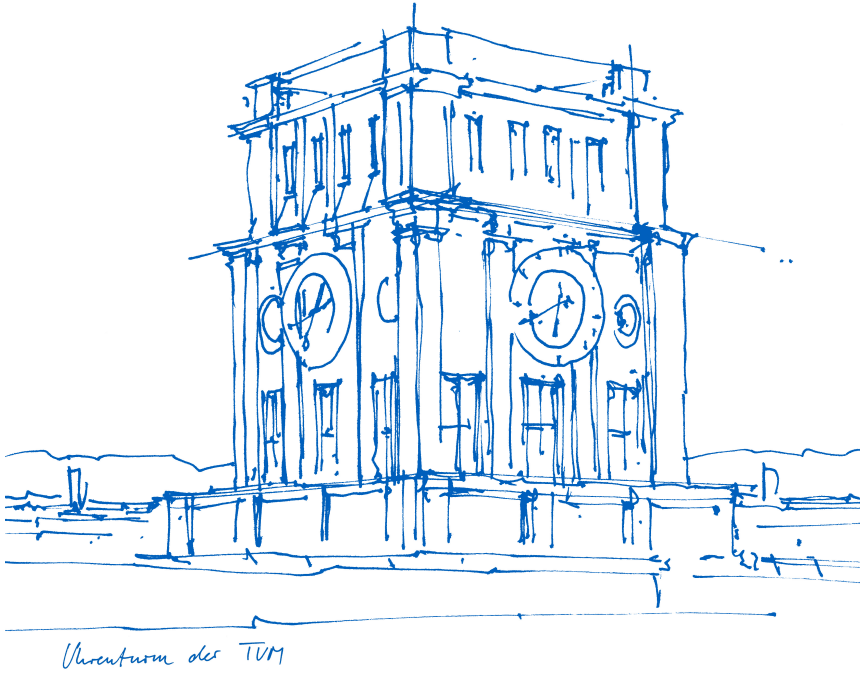
Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Exercises: Amir Raoofy
Technische Universität München
Fakultät für Informatik

Lecture 4:
Shared Memory
Challenges and Pitfalls



Summary from Last Time / OpenMP Basics

OpenMP was created to standardize the programming of shared memory systems

- First standard in 1997, currently at OpenMP 4.5
- Goals were easy of use, simplicity and portability

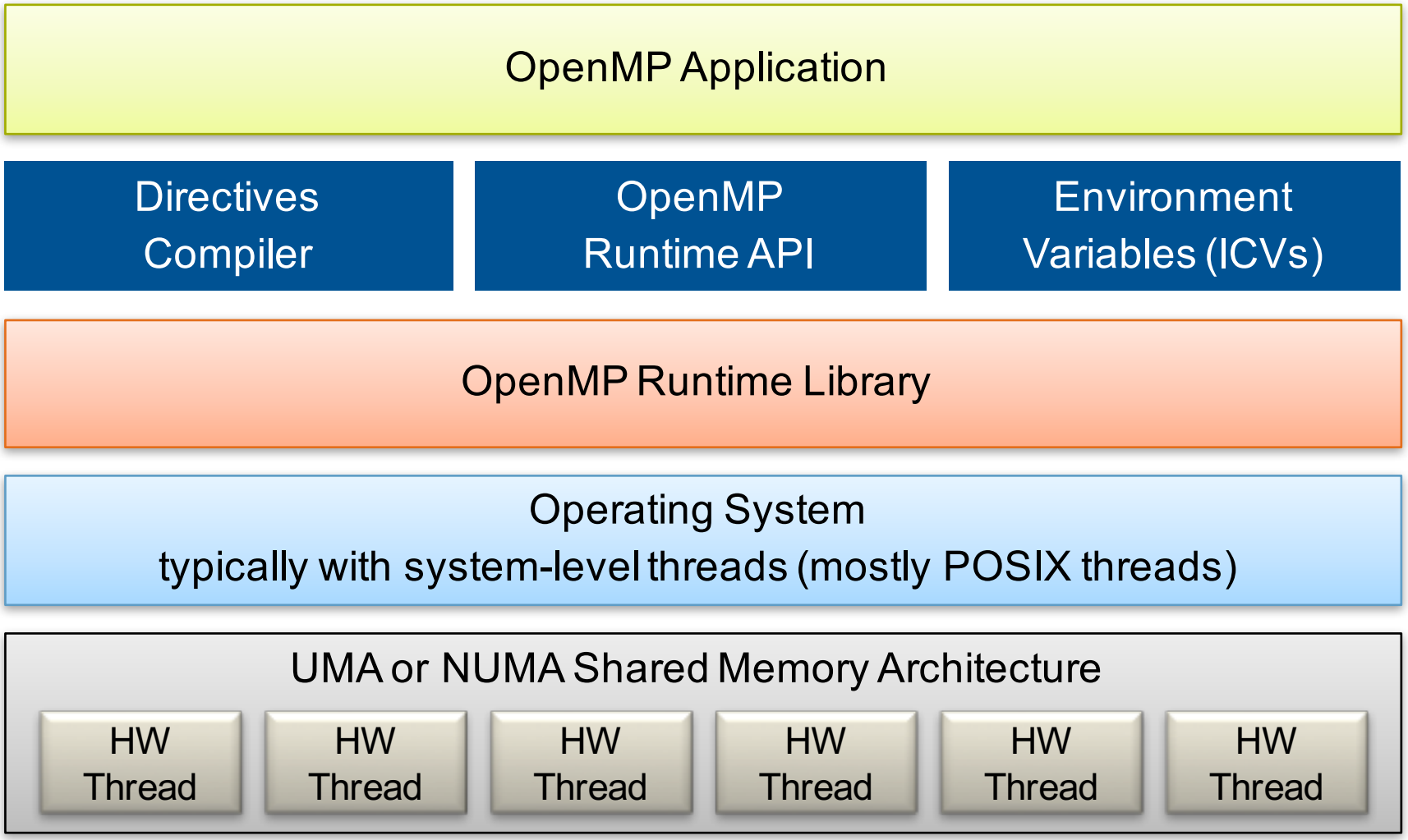
Key concepts

- Parallel regions
- Worksharing through parallel for loops
- Additional clauses to control distribution, synchronization, …
- Reduction operations
- Options to control data visibility/sharing

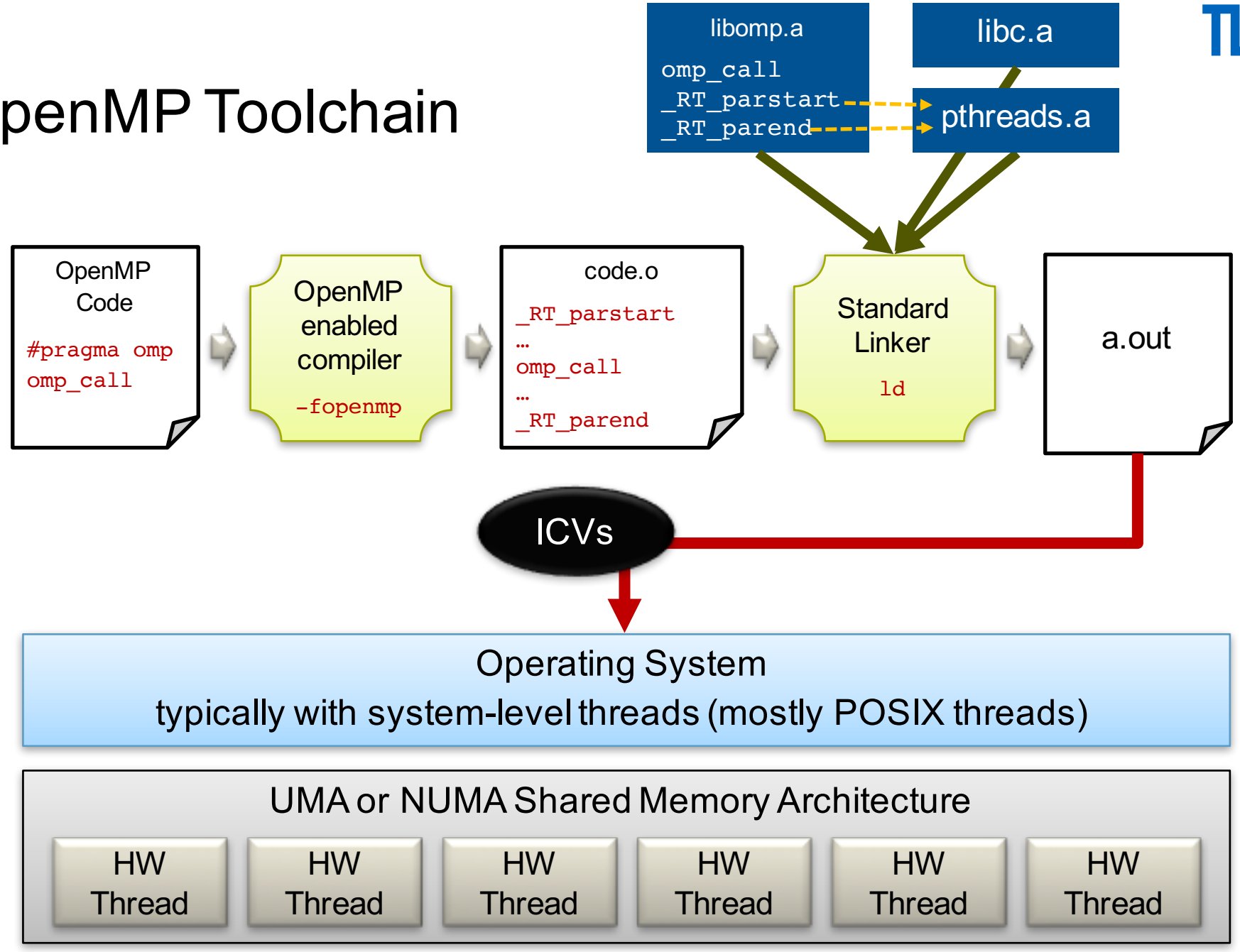
Programmer responsibility

- Ensure no loop dependencies exist
- Ensure the right variables or private or shared
- Ensure the necessary synchronization is added

