KOÇ UNIVERSITY

Data Dependency II

Didem Unat

COMP 429/529 Parallel Programming

Data Dependence

- A ***data dependence*** is an ordering on a pair of memory operations that must be preserved to maintain correctness.

- **Question:** When is parallelization guaranteed to be safe?
- **Answer:** If there are no data dependences across reordered computations.

- **Definition: Two memory accesses** are involved in a data dependence if they may refer to the same memory location and one of the accesses is a **write.**

Data Dependence Types

Flow dependence (True dependence) (RAW)

S1: X = A+B

S2: C= X+A

Anti dependence (WAR)

S1: A = X + B

S2: X= C + D

Output dependence (WAW)

S1: X = A+B

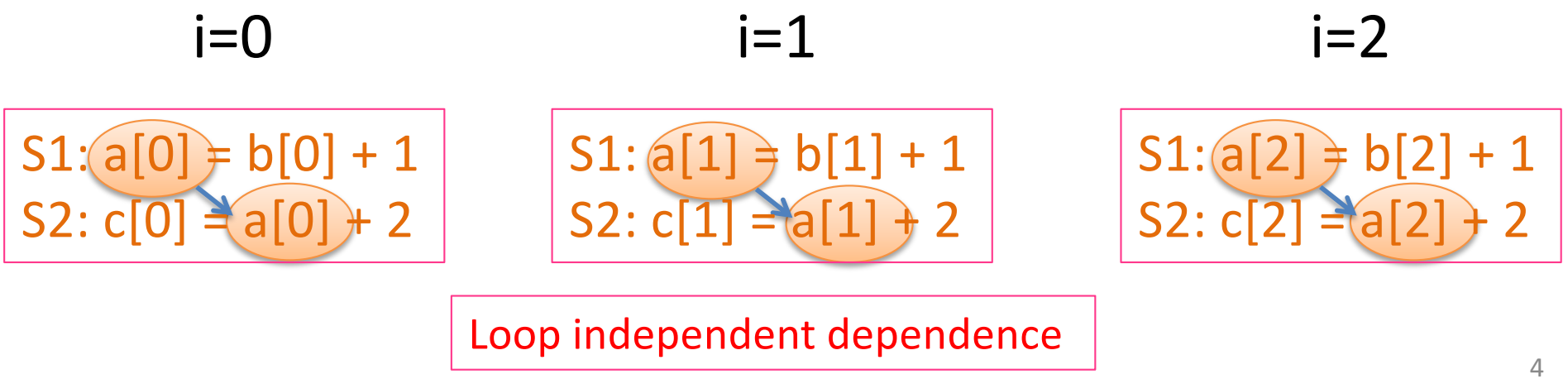
S2: X= C + D

- **Definition: Data dependence exists from a reference instance i to j iff**
  - either i or j is a write operation
  - i and j refer to the same variable
  - i executes before j

Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

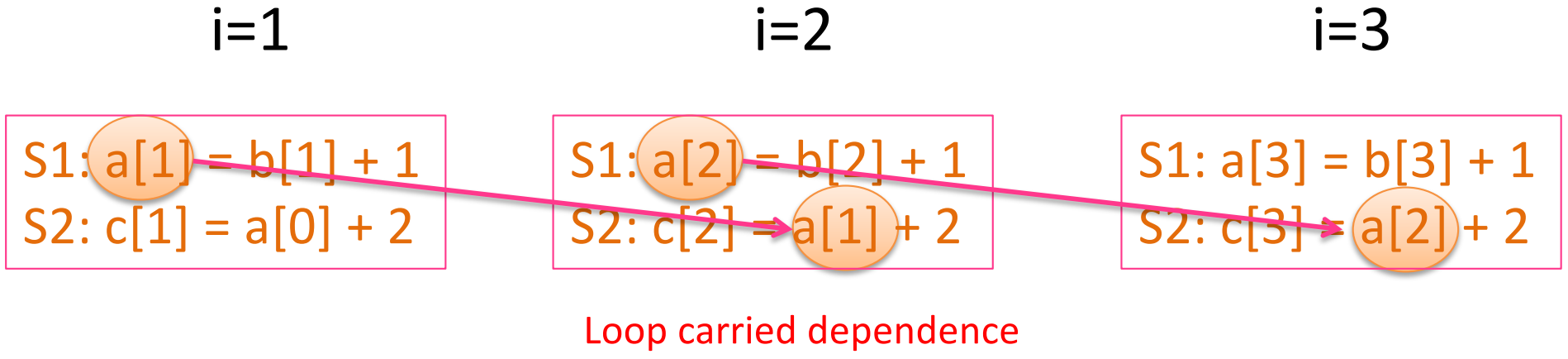
```
for (i=0; i<n; i++){
    S1    a[i] = b[i] + 1;
    S2    c[i] = a[i] + 2;
}
```



Dependences in Loops (II)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement "executions".

```
for (i=1; i<n; i++){
    S1    a[i] = b[i] + 1;
    S2    c[i] = a[i-1] + 2;
}
```



Indirect Memory Accesses

```
for (i=1; i < N; i++) {
    a[b[i]] = 2 * a[i];
}
```

- Above loop may or may not contain dependencies.
- We cannot reason about the dependencies at compile time because only at runtime we know the indices of a[ ]

- Here, the dependence cannot be determined in advance. **For safety, we have to assume** that there might be a dependence that prevents parallel execution of the loop and will not parallelize the loop.

Indirect Memory Accesses

```
for (i=1; i < N; i++) {
    a[b[i]] = 2 * a[i];
}
```

```
for (i=1; i < N; i++) {
    a[i] = 2 * a[b[i]];
}
```

```
for (i=1; i < N; i++) {
    c[i] = 2 * a[b[i]];
}
```

- Safe to parallelize?

Loop Transformations

- Often loops contain a few statements that cannot be executed in parallel and many statements that can be executed in parallel.

```
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];                  /* S1 */
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;     /* S2 */
    y[i] = z[i] - x[i];                       /* S3 */
}
```

Assuming that arrays x, y, w, a, and z do not overlap (no aliasing), statements S1 and S3 can be parallelized but statement S2 cannot be.

1. Loop Distribution

```
/* L1: parallel loop */
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];                    /* S1 */
    y[i] = z[i] - x[i];                         /* S3 */
}
/* L2: sequential loop */
for (i=0; i < n; i++) {
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* S2 */
}
```

- After the loop distribution transformation, loop L1 does not contain any statements that prevent the parallelization of the loop and may be executed in parallel. Loop L2, however, still has a non-parallelizable statement from the original loop.

2. Loop Fusion

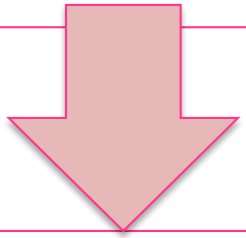
- **Loop fusion** combines several loops into a single parallel loop, and thus increase the granularity of the loop.

```
/* L1: parallel loop */
for (i=0; i < N; i++) {
    a[i] = a[i] + b[i];          /* S1 */
}
/* L2: parallel loop */
for (i=0; i < N; i++) {
    b[i] = a[i] * d[i];          /* S2 */
}
```

- It should be safe to fuse the loops. Check for dependencies

The Two Loops Fused

```
/* L1: parallel loop */
for (i=0; i < N; i++) {
    a[i] = a[i] + b[i];          /* S1 */
}
/* L2: parallel loop */
for (i=0; i < N; i++) {
    b[i] = a[i] * d[i];          /* S2 */
}
```



```
/* L3: parallel loop */
for (i=0; i < N; i++) {
    a[i] = a[i] + b[i];          /* S1 */
    b[i] = a[i] * d[i];          /* S2 */
}
```

What is the advantage of loop fusion?

What is the disadvantage of loop fusion?

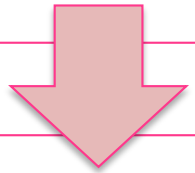
Unsafe Loop Fusion for Parallelism

```
/* L1: parallel loop */
for (i=0; i < N; i++) {
    a[i] = a[i] + b[i];       /* S1 */
}
/* L2: loop with data dependence */
for (i=0; i < N; i++) {
    a[i+1] = a[i] * d[i];     /* S2 */
}
```

- If the loops are fused, a data dependence is created from statement S2 to S1. In effect, the value of a[i] in the right hand side of statement S1 is computed in statement S2.

3. Loop Interchange

```
for (i=0; i <n; i++) {
    for (j=0; j <n; j++) {
            a[j][i+1] = 2.0*a[j][i-1];
    }
}
```



```
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
            a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

- It is generally more profitable to parallelize the outermost loop in a nest of loops, since the overheads incurred are small.
- The resulting loop incurs an overhead of parallel work distribution only once, while previously, the overhead was incurred *n* times.

Aliasing and Parallelization

```
void copy(float a[], float b[], int n) {
    int i;
    for (i=0; i < n; i++) {
            a[i] = b[i]; /* S1 */
    }
}
```

- Since variables a and b are parameters, it is possible that a and b may be pointing to overlapping regions of memory

```
copy (x[10], x[11], 20);
```

In the called routine, two successive iterations of the copy loop may be reading and writing the same element of the array x if run parallel.

Restrict Keyword

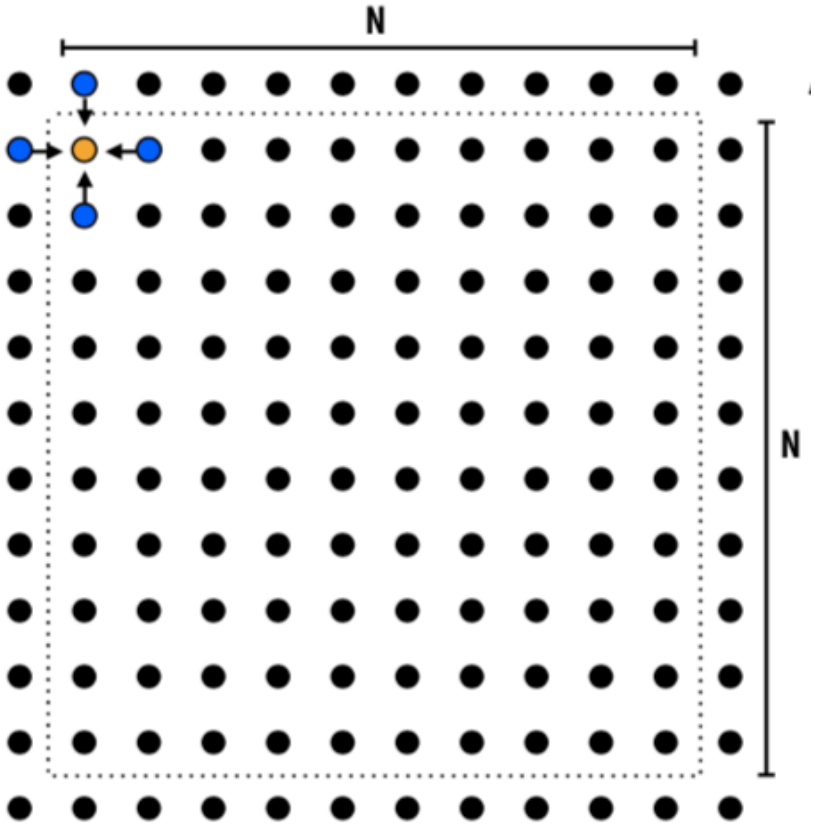
- Restricted pointers are used to specify pointers which designate distinct objects so that the compiler can perform pointer alias analysis and perform optimizations if it is safe to do so.

