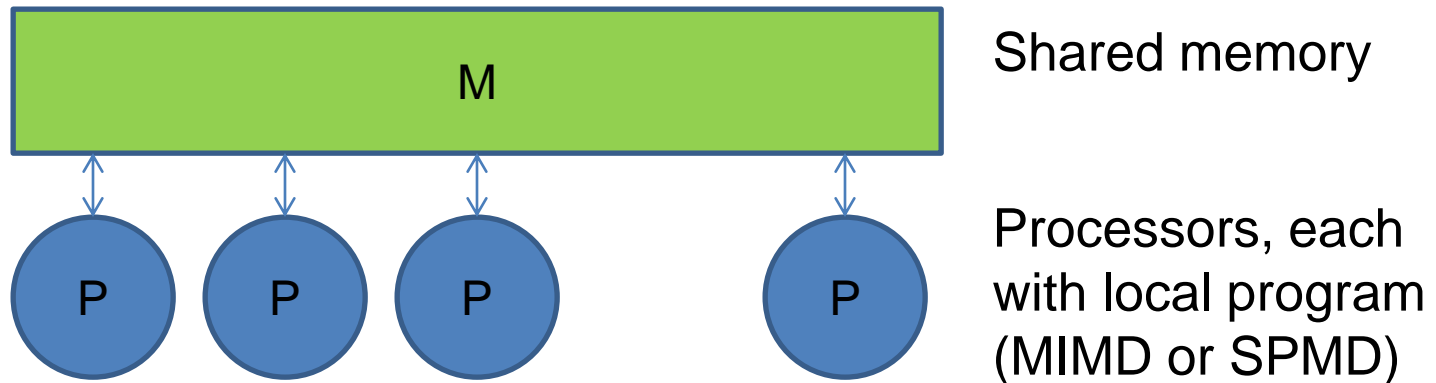


Introduction to Parallel Computing

Shared-memory systems

Jesper Larsson Träff
traff@par.tuwien.ac.at
TU Wien
Parallel Computing

Shared-memory architectures & machines



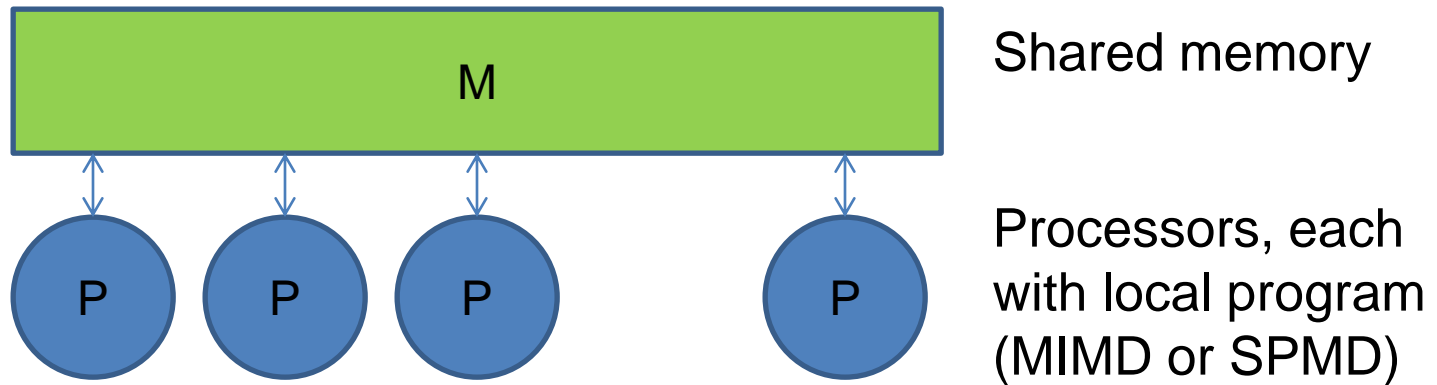
Naïve, shared-memory architecture model:

- Processors connected immediately to a memory (UMA or NUMA), all communication via read/write from/to memory, processors work independently (**not** synchronized, **not** lock-step: **not PRAM**)

Implicit assumption: Memory is consistent as defined by program order; writes and reads occur and are observable in that order.

Outcome of computation is some interleaving of the instructions

A bit
more
later



Naïve, shared-memory programming model:

- processors execute processes or threads (inside processes),
- processes/threads are not synchronized,
- methods for process/thread synchronization,
- processes/threads exchange data through shared memory,
- methods for sharing memory between processes/threads,
- NUMA, but all memory directly visible (memory model)

Naive, shared-memory programming model:

a) processors execute processes or threads

Note:

Naïve model does not say which processor (core) executes which process or thread. There may be more processes/threads than cores (“oversubscription”).

A bit more later

Symmetric Multiprocessing (SMP): All cores equal, OS decides when and where to run a process or thread

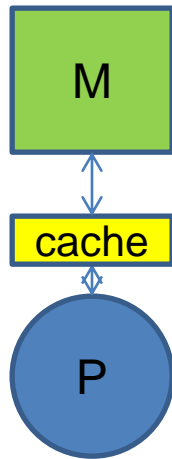
Naive, shared-memory programming model:

a) processors execute processes or threads

Performance:

Sometimes SMP gives good performance (flexibility for OS scheduler), sometimes it is (much) better to explicitly bind processes to cores (main example: utilization of private caches). This is called “pinning”. Most shared-memory programming models support/allow some kind of pinning

Parallel computing (dedicated system): As many processes/threads as cores, some pinning



“Real” (shared-memory) architectures

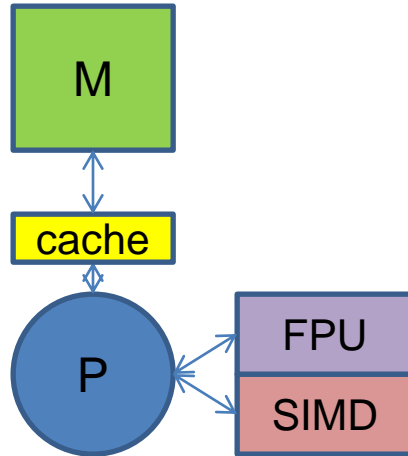
Cache: small, fast memory, close to processor, accessed main memory locations are stored temporarily in cache, reused when possible

Caches help to alleviate/hide memory (“von Neumann”) bottleneck

- Main memory: GBytes, access times > 100 cycles
- Cache: Kbytes to MBytes, access times, 1-20 cycles
- Registers: 0-1 cycles

Typically 2-3 levels of caches in modern processors, and several special caches, TLB, victim cache, instruction cache, ...

“Real” (shared-memory) architectures

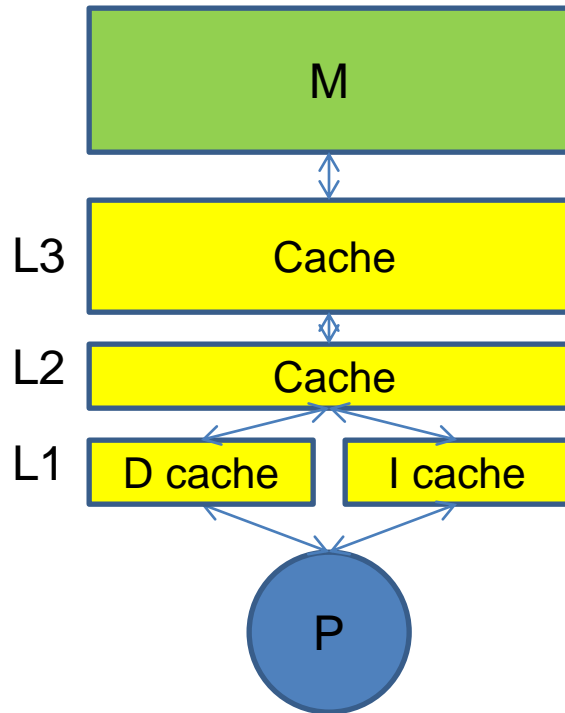


Not all instructions may take same time:

- Floating point operations longer than integer operations?
- Operations on 64bit entities longer than 32bit entities?

Special HW and instructions for small vectors: SIMD extensions to perform operations on several (2-4) words in parallel (SSE: 128 bits, Xeon Phi: 512 bits; AVX)

To expose SIMD parallelism in loops: Unrolling (compiler, OpenMP). Sometimes different algorithm needed (no help from compiler)



“Real” (shared-memory) architectures

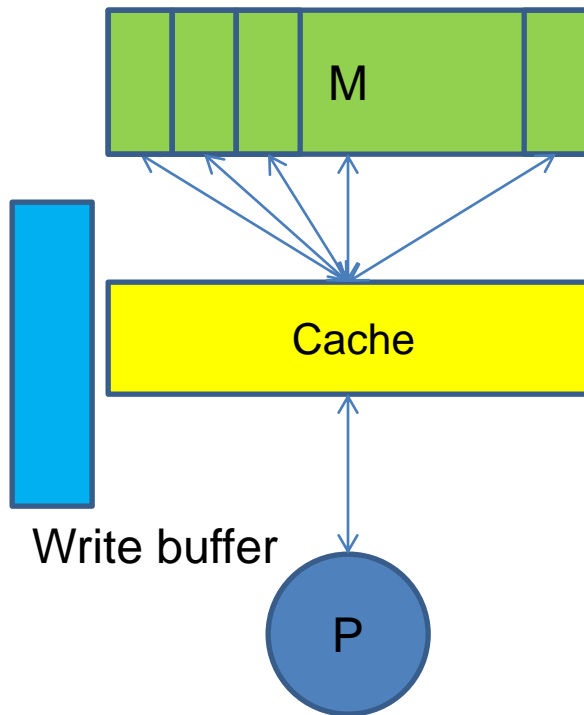
Cache: Small, fast memory, close to processor, accessed main memory locations are stored temporarily in cache, reused when possible

Caches help to alleviate/hide memory (“von Neumann”) bottleneck

- Main memory: GBytes, access times > 100 cycles
- Cache: Kbytes to MBytes, access times, 1-20 cycles

Typically 2-3 levels of caches in modern processors, and several special caches, TLB, victim cache, instruction cache, ...

Caches at higher levels shared between (some, all) cores



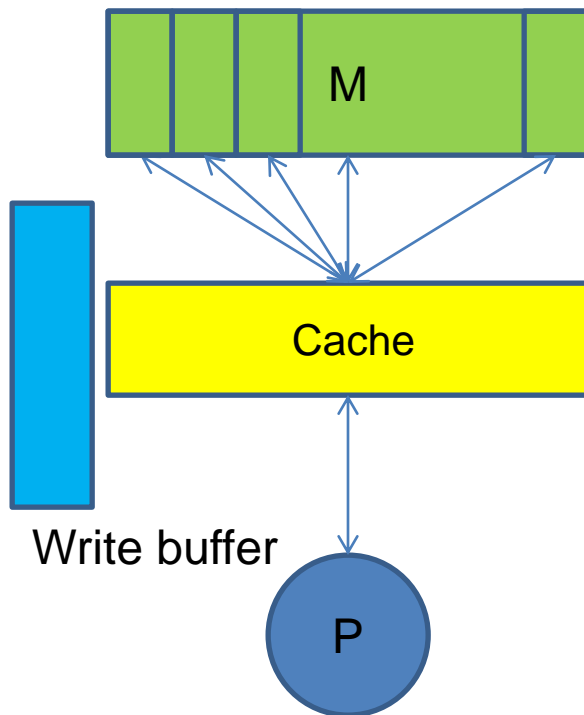
“Real” (shared-memory) architectures

Memory divided into banks, accessed through memory controllers and network: NUMA (non-uniform access times)

Write buffer help to alleviate write memory bottleneck: Pending writes (already in cache) effected in some order

Single processor architecture: Cache and memory system contribute toward sustaining the RAM abstraction (illusion)

... and have been immensely successful!



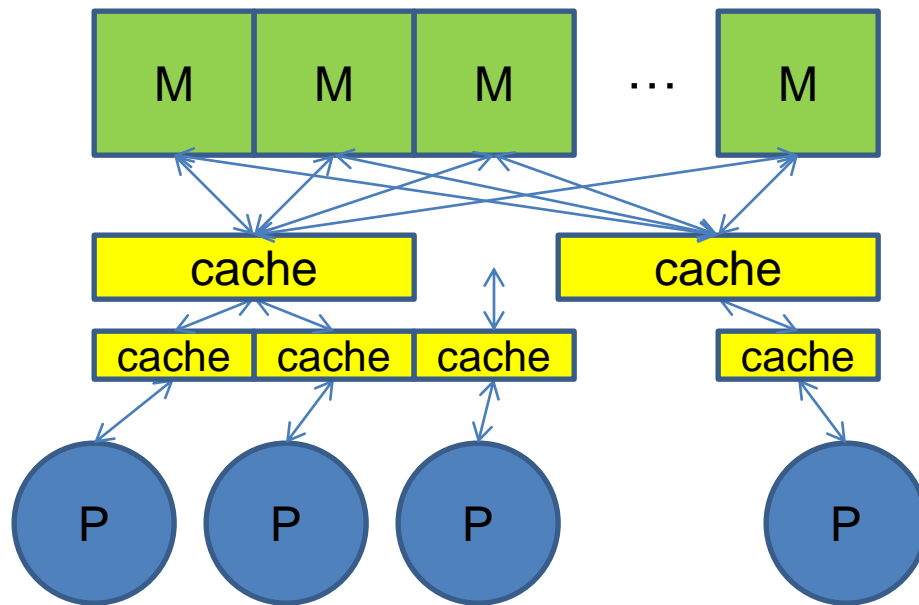
“Real” (shared-memory) architectures

Memory divided into banks, accessed through memory controllers and network: NUMA (non-uniform access times)

Write buffer help to alleviate write memory bottleneck: Pending writes (already in cache) effected in some order

Parallel processor-system architecture: Cache and memory system contribute toward sustaining the RAM abstraction (illusion)

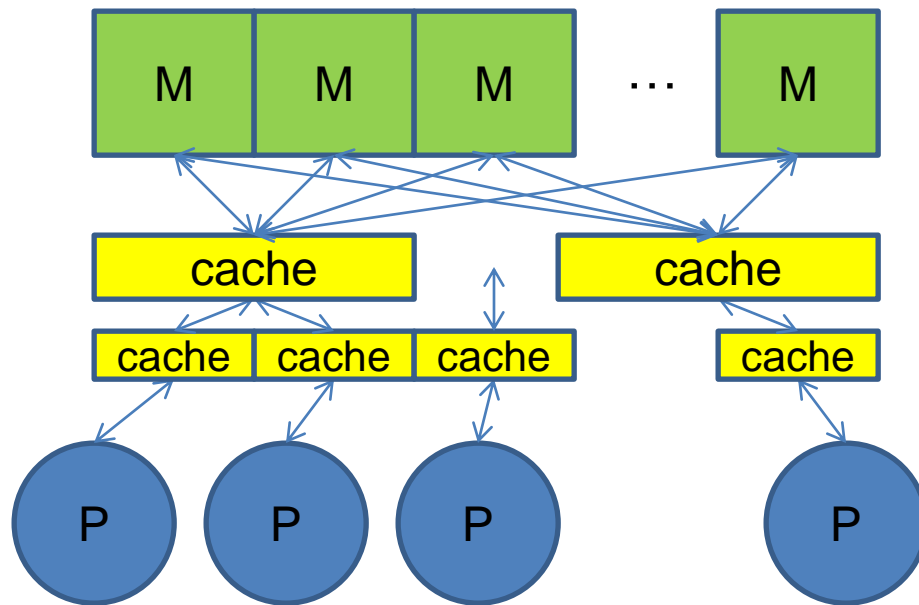
... and become problematic



“Real” parallel, shared-memory architectures

Problems:

- What happens when same memory address in several caches?
- When do memory write updates become “visible” to other processors?

