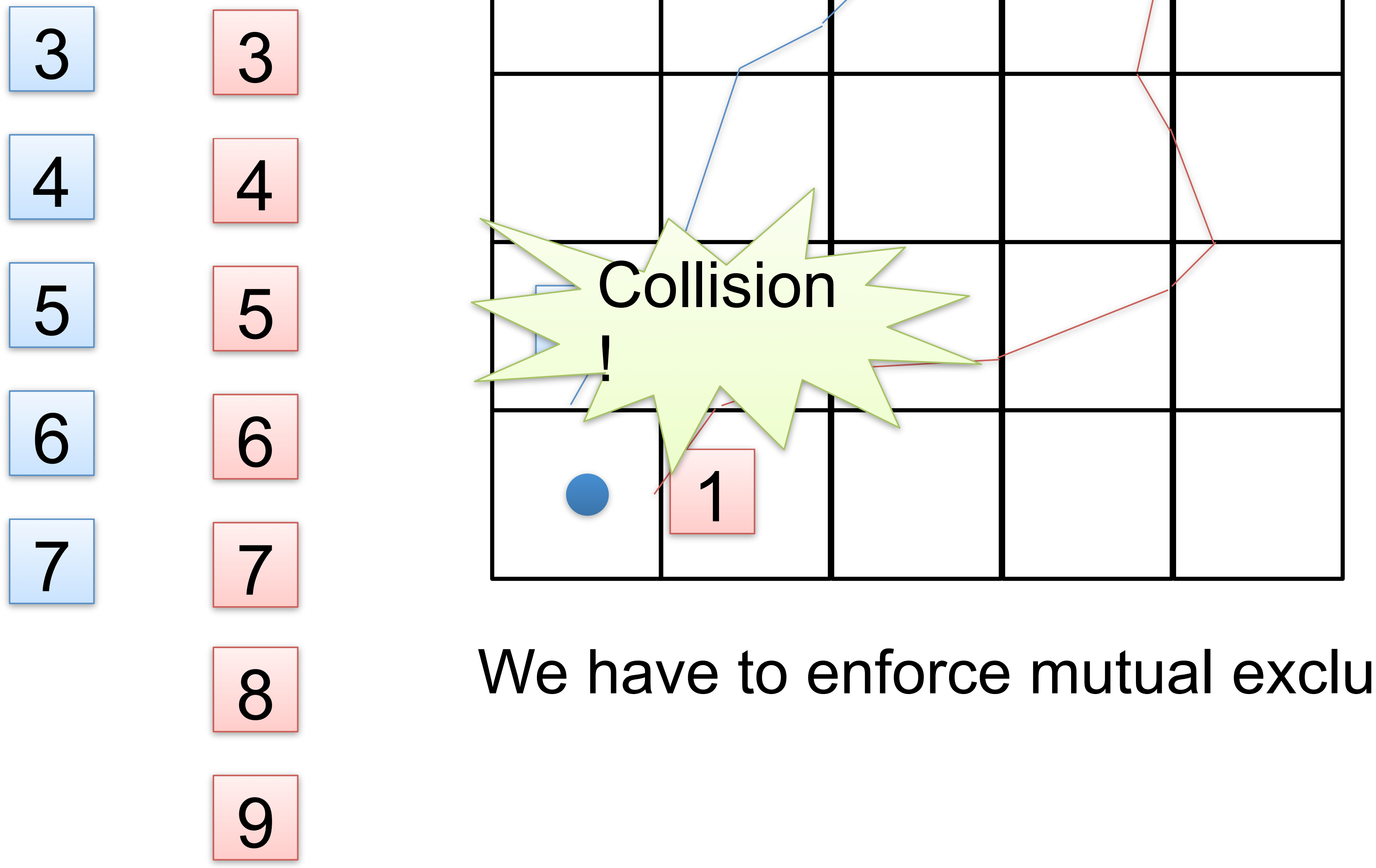
Parallel Mapping, illustration

Thread 1 Thread 2



Parallel Mapping, illustration

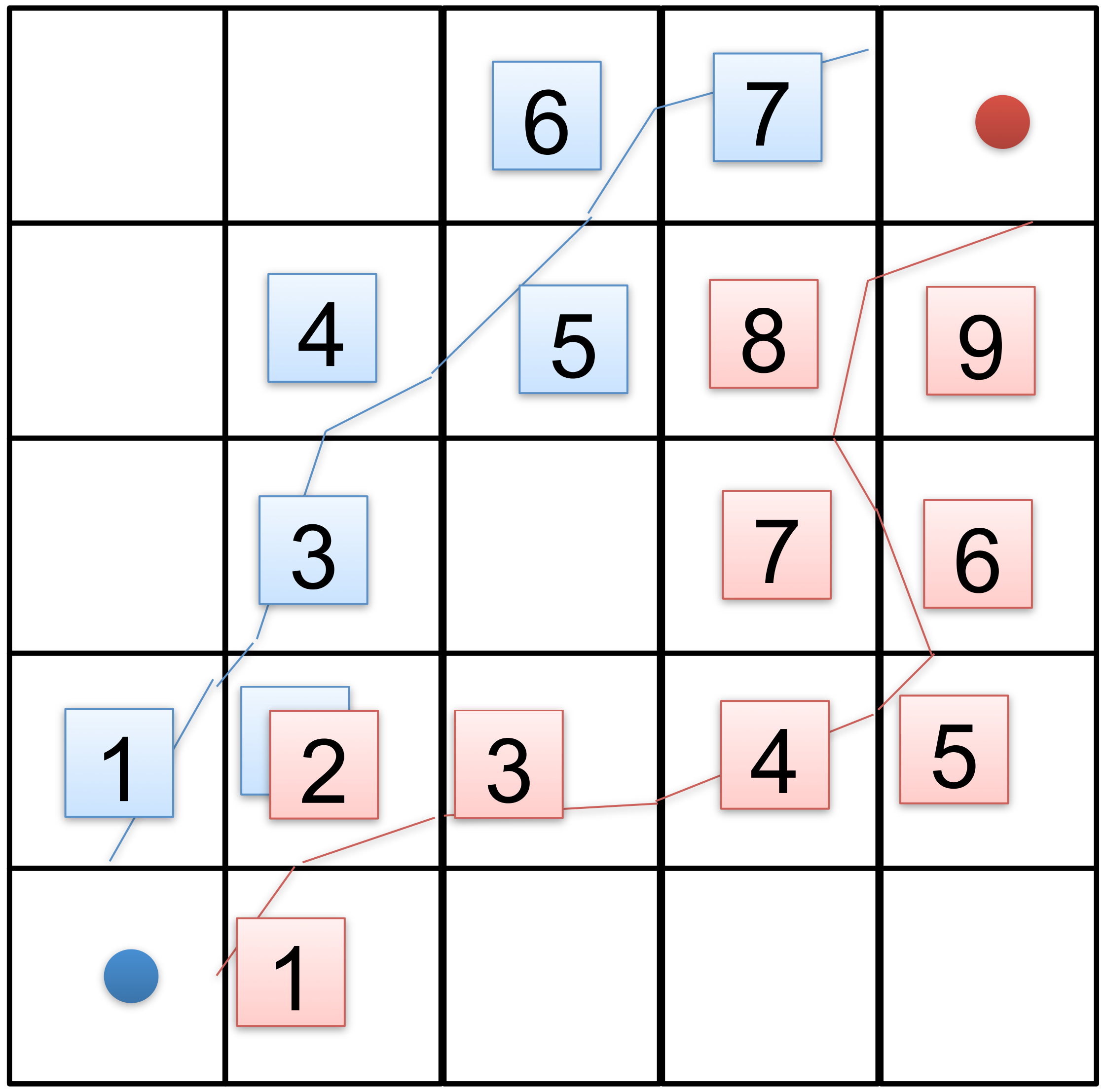
Thread 1 Thread 2



We have to enforce mutual exclusion

Parallel Mapping, illustration

Thread 1 Thread 2



We have to enforce mutual exclusion

UPPSALA UNIVERSITET

Hamster

- Single-phase tracer flow
- Uniformly refined CCAR grid
- Streamlines are implemented in a way comparable to most industrial codes (3DSL, FrontSim)
  - SPU upwinding scheme for 1-D solves
  - Simple averaging scheme for mapping
  - Dual trace for storing segments
  - 2-point flux approximation (7-point stencil)
  - GMRES solver preconditioned using BoomerAMG from the HYPRE package
  - Tracing using Pollock's method

