Lecture: Memory Hierarchy and Cache Coherence

CSCE 569 Parallel Computing

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh

Parallelism in Hardware

- **Instruction-Level Parallelism**
  - **Pipeline**
  - **Out-of-order execution, and**
  - **Superscalar**

- **Thread-Level Parallelism**
  - **Chip multithreading, multicore**
  - **Coarse-grained and fine-grained multithreading**
  - **SMT**

- **Data-Level Parallelism**
  - **SIMD/Vector**
  - GPU/SIMT

Computer Architecture, A Quantitative Approach. 5TH Edition, The Morgan Kaufmann, September 30, 2011 by John L. Hennessy (Author), David A. Patterson

Topics

- ~~Programming on shared memory system (Chapter 7)~~
  - ~~***Cilk/Cilkplus and OpenMP Tasking***~~
  - ~~***PThread, mutual exclusion, locks, synchronizations***~~
- Parallel architectures and memory
  - **Parallel computer architectures**
    - **Thread Level Parallelism**
    - **Data Level Parallelism**
    - ~~**Synchronization**~~
  - ☛ **Memory hierarchy and cache coherency**
- Manycore GPU architectures and programming
  - **GPUs architectures**
  - **CUDA programming**
  - Introduction to offloading model in OpenMP

Outline

- **Memory, Locality of reference and Caching**
- Cache coherence in shared memory system

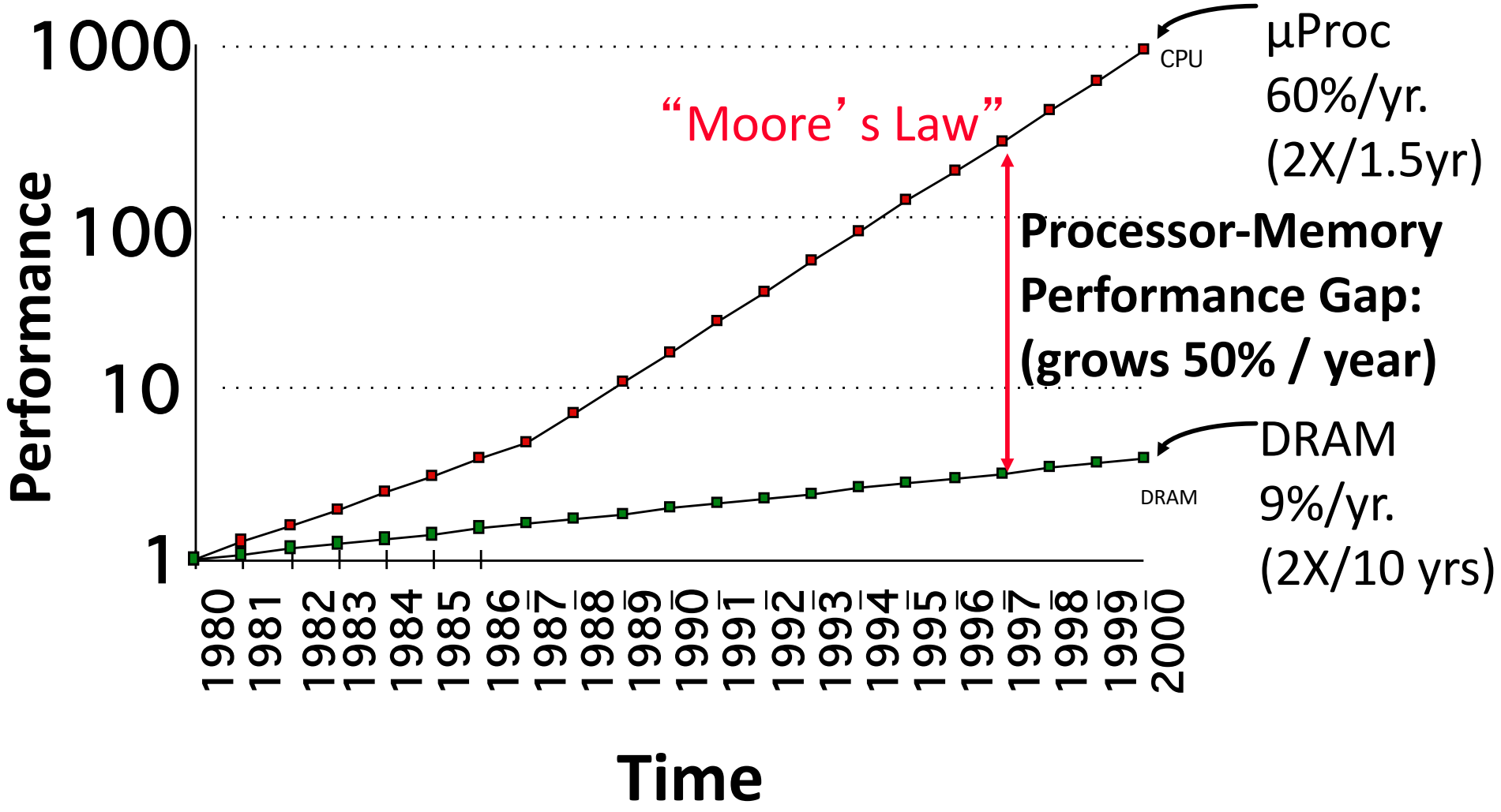
Memory until now …

- We've relied on a very simple model of memory for most this class
  - Main Memory is a linear array of bytes that can be accessed given a memory address
  - Also used registers to store values

- Reality is more complex. There is an entire memory system.
  - Different memories exist at different levels of the computer
  - Each vary in their speed, size, and cost

