ID1217 Concurrent Programming
Lecture 8

ROYAL INSTITUTE
OF TECHNOLOGY

Programming Models for Multicores and their Architectures

Vladimir Vlassov
KTH/ICT/EECS

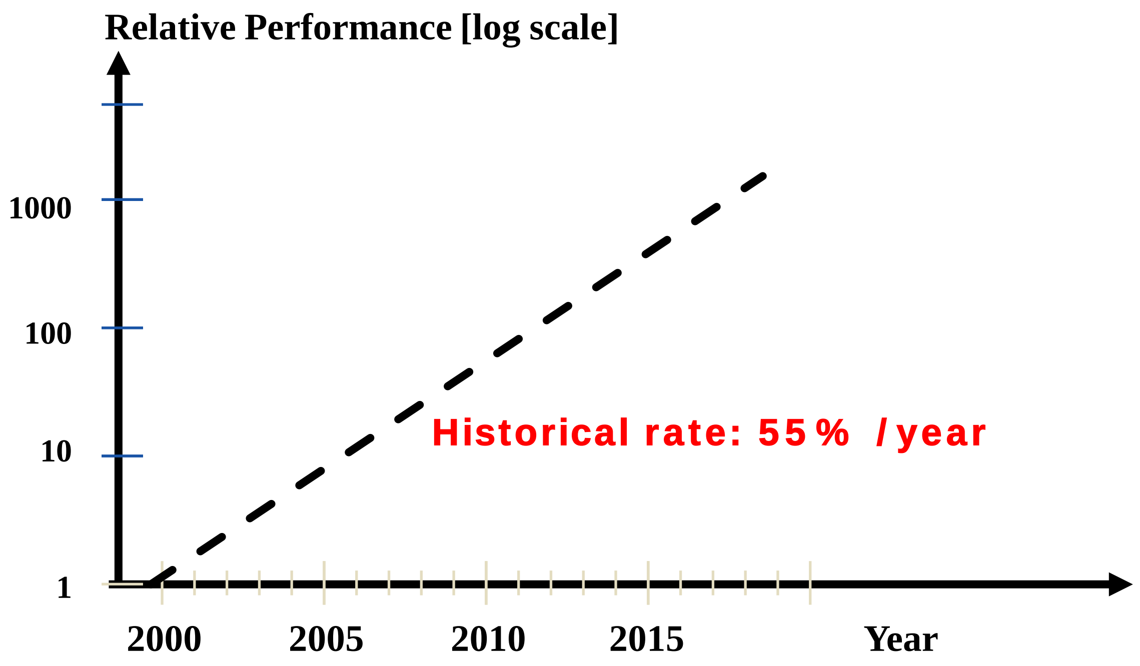
Outline

- Towards multicore CPUs
- Programming model design space
- Multicore Architecture: The main ideas, cache coherency

Moore’s Law; CPU Improvements

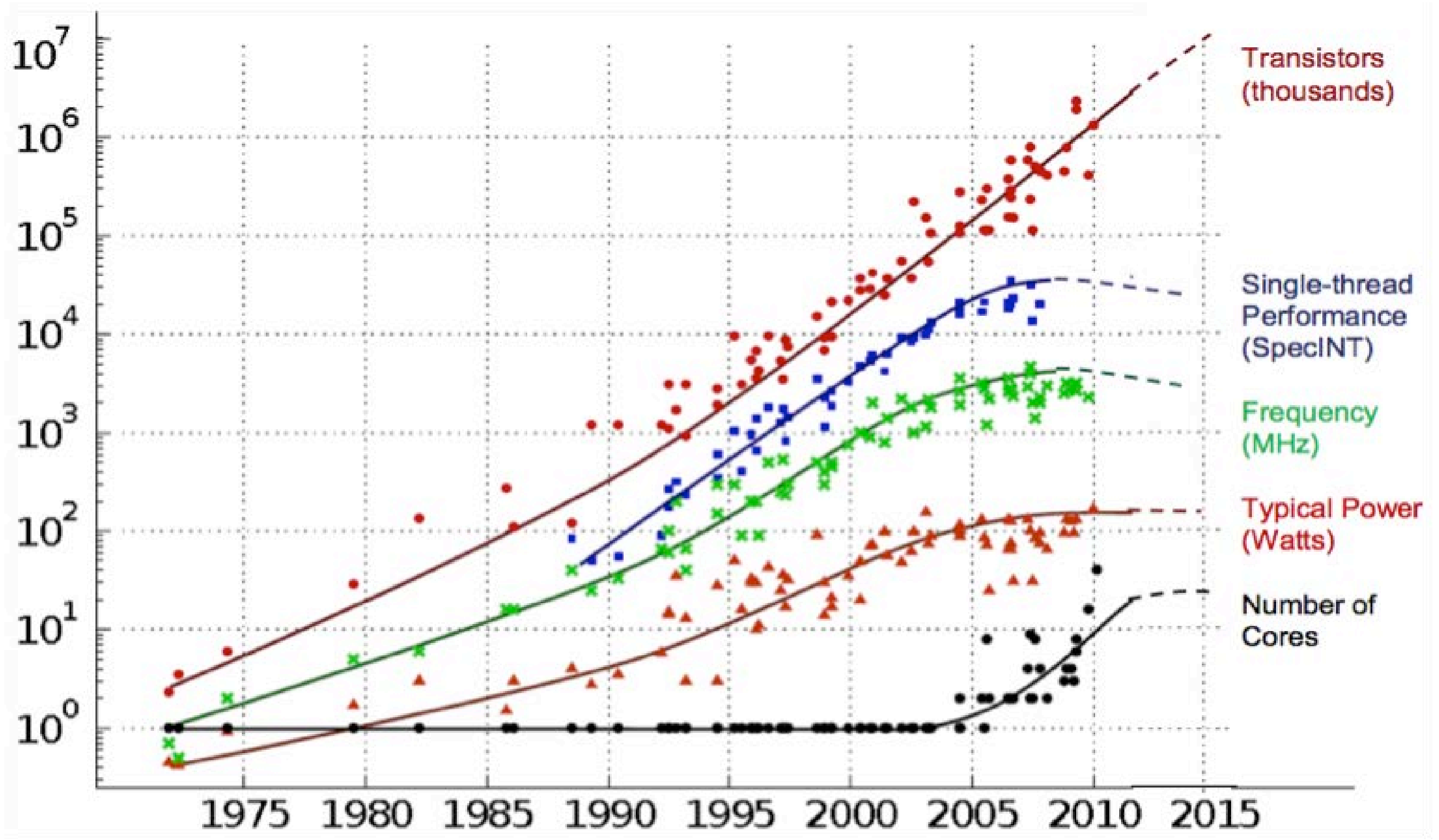
- **Moore’s law: 2X transistors / 18 months**
- Smaller transistors => faster transistors (MHz -> GHz)
- Many transistors => more on-chip state (caches)
- Many transistors => advanced architectures (ILP++)

- Processors have historically had the performance improvement rate of 50-60% per year.
- Most of that due to technology increases improvements like increasing clock frequency.



ROYAL INSTITUTE OF TECHNOLOGY

The truth...



The updated chart from "Data Processing in Exascale-Class Computing Systems" by Chuck Moore, presented at the 2011 Salishan Conference on High-speed Computing. Original data by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten, dotted line extrapolations by C. Moore

The truth…

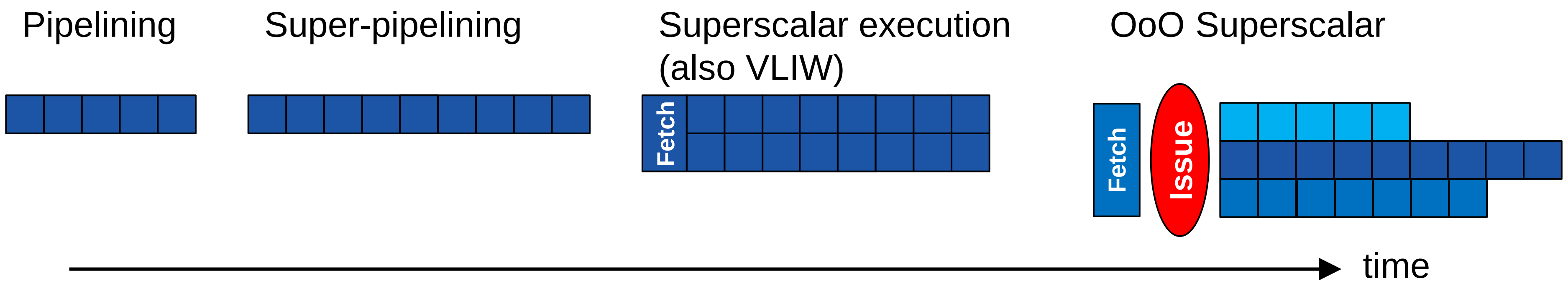
- Since the beginning of 2000, clock rate started to level off.
- The clock rate of CPUs from main vendors (Intel, AMD, IBM,…) is no longer increasing, it's even declining.
  - The top clock rate of any Intel processor was around 4 GHz, and now it's around 3-3.5 GHz.
  - It's definitely not improving.

- On the other hand the Moore's law that says that the number of transistors is doubling every 18 month, still continues.
  - Smaller transistors are faster.
  - But we cannot increase the clock rate because the power consumption goes way up, and it's impossible to cool the chip.

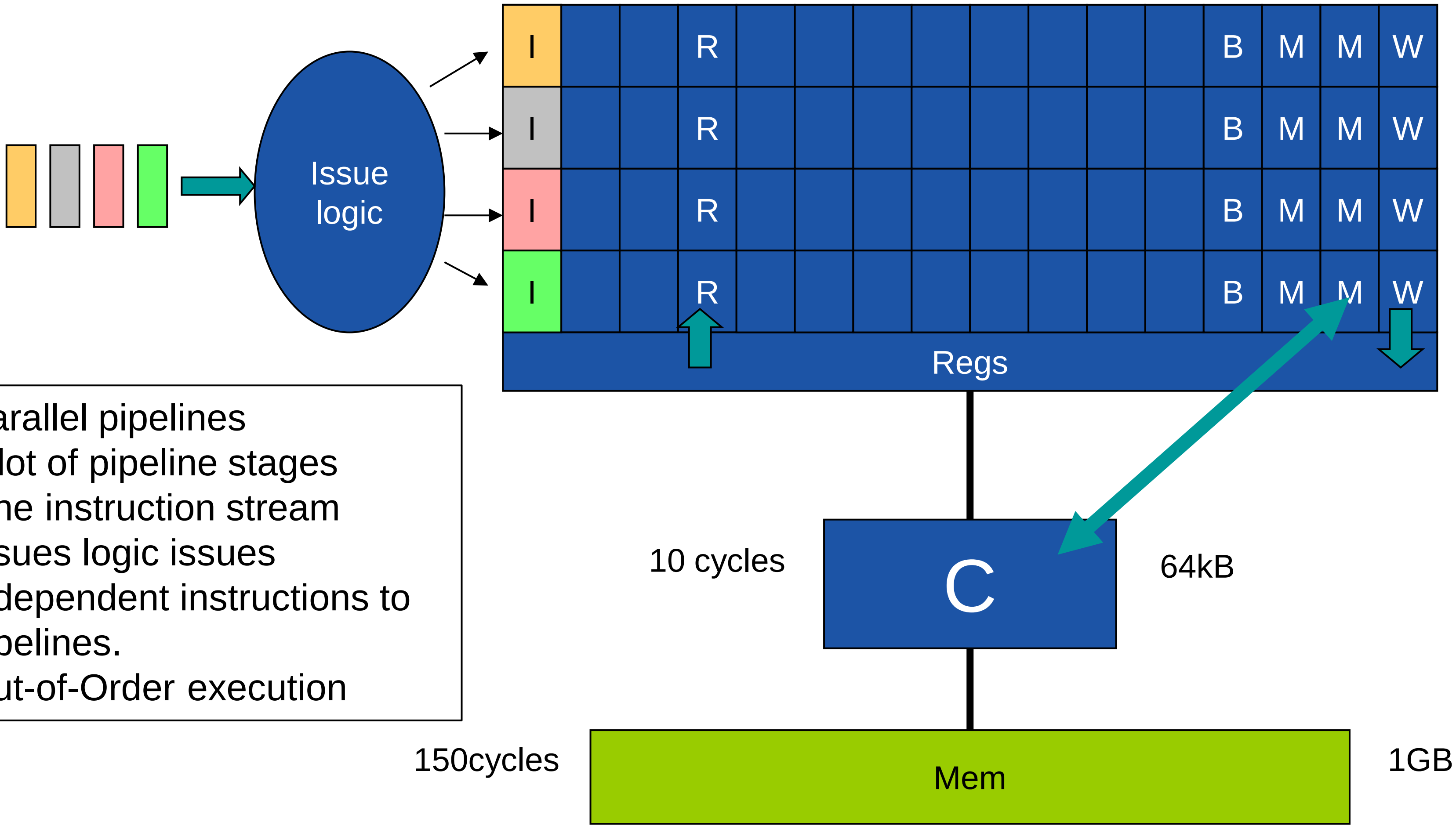
Main steps in CPU development

- Single-chip, in-order issue pipelined CPU
- Superscalar processors
- Super-pipelined processors
- Out-of-order (ooo) execution
- Speculative computing