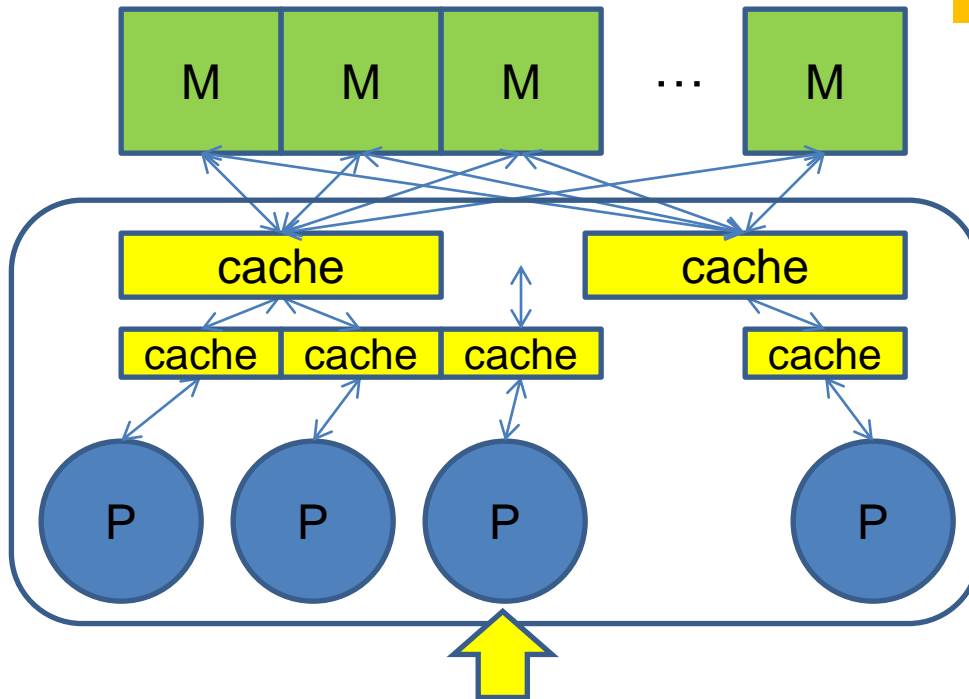


“Real” parallel, shared memory architectures

Problems:

- Cache coherency problem: What happens when processors read/write the same address?
- Memory consistency problem: What happens when processors interact (read/write) through different addresses?

“Modern” Terminology



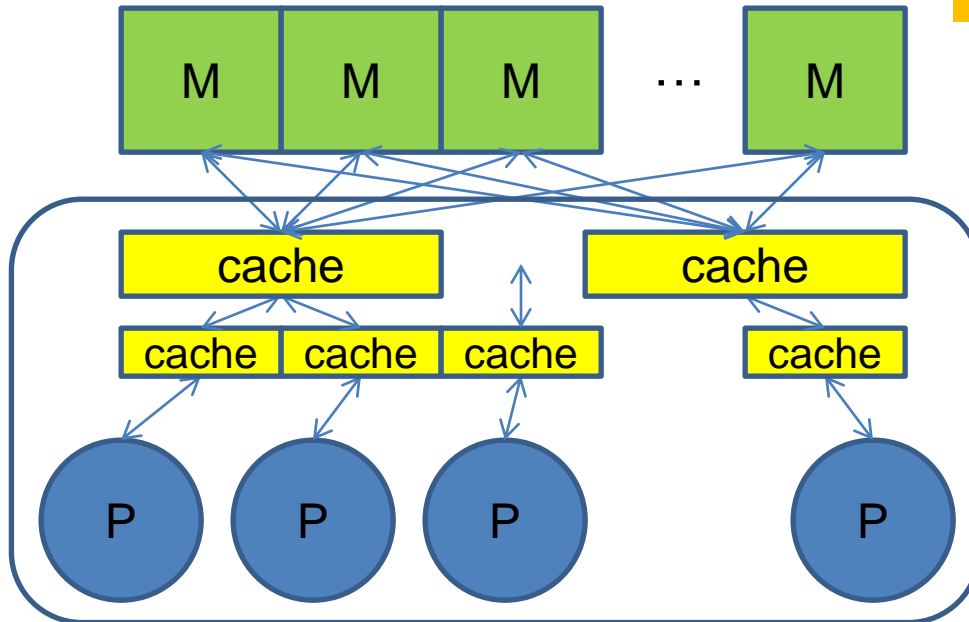
The processor (**CPU?**):
 Can consist of several cores
 or **processing elements** (PE):
 That which is capable of
 executing a program

Core (**CPU?**): A full-fledged processing-element (still sometimes called “processor”), hardware-term

Multi-core: Processor with several cores (2-32...)

Many-core: Processor with very many cores?? (some use these terms as in GPU: many-core, CPU: multi-core)

“Modern” Terminology



The processor:
 Can consist of several cores
 or **processing elements** (PE):
 That which is capable of
 executing a program

Thread, process: Programming model concepts, program under execution

Task: Programming model concept, some “piece of work” to be executed (by a process or thread)

Caches, recap

Cache consists of a number of lines that stores blocks of memory. A cache line holds a block and additional status information (dirty/valid bit, tag)

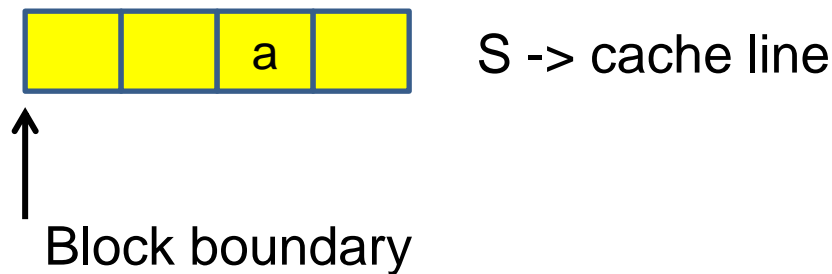
Typical block/cache line size: 64Bytes (=8 doubles, =16 ints)

Caches exploit and make sense because of:

- Temporal locality: Locations are typically used several times in close succession, several operations on same operand (address)
- Spatial locality: When a location is addressed, typically locations close to it ($a[0]$, $a[1]$, $a[2]$, ...) will be also be used

Locality is a property of algorithm/program: Often, but not always

Access to main memory in block/cache line size of S units, aligned to block boundary



Memory **read a**:

If address a already in cache, reuse from there, if not: read from memory and cache (**read cache miss**)

Whole blocks are read/written to/from cache (**granularity**)

Memory **write a**:

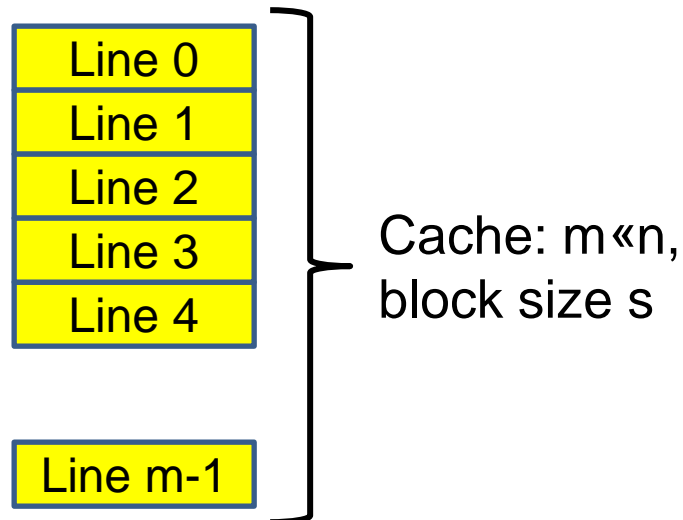
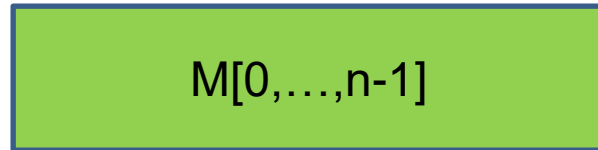
different possibilities. If a is already in cache, write overwrites; if a is not in cache (**write cache miss**)

- **Write allocate**: If a is not in cache, read a
- **Write non-allocate**: Write directly to memory

- **Write-through cache**: Each write is immediately passed on to memory (typically non-allocate)
- **Write back**: Cache line block is written back when line is evicted (typically write allocate)

Address a:

- If a can go into only one specific line of the cache: **Directly mapped**
- If a can go into any line of the cache: **Fully associative**
- If a can go into any of a small set of lines: **Set-associative** (typically 2-way, 4-way, 8-way)



Directly mapped:

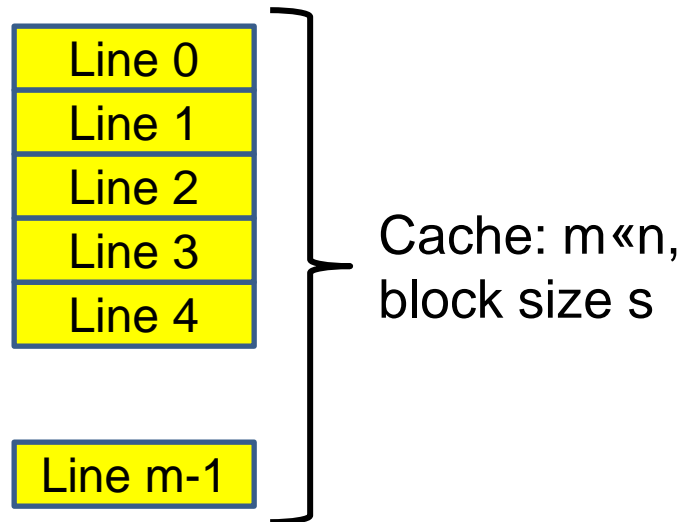
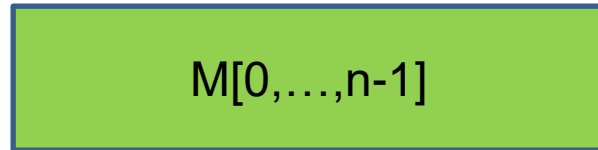
$M[i]$ cached in line $(i/s) \bmod m$

$M[0], \dots, M[s-1]$ go to line 0,
 $M[s], \dots, M[2s-1]$ to line 1,

...

$M[m], \dots, M[m+s-1]$ again to
line 0, ...

Normally $m=2^r$ for some $r>0$, $s=2^r$ (powers of two: mod and div translate into mask and shift)



k-way set associative:
m/k sets of k cache lines

$M[i]$ cached in set $(i/s) \bmod (m/k)$

Normally $m=2^r$ for some $r>0$, $s=2^r$ (powers of two: mod and div translate into mask and shift)

Cache misses, eviction: When a referenced address is not in cache, a **cache miss** occurs (read or write); address must be referenced from memory

Cold cache: Cache is empty, every (read) memory access will be a **cache miss**, called compulsory miss (or cold miss)

When cache has been in use (“warm cache”), two other types of cache misses can happen

- **capacity miss**: Cache full, some line must be evicted
- **conflict miss**: set (or specific cache line) full, but **cache itself not necessarily full**; line from set must be evicted

Hit (miss) rate: Fraction of memory references over a sequence of instructions that hits (misses) the cache

In case of capacity/conflict **miss**:

Evict cache line to make room for new block

For directly mapped cached, conflicting line is evicted

Eviction/Replacement policies for associative caches

- LRU: Least Recently Used
- LFU: Least Frequently Used

- Random replacement

Caches are typically maintained in hardware (functionally transparent), “free lunch”

(...but can and do impact performance)

Cache matters for (sequential¶llel) performance

Matrix-matrix multiplication: Given $n \times n$ matrices A and B , compute matrix $C = AB$

$$C[i,j] = \sum_{0 \leq k < n} A[i,k]B[k,j]$$

```

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

```

“Natural” $O(n^3)$ steps
implementation of
definition

Note: Much better implementations by blocking; “Strassen” and related algorithms are asymptotically better

Matrix-matrix multiplication: Given $n \times n$ matrices A and B , compute matrix $C = AB$

$$C[i,j] = \sum_{0 \leq k < n} A[i,k]B[k,j]$$

Reminder: MM is associative/distributive, but **not commutative**

Observation:

Iterations in i and j loops are fully independent (“data parallel”), iterations in k loop should be in increasing order (unless commutativity of $+$ is exploited).

Parallelization trivially possible:

Processor (i,j) , $0 \leq i < n$, $0 \leq j < n$:

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

$$W_{\text{par}}(p,n) = O(n^2 n) = O(n^3)$$

$$T_{\text{par}}(p,n) = O(n^3/p+n)$$

Same as PRAM parallelization
from first lecture

Observation:

Iterations in i and j loops are fully independent (“data parallel”), iterations in k loop should be in increasing order (unless commutativity of + is exploited).

The three loops can be interchanged freely

There are $6 = 3! = 3 \cdot 2 \cdot 1$ possible variations of this matrix-matrix implementation

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

Variant 1: ijk

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

Variant 2: ikj
(initialize C[i][j]
elsewhere)

Etc. ...

Observation:

Iterations in i and j loops are fully independent (“data parallel”), iterations in k loop should be in increasing order (unless commutativity of + is exploited).

The three loops can be interchanged freely

There are $6 = 3! = 3 \cdot 2 \cdot 1$ possible variations of this matrix-matrix implementation

Does it matter?

Try out all six variants, measure, $n=1000$

Side remark on dense matrix algorithm complexities:

Since the input size is $m=n^2$, the complexity of this matrix-multiplication algorithm as a function of input size is only $O(m\sqrt{m})$

Architecture: Intel i7-2600 @ 3.40 GHz, 8MByte cache.
 Mean of 30 repetitions (measured with clock(), not OpenMP);
 Optimization flags `-O3 -funroll-loops` (not always good)

	int		double	
	Non-opt	Opt	Non-opt	Opt
ijk	6.96 sec	6.38 sec	8.62 sec	6.82 sec

Optimization does not make a lot of difference; int/double roughly similar. Is this performance reasonable?

