PERFORMANCE ENGINEERING OF SOFTWARE SYSTEMS

Performance Issues in Parallelization

Saman Amarasinghe

Fall 2010

Today's Lecture

Performance Issues of Parallelism

- Cilk provides a robust environment for parallelization
  - It hides many issues and tries to eliminate many problems
- Last lectures we looked at
  - Cache oblivious algorithms
  - algorithmic issues → Work and Span
- Today, synchronization and memory impact on parallel performance
  - We will use OpenMP instead of Cilk
  - Most of these issues also affects Cilk programs
    - But easier to invoke and analyze without the complexities of Cilk

Issues Addressed

- Granularity of Parallelism
- True Sharing
- False Sharing
- Load Balancing

Matrix Multiply in Cilk

```
cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
        for(int k=0; k < n; ++k) {
            A[i][j] = A[i][j] + B[i][k] * C[k][j];
        }
    }
}
```

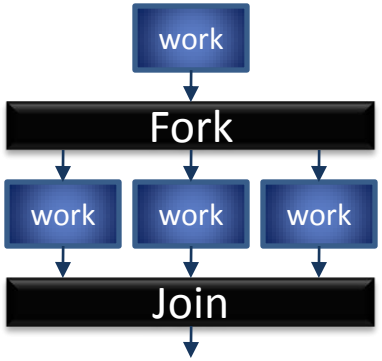
Scheduler

- Maps cilk_for into a divide and conquer pattern
- Distribute work according to a work stealing scheduler
- Hides computation distribution and load balance issues

OpenMP

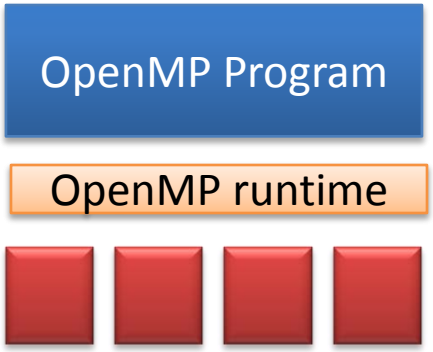
A "simplified" programming model for parallelism

- Architecture independent (all shared-memory architectures)
- Fork-join model



- Parallel loops (data parallel) and parallel sections (task parallel)
- Can select from several static and dynamic scheduling policies

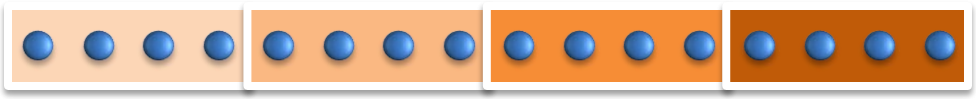
```
#pragma omp for schedule (static, chunk)
for (i=0; i<N; i++)
    for(j=0; j<N; j++)
        for (k=0; k<N; k++)
            A[i][j] += B[i][k] * C[k][j];
```



Static Schedules

Assume 4 processors

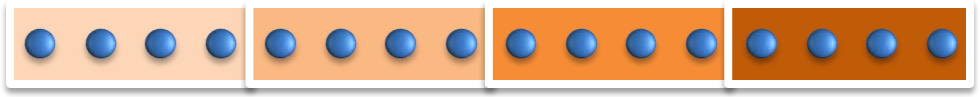
```
#pragma omp for schedule (static, 4)
for (i=0; i<16; i++)
    .......
```



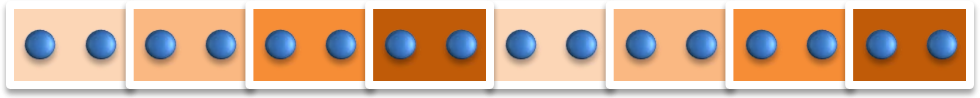
Static Schedules

Assume 4 processors

```
#pragma omp for schedule (static, 4)
for (i=0; i<16; i++)
    .......
```



```
#pragma omp for schedule (static, 2)
for (i=0; i<16; i++)
    .......
```



Pthreads

"Assembly" level parallelism

- Directly expose the processors/cores to the programmer

You need to manage your own threads.

A good strategy

- A thread per core
  - Perhaps threads < cores so a few cores are free to run other apps and OS services
- Bind the threads to cores
- SPMD (Single Program Multiple Data) Programming

Pros:

- Full control.
- Any parallel programming pattern.

Cons:

- Small Bugs, Big Bugs and Heisenbugs

Compare Performance

```
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
          A[i][j]+= B[i][k] * C[k][j];
```

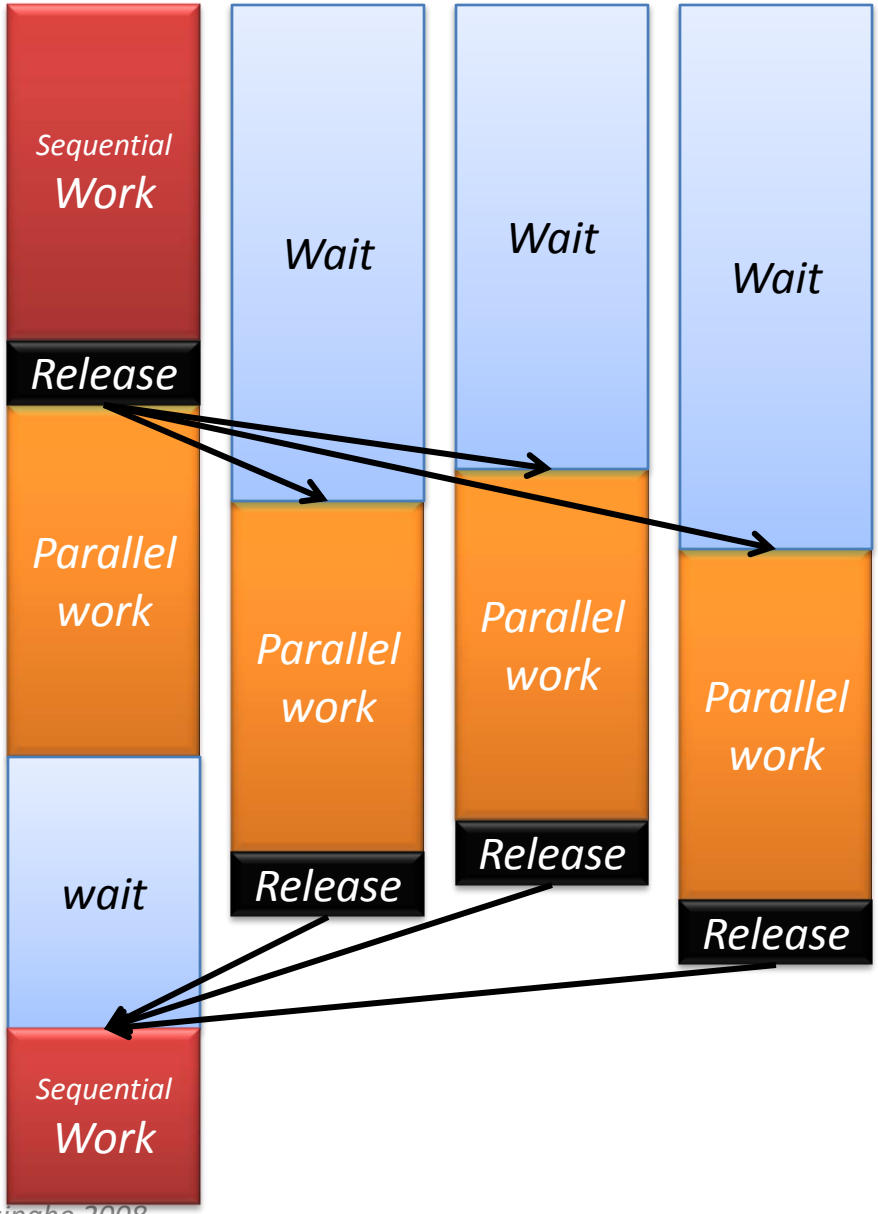
```
#pragma omp parallel for
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
          A[i][j]+= B[i][k] * C[k][j];
```

```
for(i =0; i < n; i++)
    #pragma omp parallel for
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
          A[i][j]+= B[i][k] * C[k][j];
```

	Execution Time	Speedup
Sequential	1944.29	1.00
Outer	265.08	7.33
Inner	300.03	6.48

Execution of a data parallel region

Synchronization overhead



Fine Grain Parallelism

Why?

- Too little work within a parallel region
- Synchronization (start & stop parallel execution) dominates execution time

How to Detect Fine Grain Parallelism?

- Parallel execution is slower than the sequential execution or
- Increasing the # of processors don't increase the speedup as expected
- Measure the execution time within the parallel region

How to get Coarse Grain Parallelism?

- Reduce the number of Parallel Invocations
  - Outer loop parallelism
  - Large independent parallel regions

Compare Performance

```
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
          A[i][j]+= B[i][k] * C[k][j];
```

```
#pragma omp parallel for
for(i =0; i < n; i++)
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
          A[i][j]+= B[i][k] * C[k][j];
```

```
for(i =0; i < n; i++)
    #pragma omp parallel for
    for(j =0; j < n; j++)
        for(k=0; k < n; k++)
          A[i][j]+= B[i][k] * C[k][j];
```

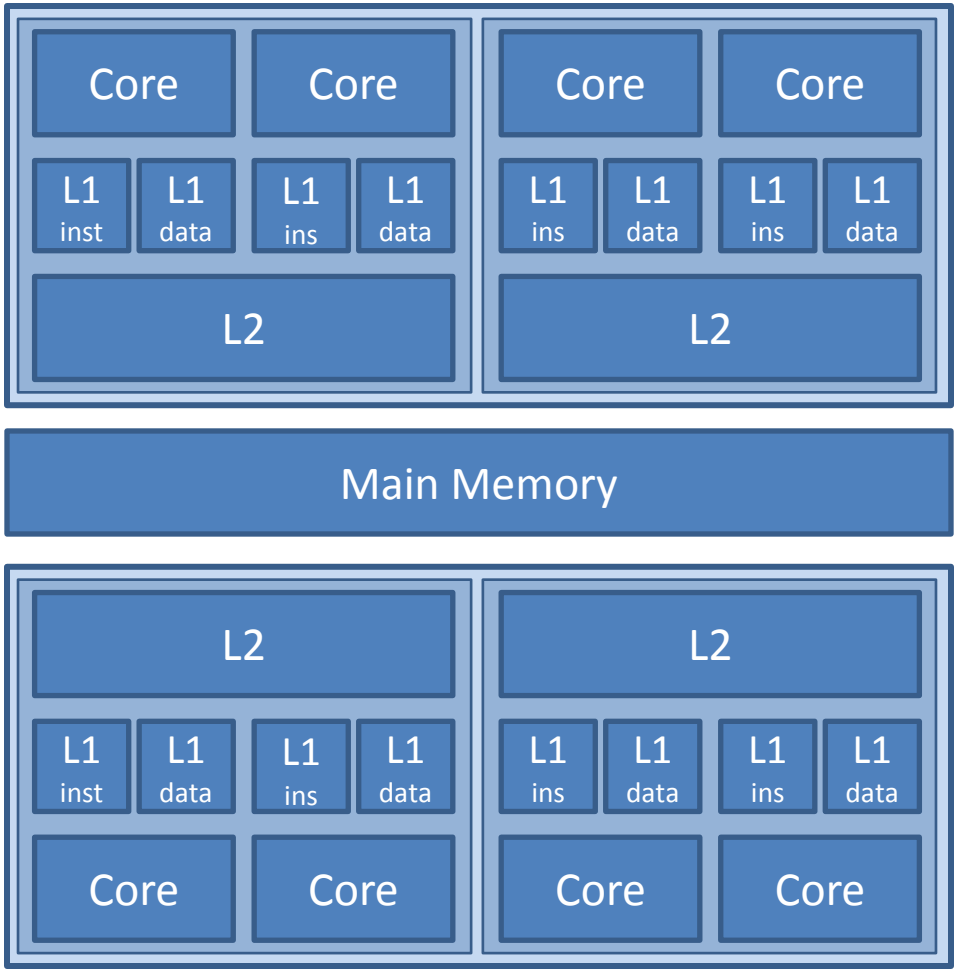
	Execution Time	Speedup	# of syncs
Sequentail	1944.29	1.00	0
Outer	265.08	7.33	n
Inner	300.03	6.48	n*n

Parallel Performance

```
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        A[i][j] = A[i][j] + B[i][j];
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        A[n - 1 - i][j] = A[n - 1 - i][j] + C[i][j];
```

	Execution Time	
Sequential	30	1.00
Parallel	35	0.86

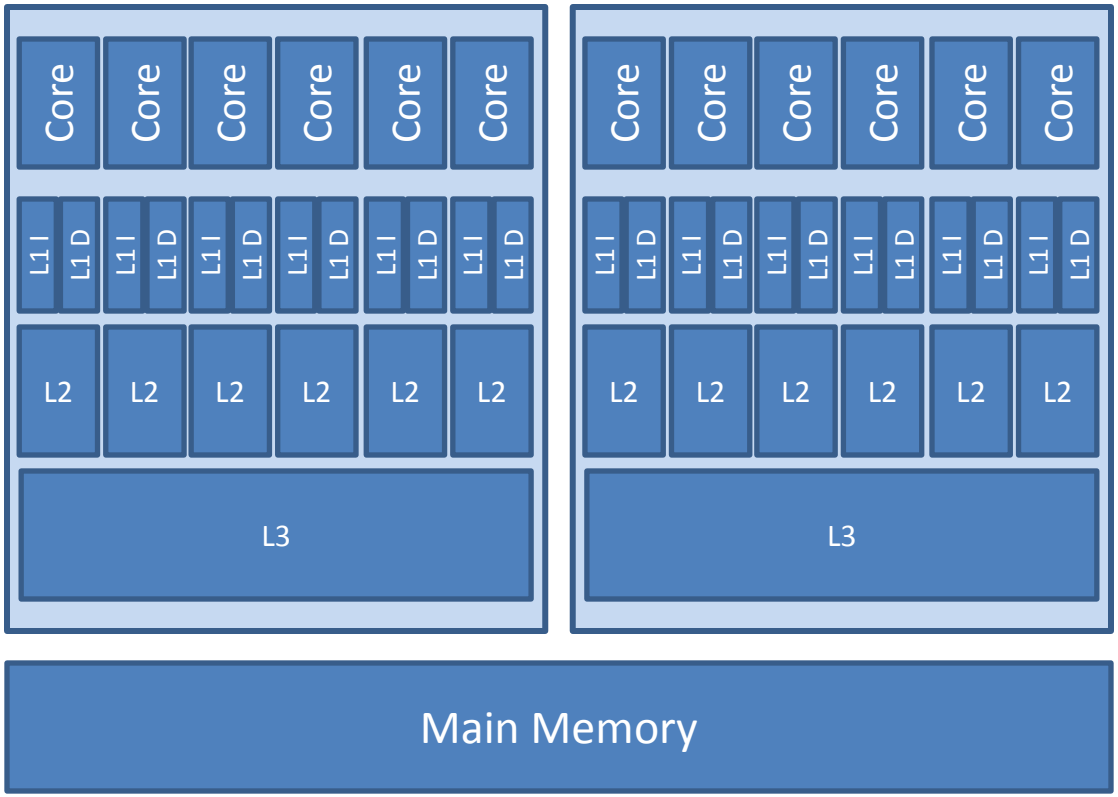
CagnodeX's memory configuration (used last year) Core 2 Quad processors



L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

CloudX's memory configuration

Nehalem 6 core processors

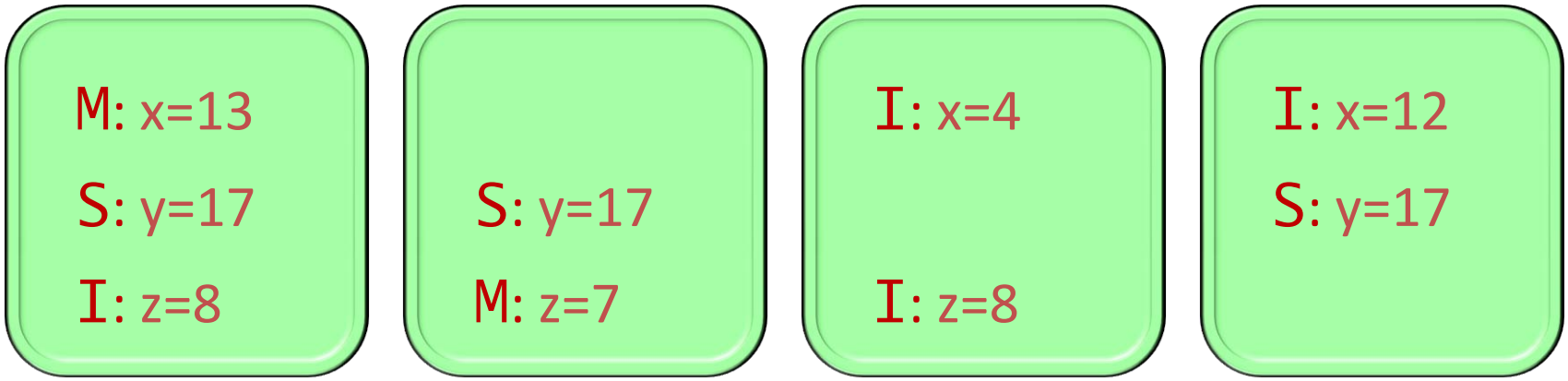


L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	4-way
L2 Cache			
Size	Line Size	Latency	Associativty
256 KB	64 bytes	10 ns	8-way
L3 Cache			
Size	Line Size	Latency	Associativty
12 MB	64 bytes	50 ns	16-way
Main Memory			
Size	Line Size	Latency	Associativty
	64 bytes	75 ns	