- All in the quest for high ***Instruction-Level Parallelism (ILP)***



A High-Performance Processor Archicture



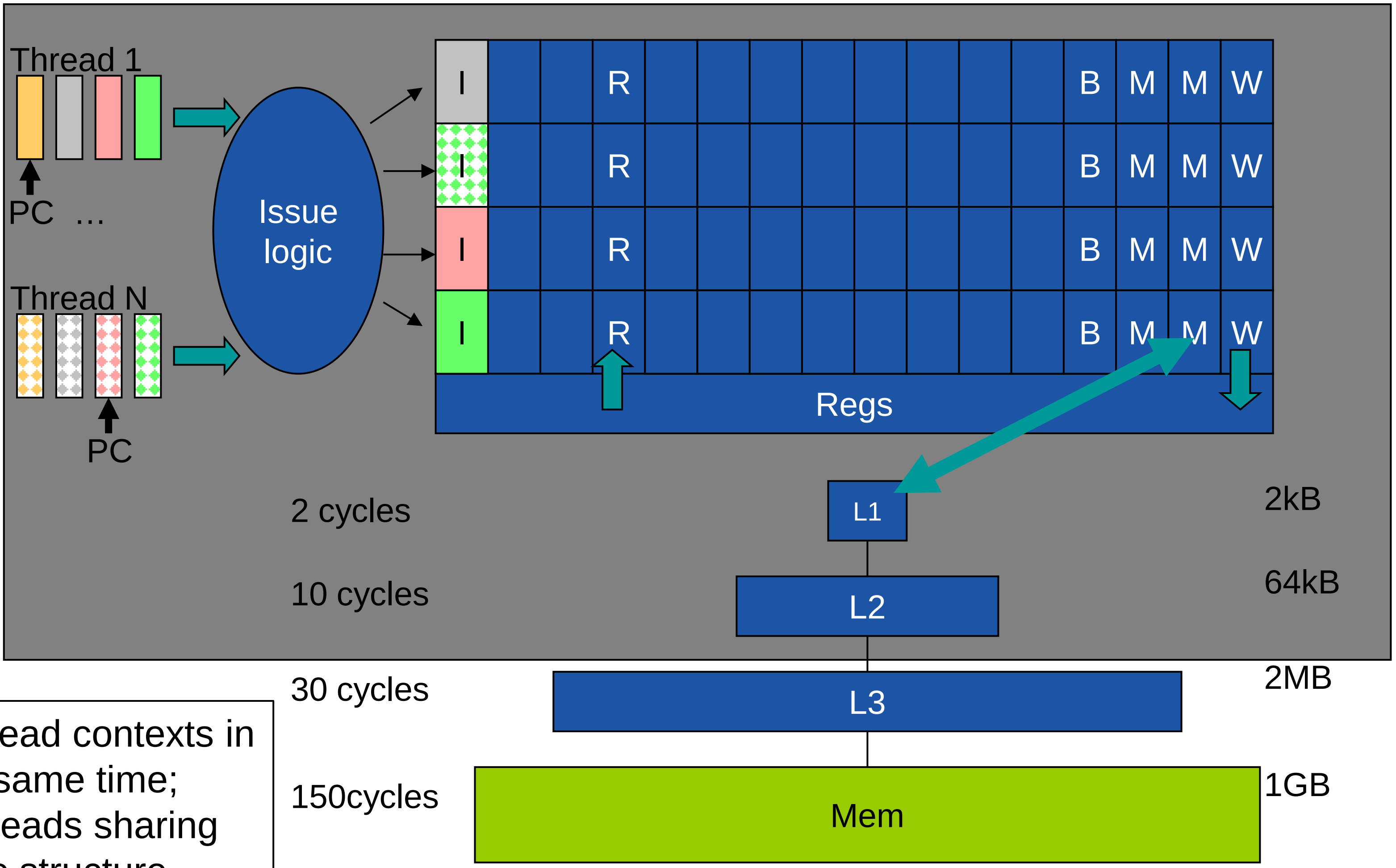
- Parallel pipelines
- A lot of pipeline stages
- One instruction stream
- Issues logic issues independent instructions to pipelines.
- Out-of-Order execution

Could we continue to make large and complicated CPUs?...

- Diminishing return on..
  - pipeline stages
  - parallel pipelines
- Smaller reach ➔
  - Hard to add complexity, smaller caches
- Memory is often the bottleneck
- We cannot keep the power budget
- The IPL Wall
  - Instruction level parallelism requires independent instructions
  - Short basic blocks lead to little opportunity to find parallelism

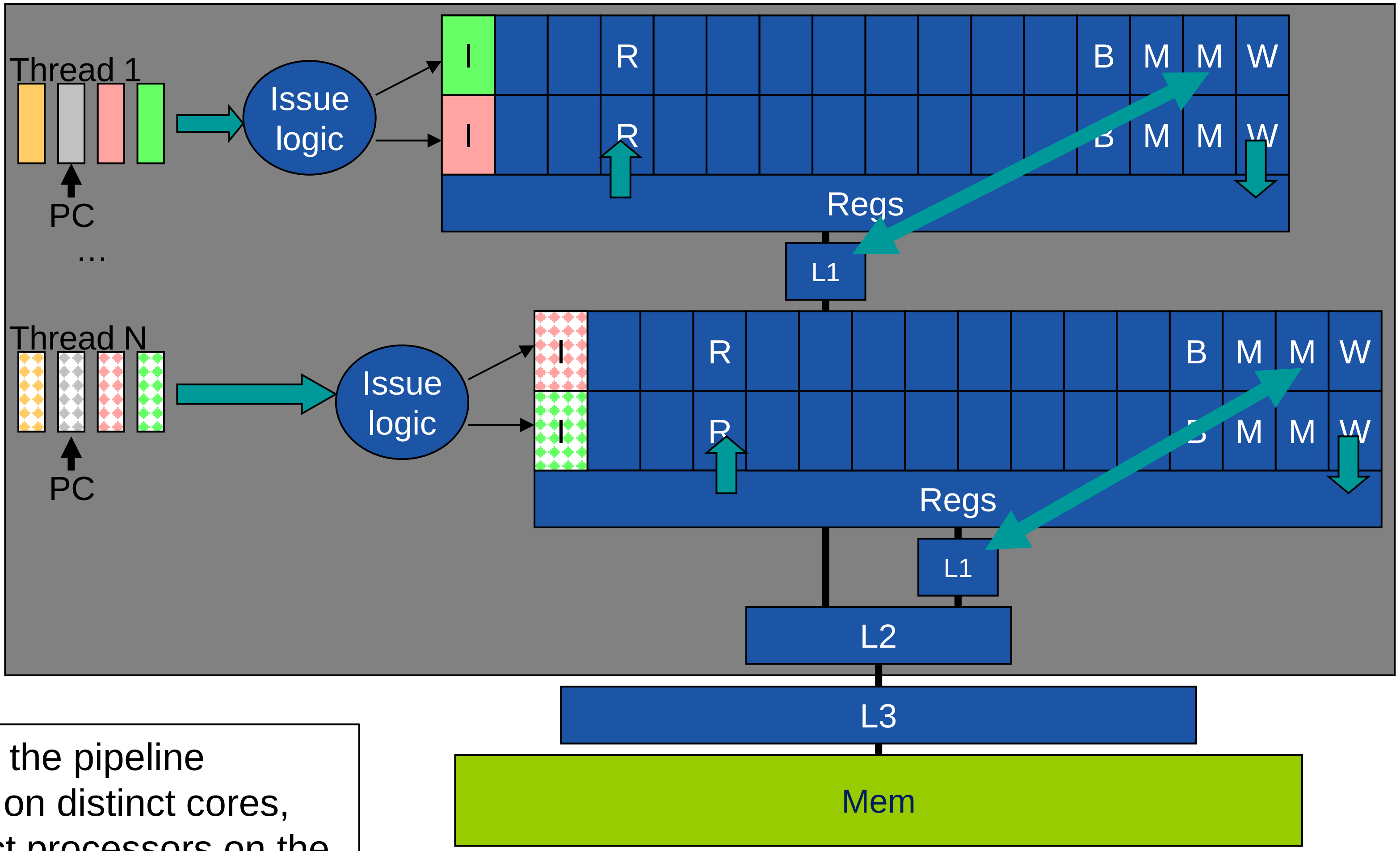
- Industry-wide trend: ***Thread-Level Parallelism (TLP)***

SMT: Simultaneous Multithreading



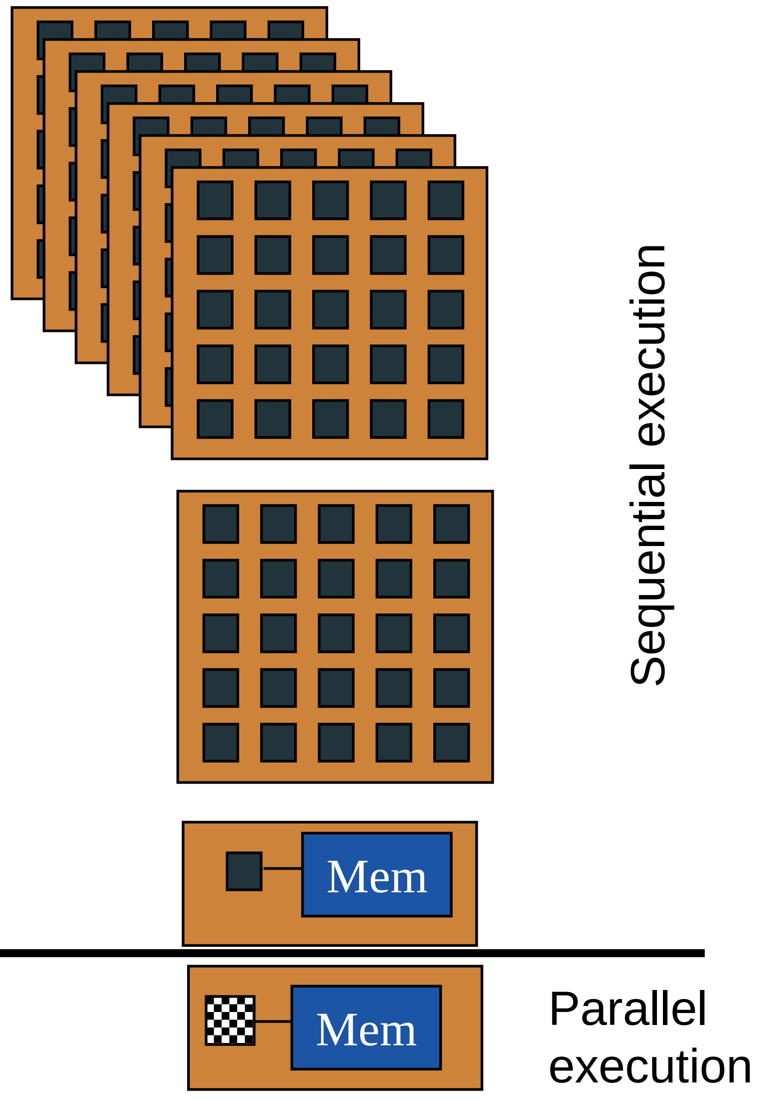
- Several thread contexts in HW in the same time;
- Multiple threads sharing the pipeline structure

CMP: Chip Multiprocessor (multi-core processor)



- Separate the pipeline structure on distinct cores, on distinct processors on the same chip, each executing a single or multiple threads.

Why not earlier?



ROYAL INSTITUTE OF TECHNOLOGY

Many years experience

- We had a working horse with many years experience, and it worked well, performed better and better every year. Sort of “horse on steroids”, but now…

New challenges...

$+$

$= \quad ?$

- Now we are facing small little rabbits that are still going to carry around the same logs as the horse before and we need to coordinate them in some way.
  - So, that is, sort of, the reason for not doing this earlier.
- A *paradigm shift in programming*.
- And every software developer has to deal with it now.

ROYAL INSTITUTE OF TECHNOLOGY

Why Multicore (1/2)

- Software developers were used to ever increasing performance, for free
  - Increasing clock frequencies
  - Increasingly advanced architectures exploiting Instruction level parallelism (ILP)
  - Larger cache memories
- However,
  - Clock frequencies no longer go up!
  - We have exhausted the available ILP in applications
- ***Parallelism is the only way increasing performance significantly***

Why Multicore (2/2)

- Because these are the only chips we know how to build with the increasing number of transistors within the power envelope
- Because of performance reasons: potential of linear performance improvements with growing number of cores
  - If we could only program them...
- Because of power reasons

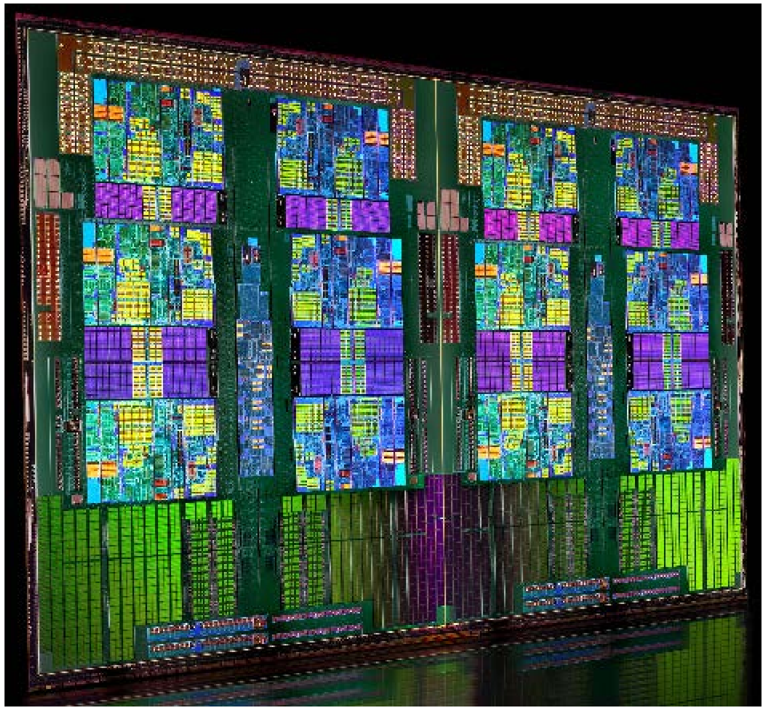
ROYAL INSTITUTE OF TECHNOLOGY

Multicore Processors

Huge design space:

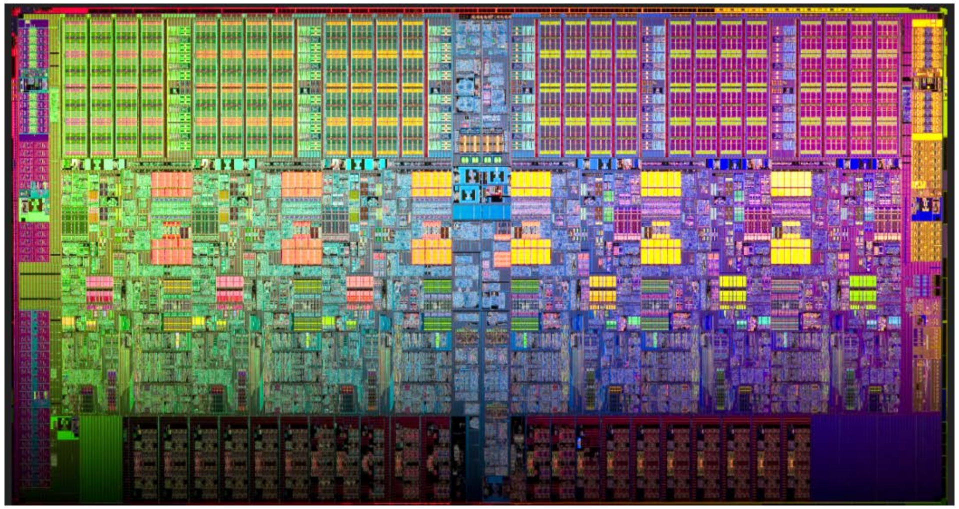
- Homogeneous vs heterogeneous
- Low count, fat cores vs High count, thin cores
- Bus-based interconnect vs Network on chip
- Shared memory vs Message passing
- …