Are these the right and final conclusions?

$n=1000$ (so $n^3=1000,000,000 \approx \text{GOPS}$)

Check if the number of operations/time matches processor specification

Architecture: Intel i7-2600 @ 3.40 GHz, 8MByte cache.
 Mean of 30 repetitions (measured with clock(), not OpenMP).
 Optimization flags `-O3 -funroll-loops`

	int		double	
	Non-opt	Opt	Non-opt	Opt
ijk	6.96 sec	6.38 sec	8.62 sec	6.82 sec
ikj	4.51	0.32	4.50	0.81
jik	6.88	6.33	7.00	6.82
jki	11.81	13.76	13.90	15.11
kij	4.53	0.34	4.72	0.89
kji	11.87	13.80	13.98	15.08

$n=1000$ (so $n^3=1000,000,000 \approx \text{GOPS}$)

Architecture: AMD Opteron 6168 (**saturn**), 512KByte L2, 6MByte L3 cache.

Mean of 30 repetitions (measured with clock(), not OpenMP)

Optimization flags `-O3 -funroll-loops`

	int		double	
	Non-opt	Opt	Non-opt	Opt
ijk	21.84	17.18	26.90	18.31
ikj	15.11	1.09	14.89	2.26
jik	21.59	17.13	24.16	18.17
jki	42.93	46.62	52.71	49.96
kij	15.56	1.12	15.72	2.71
kji	43.09	46.99	50.50	50.12

$n=1000$ (so $n^3=1000,000,000 \approx \text{GOPS}$)

Architecture: Intel Xeon E7-8850 @ 2.00GHz, 24MByte cache
 Mean of 30 repetitions (measured with clock(), not OpenMP)
 Optimization flags `-O3 -funroll-loops`

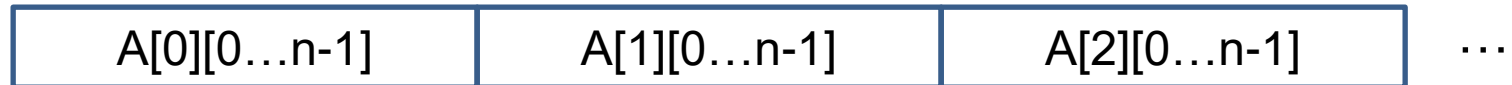
	int		double	
	Non-opt	Opt	Non-opt	Opt
ijk	12.45	10.44	13.07	10.93
ikj	9.88	0.60	9.95	1.19
jik	12.10	10.40	12.52	10.84
jki	18.96	22.40	20.21	23.78
kij	9.89	0.62	9.95	1.24
kji	18.94	22.43	20.18	23.73

$n=1000$ (so $n^3=1000,000,000 \approx \text{GOPS}$)

Exercise: Try this at home on your computer

Recall:

Matrices in C stored row wise: row major



A read into cache in blocks of 8-16 elements. Scanning A in row order (second index) reduces the number of main memory accesses by this factor. If row larger than cache size, next row forces eviction

Worst: innermost loop on i

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

Variant 4: jki

Variant 6: kji

Inner loop:

- $B[k][j]$ independent of i , put in register (compiler does this)
- Each iteration 1 load, 1 store
- **Each load cache miss, each store cache miss**

Assuming cache can store at most 2 matrix rows

Medium: innermost loop on k

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

Variant 1: ijk
Variant 3: jik

Inner loop:

- Each iteration 2 loads, 1 store
- $C[i][j]$ independent of k , update in register (compiler does this)
- Each load of $B[k][j]$ cache miss, A accessed in row order

Best: innermost loop on j

```

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

```

Variant 2: ikj

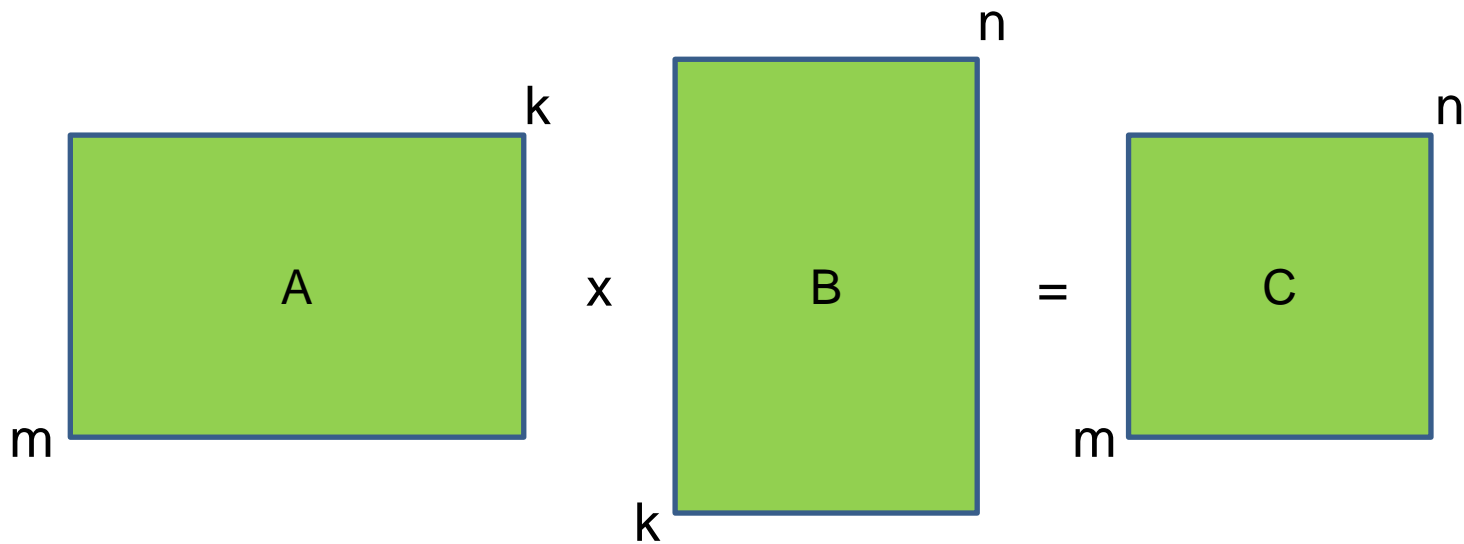
Variant 5: kij

Inner loop:

- $A[i][k]$ independent of j , put in register (compiler)
- Each iteration 1 load, 1 store
- **Both C and B accessed in row order**, miss rate as given by line size

These variants exploit spatial and temporal locality well. Much, much more can be done (well-known, but different lectures, try MKL)

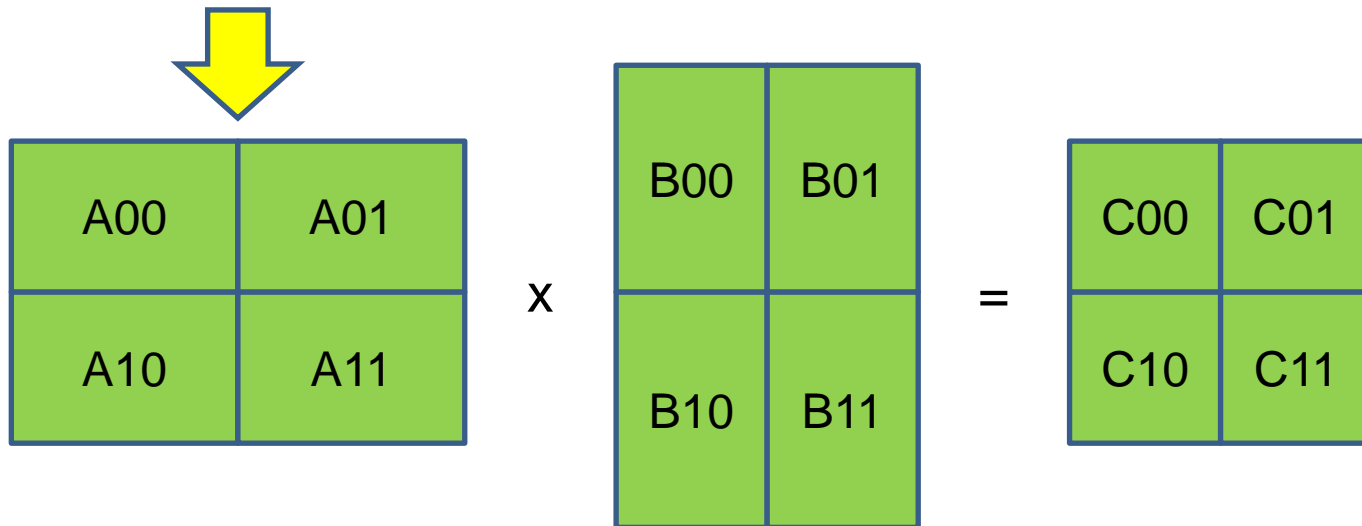
A different way of doing matrix-matrix multiplication



Standard, 3-loop implementation: $O(n^3)$ operations

... by (recursive) matrix-matrix multiplication of submatrices

Split matrices, roughly half in each dimension



$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}$$

$$C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11}$$

$$C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10}$$

$$C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11}$$

} 8 matrix multiplications, 4 matrix additions

Possibility for parallelization:

All 8 matrix multiplications can be done independently, all 4 matrix additions can be done independently

Solve recursively



$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}$$

$$C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11}$$

$$C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10}$$

$$C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11}$$

} 8 matrix multiplications, 4 matrix additions

Work and recursion depth for the MM algorithm is given by

- $W(n) = 8W(n/2) + \Theta(n^2)$
- $T(n) = T(n/2) + O(\log n)$ Last term: Depth of matrix addition

with solutions

- $W(n) = \Theta(n^3)$
- $T(n) = \Theta(\log^2 n)$

By Master Theorem

- $a=8, b=2, d=2, e=0$ (Case 3)
- $a=1, b=2, d=0, e=1$ (Case 2)

$W(n)$ and $T(n)$ is the number of nodes in an MM DAG and the length of a longest path, respectively. Keep in mind for the parallelization (later)

Note:

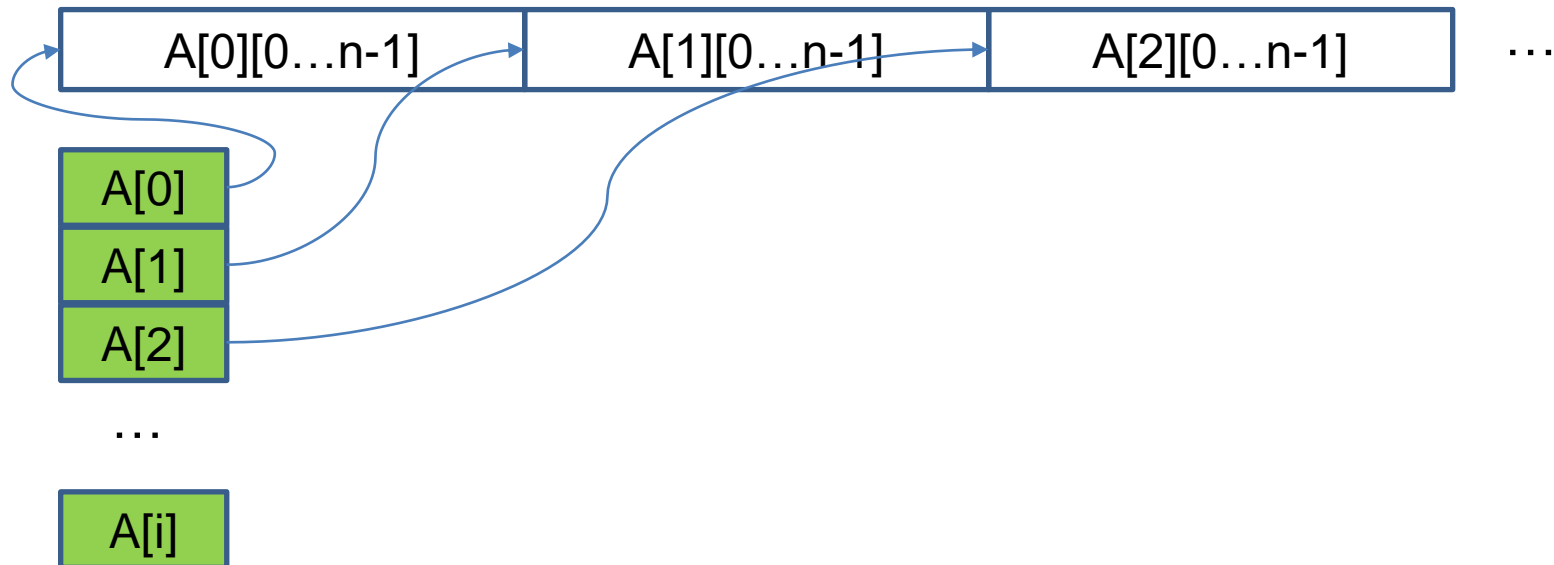
Strassen observed that matrix-matrix multiplication can be done with only 7 sub-matrix products (and 18 sums), leading to an algorithm running in $O(n^{2.81})$ operations.

Volker Strassen: Gaussian Elimination is not Optimal. Numerische Mathematik, 14(3):354-356, 1969

Major computer science result (with several follow-up improvements). These are the theoretically best known algorithms for matrix-matrix multiplication

Implementation in C

In C, 2-dimensional matrices are represented by arrays of pointers



We represent submatrices of A by start and end in each dimension, $A[m_0\dots m_1][n_0\dots n_1]$ to avoid copying into new matrices

The for the result matrix C, temporary matrices c0, c1 are needed, these have size $(m1-m0) \times (n1-n0)$.

For the recursive calls, submatrices of these matrices are computed, for this, offsets mo and no are needed.

For the matrix summation, the offsets from the parent recursive call are needed for the results to go to the right indices.