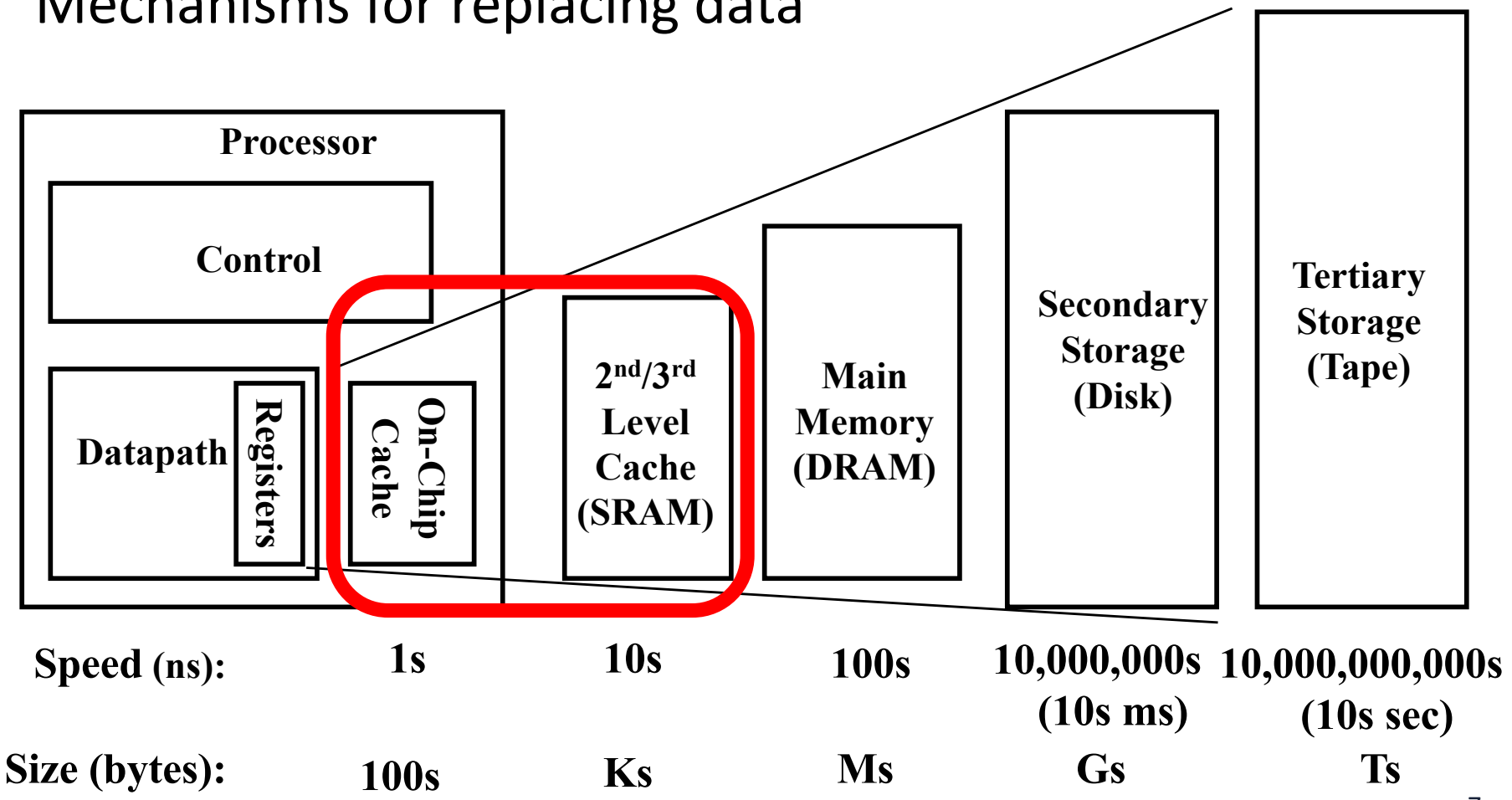
Why Cares About the Memory Hierarchy?

Processor-DRAM Memory Gap (latency) → Memory Wall



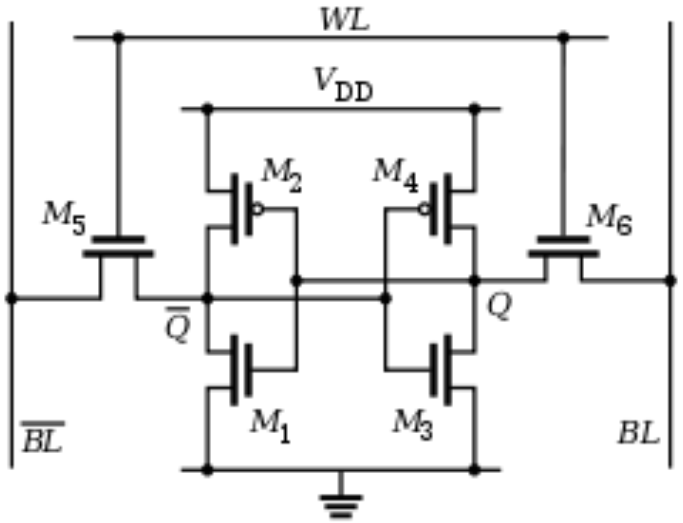
Architecture Approach: Memory Hierarchy

- Keep **most recent accessed data** and **its adjacent data** in the smaller/faster caches that are closer to processor
- Mechanisms for replacing data



Random-Access Memory (RAM)