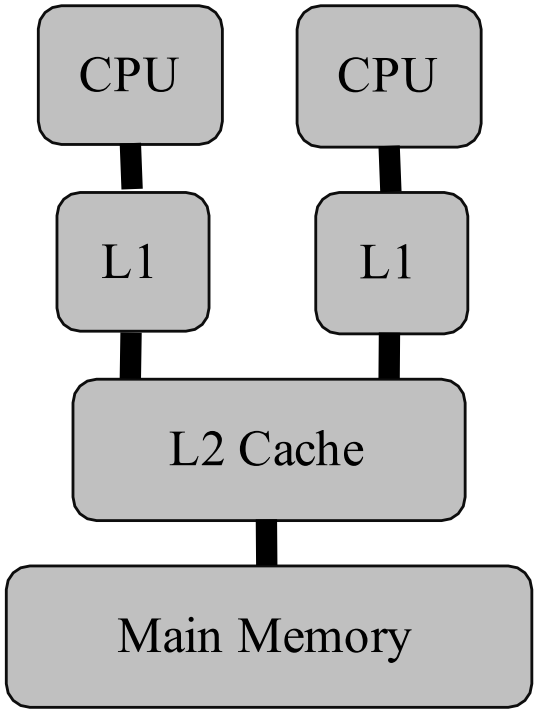
OpenMP System Stack



OpenMP Toolchain



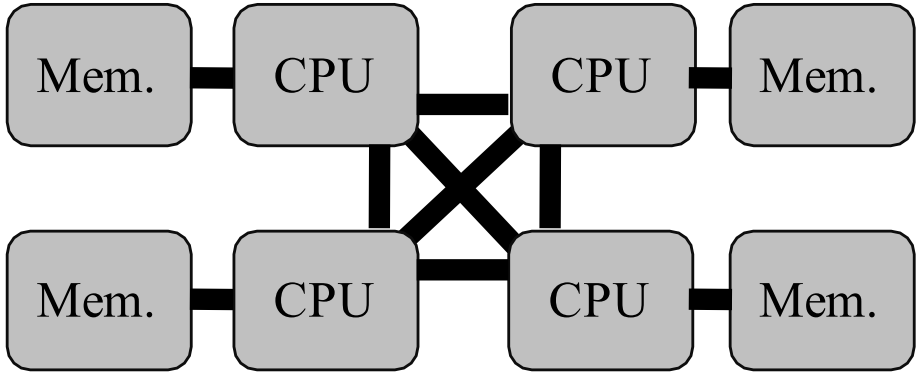
Shared Memory Programming



Many programming models for shared memory
- We talked about OpenMP and Pthreads
- C++ threading model, Java threads, …
- Extreme case: SW-DSM models

All work on the same principle
- Shared address space
- Constructs to control threads
- Communication via read/writes
- Constructs for synchronization

Similar design and optimization issues
- Directly impacted by architecture
  - Even though architecture hidden
  - Need to know underlying structure
- Memory system is critical
- Possibly synchronization instructions



Challenges and Pitfalls

How to get a parallel program?

- Dependencies
- Code transformations to remove dependencies

How to get a correct program?

- Races

How to get a fast program?

- Lock contention
  - Lock granularity
  - Lock-free data structures
- Locality!!!
  - Memory hierarchy
  - Thread and data placement

Example: Loop Parallelization

Which of those codes are parallelizable?

```
Do i=1,n
 A(i)=5*B(i)+A(i)
Enddo
```

```
Do i=1,n
 A(i-1)=5*B(i)+A(i)
Enddo
```

```
Do i=1,n
 tmp=5*B(i)
 A(i)=tmp
Enddo
```

Answer: it depends!

Need to understand dependencies

- Relationship between iterations
- Relationship between variables/arrays

Dependence and Dependence Graph

Control Dependence

S1 is dependent on S2 iff.

- There is a possible execution path from S1 to S2
- Not all paths lead from S1 to S2

Example:

```
S1: if (t == 5)
S2:     r=5.0
       else
S3:     r=3.0
```

Data Dependence

S1 is dependent on S2 iff.

- There is a feasible path from S1 to S2
- The same data is accessed in S1 and S2 with at least one being write

Example:

```
S1: pi=3.14
S2: r=5.0
S3: area=pi*r**2
```

Types of Data Dependencies

I(S) = set of memory locations read (input)
O(S) = set of memory locations written (output)

True Dependence
Flow Dependence

$O(S1) \cap I(S2) \neq \varnothing$

$(S1\ \delta\ S2)$

The output of S1 overlaps with the input of S2

```
S1: X = ...
        ...
S2: ... = X
```

Anti Dependence

$I(S1) \cap O(S2) \neq \varnothing$

$(S1\ \delta^{-1}\ S2)$

The input of S1 overlaps with the output of S2

```
S1: ... = X
        ...
S2: X = ...
```

Output Dependence

$O(S1) \cap O(S2) \neq \varnothing$

$(S1\ \delta^{o}\ S2)$

The output of S1 and S2 write to same location

```
X = ...
X = ...
```

Loop Dependencies

Loop Independent Dependencies

All dependencies are with iterations
No dependencies across iterations

Example:

```
for (i = 0; i < 4; i++)
   S1: b[i] = 8;
   S2: a[i] = b[i] + 10;
```

Iterations can be executed in parallel

Loop Carried Dependencies

Dependencies across iterations
Computation in one iteration depends on data written in another

Example:

```
for (i = 0; i < 4; i++)
   S1: b[i] = 8;
   S2: a[i] = b[i-1] + 10;
```

Iterations cannot be (easily/directly) executed in parallel

Aliasing

Question: which variables refer to the same physical location?

- Simple for individual variables
- Harder for pointers
- Really hard for C-style pointers
- Impossible to statically determine for arbitrary pointer structures

```
Foo(a,b,n)
 Do i=1,n
  A(i)=5*B(i)+A(i)
 Enddo
```

```
int A[100],B[100];

Foo(A,B,100);
Foo(&A[1],B,100);
C=A+5; Foo(A,C,100);
Foo(A,A,100);
```

Options

- Programmer has to know data structures and know about aliases
- Library developer may have to assert conditions in the API
- Dynamic inspector/executor tests
- Compiler annotations to assert non-aliasing

Program Order vs Dependence

The sequential order imposed by the program is too restrictive

- Just one way to write the problem
- Possibly hides possible parallelism

Only need to uphold all dependencies to guarantee program correctness

- Look at partial order of all dependences
- Aside from that re-orderings are legal

Code transformations can help, but they have to be safe

- A reordering transformation **preserves a dependence** if it preserves the relative execution order of the source and sink of that dependence.
- Consequence: any reordering transformation that preserves every dependence in a program leads to an **equivalent** computation.
- A transformation is said to be **valid** for the program to which it applies if it preserves all dependences in the program.

Loop Terminology

```
L1: do i1=0,n1-1
L2:   do i2=0,n2-1
        …
Ln:      do in=0,nn-1
S1:       … = …
        enddo
        …
      enddo
    enddo
```

Nested loops are loops within loops

- Loop's Nesting level = number of surrounding loops +1
- **Iteration number** in a normalized (0 to n-1) loop is equal to the value of the iterator

Iteration vector for one iteration at the innermost loop

- Tupel that contains all iteration numbers
- Example: $\mathbb{I}$ = (3, 2, …, 7)

The set of all possible iteration vectors for a statement is an **iteration space**.