The recursion is stopped when a matrix is small enough ($m1-m0 < \text{CUTOFF}$), CUTOFF determined experimentally of by a good enough performance model (difficult)

```
void MM(double A[][] , double B[][] ,
        int m0, int m1, int k0, int k1, int n0, int n1,
        double C[][] , int m0, int no) {
    if (/*n0,n1,k0,k1,m0,m1 too small: CUTOFF*/) {
        BaseMM(A,B,m0,m1,k0,k1,n0,n1,C,m0,no);
    } else {
        double c0[m1-m0][n1-n0]; // allocate properly
        double c1[m1-m0][n1-n0]; // and free at the end
        MM(A,B,m0,(m0+m1)/2,k0,(k0+k1)/2,
            n0,(n0+n1)/2,c0,0,0);
        MM(A,B,m0,(m0+m1)/2,k0,(k0+k1)/2,
            n0+n1)/2,n1,c0,0,(n1-n0)/2);
        MM(A,B,m0,(m0+m1)/2,(k0+k1)/2,k1,
            n0,(n0+n1)/2,c1,0,0);
        // remaining 5 MM calls
        ...
        // all 8 MM done, now add
```

```

// all 8 MM done, now add...
for (i=m0; i<m1; i++) { // slowly, sequentially
    for (j=n0; j<n1; j++) {
        C[mo+i][no+j] =
            c0[i-m0][j-n0]+c1[i-m0][j-n0];
    }
}
// free intermediate matrices c0 and c1
}
}

```

Call with

```
MM(A, B, 0, n, 0, k, 0, m, C, 0, 0);
```

All calls MM calls independent

Note:

The recursive matrix-matrix multiplication formulation exhibits good cache behavior, can be made cache oblivious: Good cache behavior independent of actual cache size.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, Sridhar Ramachandran: Cache-Oblivious Algorithms. ACM Trans. Algorithms 8(1): 4:1-4:22 (2012)

Matteo Frigo, Volker Strumpfen: The Cache Complexity of Multithreaded Cache Oblivious Algorithms. Theory Comput. Syst. 45(2): 203-233 (2009)

Architecture: Intel(R) Xeon(R) CPU E3-1225 v5 @ 3.30GHz, 32MB cache.

Mean of 30 repetitions (measured with clock(), not OpenMP)

Optimization flags `-O3 -funroll-loops`

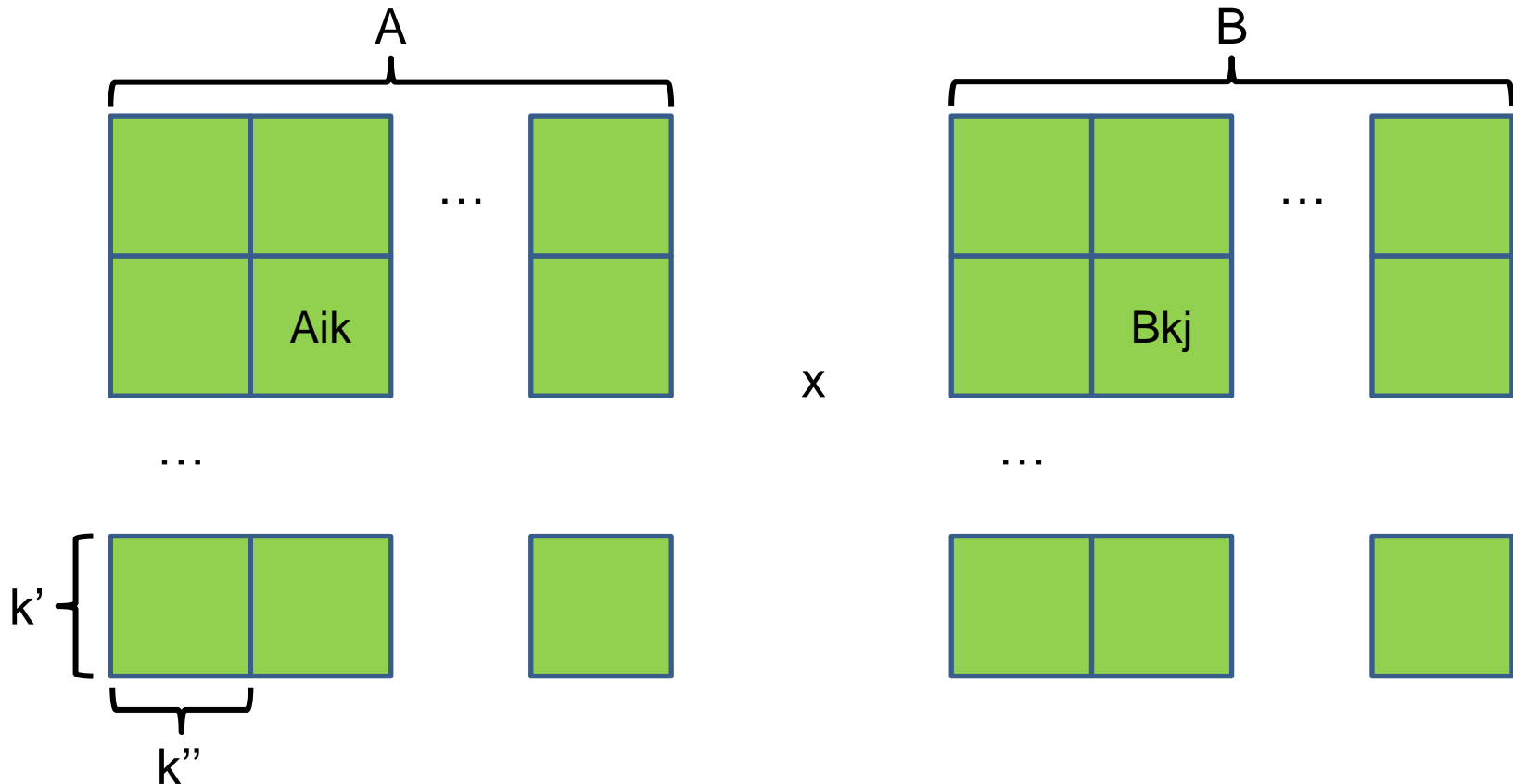
	int		double	
	Non-opt	Opt	Non-opt	Opt
ijk	5.02	0.86	5.85	1.57
ikj	3.64	0.23	3.60	0.35
jik	4.87	0.84	5.26	1.43
jki	7.92	8.01	8.85	7.43
kij	3.57	0.22	3.59	0.39
kji	7.89	8.00	8.10	7.31
recursive	5.72	0.83	5.98	0.87

CUTOFF
= 10

$n=1000$ (so $n^3=1000,000,000 \approx \text{GOPS}$)

Aside: Cache-aware matrix-matrix multiplication

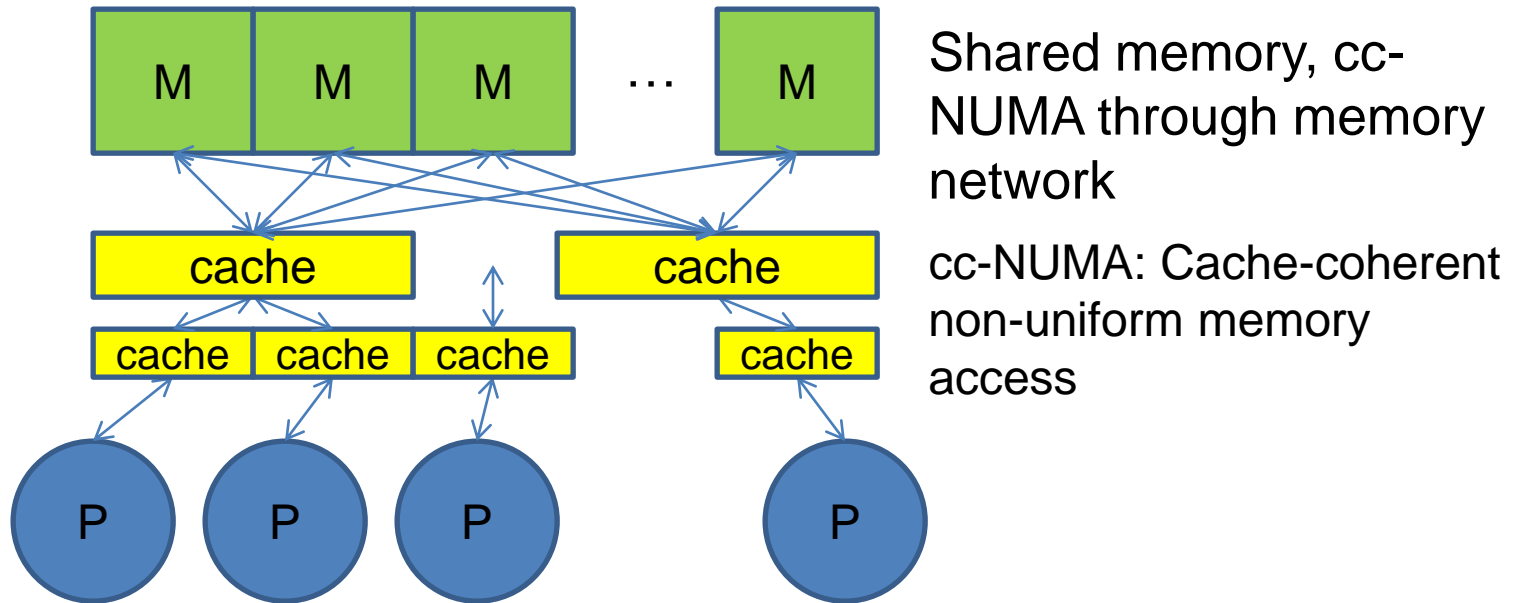
Multiply as smaller $k' \times k''$ matrices, $C = A \times B =$



k' , k'' chosen such that the small matrices fit in cache. Best choice dependent on the sizes of the caches in the cache hierarchy

Implementation takes 6 nested loops

Multiprocessor/multi-core caches

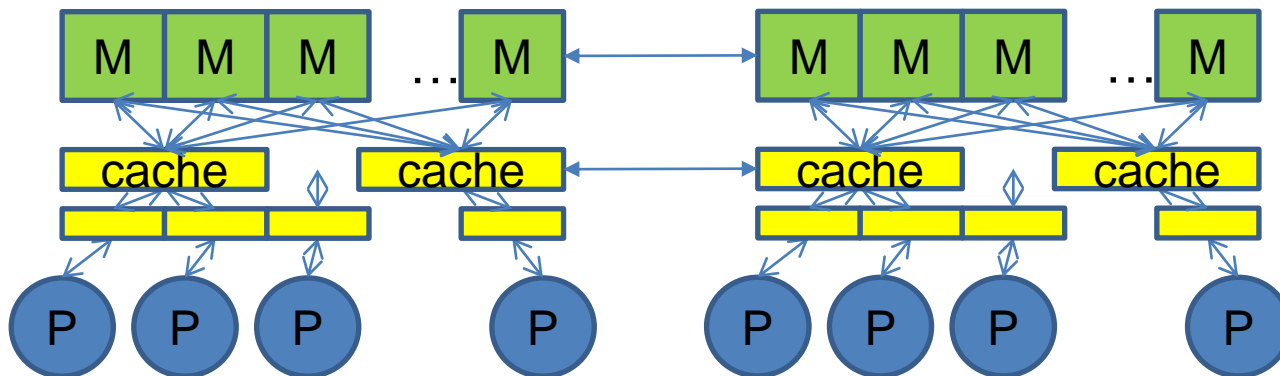


Typically, several cores shares caches at some levels

More terminology

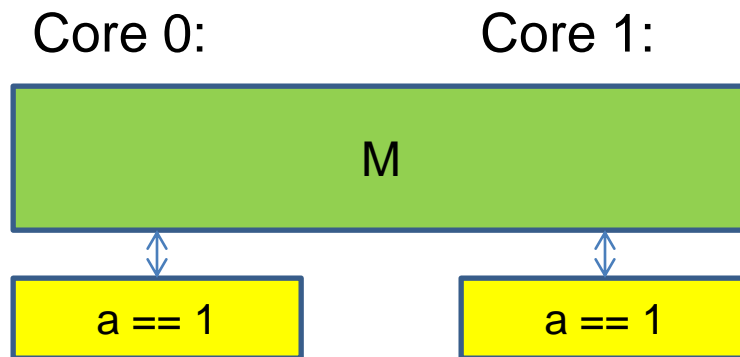
Multi-core processor: Processor with several cores/processing elements/CPU's (**physically on the same chip**). Always shared-memory (caches) of some sort

Multi-processor system: System with several processors, not on the same chip; the processors can be multi-core processors. Multi-processor may have shared-memory, and even shared caches



Cache coherence

Processor/core 0 and 1 with private caches, both have read location `a` into cache. Processor 0 writes to `a`, what happens?



`a = 7;`

`b = a; // ??`

Read by core 1 occurs
“after” write by core 0. If `b`
is still 1, never 7, cache
system is not coherent

Let the order of memory accesses to a specific location a be given by the program order

Definition: Cache system is coherent if

1. If processor P writes to a at time t_1 and reads a at $t_2 > t_1$, and there are no other writes (by P or other) to a between t_1 and t_2 , then P reads the value written at t_1
2. If P_1 writes to a at t_1 and another P_2 reads a at $t_2 > t_1$ and no other P writes to a between t_1 and t_2 , then P_2 reads the value written by P_1 at t_1
3. If P_1 and P_2 writes to a at the same time, then either the value of P_1 or the value of P_2 is stored at a

Ad 1. Program order is preserved for each processor for locations that are not written by other processors

Let the order of memory accesses to a specific **location a** be given by the program order

Definition: Cache system is coherent if

1. If processor P writes to **a** at time t_1 and reads **a** at $t_2 > t_1$, and there are no other writes (by P or other) to **a** between t_1 and t_2 , then P reads the value written at t_1
2. If P1 writes to **a** at t_1 and another P2 reads **a** at $t_2 > t_1$ and no other P writes to **a** between t_1 and t_2 , then P2 reads the value written by P1 at t_1
3. If P1 and P2 writes to **a** at the same time, then either the value of P1 or the value of P2 is stored at **a**

Ad 2. Here, t_1 and t_2 have to be “sufficiently” separated in time. Updates by P1 must eventually become visible to the other processors

There is no absolute time, and nothing immediate

Let the order of memory accesses to a specific **location a** be given by the program order

Definition: Cache system is coherent if

1. If processor P writes to **a** at time t_1 and reads **a** at $t_2 > t_1$, and there are no other writes (by P or other) to **a** between t_1 and t_2 , then P reads the value written at t_1
2. If P1 writes to **a** at t_1 and another P2 reads **a** at $t_2 > t_1$ and no other P writes to **a** between t_1 and t_2 , then P2 reads the value written by P1 at t_1
3. If P1 and P2 writes to **a** at the same time, then either the value of P1 or the value of P2 is stored at **a**

Ad 3. Writes are required to “**serialize**”. Either of the values simultaneously written will be stored. “Same time” means “sufficiently close” in time.

cc-NUMA systems (most multi-core and SMP nodes): Cache coherent, non-uniform memory access

Cache coherence maintained by hardware at the **cache line level**.
Standard approaches and protocols:

- Update based
- Invalidation based
- Snooping/bus based
- Directory based

Protocols in hardware like MESI, MOESI, ...