Example 1: AMD Opteron 6172 “Magny-Cours”



- 12 x x86 cores
- Private L1/L2 caches
- Shared L3 cache between 6 cores
- Software compatibility

Example 2: Intel® Xeon X5650 ”Gulftown”

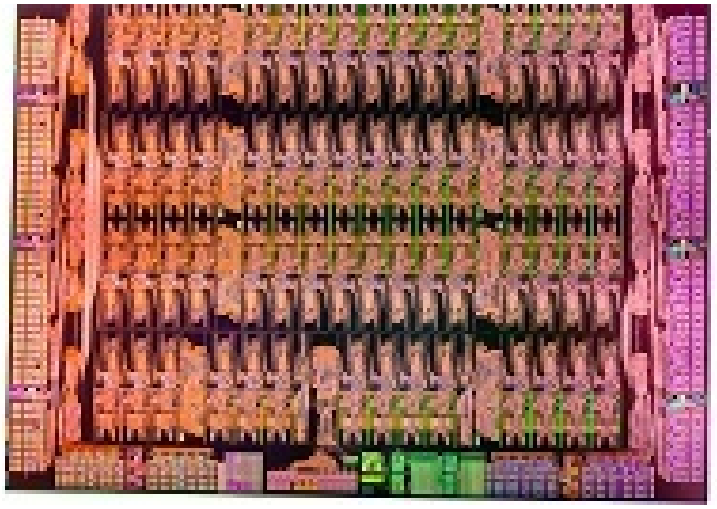


- 6 cores (12 threads)
- 2-way hyperthreading (2 hw threads / core)
- Private L2 caches
- Shared L3 cache

See: http://www.bit-tech.net/hardware/cpus/2010/03/31/amd-opteron-6174-vs-intel-xeon-x5650-review/

ROYAL INSTITUTE OF TECHNOLOGY

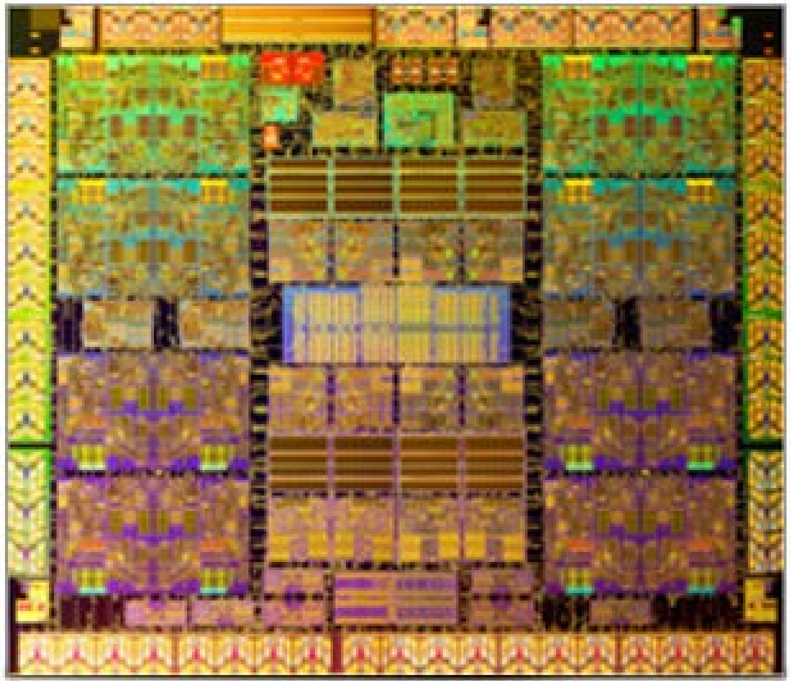
Example 3: Intel® Xeon Phi™



- 60 x86-64 cores with 4 threads each
- Cache coherence
- Ring interconnect
  - 512-bit
- Vector processing
  - 32 512-bit vector registers

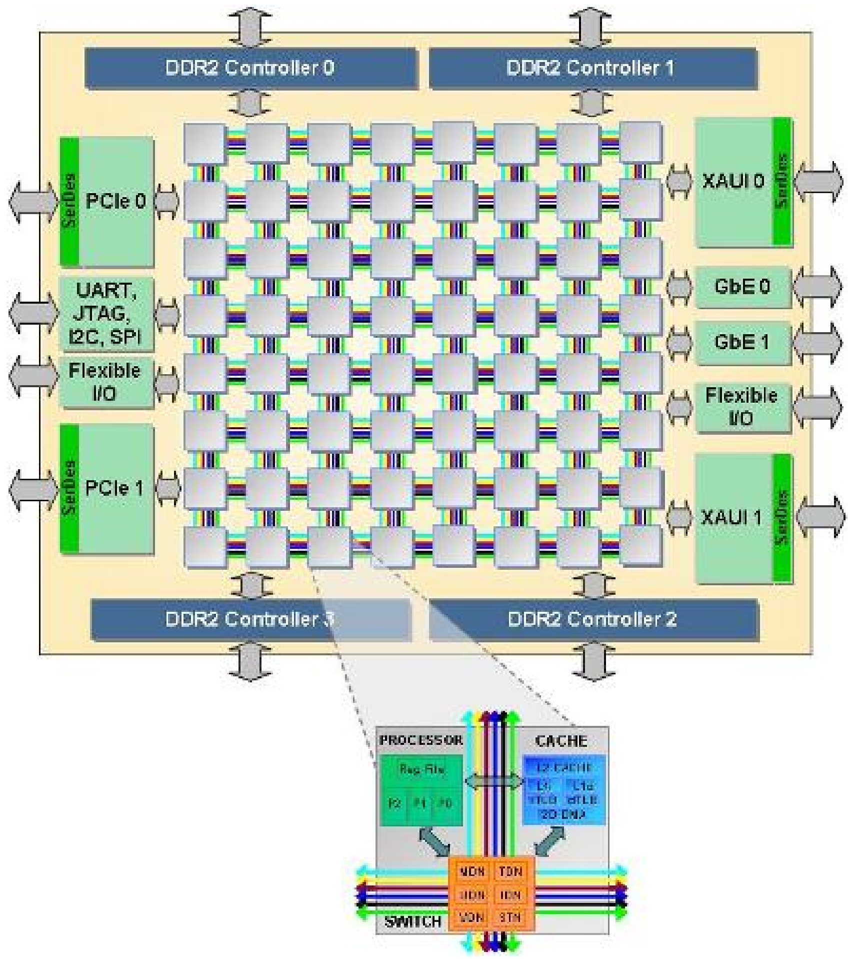
ROYAL INSTITUTE
OF TECHNOLOGY

Example 4: Oracle’s UltraSPARC T5



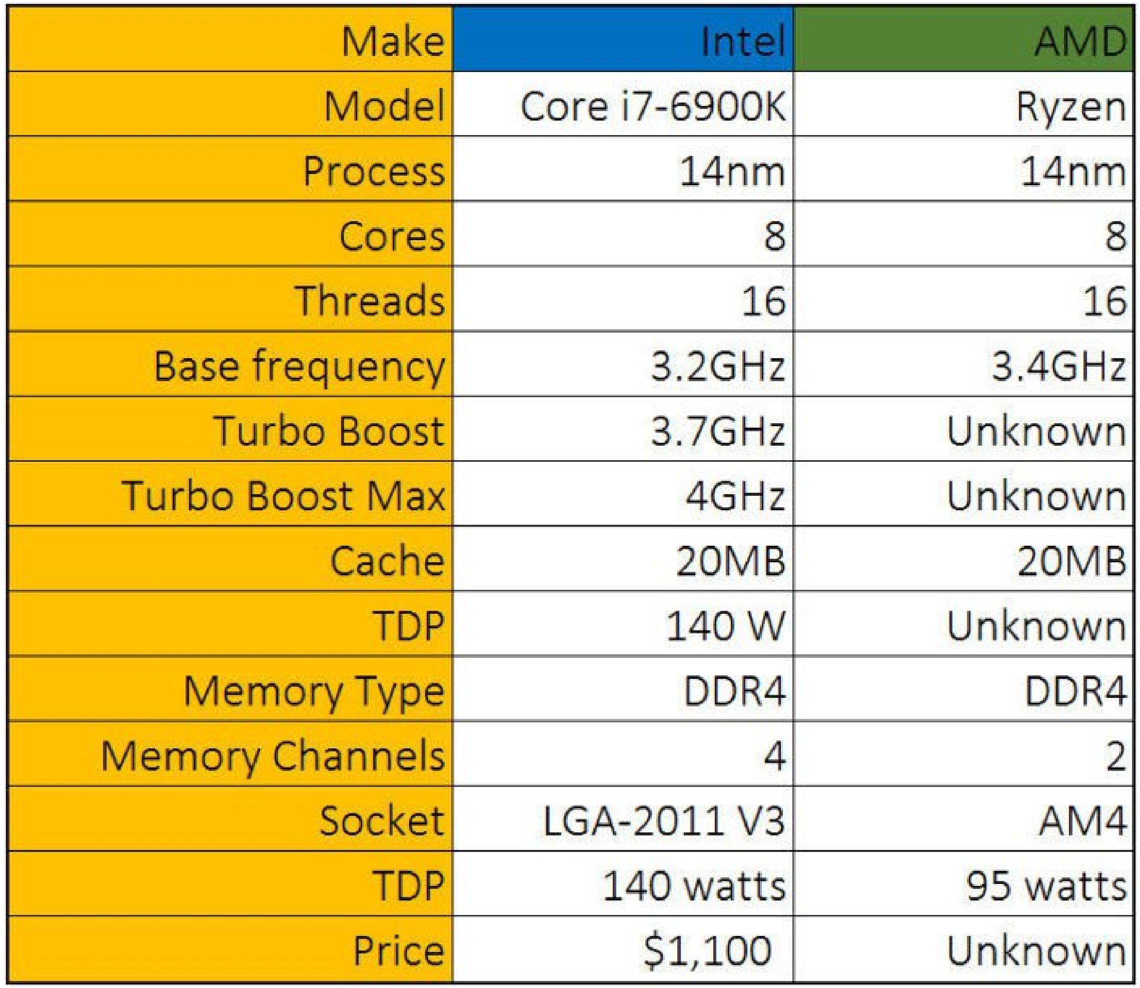
- 16 cores, 128 threads (8 threads per core)
- A three-level cache architecture
  - L1 and L2 are private
  - L3 is shared
- Cross-bar interconnect
- Huge software support
  - Linux/Solaris
  - All kinds of languages

Example 5: Tilera Tile-Gx



- 100 cores
- Cache coherent
- Standard OS (Linux)
- C/C++
- Pthreads
- Highly configurable
  - Programmable interconnect

AMD Ryzen vs Intel Core I7



Make	Intel	AMD
Model	Core i7-6900K	Ryzen
Process	14nm	14nm
Cores	8	8
Threads	16	16
Base frequency	3.2GHz	3.4GHz
Turbo Boost	3.7GHz	Unknown
Turbo Boost Max	4GHz	Unknown
Cache	20MB	20MB
TDP	140 W	Unknown
Memory Type	DDR4	DDR4
Memory Channels	4	2
Socket	LGA-2011 V3	AM4
TDP	140 watts	95 watts
Price	$1,100	Unknown

http://www.pcadvisor.co.uk/new-product/pc-components/amd-zen-processor-release-date-price-specs-features-3643552/

What to expect from the future…

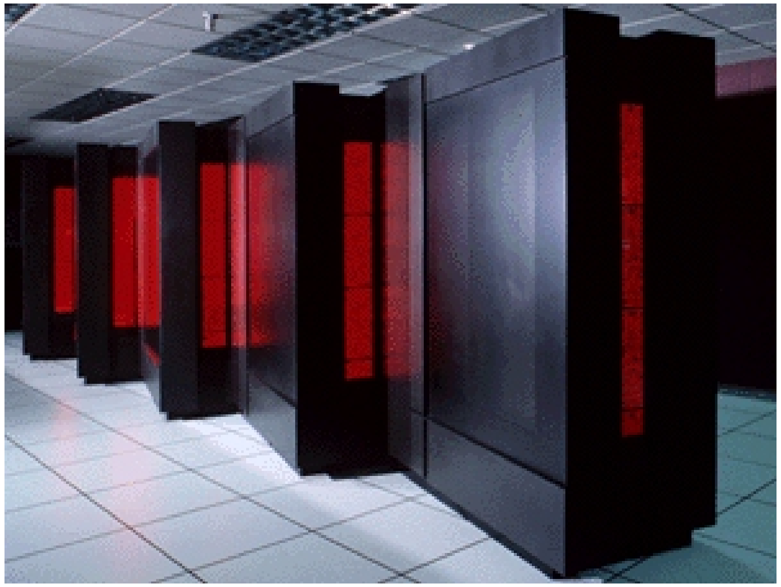
By 2020 we will have…

- 200-2000 cores per chip*
  - – 35 Billion transistors
  - – 400 mm^2 chip area
- Point-to-point on-chip interconnection networks
  - - For design productivity
- Shared distributed memory
  - - For software compatibility
- Non-reliable hardware
  - - Software makes it reliable
- Highly dynamic workload
  - - Workload adapting to architecture and
  - - Architecture adapting to workload

* International Technology Roadmap for Semiconductors

The era of the "Supercomputers"

- The one with the most blinking lights wins
- The one with the strangest languages wins
- The niftier the better!