MSI Protocol

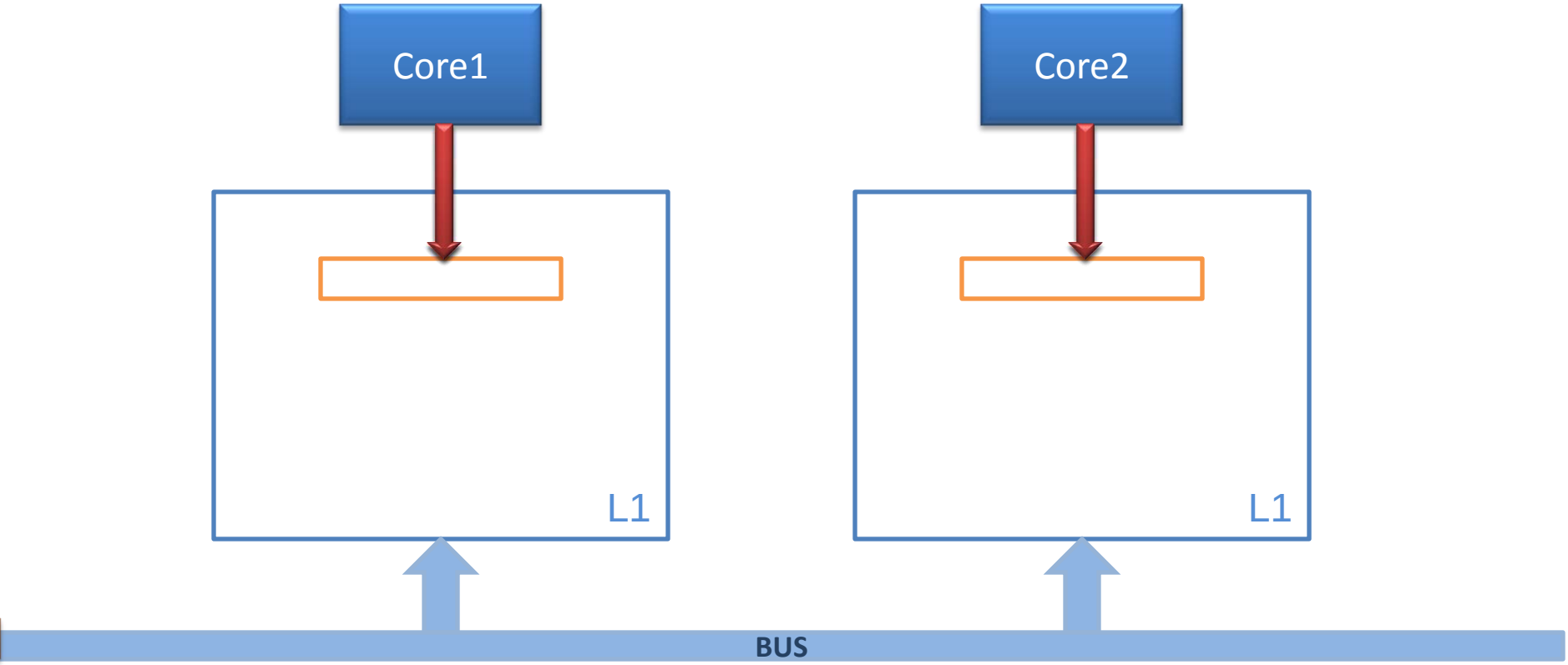
Each cache line is labeled with a state:

- M: cache block has been modified. No other caches contain this block in M or S states.
- S: other caches may be sharing this block.
- I: cache block is invalid (same as not there).

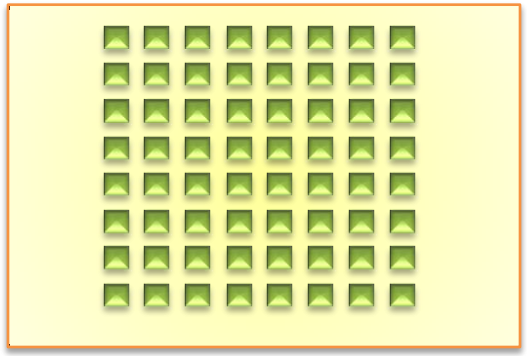


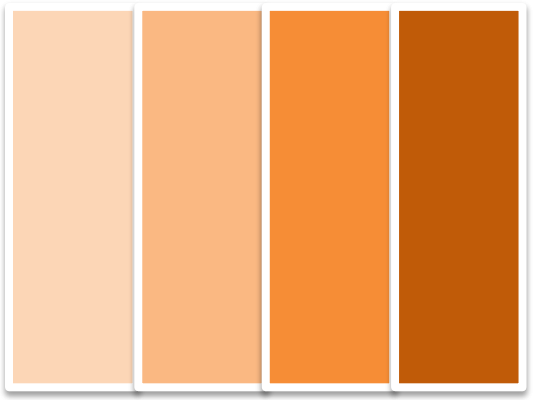
Before a cache modifies a location, the hardware first invalidates all other copies.

True Sharing



True Sharing





```
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        A[i][j] = A[i][j] + B[i][j];
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        A[n - 1 - i][j] = A[n - 1 - i][j] + C[i][j];
```

Cagnode	Execution Time		Instructions		CPI		L1 Miss Rate	L2 Miss Rate	Invalidations	
Sequential	30	1.00	2.02E+08	1.00	0.53	1.00	0.01	0	25,840	1.00
Parallel	35	0.86	6.95E+08	3.44	2.14	4.04	0.01	0	4,875,962	188.70

True Sharing

No True Sharing within a data parallel region

- There cannot be read/write or write/write conflicts

Sharing across different data parallel regions/invocations

Identifying Excessive True Sharing

- Look for cache invalidations

Eliminating Excessive True Sharing

- Try to make sharing minimal
- Data in one core's cache, lets keep it there!
- Try to "align" computation across regions
- Enforce a scheduling technique that'll keep the data aligned

Eliminate True Sharing

```
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        A[i][j] = A[i][j] + B[i][j];
#pragma omp parallel for
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        A[n - 1 - i][j] = A[n - 1 - i][j] + C[i][j];
```

```
#pragma omp parallel for
  for(i=0; i < n; i++)
    for(j=0; j < n; j++)
      A[i][j] = A[i][j] + B[i][j];
#pragma omp parallel for
  for(i=0; i < n; i++)
    for(j=0; j < n; j++)
      A[i][j] = A[i][j] + C[n - 1 - i][j];
```

Cagnode	Execution Time		Instructions		CPI		L1 Miss Rate	L2 Miss Rate	Invalidations	
Sequential	30	1.00	2.02E+08	1.00	0.53	1.00	0.01	0	25,840	1.00
Parallel	35	0.86	6.95E+08	3.44	2.14	4.04	0.01	0	4,875,962	188.70
Parallel (Transformed)	7.31	4.11	5.24E+08	2.59	0.96	1.81	0	0	197,679	7.65

Eliminate True Sharing

Cloud	Execution Time	
Sequential	23	1.00
Parallel	6	3.88
Parallel (Transformed)	5	4.60

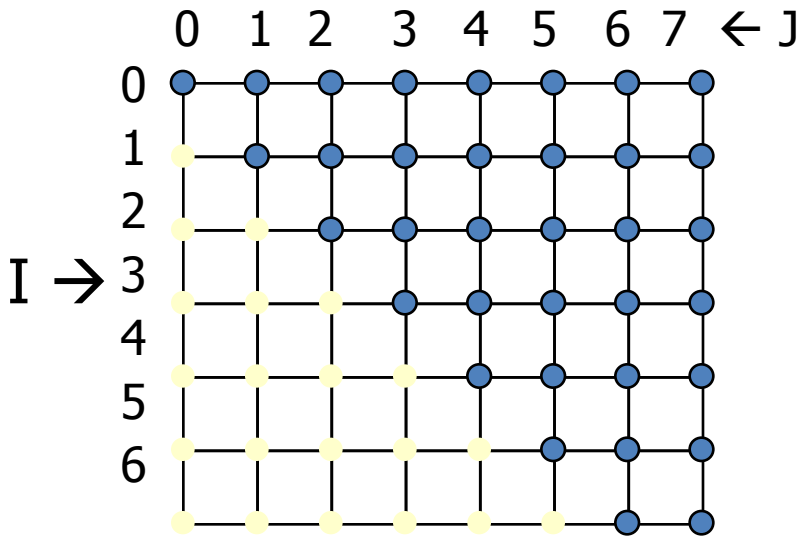
Cagnode	Execution Time		Instructions		CPI		L1 Miss Rate	L2 Miss Rate	Invalidations	
Sequential	30	1.00	2.02E+08	1.00	0.53	1.00	0.01	0	25,840	1.00
Parallel	35	0.86	6.95E+08	3.44	2.14	4.04	0.01	0	4,875,962	188.70
Parallel (Transformed)	7.31	4.11	5.24E+08	2.59	0.96	1.81	0	0	197,679	7.65

Iteration Space

N deep loops → n-dimensional discrete cartesian space

- Normalized loops: assume step size = 1

```
for(int i = 0; i < 8; i++)
    for(int j = i; j < 8; j++)
```



Iterations are represented as coordinates in iteration space

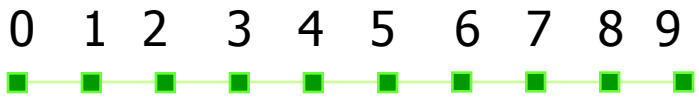
- $\bar{i} = [i_1, i_2, i_3, \ldots, i_n]$

Data Space

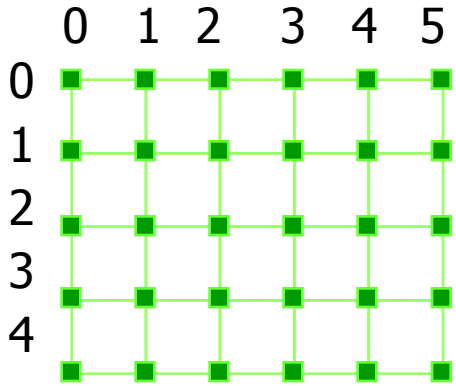
M dimensional arrays → m-dimensional discrete cartesian space

- a hypercube

```
int A[10]
```

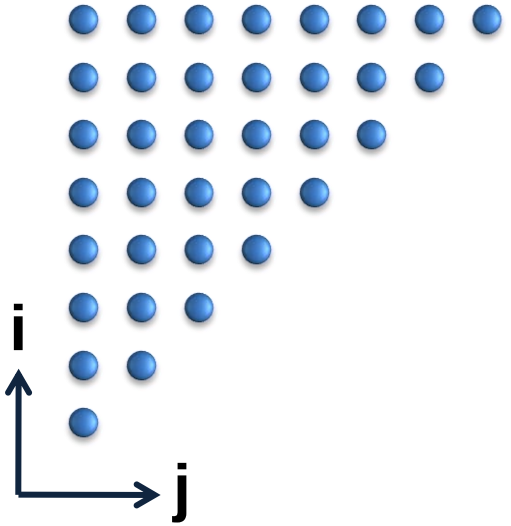


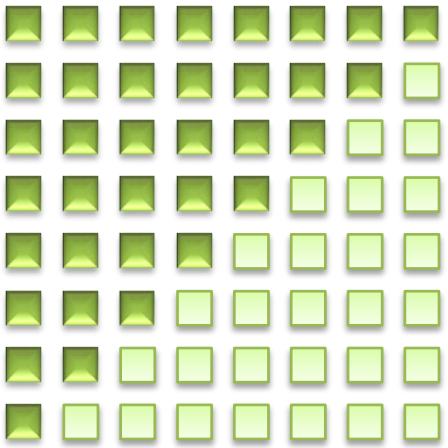

```
double B[5][6]
```



Triangular Matrix Add

```
for(i=0; i <n; i++)
    for(j=0; j<i; j++)
        A[i][j] = A[i][j] + B[i][j];
```

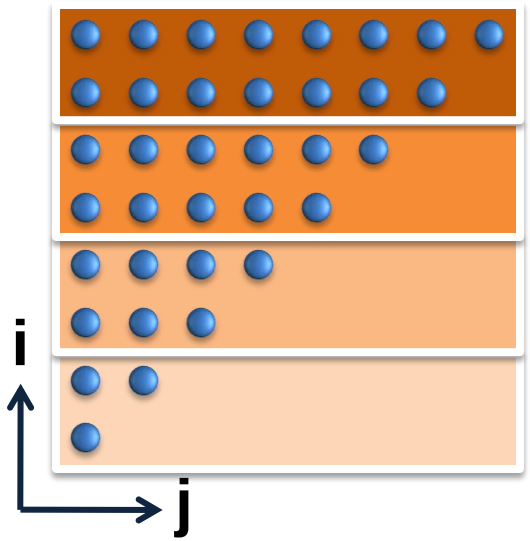


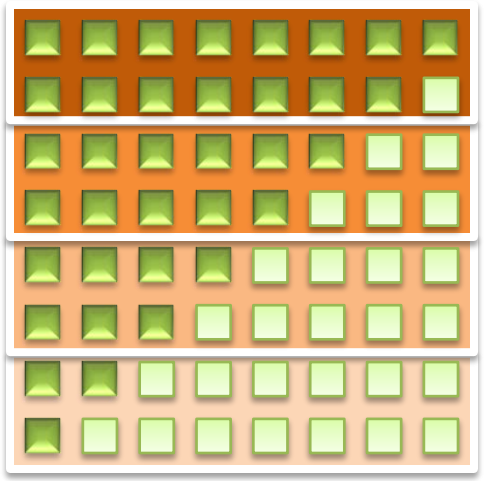


Parallelism via. Block Distribution

```
#pragma omp parallel for

for(i=0; i <n; i++)
    for(j=0; j<i; j++)
        A[i][j] = A[i][j] + B[i][j];
```

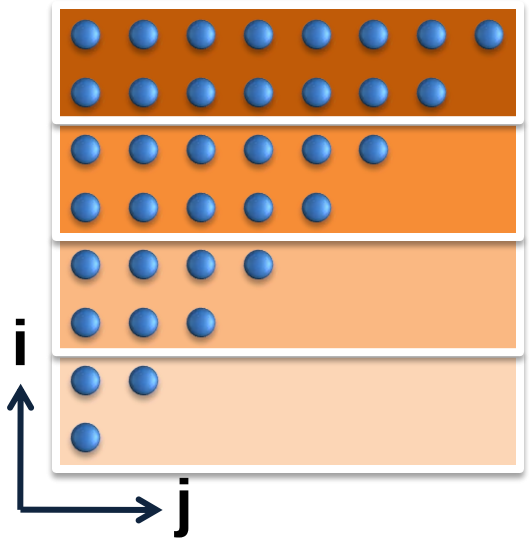




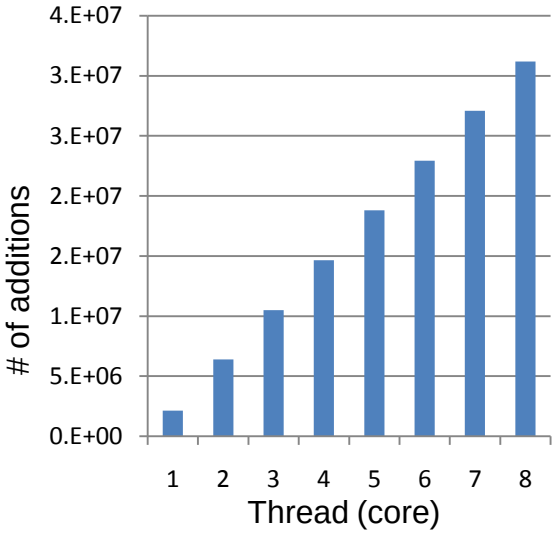
Parallelism via. Block Distribution

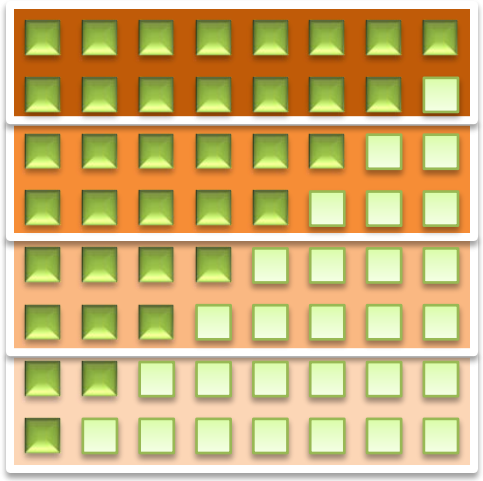
#pragma omp parallel for

```
for(i=0; i <n; i++)
    for(j=0; j<i; j++)
        A[i][j] = A[i][j] + B[i][j];
```



	Execution Time	
Sequential	97.38	1.00
Block distribution	31.60	3.08





Load Imbalance

Why?