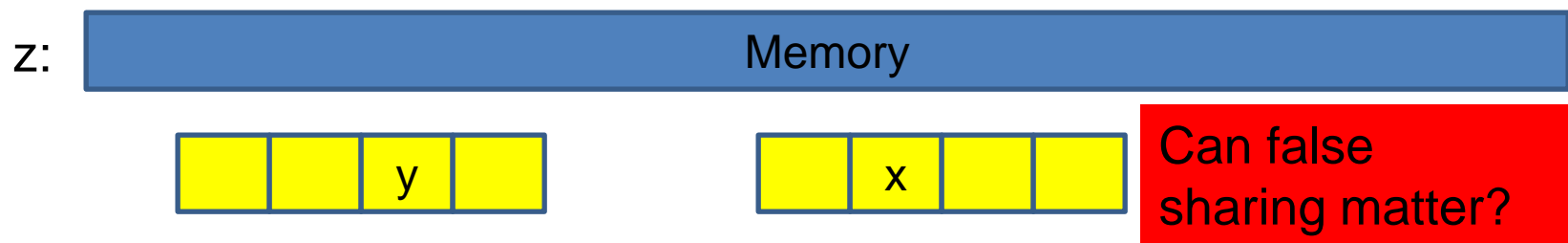
Expensive: bus/network traffic, protocol overhead, hardware (“transistors”, “power”); can affect performance negatively

Standard textbook:

Bryant, O'Hallaron: Computer Systems. Prentice-Hall, 2011
Hennessy, Patterson: Computer Architecture – A Quantitative Approach, Morgan-Kaufmann, 2011 (and later/earlier)

Sharing/false sharing

Cache coherence is maintained at the cache line level. Example: Core 0 updates y , core 1 updates x which happen to be on the same cache line (e.g., $\&x == \&z[1]$, $\&y == \&z[2]$ for some array z)



for ($i=0$; $i<n$; $i++$) $y += i-1$; for ($i=0$; $i<n$; $i++$) $x += 2*i$;

Core 0:

Core 1:

Although x and y are different memory locations, each update will cause cache coherency activity because cache coherency is at the cache line level. Variables x and y said to be falsely shared

Example: Matrix $x[m][n]$, compute all row sums $x[j][0] = \sum x[j][i]$

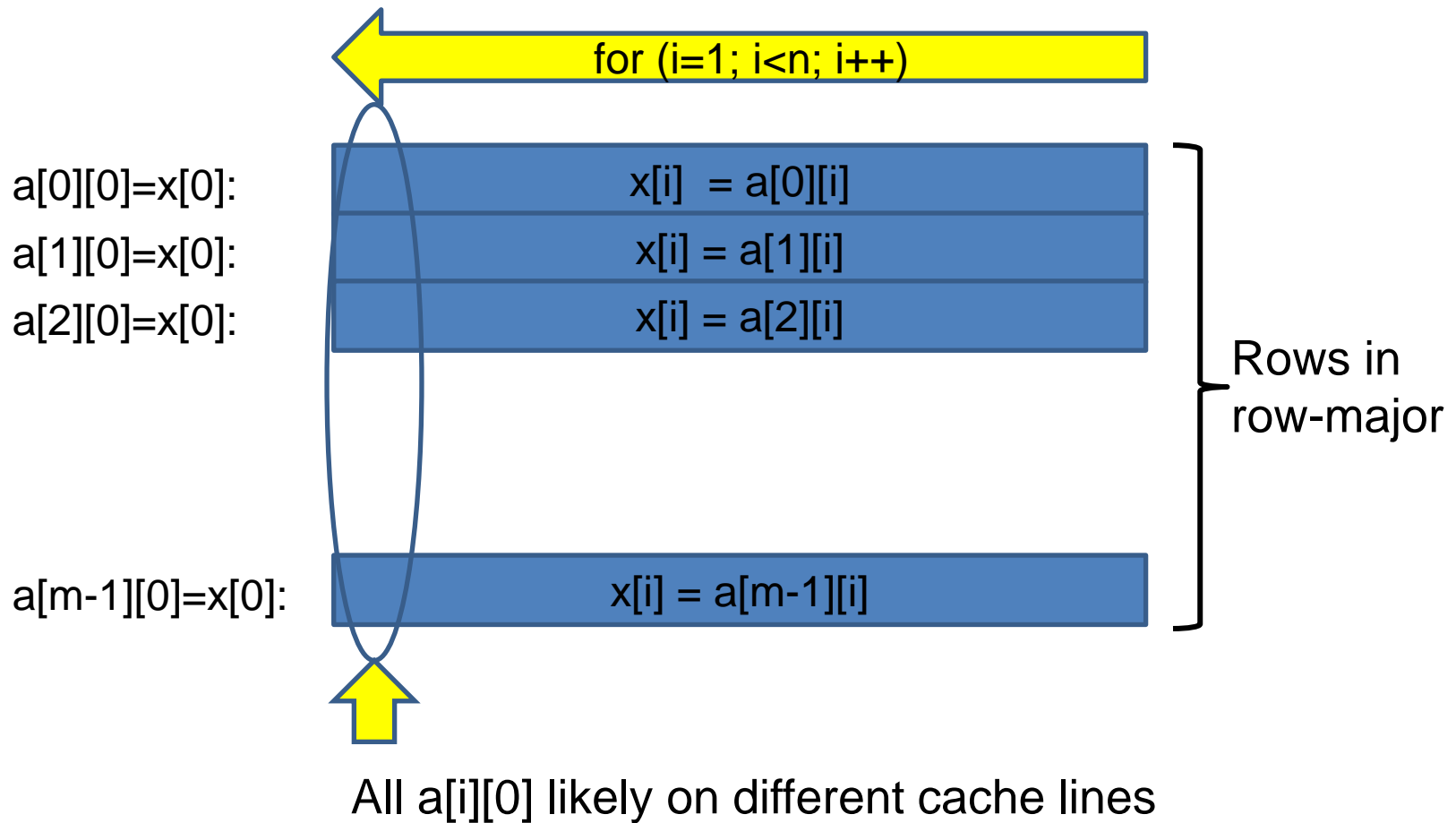
Matrix stored in row-major (as in C):

```
// a[m][n] two dimensional matrix
int *a = (int*)malloc(m*n*sizeof(int));
// row order access
#pragma omp parallel for
for (j=0; j<m; j++) {
    int *x = a+j*n;
    int i;
    for (i=1; i<n; i++) {
        x[0] = x[0]+x[i];
    }
}
```

OpenMP parallelization construct

Beware: must be local variables

Rows j and $j+1$ not spatially close, likely not in same cache line



Example: Matrix $x[m][n]$, compute all row sums $x[j][0] = \sum x[j][i]$

Matrix stored in column-order (e.g., FORTRAN)

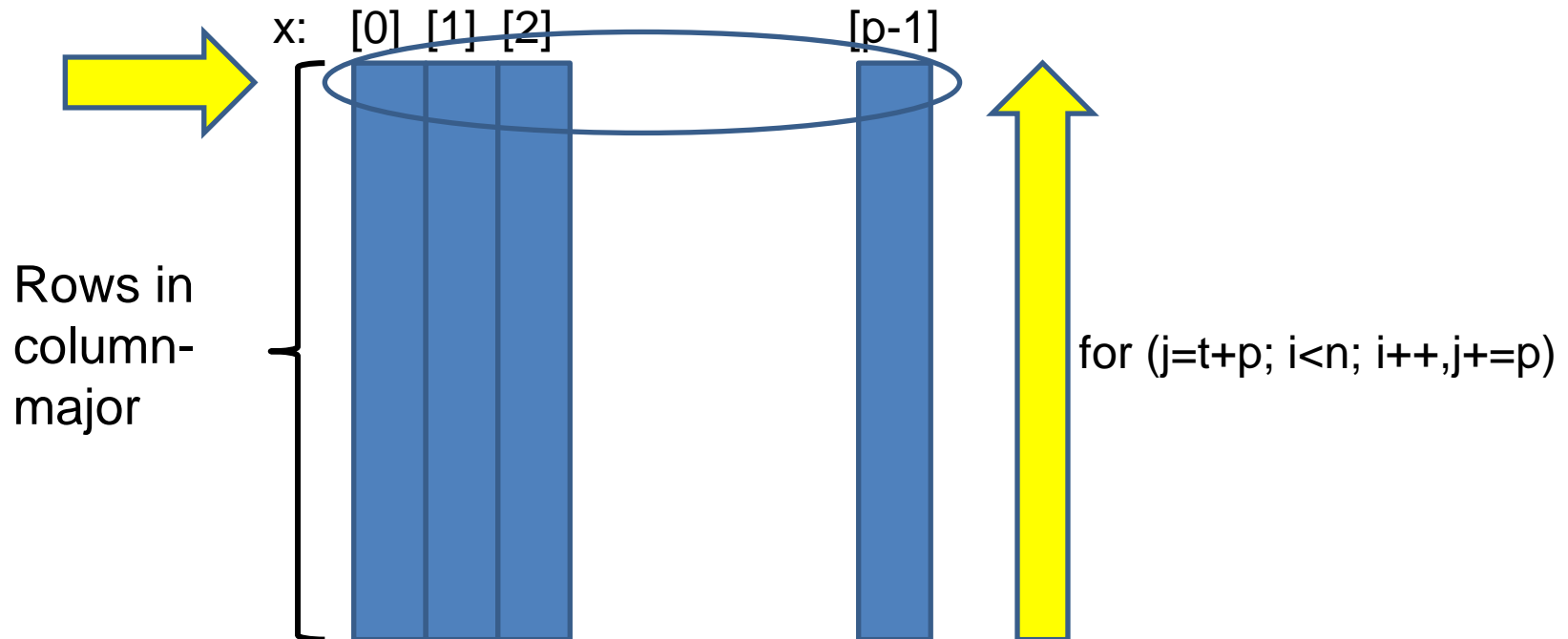
```
// x[m][n] two dimensional matrix
int *a = (int*)malloc(n*m*sizeof(int));
// row order access
#pragma omp parallel
{
    int t = omp_get_thread_num(); // thread id
    int i, j;
    int *x = a;
    for (j=t+p, i=1; i<n; i++,j+=p) {
        x[t] = x[t]+x[j];
    }
}
```

Assumption: p=m

All local

$x[0], x[1], x[2], \dots, x[p-1]$ may lie on same cache line, updated pn times

Same cache line



Example: Matrix $x[m][n]$, compute all row sums $x[j][0] = \sum x[j][i]$

Matrix stored in column-order (e.g., FORTRAN): control version

```
// x[m][n] two dimensional matrix
int *a = (int*)malloc(n*m*sizeof(int));
// row order access
#pragma omp parallel
{
    int t = omp_get_thread_num(); // thread id
    int i, j;
    int *x = a;
    register int sum = 0;
    for (j=t*p, i=1; i<n; i++,j+=p) {
        sum = sum+x[j];
    }
    x[t] = sum;
}
```

But has another,
cache related
problem (which?)

Should have no false sharing

Architecture:

Intel i7-2600 @ 3.40 GHz, 8MByte cache, 4 cores

Mean of 100 repetitions (measured with OpenMP).

```
gcc -O3 -fopenmp
```

m	n	row	column	register
2	10,000	2.41	6.59	2.43
4		2.86	8.86	3.57
8		4.51	18.91	4.49
2	1,000,000	947.66	2347.51	564.56
4		955.55	2822.79	1285.36
8		1838.17	7653.76	2631.49

Architecture:

AMD Opteron 6168, 512KByte L2, 6MByte L3 cache, 48 cores

Mean of 100 repetitions (measured with OpenMP)

```
gcc -O3 -fopenmp
```

m	n	row	column	register
2	1,000,000	1334.95	6867.88	3914.69
4		1316.17	18682.78	8225.76
8		3998.82	52483.61	32205.00
12		2600.00	84042.84	49556.98
24		11646.59	240534.15	211900.23
48		9529.59	241729.00	236951.06

NB: Measurements not very stable, but same behavior (factors: clock resolution, disturbance)

Architecture:

Intel Xeon E7-8850 @ 2.00GHz, 24MByte cache, 80 cores

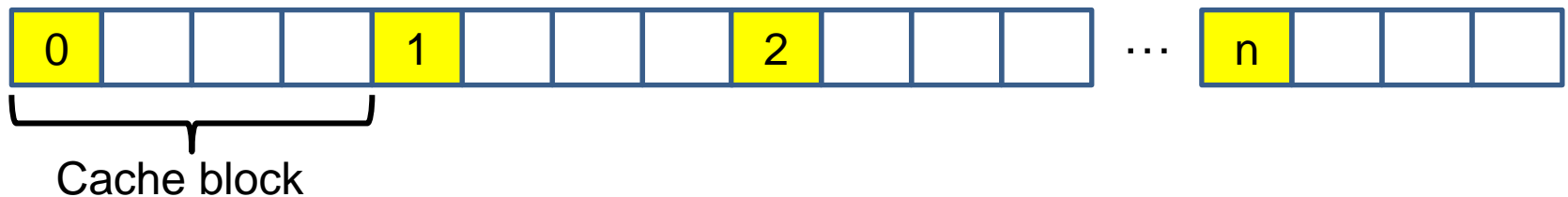
Mean of 100 repetitions (measured with OpenMP)

`gcc -O3 -fopenmp (icc worse?)`

m	n	row	column	register
2	1,000,000	1330.06	2090.08	1278.82
5		1360.92	3999.19	3009.37
10		1374.28	42667.57	40673.03
20		3774.24	83096.40	84314.09
40		7548.22	99135.79	95015.96
60		8735.67	98547.61	97293.80
80		15922.08	135945.35	181497.75
160		15958.70	215910.70	154105.94

Avoiding false sharing:

- Make sure simple, shared variables updated by different threads appear in different cache lines (blocks): Pad data structures, e.g., padded array



- Not always possible, (too) wasteful in memory space
- Use simple local variables (compiler may put these on different cache lines)

Not all systems are cache-coherent:



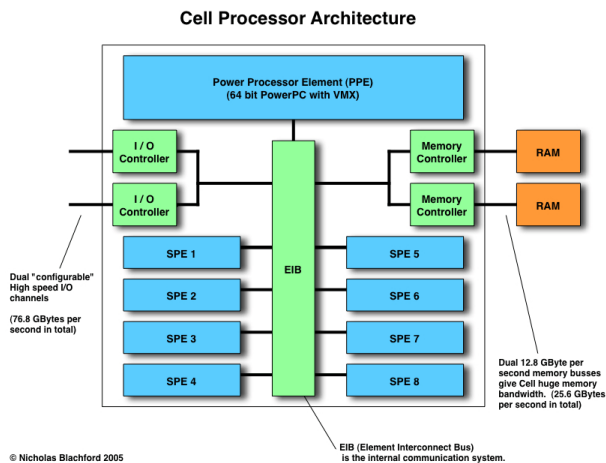
(NEC) Vector computers:
8-16 processors per node (sequential
processor+vector processor)