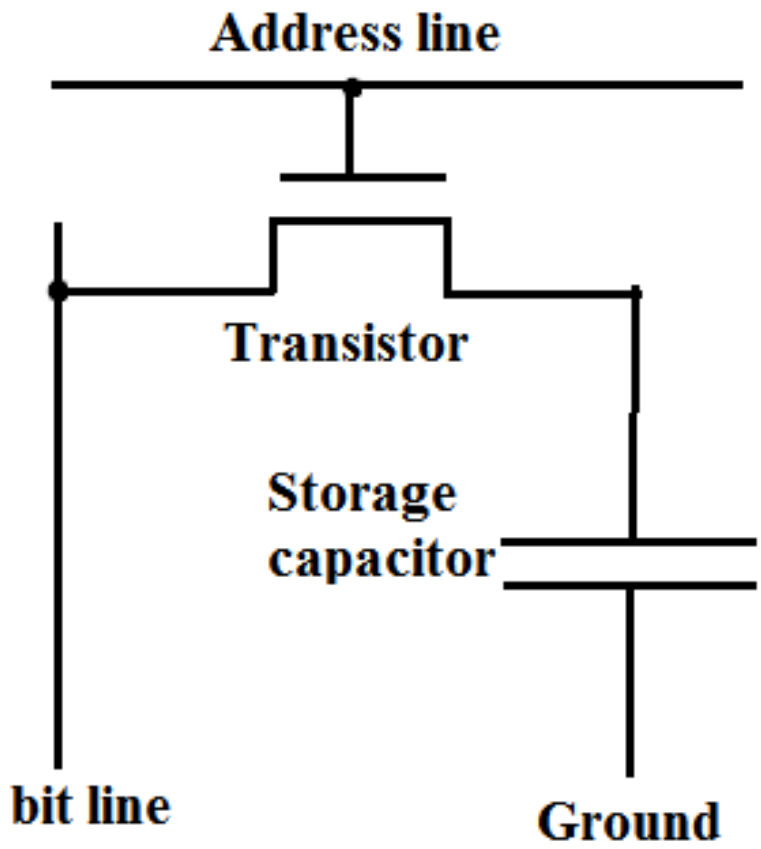
- Key features
  - RAM is packaged as a chip.
  - Basic storage unit is a cell (one bit per cell).
  - Multiple RAM chips form a memory.



- **Static RAM (SRAM)**
  - Each cell stores bit with a six-transistor circuit.
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to disturbances such as electrical noise.
  - Faster and more expensive than DRAM.

Random-Access Memory (RAM)

- **Dynamic RAM (DRAM)**
  - Each cell stores bit with a capacitor and transistor.
  - Value must be refreshed every 10-100 ms.
  - Sensitive to disturbances.
  - Slower and cheaper than SRAM.



How to Exploit Memory Hierarchy: Program Behavior: Principle of Locality

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time
  - Temporal locality: Recently referenced items are likely to be referenced in the near future

Locality Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data**
  - **Reference array elements in succession (stride-1 reference pattern):** **Spatial locality**
  - **Reference `sum` each iteration:** **Temporal locality**
- **Instructions**
  - **Reference instructions in sequence:** **Spatial locality**
  - **Cycle through loop repeatedly:** **Temporal locality**

Sources of locality

- Temporal locality
  - Code within a loop
  - Same instructions fetched repeatedly
- Spatial locality
  - Data arrays
  - Local variables in stack
  - Data allocated in chunks (contiguous bytes)

```
for (i=0; i<N; i++) {
    A[i] = B[i] + C[i] * a;
}
```

Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

Matrix Multiplication Example

- Major cache effects to consider
  - Total cache size
    - Exploit temporal locality and blocking)
  - Block size
    - Exploit spatial locality

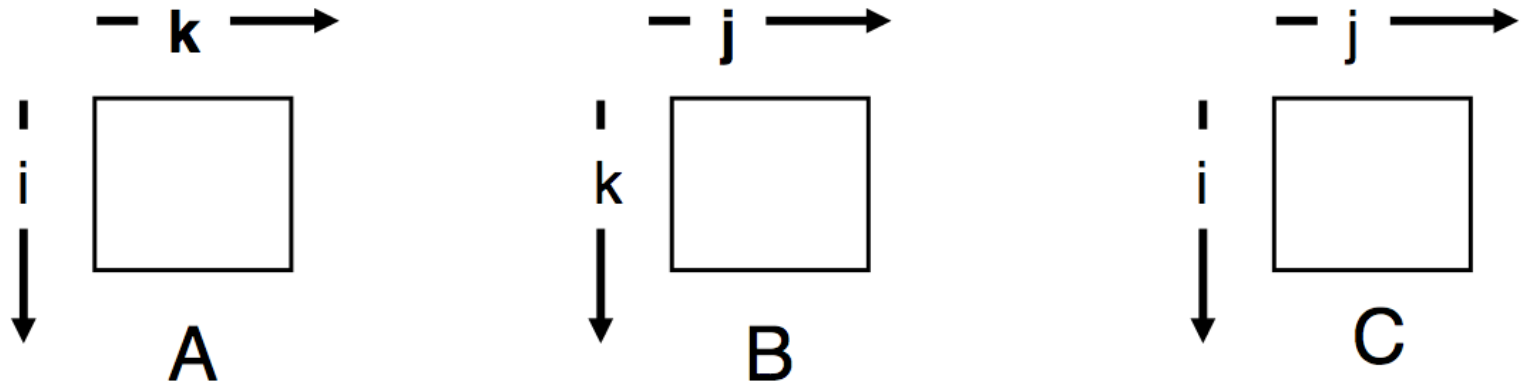
- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable ***sum*** *held in register*

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Cache line size = 32 Bytes (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis method:
  - Look at access pattern of inner loop