- Each parallel sub-region has different amount of work
- Static: The amount of work for each sub-region is known at compile time
- Dynamic: The amount of work varies at runtime (cannot predict)

How to Detect Load Imbalance?

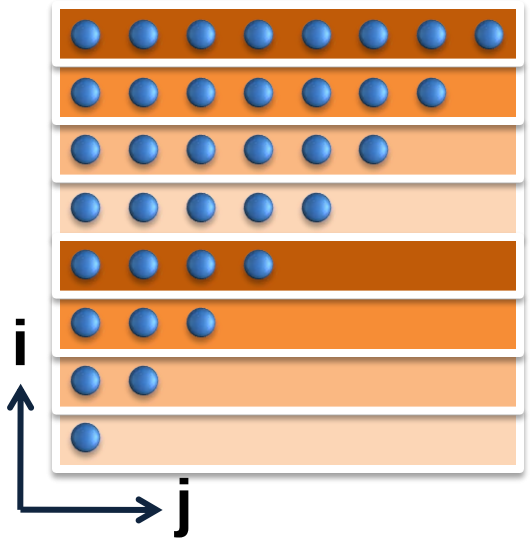
- Work done by each thread is not identical
  - However running many parallel sections can average this out
- Measure the difference between min and max time taken by each of the sub-regions of a parallel section. (keep the max of that and average parallel execution time over many invocation of the parallel region).

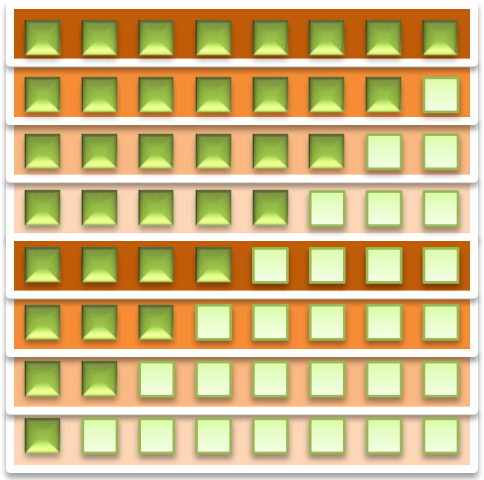
How to Eliminate Load Imbalance?

- Static: Use cyclic distribution
- Dynamic & Static: Use a runtime load balancing scheduler like a work queue or work stealing scheduler

Parallelism via. Cyclic Distribution

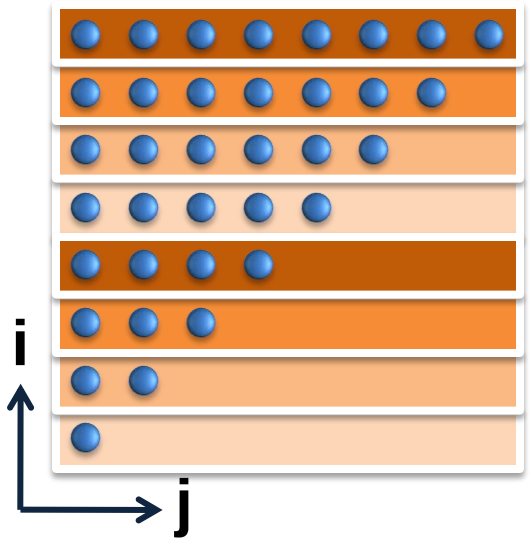
```
#pragma omp parallel for
                        schedule(static 1)
for(i=0; i <n; i++)
    for(j=0; j<i; j++)
        A[i][j] = A[i][j] + B[i][j];
```



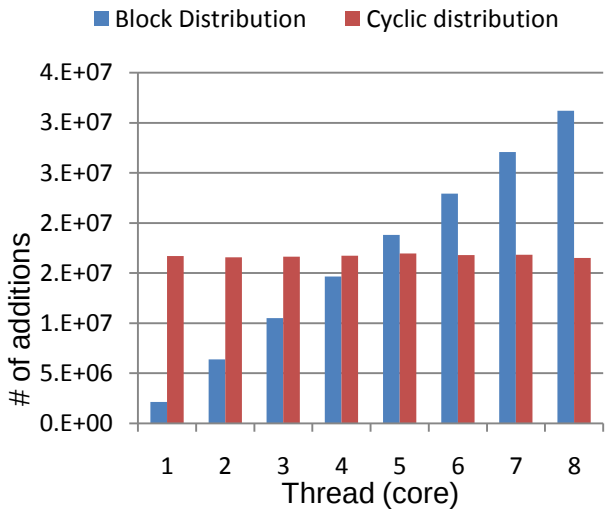


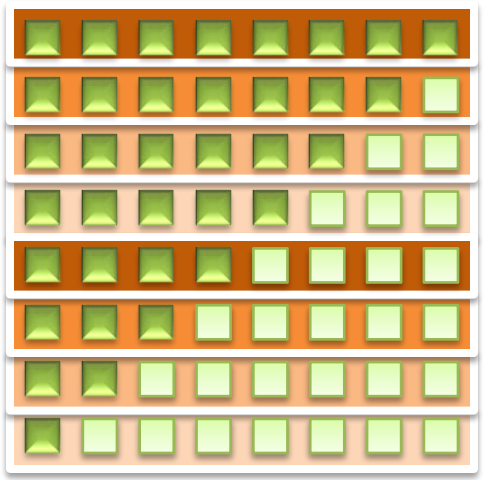
Parallelism via. Cyclic Distribution

```
#pragma omp parallel for
                    schedule(static 1)
for(i=0; i <n; i++)
    for(j=0; j<i; j++)
        A[i][j] = A[i][j] + B[i][j];
```



	Execution Time	
Sequential	97.38	1.00
Block distribution	31.60	3.08
Cyclic distribution	37.23	2.62

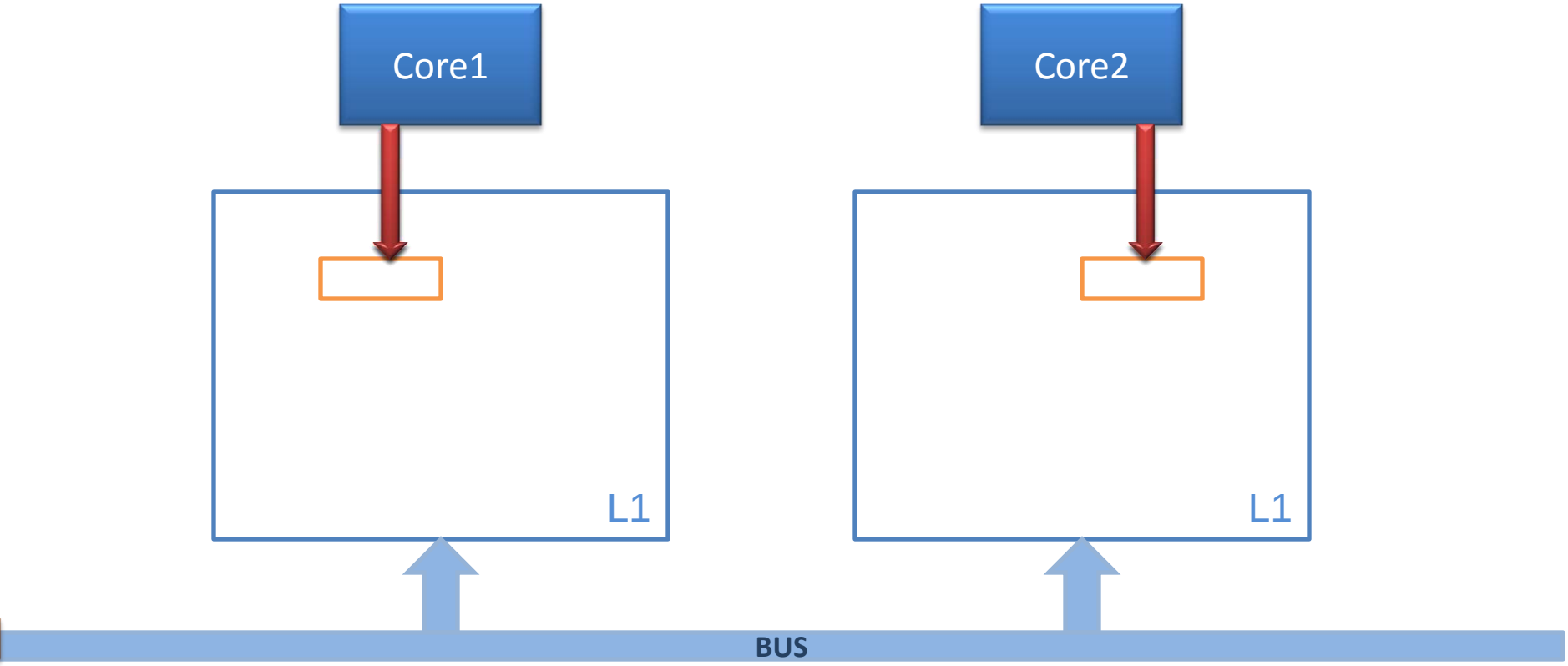




Load Balance but no Speedup?

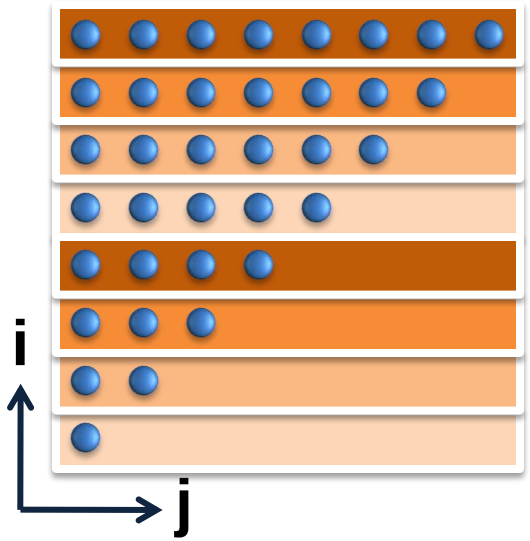
	Execution Time		Instructions		CPI		L1 Miss Rate	L2 Miss Rate	Invalidations	
Sequential	97.38	1.00	666,337,984	1.00	0.48	1.00	0.01	0	2,331	1.00
Block distribution	31.60	3.08	144,004,800	0.22	0.75	1.56	0.01	0	67,816	29.09
Cyclic distribution	37.23	2.62	1,462,153,984	2.19	0.96	2.00	0.01	0	1,196,448	513.28

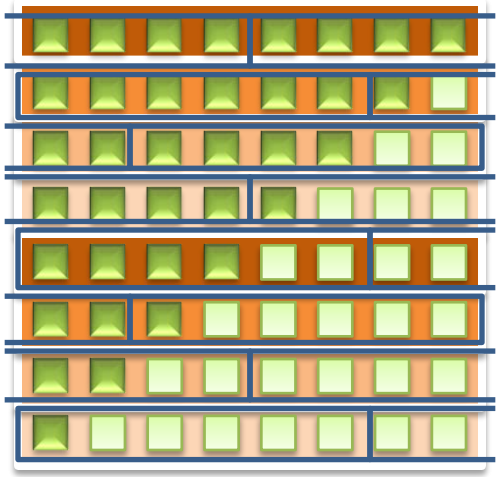
False Sharing



False Sharing in Cyclic Distribution

```
#pragma omp parallel for
                        schedule(static 1)
for(i=0; i <n; i++)
    for(j=0; j<i; j++)
        A[i][j] = A[i][j] + B[i][j];
```





False Sharing

Why?

- Cache Line Bigger Than Data Size
- Cache line is shared while data is not
- Can be a problem in data parallel loops as well as across regions

How to Detect False Sharing?

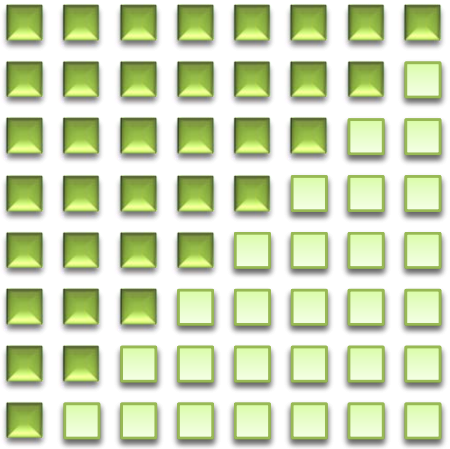
- Too many conflicts (especially in a data parallel loop)

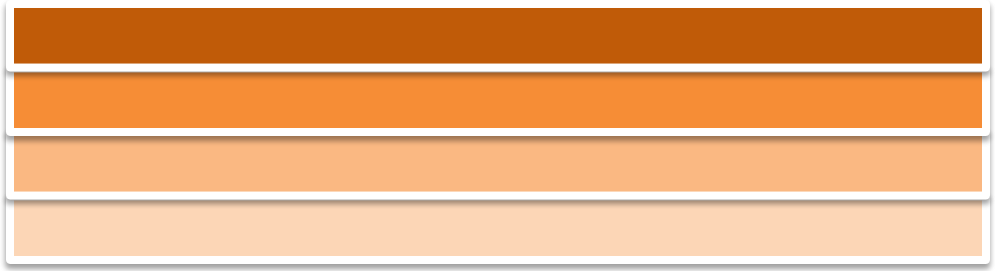
How to Eliminate False Sharing?

- Make data used within a core contiguous in memory
- Pad the ends so that no false sharing occurs at the boundaries

Data Transformation

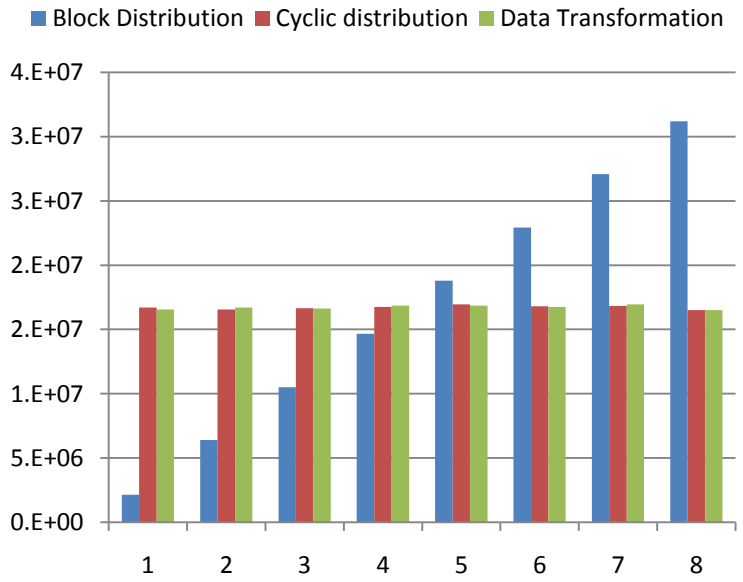
```
int A[NP][N/NP][N];
for(p=0; p<NP; p++)
   for(i=0; i <N/NP; i++)
      for(j=0; j<i*NP+P; j++)
         A[p][i][j]=A[p][i][j]+B[p][i][j];
```





Data Transformation

```
int A[NP][N/NP][N];
for(p=0; p<NP; p++)
   for(i=0; i <N/NP; i++)
      for(j=0; j<i*NP+P; j++)
         A[p][i][j]=A[p][i][j]+B[p][i][j];
```



	Execution Time		Instructions		CPI		L1 Miss Rate	L2 Miss Rate	Invalidations	
Sequential	97.38	1.00	666,337,984	1.00	0.48	1.00	0.01	0	2,331	1.00
Block distribution	31.60	3.08	144,004,800	0.22	0.75	1.56	0.01	0	67,816	29.09
Cyclic distribution	37.23	2.62	1,462,153,984	2.19	0.96	2.00	0.01	0	1,196,448	513.28
Data Transformation	18.00	5.41	1,121,090,048	1.68	0.75	1.56	0.01	0	108,262	46.44

Cache Issues

Cold Miss

- The first time the data is available
- Prefetching may be able to reduce the cost

Capacity Miss

- The previous access has been evicted because too much data touched in between
- "Working Set" too large
- Reorganize the data access so reuse occurs before getting evicted.
- Prefetch otherwise

Conflict Miss

- Multiple data items mapped to the same location. Evicted even before cache is full
- Rearrange data and/or pad arrays

True Sharing Miss

- Thread in another processor wanted the data, it got moved to the other cache
- Minimize sharing/locks

False Sharing Miss

- Other processor used different data in the same cache line. So the line got moved
- Pad data and make sure structures such as locks don't get into the same cache line

Dependences

True dependence

```
a   =
    =   a
```

Anti dependence

```
    =   a
a   =
```

Output dependence

```
a   =
a   =
```

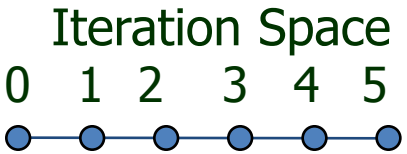
Definition:
Data dependence exists for a dynamic instance i and j iff

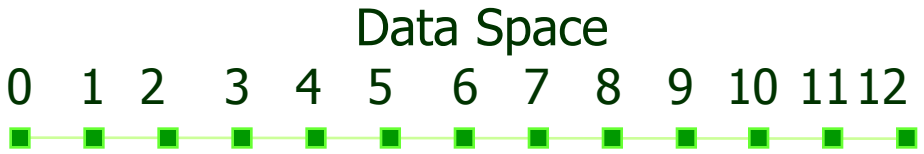
- either i or j is a write operation
- i and j refer to the same variable
- i executes before j

How about array accesses within loops?