Caches, but **no coherence**

IBM/Sony/Toshiba Cell BE: Sequential
PowerPC processor, 8 SPU's

No caches: scratchpad memory

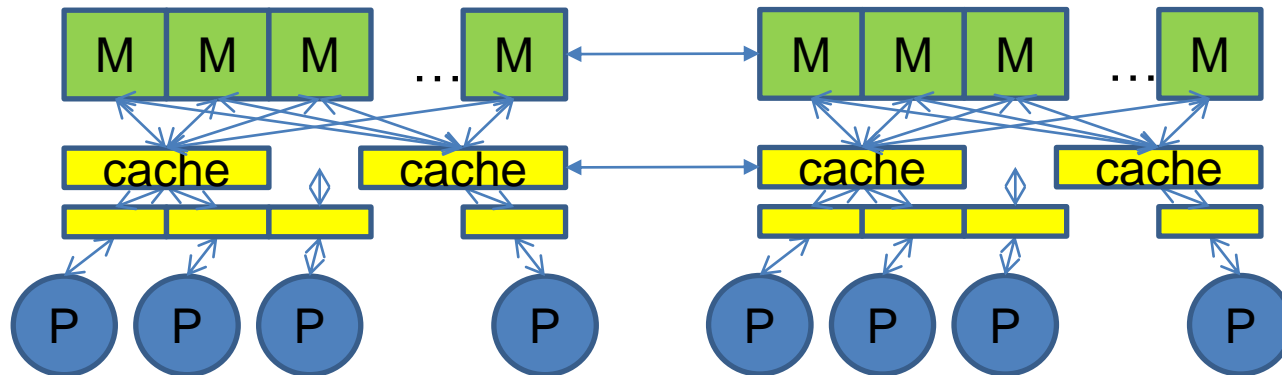
IBM BlueGene/L: 2-4 cores, non-
coherent



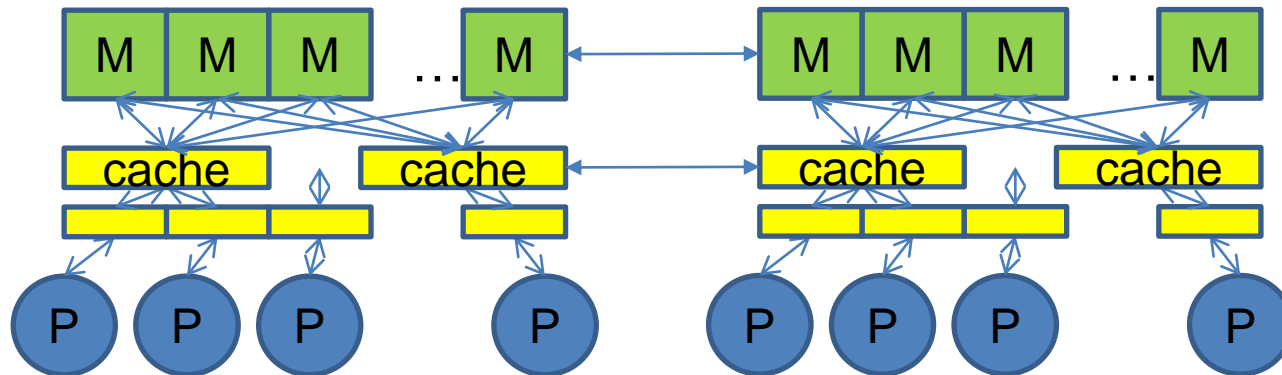
On multi-core cache debate (“to be coherent or not”), see, e.g.,

Milo M. K. Martin, Mark D. Hill, Daniel J. Sorin: Why on-chip cache coherence is here to stay. *Commun. ACM* 55(7): 78-89 (2012)

Memory (“von Neumann”) bottleneck, NUMA



- Connection to memory via memory-controllers
- Memory in banks
- Memory banks have affinity to memory controllers
- Typically fewer memory controllers than cores (1-4 per CPU)
- Cores share memory bandwidth



Memory access times **highly non-uniform** (NUMA). Accessing the memory of a memory controller closest to a core faster than accessing memory on a controller of a different CPU

NUMA performance matters

To exploit memory system well (avoid being penalized), applications can aim to allocate memory for the cores “close” to the core that will use that memory the most

“First-touch” OS support: The first core (thread) to touch a (virtual) memory page will cause the page to be allocated in memory close to that core

Does this matter?

Worst-case program (with OpenMP, see later):

- Each thread (running on a fixed core) allocates medium large array (n=1.000.000 elements), touches its array
- In a number of repetitions, goes through all elements, perform some computation
- Uses either own array (Local), or array allocated by core “farthest away” (Remote)


```

start = omp_get_wtime();
#pragma omp parallel
{
    double *a = (double*)malloc(n*sizeof(double));
    double c;
    int i,r;

    c = 0.0;
    for (i=0; i<n; i++) a[i] = c; // first touch
    aa[omp_get_thread_num()] = a;
#pragma omp barrier
    double *b = aa[(omp_get_thread_num()+20)%t];

    for (r=0; r<REPEAT; r++) { ... / some computation
        for (i=0; i<n; i++) { c += b[i]; c /= (i+1);}
        for (i=0; i<n; i++) b[i] = c;
    }
    free(a);
}
stop = omp_get_wtime();

```

← OpenMP later

Distance (0: best, 20: more or less worst)

↓

80-cores, 8 CPU (“socket”) system, Intel(R) Xeon(R) E7-8850 @ 2.00GHz, with

```
OMP_NUM_THREADS=80  
OMP_PROC_BIND=close  
OMP_PLACES=cores
```

Outcome:

Max threads 80

Local: n=1000000 time (micros) 4283003.39

Remote: n=1000000 time (micros) 6008407.72



80-cores, 8 CPU (“socket”) system, Intel(R) Xeon(R) E7-8850 @ 2.00GHz

But there are other things going on
OMP_NUM_THREADS=1

Outcome:

Max threads 1

Local: n=1000000 time 578939.91

Remote: n=1000000 time 573594.96

Is this the price for cache coherence? Or the effect of limited total bandwidth?

Factor 8 faster!

Approaches to alleviating memory bottleneck: Latency hiding

- Prefetching: Start loading operands well before use (HW or SW)
- Hardware/software multi-threading: When a thread (“virtual processor”) issues a load, switch to another thread, come back when data have arrived

Multi-threading requires explicitly parallel programs (EPIC)

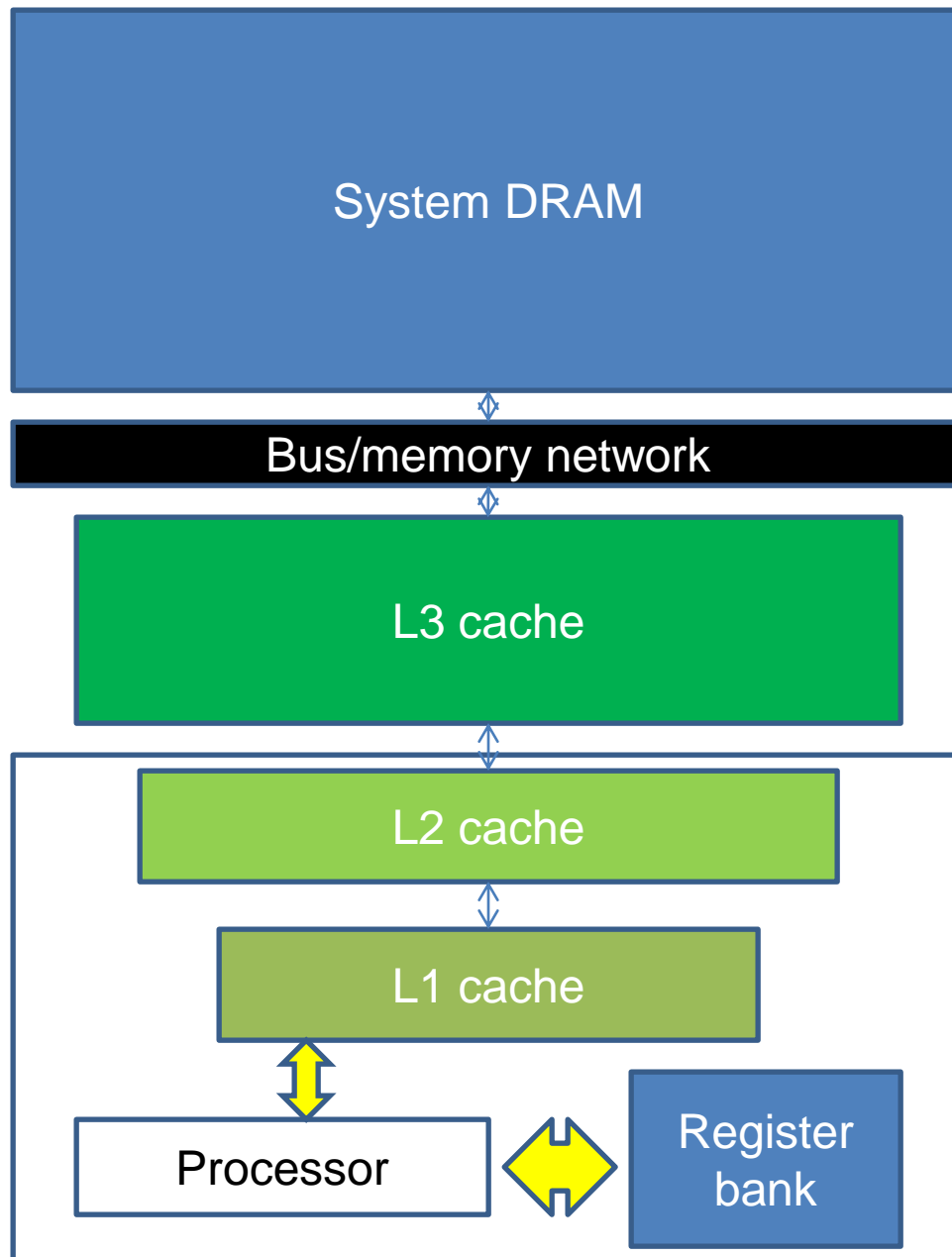
Both prefetching and multi-threading are latency hiding techniques. Memory bandwidth still required for the number of outstanding memory requests.

Disproof (II): Is super-linear speed-up possible?

Simulation argument shows that linear/perfect absolute speed-up is best possible!

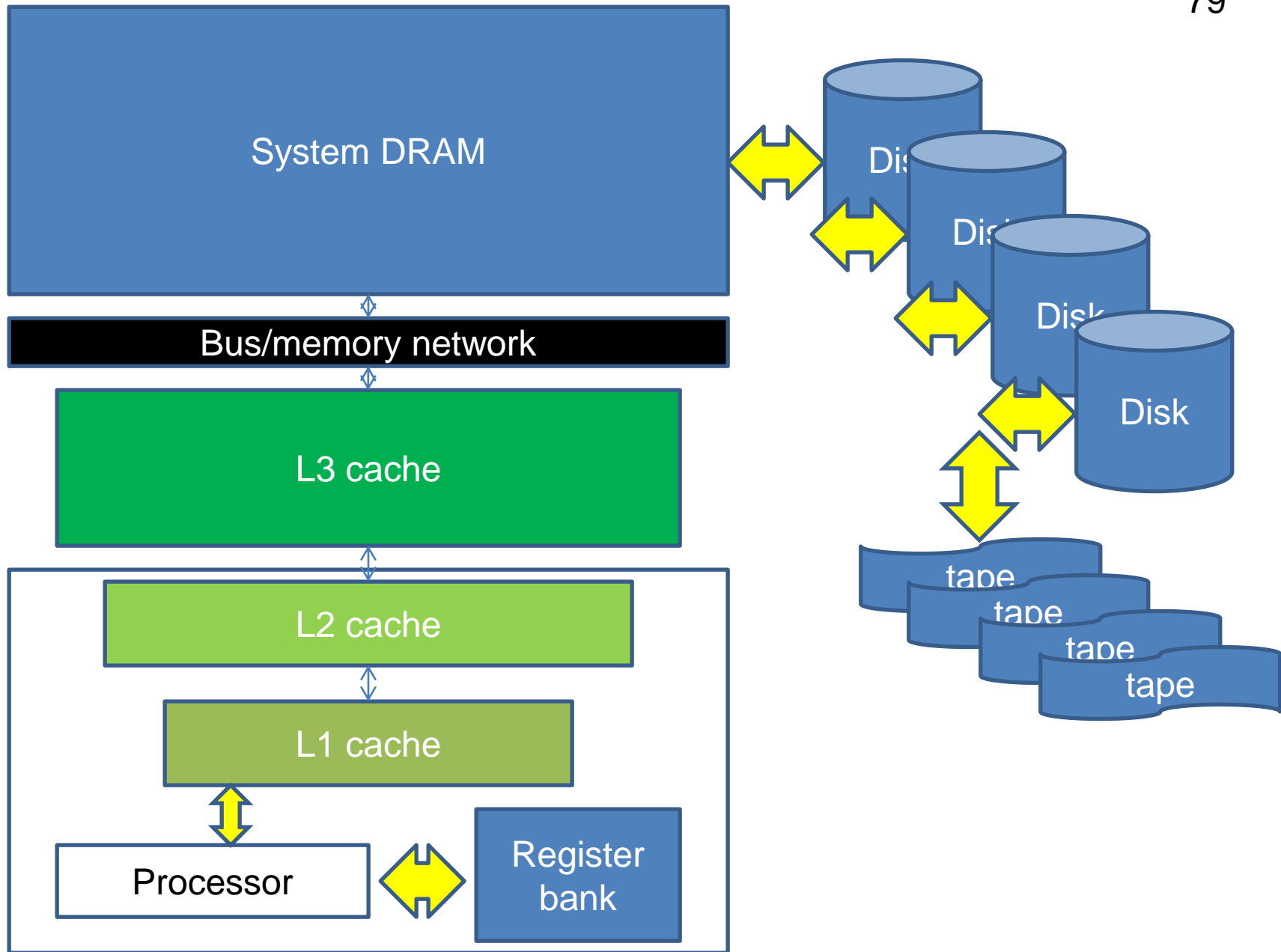
But: Argument assumes that sequential and parallel processors are of the same sort, in particular same memory behavior