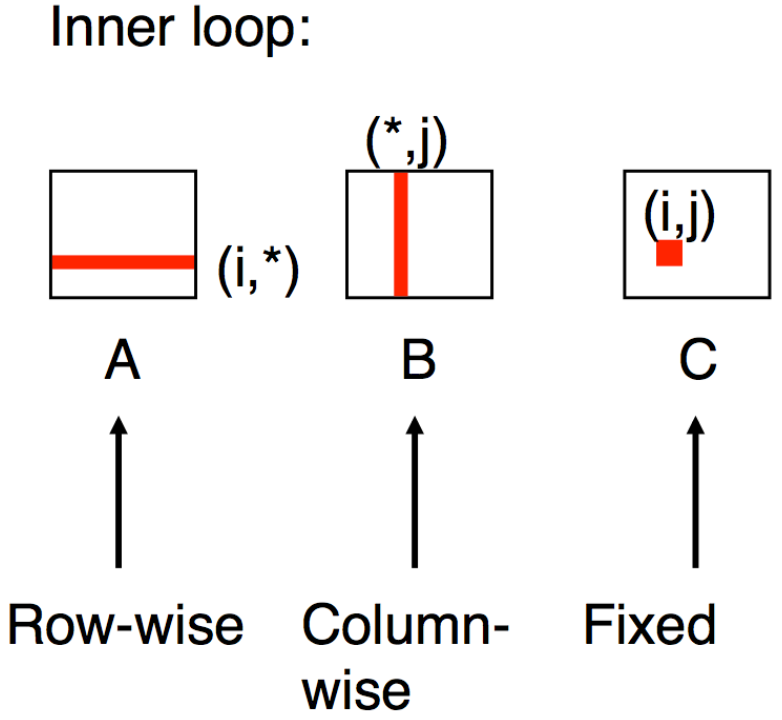


Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for(i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
  - `for(i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