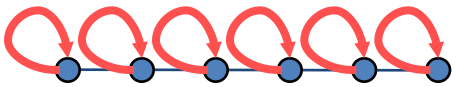
Array Accesses in a loop

```
for(int i = 0; i < 6; i++)
  A[i] = A[i] + 1
```

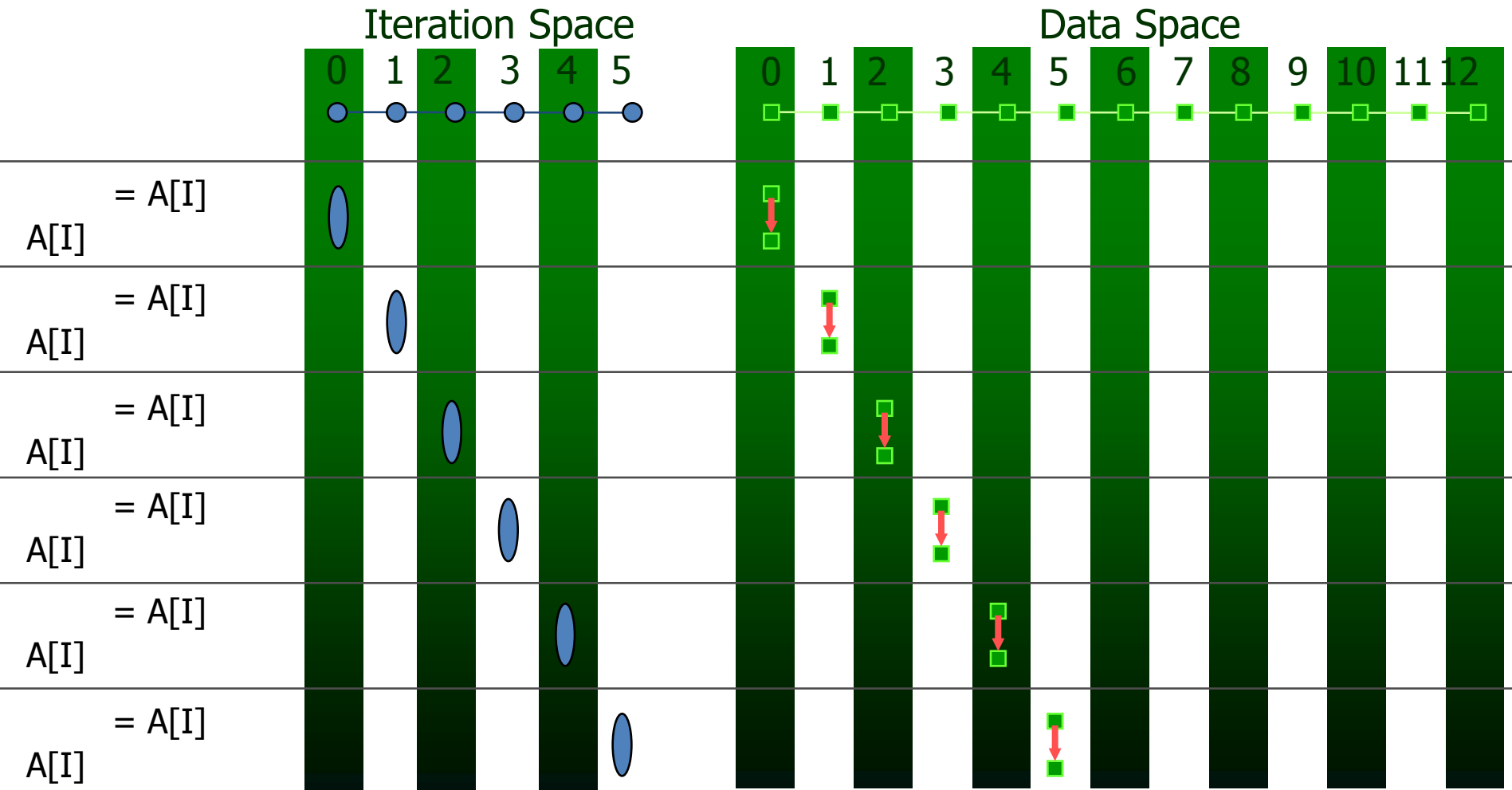




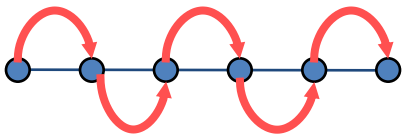
Array Accesses in a loop



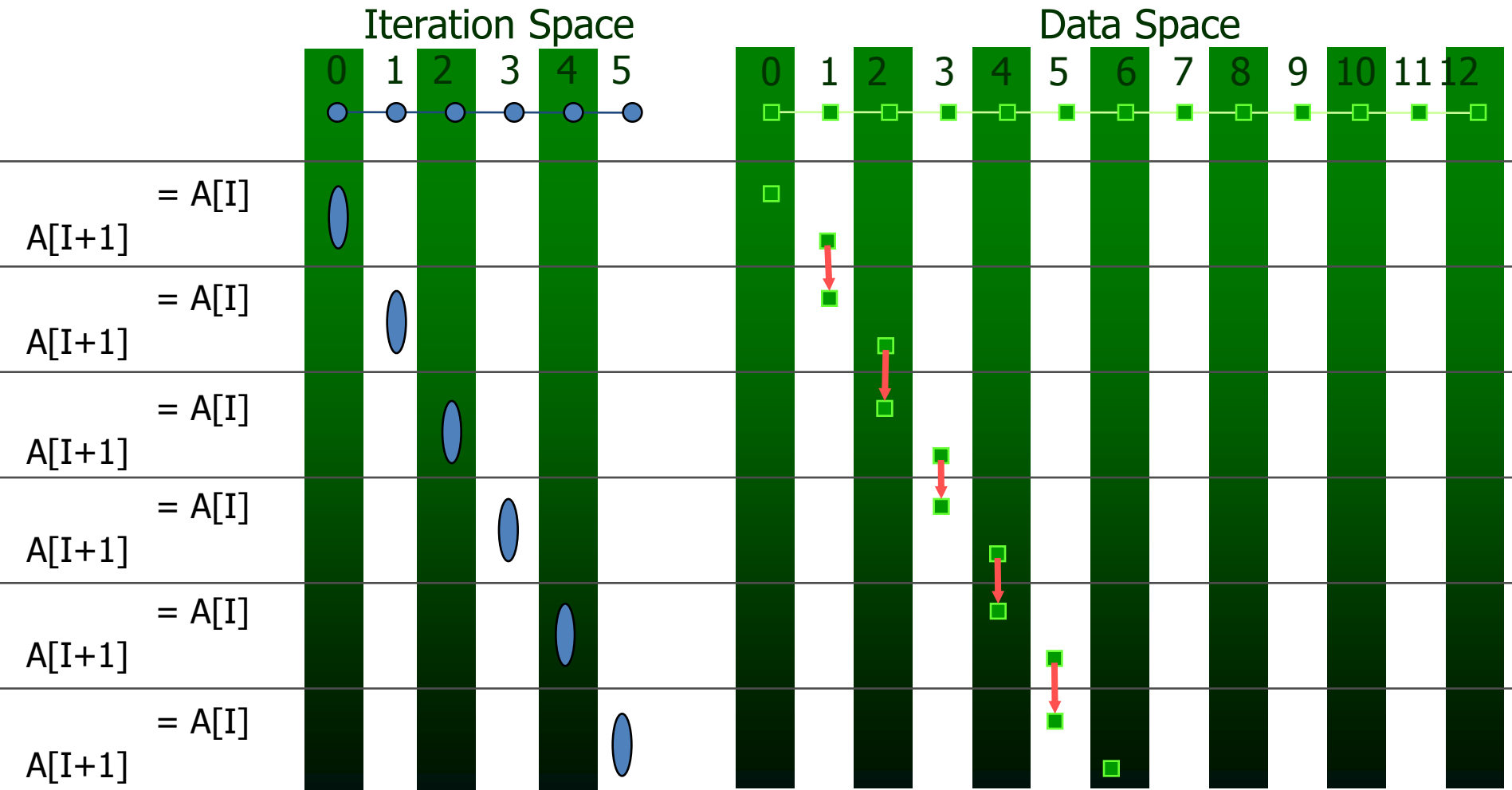
```
for(int i = 0; i < 6; i++)
  A[i] = A[i] + 1;
```



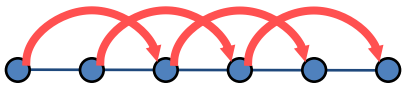
Array Accesses in a loop



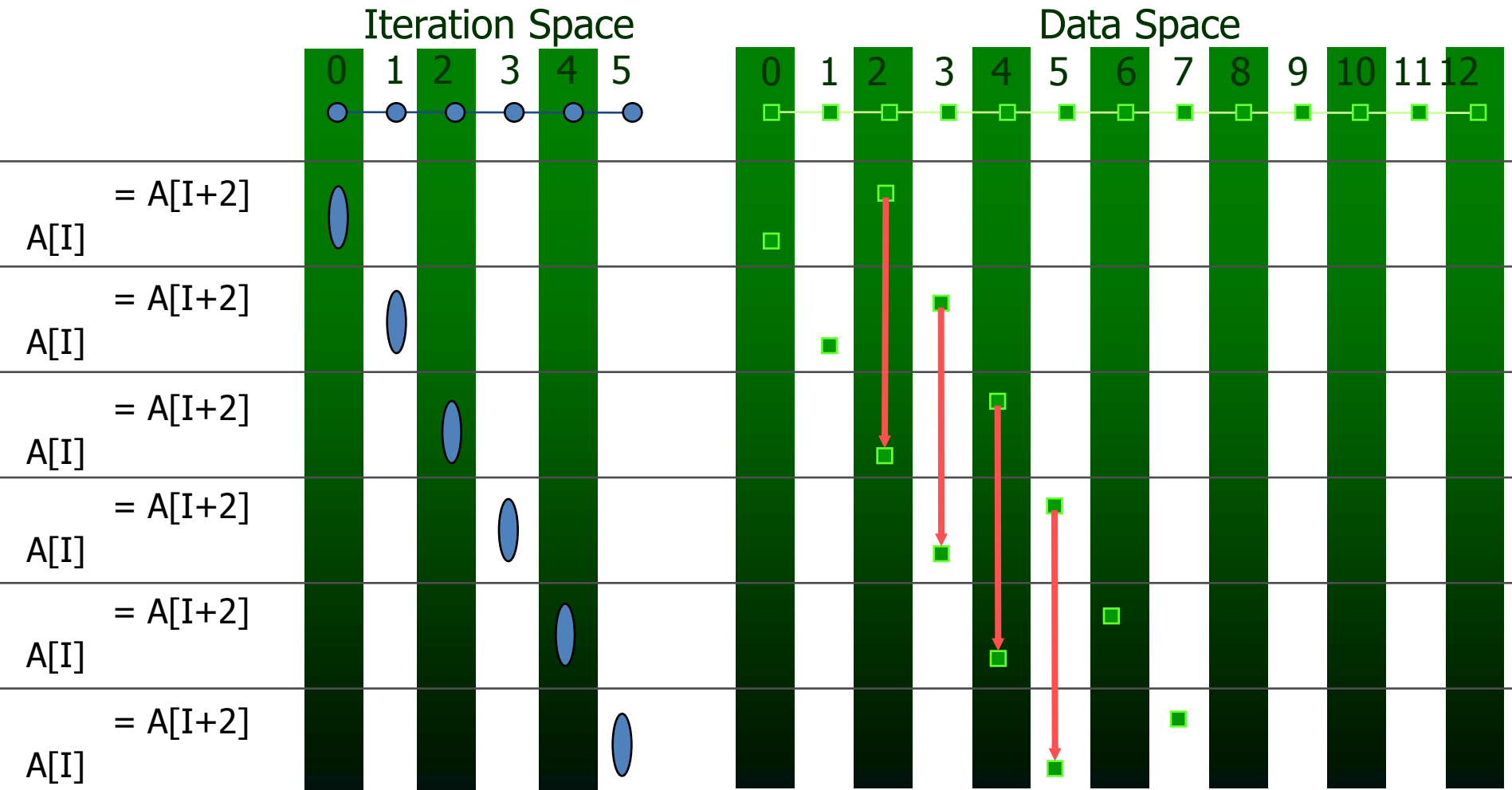
```
for(int i = 0; i < 6; i++)
  A[i+1] = A[i] + 1;
```



Array Accesses in a loop

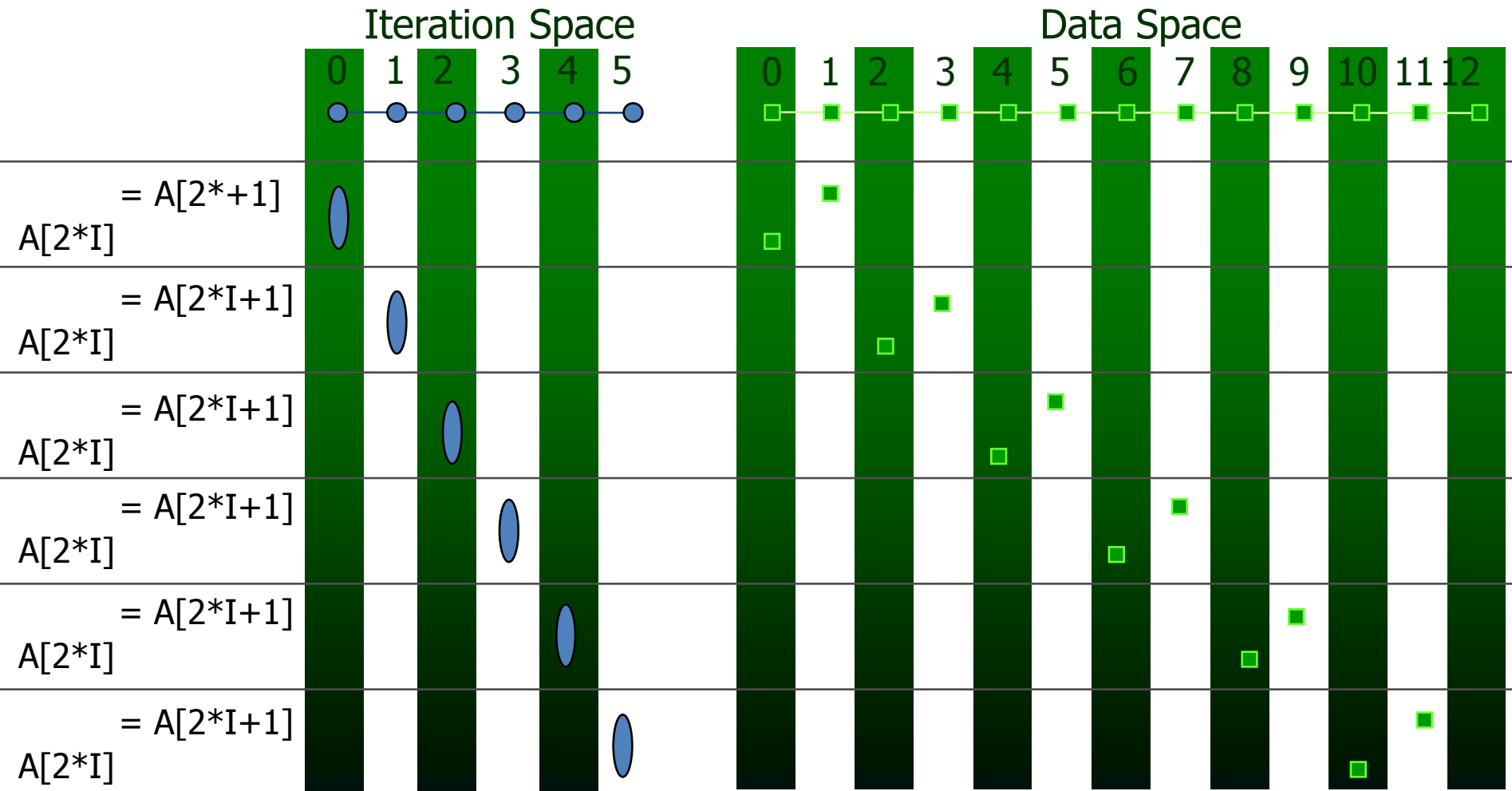


```
for(int i = 0; i < 6; i++)
  A[i] = A[i+2] + 1;
```