Iteration $\mathbb{I}$ **precedes** iteration $\mathbb{J}$, denoted $\mathbb{I} < \mathbb{J}$, iff

$\exists k \quad \mathbb{I}[r] = \mathbb{J}[r] \quad \forall r\ 1 \leq r < k$ AND $\mathbb{I}[k] < \mathbb{J}[k]$

For each statement S in a loop nest with n loops

- exists a specific **statement instance** for each iteration vector $\mathbb{I}$
- We label this as S($\mathbb{I}$)

Loop Dependences

There exists a dependence from S1($\mathbb{I}$) to S2 ($\mathbb{J}$) in a common nest of loops, iff,

1. $\mathbb{I} < \mathbb{J}$ or
   $\mathbb{I} = \mathbb{J}$ and there is a path from S1 to S2 in the body of the loop,
2. Statement S1 accesses memory location M on iteration $\mathbb{I}$
3. Statement S2 accesses location M on iteration $\mathbb{J}$
4. At least one of these accesses is a write.

We denote this as **(S1($\mathbb{I}$) δ S2($\mathbb{J}$))**

Loop dependence

There exists a dependence from statement S1 to statement S2
in a common nest of loops

if and only if

there exist $\mathbb{I}$ and $\mathbb{J}$ such that (S1($\mathbb{I}$) δ S2($\mathbb{J}$))

Dependence Distance for (S1($\mathbb{I}$) δ S2($\mathbb{J}$)): $\mathbb{J}$- $\mathbb{I}$
Dependence Direction: tupel with “<“, “=“, “>” for each loop showing sign of distance

Transformation 1: Loop Interchange

```
do l=1,10
  do m=1,10
    do k=1,10
      A(l,m,k)=C(l,m,k)+B
    enddo
  enddo
enddo
```

≡

```
do l=1,10
  do k=1,10
    do m=1,10
      A(l,m,k)=C(l,m,k)+B
    enddo
  enddo
enddo
```

A loop interchange is safe for a perfect loop nest, if
the direction vectors of all dependences has only ”=“

A loop interchange is safe for a perfect loop nest, if
the direction vectors of all dependences
do not have a “>” as the leftmost non-”=“ direction.

Assuming no aliasing

Loop Interchange (2)

```
do l=1,10
  do m=1,10
    do k=1,10
      A(l+1,m+2,k+3)=A(l,m,k)+B
    enddo
  emddo
enddo
```

≡

```
do l=1,10
  do k=1,10
    do m=1,10
      A(l+1,m+2,k+3)=A(l,m,k)+B
    enddo
  enddo
enddo
```

Assuming no aliasing

Transformation 2: Loop Distribution / Loop Fission

Transforms loop-carried dependences into loop-independent dependences.

```
do j=2,n
S1:  a(j)= b(j)+2
S2:  c(j)= a(j-1) * 2
enddo
```

```
do j=2,n
S1:  a(j)= b(j)+2
enddo

do j=2,n
S2:  c(j)= a(j-1) * 2
enddo
```

Safety of loop distribution

- Two sets of statements in a loop nest can be distributed into separate loop nests, if no dependence cycle exists between those groups.
- Dependences carried by outer non-distributed loops can be ignored.
- The order of the new loops has to preserve the dependences among the statement sets.

To consider

- It generates multiple parallel loops, thus decreasing granularity.
- Barrier synchronization might be required between the generated loops.

Transformation 3: Loop Fusion

Combine two loops

```
do i=1,n
  a(i)= b(i)+2
enddo

do i=1,n
  c(i)= d(i+1) * a(i)
enddo
```

```
do i=1,n
  a(i)= b(i)+2
  c(i)= d(i+1) * a(i)
enddo
```

Loop distribution vs loop fusion

- Loop fusion is the dual transformation. It combines subsequent loops.
- Loop distribution reduces the granularity.
- Loop fusion increases granularity.
- Loop distribution might increase parallelism by separating parallelizable and non-parallelizable parts of a loop nest.
- Loop fusion might reduce parallelism by creating a sequential loop.
- Loop fusion can introduce loop carried dependencies

Possible Pitfalls for Loop Fusion

Incorrect Loop Fusion

```
do i=1,n
S1:   a(i)= b(i)+2
enddo

do i=1,n
S2:   c(i)= d(i+1) * a(i+1)
enddo
```

```
do i=1,n
S1:   a(i)= b(i)+2
S2:   c(i)= d(i+1) * a(i+1)
enddo
```

Correct Loop Fusion

```
do i=1,n
S1:   a(i)= b(i)+2
enddo

do i=1,n
S2:   c(i)= d(i+1) * a(i-1)
enddo
```

```
do i=1,n
S1:   a(i)= b(i)+2
S2:   c(i)= d(i+1) * a(i-1)
enddo
```

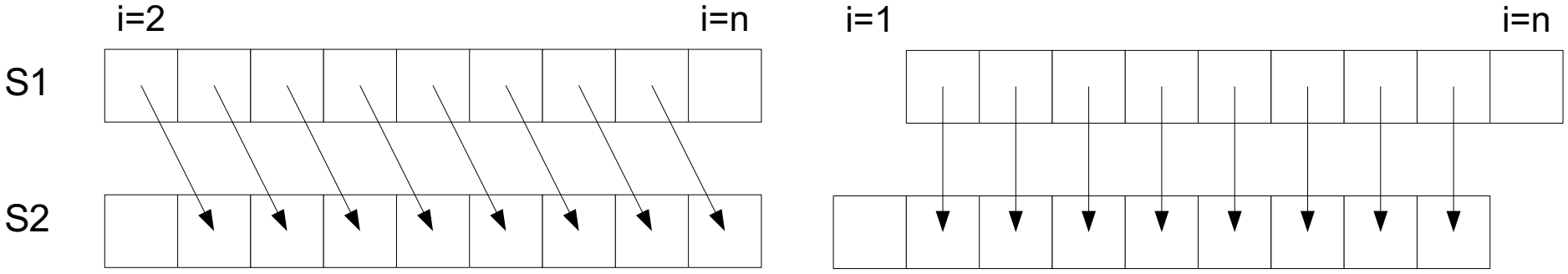
Transformation 4: Loop Alignment

Loop alignment changes a carried dependence into an independent dependence.

```
do i=2,n
S1:   a(i)= b(i)+2
S2:   c(i)= a(i-1) * 2
enddo
```

Idea: Shift of computations into other iterations.

Extra iterations and tests are overhead.



Loop Alignment

Loop alignment changes a carried dependence into an independent dependence.

```
do i=2,n
S1:  a(i)= b(i)+2
S2:  c(i)= a(i-1) * 2
enddo
```

```
do i=1,n
S1:  if (i>1) a(i)= b(i)+2
S2:  if (i<n) c(i+1)= a(i) * 2
enddo
```

First and last iteration can be peeled of:

```
c(2)=a(1) * 2
do i=2,n-1
S1:  a(i)= b(i)+2
S2:  c(i+1)= a(i) * 2
enddo
a(n)=b(n) + 2
```

In general, we can eliminate all carried dependences in a single loop if

- Loop contains no recurrence (dependence cycle) and
- Distance of each dependence is a constant independent of the loop index

Summary Loop Transformations

Loops that do not carry any dependences can be parallelized

The outermost loop should be parallelized, if possible
- Parallel inner loops can be moved outside by loop interchange.

In a perfect nest of loops, a particular loop can be parallelized at the outermost level iff. the appropriate entry in all direction vectors contains only “=“ entries.

Transformations can eliminate carried dependences

Examples:
- Loop distribution
- Loop alignment

Transformations can improve efficiency

Examples:
- Loop fusion
- Loop interchange

Correctness / Dealing with Races

Two threads access the same shared variable

- At least one thread modifies the variable
- The accesses are concurrent, i.e. unsynchronized

Leads to non-deterministic behavior

- Depends on timing of accesses

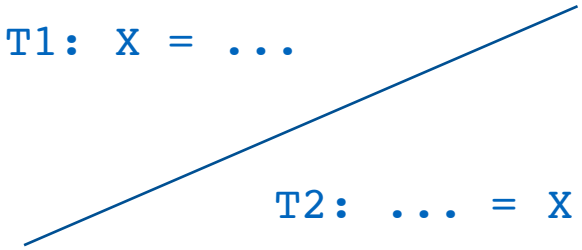
Example:

```
static double farg1,farg2;
#define FMAX(a,b) (farg1=(a),farg2=(b),farg1>farg2?farg1:farg2)

1619: #pragma omp parallel for shared(bar, foo, THRESH)
1620: for (x=0; x<1000;x++)
1621:     T = FMAX(0.1111*foo*bar[x],THRESH);
```

