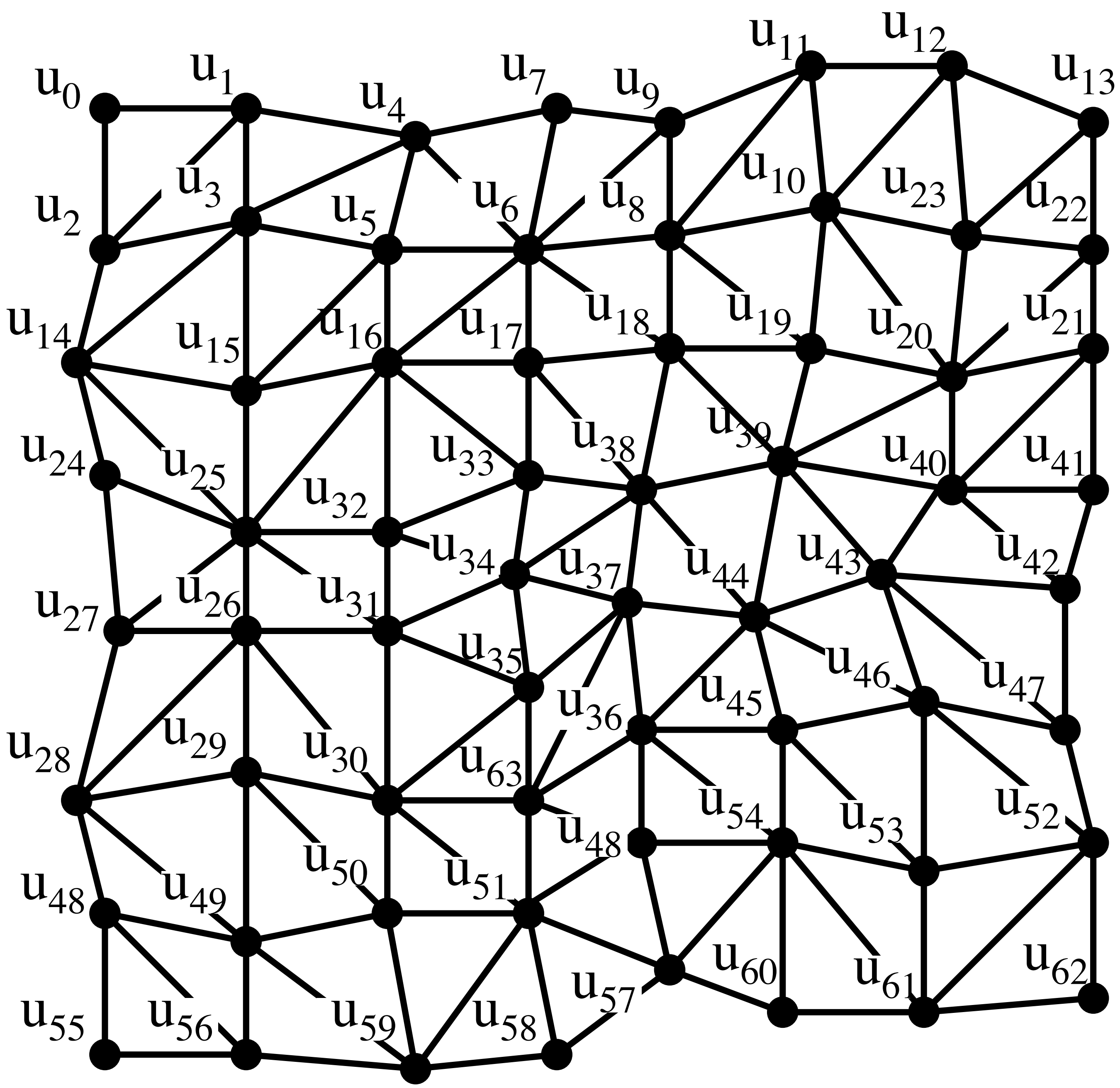
Parallelizing Gauss-Seidel Using Full Sparse Tiling

Michelle Mills Strout

CSCAPES Seminar
May 15, 2007

Colorado State University
Knowledge to Go Places

Goal: Make computations over meshes fast



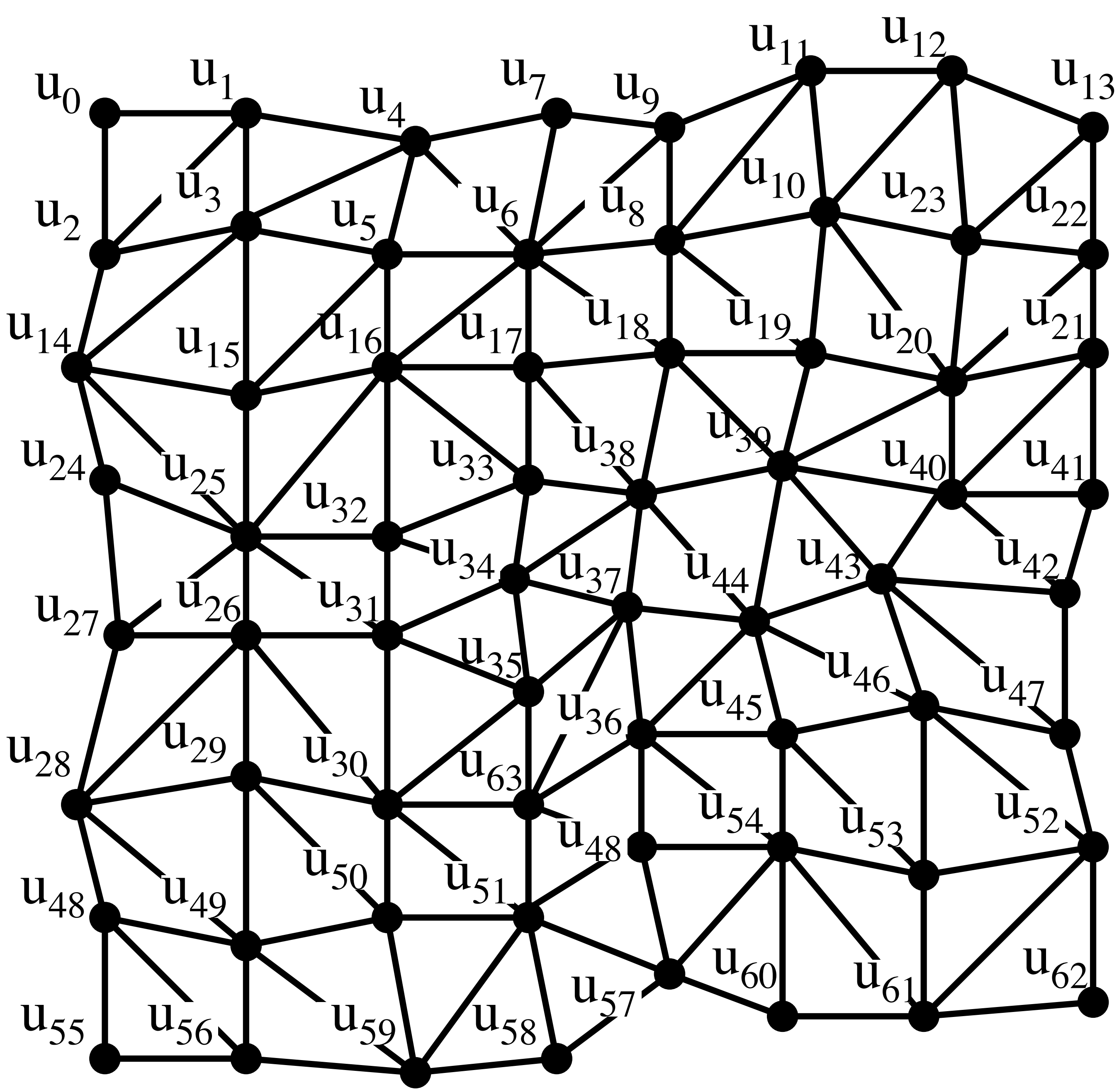
- Iterative smoothers such as Gauss-Seidel dominate the execution time of finite element applications
- Need to effectively utilize the memory hierarchy
- Performance should scale to multiple processors

Colorado State University
Knowledge to Go Places

Full Sparse Tiling to the Rescue!

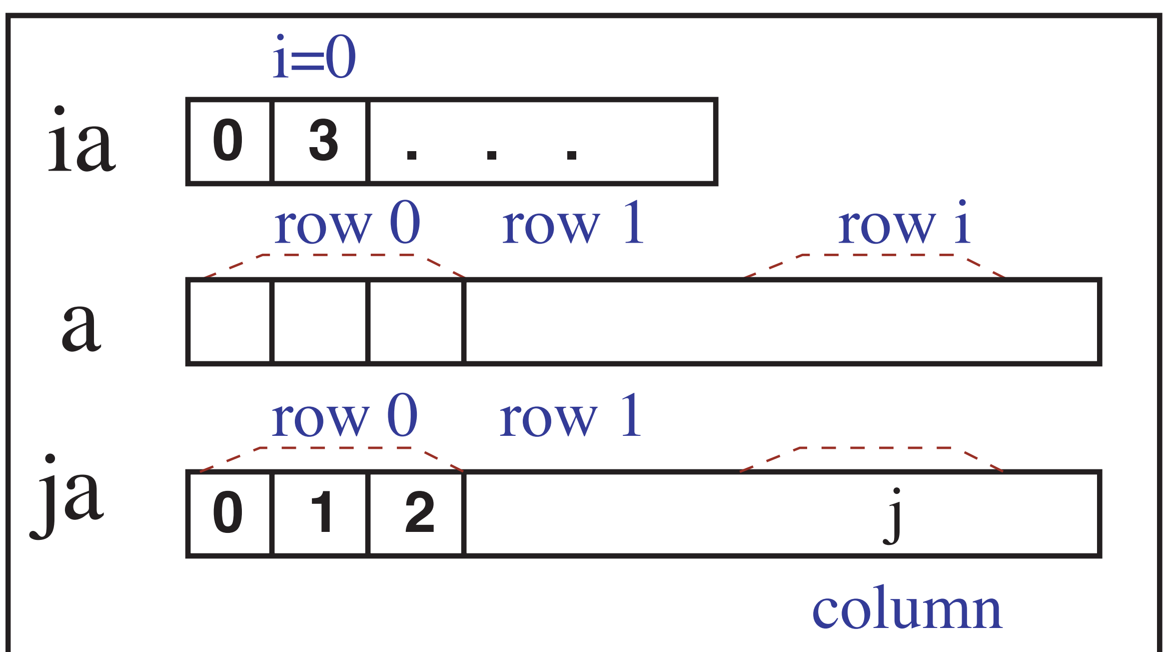
- **Problem**
  - Compile-time data reordering and computation scheduling to improve data locality and parallelism is not possible due to irregular memory references A[B[i]]
- **Solution**
  - Inspector/executor strategies perform data and computation reordering at runtime
  - Full sparse tiling is one such strategy that improves performance by exploiting parallelism, intra-iteration data reuse, and inter-iteration data reuse

Gauss-Seidel Iteratively Solves Au = f



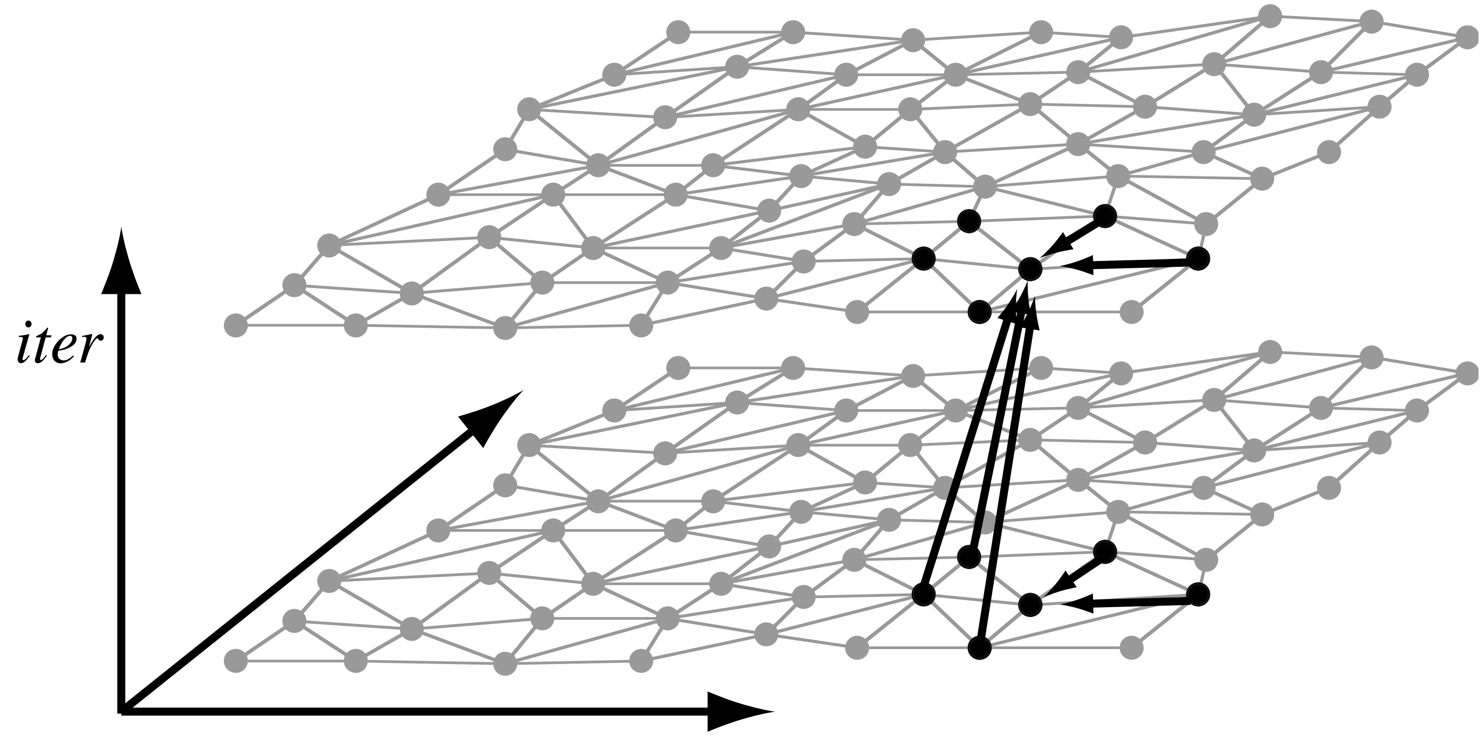
Matrix Graph

- u is a vector of unknowns
- A is a sparse matrix stored in the compressed sparse row format (CSR)

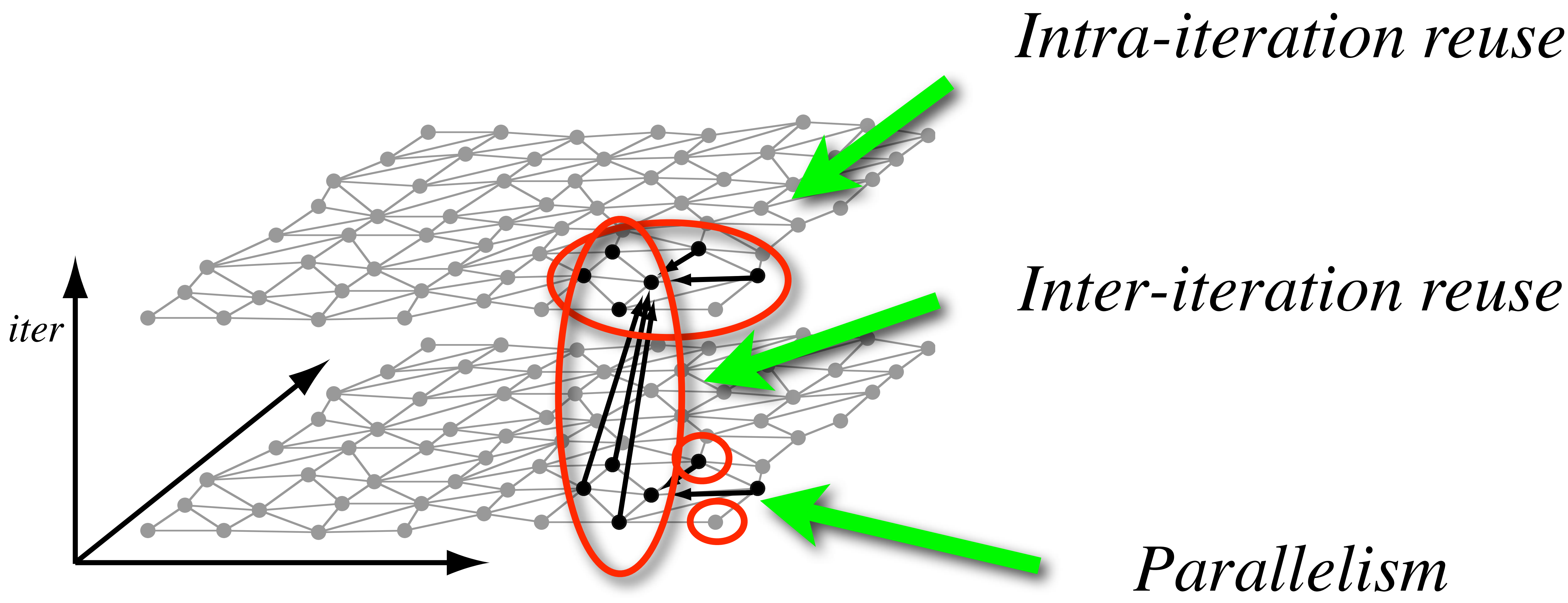


Gauss-Seidel Iteration Space

```
do iter = 1, T
  do i = 1, R
    u(i) = f(i)
    do p = ia(i), ia(i+1) - 1
      j = ja(p)
      if (j != i)
        u(i) -= a(p) * u (ja(p))
      else
        diag = a(p)
      endif
    enddo
    u(i) = u(i) / diag
  enddo
enddo
```

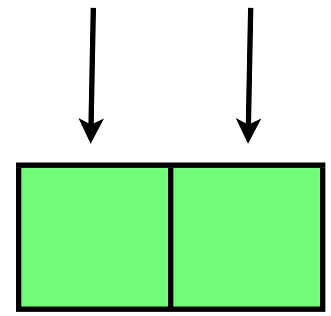


Performance Improvement Opportunities

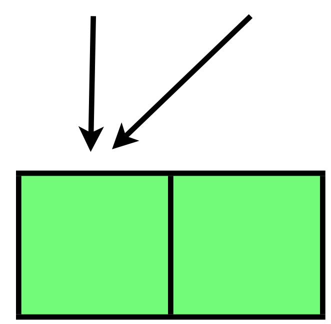


Turn Data Reuse into Data Locality

S *Spatial locality* occurs when memory locations mapped to the same cache-line are used before the cache line is evicted