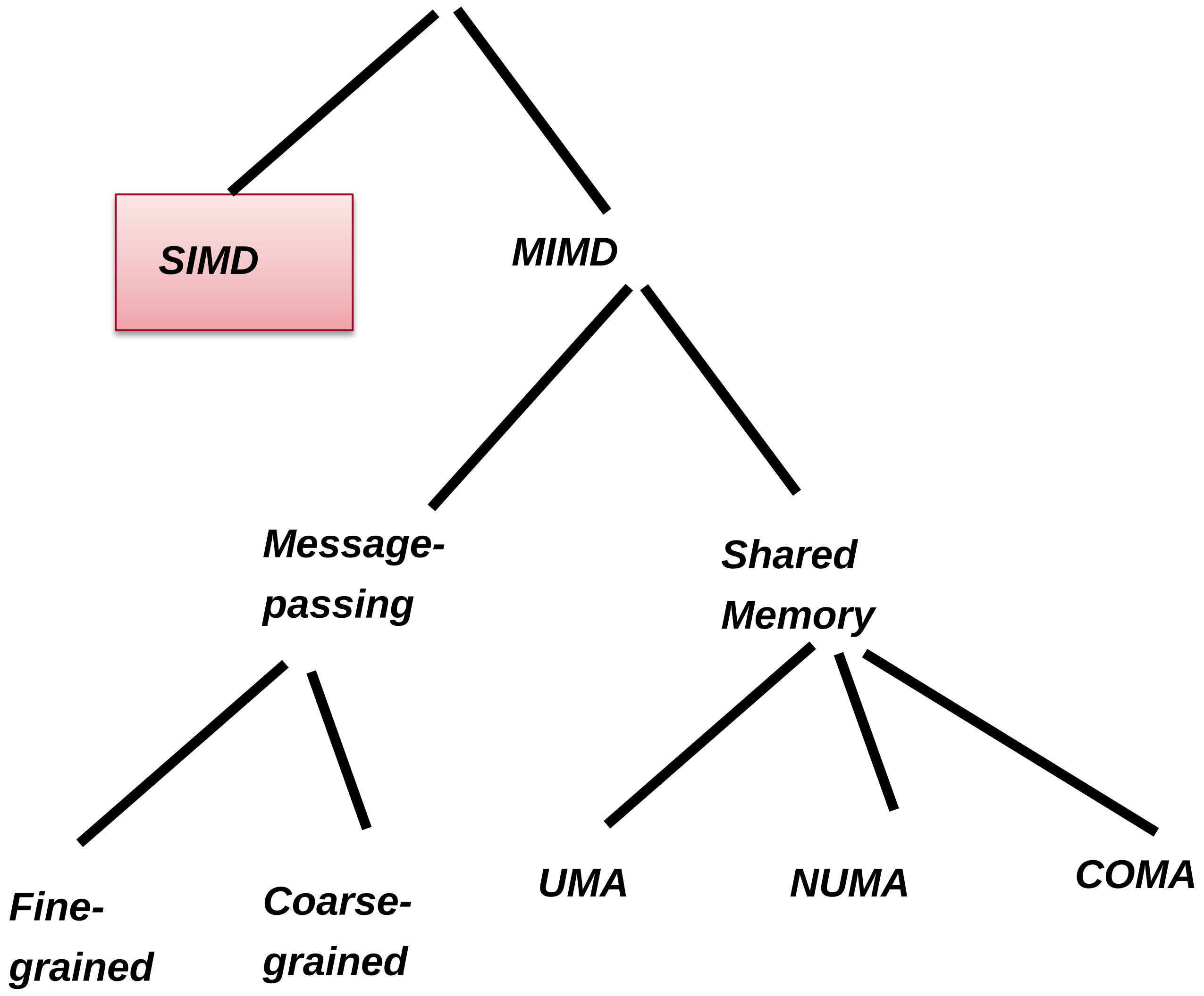
Connection Machine 5 by Thinking Machines Corporation

Flynn’s Taxonomy

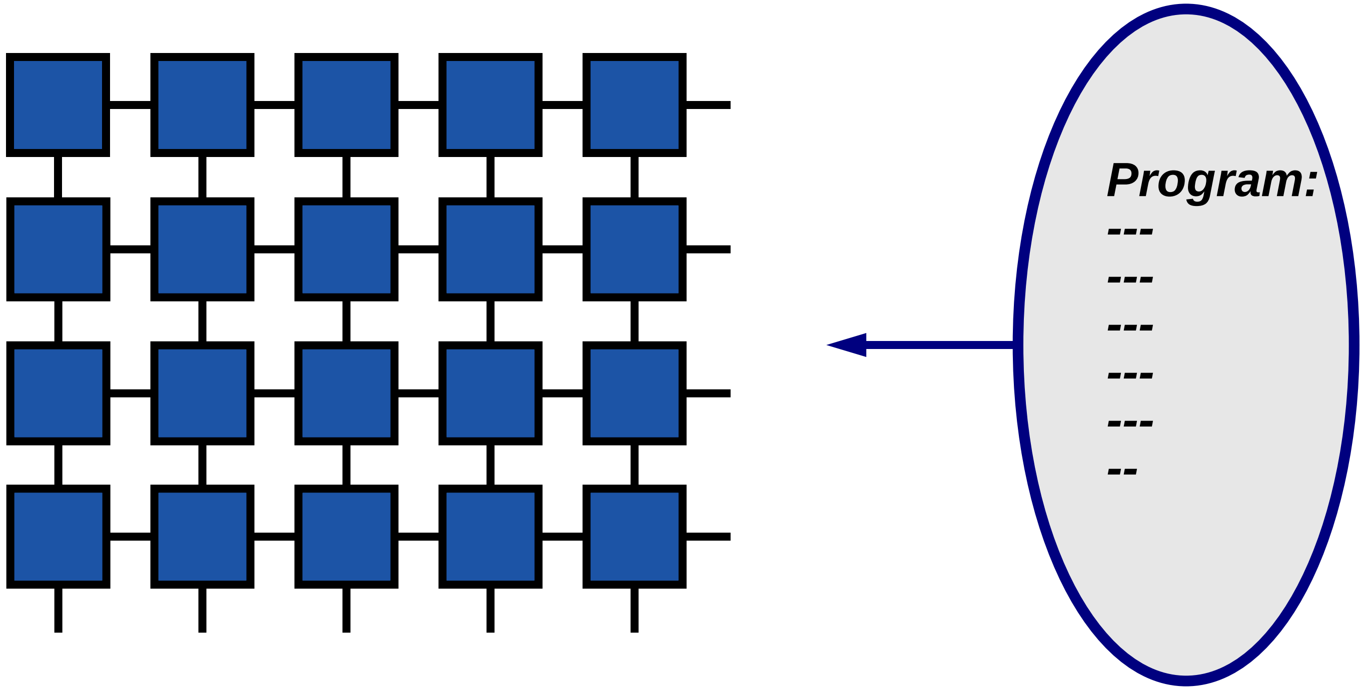
{Single,Multiple}Instruction + {Single,Multiple}Data

- SISD - Our good old simple CPUs
- SIMD – Vectors, ”SSE”, DSPs, CM-2,...
- MIMD – TLP, cluster, shared-mem MP,...
- MISD – Can’t think of any…

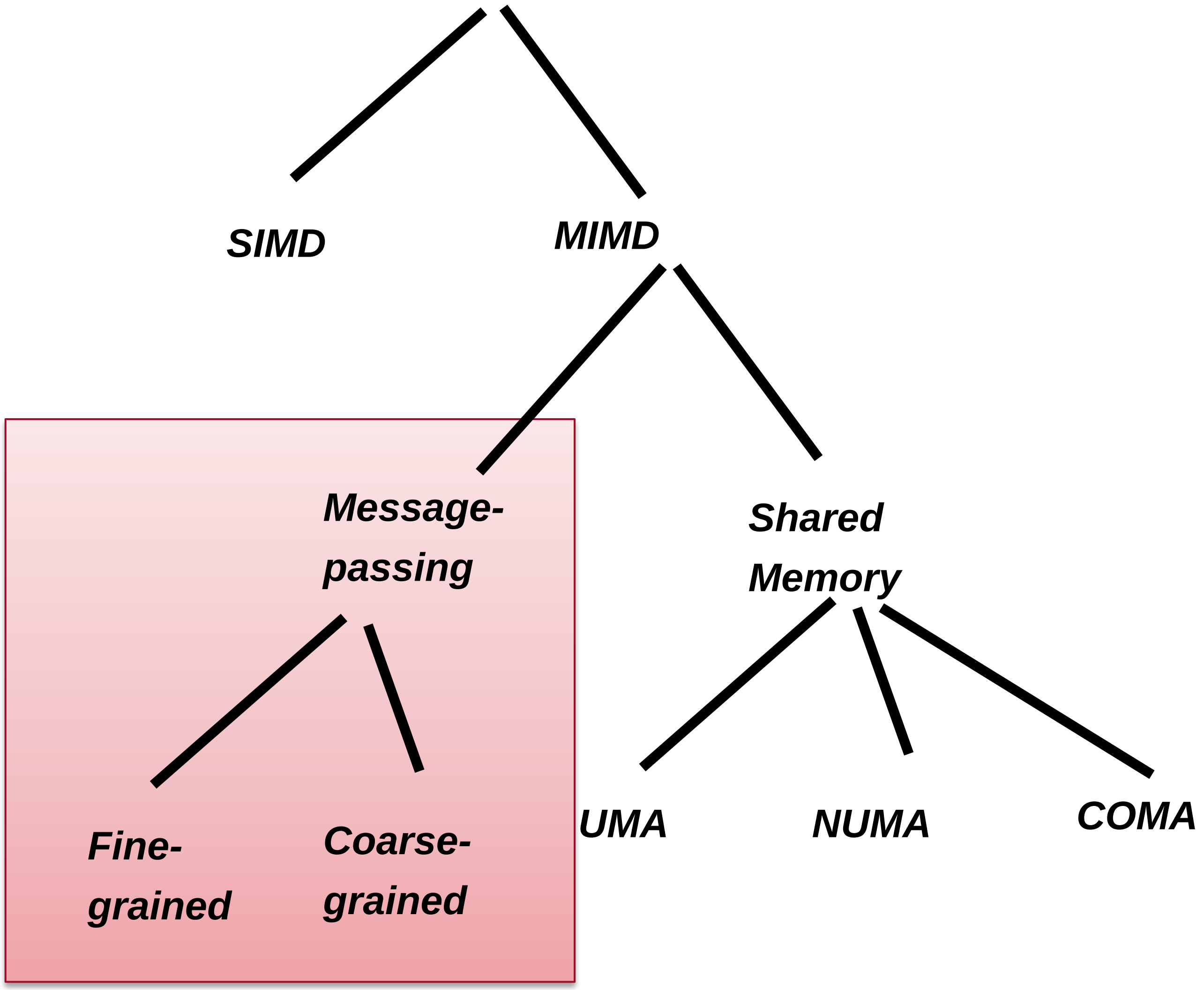
MP Taxonomy


SIMD
MIMD
Message-passing
Shared Memory
Fine-grained
Coarse-grained
UMA
NUMA
COMA