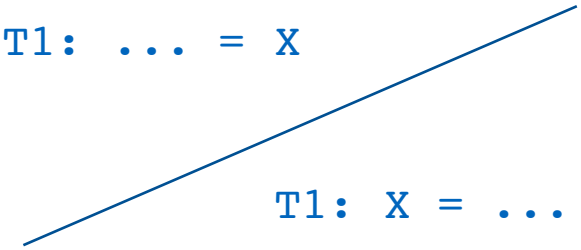
Types of Races

Analog to dependencies

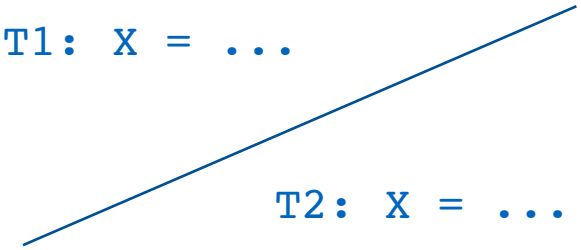
Read after Write (RAW)



Write after Read (WAR)



Write after Write (WAW)



Races can be benign
- Can deal with old or new value at a read
- Know the same value gets written

In most cases, we need to either introduce additional synchronization
- Add happens before relationship
- Examples: mutexes, barriers, …

Or we need to apply the right privatization

Race Detection Tools

Races are hard to find with traditional approaches
- Non-deterministic
- Debugging changes timing

Use dedicated race detection tools
- Static and/or dynamic instrumentation of all memory accesses
- Tracking synchronization
- Detection of unsynchronized accesses to the same memory

Examples of tools (not exhaustive)
- Helgrind (open source)
  - Dynamic, based on valgrind tool suite
- Intel Inspector (commercial)
  - Part of Intel's development suite
- Thread Sanitizer (open source)
  - Static instrumentation via LLVM
- Archer (open source)
  - Combines Thread Sanitizer's approach with OpenMP semantics

Performance Aspects

Issue 1:

Synchronization Overhead

Issue 2:

Cache Behavior and Locality

Issue 3:

Thread and Data Locality / Mapping

Performance Aspects
Issue 1: Synchronization Overhead

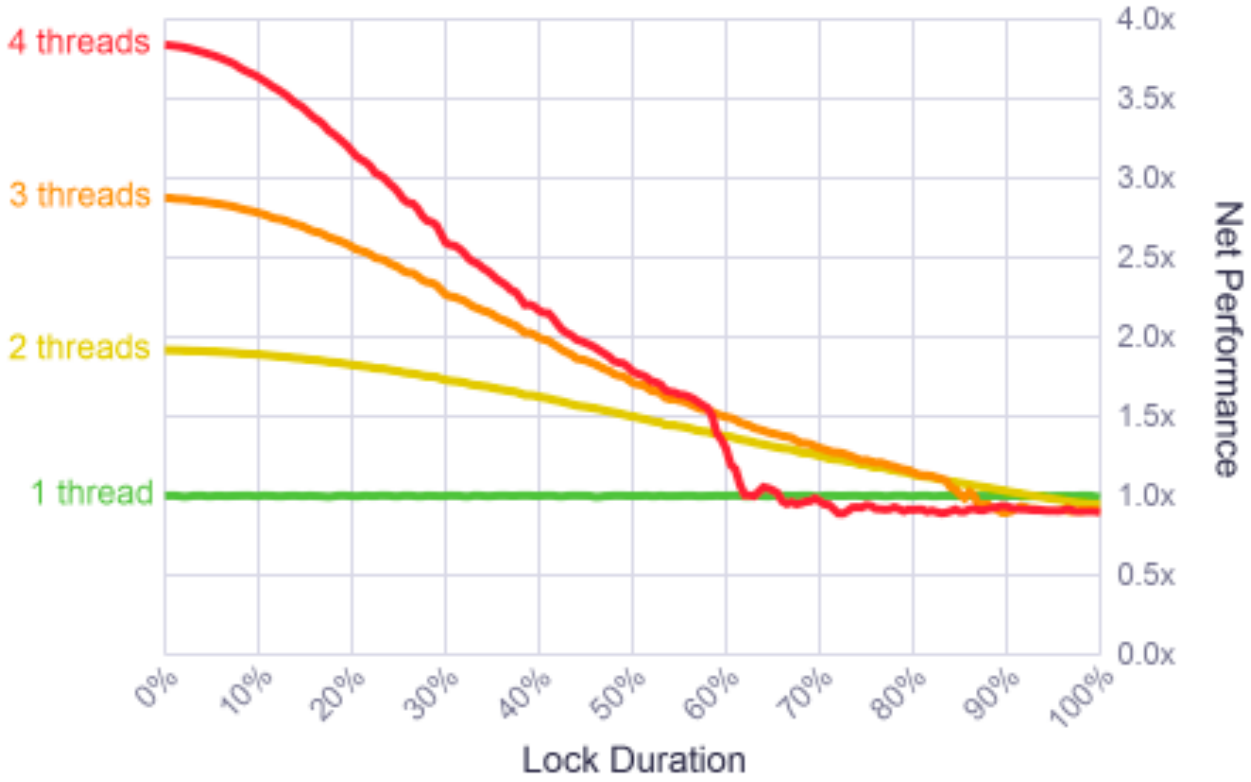
Synchronization causes overhead

- No useful work being done
- Directly impacts speedup

Good Locking Behavior

- Only use locks when needed
- Reduce time spent in locks
- Carefully order lock usage to avoid deadlocks
- If critical sections have to be substantial, try overlapping

Lock Contention



From: http://preshing.com/20111118/locks-arent-slow-lock-contention-is/

Performance Aspects
Issue 1: Synchronization Overhead

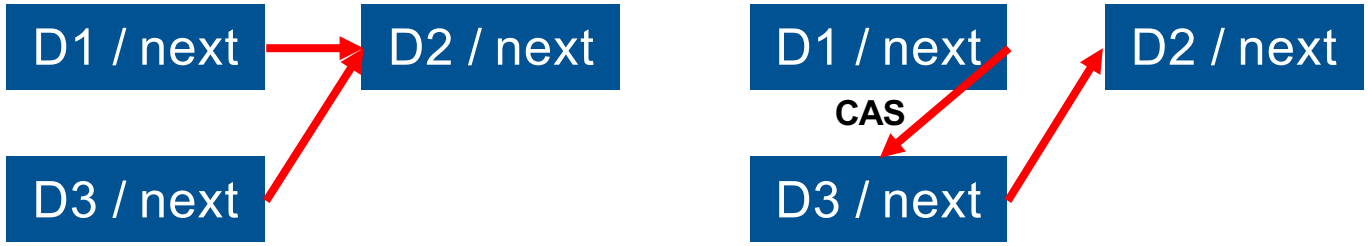
Synchronization causes overhead
- No useful work being done
- Directly impacts speedup

Good Locking Behavior
- Only use locks when needed
- Reduce time spent in locks
- Carefully order lock usage to avoid deadlocks
- If critical sections have to be substantial, try overlapping

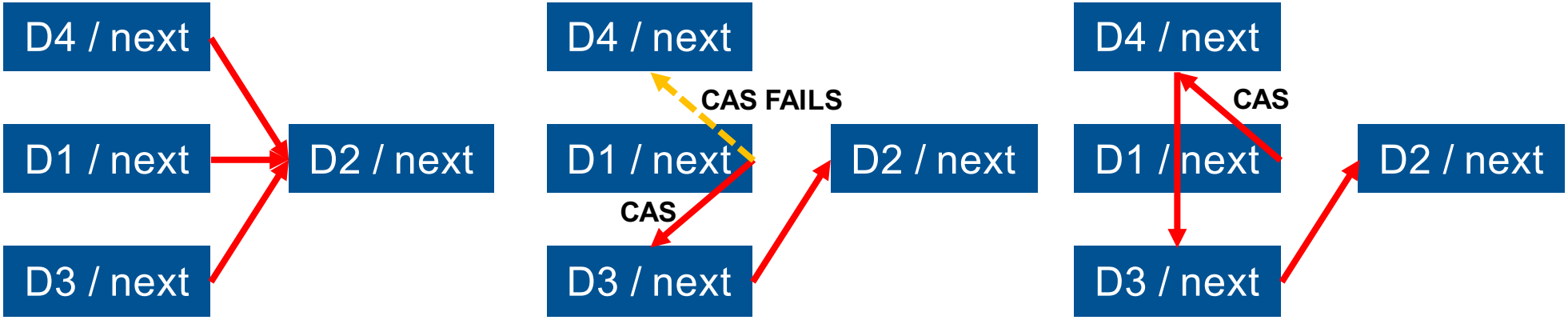
Alternative: lock-free data structures
- Avoid use of mutex/critical sections
- Carefully manipulate memory to avoid bad “intermediate” state
- Hardware support essential
  - Compare and swap is common
  - Compare two values and only write new value if compare succeeds
  - See also lecture 2

Example: Linked List

Uncontended Insert



Contended Insert



Example: Linked List (cont.)