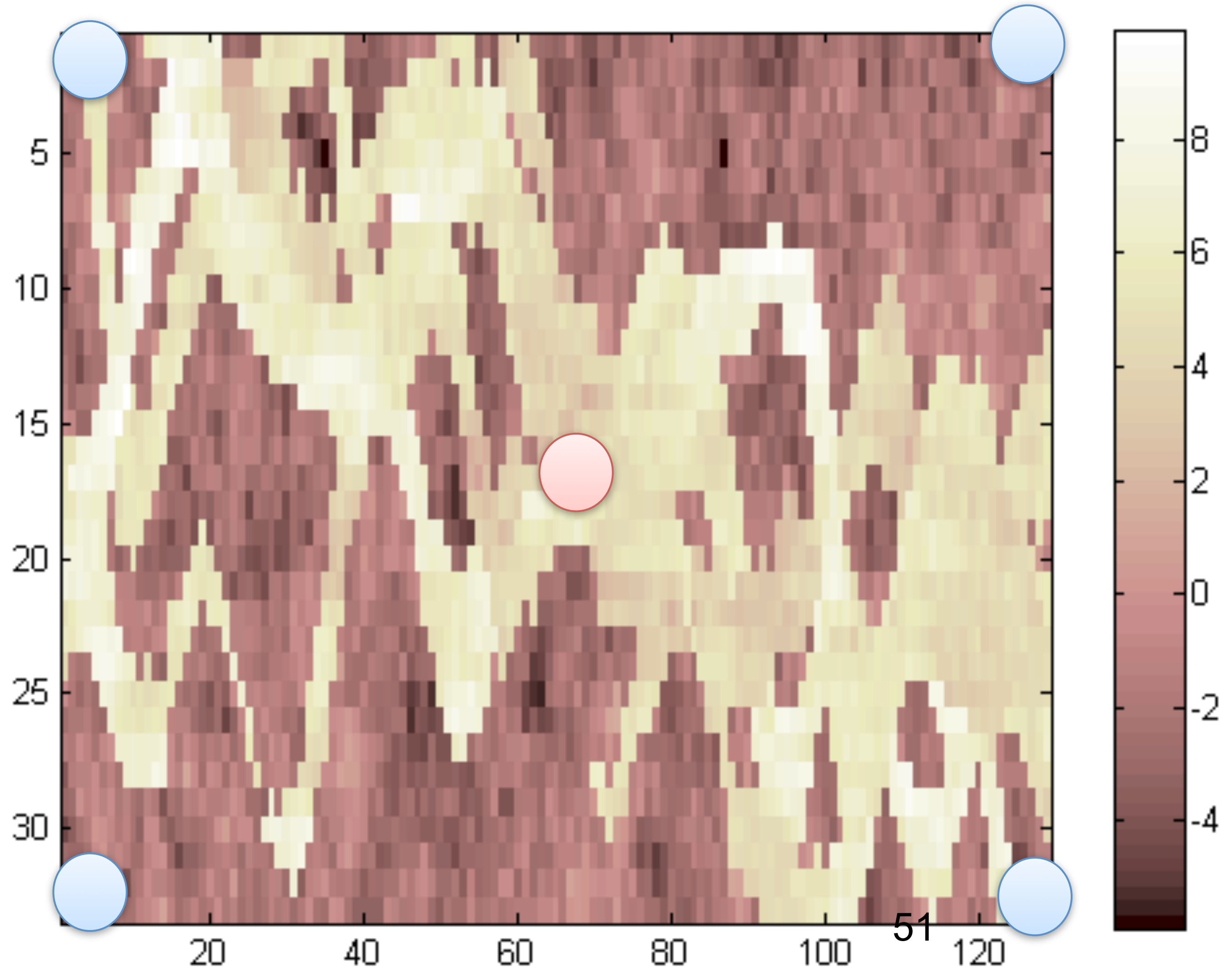
UPPSALA UNIVERSITET

Test Case #1, SPE10

🔵 Production well at constant BHP of 4000 psi

🔴 Injection well at constant BHP of 10000 psi

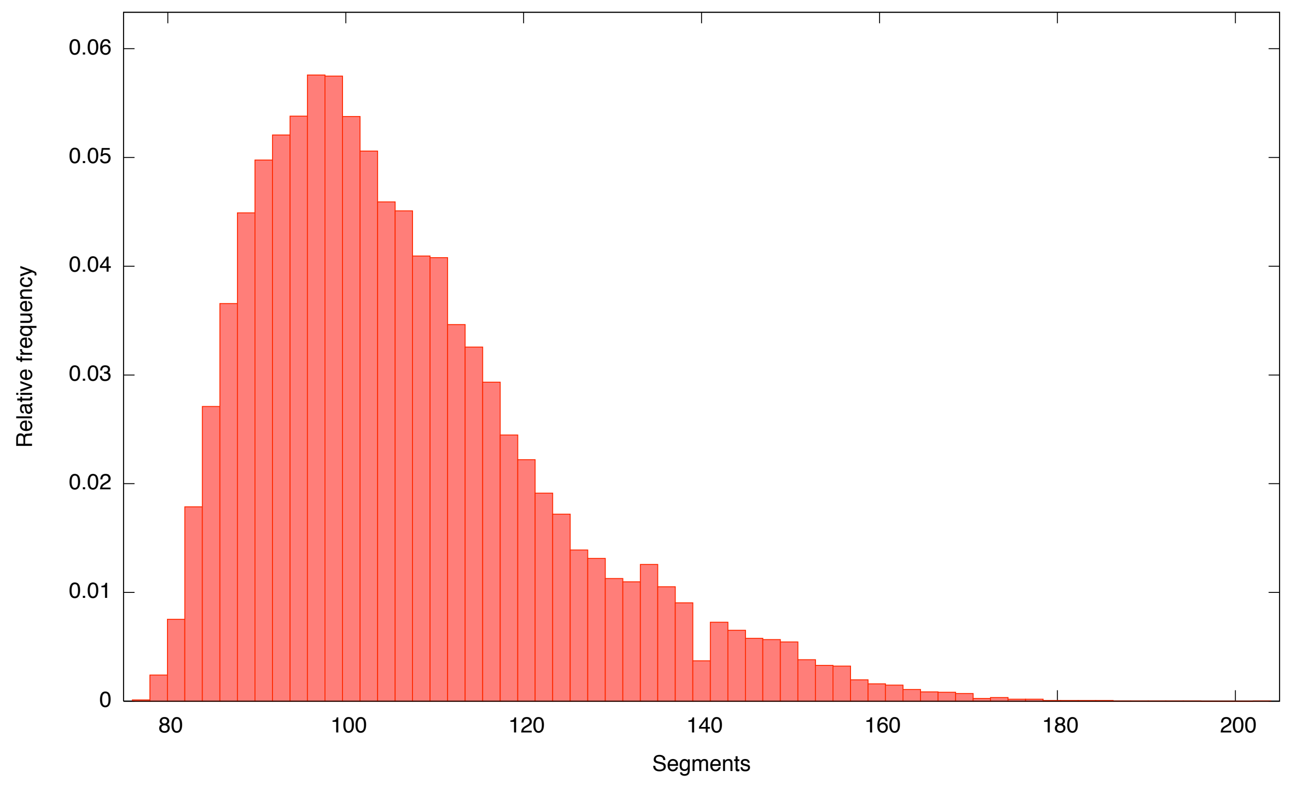
- We cut out a domain of size 32x128x32 from the bottom 32 layers of SPE10
  - 131,072 cells
- One global time step of 2000 days (~5 years)
- 10,096 streamlines