```
void copy(double * restrict a, double * restrict b, int n)
```

-restrict option as a compiler flag

Example: A 2D-grid based solver

- Solve partial differential equation (PDE) on (N+2) x (N+2) grid
- Iterative solution
  - Perform Gauss-Seidel sweeps over grid until convergence

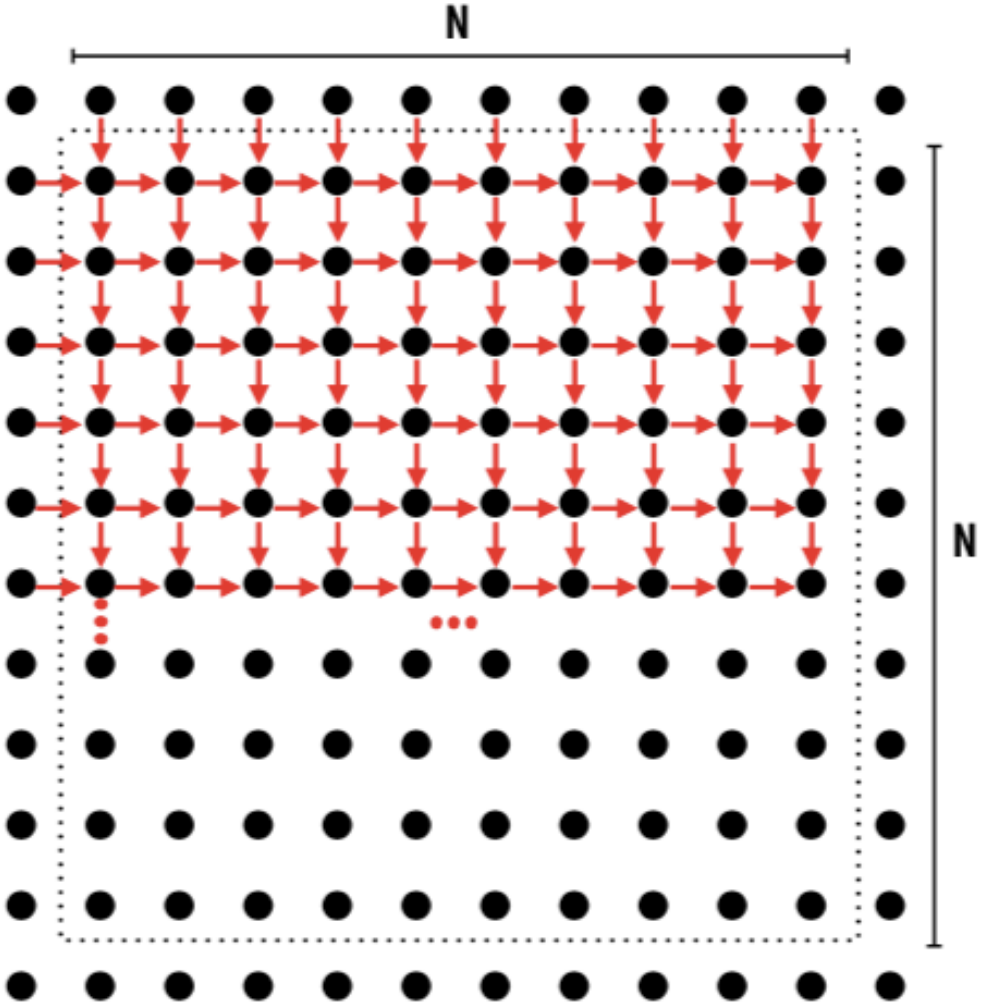



```
A[i,j] = 0.2 *
         (A[i,j] +
          A[i,j-1] + A[i-1,j] +
          A[i,j+1] + A[i+1,j]);
```

Grid Solver Algorithm

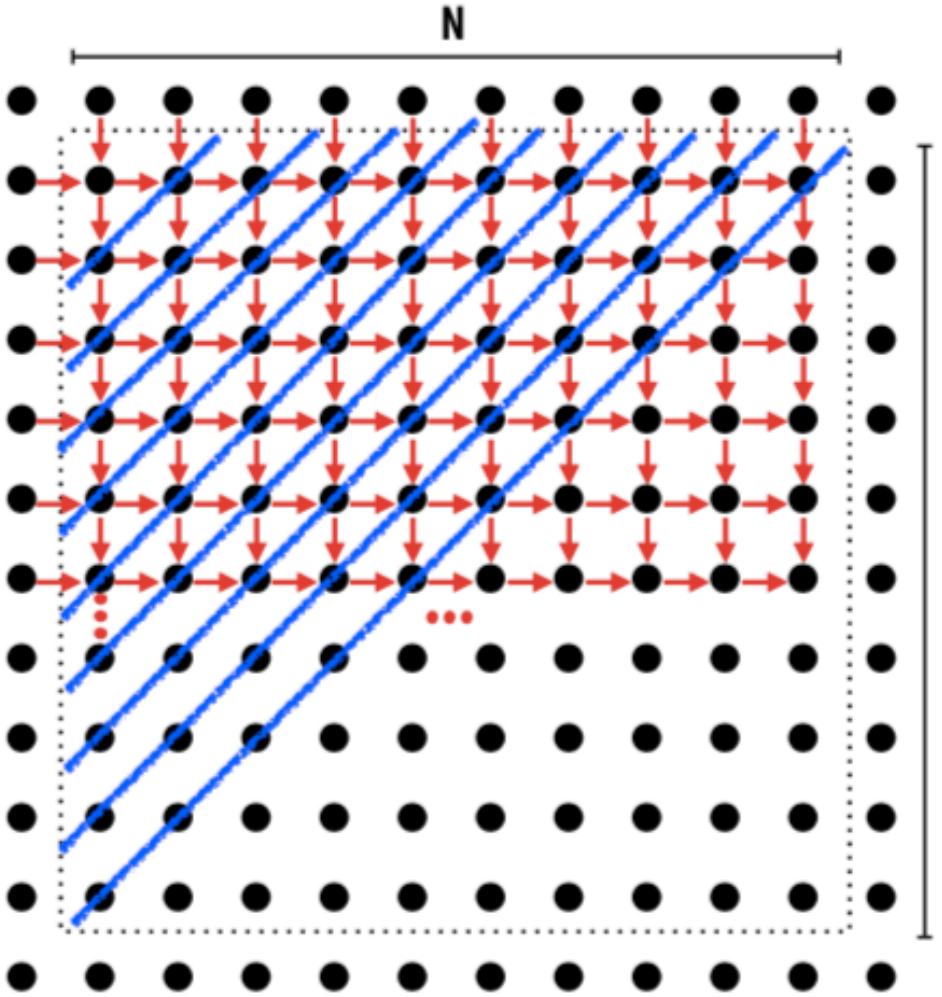
```
const int n;
float* A;