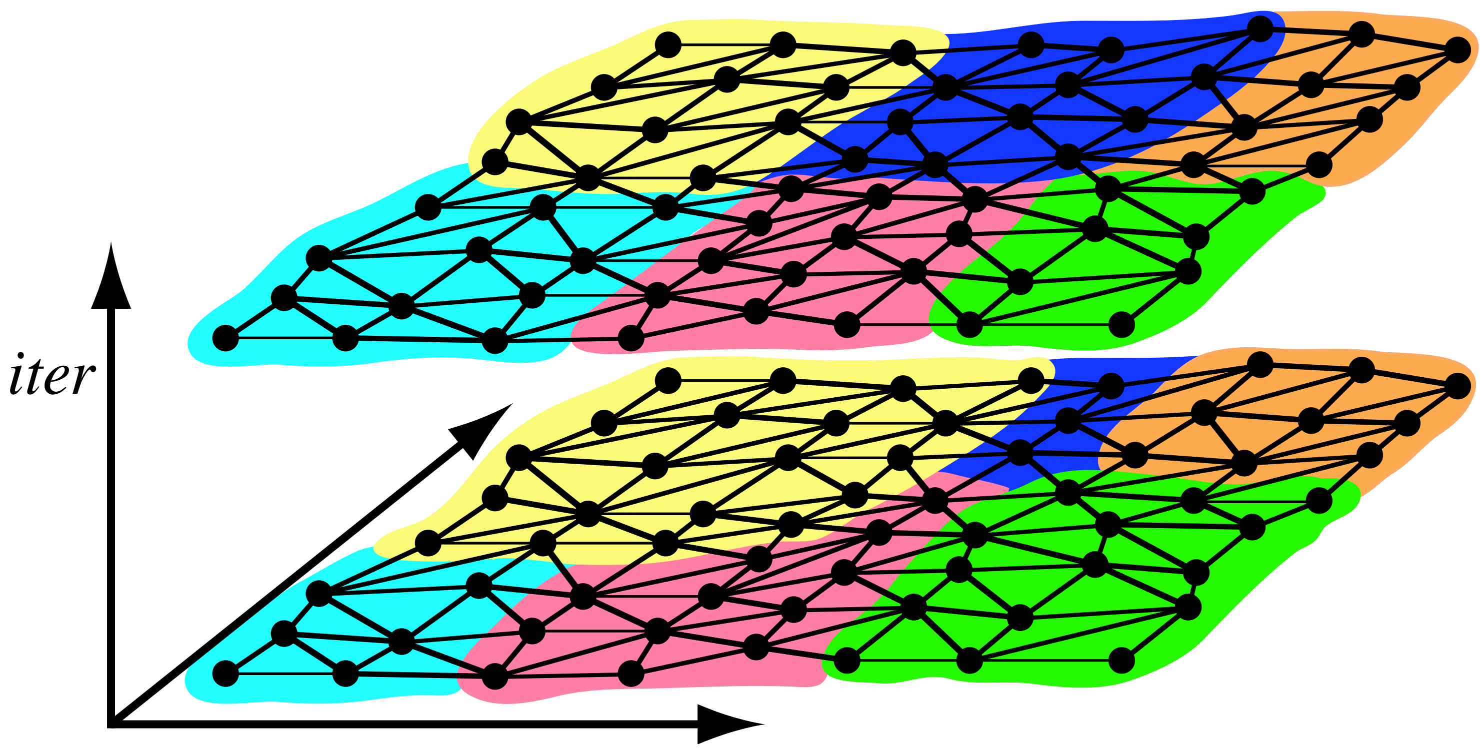


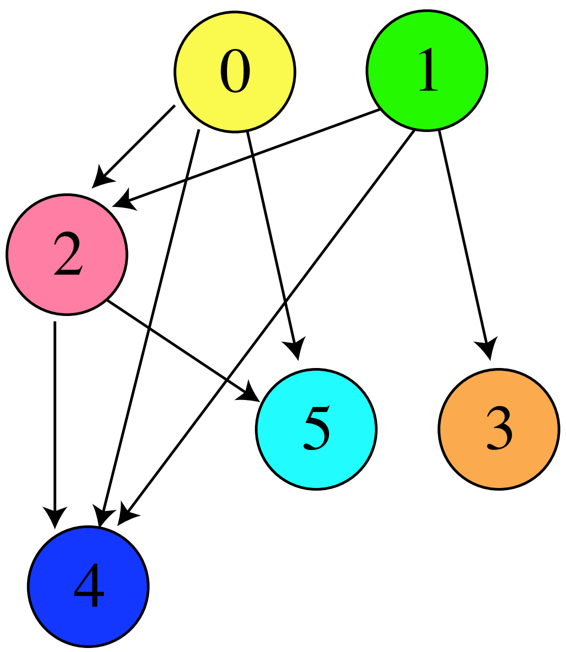
T *Temporal locality* occurs when the *same memory location* is reused before its cache line is evicted



Colorado State University
Knowledge to Go Places

Full Sparse Tiling





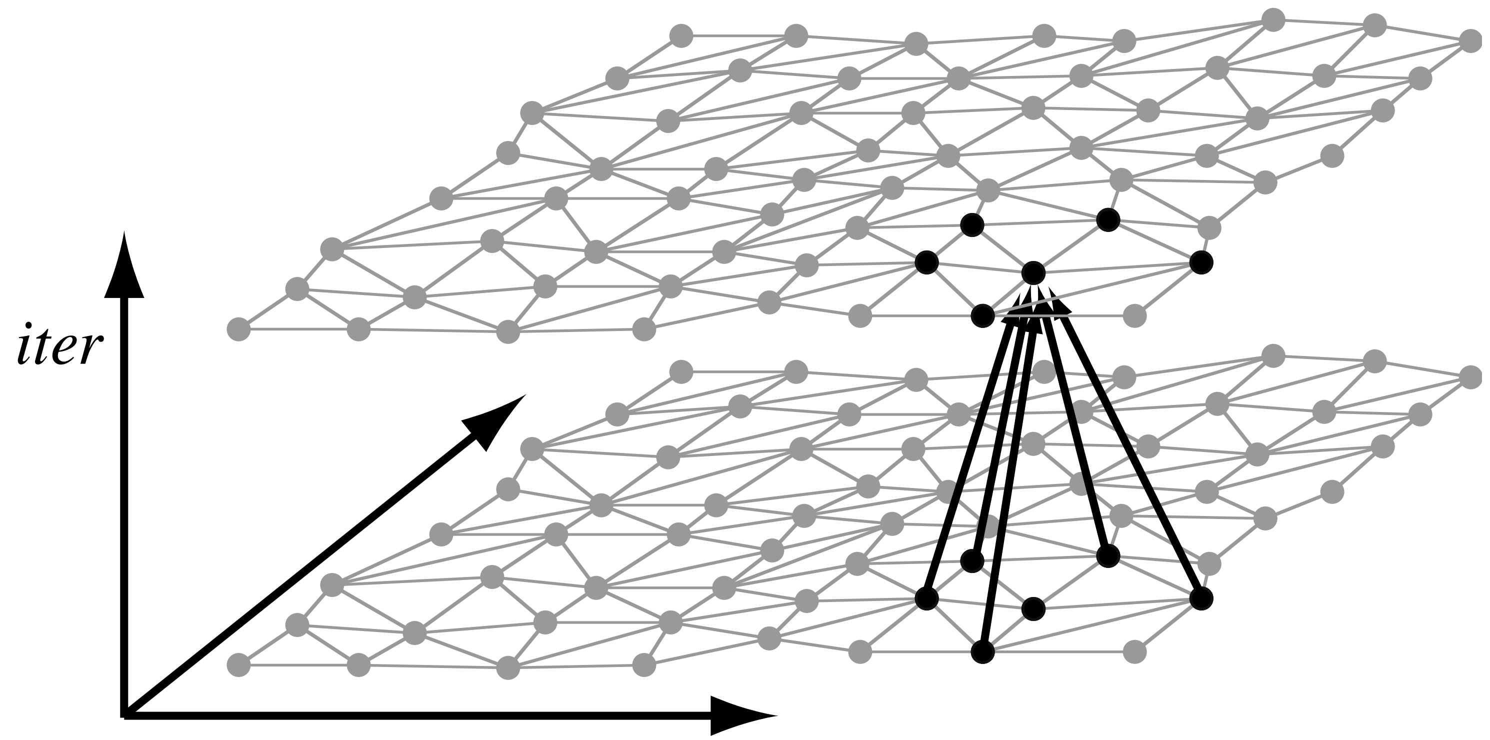
- Breaks up computation into pieces
- Each piece has intra and inter-iteration data locality
- Some pieces can be executed in parallel

Talk Outline

- Overview
- Parallelizing Jacobi with the owner computes method or full sparse tiling
- Increasing the parallelism in full sparse tile schedules
- Extra work needed to handle Gauss-Seidel
- Experimental Results

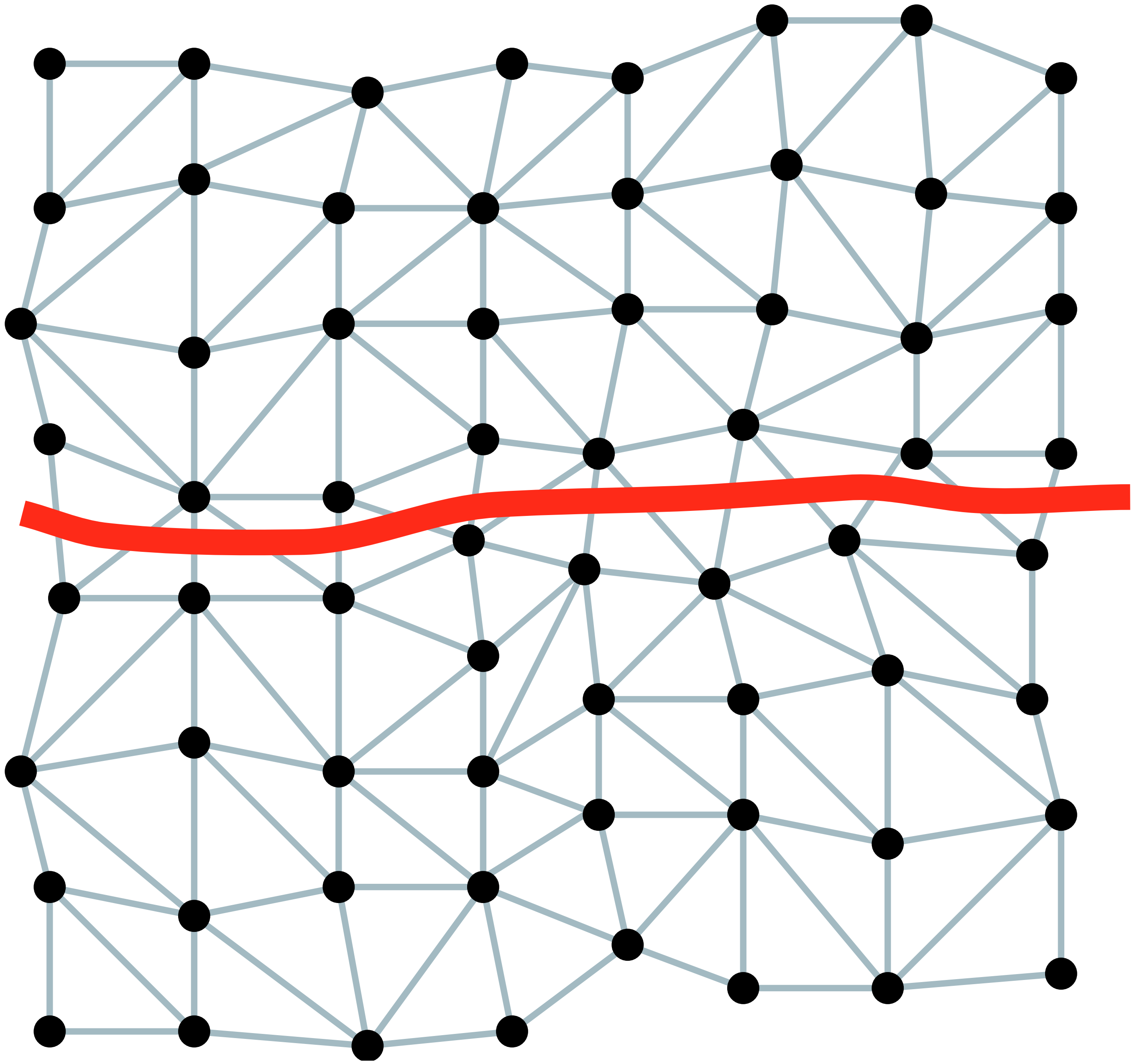
Jacobi: similar yet simpler

```
do iter = 1, T
  do i = 1, R
      u(i) = f(i)
      do p = ia(i), ia(i+ 1) - 1
         j = ja(p)
         if (j != i) then
            u(i) -= a(p)*tmp (j)
         else
            diag = a(p)
         endif
      enddo
      u(i) = u(i)/diag
  enddo
  do i = 1, R
      tmp (i) = u(i)
  enddo
enddo
```



- data dependences not known until runtime
- no intra-iteration dependences
- i loop is a reduction, therefore parallelizable

Parallelize with Owner Computes Method

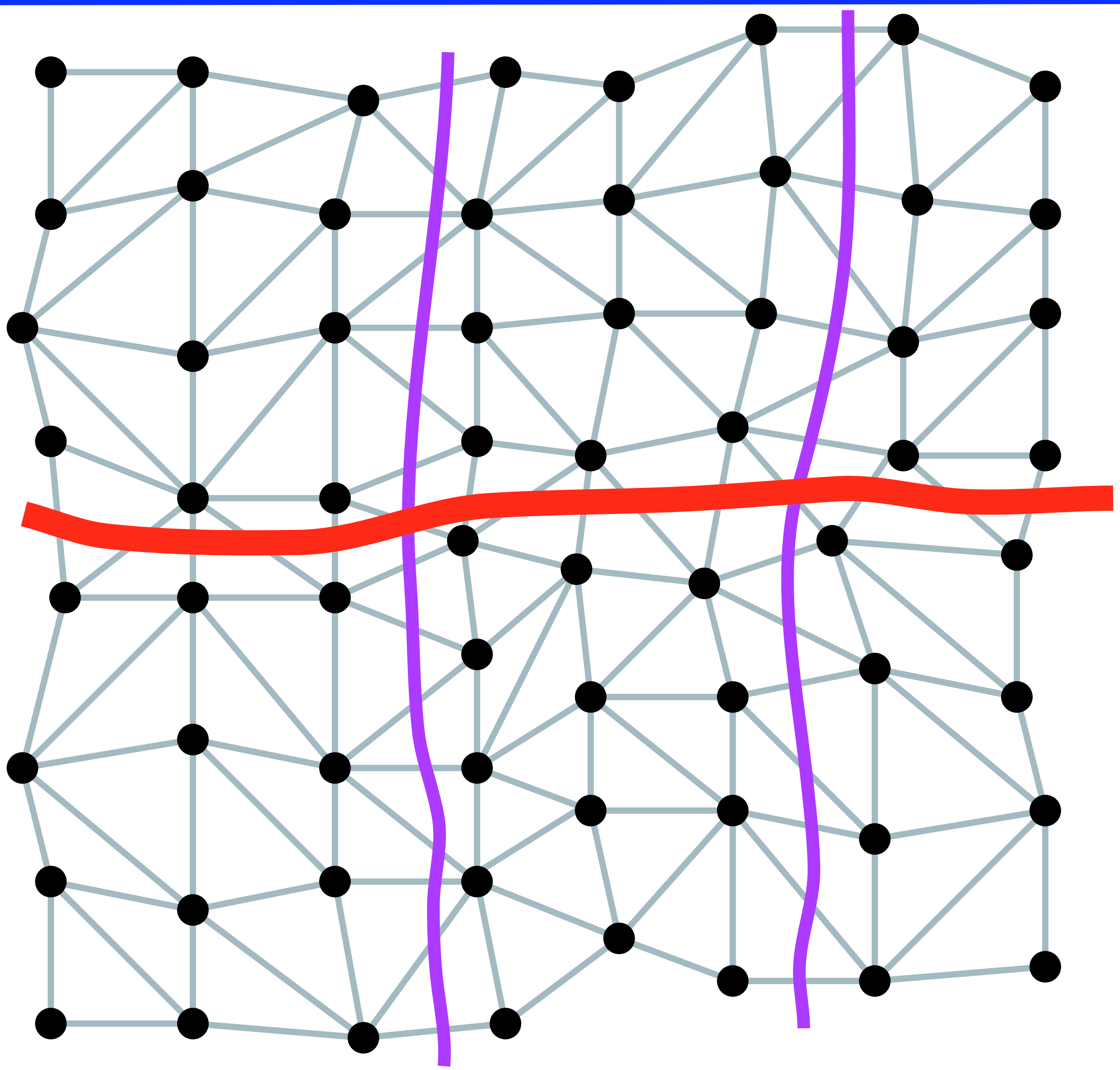


At each convergence iteration ...