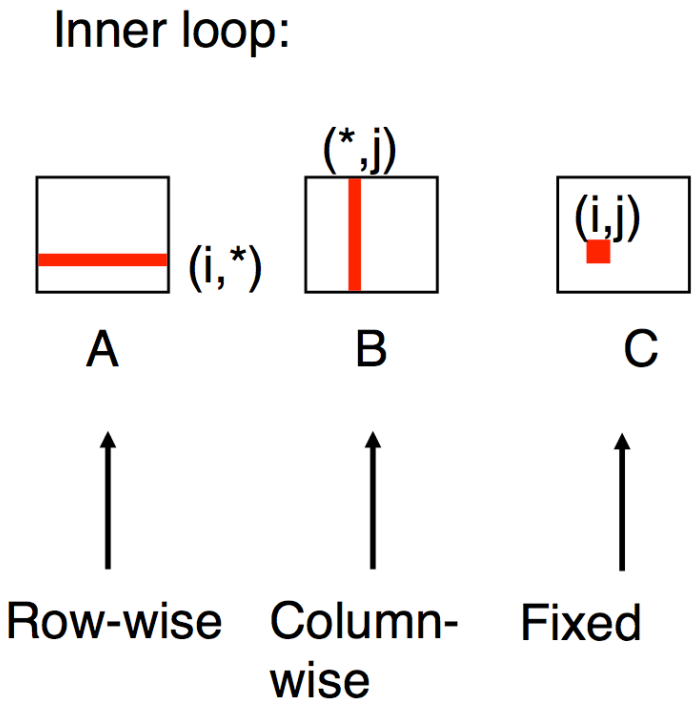


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

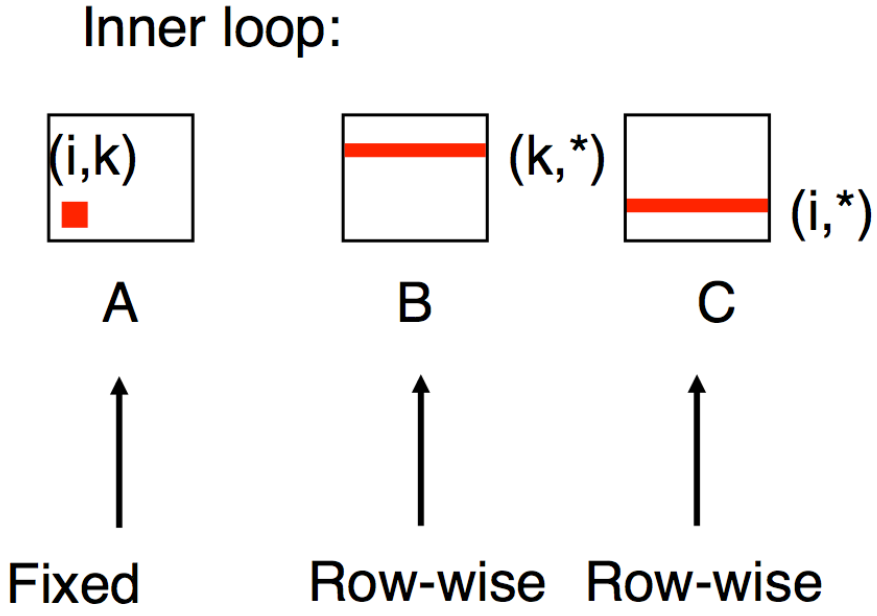


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

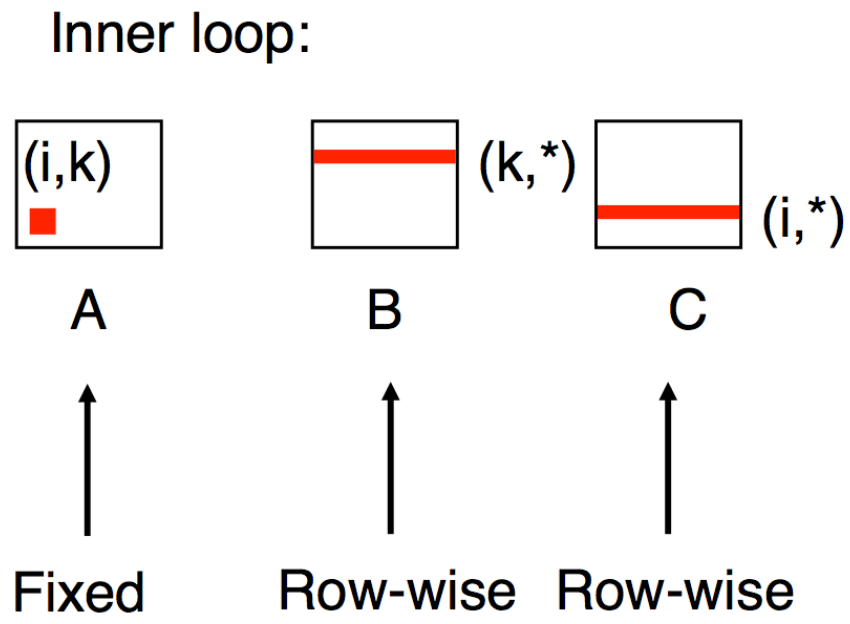


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

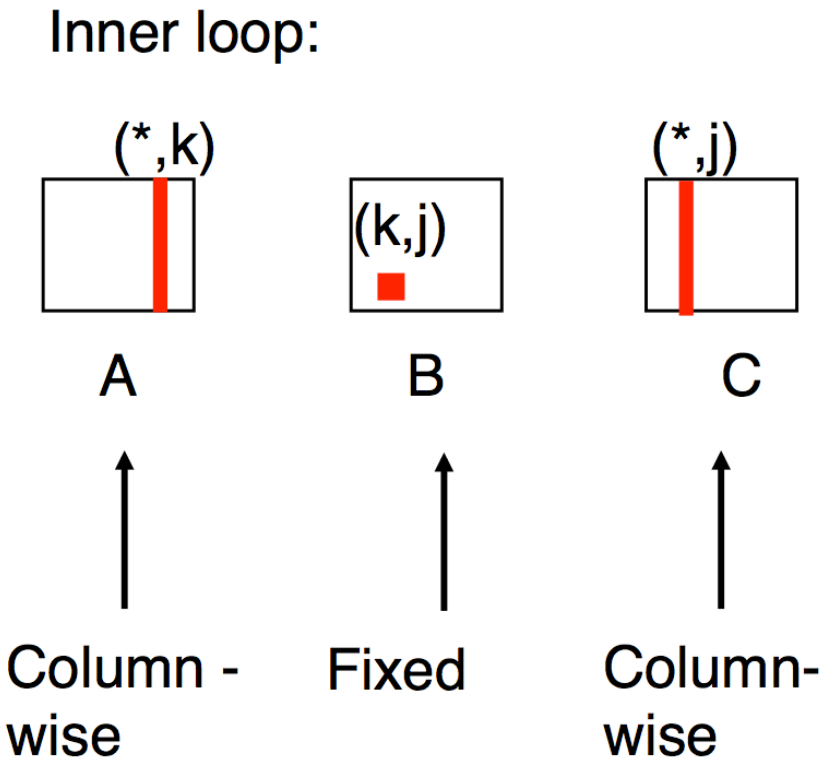


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

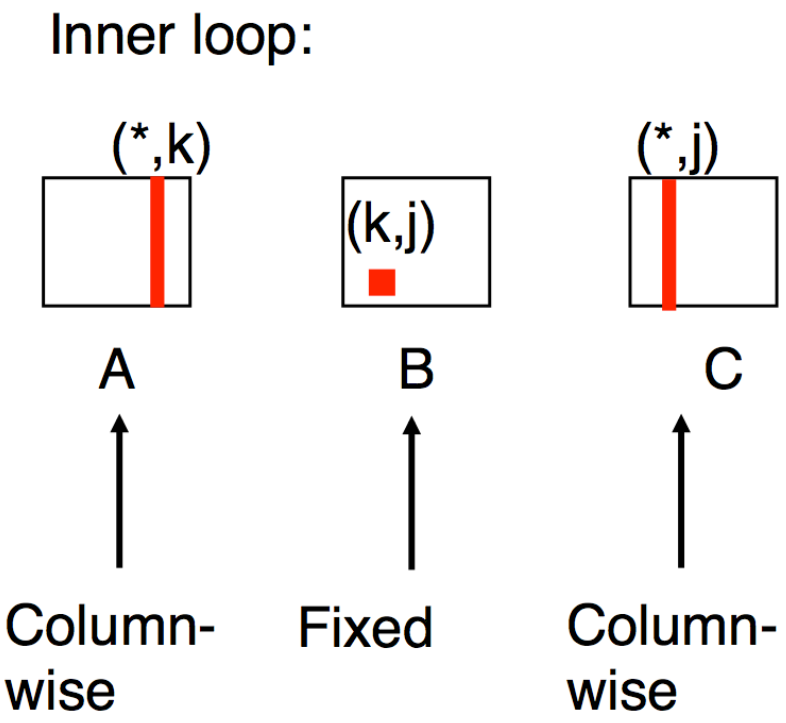


- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (i=0; i<n; i++)  {
   for (j=0; j<n; j++) {
      sum = 0.0;
      for (k=0; k<n; k++)
         sum += a[i][k] *
      b[k][j]; c[i][j] = sum;
   }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++) {
      r = b[k][j];
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * r;
   }
}
```

Outline

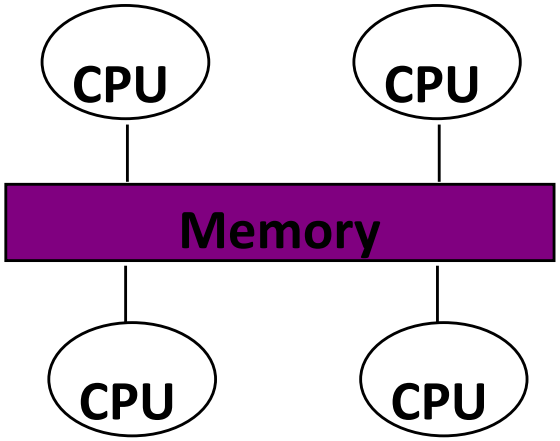
- Memory, Locality of reference and Caching
- **Cache coherence in shared memory system**

Shared Memory Systems

- All processes have access to the same address space
  - E.g. PC with more than one processor
- Data exchange between processes by writing/reading shared variables
  - Shared memory systems are easy to program
  - Current standard in scientific programming: OpenMP
- Two versions of shared memory systems available today
  - Centralized Shared Memory Architectures
  - Distributed Shared Memory architectures

Centralized Shared Memory Architecture

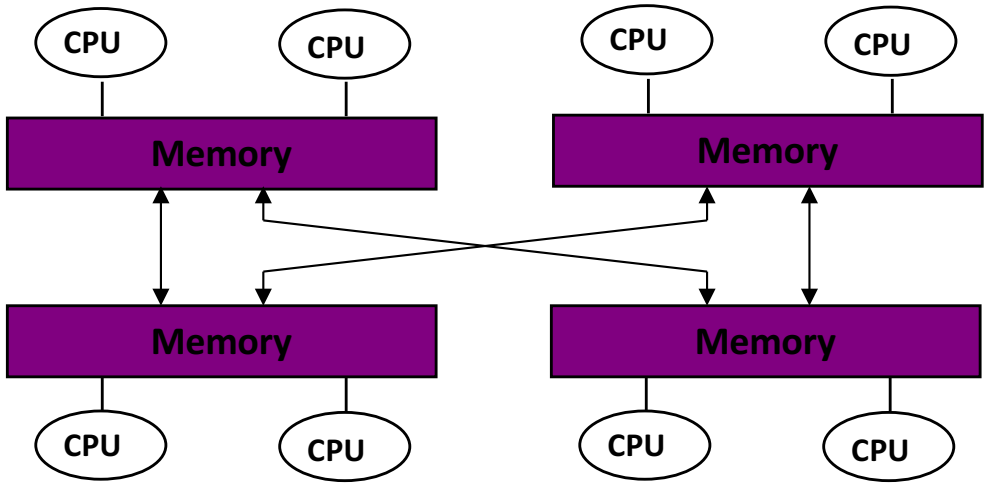
- Also referred to as Symmetric Multi-Processors (SMP)
- All processors share the same physical main memory