void solve(float* A) {

  float diff, prev;
  bool done = false;
  while (!done) {
      // assume allocated to grid of (N+2) x (N+2) elements
      diff = 0.f;
      // outermost loop: iterations
      for (int i=1; i<n i++) {// iterate over non-border points of grid
        for (int j=1; j<n; j++) {
            prev = A[i,j];
            A[i,j] = 0.2f * (A[i,j] + A[i,j-1] + A[i-1,j] +
                             A[i,j+1] + A[i+1,j]);
            // compute amount of change
            diff += fabs(A[i,j] - prev);
      }}
      // quit if converged
      if (diff/(n*n) < TOLERANCE)
        done = true;
  }
}
```

Identify Dependencies



- Each row element depends on element to left.
- Each row depends on previous row.
- Note: the dependencies illustrated on this slide are element data dependencies in one iteration of the solver (in one iteration of the "while(!done)" loop)

Identify Dependencies

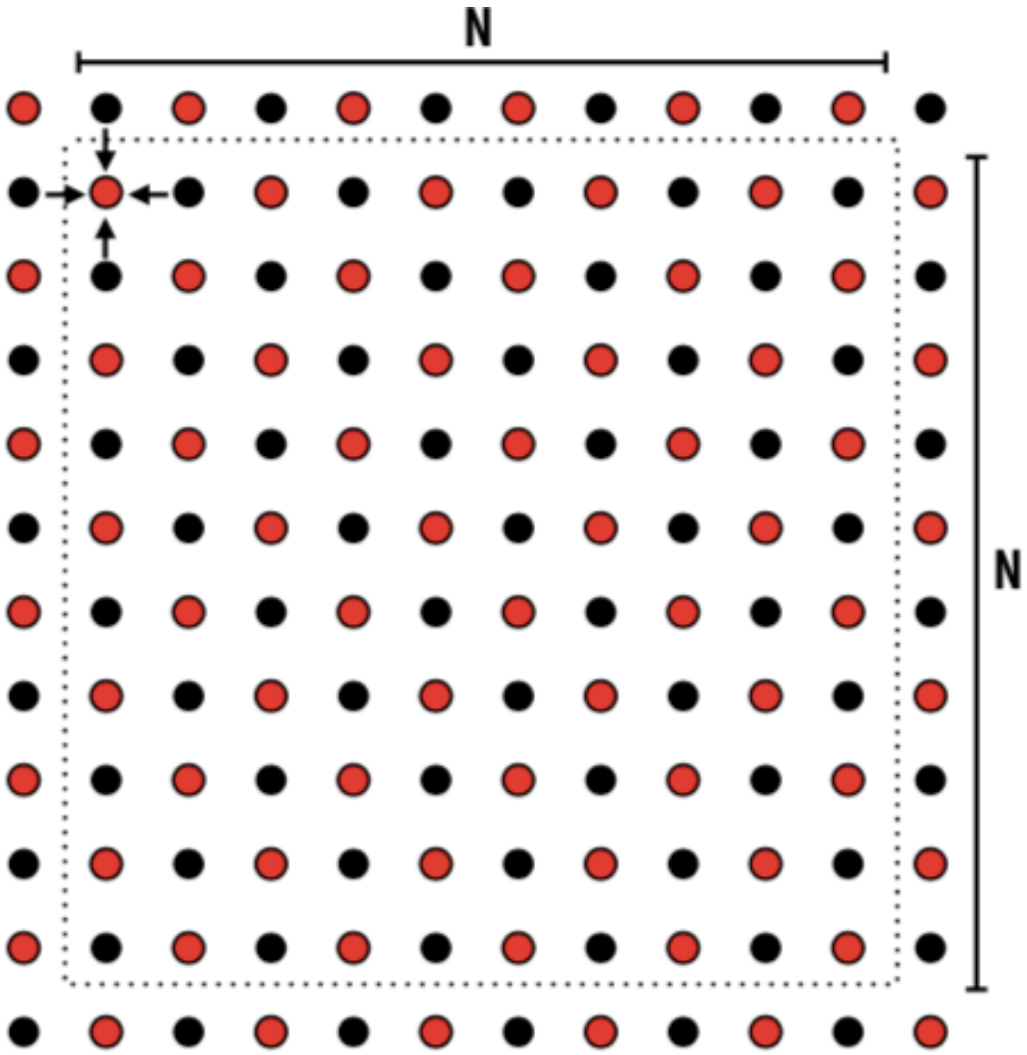


- There is independent work along the diagonals!
- Good: parallelism exists!
  - An implementation strategy:
    - Partition grid cells on a diagonal into tasks
    - Update values in parallel
    - When complete, move to next diagonal
- Bad: independent work is hard to exploit
  - Not much parallelism at beginning and end of computation.
  - Frequent synchronization (after completing each diagonal)

Change the algorithm

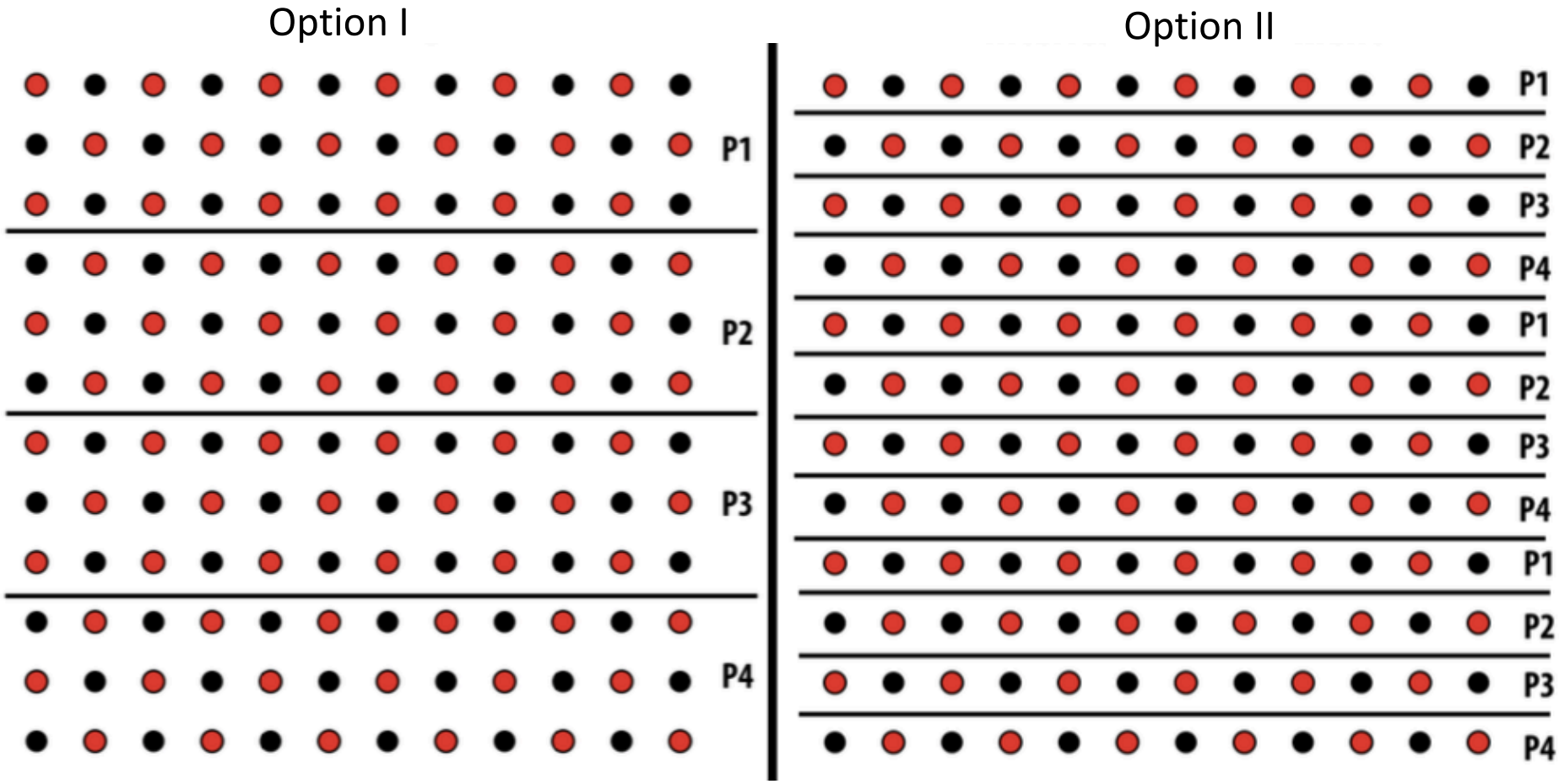
- Idea: improve performance by changing the algorithm to one that is more amenable to parallelism
  - Change the order grid cell cells are updated
  - New algorithm iterates to same solution (approximately), but converges to solution differently
  - Note: floating-point values computed are different, but solution still converges to within error threshold
  - Yes, we need domain knowledge of Gauss-Seidel method for solving a linear system to realize this change is permissible for the application

Red-black Coloring



- Update all red cells in parallel
- When done updating red cells , update all black cells in parallel (respect dependency on red cells)
- Repeat until convergence

Possible Partitioning

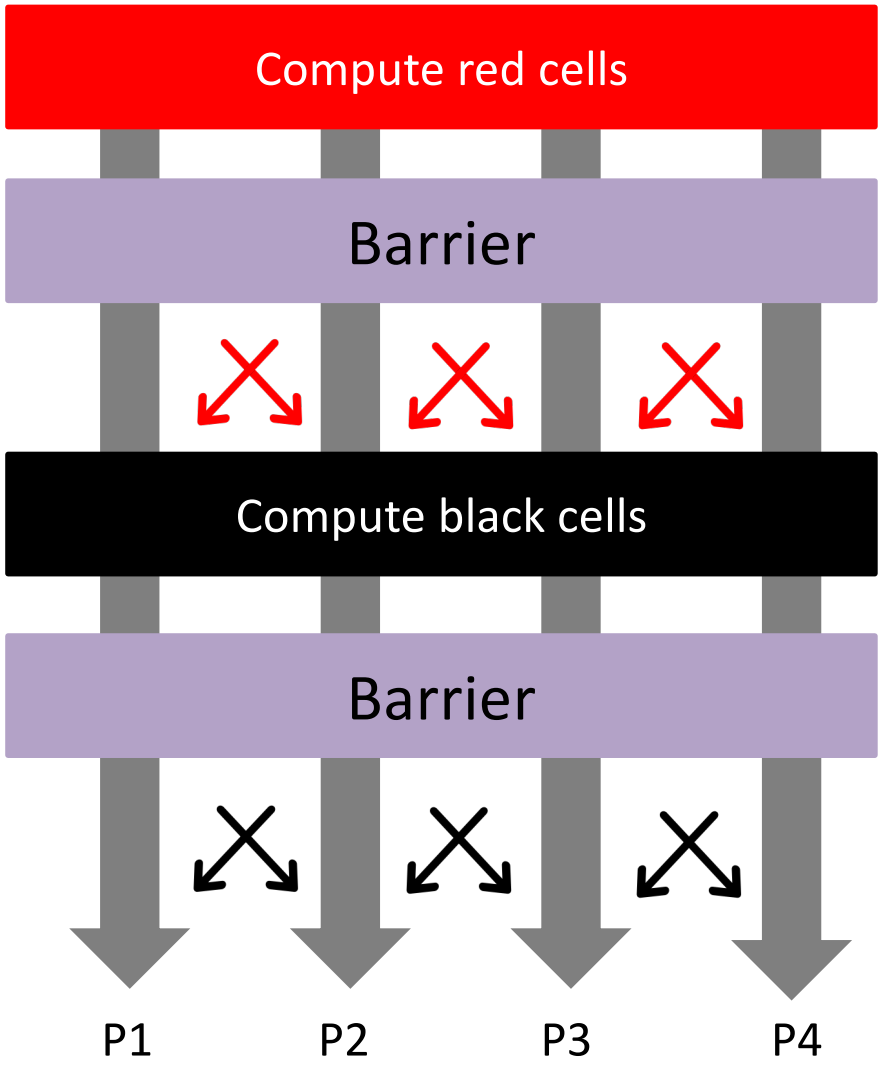


- You can implement these partitioning with OpenMP Loop scheduling

Barriers

- Perform red update in parallel
- Wait until all threads done with update
  - Synchronize!!!
- Perform black update in parallel
- Wait until all threads done with update
  - Synchronize!!!
- Repeat

Programmer is responsible for synchronization



Lab 3: Stream Benchmark

- Stream is a simple benchmark that mesuares the memory bandwidth of your system

Stream Benchmark

```
for (j=1; j < N; j++) {
    c[j] =  a[j];
}
```

- Copy

```
for (j=1; j < N; j++) {
    b[j] =  scalar* c[j];
}
```

- Scale

```
for (j=1; j < N; j++) {
    c[j] =  a[j]+b[j];
}
```

- Add

```
for (j=1; j < N; j++) {
    a[j] =  b[j]+ scalar* c[j];
}
```

- Triad

Lab 3: Stream Benchmark

- Stream is a simple benchmark that mesuares the memory bandwidth of your system
- ssh username@login.kuacc.ku.edu.tr
- Copy the lab from here to your home directory

  cp –r /kuacc/users/dunat/COMP429/OpenMP/Labs/Lab3-stream

  Copy the entire folder!

- Request time in an interactive queue
  - srun -N 1 -n 16 -p short --time=00:30:00 --pty bash

Lab 3: Stream Benchmark

- TODO 1:
  - Compile and run
  - What is the bandwidth rate?

- TODO 2:
  - On command line, type

  export OMP_NUM_THREADS=1

  Rerun the code, what is the bandwidth rate? Why?

Lab 3: Stream Benchmark

- TODO 3
  - Search for TODO 3 in stream.c
  - Comment out line 268
  - This loop initializes the arrays used in the benchmark
  - Compile and rerun with 16 threads
    - export OMP_NUM_THREADS=16
  - Why do you observe?

Serial vs. Parallel Initialization

- Stream Benchmark
  - https://github.com/jeffhammond/STREAM
- Lessons learned
  - Parallel initiation is important
  - Need to use multiple threads to sustain full memory bandwidth

Performance Tips

Tips are not only for OpenMP, for all shared memory programming models

1. Number of Parallel Regions/Thread Creations
2. First-touch policy
3. False sharing

- Note that these tips are not related to correctness of your program but should only affect the performance

1. Parallel Regions