- each partition receives data from previous convergence iteration
- each partition executes in parallel
- each partition sends data

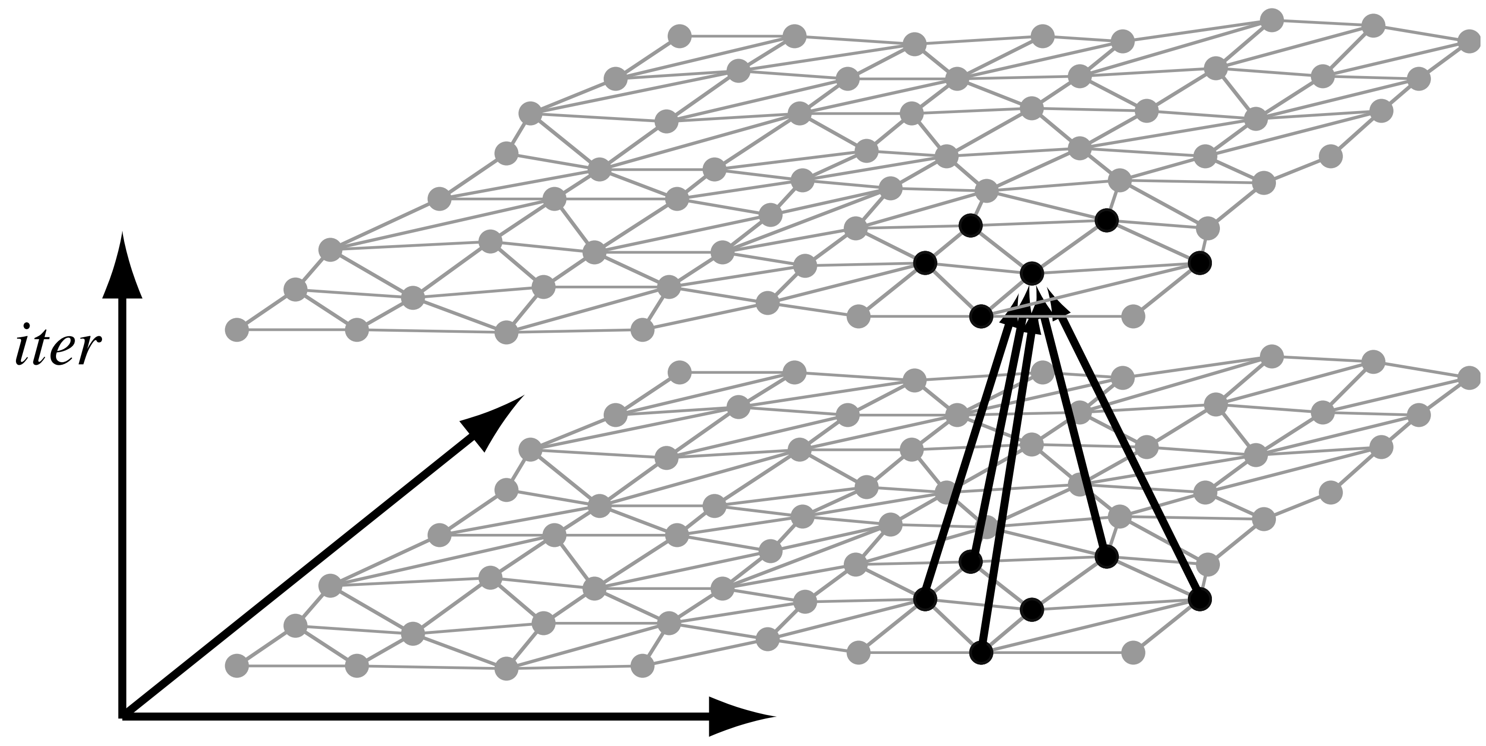
Colorado State University
Knowledge to Go Places

Locality in Owner Computes Method

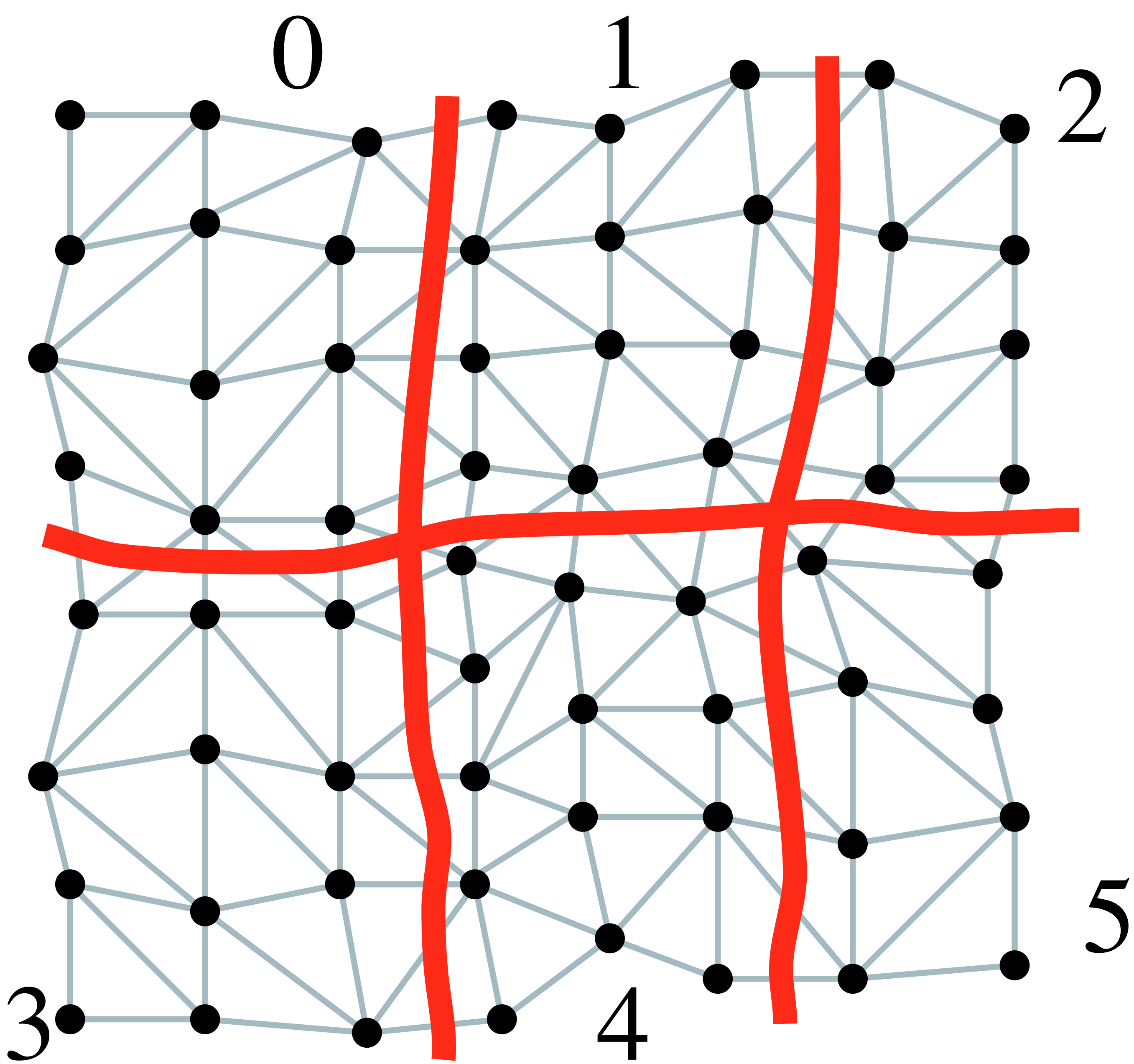


- Intra-iteration locality
  - schedule by sub-part
  - reorder data for consecutive order in sub-parts
- NO Inter-iteration locality, main partitions don't fit in cache

Jacobi Iteration Space

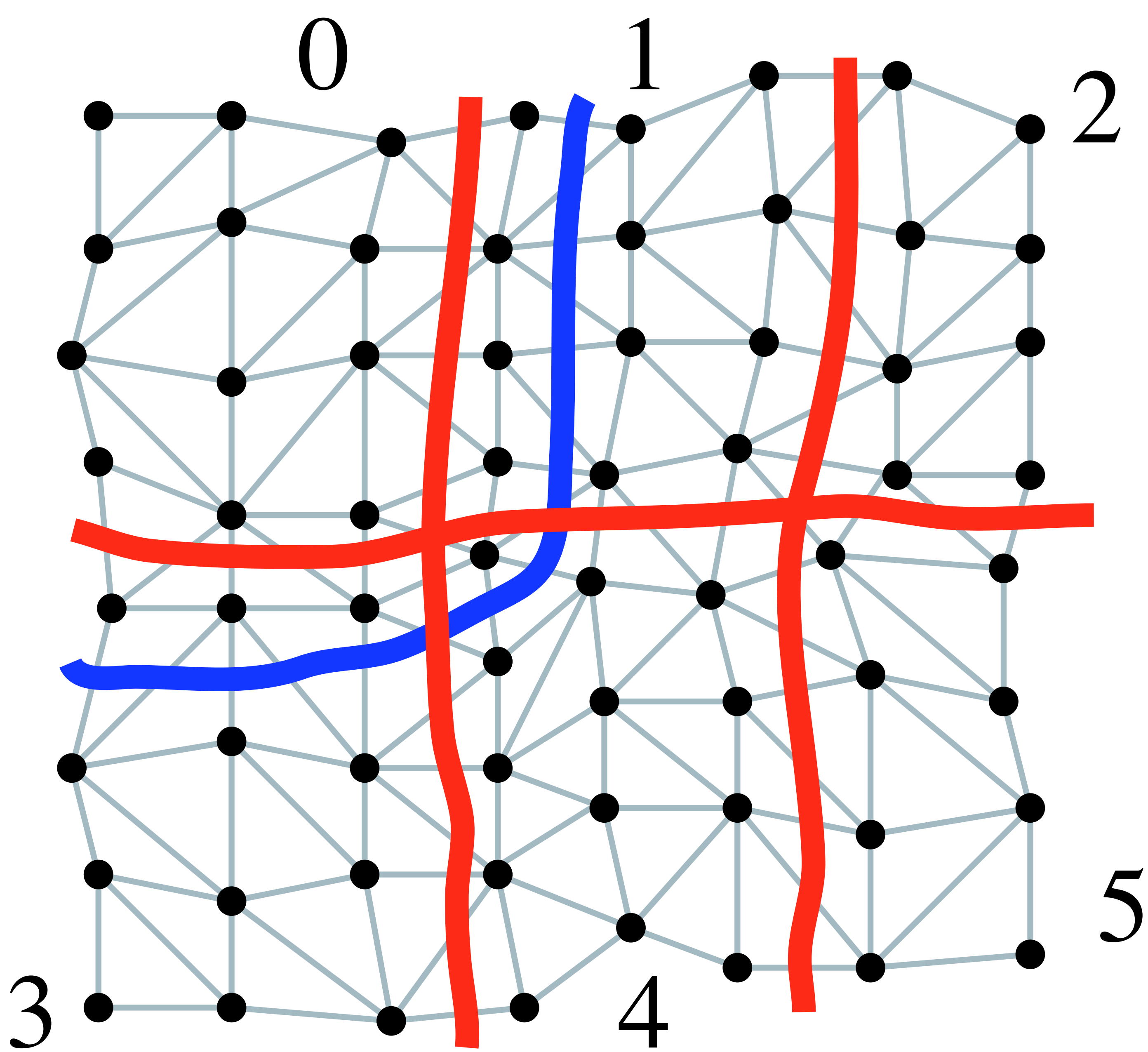


Parallelize with Full Sparse Tiling



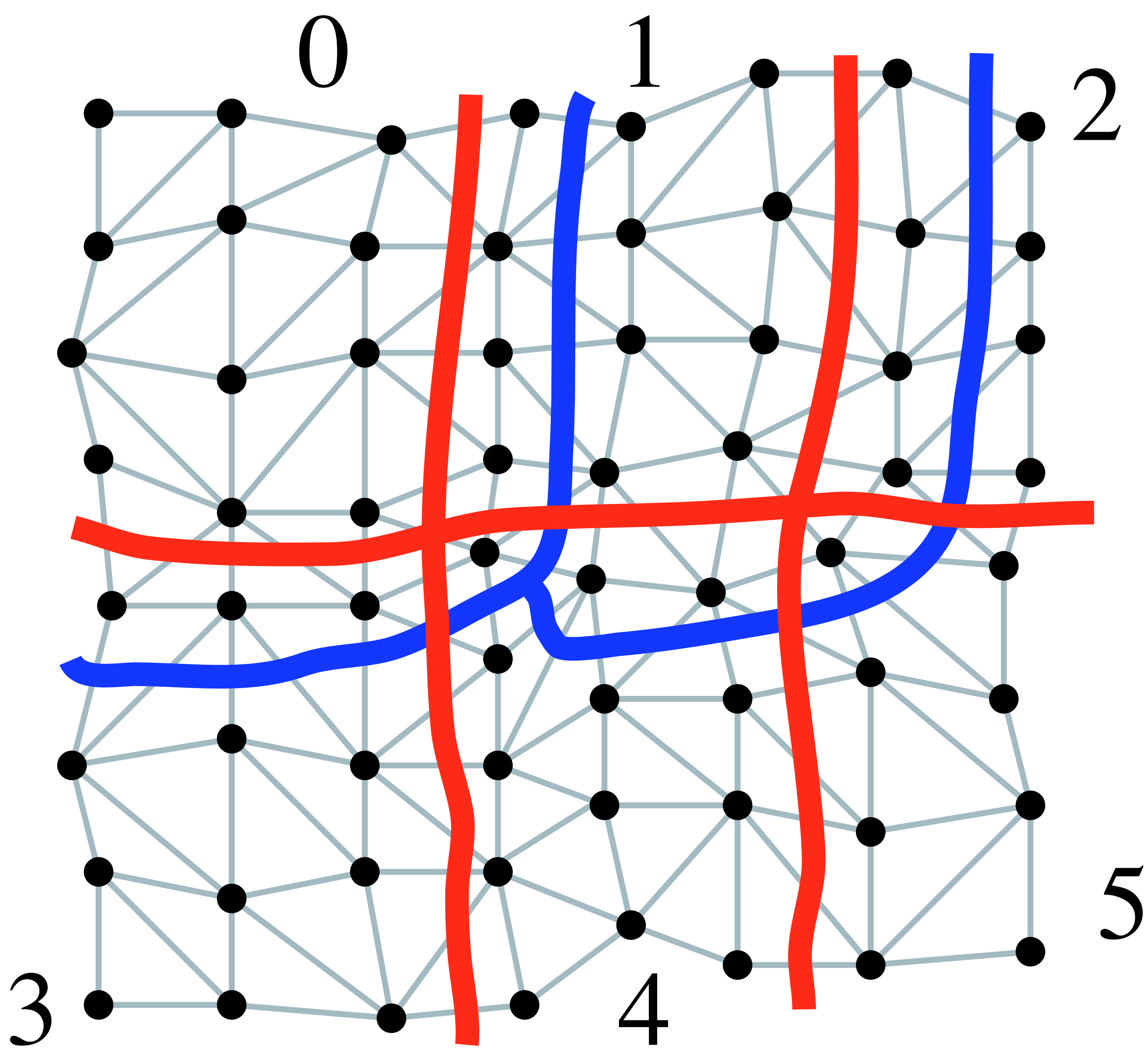
- Create seed partitioning at *iter* = 2

Parallelize with Full Sparse Tiling



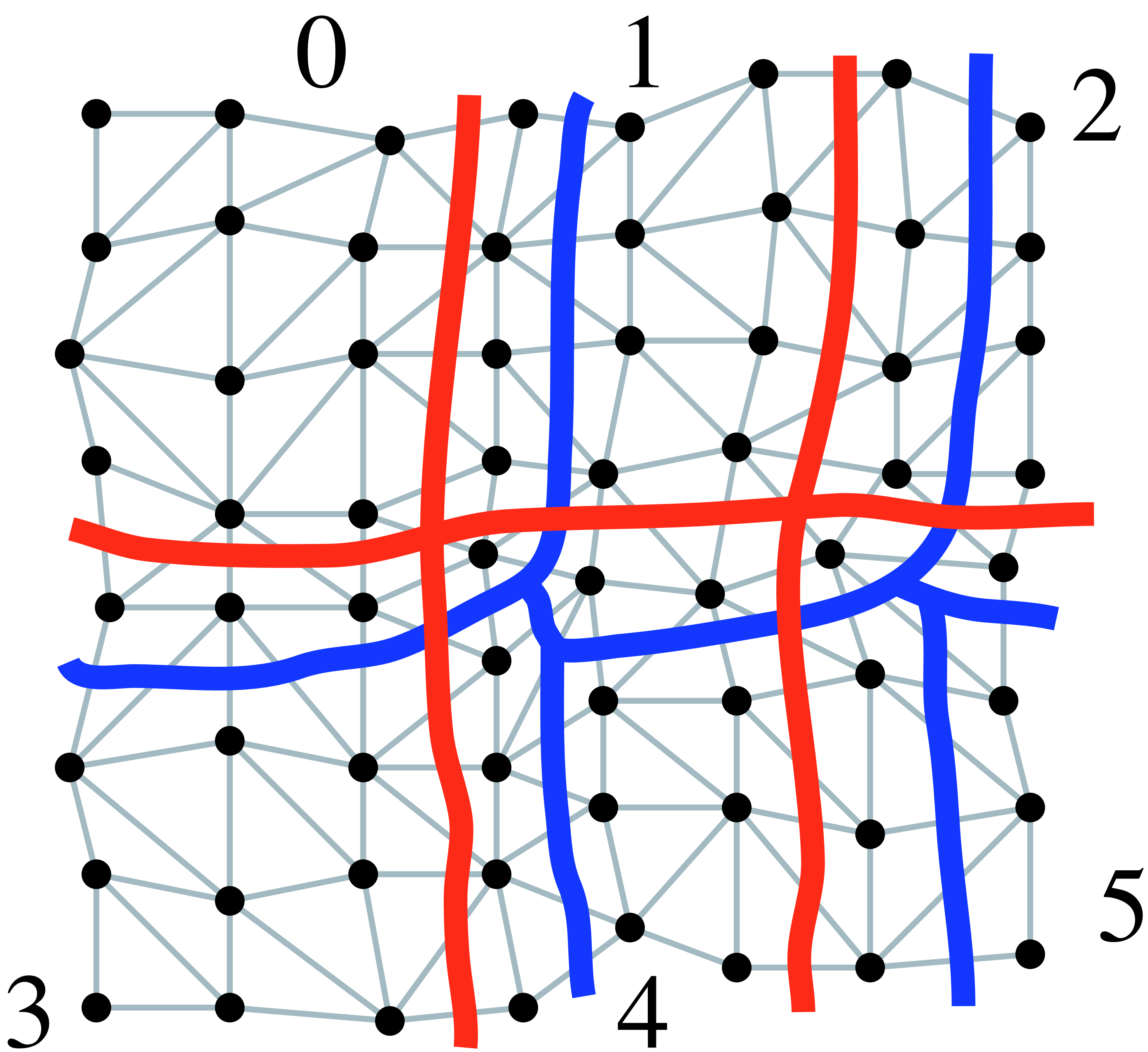
- Create seed partitioning at *iter* = 2
- Grow tiles to *iter* = 1 based on ordering of partitions