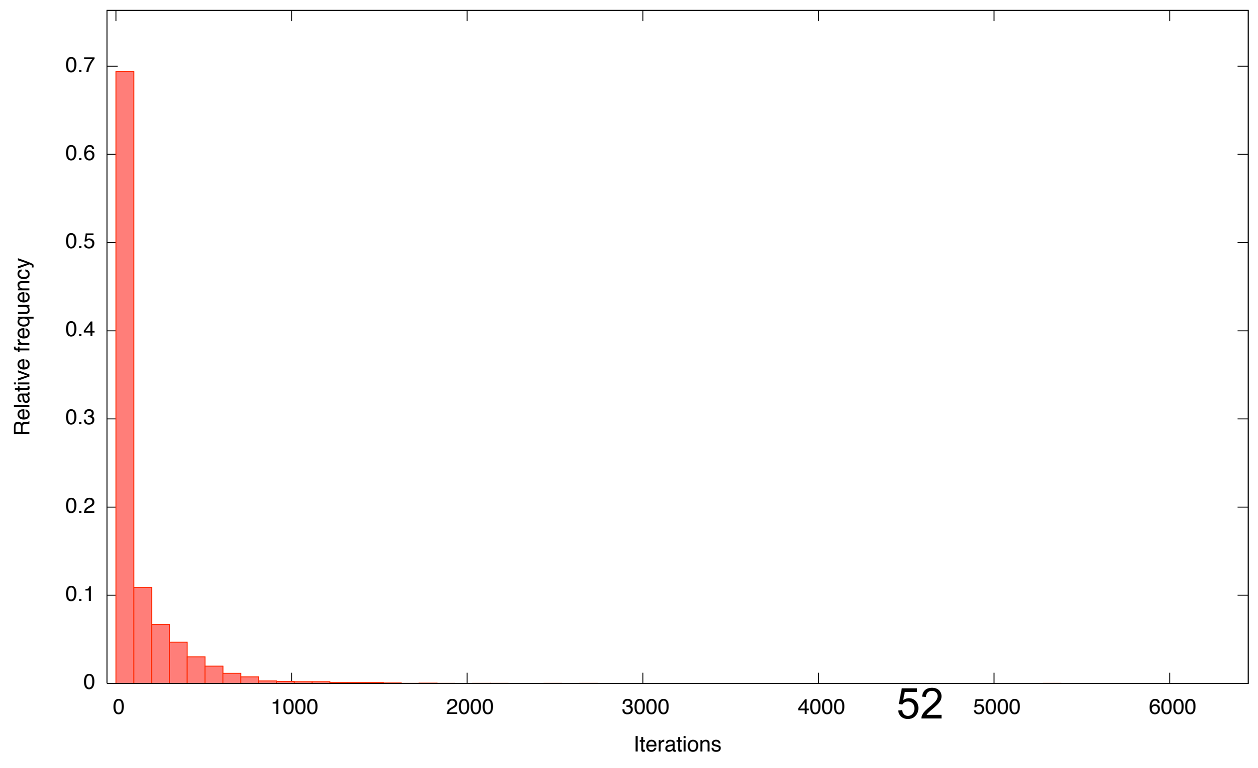


UPPSALA UNIVERSITET

Streamline Statistics, SPE10

	Min	Max	Median	Average	Std.Dev
Number of Segments	77	203	103	107	106
Number of Iterations	1	6296	19	119	244





UPPSALA
UNIVERSITET

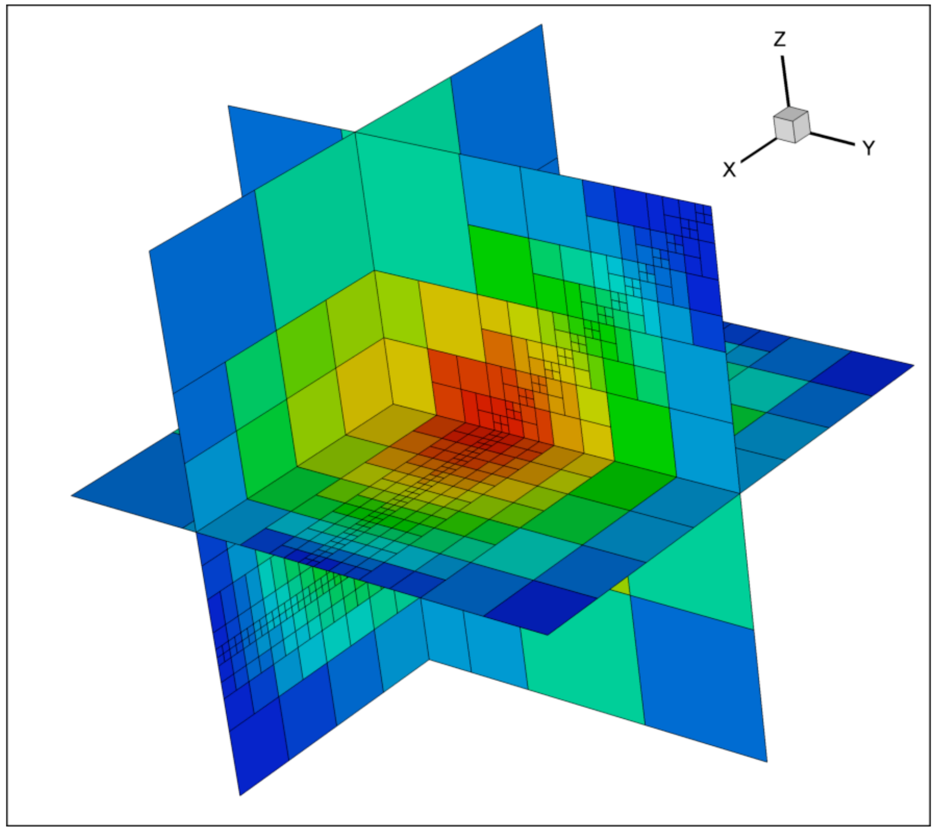
Performance on 4 way dual-core Opteron

Threads	Execution Time (s)	Speedup
1	12.01	1.00
2	6.64	1.82
4	3.74	3.23
8	2.21	5.70