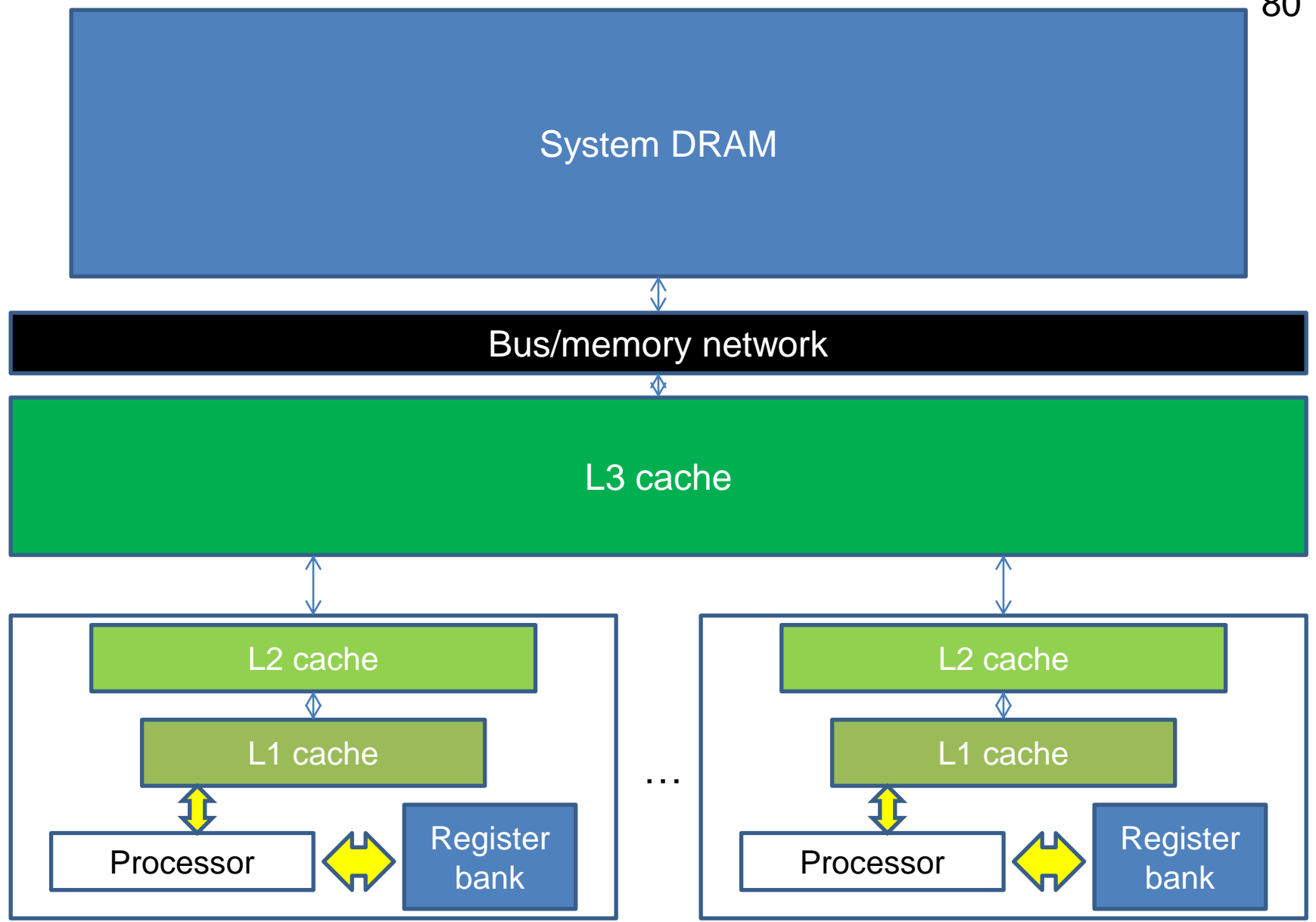
This is not true for (real) systems with a deep memory hierarchy:



The memory hierarchy

3. Banked memories
2. Memory network
1. Several levels of caches (registers, L1, L2, ...)

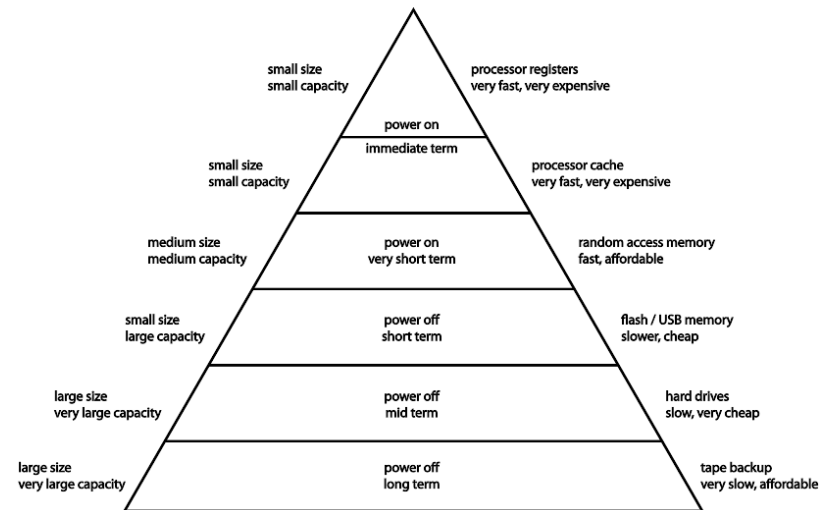




Typical latency values for memory hierarchy:

Registers:	0 cycles
L1 cache:	1 cycles
L2 cache	10 cycles
L3 cache	30 cycles
Main memory:	100 cycles
Disk:	100,000 cycles
Tape:	10,000,000 cycles

Computer Memory Hierarchy



Bryant, O'Halloran: Computer Systems, Prentice-Hall, 2011 (and later)

Example:

- Single processor implementation on huge input of size n that needs to use full memory hierarchy

vs.

- Parallel algorithm on distributed data of size n/p where each processor may work on data in main memory, or even cache (p is large, too)

- Let $\text{ref}(n)$ be the number of memory references
- $\text{Mseq}(n)$ average time per reference for the single-processor system (**deep hierarchy**)
- $\text{Mpar}(p,n)$ average time per reference per processor for parallel system (**flat hierarchy**)

Then (assuming performance determined by cost of memory references) for perfectly parallelizable memory references

$$S_p(n) = \text{ref}(n) * \text{Mseq}(n) / ((\text{ref}(n)/p) * \text{Mpar}(p,n)) = p * \text{Mseq}(n) / \text{Mpar}(p,n)$$

Example: With $\text{Mseq} = 1000$, $\text{Mpar} = 100$ (independent of n and p), this gives **super-linear speed-up** (**misnomer**: linear, but greater than p)

$$S_p(n) = 10p$$

Application performance and memory system

Application is

- Memory bound, if the operations to be performed per data element can be performed faster than reading/writing the data element
- Compute bound, if the operations to be performed per data element take longer than reading/writing the data element

For memory bound applications, concrete speed-up is limited by the memory bandwidth: How much faster can p cores read/write data than only one core?

Examples:

Do not expect too large speed-up

- **Memory bound**: Merge, prefix-sums (only a few operations per Byte)
- **Less memory bound**: Matrix-matrix multiplication, $O(n^3)$ operations on $O(n^2)$ data

Quantitative formulation

Application is

- Memory bound, if the operations to be performed per data element can be performed faster than reading/writing the data element
- Compute bound, if the operations to be performed per data element take longer than reading/writing the data element

Given application (implementation) A, define the Operational Intensity O as the average number of operations per Byte read/written. Given multi-core system with performance P (GOPS, e.g.) and memory bandwidth B :

- A is memory bound, if $P/O > B$
- A is compute bound if $P/O \leq B$

Example:

Prefix-sums performs, say, one FLOP per word (8 Bytes) read and written (16 Bytes), $O = 1/16$ FLOP/Byte. Processor can do 2GFLOPS

Prefix-sums memory bound, if memory bandwidth less than 32 GByte/s.

This performance model/estimate is called Roofline

Samuel Williams, Andrew Waterman, David A. Patterson: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52(4): 65-76 (2009)

More Roofline in HPC lecture

Program order and memory consistency

```
if (id==0) {  
    a = 1;  
    a = 2;  
    b = 3;  
} else {  
    b = 7;  
    a = 1;  
    a = 2;  
}
```

Sequential code, expectation:

Memory read/write operations take effect (become visible) in the order specified by the (execution of the) program

Example property: $a=2$, and ($b=3$ if $id=0$, or $b=7$ if $id \neq 0$)

“Easy” to prove properties of programs: Invariants etc.

Parallel program order:

p processors execute program (SPMD or MIMD)

```
a = 1;  
a = 2;  
b = 3;
```

Some interleaving of the p programs is executed by the p processors.

```
b = 7;  
a = 1;  
a = 2;
```

(Natural) Expectation:

Memory read/write operations take effect in this order

Still “easy” to prove properties of programs

Parallel program order:

p processors execute program (SPMD or MIMD)

```

a = 1;
a = 2;
           b = 7;
           a = 1;
           a = 2;
b = 3;

```

Some interleaving of the p programs is executed.

(Natural) Expectation:

Memory read/write operations take effect in this order

Possible outcome: a=2, b=3: or a=2, b=7 **NOT:** a=1, b=3

Still “easy” to prove properties of programs

Sequential consistency: Outcome of parallel program is as if some interleaving of the instructions has been executed, with each memory operation taking effect immediately

Parallel program order:

p processors execute program (SPMD or MIMD)

```

a = 1;
a = 2;
           b = 7;
           a = 1;
           a = 2;
b = 3;
  
```

Some interleaving of the p programs is executed.

(Natural) Expectation:

Memory read/write operations take effect in this order

Possible outcome: a=2, b=3: or a=2, b=7 **NOT:** a=1, b=3

Still “easy” to prove properties of programs

Note: Hardware guarantees sequential consistency for sequential programs, reads and writes observe program order. Caches, write buffers, ... are logically transparent (not performance transparent)

Memory consistency

In what order do writes to different locations become visible in memory to other cores?

Core 0:

```
x = 0;  
// ... some code  
x = 1;  
if (y==0) {  
    // body  
}
```

Core 1:

```
y = 0;  
// ... some code  
y = 1;  
if (x==0) {  
    // body  
}
```

Assumption: x
not in cache of
core 1, y not in
cache of core 0

Can core 0 and core 1 both execute body of if-statement?

Core 0:

```
x = 0;
// ... some code
x = 1;
if (y==0) {
    // body
}
```

Core 1:

```
y = 0;
// ... some code
y = 1;
if (x==0) {
    // body
}
```

Can core 0 and core 1 both execute body of if-statement?

No: If core 0 in body, it has executed $x = 1$; and core 1 has executed $y = 0$; but not yet $y=1$; therefore core 1 cannot enter body because $x=1$

Note:

If $x=1$; $y=1$; appears at the same time, **no cores execute body**, so probably not a good lock-algorithm (see later)...

Core 0:

```
x = 0;
// ... some code
x = 1;
if (y==0) {
    // body
}
```

Core 1:

```
y = 0;
// ... some code
y = 1;
if (x==0) {
    // body
}
```

Can core 0 and core 1 both execute body of if-statement?

No: If core 0 in body, it has executed $x = 1$; and core 1 has executed $y = 0$; but not yet $y=1$; therefore core 1 cannot enter body because $x=1$

Argument
correct?

Only under certain consistency assumptions

Core 0:

```

x = 0;
// ... some code
x = 1;
if (y==0) {
    // body
}

```

Core 1:

```

y = 0;
// ... some code
y = 1;
if (x==0) {
    // body
}

```

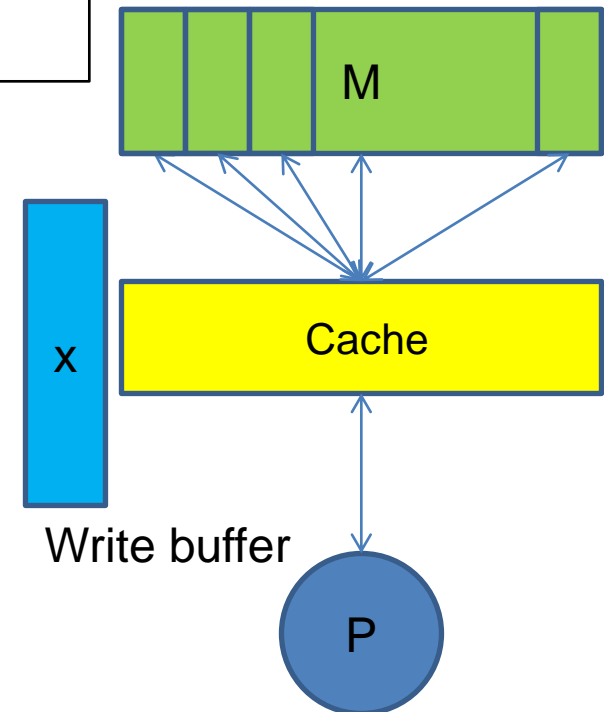
With write buffer:

Core 0: x=0; but x=1; **is delayed to memory**

Core 1: y=0; but y=1; **is delayed to memory**

Both core 0 and core 1 execute body

This execution is not an interleaving of the two program fragments



Sequential consistency: Memory operations of each processor are performed in program order; program result is some interleaving of the memory accesses of all processors

Leslie Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9): 690-691 (1979)

Sequential consistency is typically **not guaranteed** by modern multiprocessors (x86, POWER, SPARC):

- Caches: may delay writes
- Write buffers: may delay and/or reorder writes
- Memory network: may reorder writes
- Compiler: may reorder updates

Insisting on sequential consistency could sacrifice performance

Sequential consistency: Memory operations of each processor are performed in program order; program result some interleaving of the memory accesses of all processors

Programming model (memory model): Special synchronization/coordination constructs to enforce specific (set of) interleaving(s); “fence” operation to flush write buffers/etc.


Core 0:

```
x = 0;
fence;
// ... some code
x = 1;
fence;
if (y==0) {
    // body
}
```

Core 1:

```
y = 0;
fence;
// ... some code
y = 1;
fence;
if (x==0) {
    // body
}
```

Ready to
execute
“code” and
enter “body”



Relaxed consistency models pose weaker constraints on hardware.
More difficult to reason about correctness

Relaxed models may permit:

- Loads (reads) reordered after loads
- Loads reordered after stores (writes)
- Stores reordered after loads
- Stores reordered after stores

See AMP lecture

Special instructions: memory fences/memory barriers enforce pending memory operations to complete.