Remove operation is similar
- CAS to unlick data element
- Ensure inconsistent list state is not exposed

ABA Problem
- A tries to remove an element
  - Has to wait
- B removes the elemet
- Frees it
- Reallocates it (new element, same address)
- B inserts element again
- A compares against the old/new element

ABA can happen when elements are pointers
- With just values it doesn't matter
- Ensure that data isn't reused (esp. for languages with implicit memory mngt.)
- Or: use generational counters

Thoughts on Lock-free Programming

Lock-free data structures can improve performance
- Avoids calls to mutex routines
- Many implementations available
- Good reference: http://www.audiomulch.com/~rossb/code/lockfree/

Drawbacks
- Hard to get right / can be very subtle (see ABA problem)
- Requires hardware support
  - Most modern processors have that
  - Compare with mutex implementations in lecture 2
- Contention for a shared resource still the case

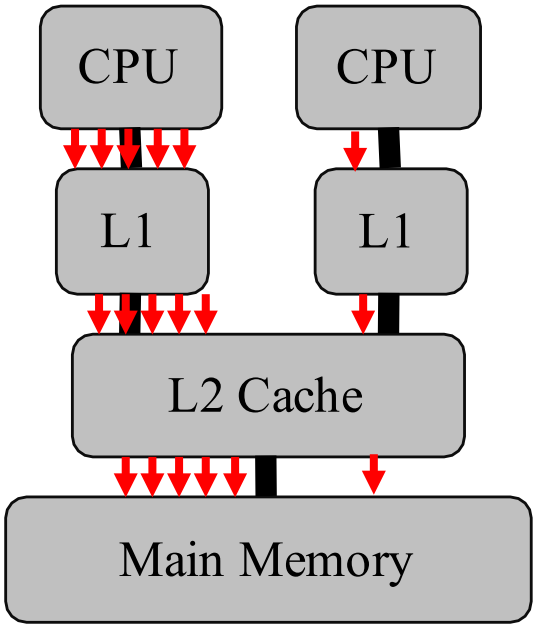
Don't confuse with **wait-free** programming
- Definition: every operation finishes in a finite number of steps
- Wait free requires lock free, but not the other way around
- Wait free needed for hard real time scenarios
- Much harder to achieve, in some cases impossible

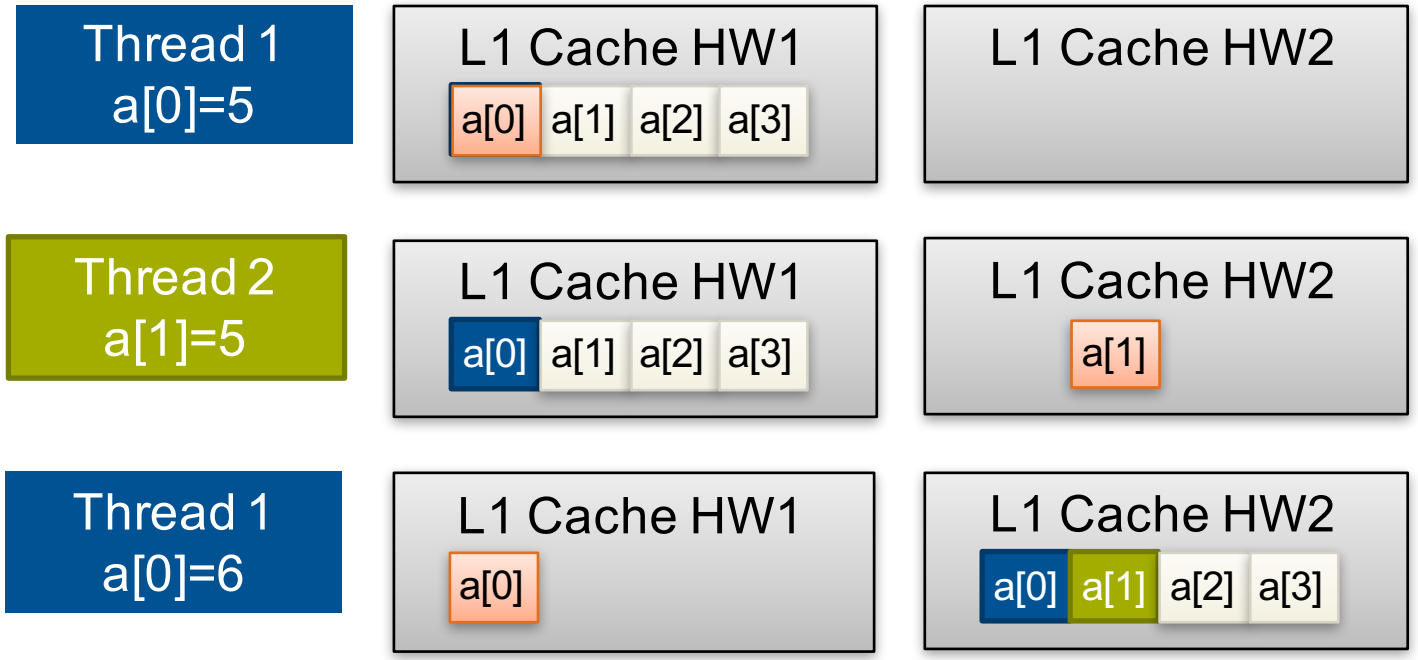
Performance Aspects
Issue 2: Cache Behavior and Locality

Improving cache performance is critical for sequential code

Situation even worse in parallel case
- More threads need bandwidth
- More pressure on caches
- Contention on main memory

Additional problem: false sharing





Example Impact of False Sharing

```
! Cache line  UnAligned
real*4, dimension(100,100)::c,d
!$OMP PARALLEL DO
do i=1,100
  do j=2, 100
     c(i,j) = c(i, j-1) + d(i,j)
  enddo
enddo
!$OMP END PARALLEL DO
```

```
! Cache line  Aligned
real*4, dimension(112,100)::c,d
!$OMP PARALLEL DO SCHEDULE(STATIC, 16)
do i=1,100
  do j=2, 100
     c(i,j) = c(i, j-1) + d(i,j)
  enddo
enddo
!$OMP END DO
```

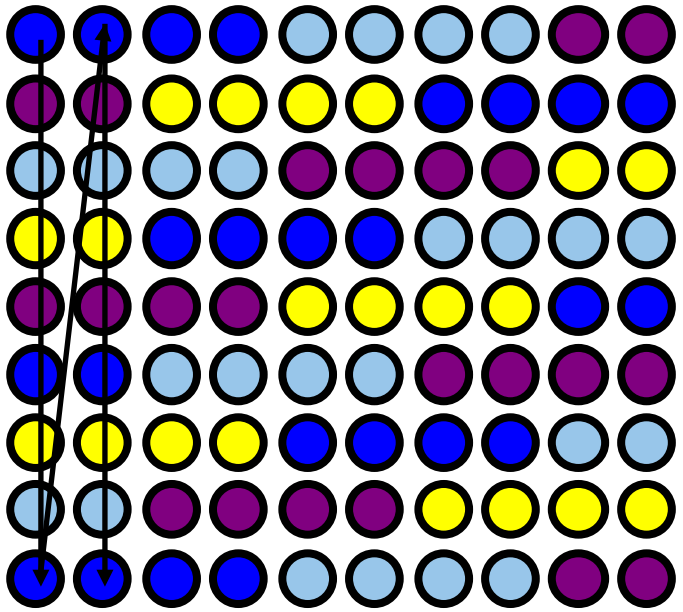
Same computation, but careful attention to alignment and independent OMP parallel cache-line chunks can have big impact
L3_EVICTIONS a good measure (measured using HW counters)

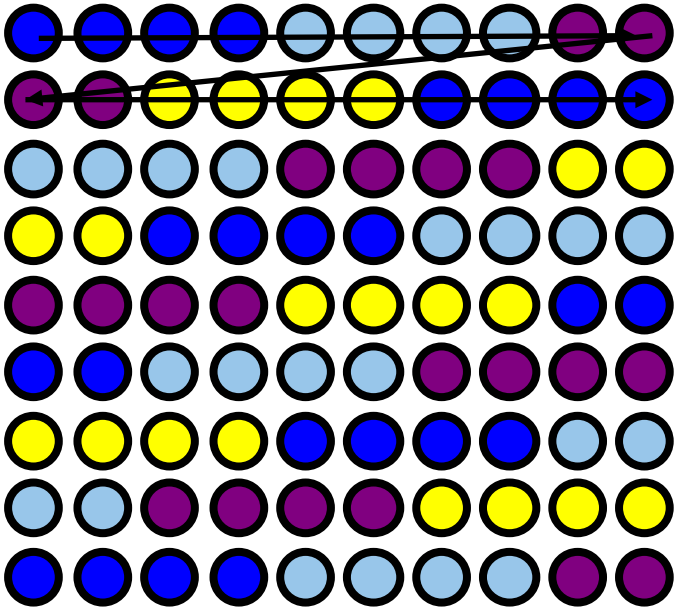
	Run Time	L3_EVICTIONS: ALL	L3_EVICTIONS: MODIFIED
Aligned	6.5e-03	9	3
UnAligned	2.4e-02	1583	1422
Perf. Penalty	3.7	175	474