UPPSALA UNIVERSITET

Scalability Problems

- Did not take NUMA into account
  - Linux does not yet support page migration
- Cell and face ordering on unstructured grids
  - Every cache line should be used
  - Communication
  - False sharing



UPPSALA UNIVERSITET

Performance on Sun Niagara 2 (UltraSPARC-T2)

Threads	Execution Time (s)	Speedup
1	84.40	1.00
2	42.44	1.99
4	21.36	3.95
8	10.91	7.73
16	6.27	13.47
32	3.81	22.15
64	157.88	0.53

Manual (Optimal) Load Balancing

- In the static case, we can improve load balance by assigning a **weight** representing the compute time
- We then formulate a discrete optimization problem using these weights
- Possible parameters for load balancing weights
  1. Rock properties (segment counts)
  2. Total time of flight (iterations)
  3. Well placement (segment count)
  4. Size of multi-phase region (flash calculations)
  5. Cache misses and communication costs (hardware)

UPPSALA
UNIVERSITET

Load Balanced Algorithm

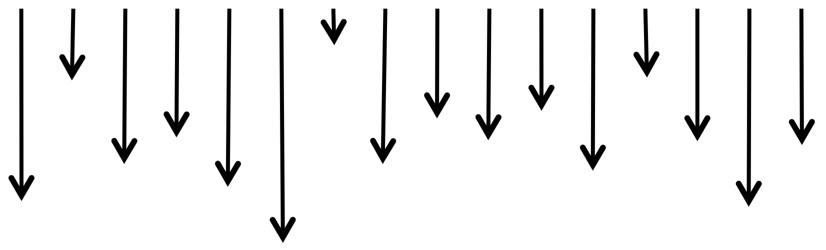
Algorithm 2 Load Balanced Streamline Simulation

```
Require: $T_{end} \leftarrow$ simulation end time
Require: $dt \leftarrow$ global timestep
Require: $T \leftarrow dt$
 1: repeat
 2:    Solve pressure equation
 3:    Calculate velocity field
 4:    repeat
 5:       Select launch points for streamlines and build bundle
 6:       Assign launch points in bundle to threads
 7:       for all streamlines in bundle do
 8:          1:st trace, count segments
 9:          2:nd trace, pick up pressure grid data, store the segments
10:       end for
11:       Load balance bundle
12:       for all streamlines in bundle do
13:          Build streamline grids from segment and pressure grid data
14:          Solve the corresponding 1-D transport equations
15:       end for
16:       Map new values of the transport variables to the pressure grid
17:    until Domain is sufficiently covered
18:    $T \leftarrow T + dt$
19: until $T \geq T_{end}$
```