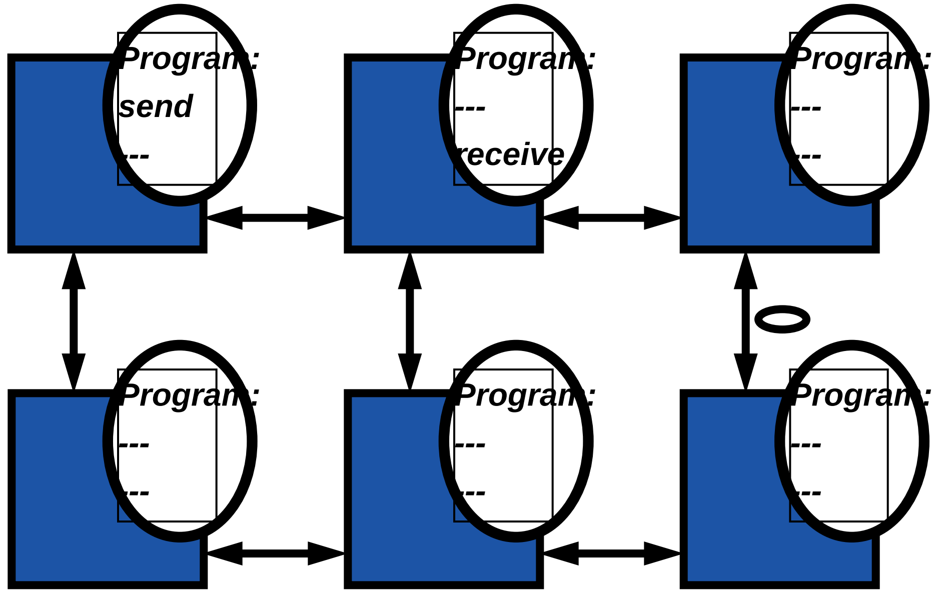
SIMD



Message-passing

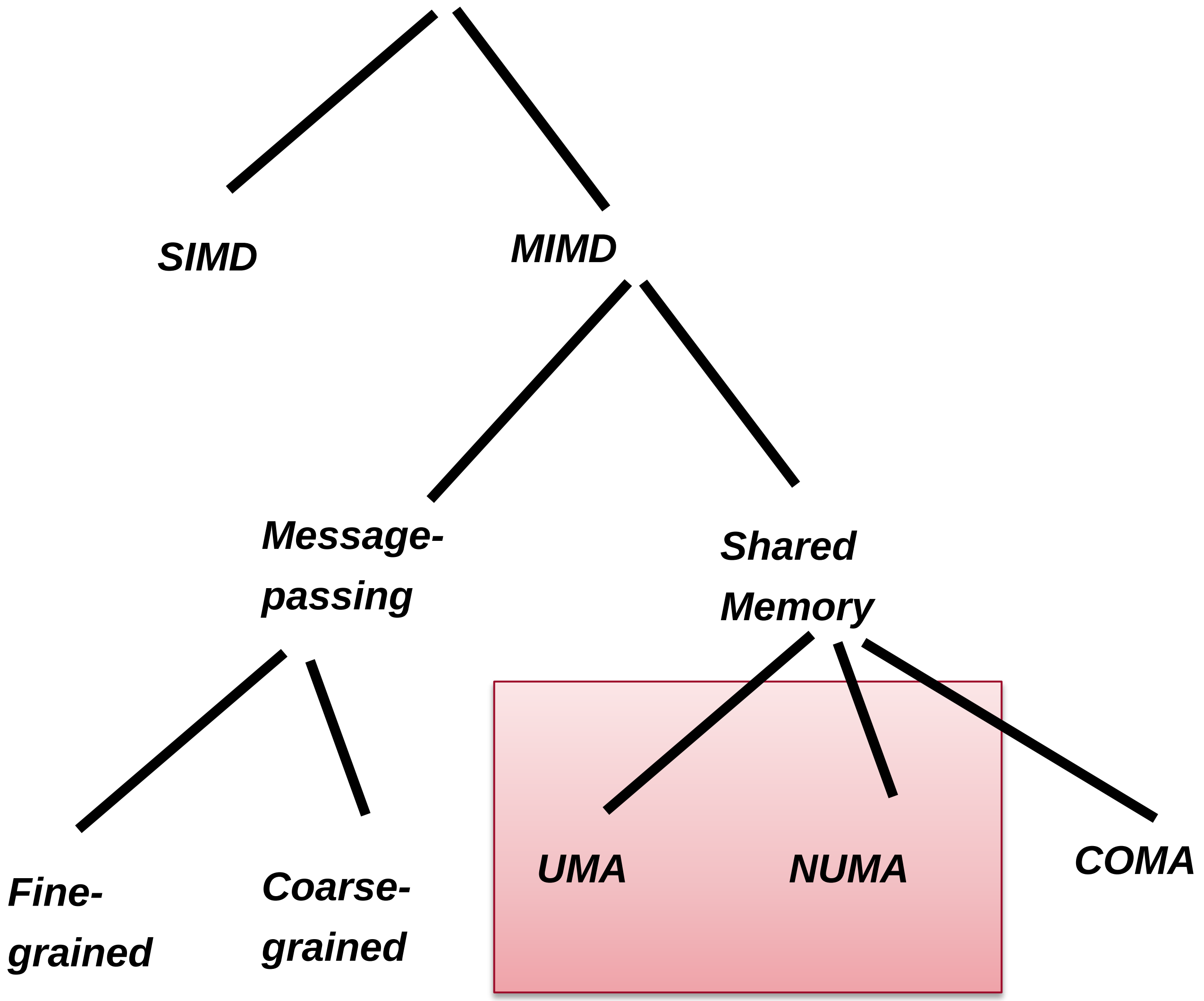


Message-passing Arch MIMD

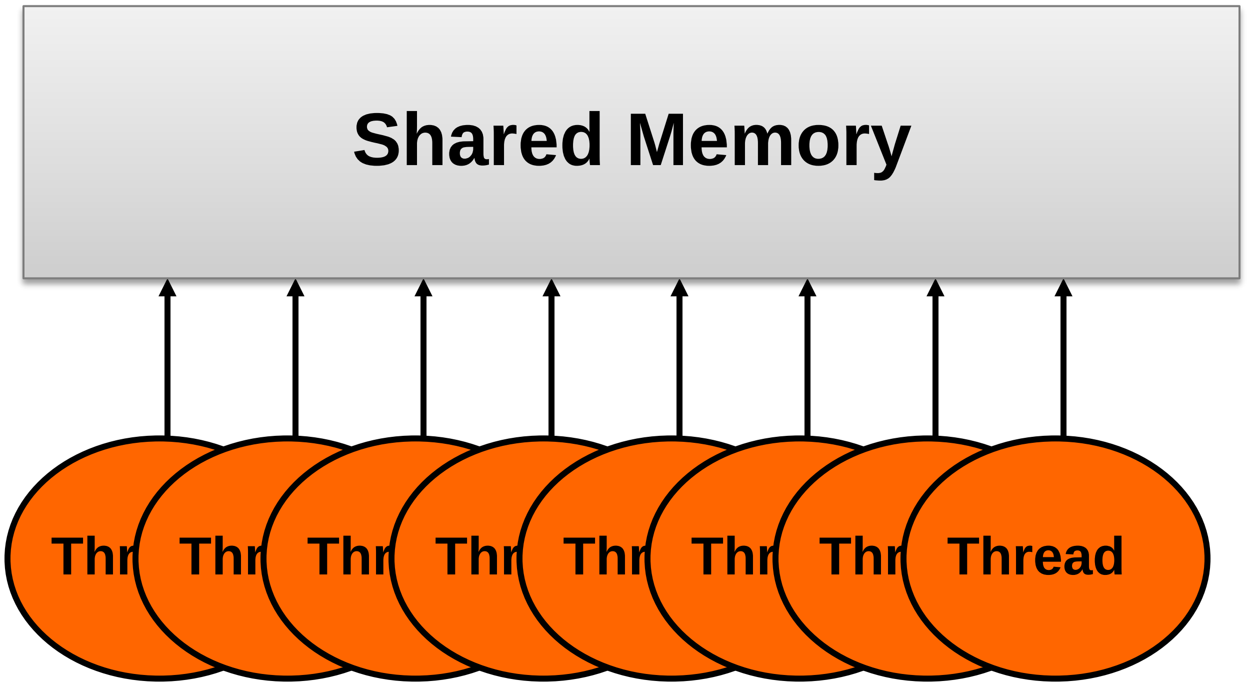


Explicit Messages

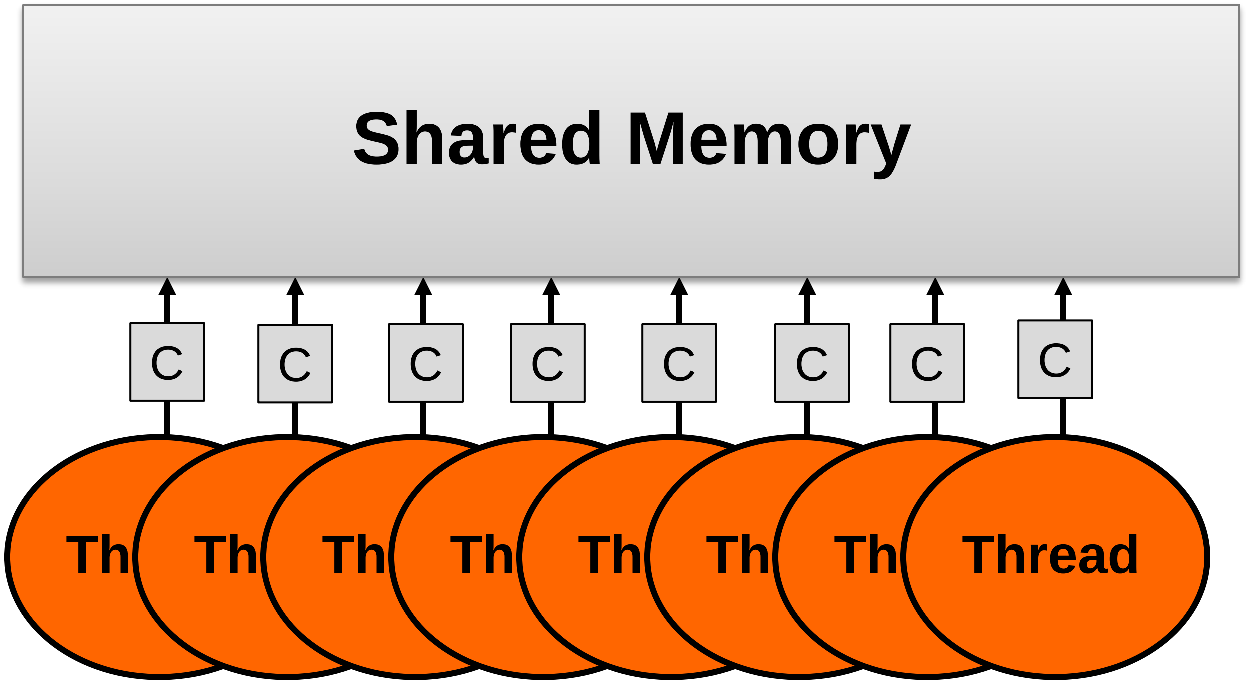
Shared Memory MIMD



The Shared Memory Model

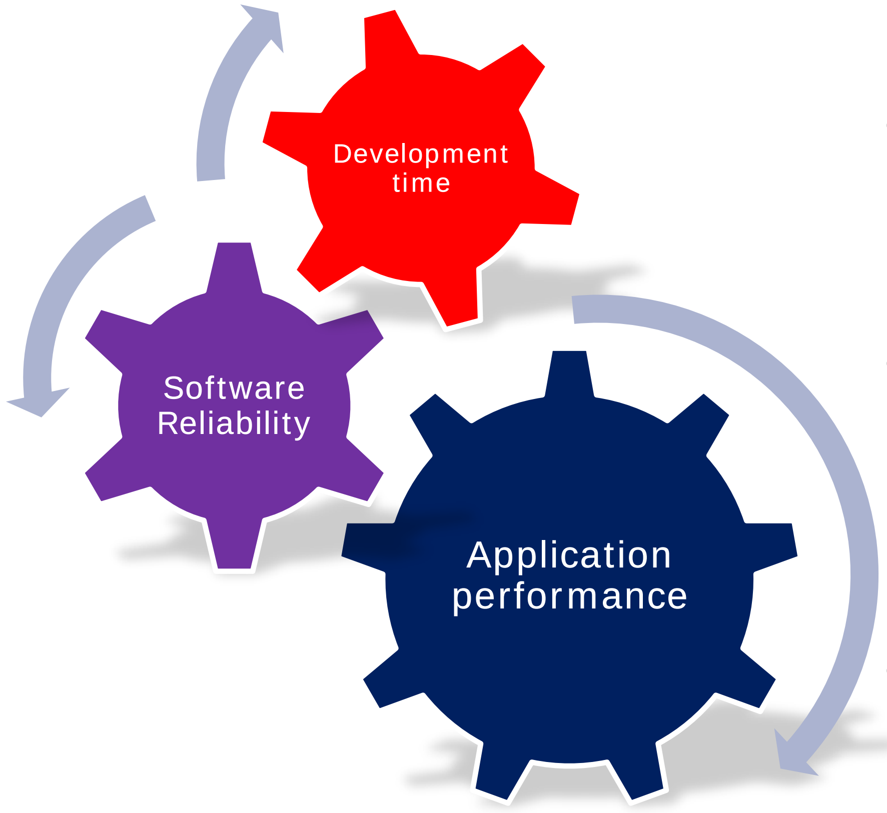


Adding Caches: More Concurrency



- Two coherence options
  - Snoop-based
  - Directory-based

The Multicore Software Triad

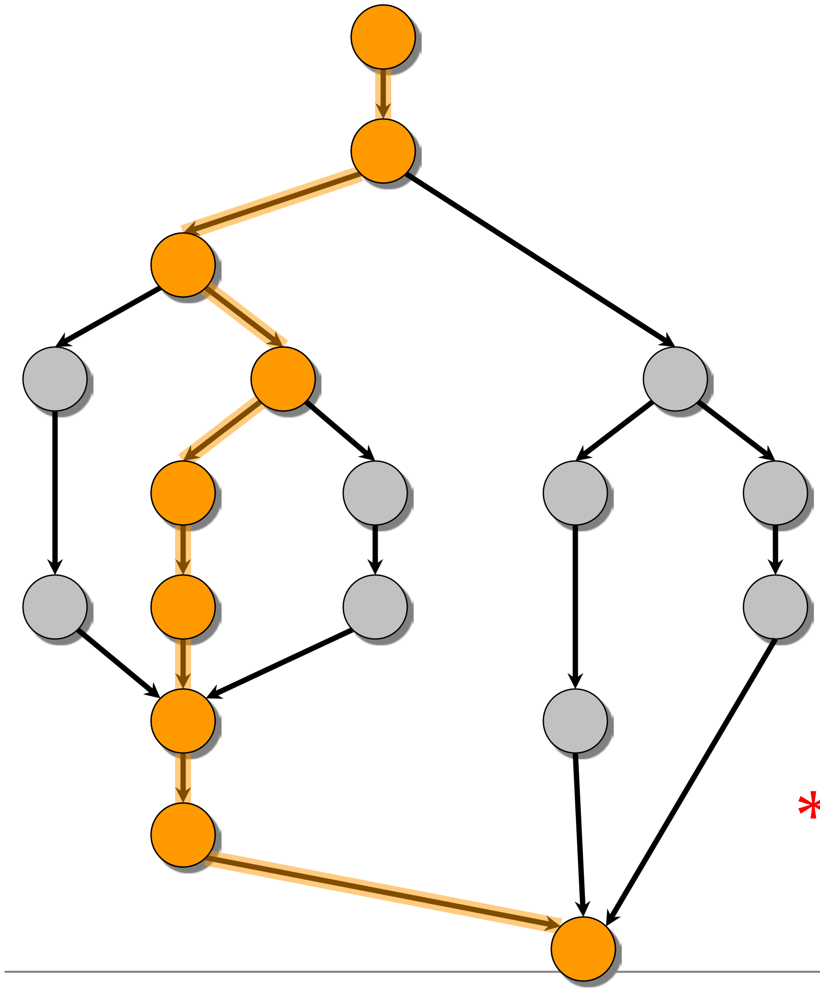


Desirable properties

- Application performance: Scalable up and down
- Software reliability: Do not introduce new bugs with parallelism
- Development time: Easy to write and maintain

Algorithmic Complexity Measures

T_P = execution time on P processors



T_1 = *work*

T_∞ = *span**

Lower Bounds

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

*Also called *critical-path length* or *computational depth*.