Array Accesses in a loop

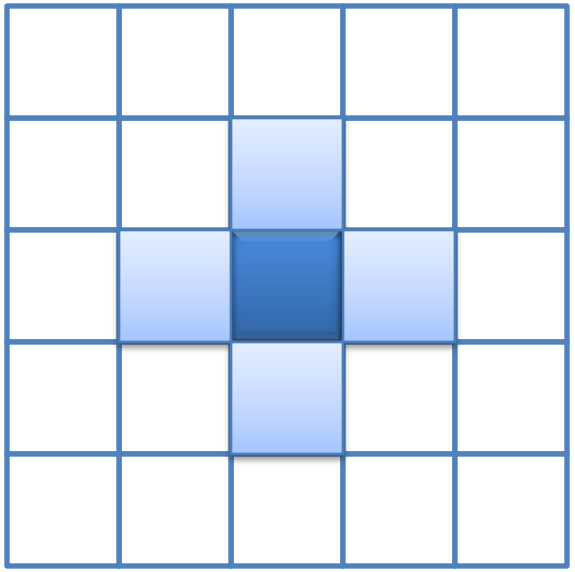
```
for(int i = 0; i < 6; i++)
  A[2*i] = A[2*i+1] + 1;
```



How to Parallelize SOR

SOR – Successive Over Relaxation

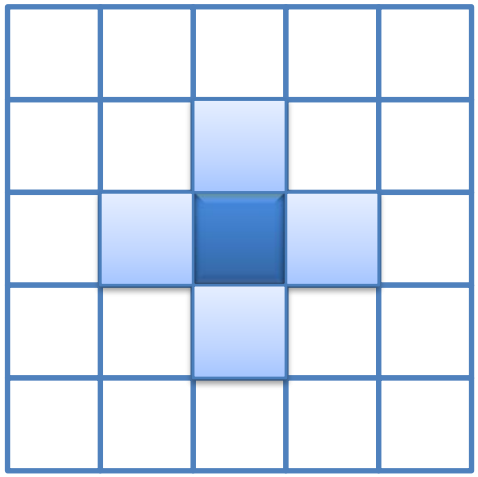
- Ex: Simulate the flow of heat through a plane



```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int i=1; i < N-1; i++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

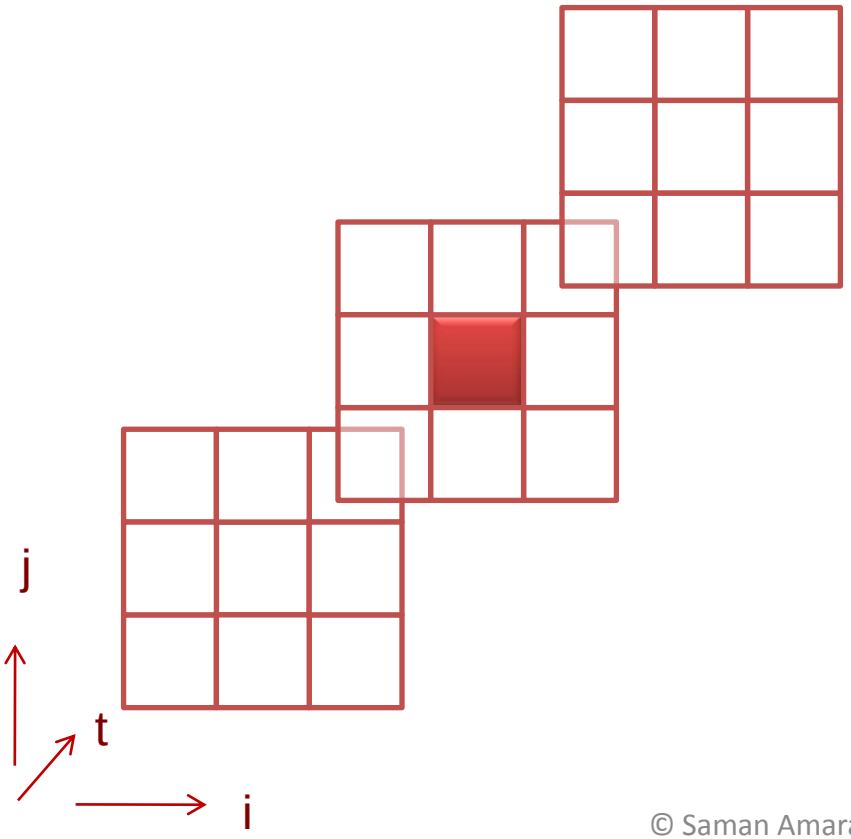
Data Dependences in SOR

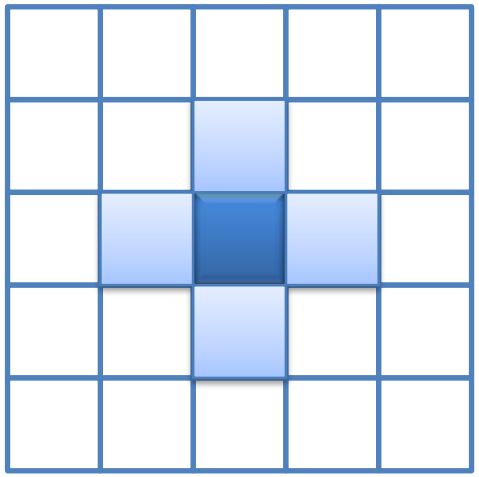
```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```



Data Dependences in SOR

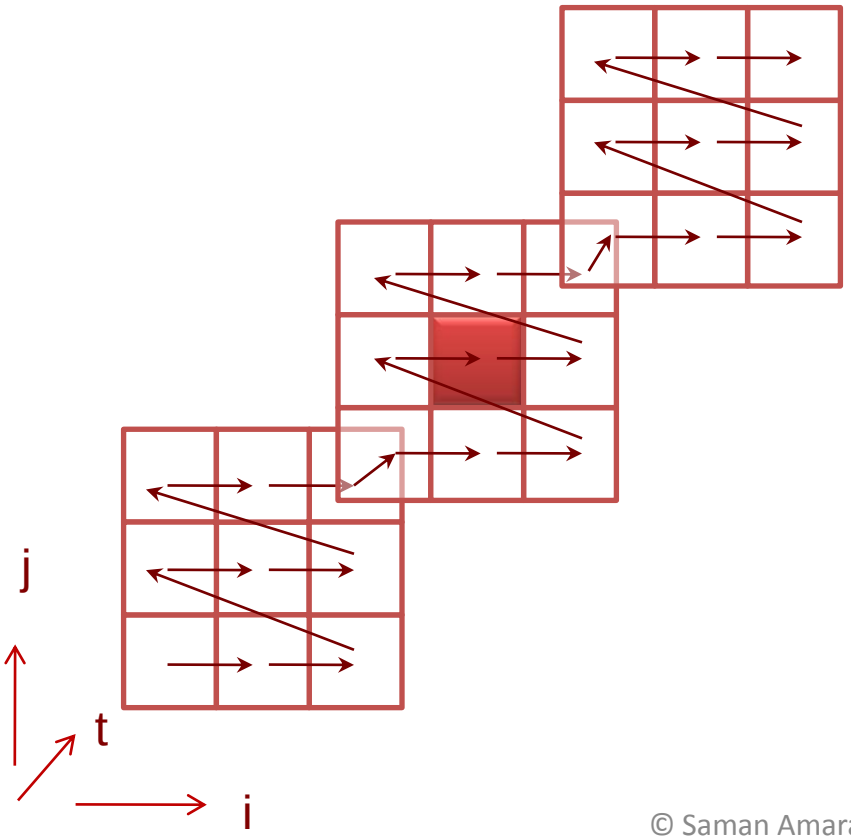
```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

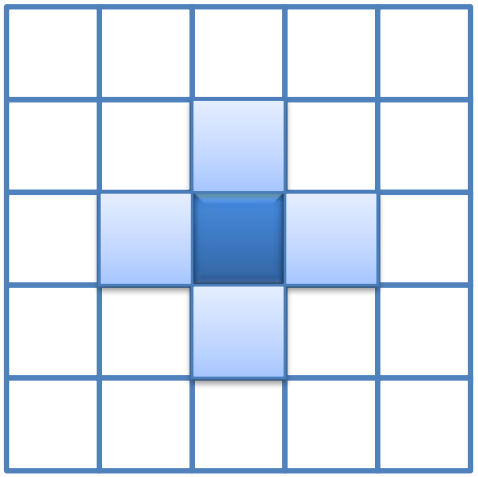




Data Dependences in SOR

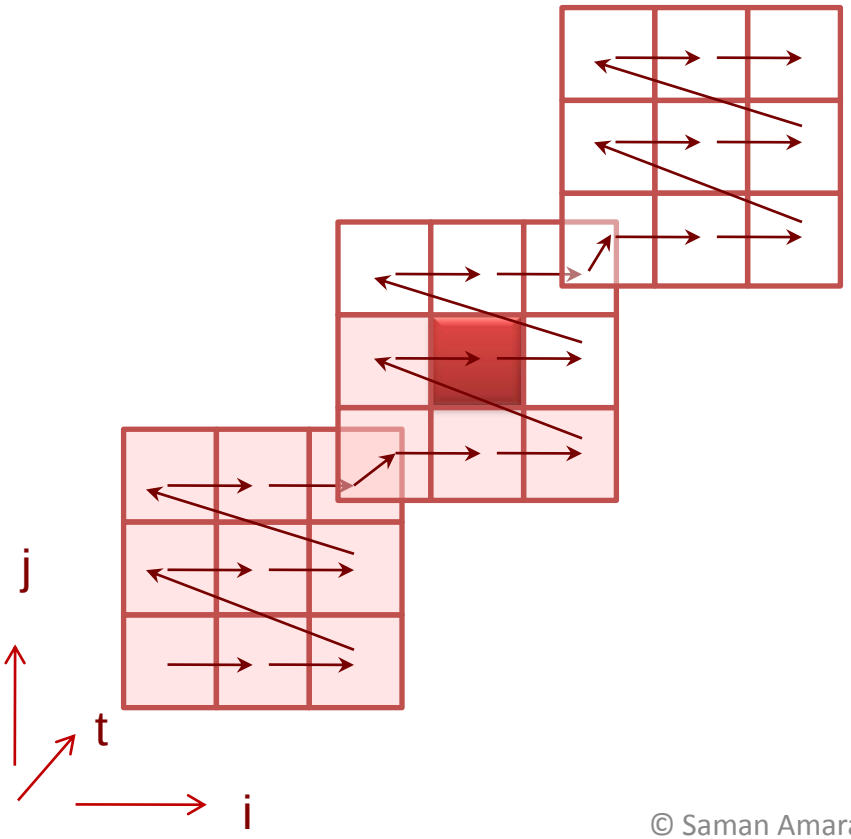
```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

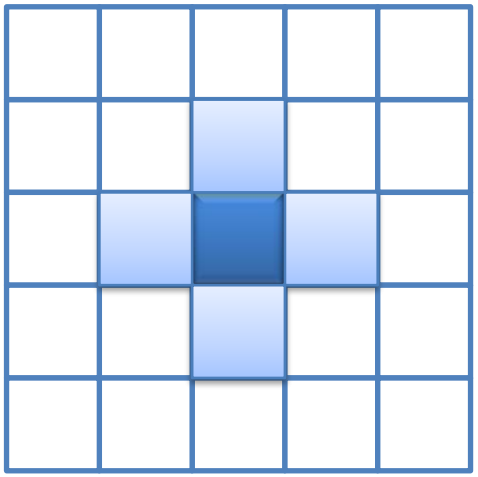




Data Dependences in SOR

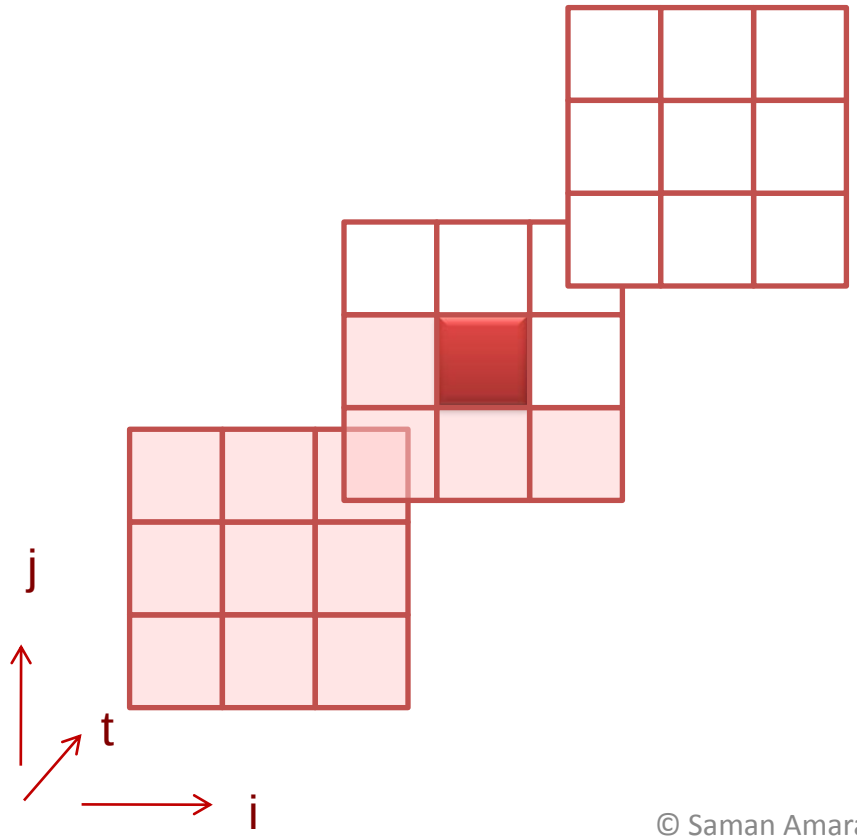
```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

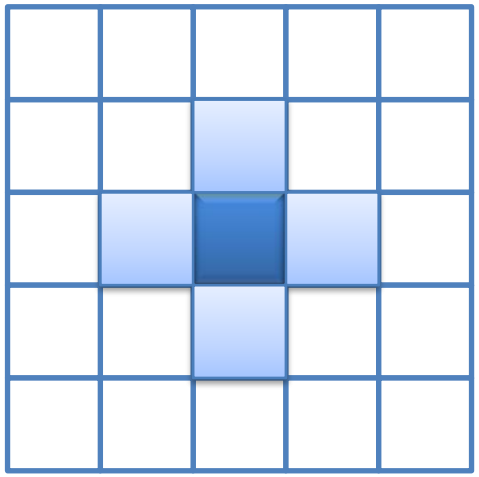




Data Dependences in SOR

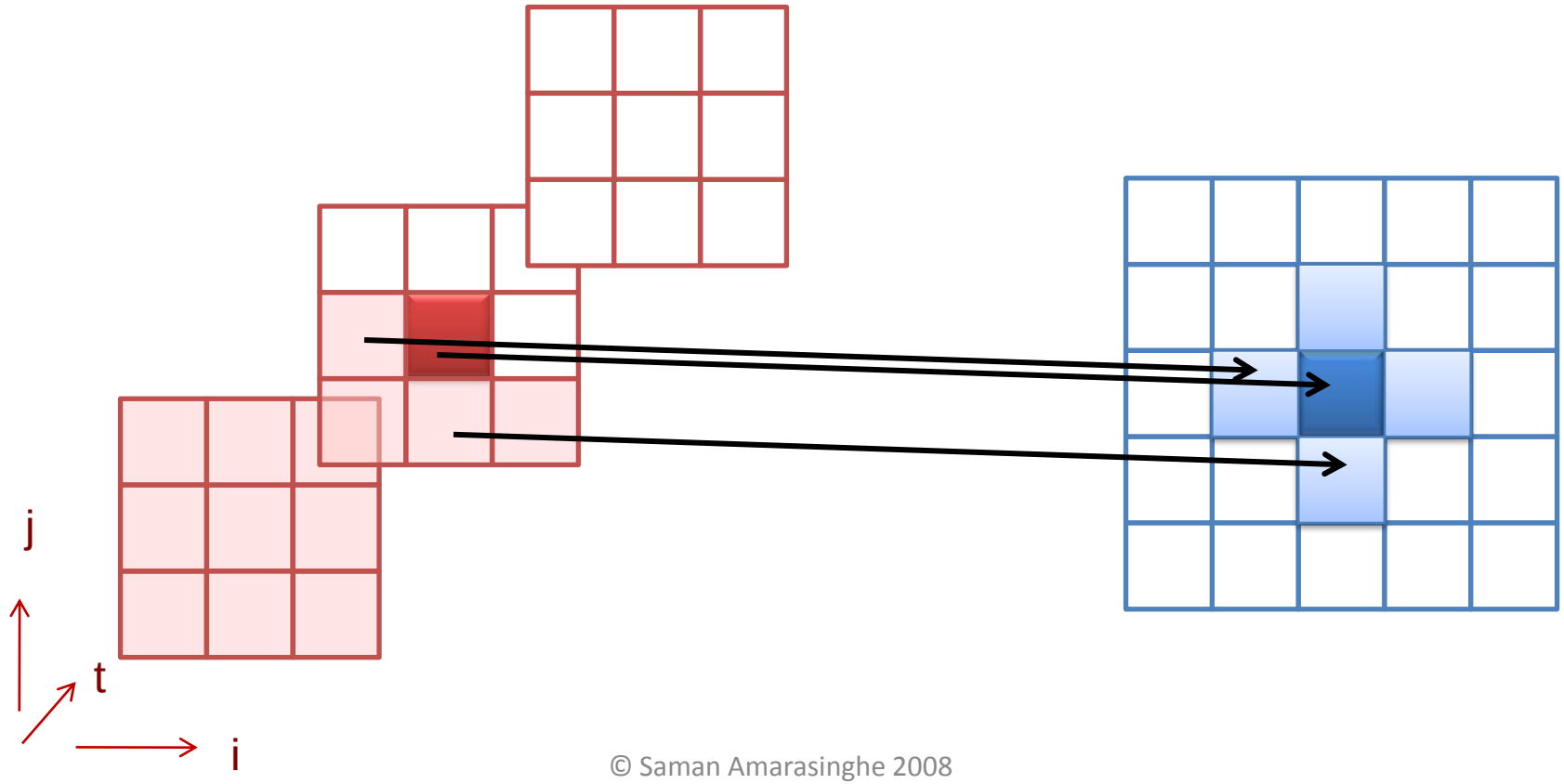
```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```