UPPSALA UNIVERSITET

Reassignment problem, 16 streamlines

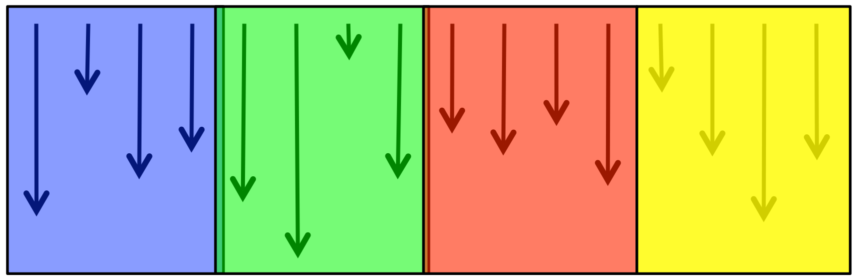
Tracing



UPPSALA
UNIVERSITET

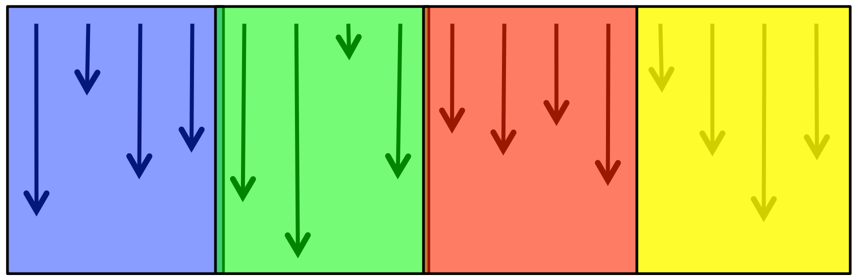
Reassignment problem, 16 streamlines

Tracing

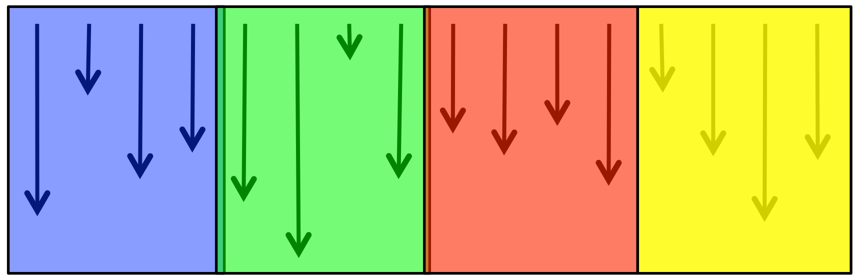


Reassignment problem, 16 streamlines

Tracing

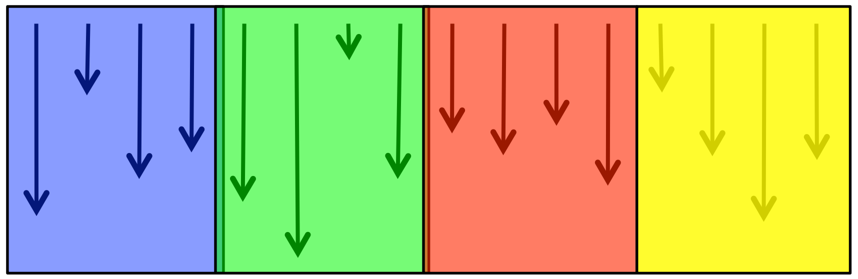


No Load Balancing

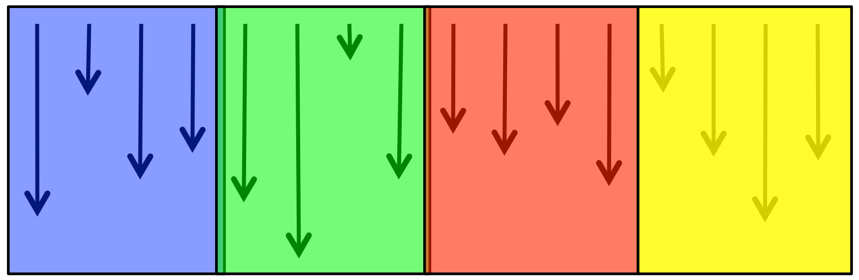


Reassignment problem, 16 streamlines

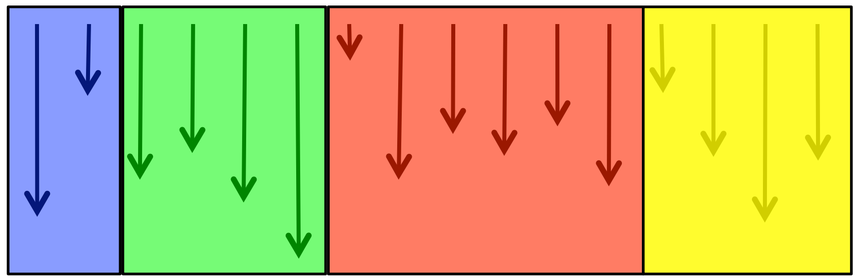
Tracing



No Load Balancing



Load Balancing



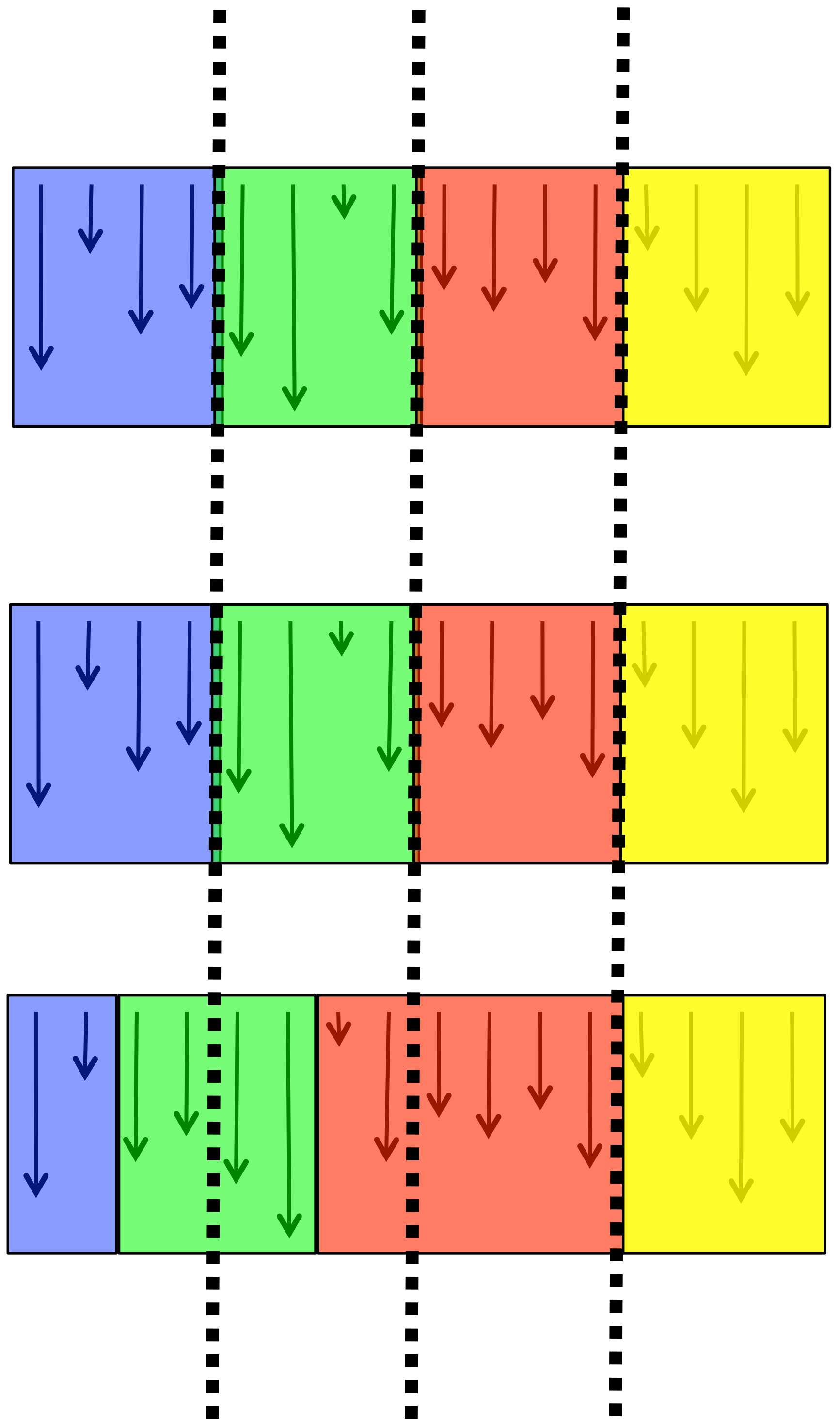
UPPSALA UNIVERSITET

Reassignment problem, 16 streamlines

Tracing

No Load Balancing

Load Balancing