Parallelize with Full Sparse Tiling



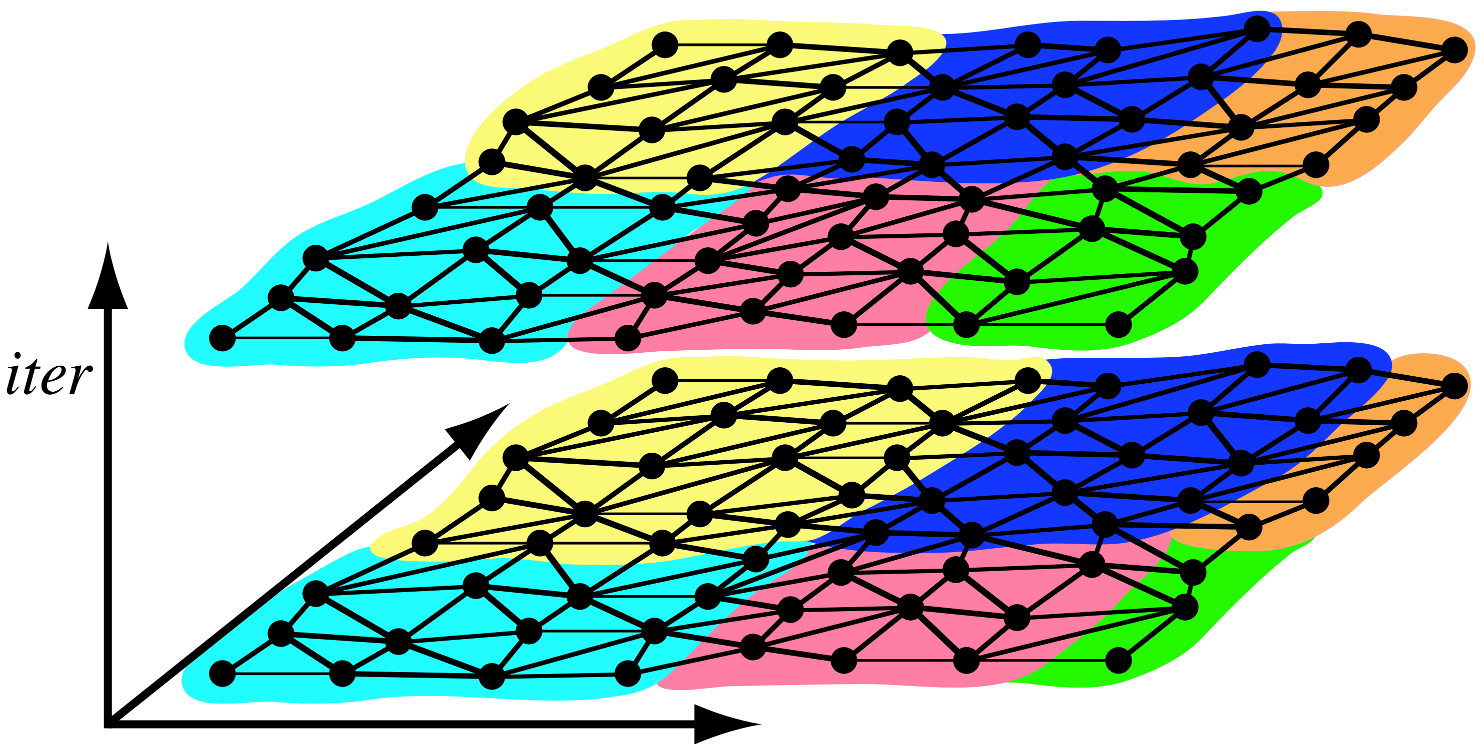
- Create seed partitioning at *iter* = 2
- Grow tiles to *iter* = 1 based on ordering of partitions

Parallelize with Full Sparse Tiling



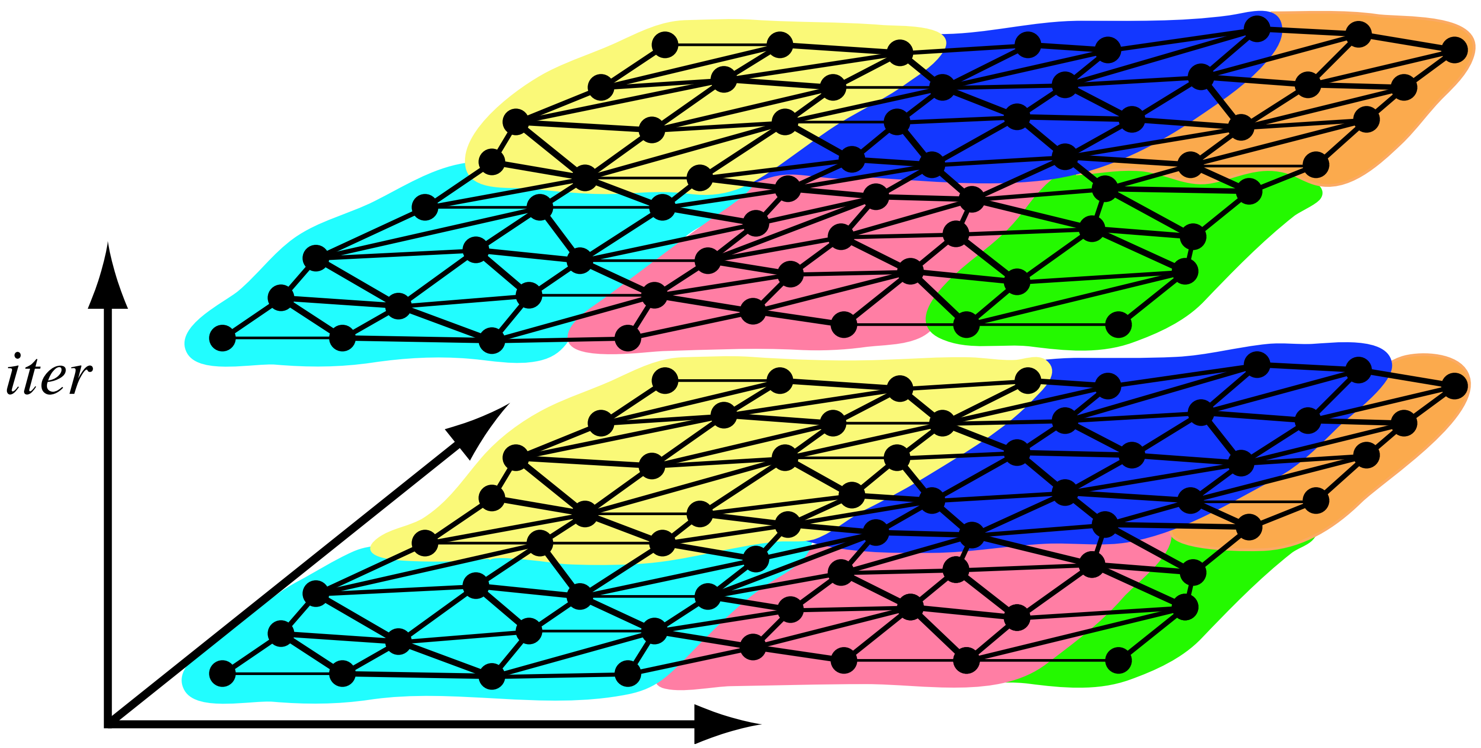
- Create seed partitioning at *iter* = 2
- Grow tiles to *iter* = 1 based on ordering of partitions

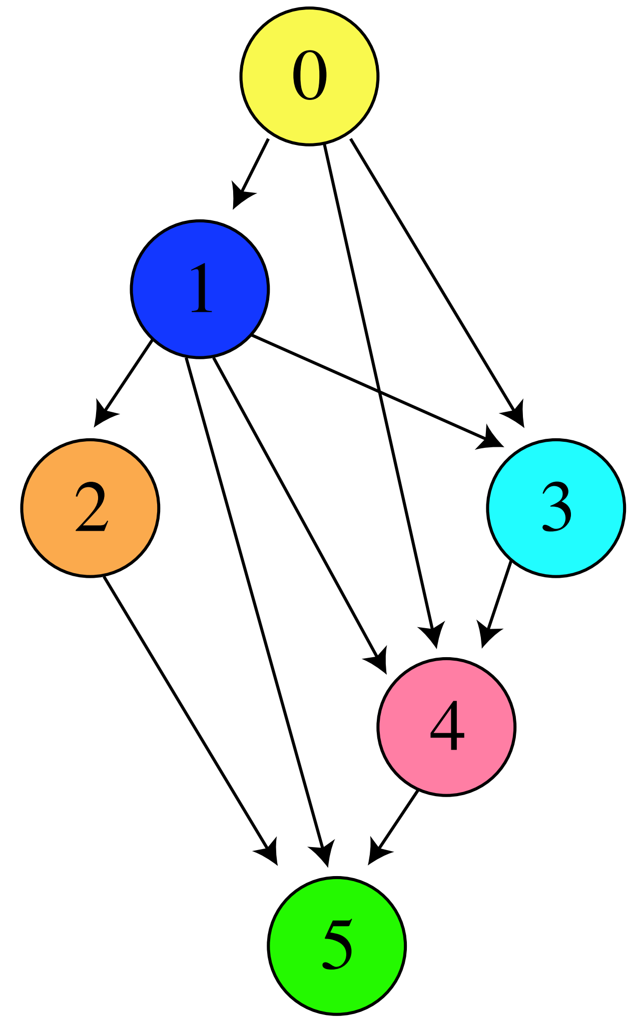
Locality in Full Sparse Tiled Jacobi



- **Intra-iteration** locality achieved by consecutively ordering matrix graph nodes within a seed partition
- **Inter-iteration** locality achieved with tile-by-tile execution

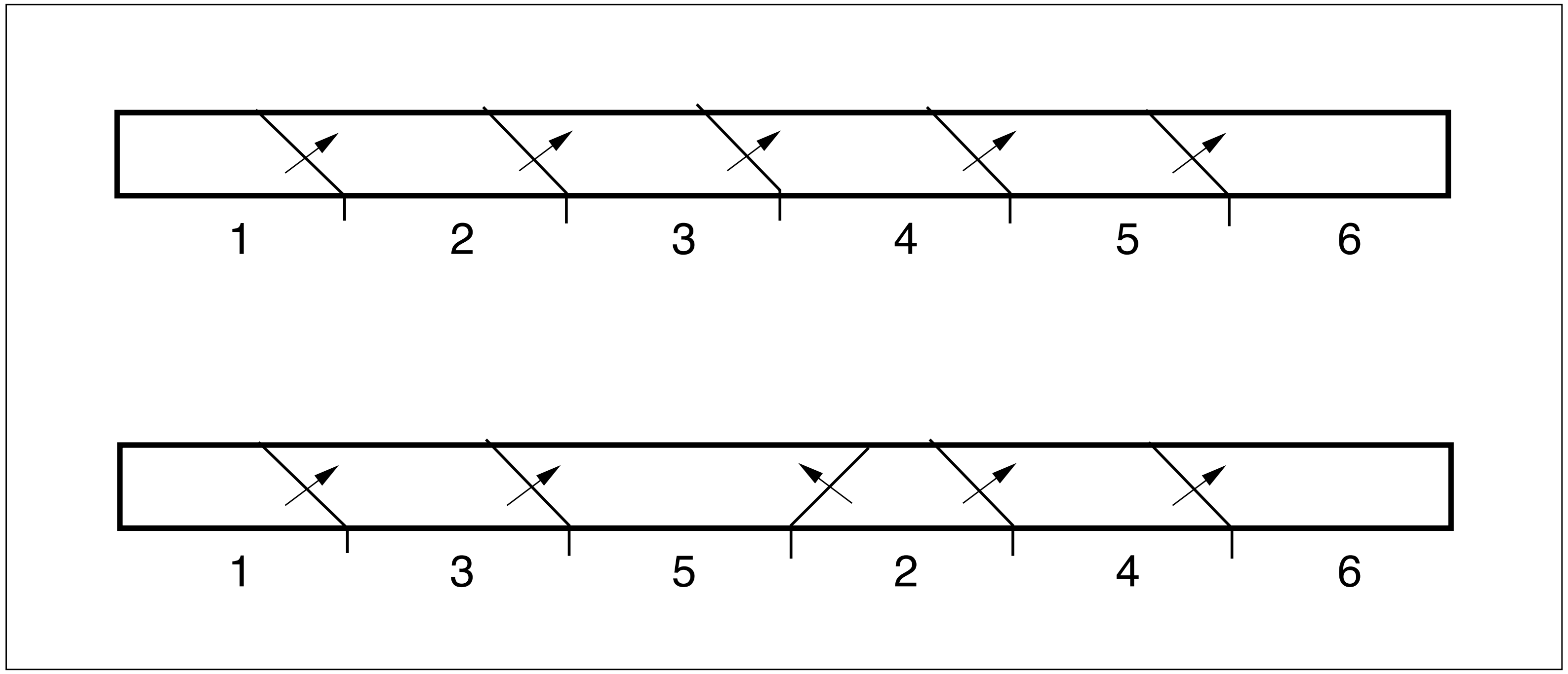
Parallelism in Full Sparse Tiled Jacobi





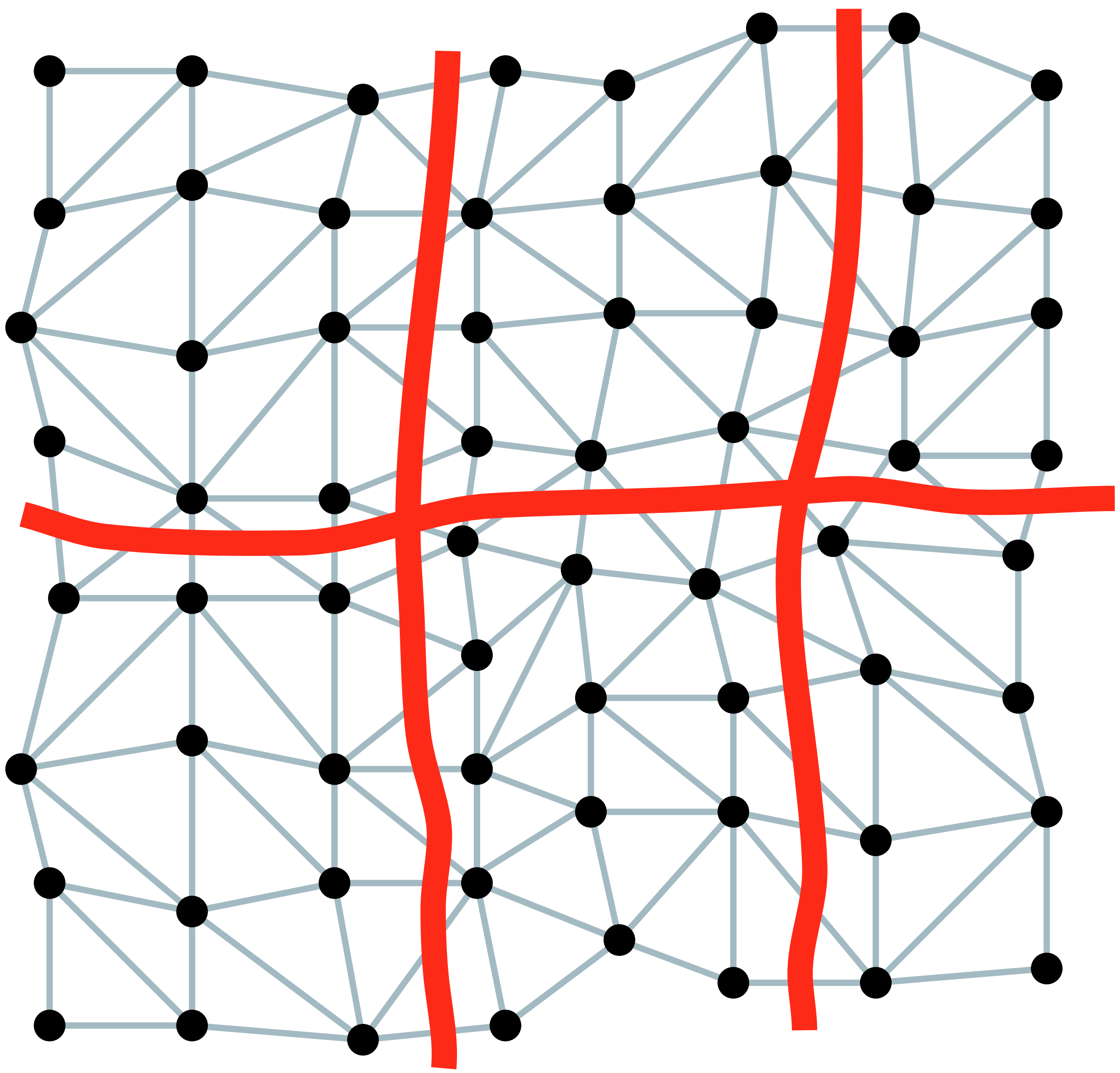
Average parallelism = (# tiles) / (# tiles in critical path)
= 6 / 5 = 1.2

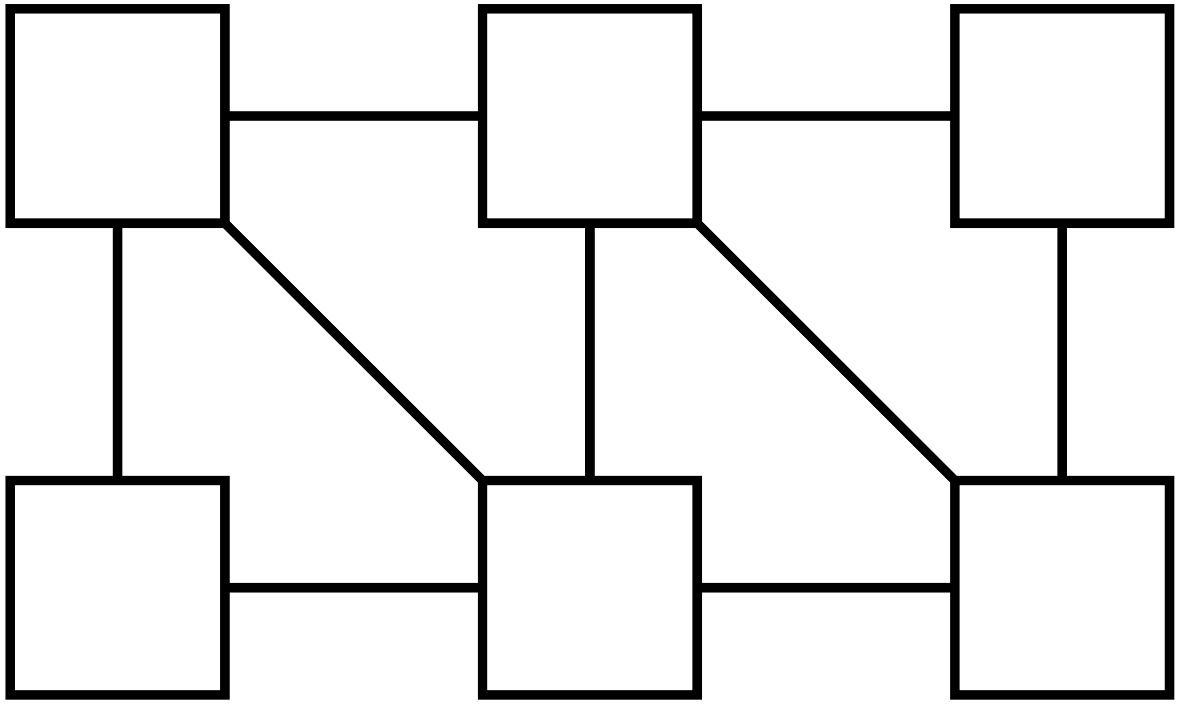
How can we increase parallelism between tiles?