```
#pragma omp parallel
{

}
#pragma omp parallel for
for (i=0; i<size; i++) {

}
#pragma omp parallel {


}
```

```
#pragma omp parallel
{


}
```

- Avoid creating too many parallel regions if you can
- Once the threads are created, you want to use them as much as possible.

2.Initialization – First Touch Policy

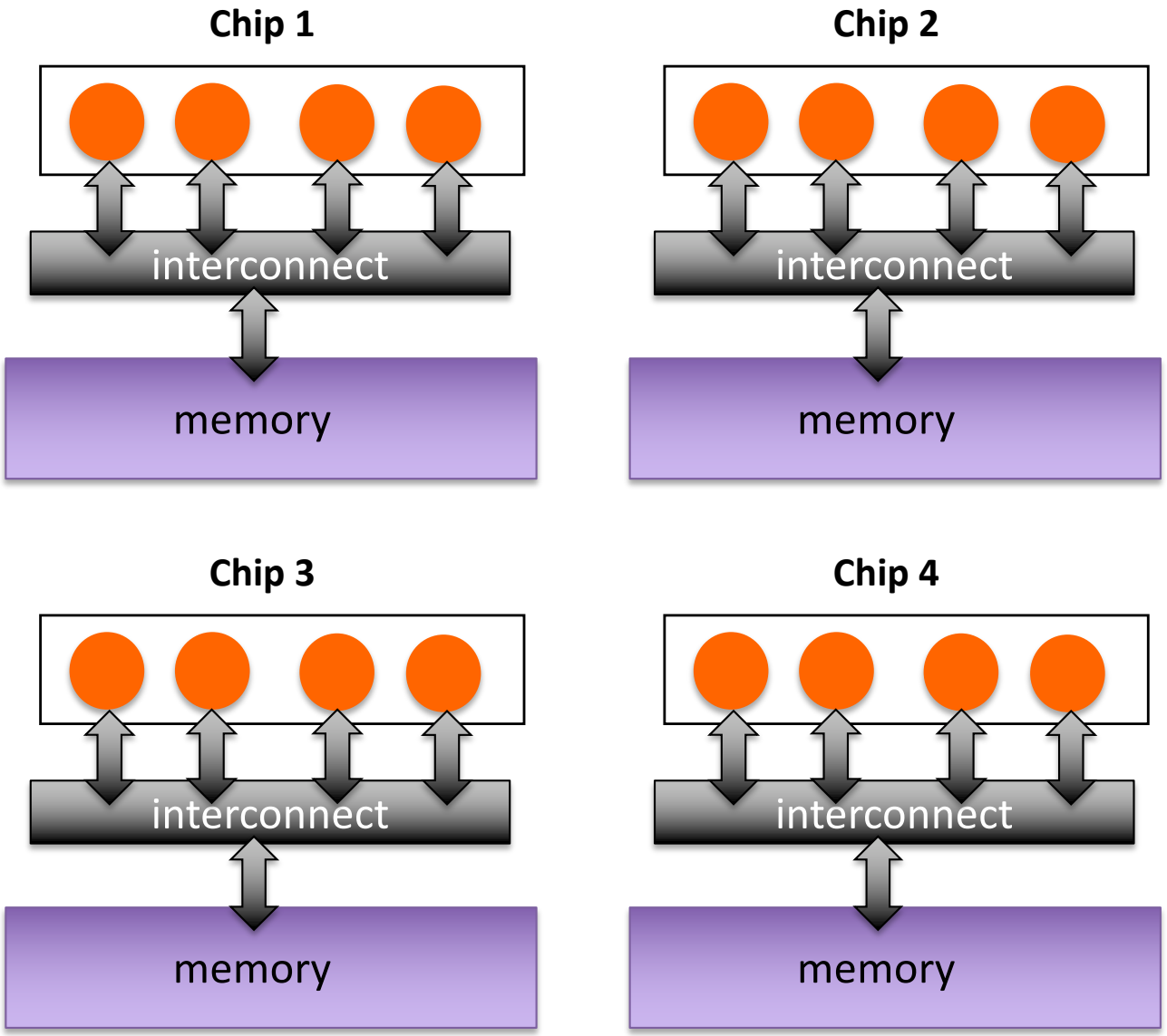
- Master thread can allocate the memory
- But initialization should be done in parallel so that data locality is ensured
- Data is close to the thread that uses it

```
for (i=0; i<N; i++)
    A[i] = 1.0; //initialization

...

#pragma omp parallel
{
   //calculation on A
}
```

VS

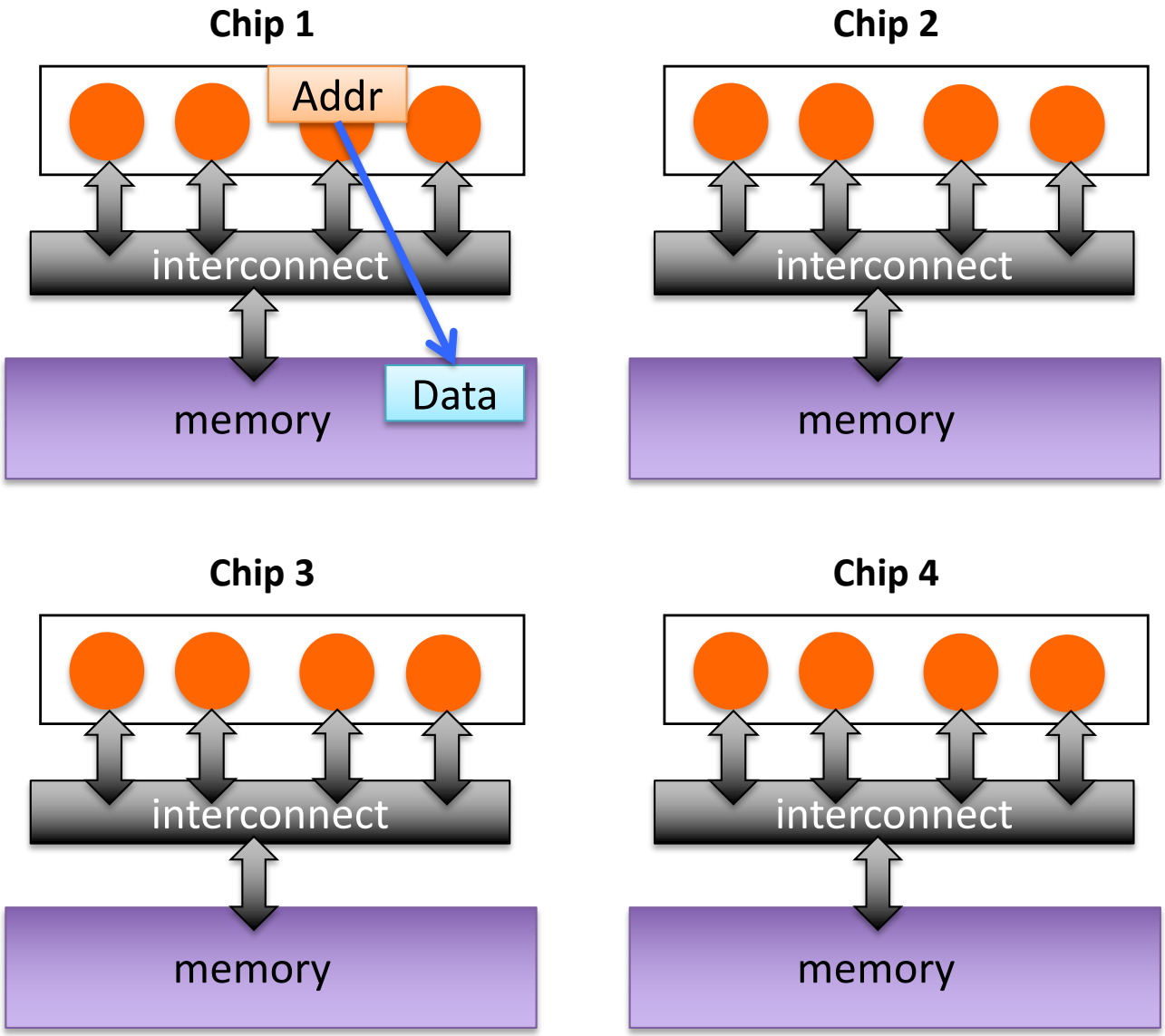
```
#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = 1.0; //initialization