- Memory bandwidth per processor is limiting factor for this type of architecture
- Typical size: 2-32 processors

Distributed Shared Memory Architectures

- Also referred to as Non-Uniform Memory Architectures (NUMA)
- Some memory is closer to a certain processor than other memory
  - The whole memory is still addressable from all processors
  - Depending on what data item a processor retrieves, the access time might vary strongly



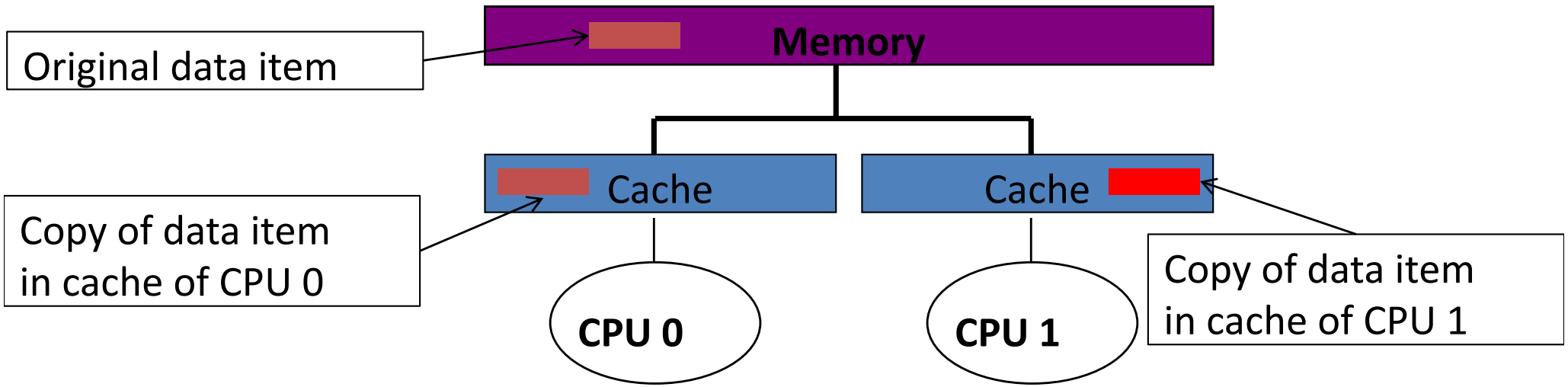
NUMA Architectures (II)

- Reduces the memory bottleneck compared to SMPs
- More difficult to program efficiently
  - E.g. first touch policy: data item will be located in the memory of the processor which uses a data item first
- To reduce effects of non-uniform memory access, caches are often used
  - ccNUMA: cache-coherent non-uniform memory access architectures
- Largest example as of today: SGI Origin with 512 processors

Distributed Shared Memory Systems

Cache Coherence

- Real-world shared memory systems have caches between memory and CPU
- Copies of a single data item can exist in multiple caches
- Modification of a shared data item by one CPU leads to outdated copies in the cache of another CPU



Cache Coherence (II)

- Typical solution:
  - Caches keep track on whether a data item is shared between multiple processes
  - Upon modification of a shared data item, ‘notification’ of other caches has to occur
  - Other caches will have to reload the shared data item on the next access into their cache