- Order that tile growth is performed matters
- Best is to first grown tiles whose seed partitions are not adjacent

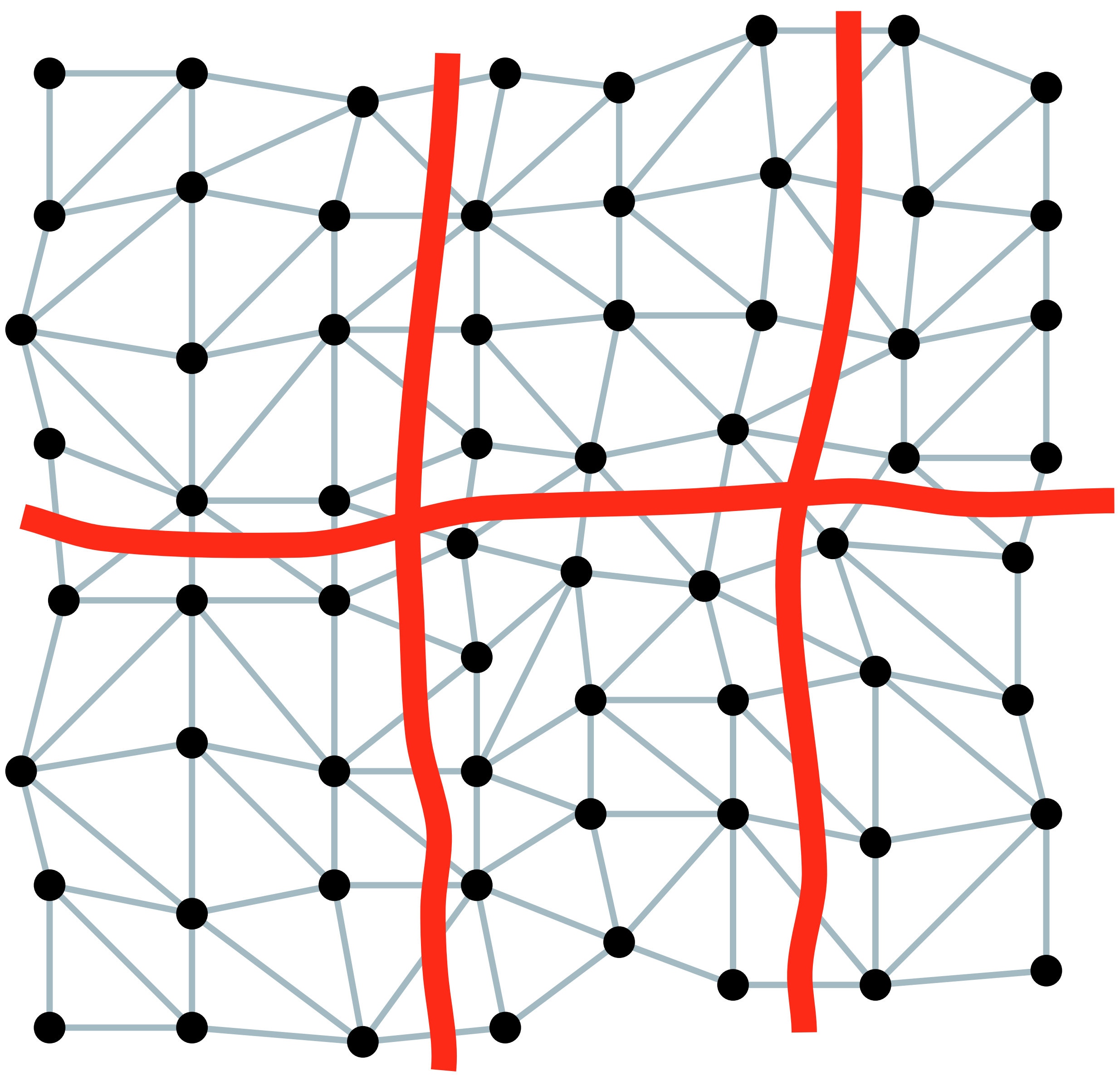
Improving Average Parallelism Using Coloring

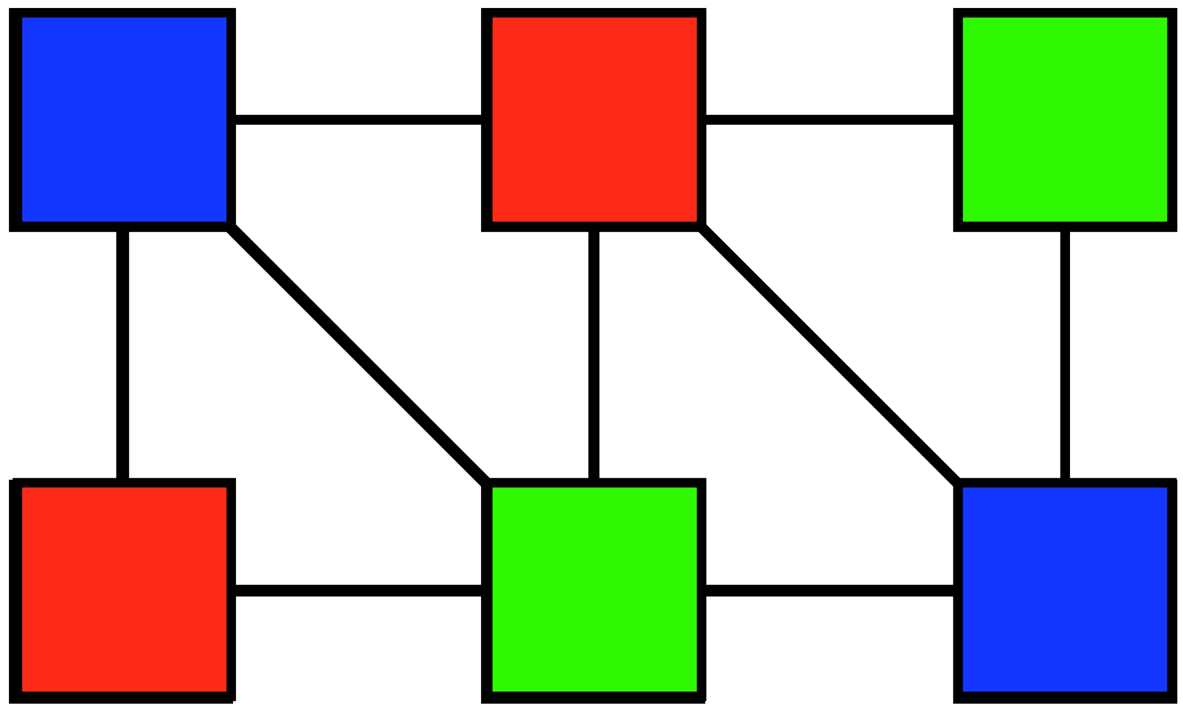




- Create a partition graph

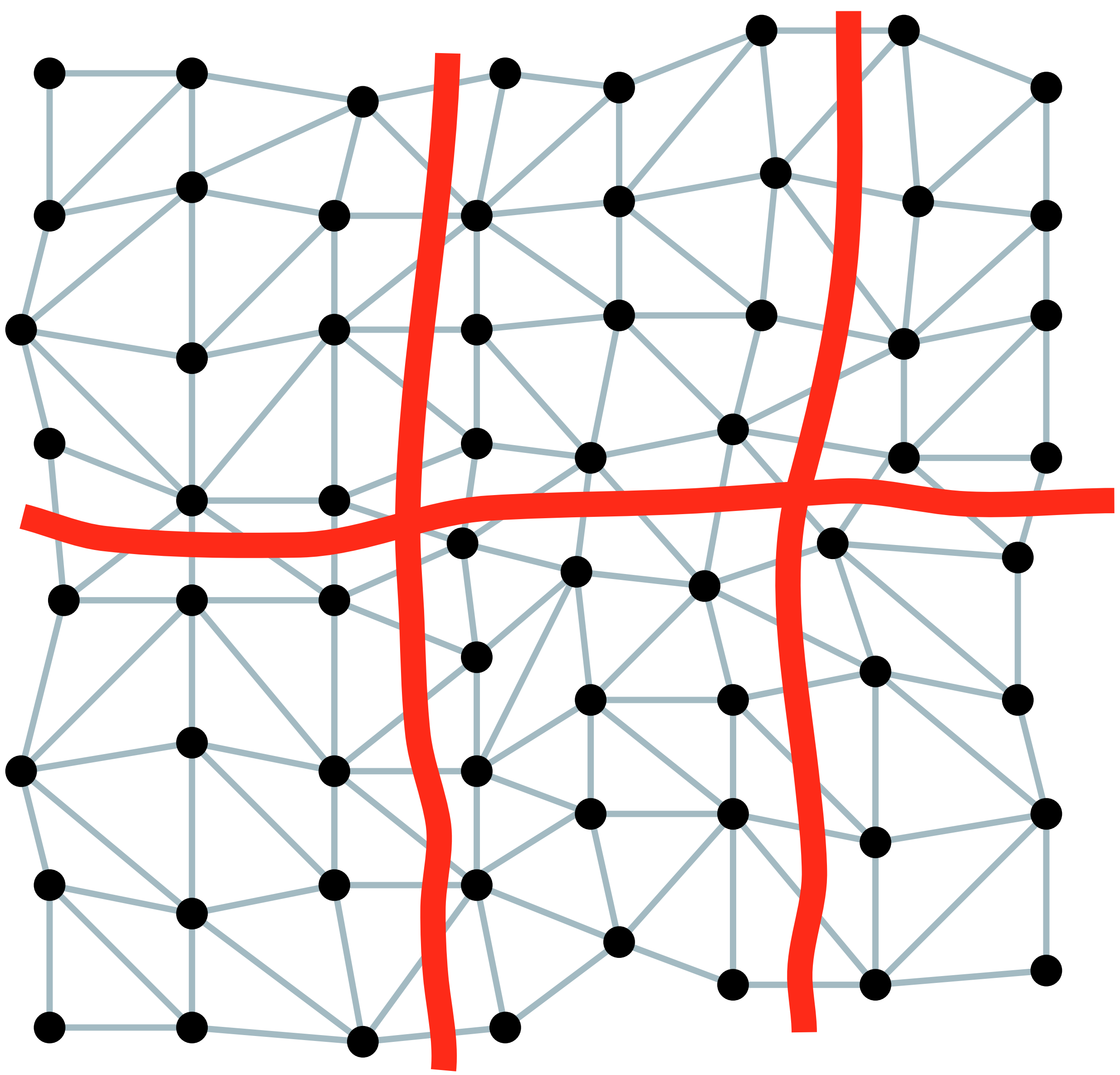
Improving Average Parallelism Using Coloring

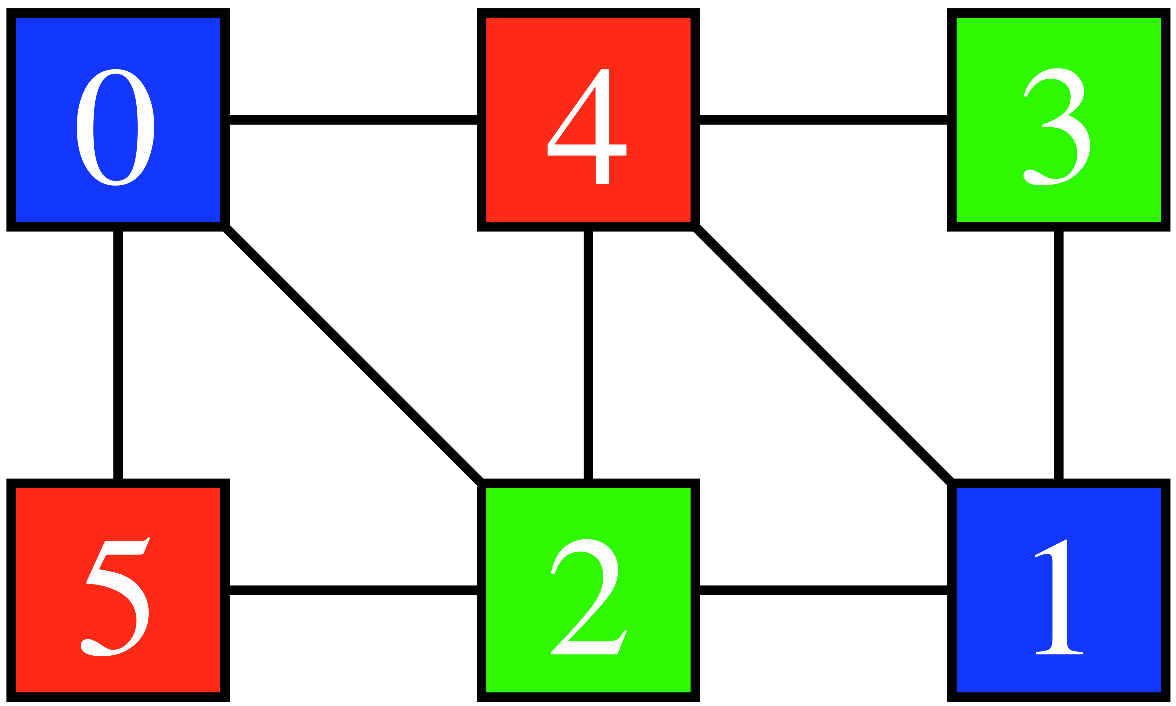




- Create a partition graph
- Color the partition graph

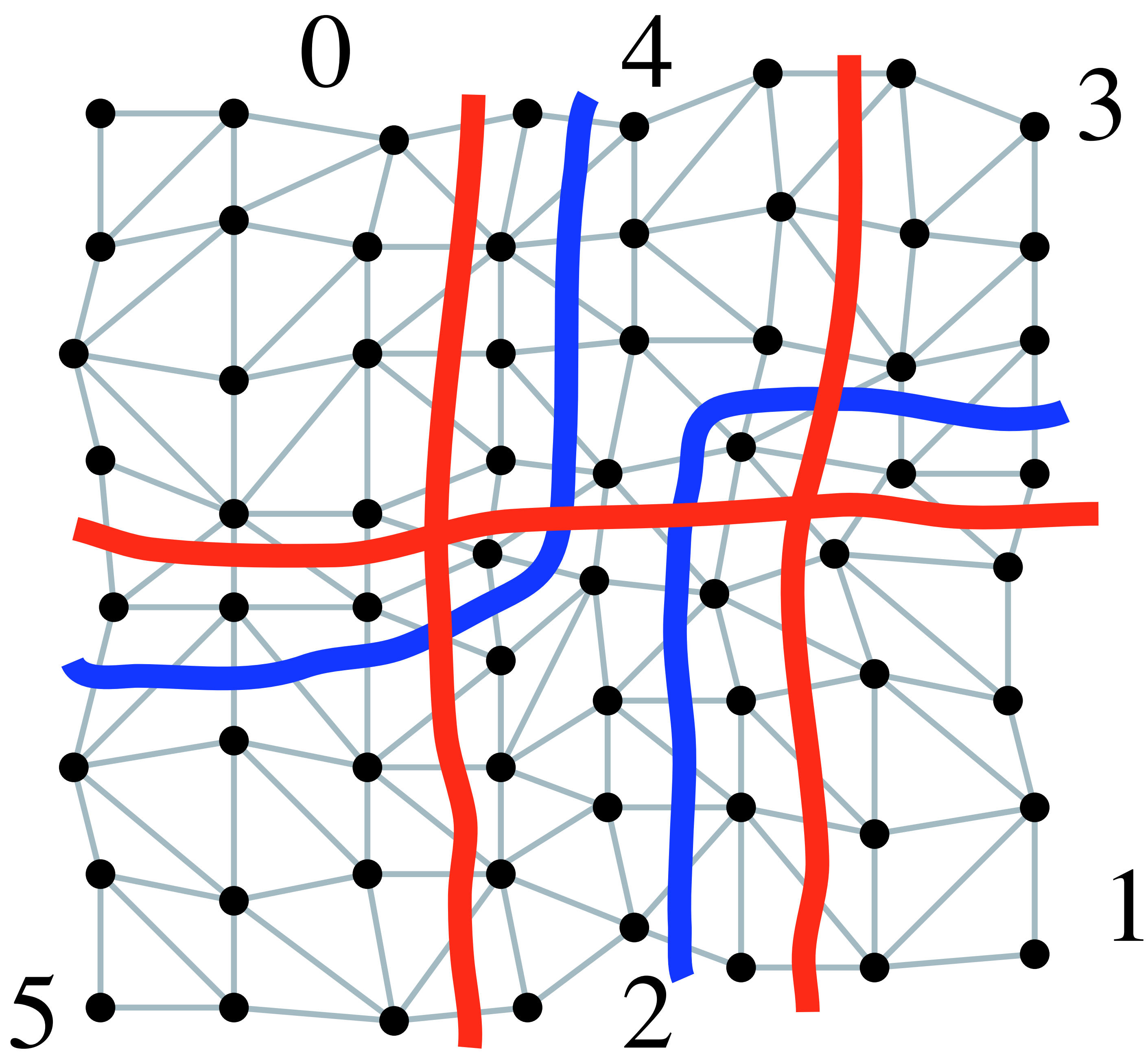
Improving Average Parallelism Using Coloring





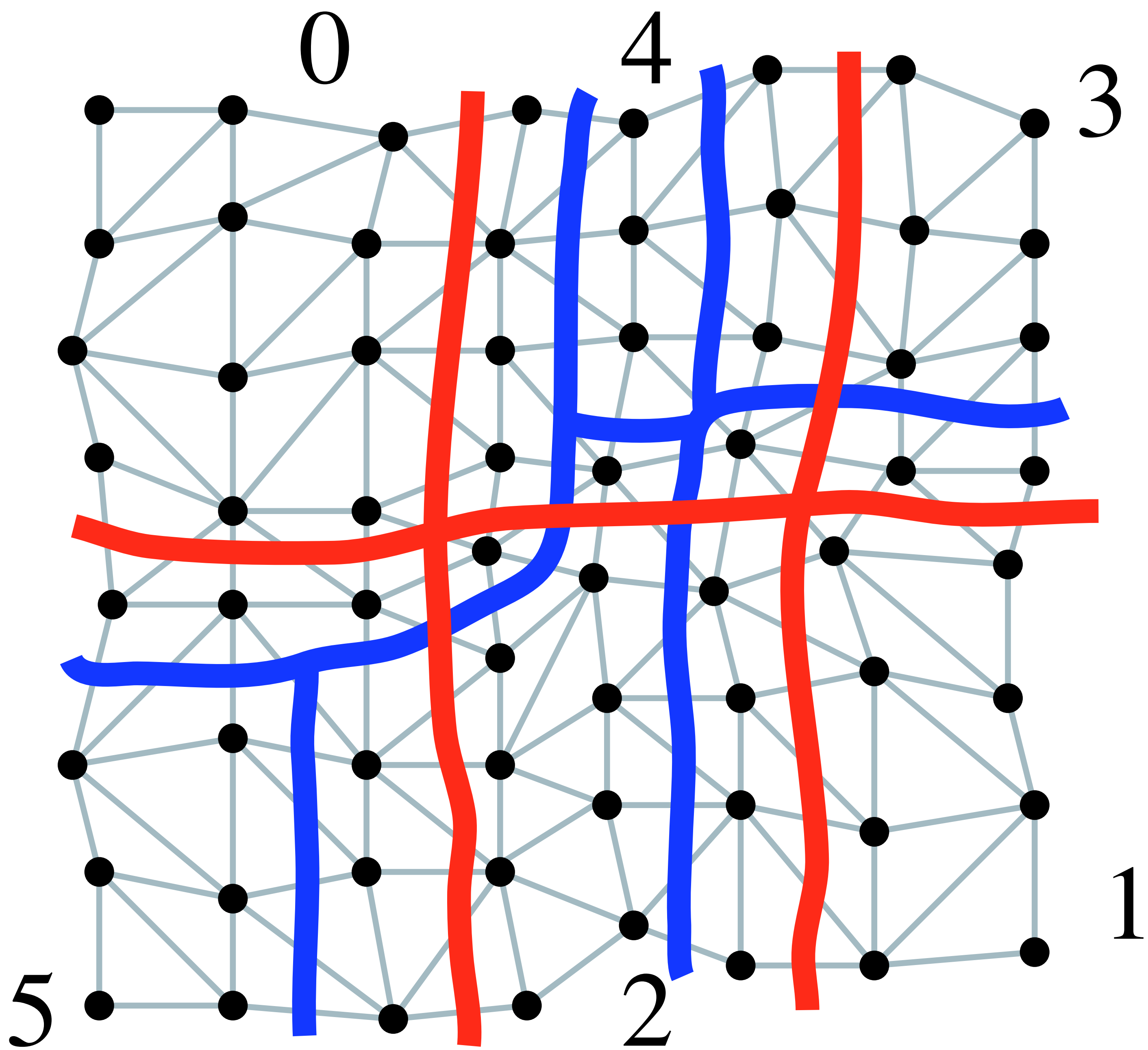
- Create a partition graph
- Color the partition graph
- Renumber partitions consecutively by color