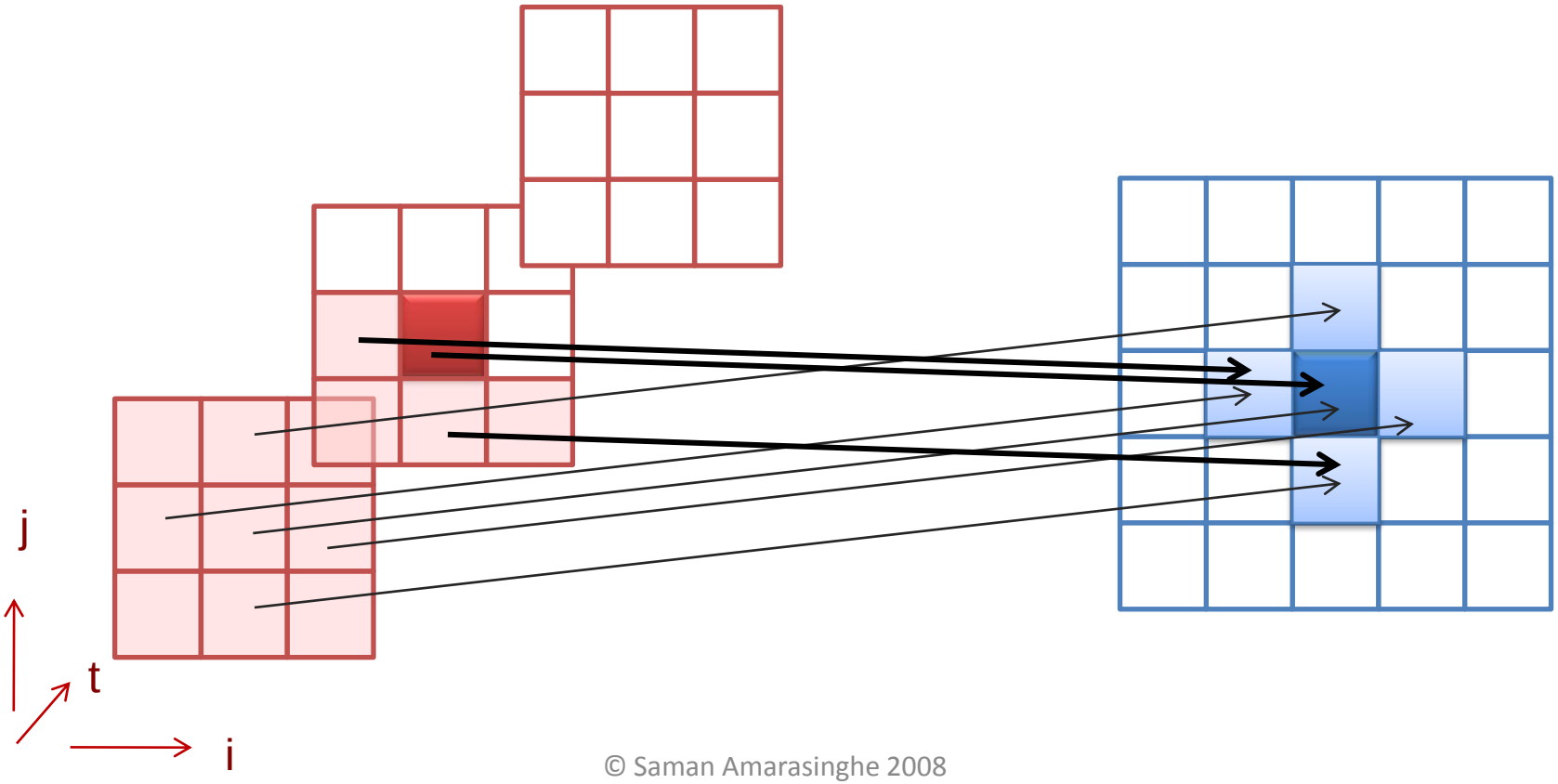
Data Dependences in SOR

```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```



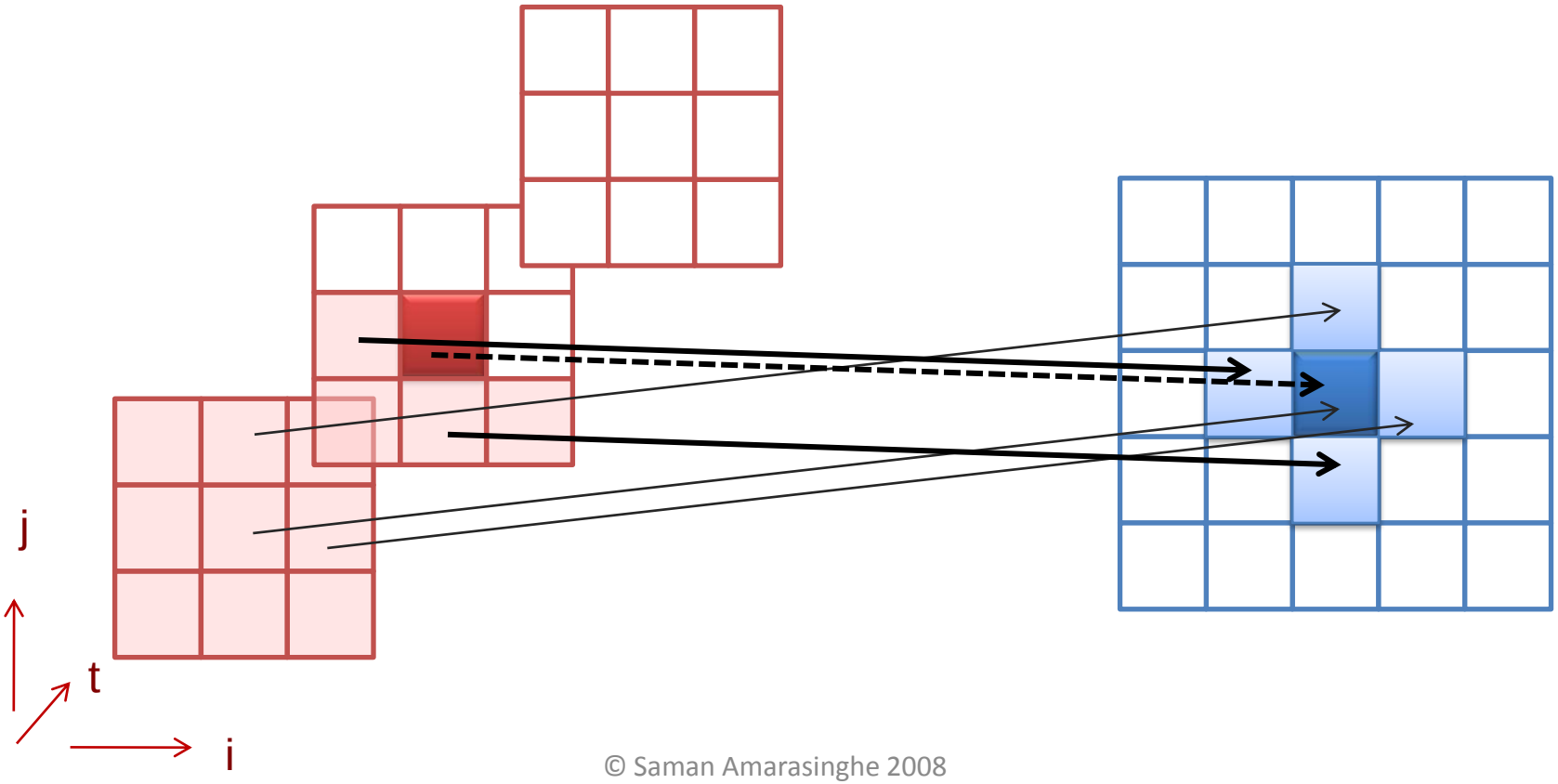
Data Dependences in SOR

```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```



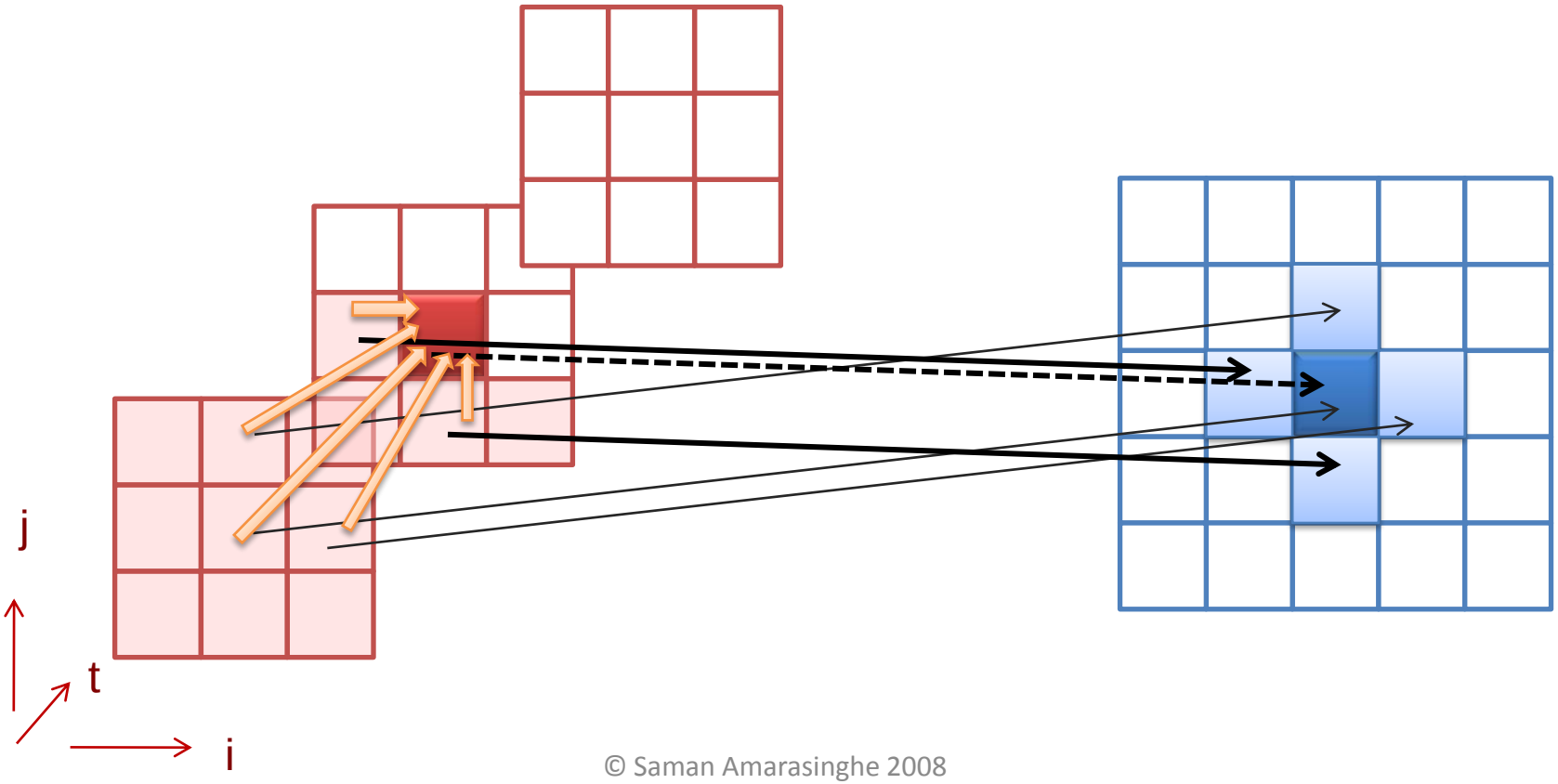
Data Dependences in SOR

```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```



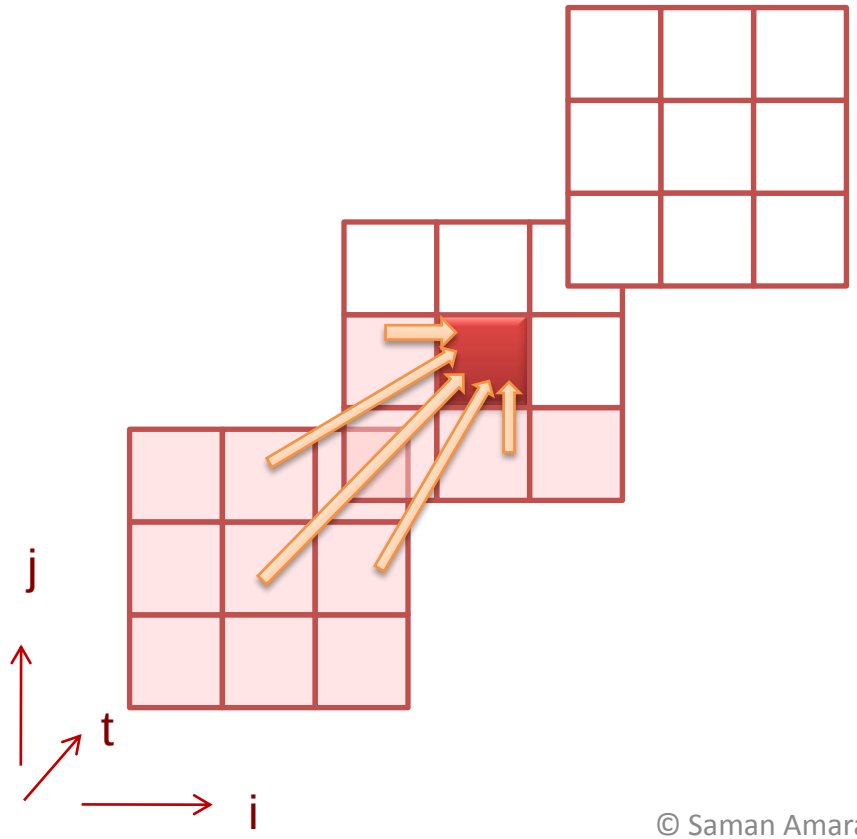
Data Dependences in SOR

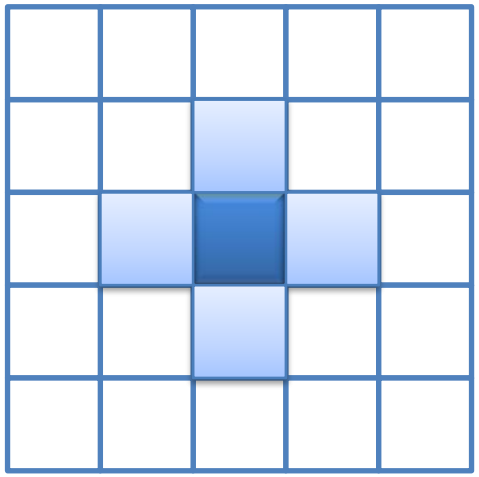
```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```