. . .
#pragma omp parallel
{
   //calculation on A
}
```

NUMA



- 4 NUMA nodes
  - Four sockets

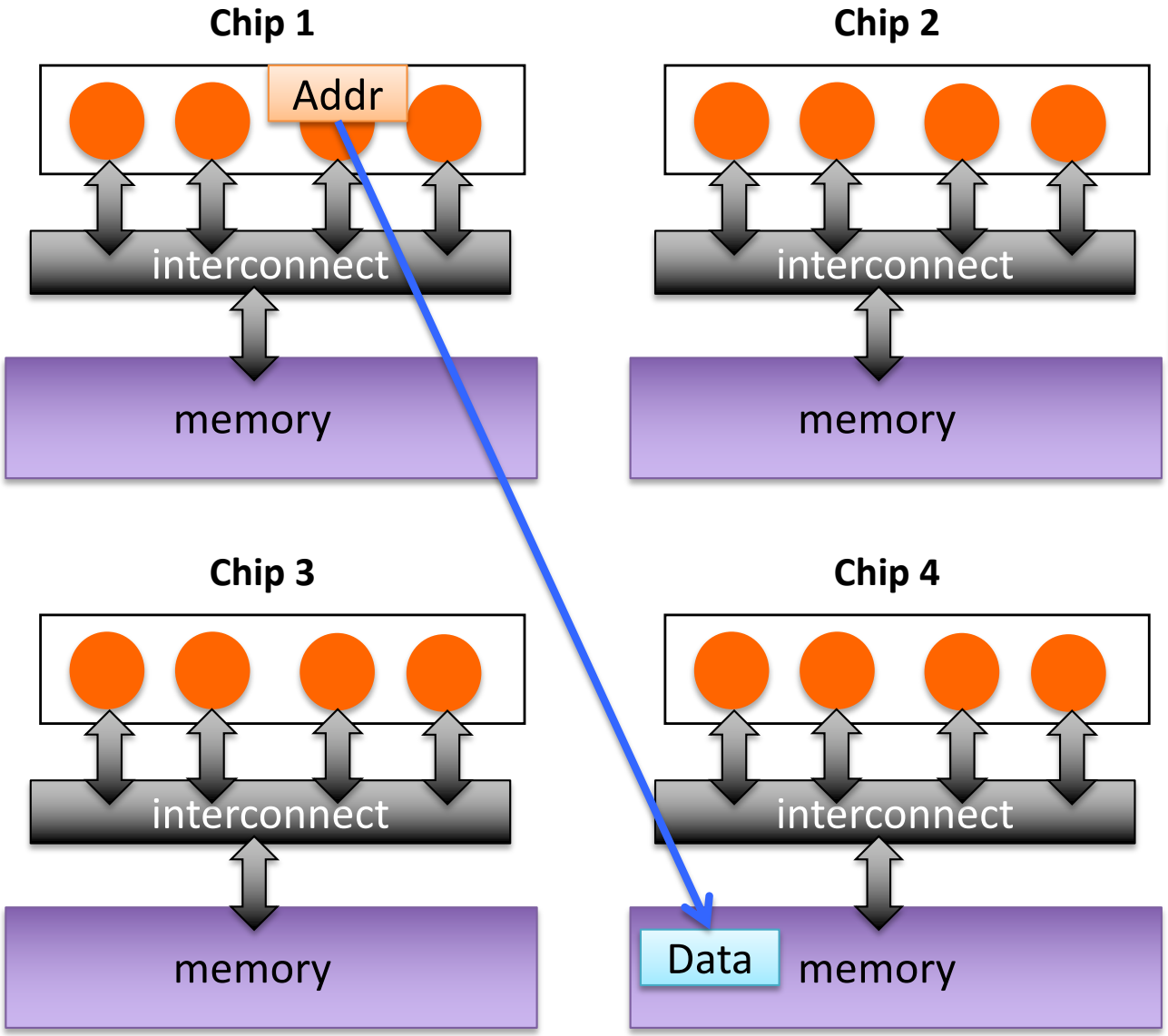
NUMA



- Accessing local memory is fast!!

- 4 NUMA nodes
  - Four sockets

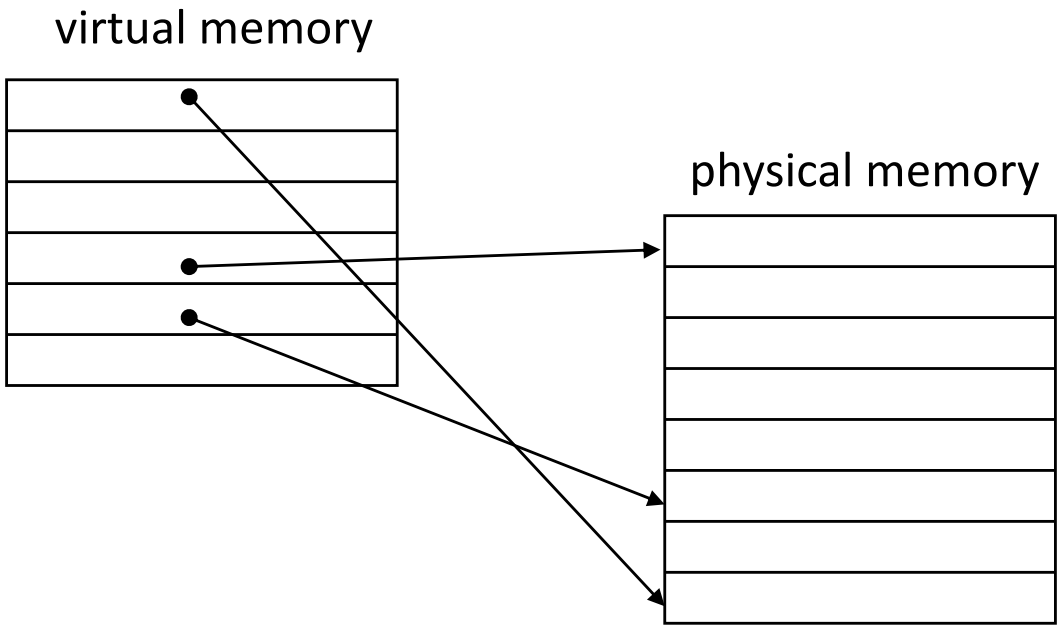
NUMA



- Accessing remote memory is slow! Even if remote memory is also shared
- 4 NUMA nodes
  - Four sockets

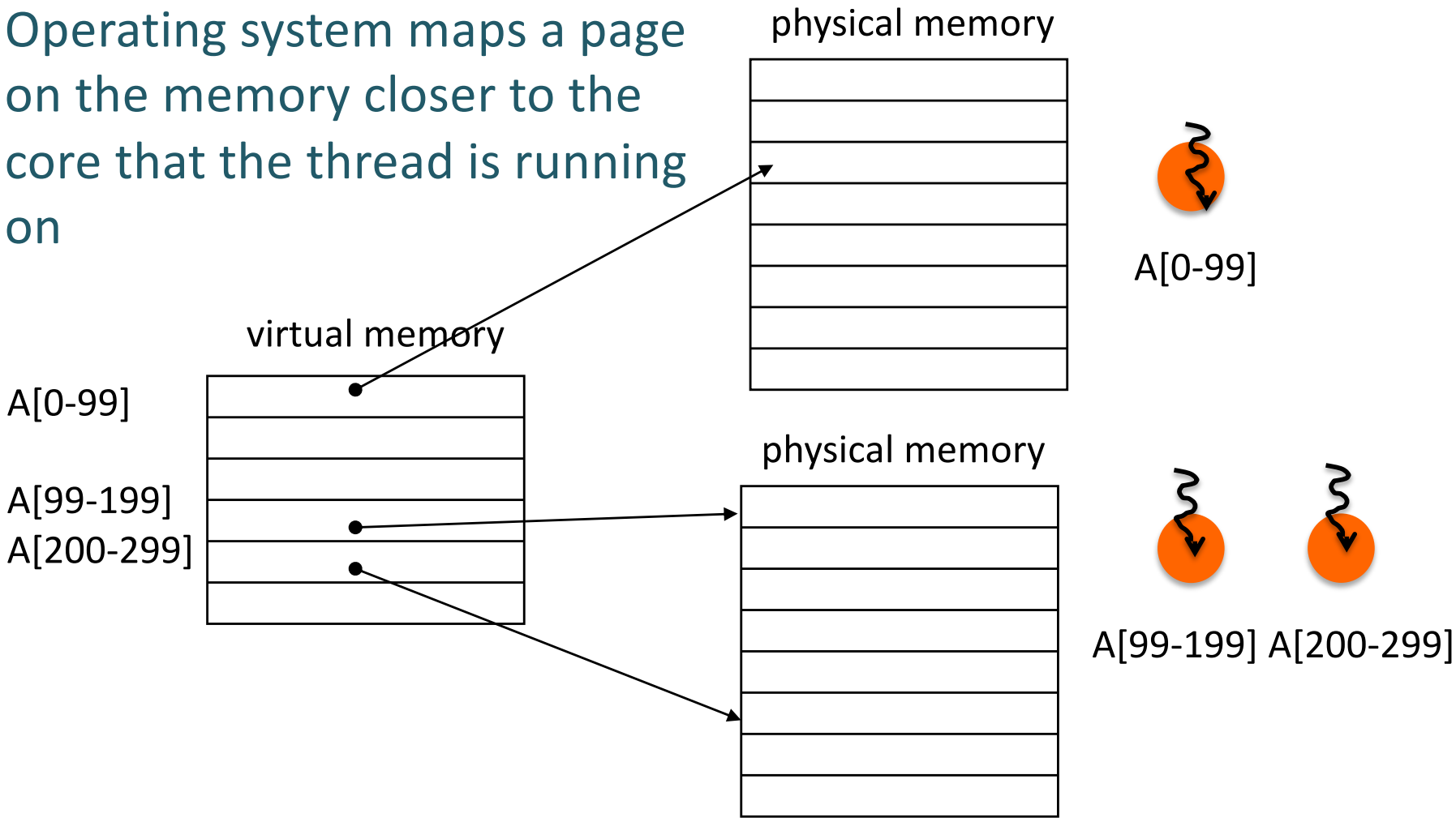
Memory Allocation

- Memory is allocated when it is first touched!
  - Call to malloc() only creates pages in virtual memory
  - Virtual memory to physical memory mapping is performed when a process/thread first touches to those pages



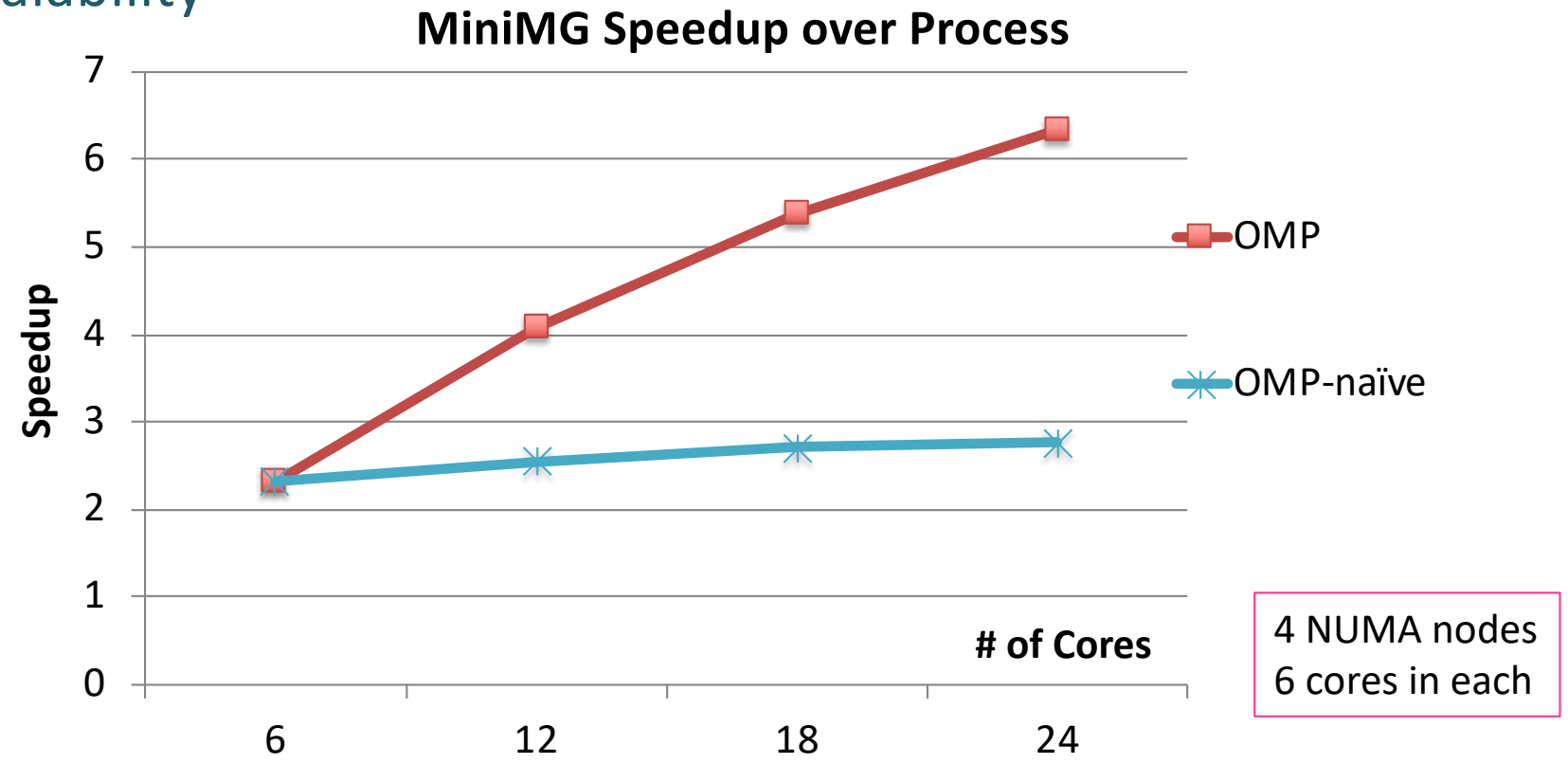
First Touch Policy on NUMA

- Operating system maps a page on the memory closer to the core that the thread is running on



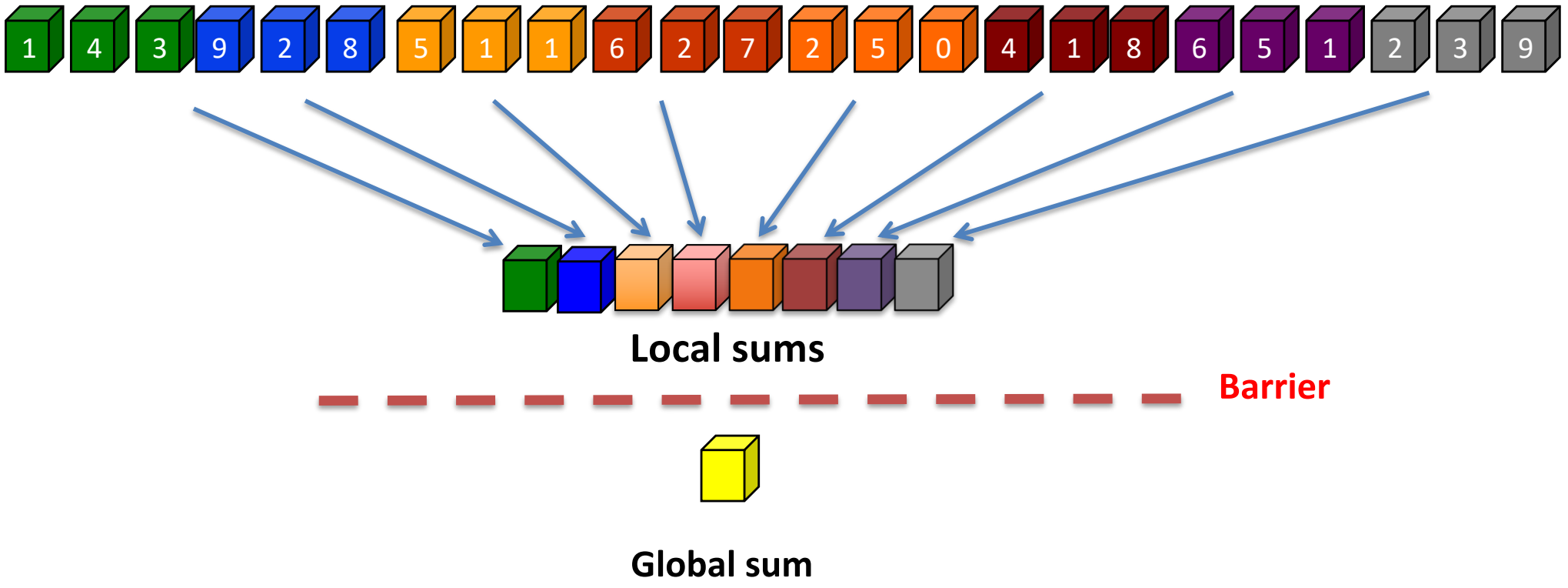
Scalability

- Unless pages are dynamically migrated (by the Operating system or by the advanced programmer) the initial mapping does not change
- Not following first-touch policy (naïve) can seriously inhibit scalability



Summing Numbers in Shared Memory

- Ensure all the local sums are ready (all the threads are done calculating their local sums)

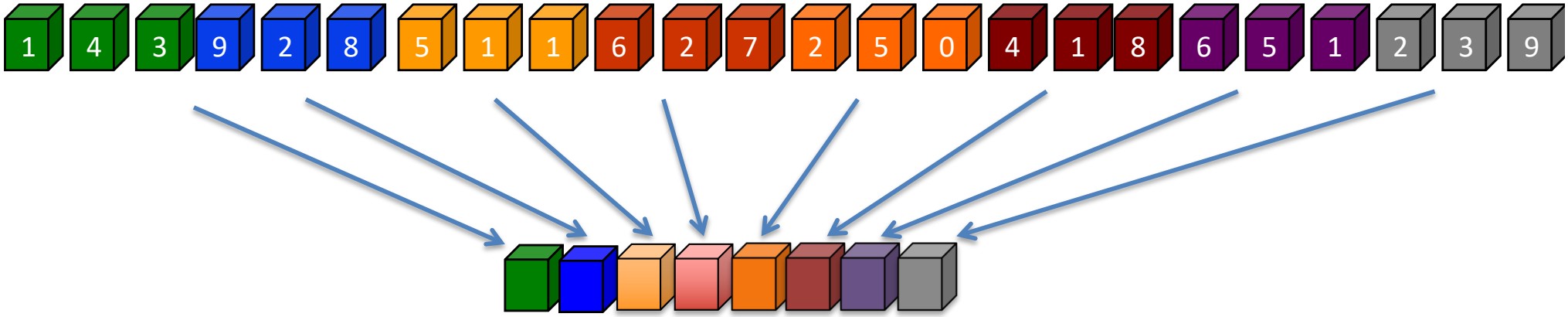


Version 5: Add a barrier

- Master waits for others to finish