Grow Using New Partition Order



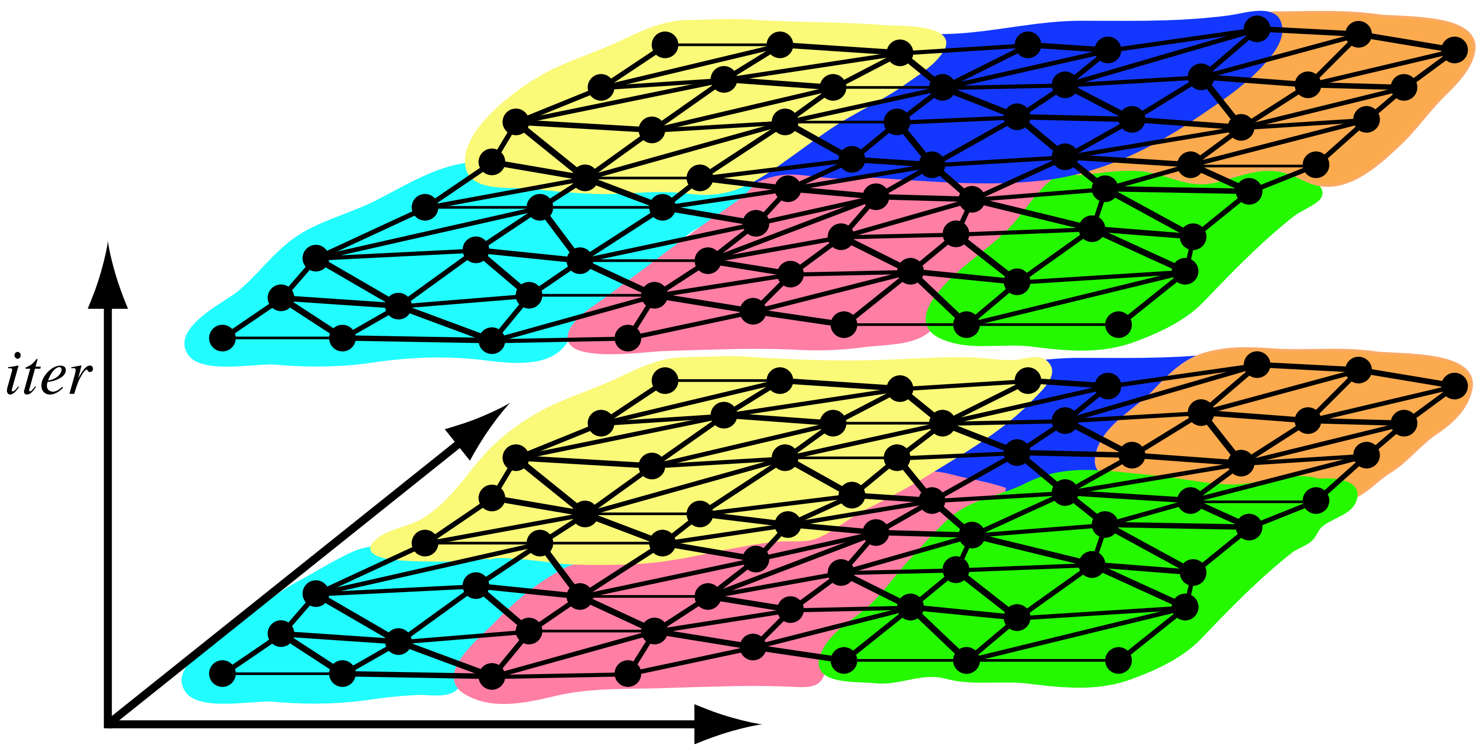
- Renumber the seed partition cells based on coloring
- Grow tiles using new ordering
- Notice that tiles 0 and 1 may be executed in parallel

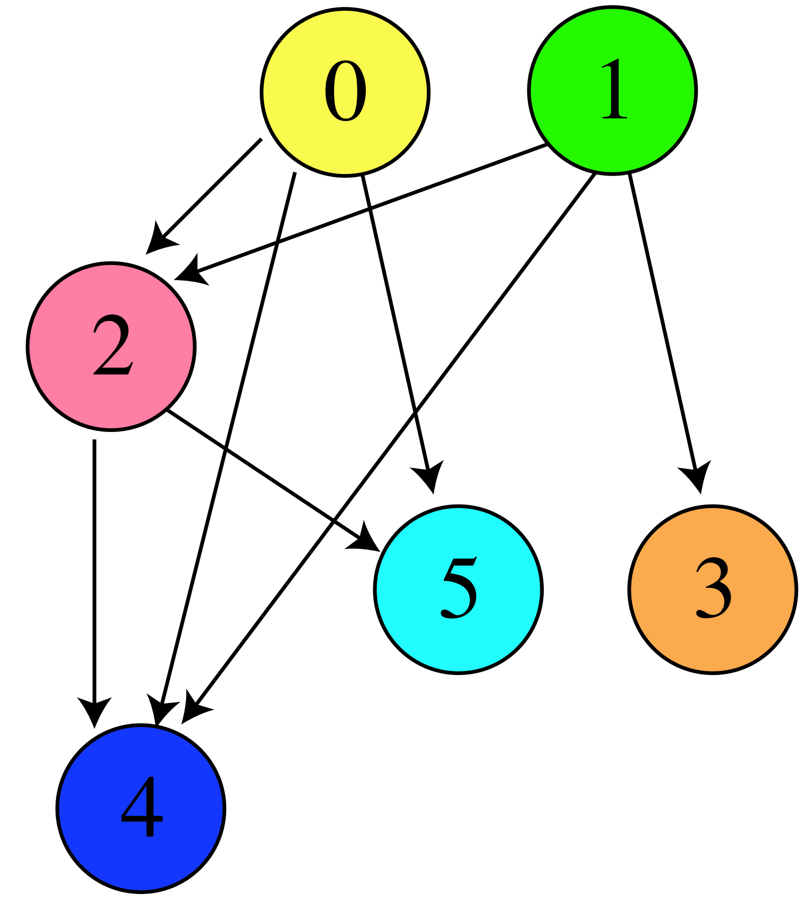
Re-grow Using New Partition Order



- Renumber the seed partition cells based on coloring
- Grow tiles using new ordering
- Notice that tiles 0 and 1 may be executed in parallel
- Tiles 4 and 5 may also be executed in parallel

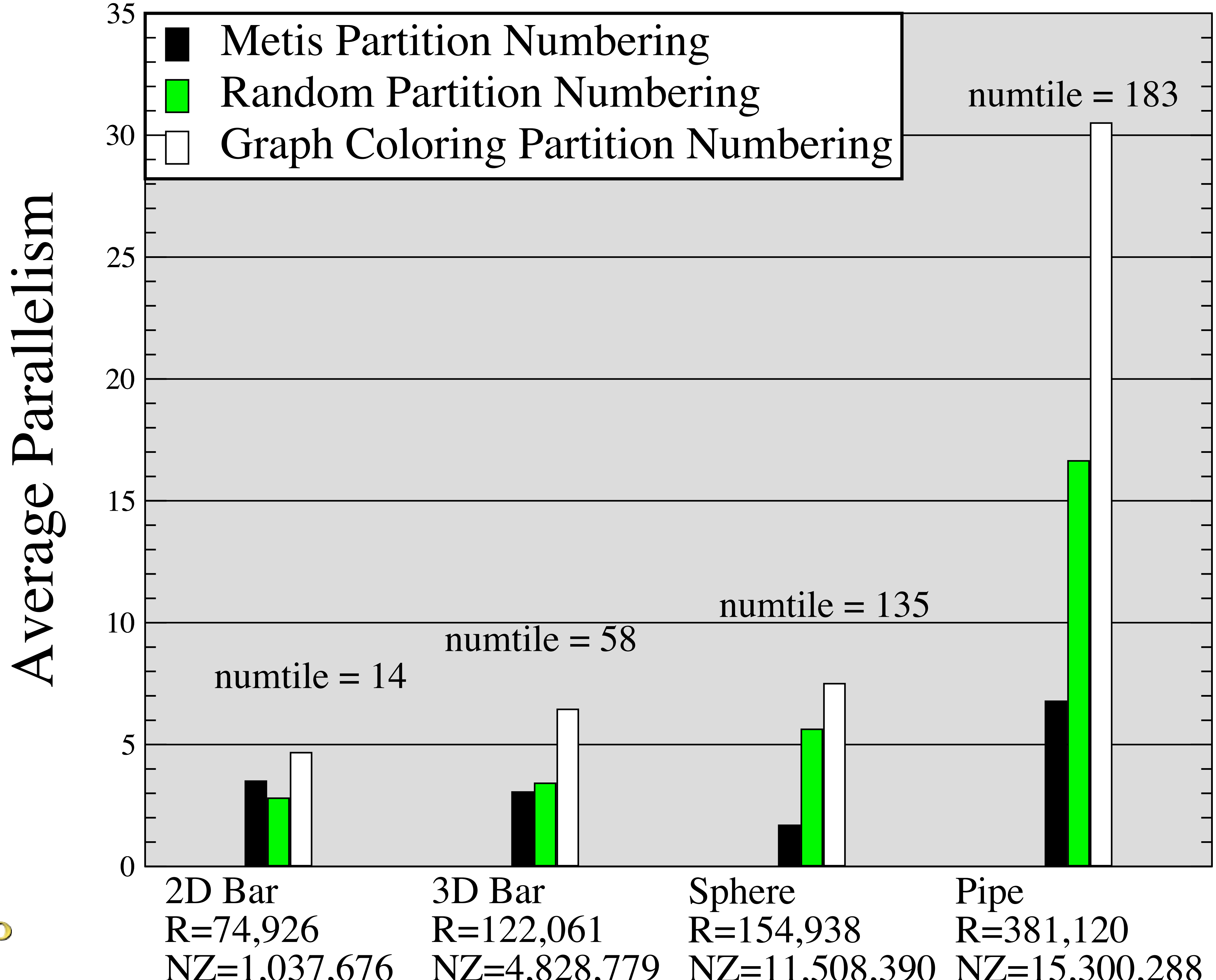
Average Parallelism is Improved





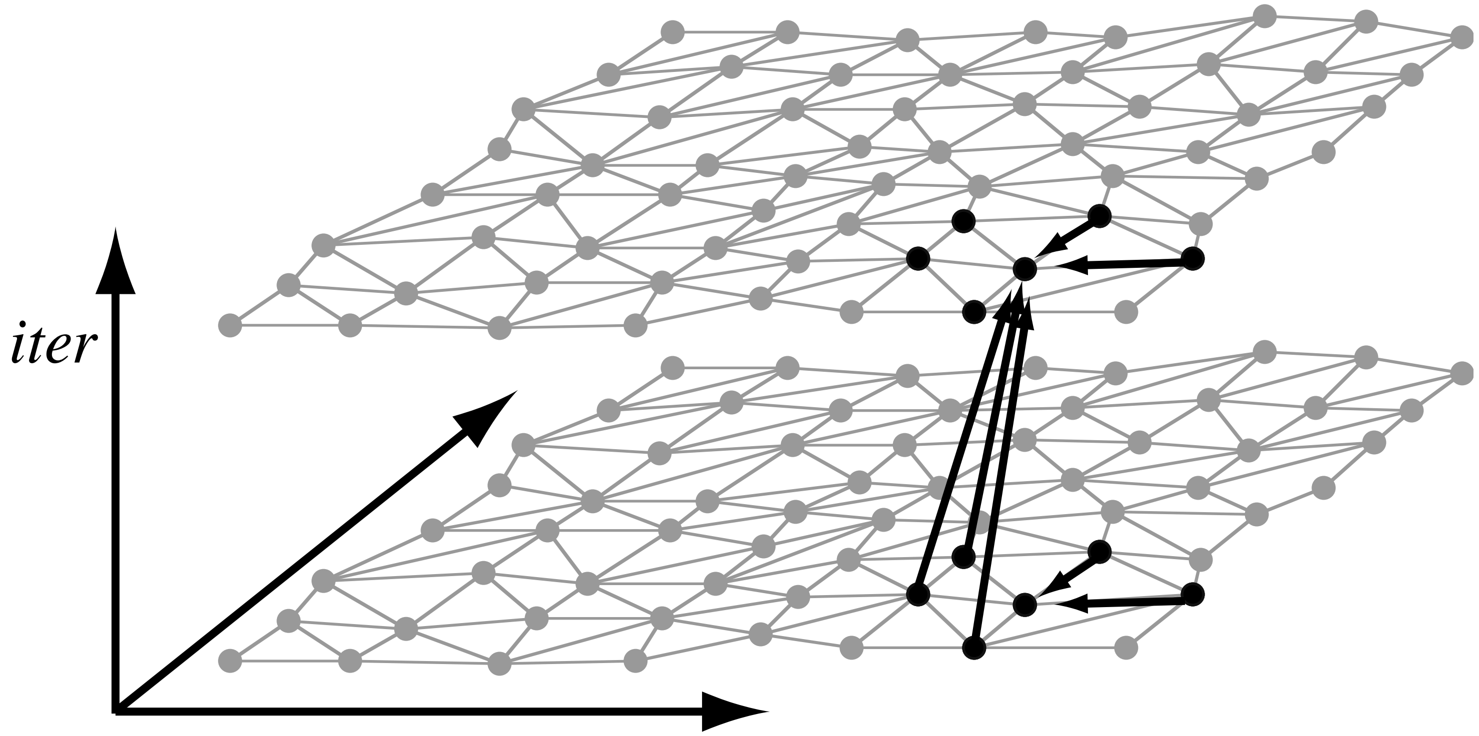
Average parallelism = (# tiles) / (# tiles in critical path)
= 6 / 3 = 2

Improvement with Real Matrix Graphs



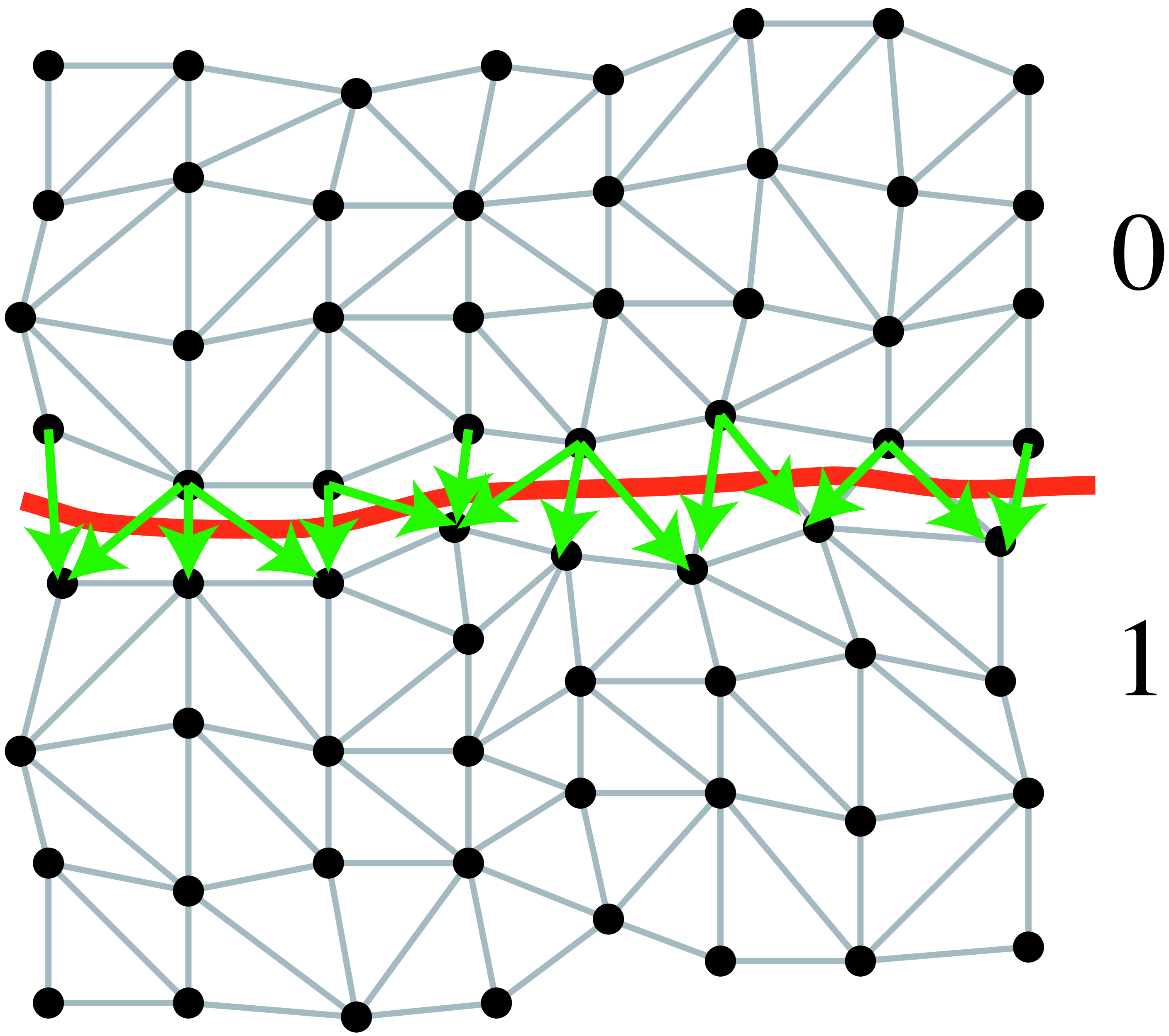
Colorado State University
Knowledge to Go Places

Gauss-Seidel



- Loop carried dependences within convergence iteration as well as between them
- Dependences depend on the ordering of the nodes
- Nodes can be reordered apriori