ROYAL INSTITUTE OF TECHNOLOGY

Parallelism

Defined as: T_1/T_∞

1. T_1/T_∞ is the average amount of work along each step of the critical path
2. T_1/T_∞ is the maximum possible speedup on any number of processors
3. Perfect linear speedup ($T_1/T_P = P$) cannot be obtained for any $P > T_1/T_\infty$

Programming model design space

- General-purpose languages (GPL)
  - Native threads/processes
  - Data parallelism
  - SPMD
  - Loop-based parallelism
  - Task-based parallelism
- Domain-specific languages (DSL)
  - Erlang/SDL/Matlab/Labview
  - GPU-languages, e.g., CUDA
- Automatic parallelization
- Tool support

ROYAL INSTITUTE OF TECHNOLOGY

A coding example

```c
#include <stdlib.h>
#include <stdio.h>
long fib(long n) {
    long x, y;
    if (n < 2)
      return n;
    else {
      x = fib(n - 1);
      y = fib(n - 2);
    }
    return x + y;
}
int main(int argc, char *argv[]) {
    long n = (argc > 1) ? atoi(argv[1]) : 30;
    long result;
    result = fib(n);
    printf("fib(%ld) = %ld\n", n, result);
}
```

- Serial code
- Fibonacci numbers
- Stupid algorithm
- Good didactic example

ROYAL INSTITUTE OF TECHNOLOGY

Explicit threading: pthreads

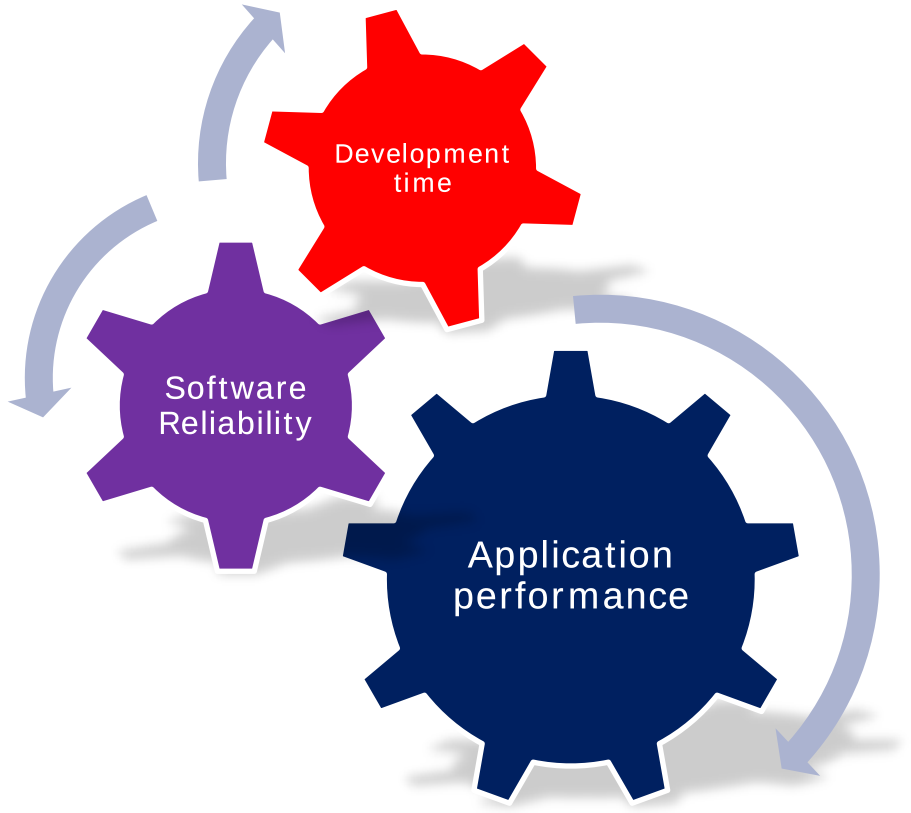
```c
#ifndef _REENTRANT
#define _REENTRANT
#endif
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
long fib(long n){
    long x, y;
    if (n < 2) return n;
    else {
        x = fib(n-1);
        y = fib(n-2);
    }
    return x + y;
}
typedef struct {
    long input;
    long output;
} thread_args;

void *thread_func(void *ptr){
    long i = ((thread_args *)ptr)->input;
    ((thread_args *)ptr)->output = fib(i);
return NULL;
}
```

```c
int main(int argc, char *argv[]){
    pthread_t thread;
    thread_args args;
    long n = (argc > 1)? atoi(argv[1]) : 30;
    long result;
    if (n < 30) result = fib(n);
    else {
      args.input = n-1;
      pthread_create(&thread, NULL, thread_func,
                     (void*)&args);
    result = fib(n-2);
    pthread_join(thread, NULL);
    result += args.output;
    }
    printf("fib(%ld) = %ld\n", n, result);
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Consequences of pthreads



Desirable properties:

- Scalable up and down
  - Code is fixed to two threads
- Do not introduce new bugs with parallelism
  - Code no longer modular
- Easy to write and maintain
  - 50 year leap backwards!

ROYAL INSTITUTE OF TECHNOLOGY

Same example using OpenMP

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
long fib(long n){
    long x, y;
    if (n < 2) return n;
    else {
#pragma omp task shared(x)
        x = fib(n-1);
#pragma omp task shared(y)
        y = fib(n-2);
#pragma omp taskwait
    }
    return x + y;
}
```

```c
int main(int argc, char *argv[]){
    long n = (argc > 1)? atoi(argv[1]) : 30;
    long result;
#pragma omp parallel
#pragma omp single
    result = fib(n);
    printf("fib(%ld) = %ld\n", n, result);
}
```

(some) Concurrency platforms

- Task-based languages:
  - OpenMP
    - also loop parallelism
  - Cilk(++)
- Data parallel languages
  - Ct
  - HPF

- Message passing libraries:
  - MPI
  - MCAPI
- Task-parallelism libraries:
  - Intel Threading Building Blocks
  - Microsoft TPL
  - Wool
  - Apple GCD

Another coding example

$$X_i^{(t+1)} = \frac{X_{i-1}^{(t)} + 2X_i^{(t)} + X_{i+1}^{(t)}}{4}, \quad 0 < i \le N, 0 \le t < T$$

```c
#include <stdlib.h>
#include <stdio.h>
#define T 10
#define N 100
double X1[N+2], X2[N+2]; // initially zeros
int main(int argc, char *argv[]) {
  int t, i;
  int size = (argc > 1)? atoi(argv[1]) : N;
  if (size > N) size = N;
  int numIterations = (argc>2)? atoi(argv[2]) : T;
  X1[0] = X2[0] = X1[size+1] = X2[size+1] = 100;
  for (t = 0; t < numIterations; t++) {
     if ((t % 2) == 0) {
        for (i = 1; i < size+1; i++)
           X2[i] = (X1[i-1]+2*X1[i]+X1[i+1]) / 4;
     } else {
        for (i = 1; i < size+1; i++)
           X1[i] = (X2[i-1]+2*X2[i]+X2[i+1]) / 4;
     }
  }
```

```c
/* print the vector */
  printf("[ ");
  if (numIterations %2 == 0)
    for (i = 0; i < size+2; i++)
      printf(" %f", X1[i]);
  else
    for (i = 0; i < size+2; i++)
      printf(" %f", X2[i]);
    printf(" ]\n");
}
```

Same example using OpenMP

```c
#include <stdlib.h>
#include <stdio.h>
#define T 10
#define N 100
double X1[N+2], X2[N+2]; // initially zeros
int main(int argc, char *argv[]) {
  int t, i;
  int size = (argc > 1)? atoi(argv[1]) : N;
  if (size > N) size = N;
  int numIterations = (argc>2)? atoi(argv[2]) : T;
  X1[0] = X2[0] = X1[size+1] = X2[size+1] = 100;
#pragma omp parallel private(t,i) num_threads(4)
  for (t = 0; t < numIterations; t++) {
    if ((t % 2) == 0) {
#pragma omp for
      for (i = 1; i < size+1; i++)
        X2[i] = (X1[i-1] + 2*X1[i] + X1[i+1]) / 4;
    } else {
#pragma omp for
    for (i = 1; i < size+1; i++)
      X1[i] = (X2[i-1] + 2*X2[i] + X2[i+1]) / 4;
    }
  }
```

```c
  /* print the vector */
  printf("[ ");
  if (numIterations %2 == 0)
    for (i = 0; i < size+2; i++)
      printf(" %f", X1[i]);
    else
      for (i = 0; i < size+2; i++)
        printf(" %f", X2[i]);
  printf(" ]\n");
}
```

ROYAL INSTITUTE
OF TECHNOLOGY

Same example in ”MPI”

```
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &id);

int mystart = id * N / p + 1;
int myend = (id + 1) * N / p + 1;

for (t = 0; t < T; t++) {
  if ((t % 2) == 0) {
    for (i = mystart; i < myend; i++)
      X2[i] = (X1[i-1]+2*X1[i]+X1[i+1])/4;
    if (id != (p-1)) {
      send(X2[myend-1], id+1, sizeof(double));
      receive(X2[mystart-1], id-1, sizeof(double));
      /* blocking send and receive */
}
```

```
    if (id != 0) {
      send(X2[mystart], id-1, sizeof(double));
      receive(X2[myend], id+1, sizeof(double));
      /* blocking send and receive */
    }
  } else {
    for (i = mystart; i < myend; i++)
      X1[i] = (X2[i-1]+2*X2[i]+X2[i+1])/4;
    if (id != (p-1)) {
      send(X1[myend-1], id+1, sizeof(double));
      receives(X1[mystart-1], id-1, sizeof(double));
      /* blocking send and receive */
    }
    if (id != 0) {
      send(X1[mystart], id-1, sizeof(double));
      receive(X1[myend], id+1, sizeof(double));
      /* blocking send and receive */
    }
  }
}
```

Sending a message

```
MPI_Send(&N, 1, MPI_INT, i, tag,
         MPI_COMM_WORLD);
```

- *&N* – address to send buffer
- *1* – the number of elements to send
- *MPI_INT* – the datatype of message
- *i* – the receiving process
- *tag* – used to label this message
- *MPI_COMM_WORLD* – Communicator

- Buffered and non-blocking

Receiving a message

```
MPI_Recv(&tmp, 1, MPI_INT, i,
                    tag, MPI_COMM_WORLD, &status);
```

- *&tmp* – address to receive buffer
- *1* – the *maximum* number of elements to receive
- *MPI_INT* – the datatype of message
- *i* – the sending process
- *tag* – used to label this message. This must match the corresponding tag in MPI_Send
- *status* – information about the message received