Example by Mahesh Rajan (SNLs)

Example: Improving Data Locality

```
Do j=1,n
    do i=1,m
        b(i,j)=5.0
    enddo
enddo
```

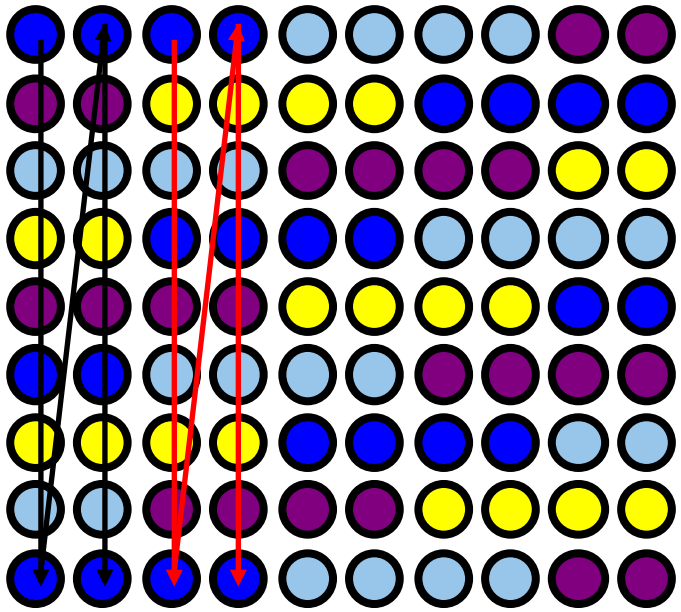


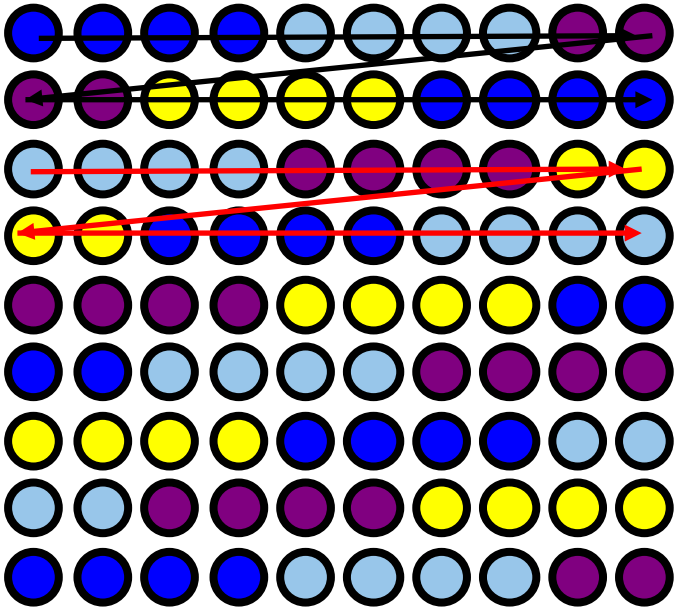


```
Do i=1,n
    do j=1,m
        b(i,j)=5.0
    enddo
enddo
```

Example: Improving Data Locality

```
Do j=1,n
    do i=1,m
        b(i,j)=5.0
    enddo
enddo
```

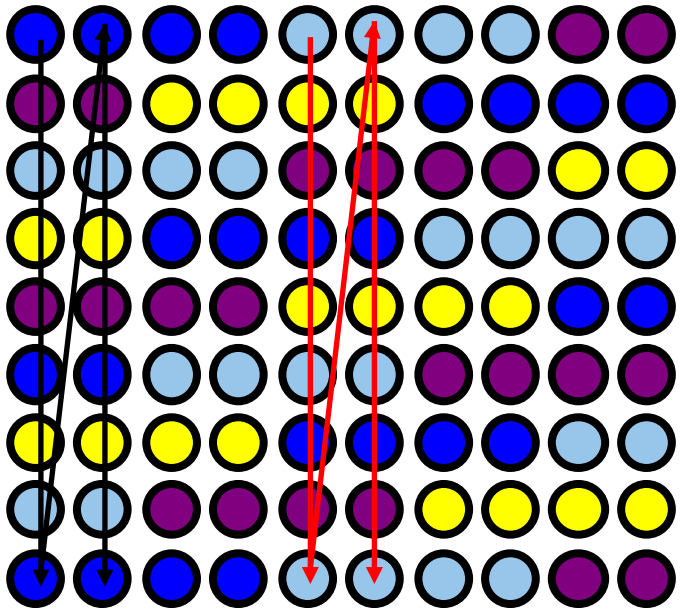


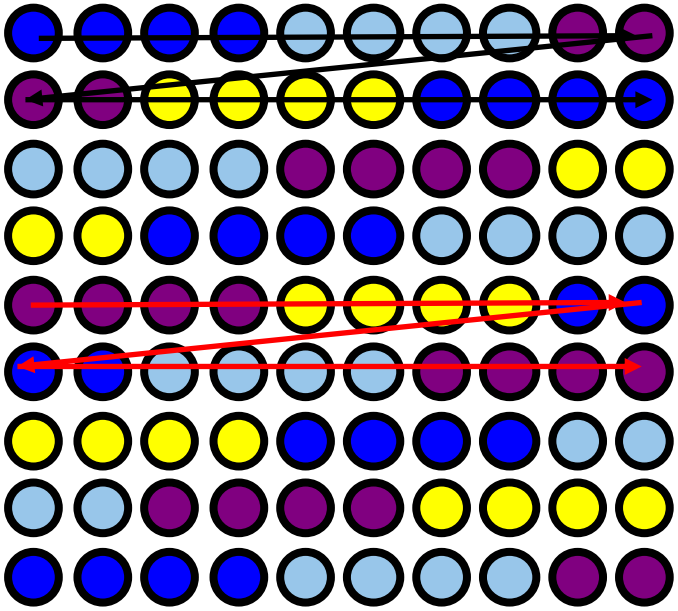


```
Do i=1,n
    do j=1,m
        b(i,j)=5.0
    enddo
enddo
```

Example: Improving Data Locality

```
Do j=1,n
    do i=1,m
        b(i,j)=5.0
    enddo
enddo
```



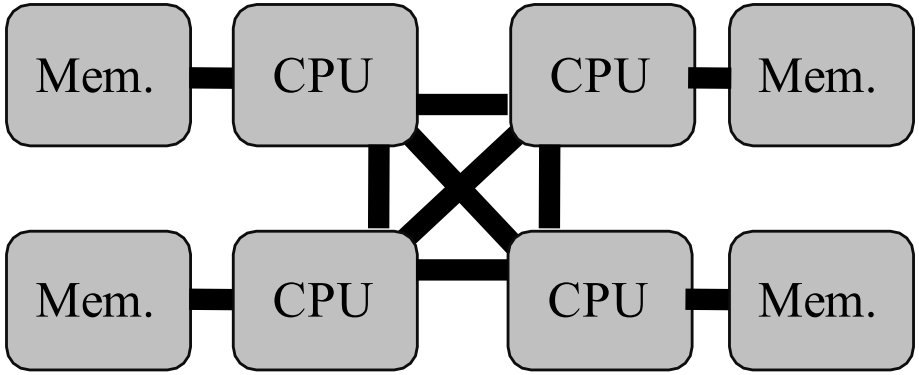


```
Do i=1,n
    do j=1,m
        b(i,j)=5.0
    enddo
enddo
```

Performance Aspects
Issue 3: Thread and Data Locality / Mapping

NUMA systems are dominating

- Multiple sockets per node/system
- Multiple cores per socket
- Each socket has its own memory
- All memory globally addressable



Different data access latencies

- Local accesses to memory located on the same socket as the thread
  - Short access latency
- Remote accesses to memory located on a different socket as the thread
  - Long access latency

Location of data and threads and if data vs. threads important

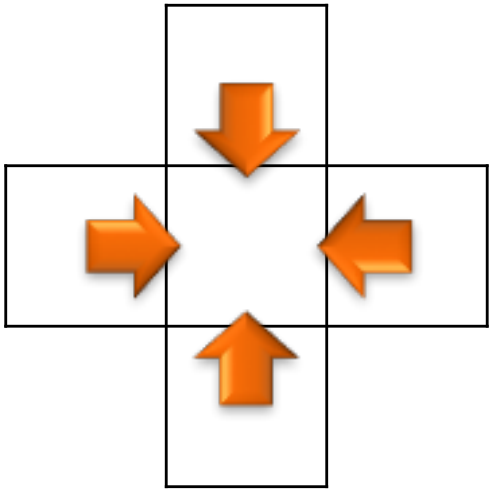
- Need to coordinate threads and memory
- Requires at least logical association