Data Dependences in SOR

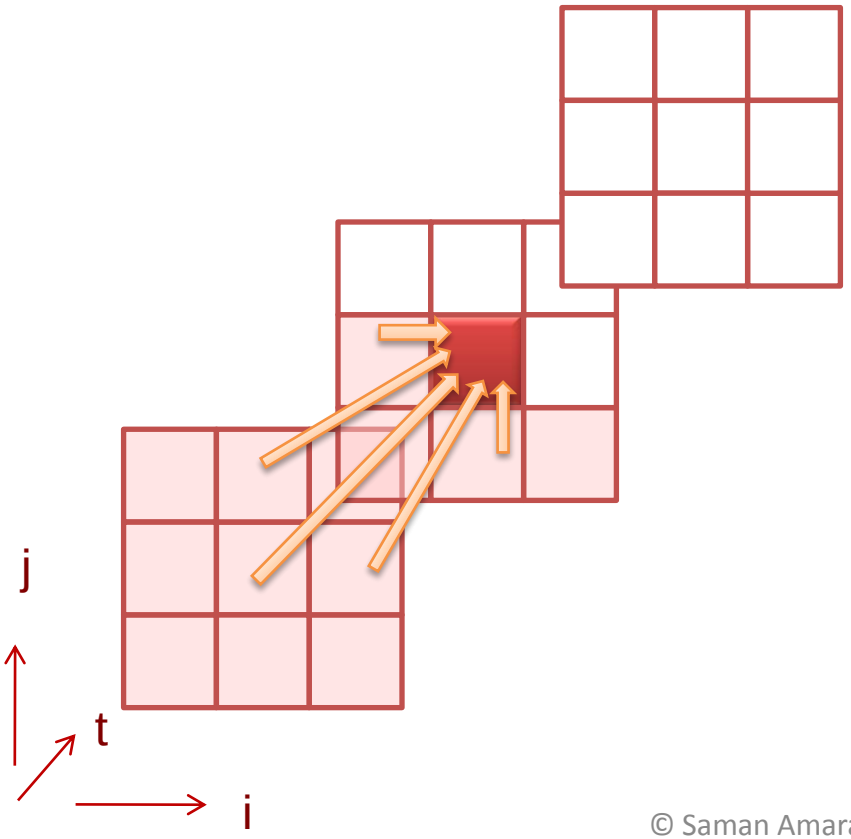
```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

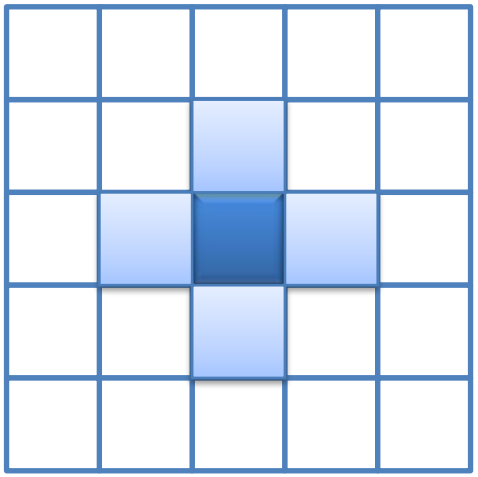




Creating a FORALL Loop

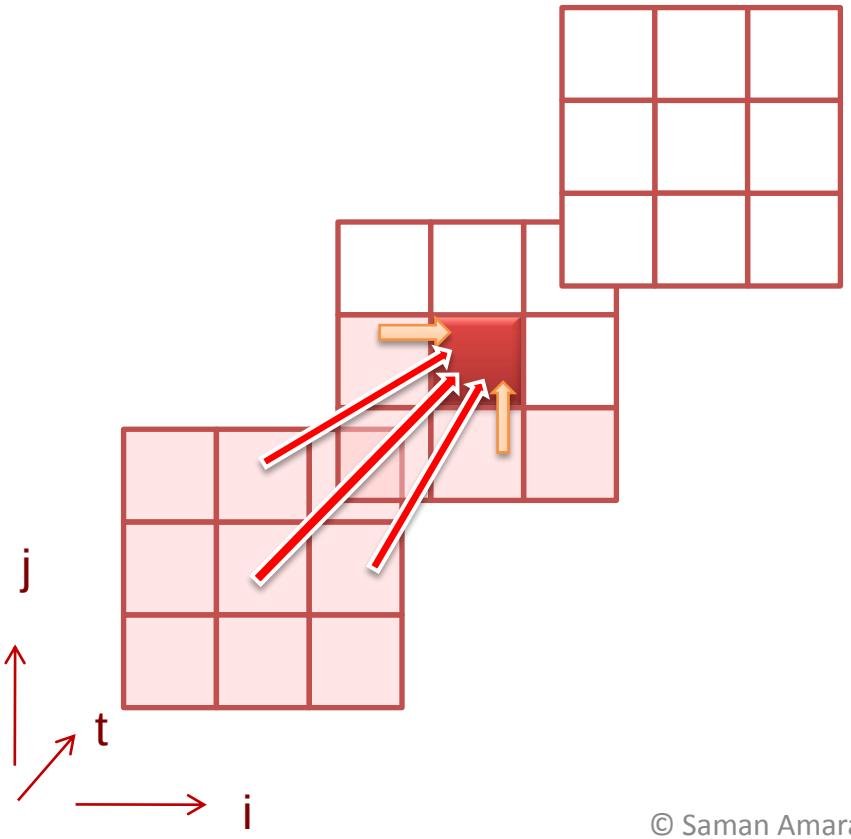
```
forall??(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

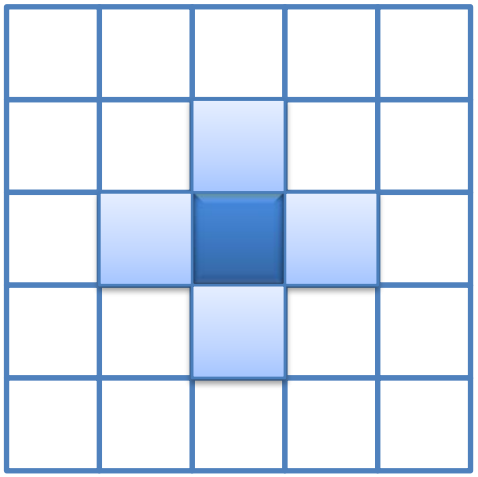




Creating a FORALL Loop

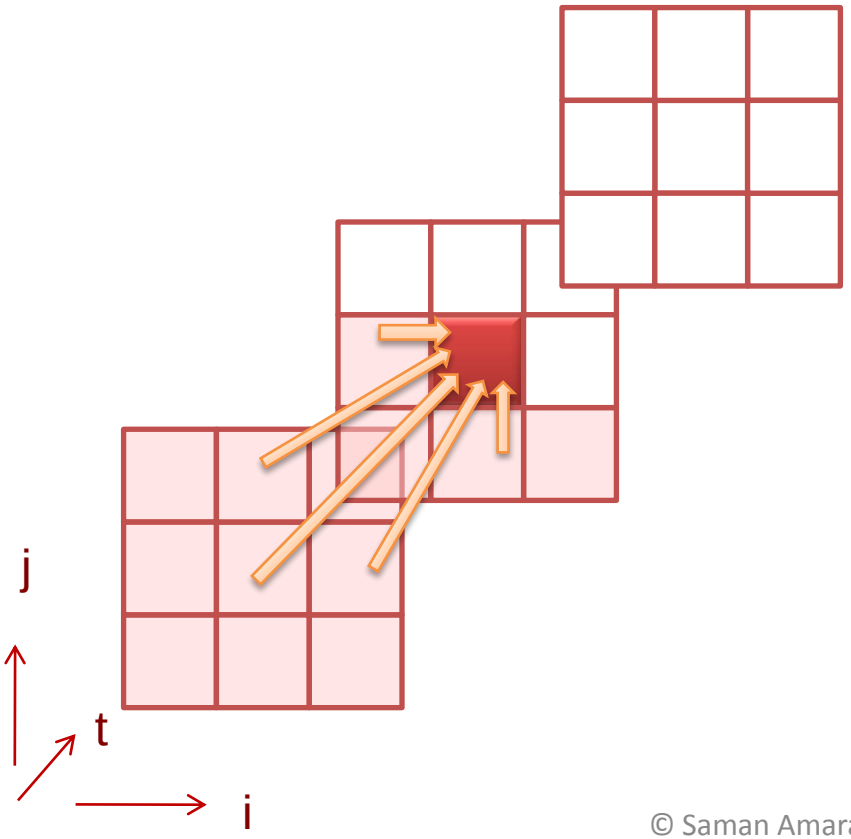
```
forall??(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

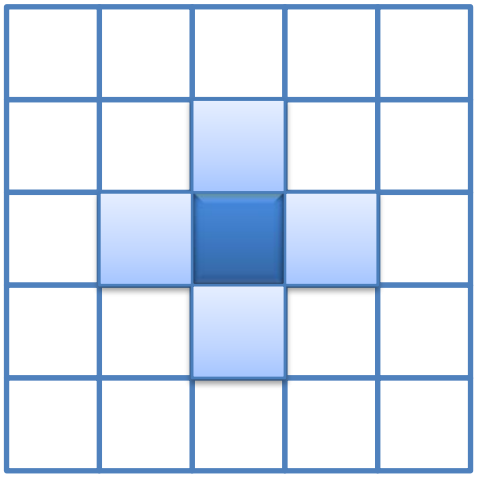




Creating a FORALL Loop

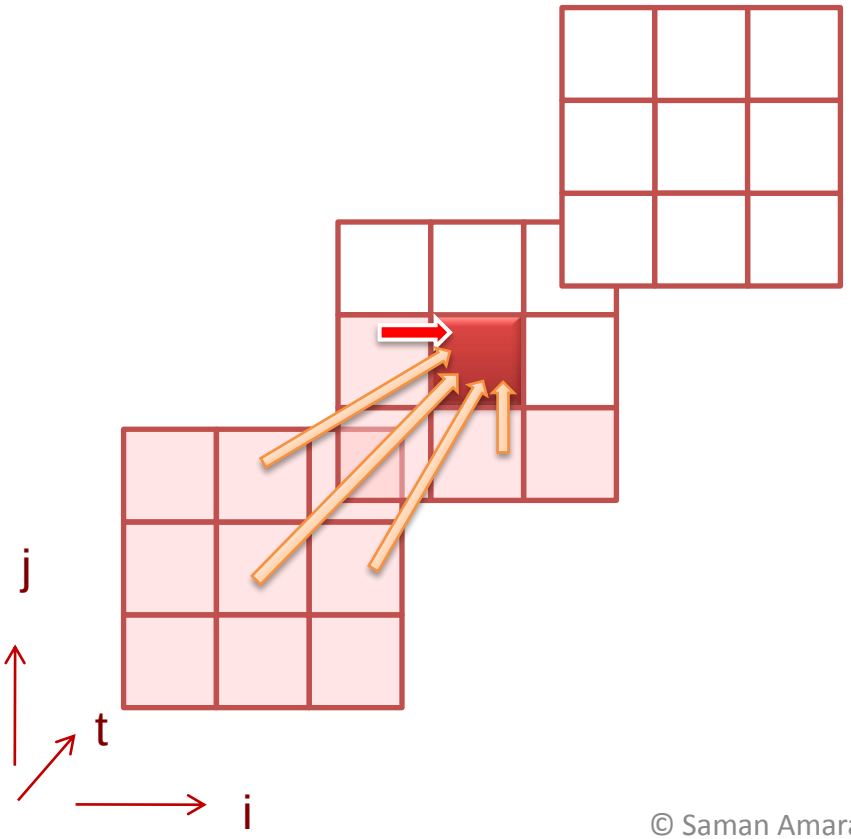
```
for(int t=1; t < steps; t++)
    forall??(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

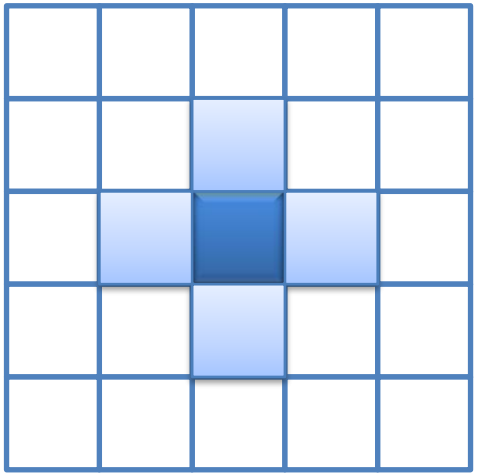




Creating a FORALL Loop

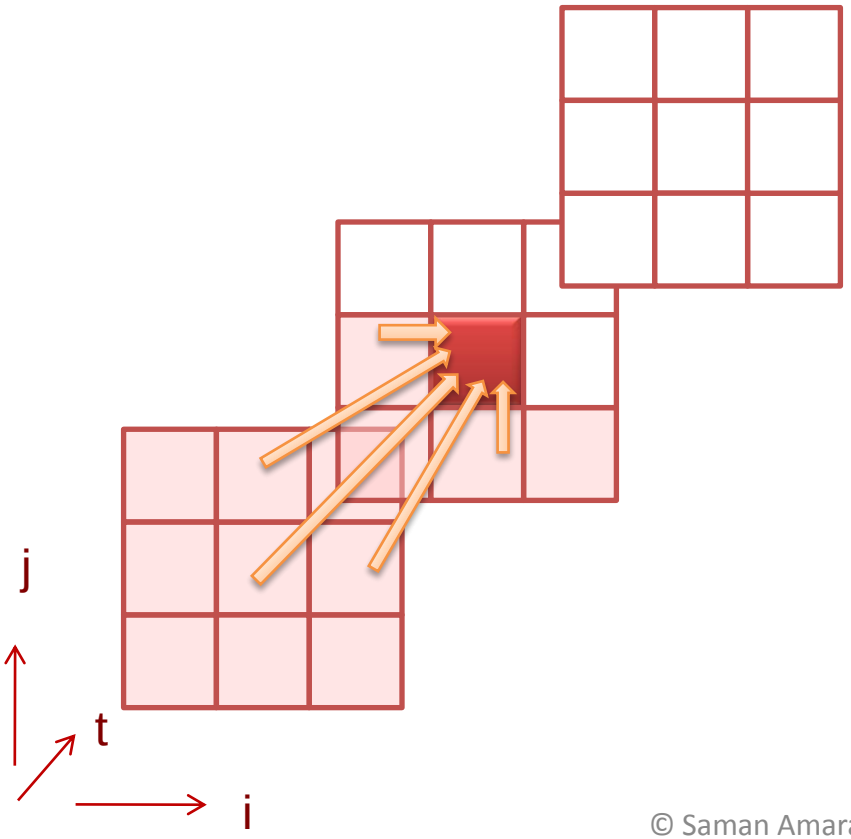
```
for(int t=1; t < steps; t++)
    forall??(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

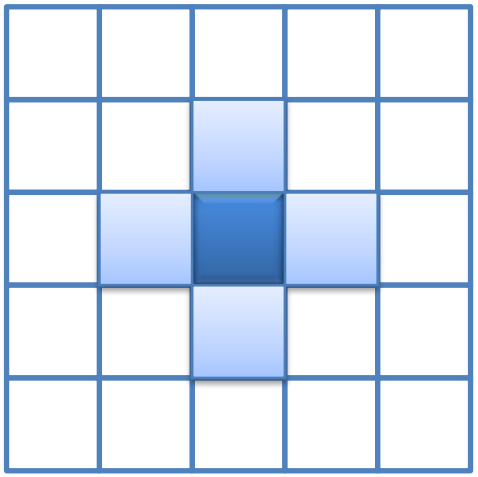




Creating a FORALL Loop

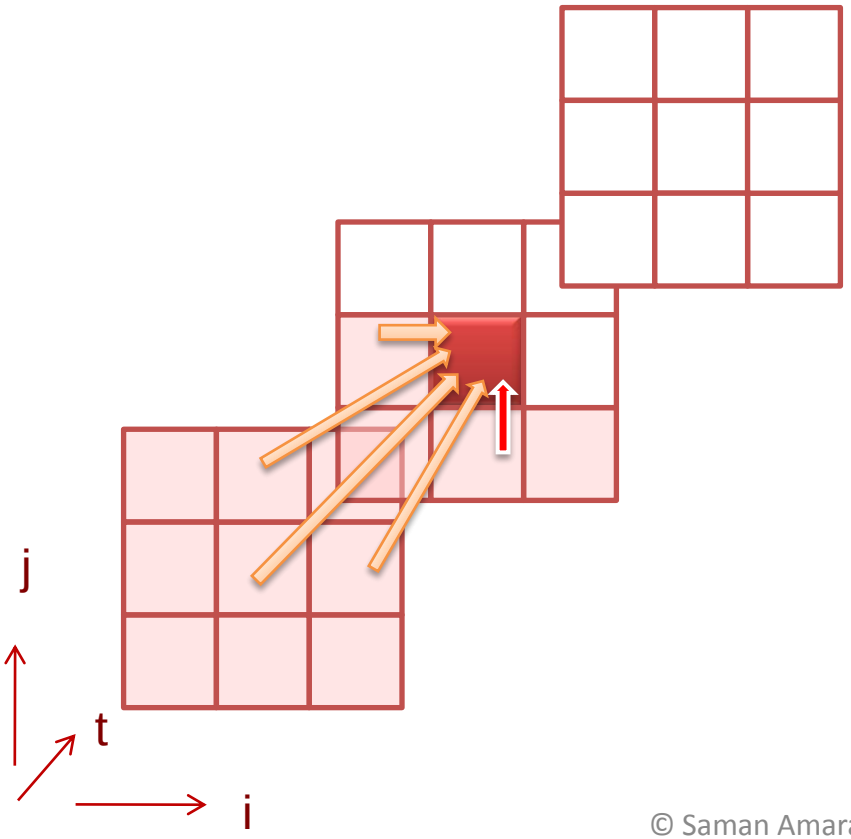
```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        forall??(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