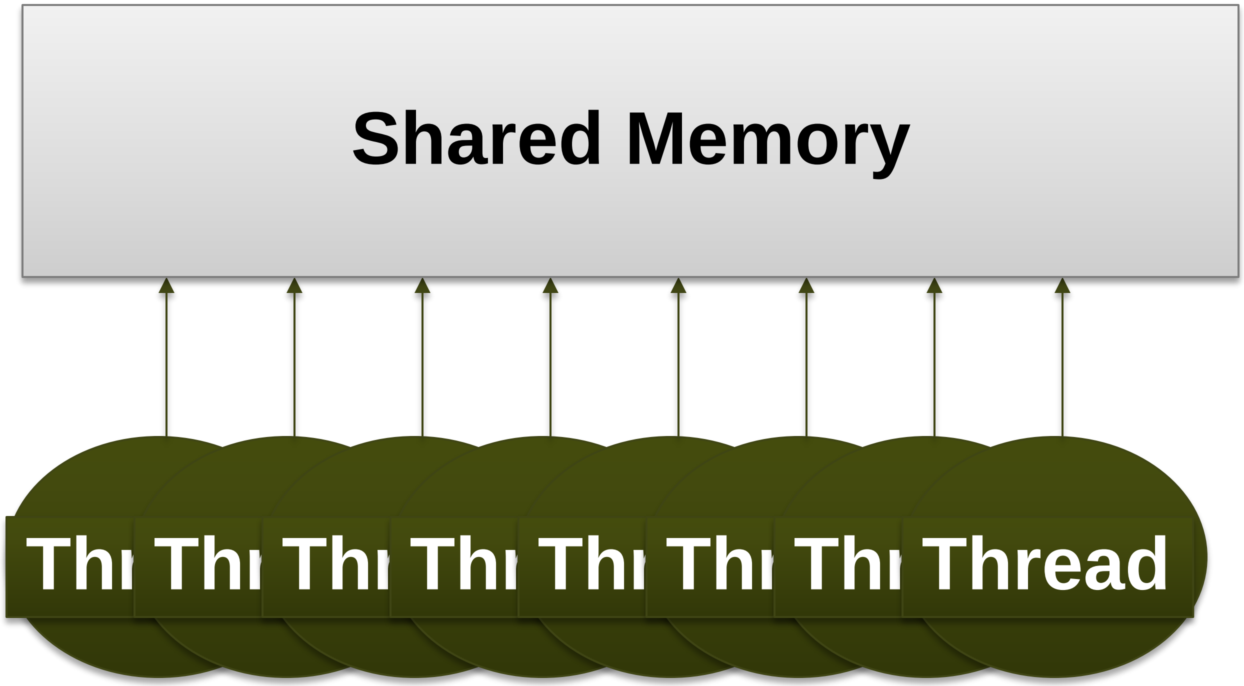
- Blocking

ROYAL INSTITUTE OF TECHNOLOGY

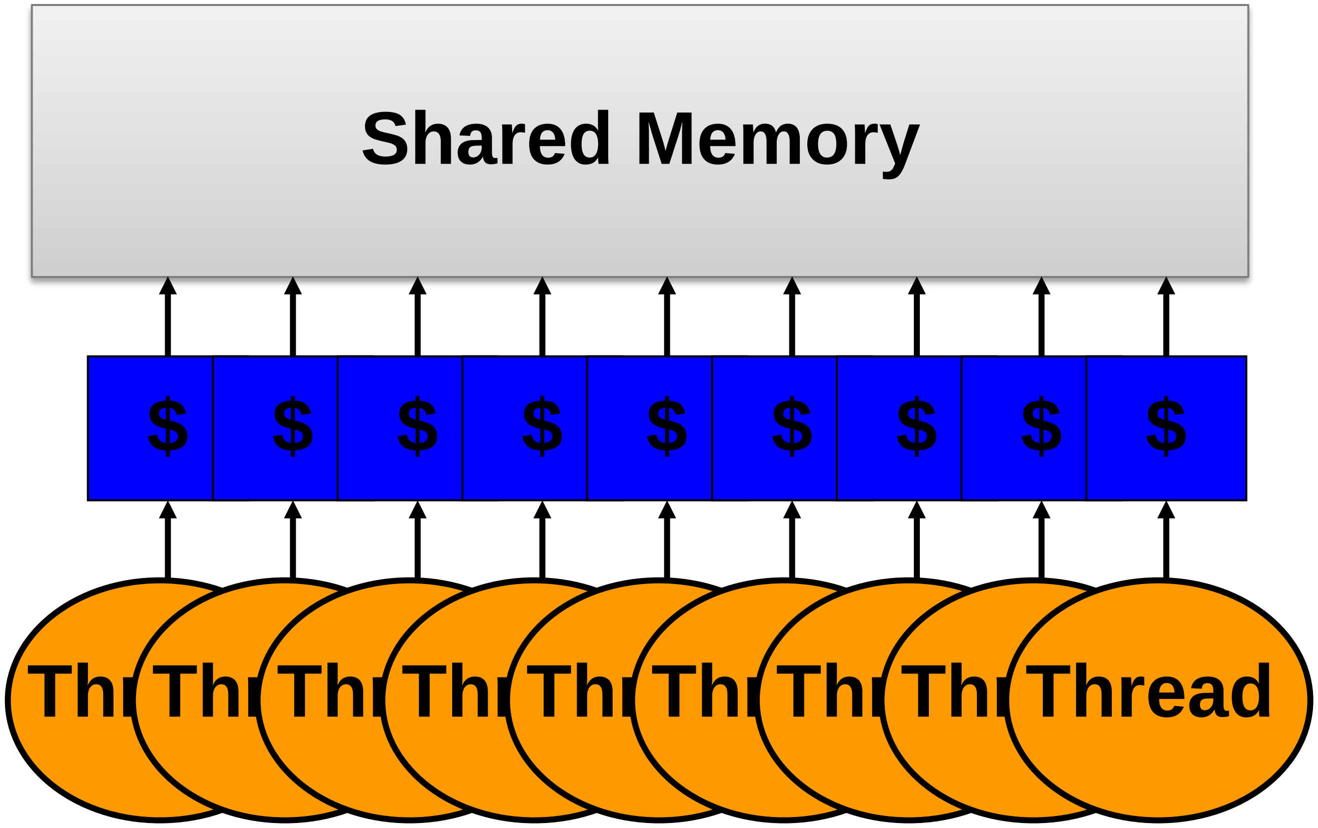
Multicore Architecture***

The main ideas and future trends

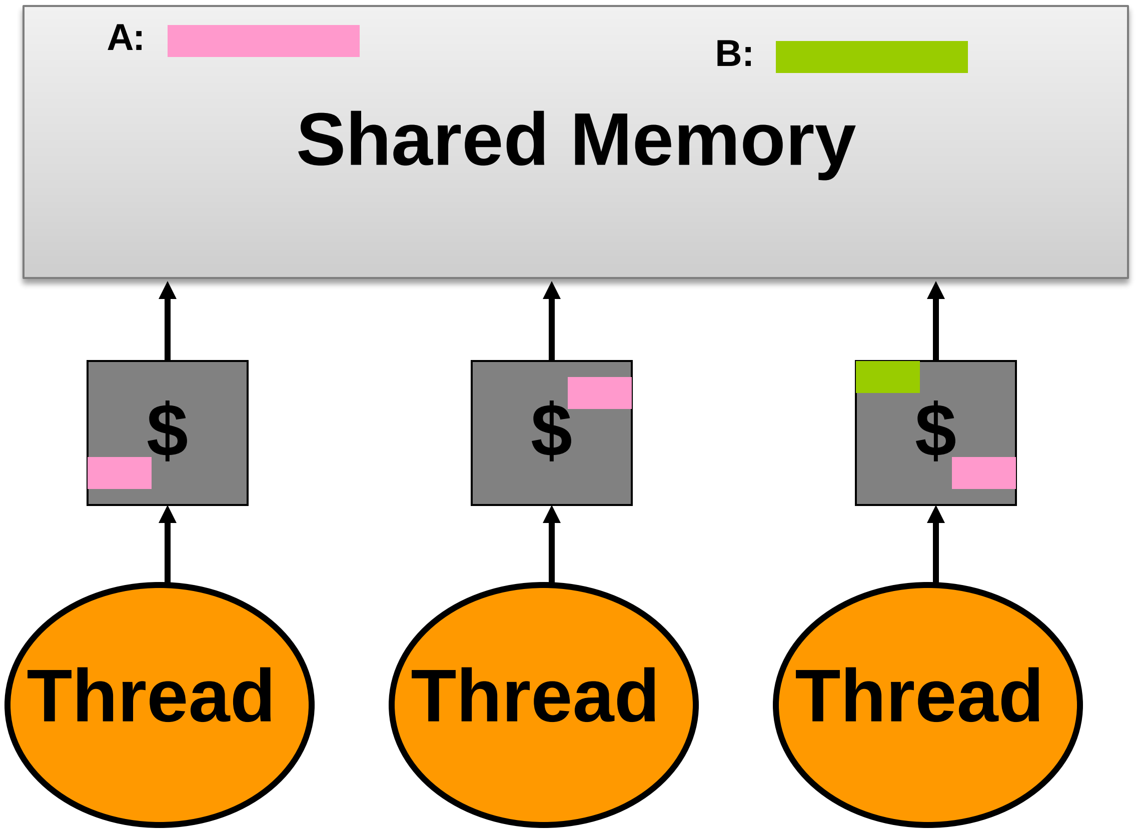
The Shared Memory Model



Adding Caches: More Concurrency

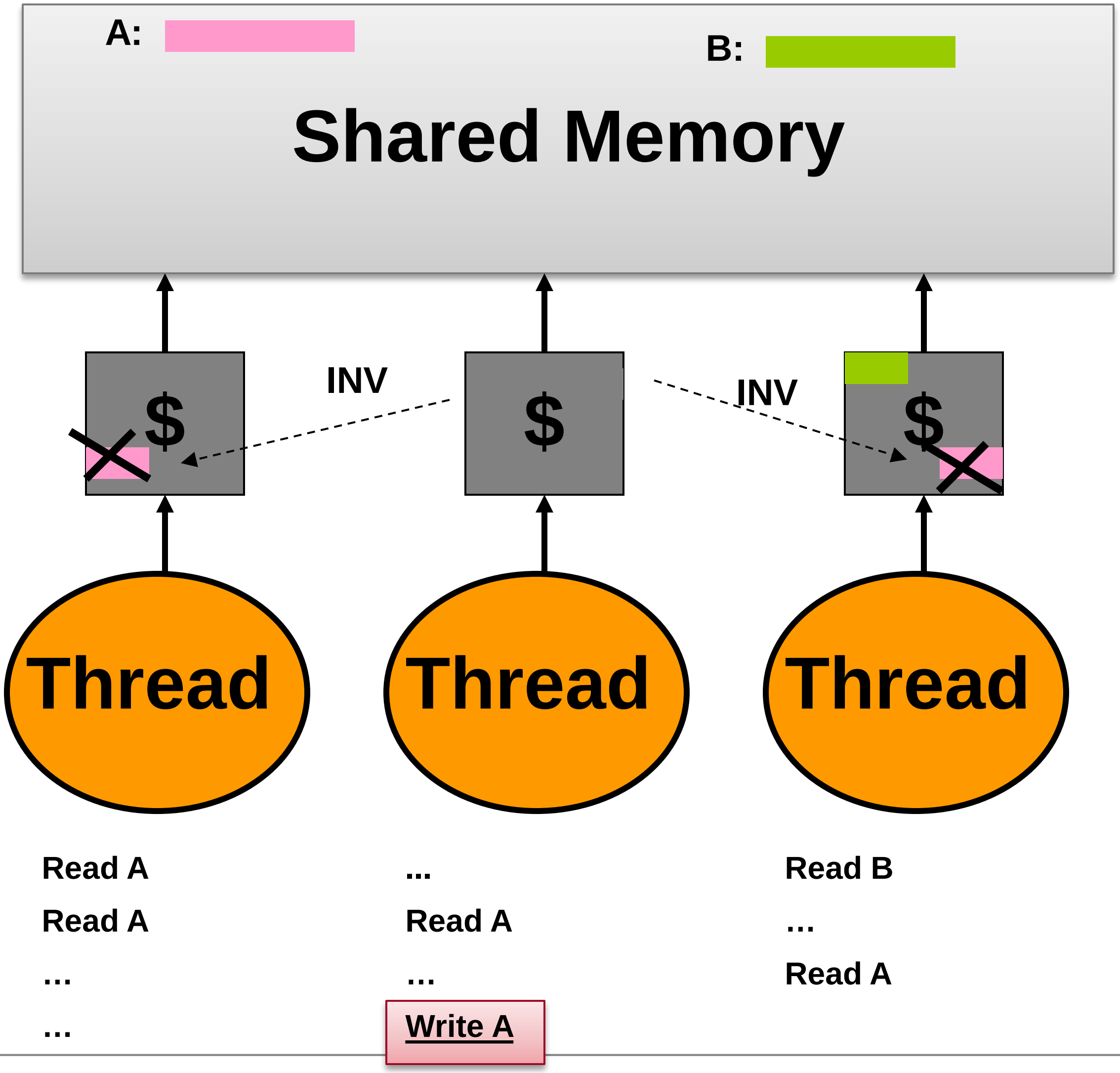


Automatic Replication of Data

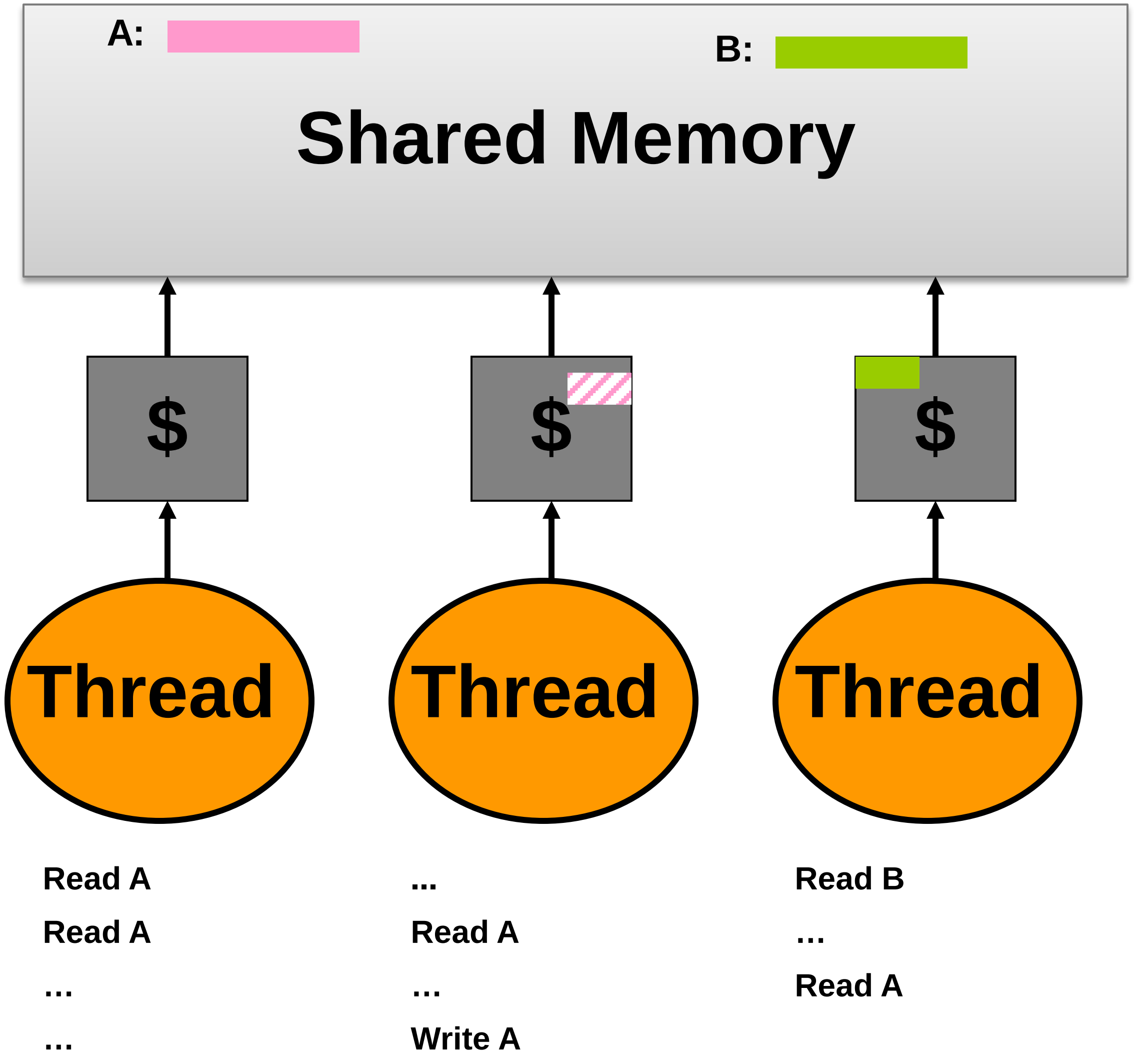


Read A	...	Read B
Read A	Read A	…
…	…	Read A
…		
Read A		

The Cache Coherent Memory System



The Cache Coherent Memory System

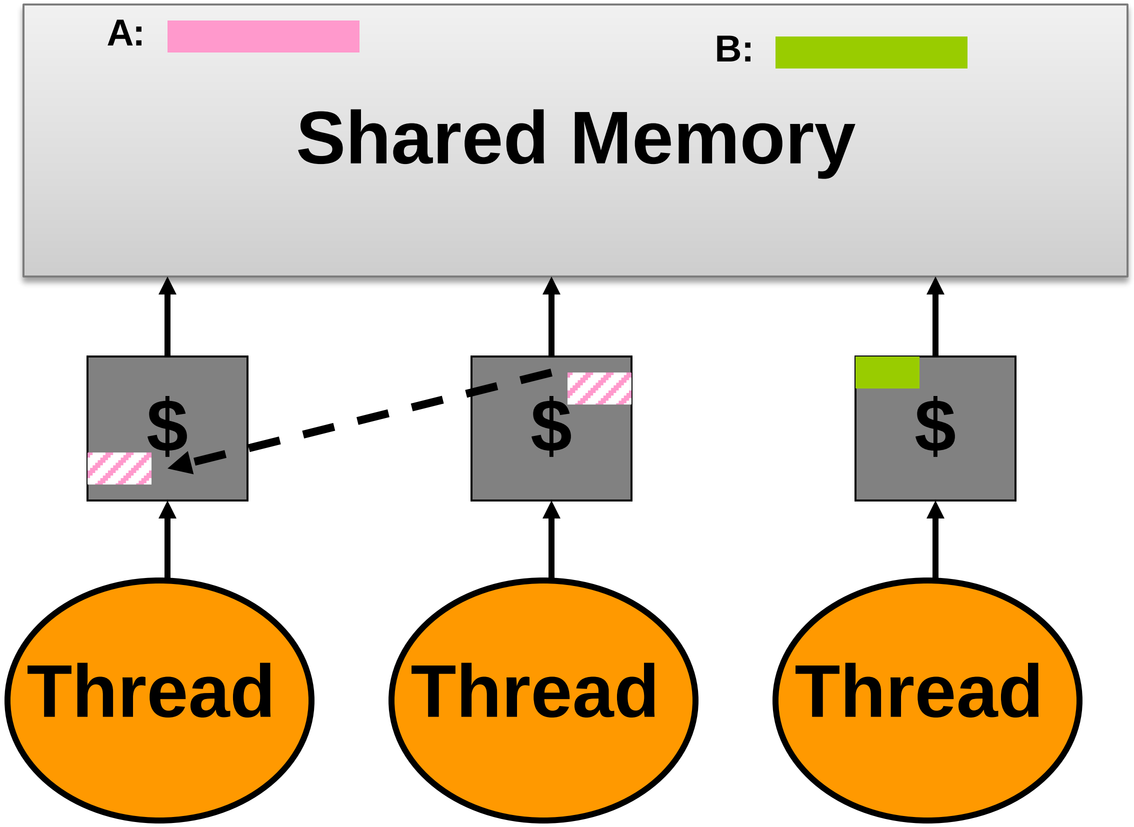


Read A
Read A
…
…

...
Read A
…
Write A

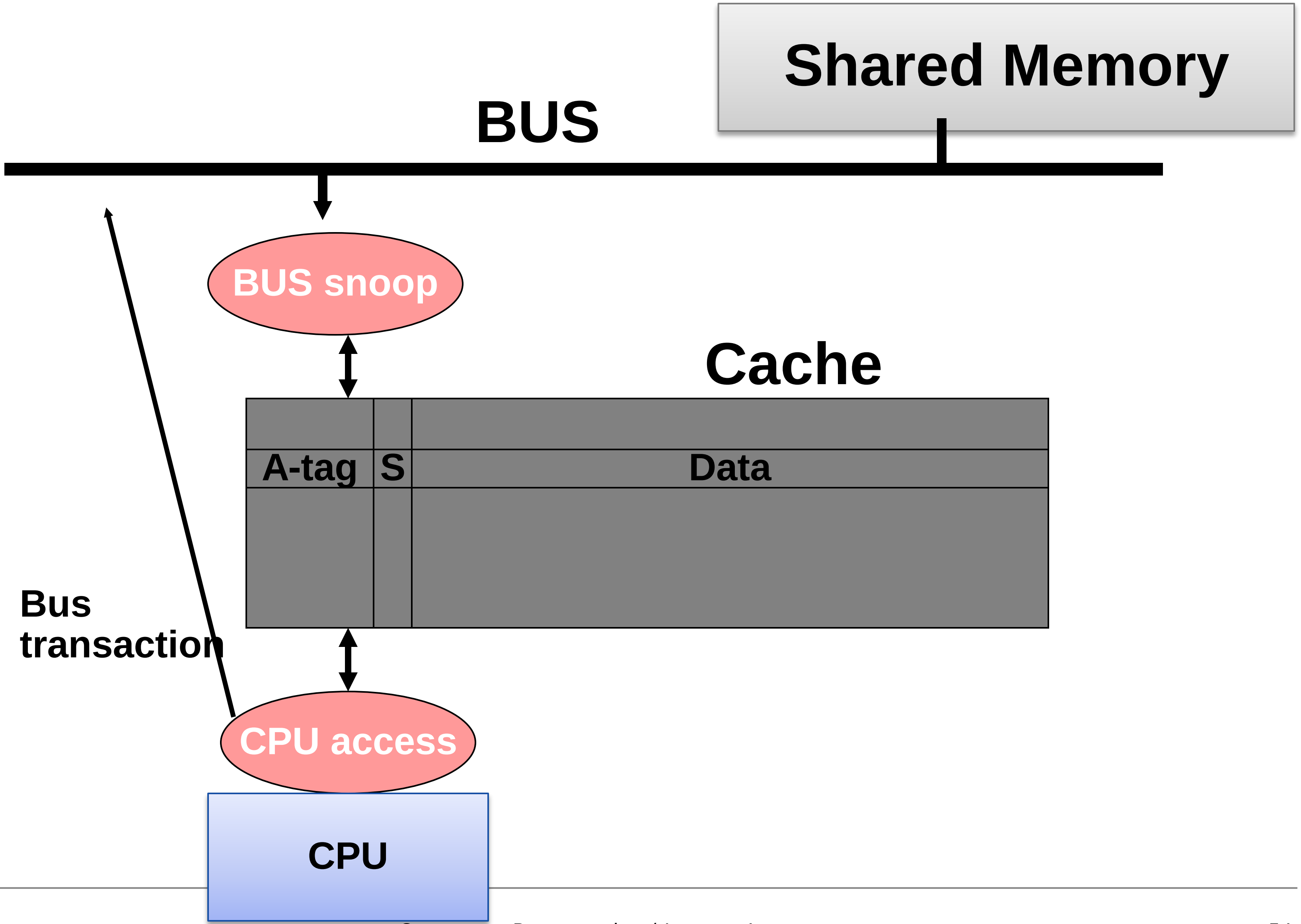
Read B
…
Read A

The Cache Coherent Memory System



Read A	...	Read B
Read A	Read A	…
…	…	Read A
…	Write A	
Read A		

Snoop-based Protocol Implementation