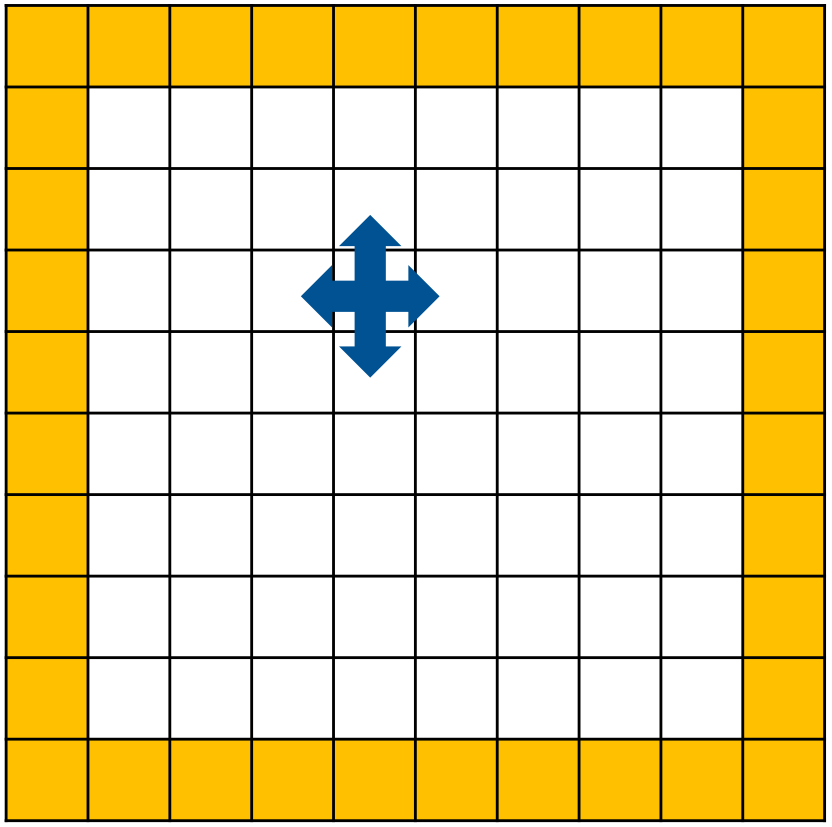
  “Know where your data is and where it is used”

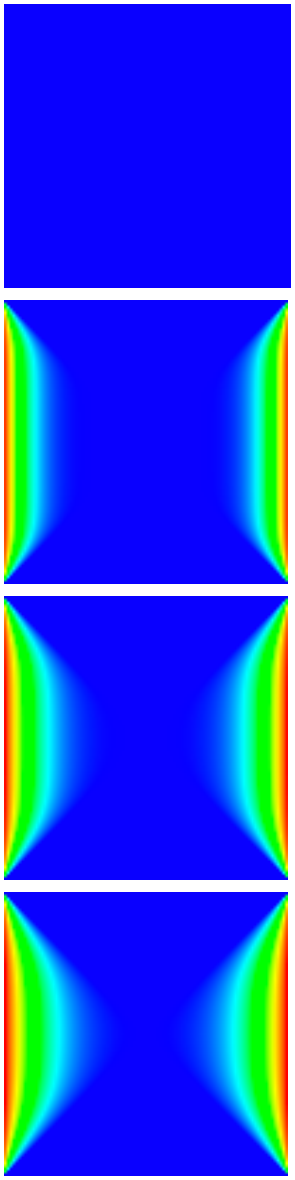
Example: Stencil Code

Example: Heat Diffusion

- Calculate heat distribution
- Solved using a stencil
- Example in 2D
- Dense matrix







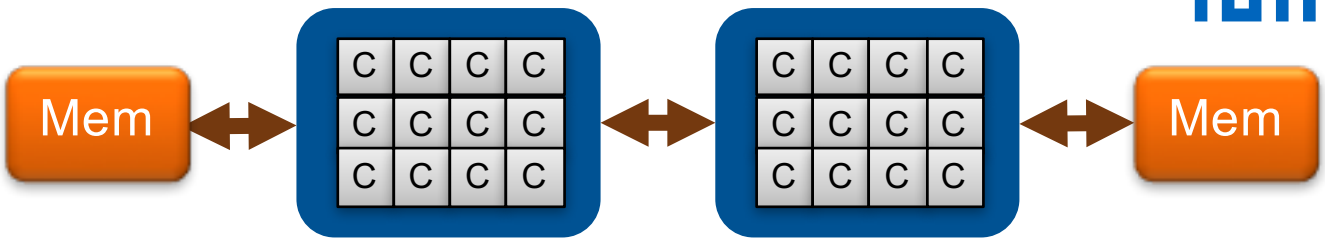
$$M(x,y) = \frac{M(x-1,y) + M(x+1,y) + M(x,y-1) + M(x,y+1)}{4}$$

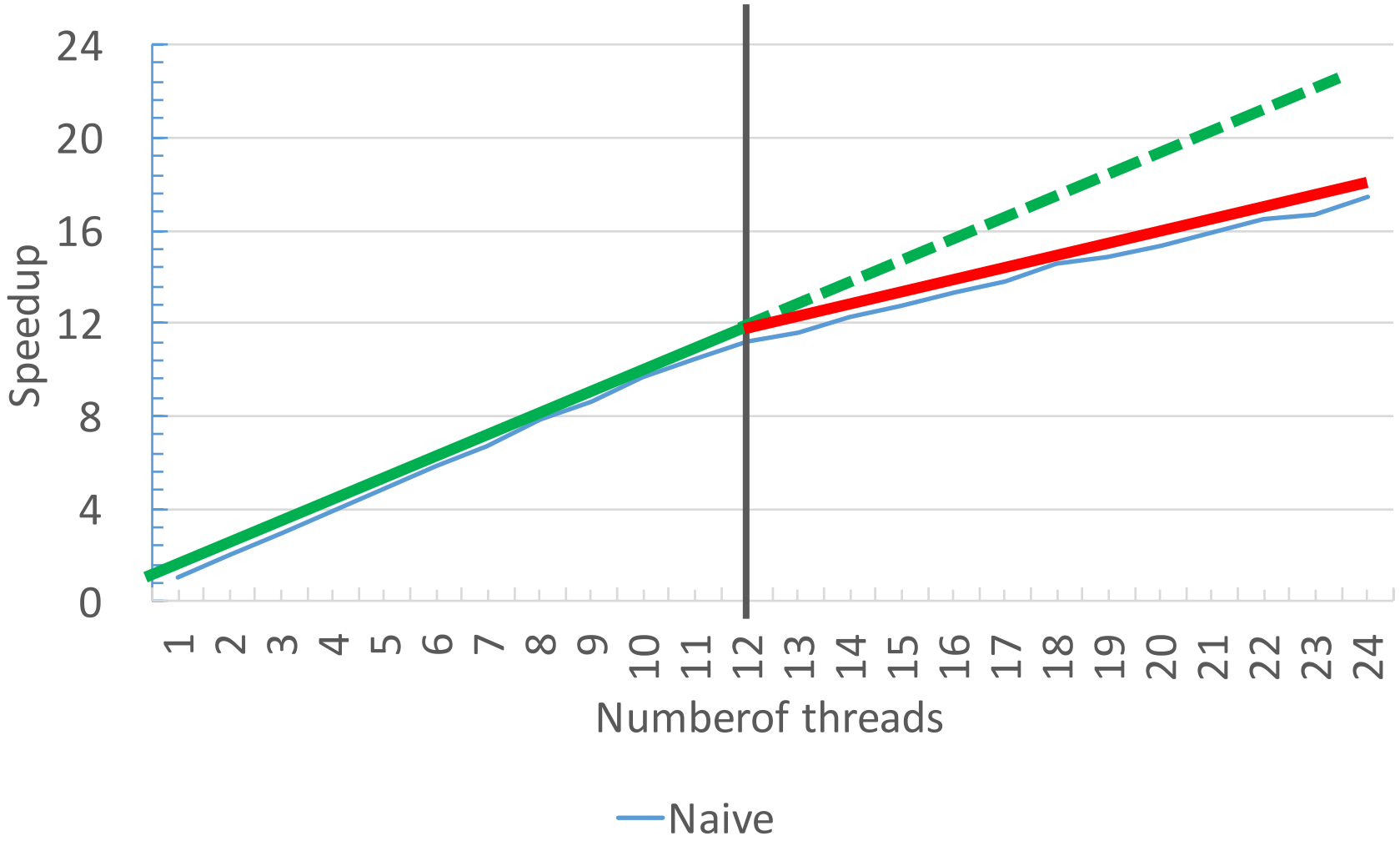
Example: Stencil Code

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 5000
#define ITER 100

double A[N+2][N+2];

int main(int argc, char** argv)
{
  double starttime,endtime;
  int threads;


  for (int i=0; i<N+2; i++)
    for (int j=0; j<N+2; j++)
      A[i][j]=0.0;

  for (int i=0; i<N+2; i++)
    A[i][0]=1.0;

  for (int i=0; i<N+2; i++)
    A[i][N+2]=1.0;
```

```
starttime=omp_get_wtime();