UPPSALA UNIVERSITET

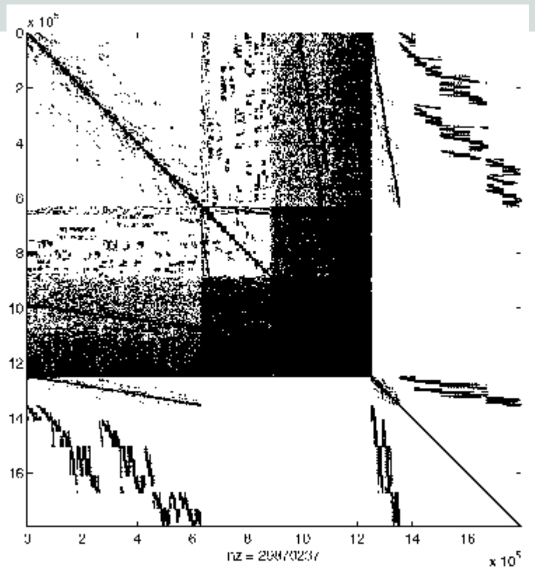
Percentage of all streamlines moved

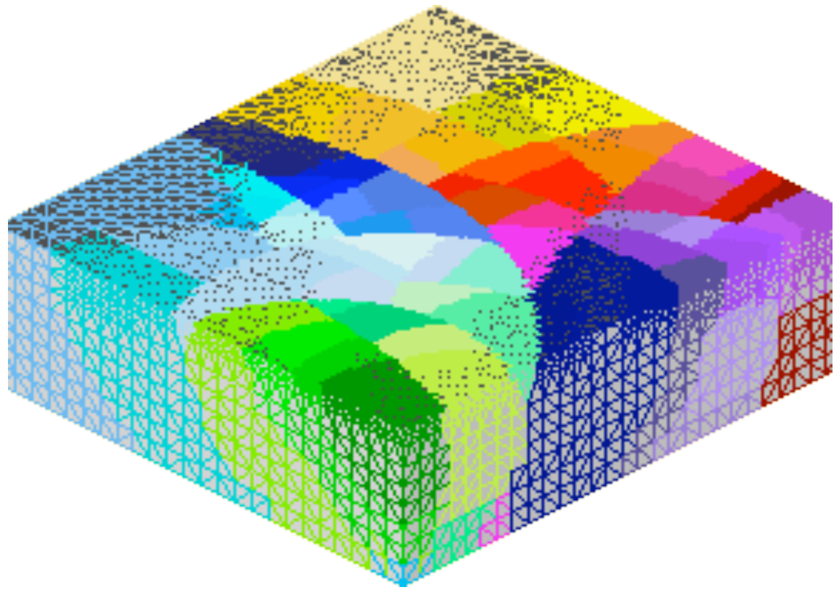
	8 Threads	16 Threads
SPE10	28.9%	40.9%
ANOTHER CASE	15.6%	31.0%

UPPSALA UNIVERSITET

Origins of non-zero structure

- Every row in the matrix corresponds to a set of cells, a small **sub-domain**
- The non-zero pattern can be controlled by reordering of the unknowns (cells)
  - Graph partitioning, Space Filling Curves, Reverse Cuthill-Mckee, ..
- Load balance can be achieved if we replace the simple **N/num_threads** scheme
  - Assumes an efficient a-priori load estimator
  - Area and connectivity of the sub-domains control the amount of communication needed