```
int items_per_task = n/t;

shared int my_sum[t]; //number of threads
int start = thread_id * items_per_task;

for (i=start; i<start + items_per_task; i++)   {
     my_x = Compute_next_value(…);
     my_sum[thread_id] += my_x;
}

synchronize_threads(); // barrier for all participating threads

if (thread_id == 0 ) //master thread
{
   sum = my_sum[0];
    for(i=1; i< t; i++) sum+ = my_sum[i];
}
```

False Sharing

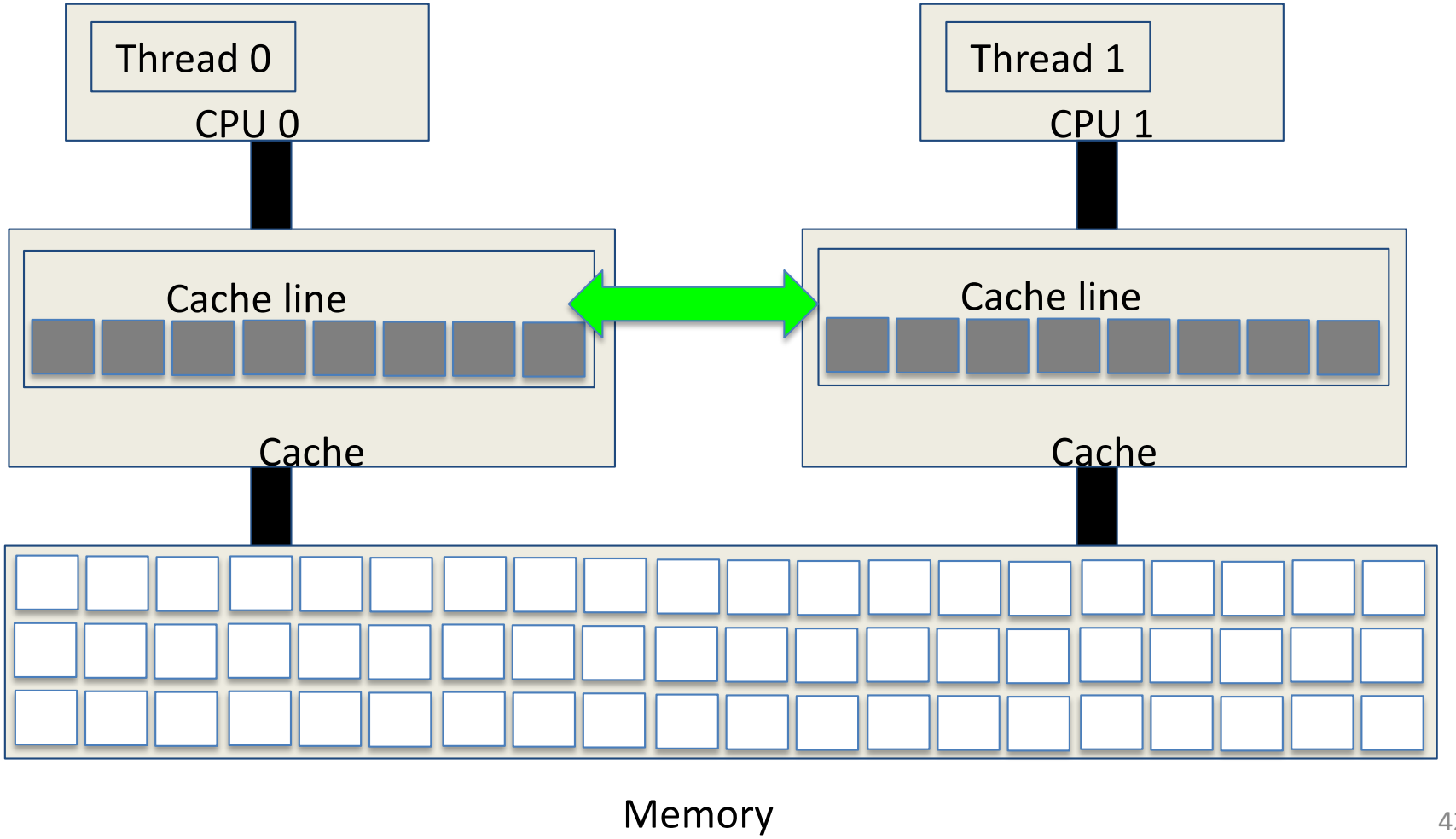