A fence (~~#~~barrier) is a local operation for the core (no to be confused with a barrier which involves all processors)

Relaxed consistency models pose weaker constraints on hardware.
More difficult to reason about correctness

Example (`flag = 0`):

```
while (!flag) { }  
a = otherval;
```

```
otherval = 42;  
flag = 1;
```

Intended (under sequential consistency) outcome: `a==42`

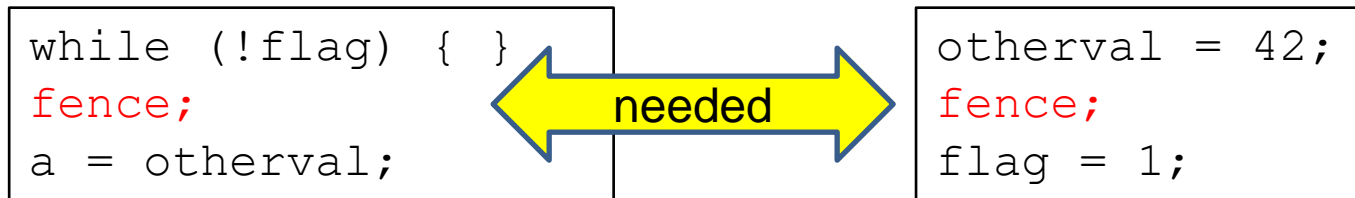
But:

Writes could be reordered. Any old value could be stored in `a`.
Compiler should not attempt to move assignment to `a` before loop

Beware: Compiler might remove while loop altogether. Declare `flag` as `volatile`

Relaxed consistency models pose weaker constraints on hardware.
More difficult to reason about correctness

Example (`flag = 0`):



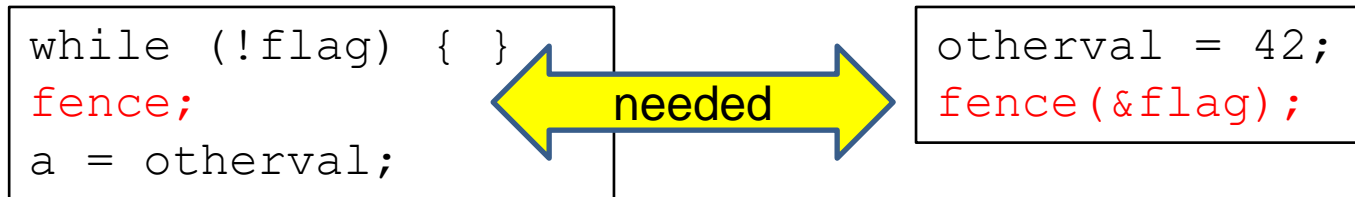
...depending on exact memory model, either might be
superfluous

More on consistency models (huge literature):

Sarita V. Adve, Kourosh Gharachorloo: Shared Memory Consistency Models: A Tutorial. IEEE Computer 29(12): 66-76 (1996)

Relaxed consistency models pose weaker constraints on hardware.
More difficult to reason about correctness

Example (`flag = 0`):



Memory `fence(&flag)`: Completes all writes executed before the fence and sets flag `f`

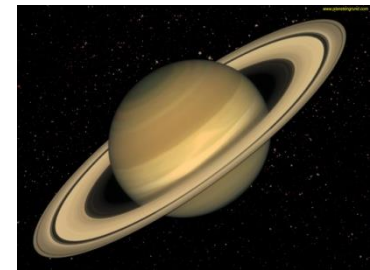
Another processor waiting for `flag` is now assured that all writes of the other processor before `flag` was set will have been completed

Typical shared-memory multi-core system (older TU Wien “Saturn”)

4xAMD “magny cours” 12-core Opteron 6168 processors
128GByte main memory, 1.9GHz, total number of cores 48

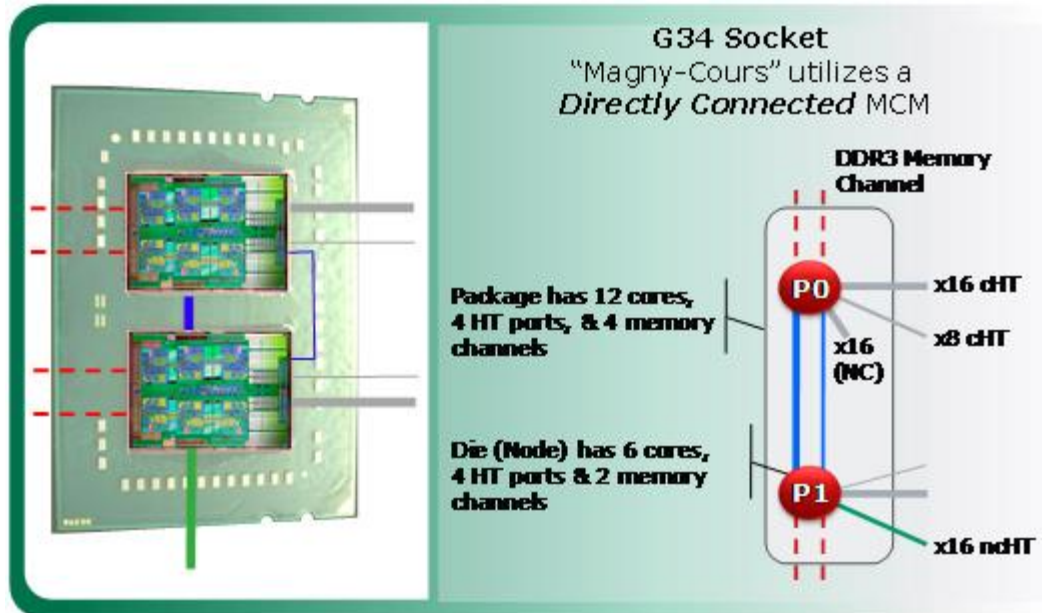
2011, now outdated

- Per core L1 cache: 128KB
- Per core L2 cache 512KB
- Shared L3 cache 12288KB



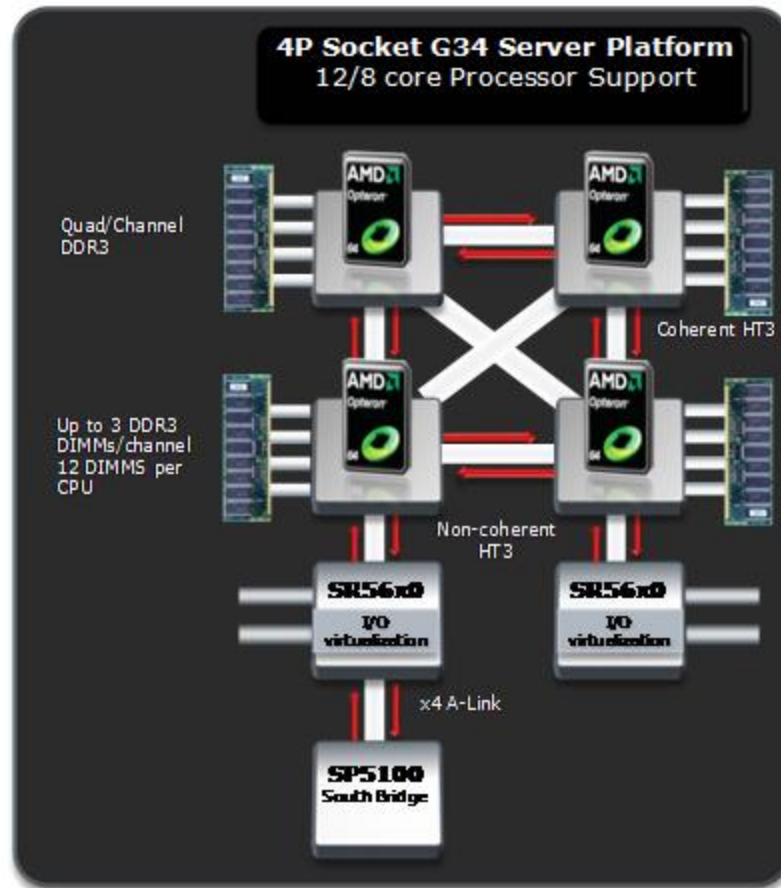
Memory consistency model: x86 total store order (TSO), writes may be reordered due to write buffer

12 core = 2x6 cores, 2 dies on chip?



Multi-processor system, built from multi-core processors

HT: HyperTransport, a standardized processor-processor interconnect

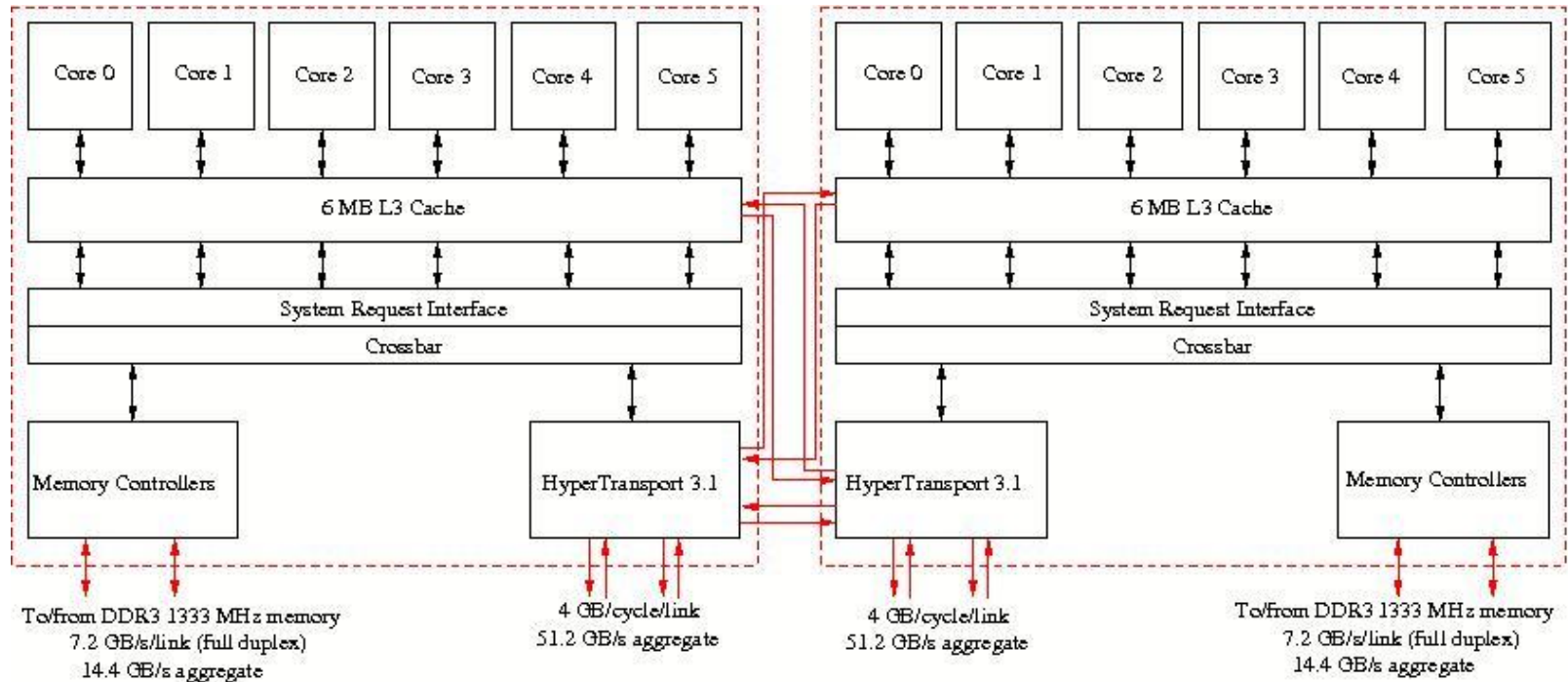


48-core shared-memory system from 4x12-core multi-core processors

Communication network lead to NUMA effects (and limits bandwidth, many cores competing for the same HT)

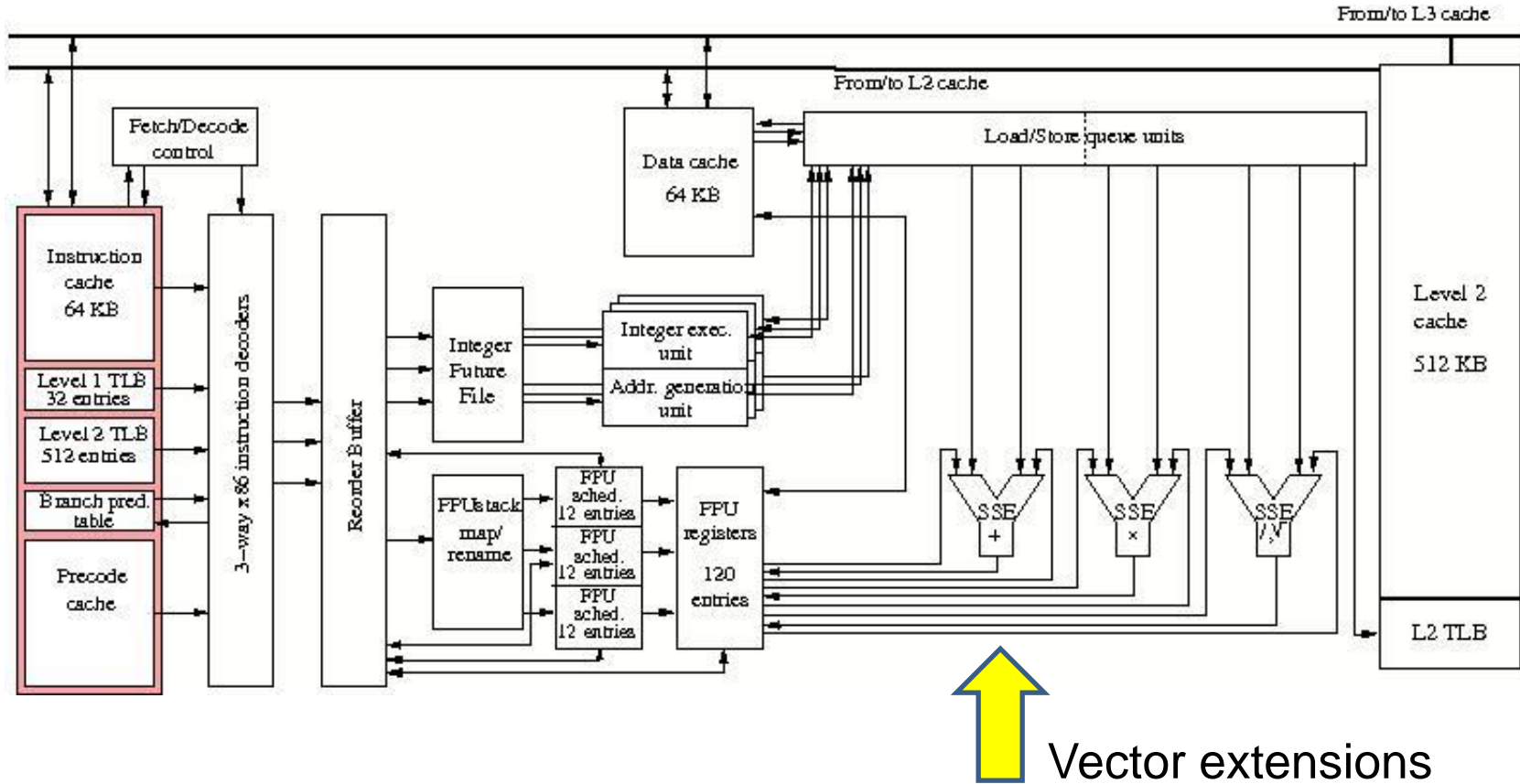
Example: 4xAMD Opteron 6168

From University of Utrecht, EuroBen: www.phys.uu.nl/eurben



Home-exercise:

Try to find the (superscalar) issue width? Peak performance? of the Opteron/Magny Cours processor



L1 cache: 64KB data, 64KB instruction

Lecture summary, checklist

- Shared-memory machines, caches, NUMA, memory hierarchy
- False sharing
- Matrix-matrix multiplication, cache behavior, recursive matrix-matrix multiplication
- NUMA performance effects, “first-touch” allocation
- Super-linear speed-up due to memory hierarchy
- Cache coherence problem
- Memory consistence problem