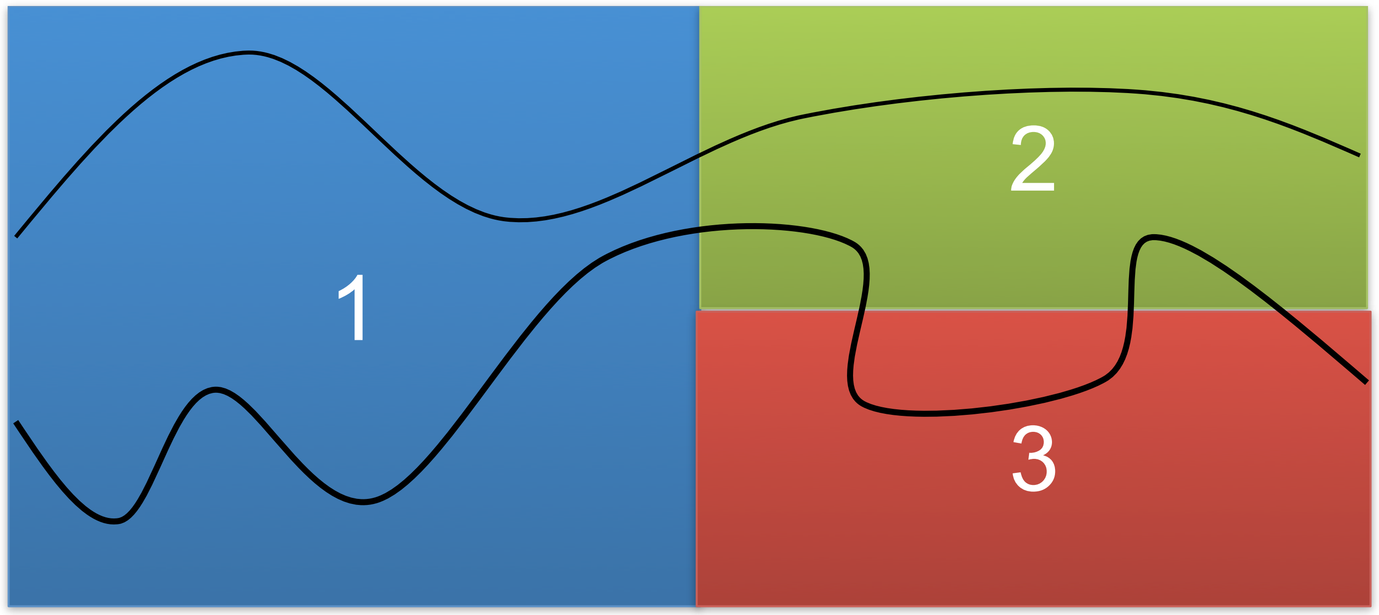




UPPSALA UNIVERSITET

Domains and Streamlines

- Consider a partitioning of the pressure grid into three partitions
  - This is needed for extracting parallelism from the pressure solver
  - Each sub-domain corresponds to set of cells or equivalently, a set of rows of the iteration matrix



UPPSALA UNIVERSITET

Naive “Parallel” Solution

- Collect the entire velocity field of the sub-domains to a master node
- Trace all the streamlines on the master node
- Copy the streamline grids back to the other nodes
  - Scheduling and load balancing
- Do local transport solve
- Map back

UPPSALA
UNIVERSITET

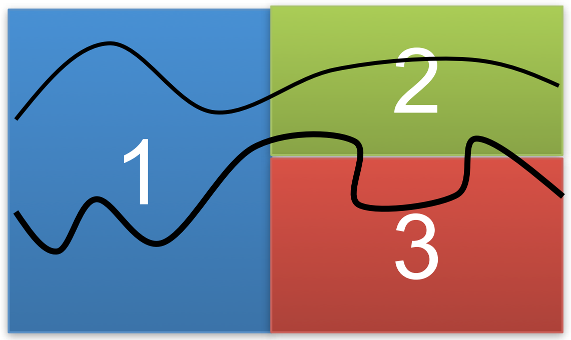
Analysis

- The entire velocity field must fit in the memory of the master node
  - No chance for giga-cell models
- A node generates lots of communication and the communication will be localized in time (bursty)
  - Very bad for slow interconnects
- A node can only map back streamline data to the sub-domain that it owns
  - More communication needed to pass all the streamlines around in the mapping step
- Probably not very scalable

UPPSALA UNIVERSITET

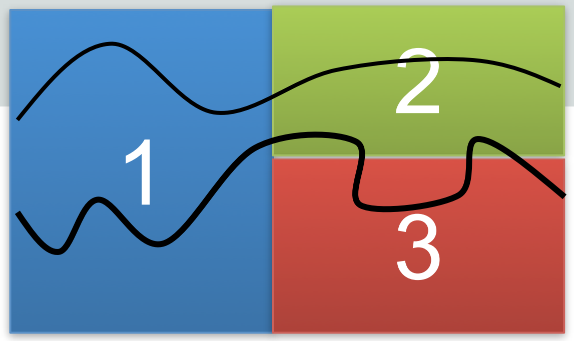
A Pipelined Solution

- For simplicity assume that we trace from left to right
- The segments of an individual streamline may cross all three domains
  - Sub-domains 2 and 3 must wait until the set of streamlines crossing sub-domain 1 have reached the boundary
  - While domains 2 and 3 trace the continuation of this set we can start tracing a new set in sub-domain 1
- In this way we can overlap or **pipeline** the tracing of the streamlines



UPPSALA UNIVERSITET

Pipelined Algorithm



- For each processor (sub-domain), repeat:
  1. Wait for start points
  2. Trace all local streamlines
  3. Hand-off exit points to neighbor
- Think of the sub-domains as grid cells in a coarse grid.
- Every sub-domain contains one segment
  - Which consist of smaller segments based on the actual grid cells

UPPSALA
UNIVERSITET

Domain Graph