```
for (i=start; i<start + items_per_task; i++)
{
        my_x = Compute_next_value(…);
        my_sum[thread_id] += my_x;
}
```

- Multiple elements of `my_sum` are in the same cache line
- A 64 byte cache line can hold 8 doubles or 16 integers
- Cache ping-pong effect is a performance problem, also called *false sharing*

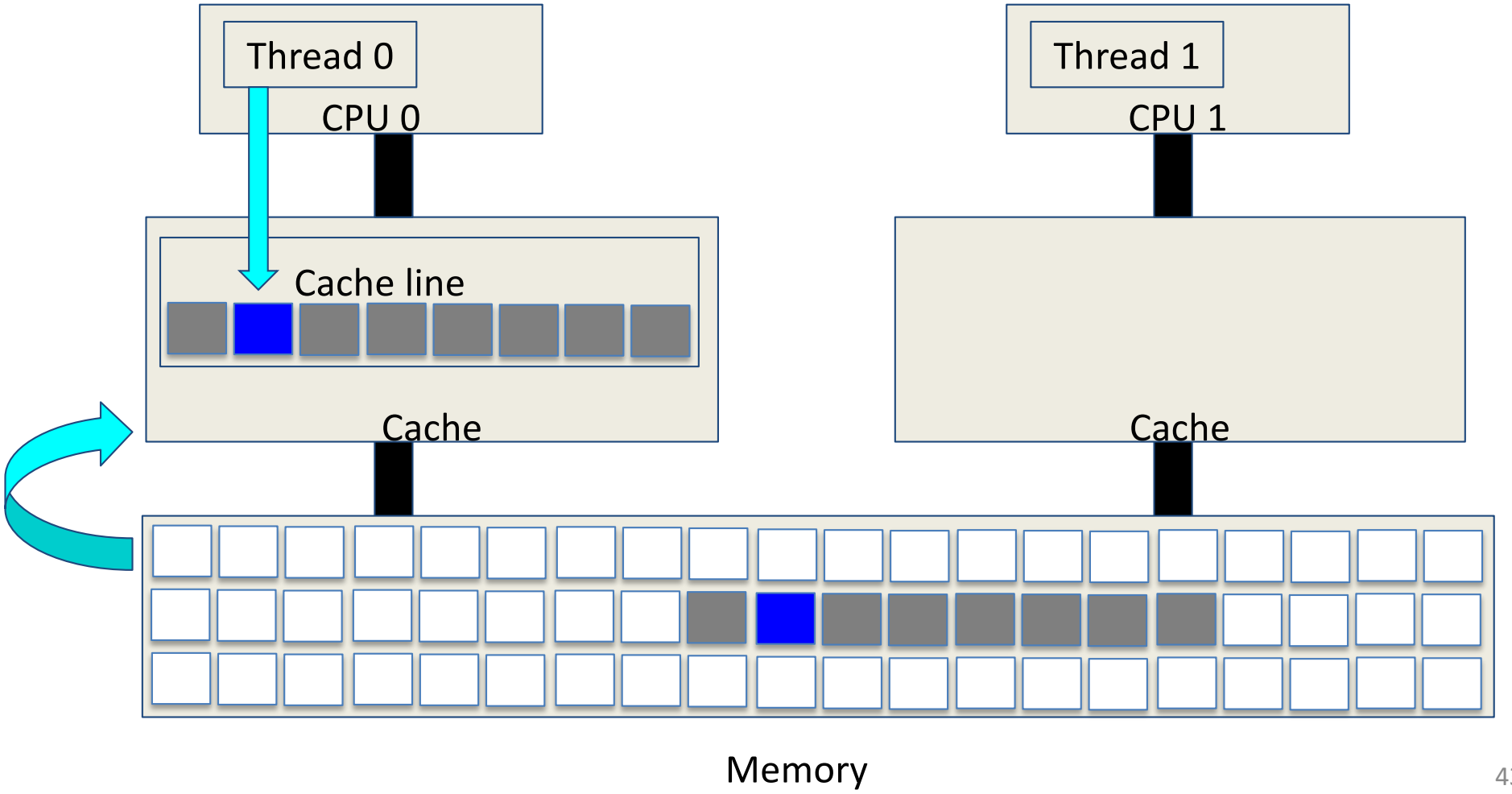
Inter-Thread Communications

- Threads communicate through cache lines



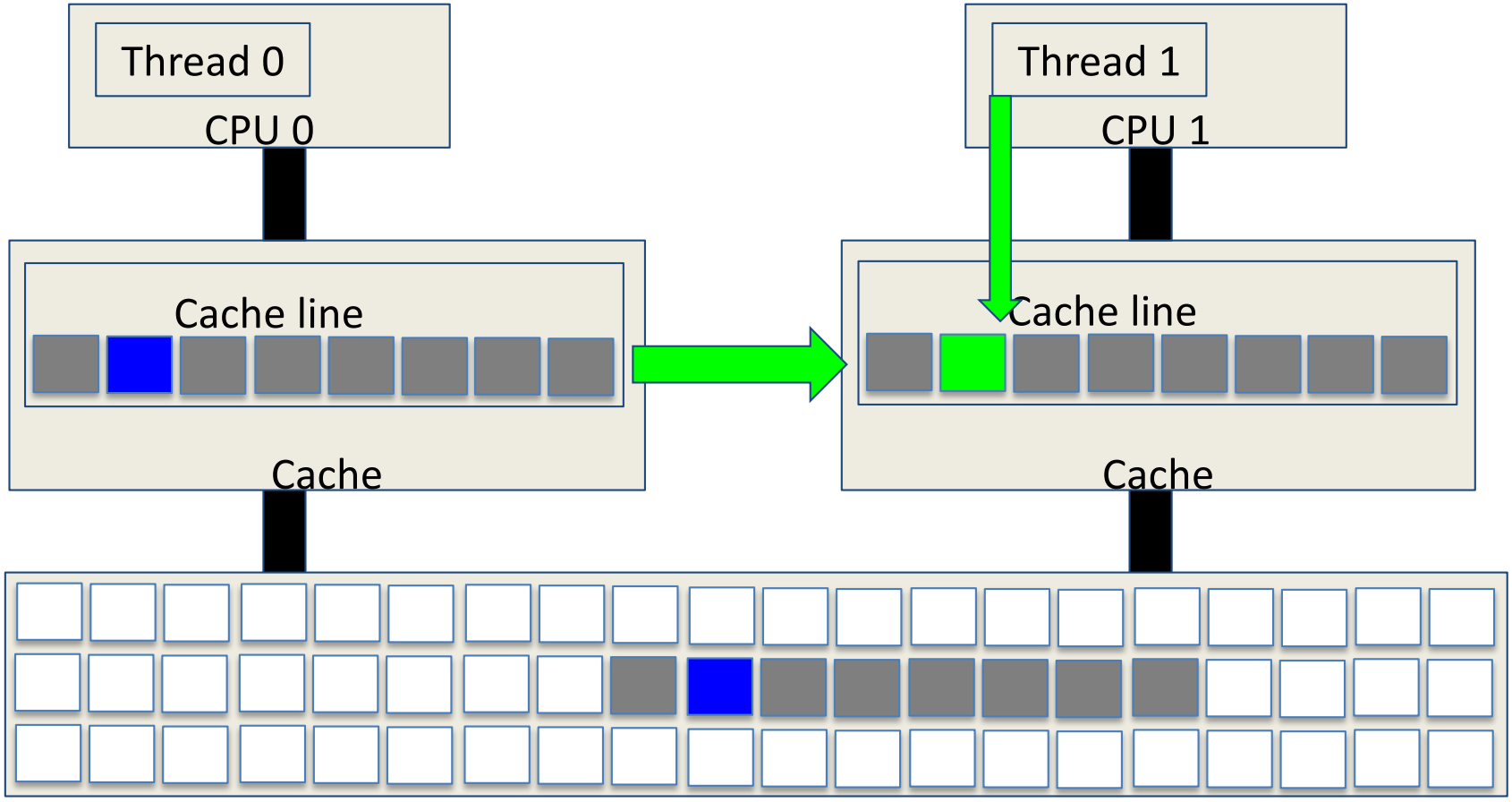
Inter-Thread Communications

- Memory access by CPU 0



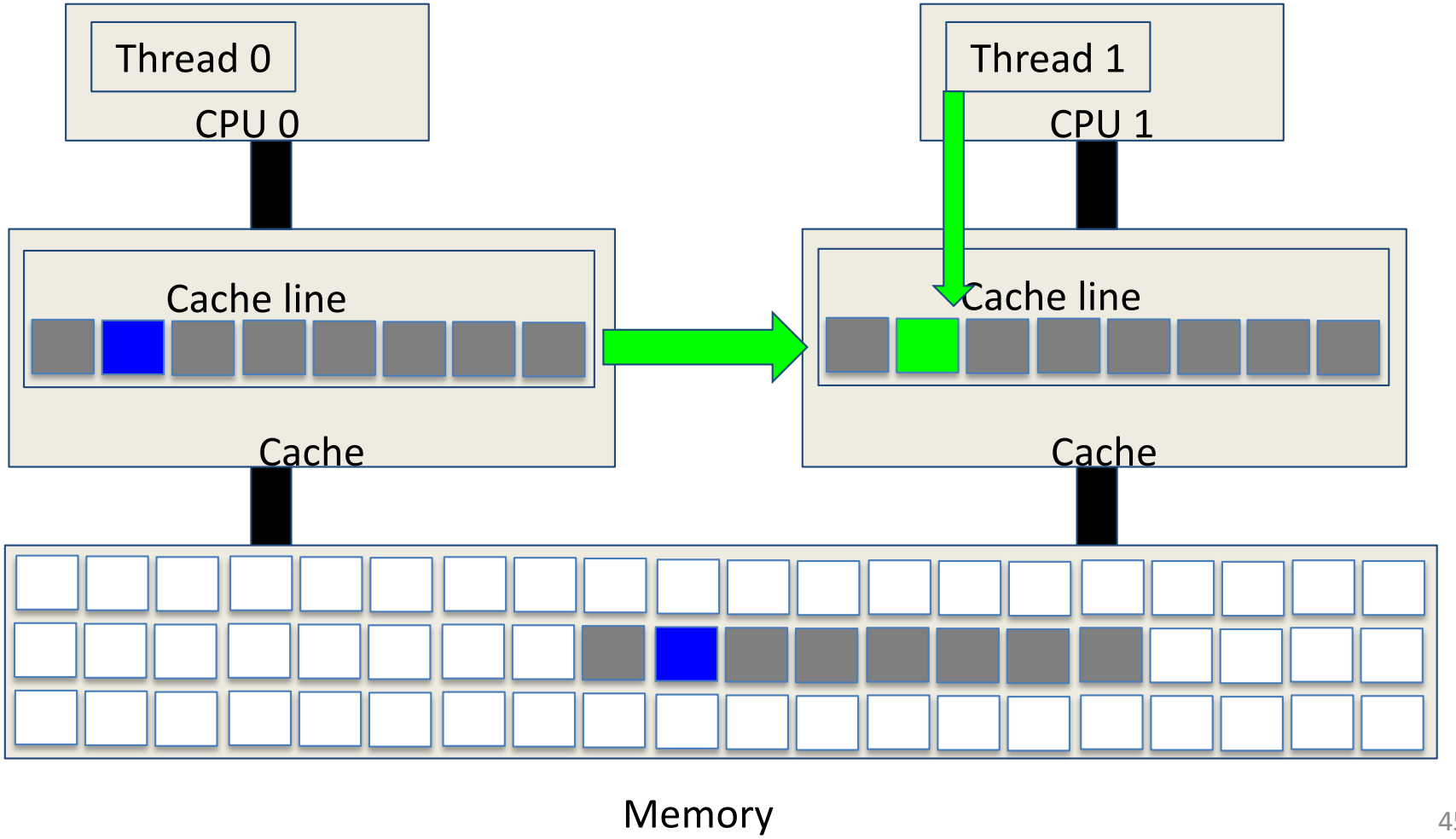
Inter-Thread Communications

- Memory access by CPU 1
- Accessed memory line already exists in the cache of CPU 0, the cache line is transferred from CPU 0 to CPU 1.



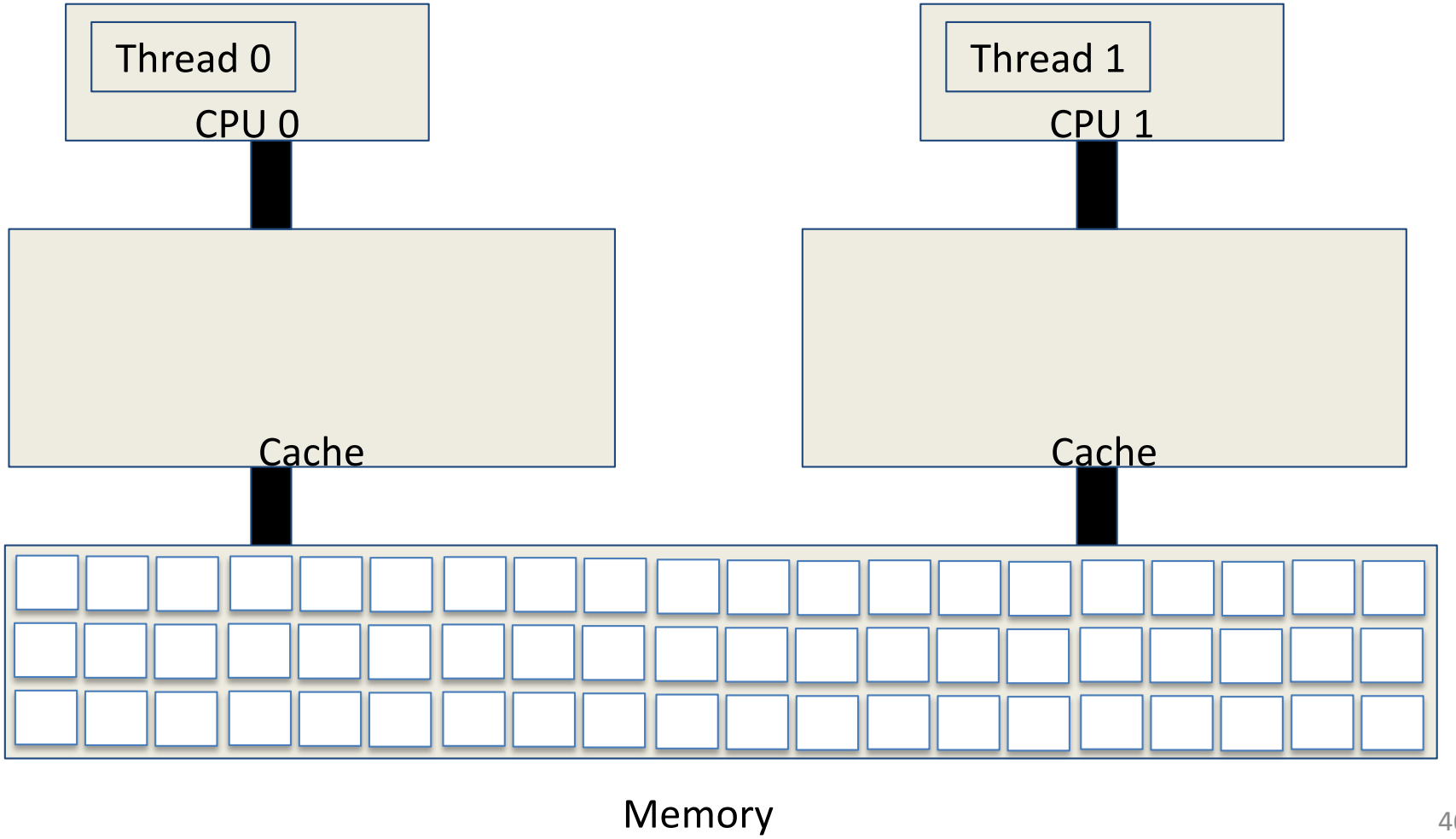