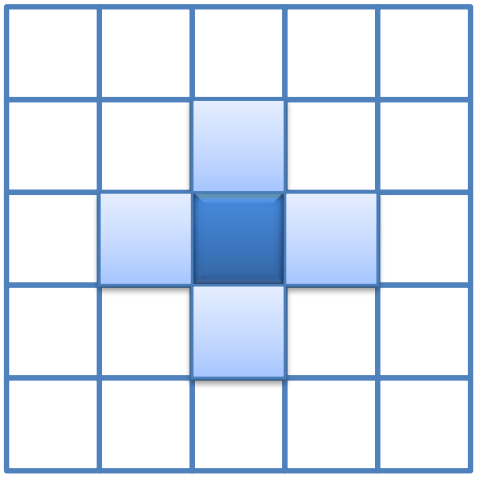




Creating a FORALL Loop

```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        forall??(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```

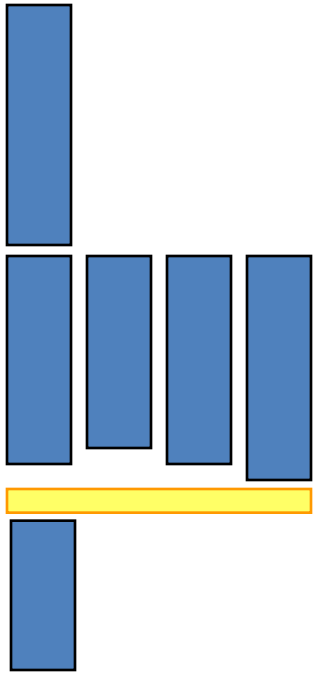




Programmer Defined Parallel Loop

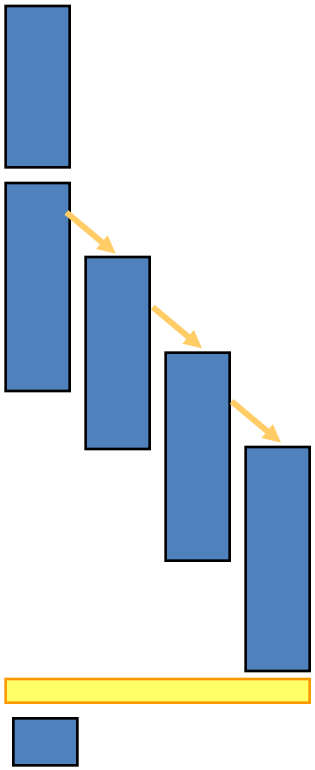
FORALL

- No "loop carried dependences"
- Fully parallel



FORACROSS

- Some "loop carried dependences"

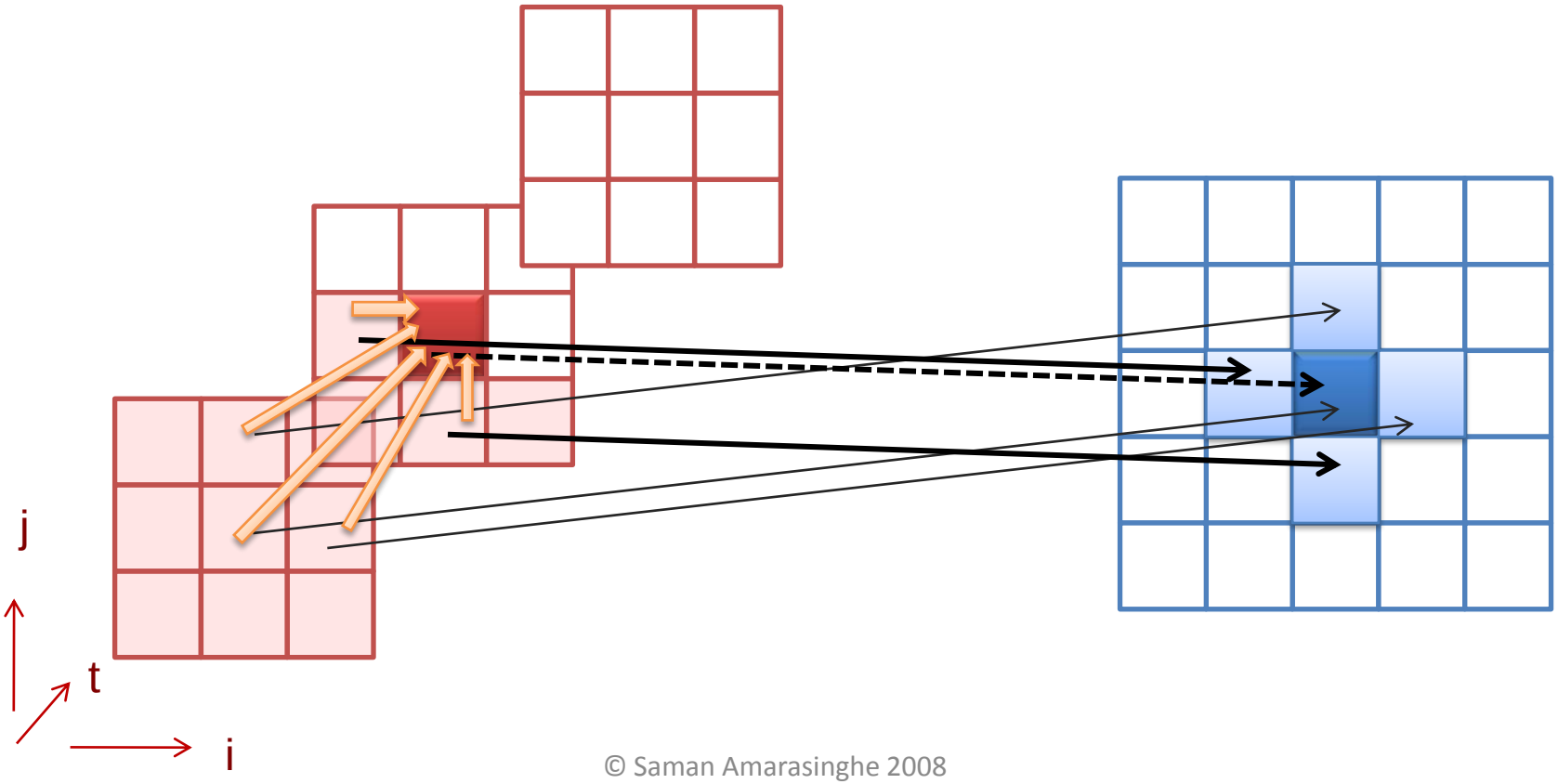


FORACROSS

```
for(int t=1; t < steps; t++) {
#pragma omp parallel for schedule(static, 1)
  for(int i=1; i < N-1; i++) {
    for(int j=1; j < N-1; j++) {
      if (i > 1)
        pthread_cond_wait(&cond_vars[i-1][j], &cond_var_mutexes[i-1][j]);
      A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
      if(I < N-2)
        pthread_cond_signal(&cond_vars[i][j]);
    }
  }
}
```

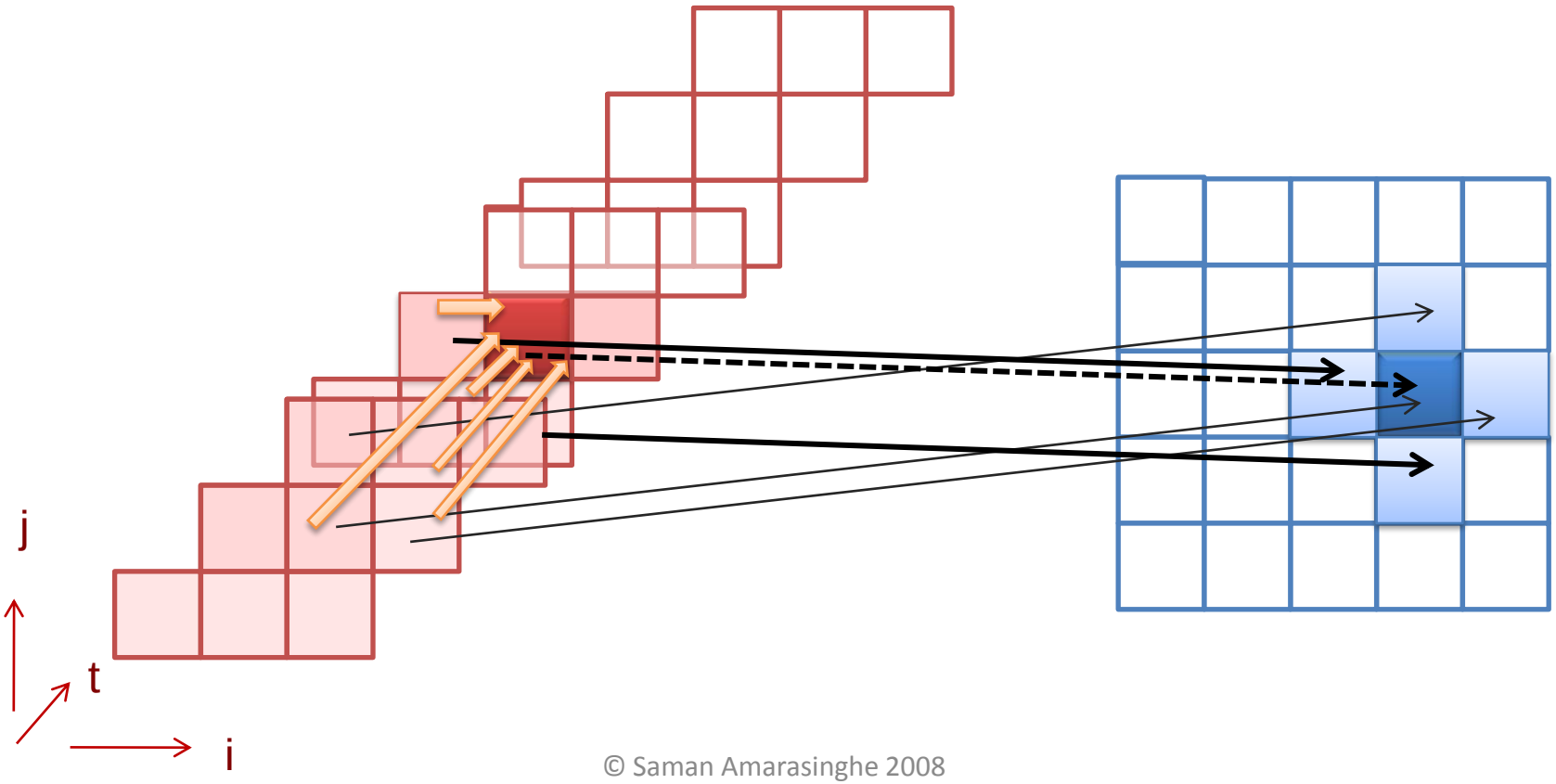
Wavefront Execution

```
for(int t=1; t < steps; t++)
    for(int i=1; i < N-1; i++)
        for(int j=1; j < N-1; j++)
            A[i][j] = (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/5
```



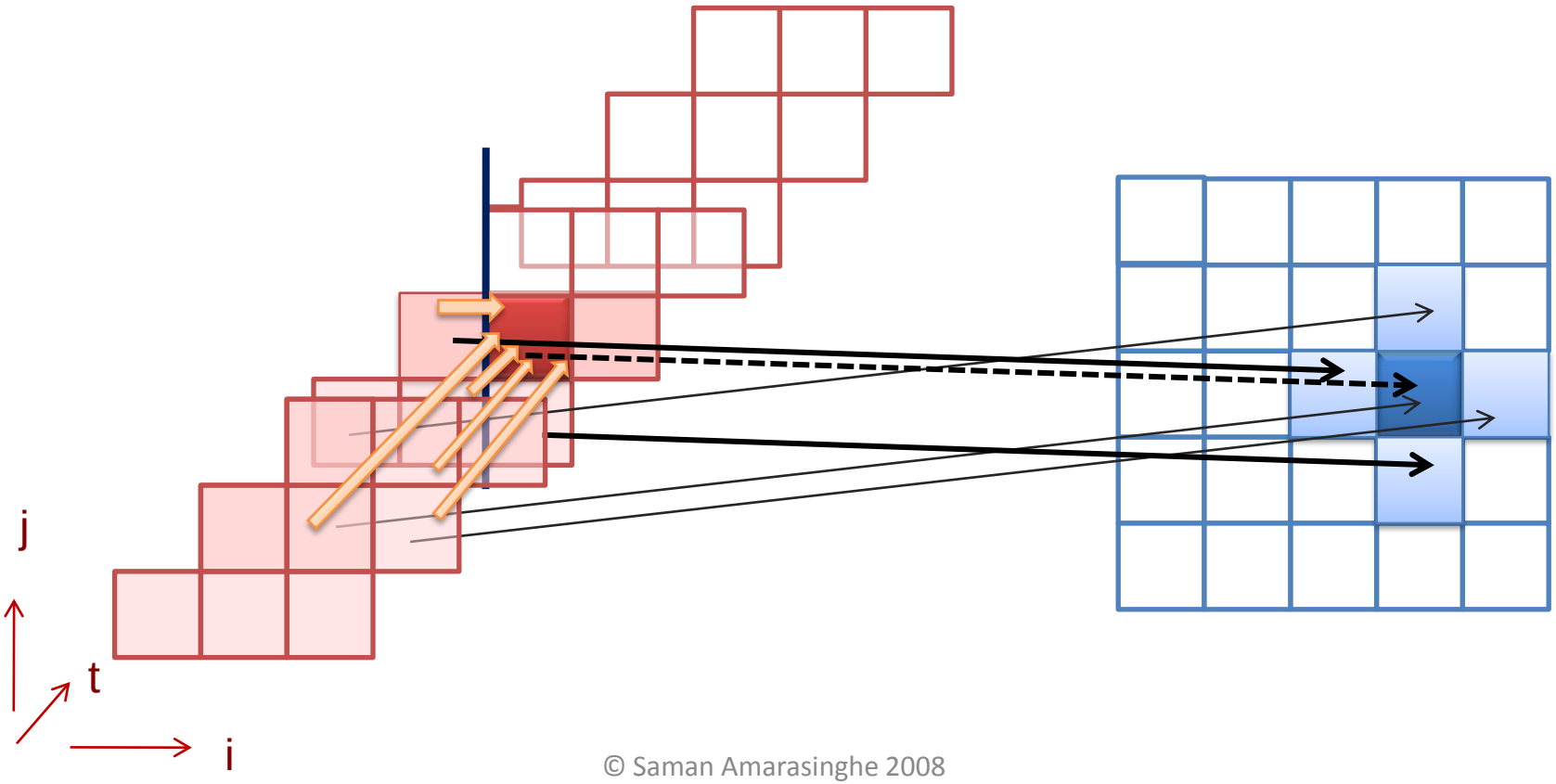
Wavefront Execution

```
for(int t=1; t < steps; t++)
    for(int i=1; i < 2*N-3; i++)
        for(int j=max(1,i-N+2); j < min(i, N-1); j++)
            A[i-j+1][j] =(A[i-j+1][j]+ A[i-j][j]+ A[i-j+2][j]+ A[i-j+1][j-1]+ A[i-j+1][j+1])/5
```



Parallelism via Wavefront

```
for(int t=1; t < steps; t++)
  for(int i=1; i < 2*N-3; i++)
    forall(int j=max(1,i-N+2); j < min(i, N-1); j++)
      A[i-j+1][j] =(A[i-j+1][j]+ A[i-j][j]+ A[i-j+2][j]+ A[i-j+1][j-1]+ A[i-j+1][j+1])/5
```



6.172 Performance Engineering of Software Systems

Fall 2010