- Cache coherence is only an issue in case multiple tasks access the same item
  - Multiple threads
  - Multiple processes have a joint shared memory segment
  - Process is being migrated from one CPU to another

Cache Coherence Protocols

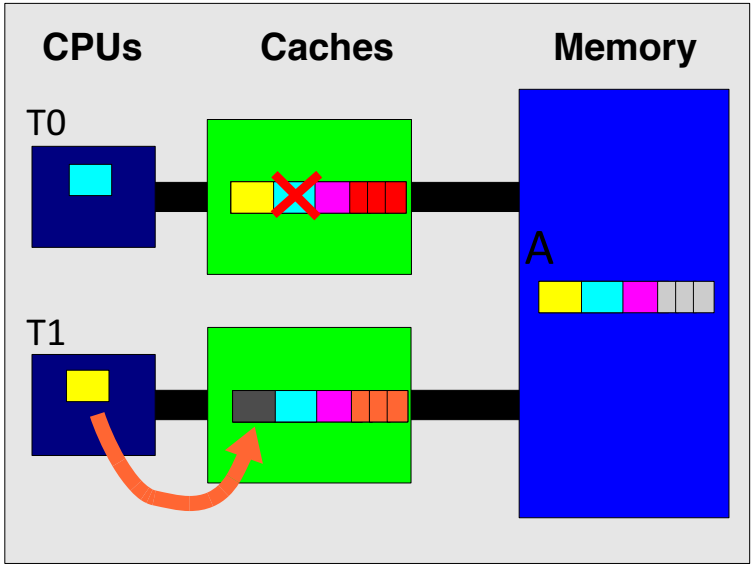
- Snooping Protocols
  - Send all requests for data to all processors
  - Processors snoop a bus to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for centralized shared memory machines
- Directory-Based Protocols
  - Keep track of what is being shared in centralized location
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Commonly used for distributed shared memory machines

Categories of Cache Misses

- Up to now:
  - Compulsory Misses: first access to a block cannot be in the cache (cold start misses)
  - Capacity Misses: cache cannot contain all blocks required for the execution
  - Conflict Misses: cache block has to be discarded because of block replacement strategy
- In multi-processor systems:
  - Coherence Misses: cache block has to be discarded because another processor modified the content
    - true sharing miss: another processor modified the content of the request element
    - false sharing miss: another processor invalidated the block, although the actual item of interest is unchanged.

False Sharing in OpenMP

- False sharing
  - When at least one thread write to a cache line while others access it
    - Thread 0: = A[1] (read)
    - Thread 1: A[0] = … (write)
- Solution: use array padding

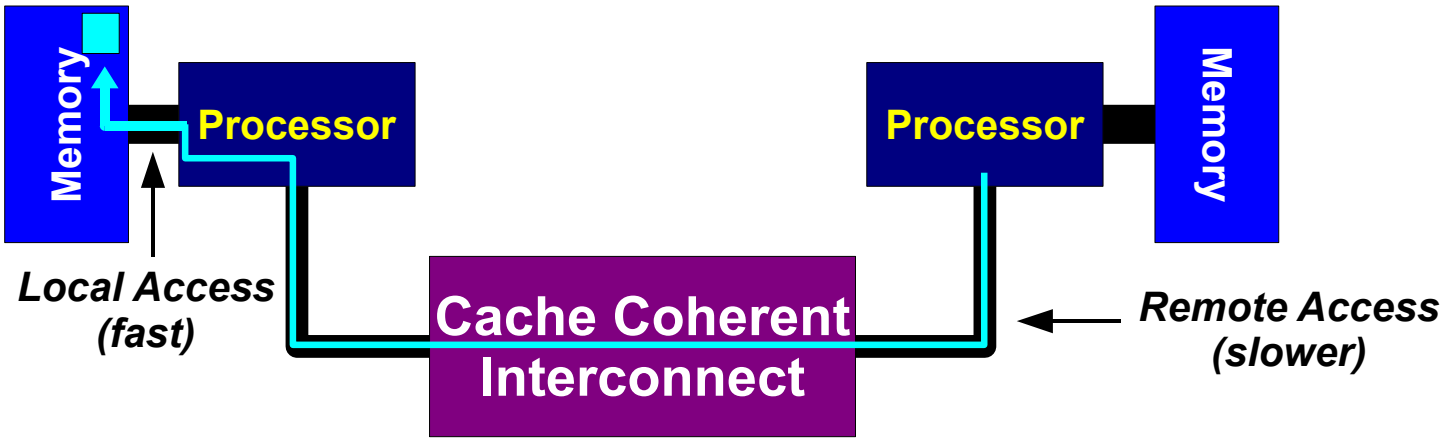