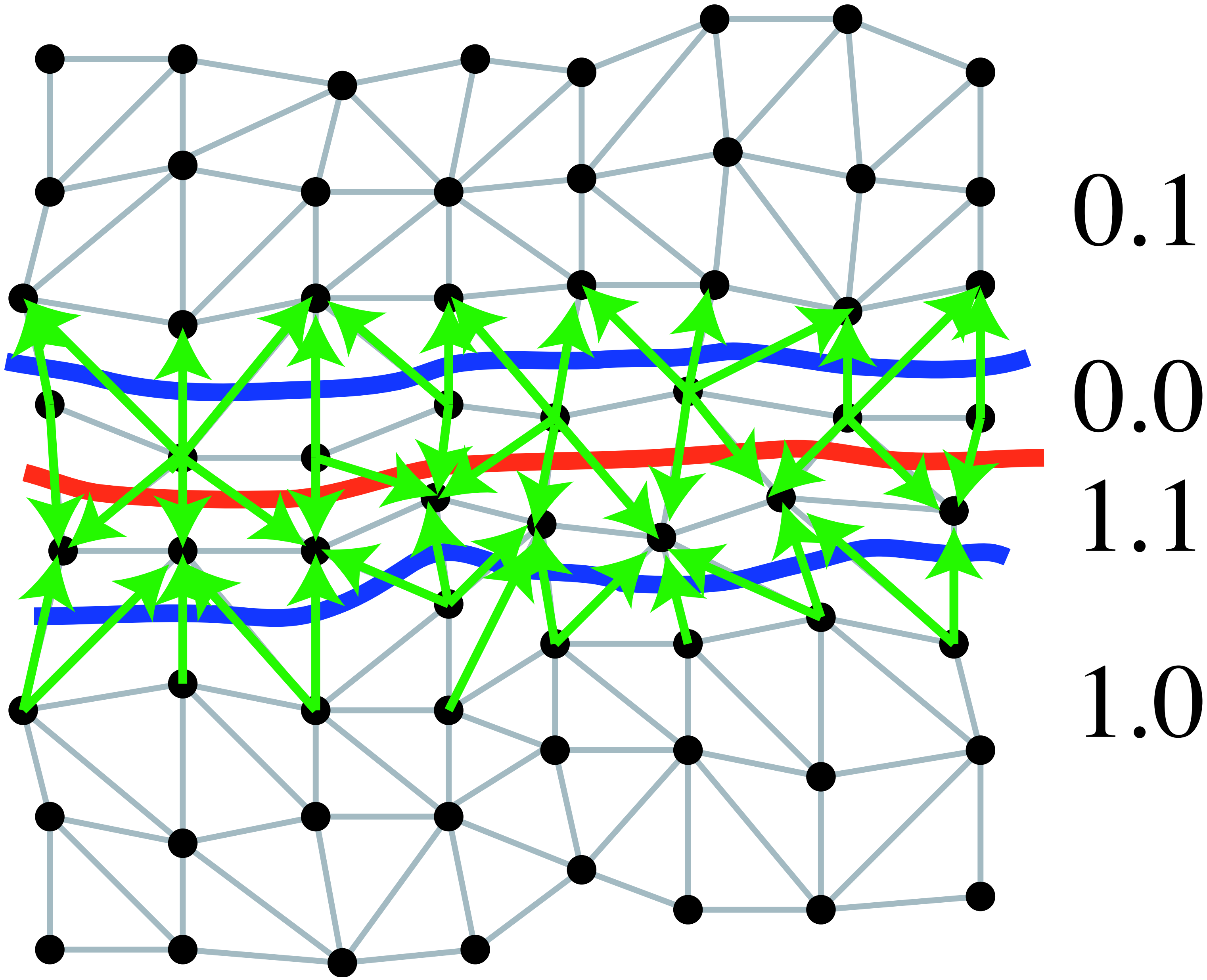
Owner Computes Method
Nodal Gauss-Seidel [Adams 2001]



- Renumbers nodes (rows/columns in sparse matrix)
- Make data dependences between main partitions consistent

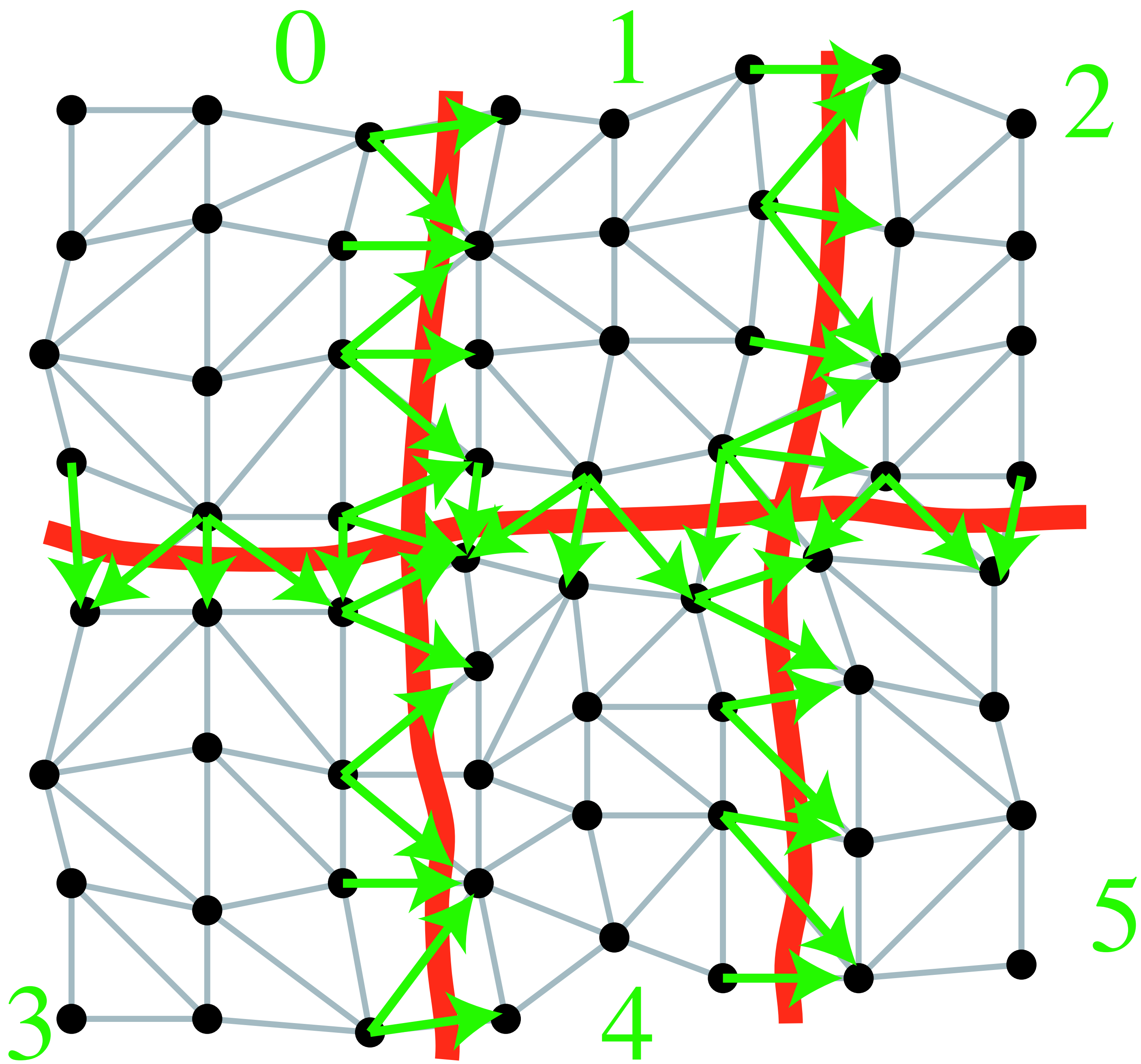
Colorado State University
Knowledge to Go Places

Owner Computes Method
Nodal Gauss-Seidel [Adams 2001]



- Renumbers nodes (rows/columns in sparse matrix)
- Make data dependences between main partitions consistent
- Subpartition for better parallelism

Full Sparse Tiled Gauss-Seidel



- Tile growth creates and maintains a partial ordering between nodes in matrix graph
- Reorder nodes so that partial ordering is maintained

Experimental Methodology

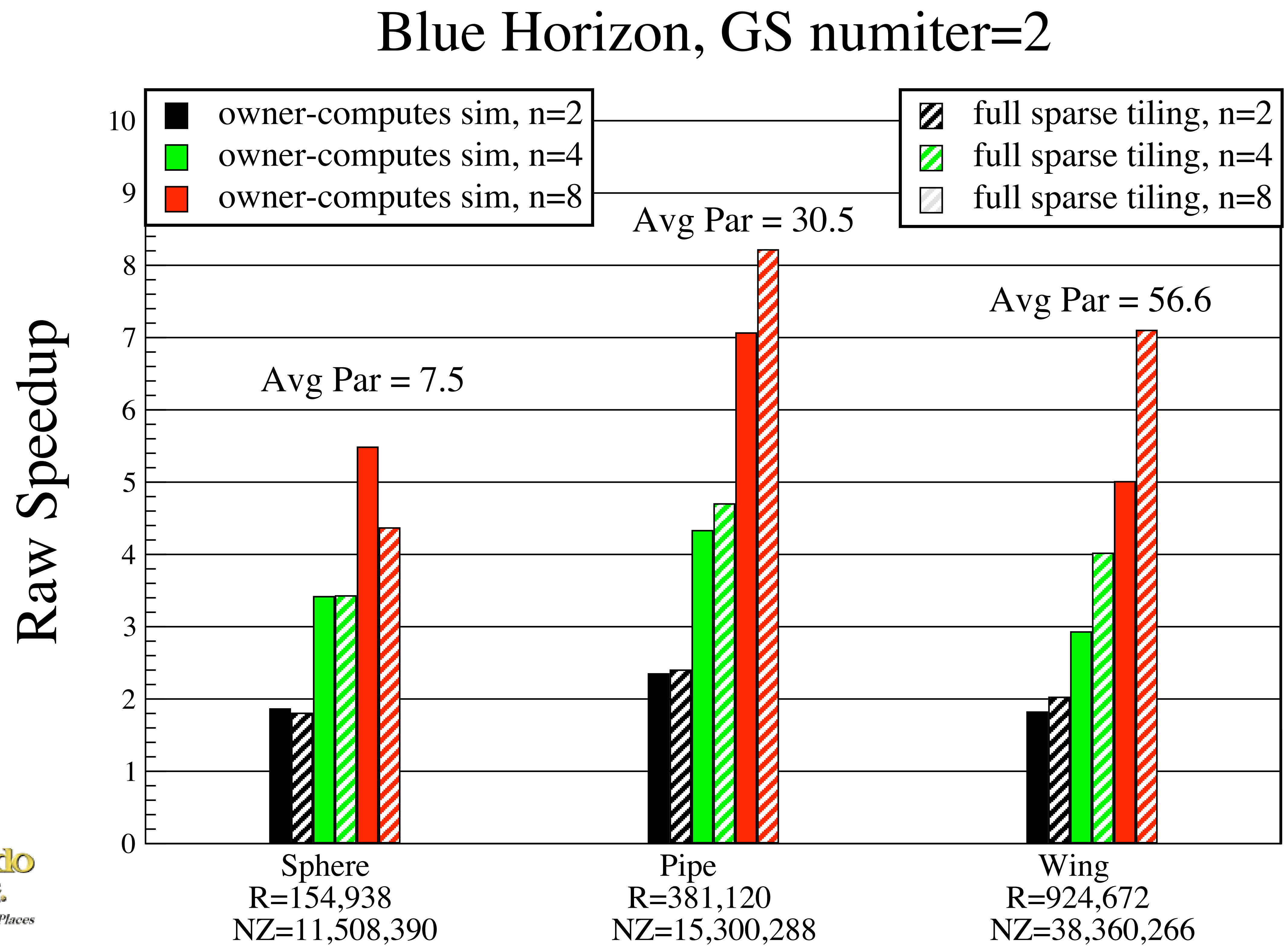
- **Baseline is Gauss-Seidel implemented CSR and uses provided ordering**
- **Owner computes implementation**
  - Partition matrix graph into equal-sized cells for each processor
  - Sub-partition and reorder on each processor for intra-iteration locality
  - Violate intra-iteration dependences for an idealistic parallel efficiency
- **Full sparse tiling implementation**

Overhead of Full Sparse Tiling

Blue Horizon, Gauss-Seidel with numiter=2, Parallel Reschedule							
		Savings/Execution (sec)			Break Even		
Input Matrix	Overhead(sec)	n=2	n=4	n=8	n=2	n=4	n=8
Matrix9	2.03	0.02	0.03	0.04	90	63	57
Matrix12	13.69	0.14	0.20	0.21	96	71	66
Sphere150K	31.41	0.17	0.26	0.29	191	120	110
PipeOT15mill	48.28	0.39	0.52	0.58	125	93	83
Wing903K	116.86	0.65	0.96	1.10	182	122	107

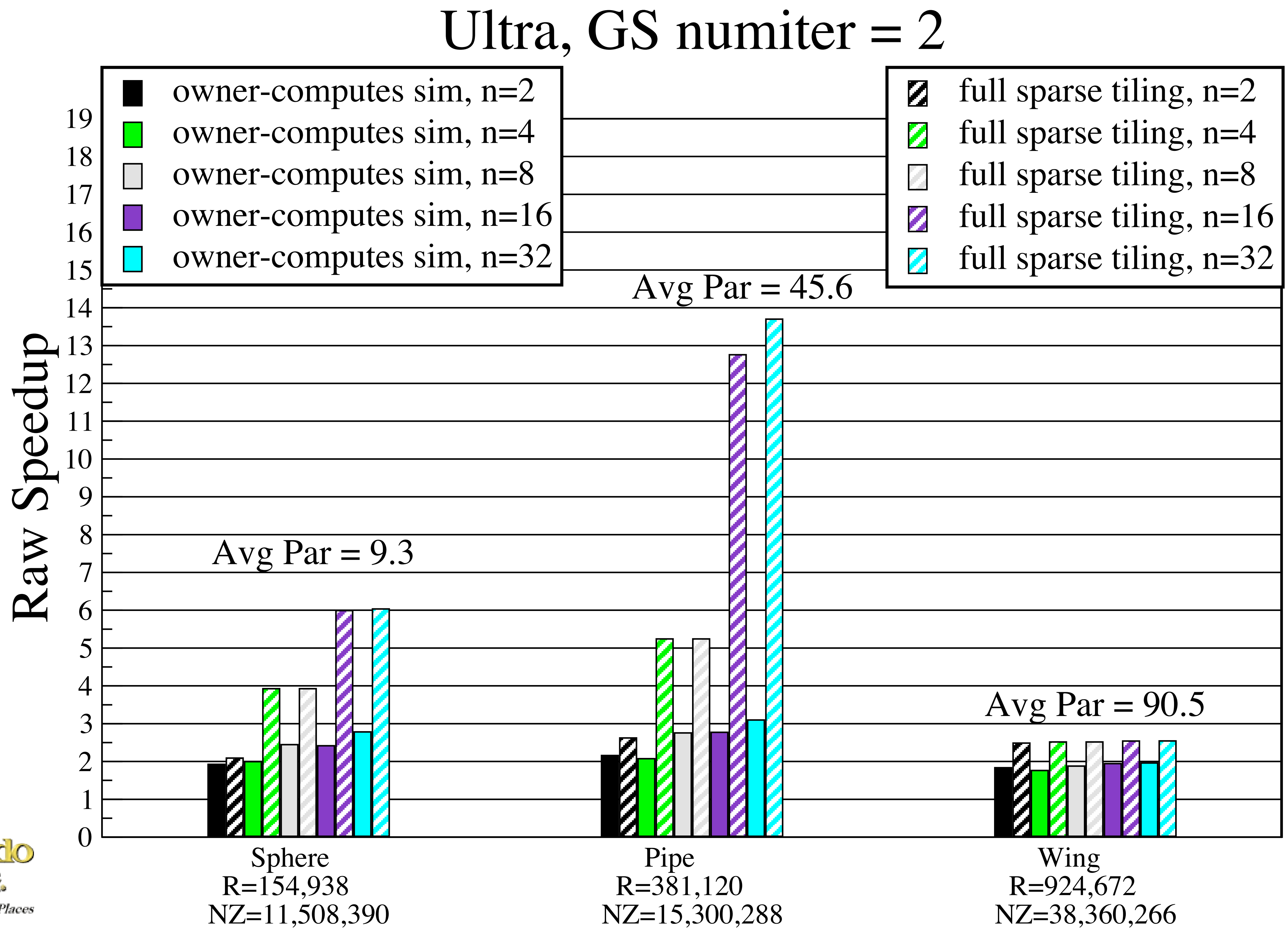
Percentage Time of Overhead		
Input Matrix	Partitioning	Data Remapping
Matrix9	79%	14%
Matrix12	72%	13%
Sphere150K	68%	17%
PipeOT15mill	81%	10%
Wing903K	84%	10%

Experimental Results on IBM Blue Horizon



Colorado State University
Knowledge to Go Places

Experimental Results on Sun Ultra



Colorado State University
Knowledge to Go Places

Conclusions

- Full sparse tiling exploits parallelism, intra-iteration data reuse, and inter-iteration data reuse
- Coloring the partition graph to number seed partitions improves the average parallelism in the tile dependence graph
- On shared memory processors, full sparse tiling can outperform owner-computes methods when enough parallelism is present

Colorado State University
Knowledge to Go Places

Future Work

- Develop distributed memory execution framework for full sparse tiling
- Scaling studies for distributed memory execution
- Apply to other benchmarks and kernels
  - Jacobi
  - Moldyn
  - Molecular dynamics benchmark from Cornell
- Automate generation of inspectors and executors

PETSc - Portable Extensible Toolkit for Scientific Computing

- Parallel Partial Differential Equations (PDE) solvers using Object Oriented Programming (OOP)
- Used in many applications: CFD, Optimization, Biology, Finite Element Analysis, etc.
- Software architecture contains many aspects
  - Usage levels: beginner, intermediate, etc.
  - Many Krlov methods, preconditioners, and sparse matrix formats
  - Profiling, logging, options, etc.