True Sharing

- This type of communication is called **TRUE SHARING**.
- Because both threads access the **same memory region** in the **same cache line**



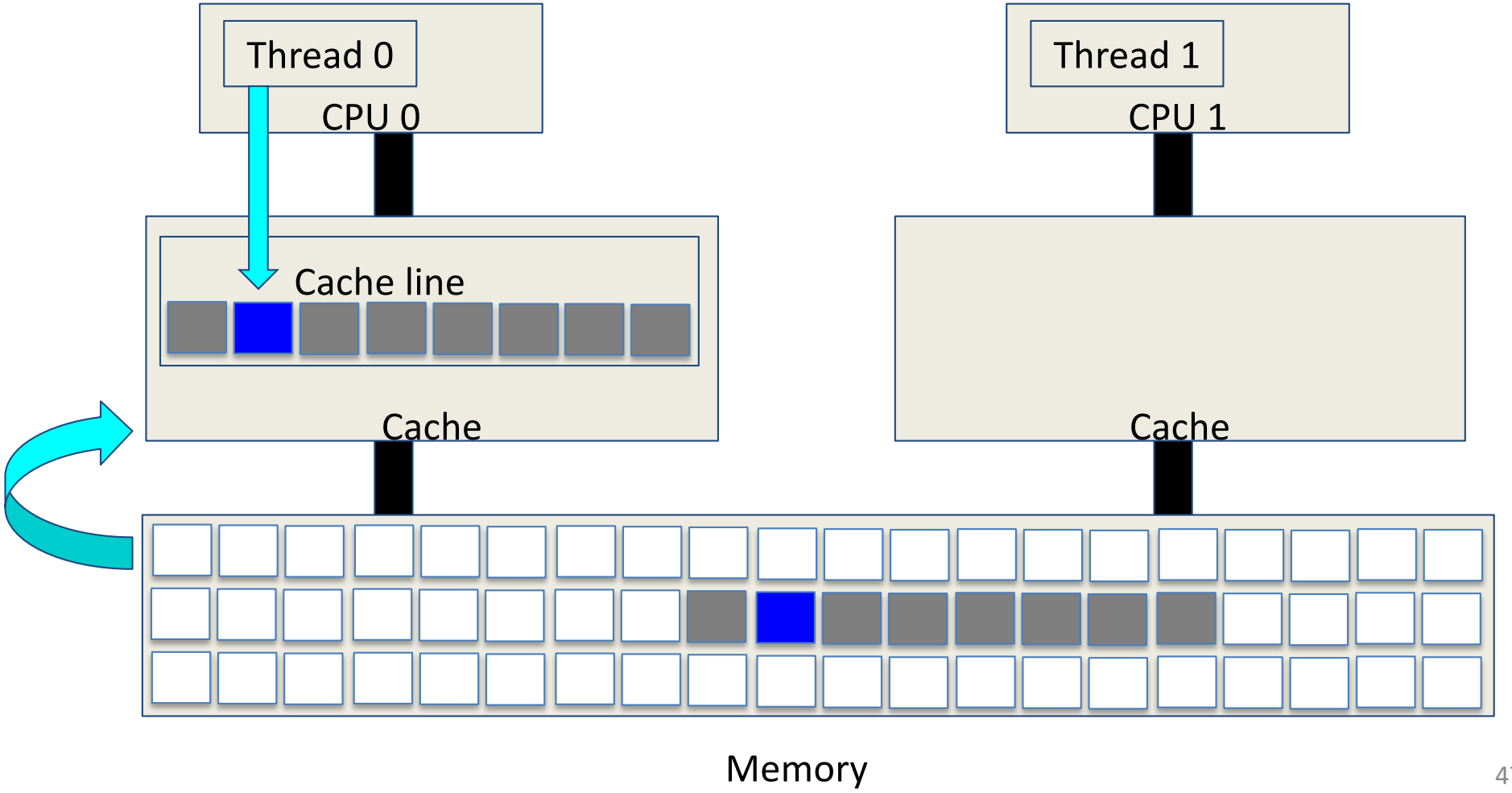
False Sharing

- There is also **FALSE SHARING** between threads



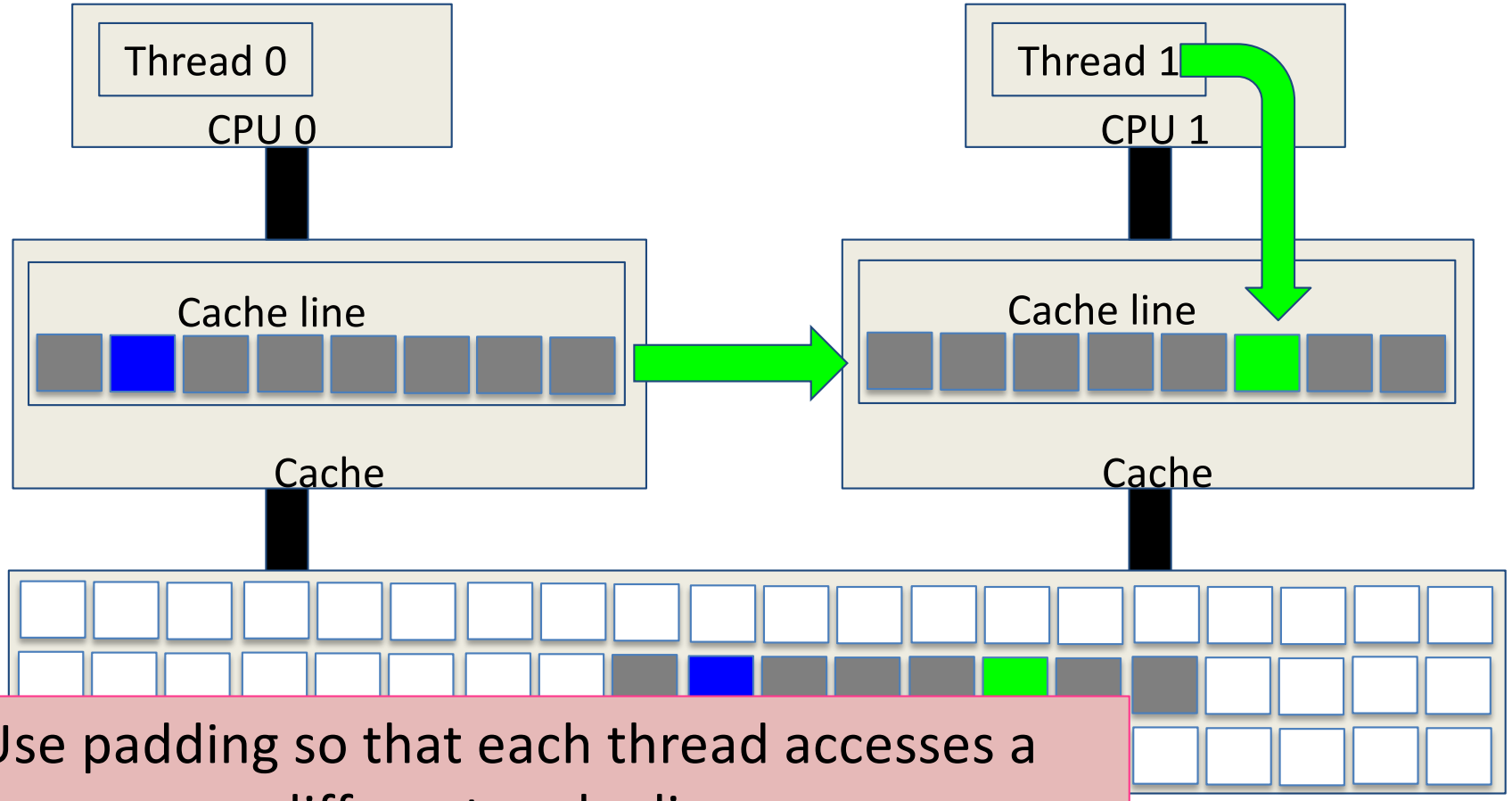
False Sharing

- Memory access by CPU 0



False Sharing

- Threads 0 and 1 access **different memory regions** in the **same cache line.**
- This is also considered to be inter-thread communication.



Use padding so that each thread accesses a different cache line

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - The course book (Pacheco)
  - https://docs.oracle.com/cd/E19205-01/819-5265/bjafa/index.html
  - Grid solver example from: Culler, Singh, and Gupta