```
int a[max_threads];
#pragma omp parallel for schedule(static,1)
for(int i=0; i<max_threads; i++)
    a[i] +=i;
```

```
int a[max_threads][cache_line_size];
#pragma omp parallel for schedule(static,1)
for(int i=0; i<max_threads; i++)
    a[i][0] +=i;
```

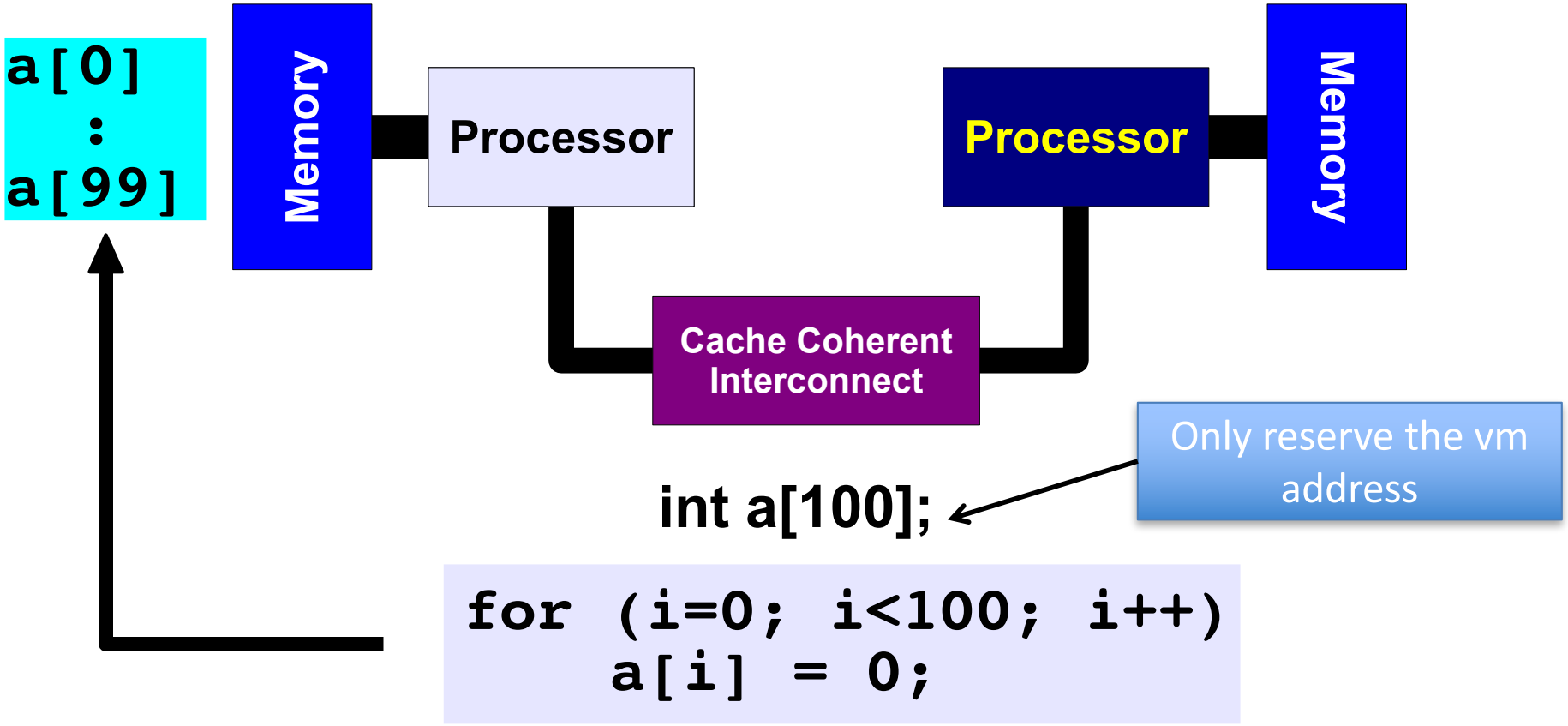
NUMA and First Touch Policy

- Data placement policy on NUMA architectures



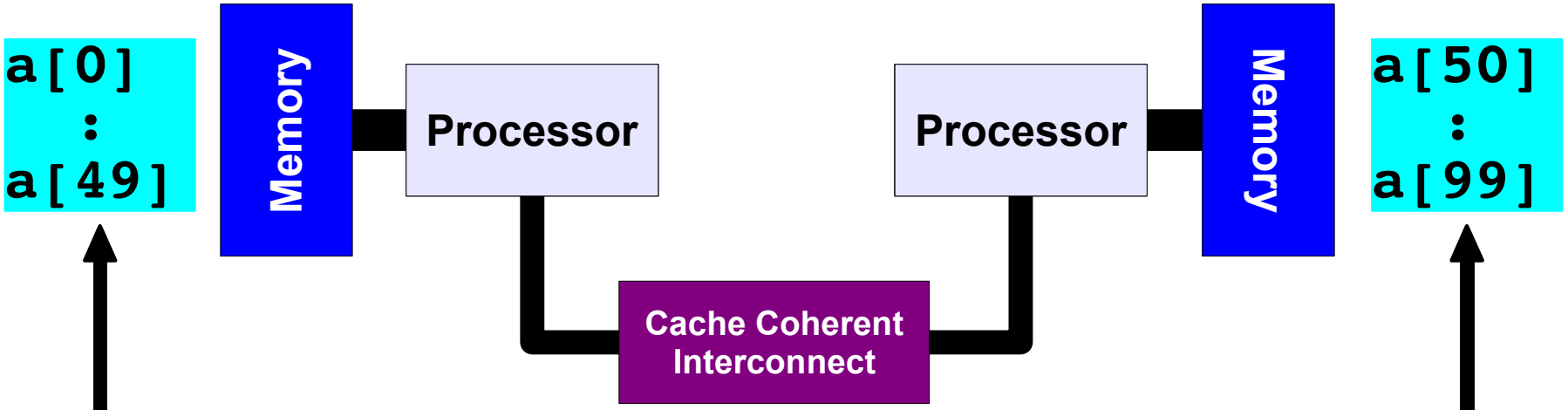
- First Touch Policy
  - The process that first touches a page of memory causes that page to be allocated in the node on which the process is running

NUMA First-touch Placement/1



First Touch
All array elements are in the memory of the processor executing this thread

NUMA First-touch Placement/2




```
#pragma omp parallel for num_threads(2)
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
Both memories each have "their half" of the array

Work with First-Touch in OpenMP

- First-touch in practice
  - Initialize data consistently with the computations

```
#pragma omp parallel for
for(i=0; i<N; i++) {
   a[i] = 0.0; b[i] = 0.0 ; c[i] = 0.0;
}
readfile(a,b,c);

#pragma omp parallel for
for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
  - Work with cache coherence protocol and NUMA first touch policy

References

- Computer Architecture, A Quantitative Approach. 5TH Edition, The Morgan Kaufmann, September 30, 2011 by John L. Hennessy (Author), David A. Patterson
- A Primer on Memory Consistency and Cache Coherence Daniel J. Sorin Mark D. Hill David A. Wood, SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE Mark D. Hill, Series Editor, 2011