//Iteration
for (int n=0; n<100; n++)
{
  #pragma omp parallel
  {
    threads=omp_get_num_threads();

    #pragma omp for
    for (int i=1; i<N+1; i++)
      for (int j=1; j<N+1; j++)
          A[i][j]=(A[i+1][j]+A[i-1][j]+
              A[i][j-1]+A[i][j+1])/4.0;
   }
}

endtime=omp_get_wtime();

printf("threads: %i  time %f\n",
       threads,
       endtime-starttime);
return 0;
}
```

Performance





Placing Memory

Where does memory get physically located?

Typical policy: first touch
- Allocation only creates virtual space
- First access allocates physical memory
- Memory located close to accessing thread
- No later migration

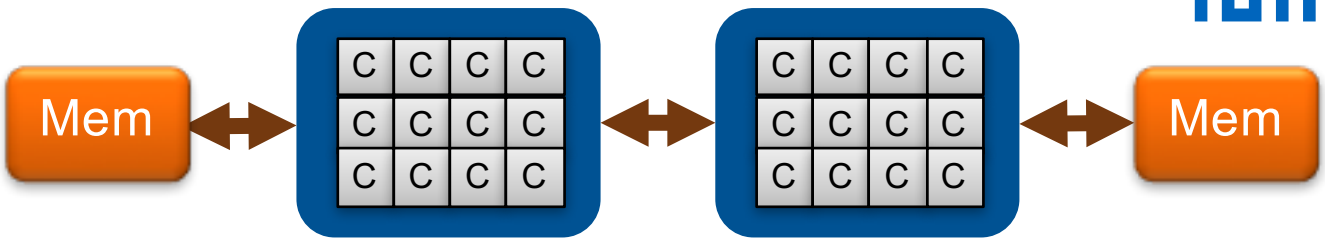
Consequences
- Initialization is important, as it determines locality
  - Use a parallel loop to initialize memory
- Should prevent threads from migrating later
  - Typical for HPC and default in OpenMP
  - Not common for usual thread programming
- Memory placement granularity is pages
  - Important if using large pages

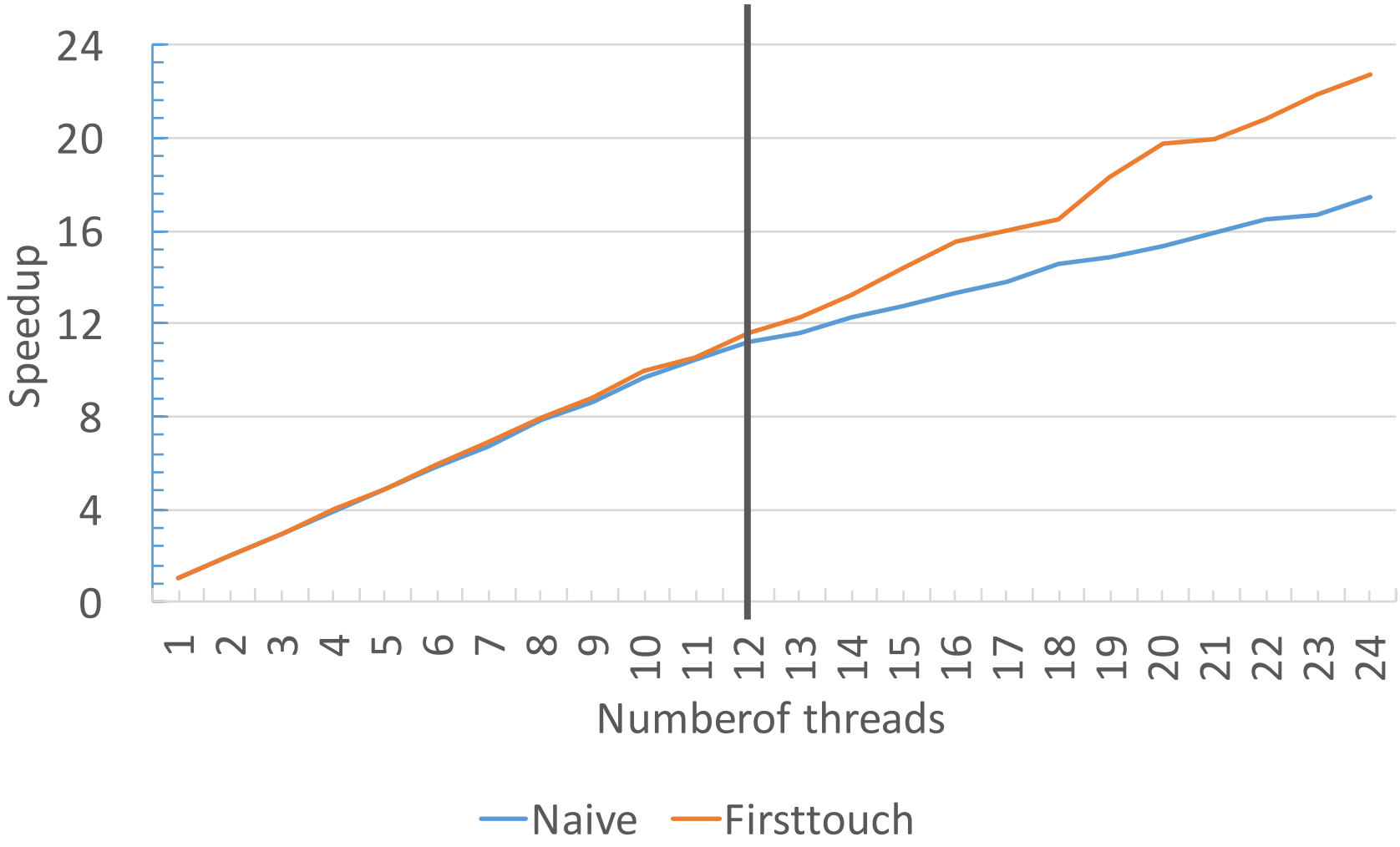
Example: Stencil Code

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 5000
#define ITER 100

double A[N+2][N+2];

int main(int argc, char** argv)
{
  double starttime,endtime;
  int threads;

  #pragma omp parallel for
  for (int i=0; i<N+2; i++)
    for (int j=0; j<N+2; j++)
      A[i][j]=0.0;

  for (int i=0; i<N+2; i++)
    A[i][0]=1.0;

  for (int i=0; i<N+2; i++)
    A[i][N+2]=1.0;
```

```
starttime=omp_get_wtime();

//Iteration
for (int n=0; n<100; n++)
{
  #pragma omp parallel
  {
    threads=omp_get_num_threads();

    #pragma omp for
    for (int i=1; i<N+1; i++)
      for (int j=1; j<N+1; j++)
          A[i][j]=(A[i+1][j]+A[i-1][j]+
              A[i][j-1]+A[i][j+1])/4.0;
   }
}

endtime=omp_get_wtime();

printf("threads: %i  time %f\n",
       threads,
       endtime-starttime);
return 0;
}
```

Performance





Thread Locations in OpenMP

Control by ICV **OMP_PROC_BIND**

- **true:** threads are bound to cores/hw-threads
- **false:** threads are not bound and can be migrated
- **master:** new threads are always co-located with master
- **close:** new threads are located “close” to their master threads
- **spread:** new threads are spread out as much as possible

Note

- If not defined or a different value, behavior is implementation defined
- Can also be list of values to define each nesting level

Description of available locations, also known as places: **OMP_PLACES** ICV

- Describes ordered list and hierarchy of all available hardware threads
- Example: `{0:4},{4:4},{8:4},{12:4}`
- Master thread executes on first place

NUMA Management via `libnuma`

Selection of routines to enable NUMA management for Linux

- Routines to influence thread location
- Routines to influence data location
- See: `man numa`

Examples

- Memory allocation
  - **`void *numa_alloc_onnode(size_t` *`size`*`, int` *`node`*`);`**
  - **`void *numa_alloc_local(size_t` *`size`*`);`**
- Thread binding
  - **`void numa_bind(struct bitmask *nodemask);`**
- Query routines
  - **`int numa_num_configured_nodes();`**

Summary: Aspects of "Good" Shared Memory Programming

Writing parallel code

- Aim at parallelizing the outer loop first
- Understand and resolve data dependencies
- Code transformations can help in removing loop carried dependecies
- Code transformations can also impact performance (good and bad!)

Correctness is mostly up to the programmer

- Little support from languages, runtimes or compilers
- Avoid race behaviors
- Tools can help in finding races

Performance is often tied directly to the underlying architecture

- Reduce or avoid synchronization
- Pay attention to the memory hierarchy
- Find good thread/data mappings in NUMA systems

Remember: May 7th – Guest lecture by Michael Klemm, CEO of the OpenMP ARB