Sparse Tiling Extension to PETSc (STPetsc)

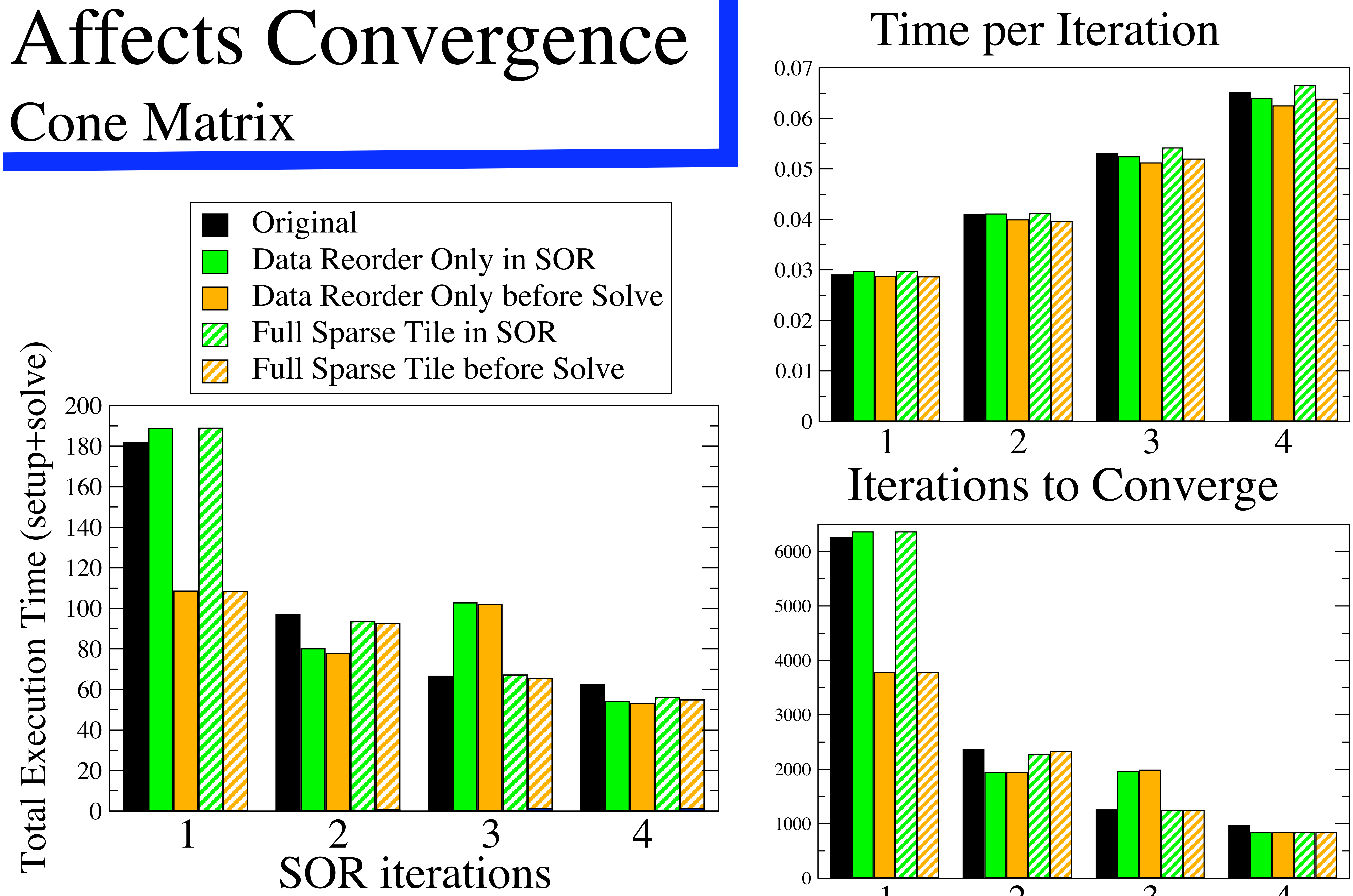
- How should run-time reordering transformations be incorporated in existing domain-specific libraries?
- Interface
  - Beginner: Call MatUseST_SeqAIJ(A)
  - Intermediate: MatSparseTilingCreate(), etc.
- Implementation
  - Sparse tiling ***inspector*** occurs upon first call to SOR or before SLESSolve
  - SOR for SeqAIJ (CSR) matrix format is transformed into a sparse tiling ***executor***

Cone Matrix on Pentium 4 2GHz

(N=22,032, NZ=1,433,068)

Data Reordering Affects Convergence

Cone Matrix

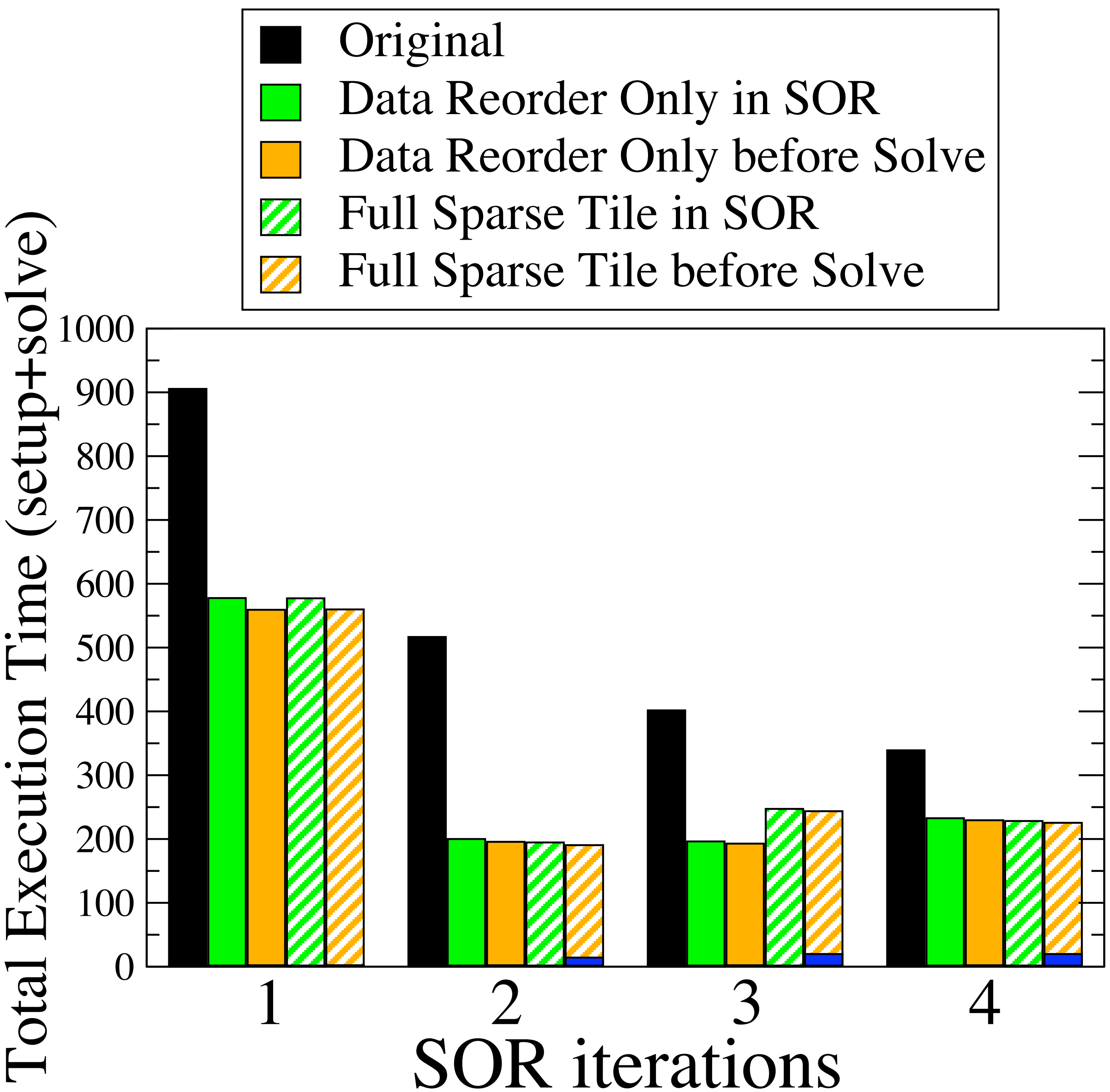


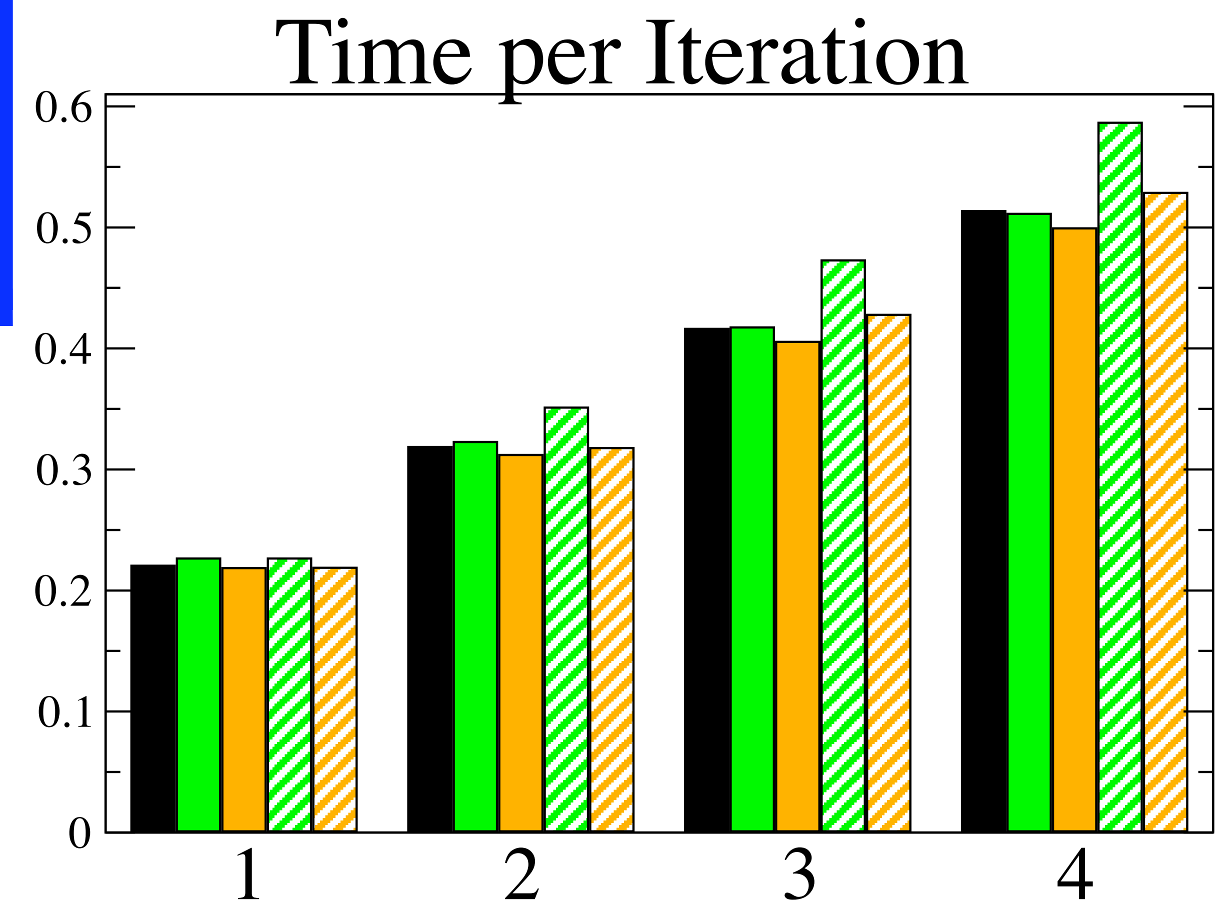
Sphere Matrix on Pentium 4 2GHz

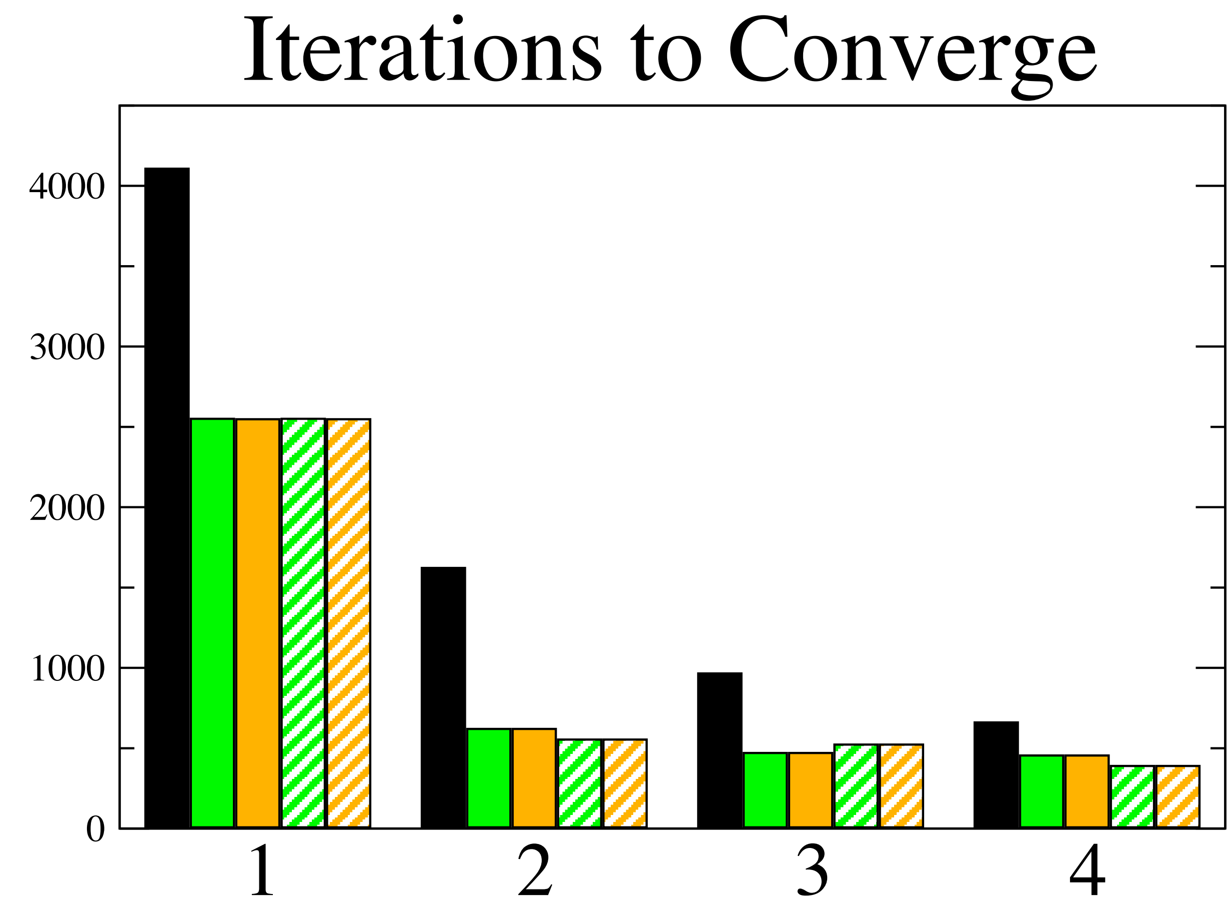
(N=154,938, NZ=11,508,390)

Data Reordering Affects Convergence

Sphere Matrix







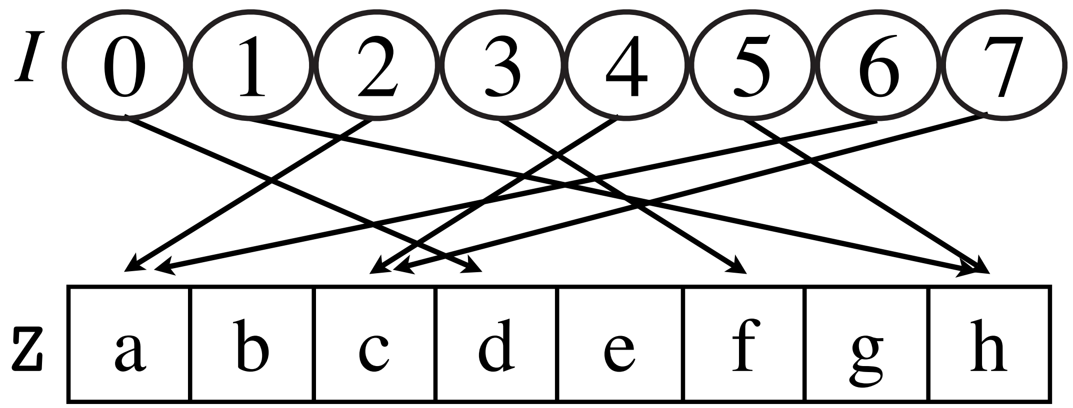
Framework Elements

[Kelly & Pugh 95] [Pugh & Wonnacott 95]

Data Mapping

Compile-time: $M_{I \to Z} = \{[i] \to [x(i)] \mid 0 \le i \le 7\}$

Run-time:

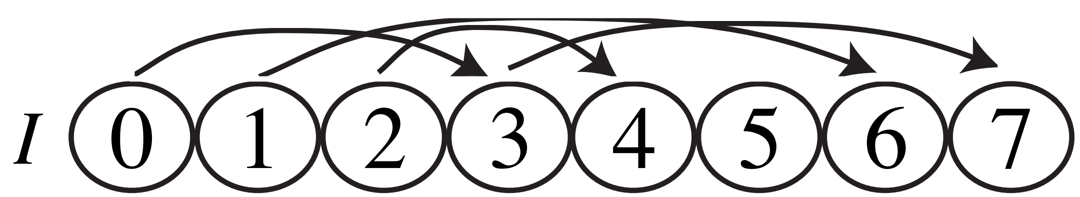


Data Dependences

Compile-time:

$$D_{I \to I} = \{[i] \to [j] \mid (0 \le i < j \le 7) \wedge (i = b(j) \vee j = b(i))\}$$

Run-time:



Key Insights for Composing Run-time Reordering Transformations

- The inspectors ***traverse*** the data mappings and/or the data dependences

- We can ***express*** how the data mappings and data dependences will change

- Subsequent inspectors ***traverse the new*** data mappings and data dependences

Data Mappings and Data Dependences for MOLDYN

```
for ts=1,T

  for i=1,N
    Z[i]= ...
  endfor

  for j=1,M
    ... = Z[l[j]]
    ... = Z[r[j]]
  endfor

endfor
```

Data Mapping for i loop

$$M_{I_0 \to Z_0} = \{[i] \to [i]\}$$

Data Mapping for j loop

$$M_{J_0 \to Z_0} = \{[j] \to [i] \mid (i = l(j)) \vee (i = r(j))\}$$

Data Dependences between i and j loop

$$D_{I_0 \to J_0} = \{[i] \to [j] \mid (i = l(j)) \vee (i = r(j))\}$$

Composing the Inspector at Compile-time for MOLDYN

$M_{J_0 \to Z_0} = \{[j] \to [i] \mid (i = l(j)) \vee (i = r(j))\}$ $R_{Z_0 \to Z_1} = T_{I_0 \to I_1} = \{[i] \to [\sigma(i)]\}$	1. Traverse $M_{J_0 \to Z_0}$ to generate data reordering function σ
$M_{J_0 \to Z_1} = \{[j] \to [\sigma(i)] \mid (i = l(j)) \vee (i = r(j))\}$ $T_{J_0 \to J_1} = \{[j] \to [\delta(j)]\}$	2. Traverse $M_{J_0 \to Z_1}$ to generate iteration reordering function δ
$D_{I_1 \to J_1} = \{[\sigma(i)] \to [\delta(j)] \mid (i = l(j)) \vee (i = r(j))\}$ $T_{I_1 \to I_2} = \{[i_1] \to [\vartheta(1, i_1), 1, i_1]\}$ $T_{J_1 \to J_2} = \{[j_1] \to [\vartheta(2, j_1), 2, j_1]\}$	3. Full sparse tile by traversing $D_{I_1 \to J_1}$ to generate tiling ϑ