MOSI Protocol, 1(2)

Bus Transactions

BUSrts: ReadtoShare (reading the data with the intention to read it)

BUSrtw, ReadToWrite (reading the data with the intention to modify it)

BUSwb: Writing data back to memory

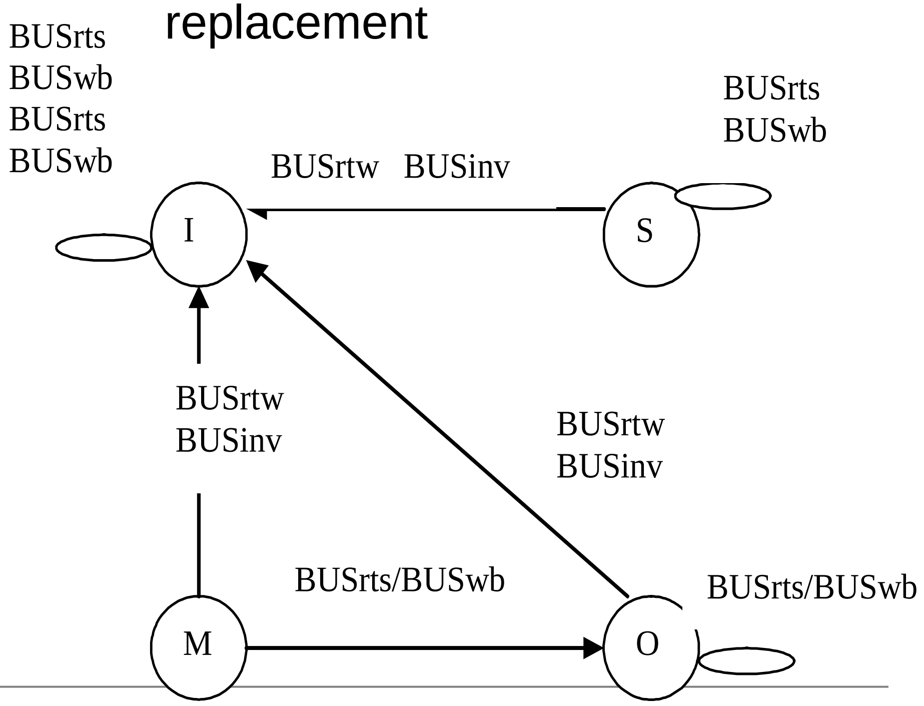
BUSinv: Invalidating other caches copie

CPU-Initiated Events

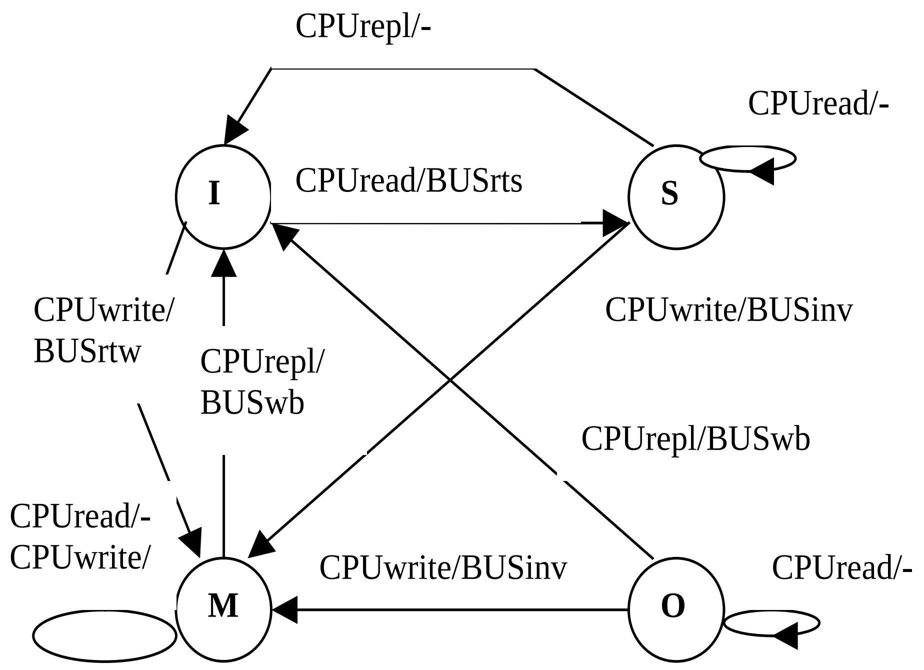
CPUwrite: Caused by a store miss

CPUread Caused by a loadmiss

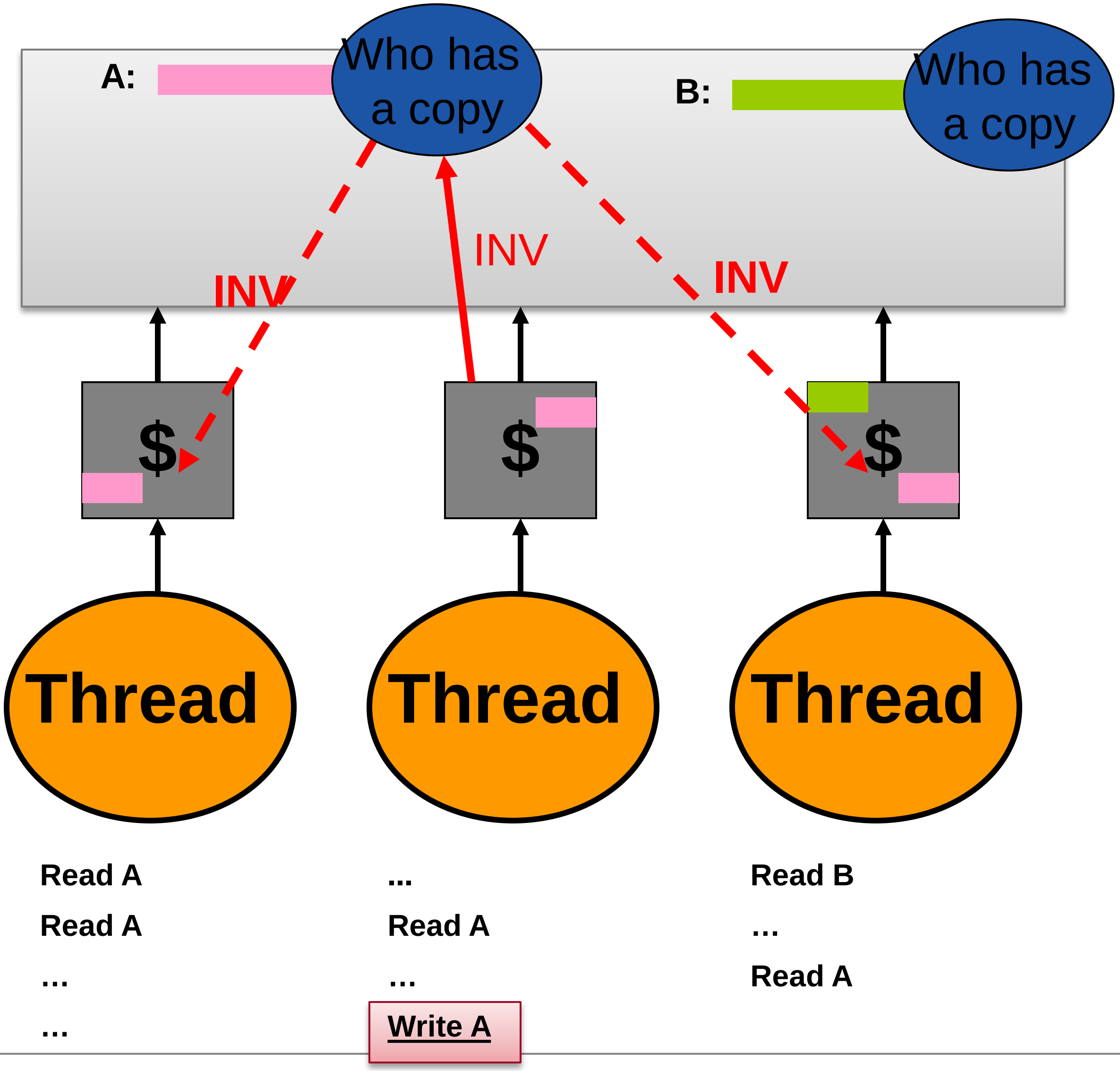
CPUrepl: Caused by a replacement



MOSI Protocol 2 (2)



Directory-based snooping



ROYAL INSTITUTE OF TECHNOLOGY

Why do you miss in a cache?

- Capacity misses – *the cache is too small*
- Conflict misses – *the cache organization is not perfect*
- Compulsory misses – *touching the data for the first time*
- Coherence misses – *caused by communication*

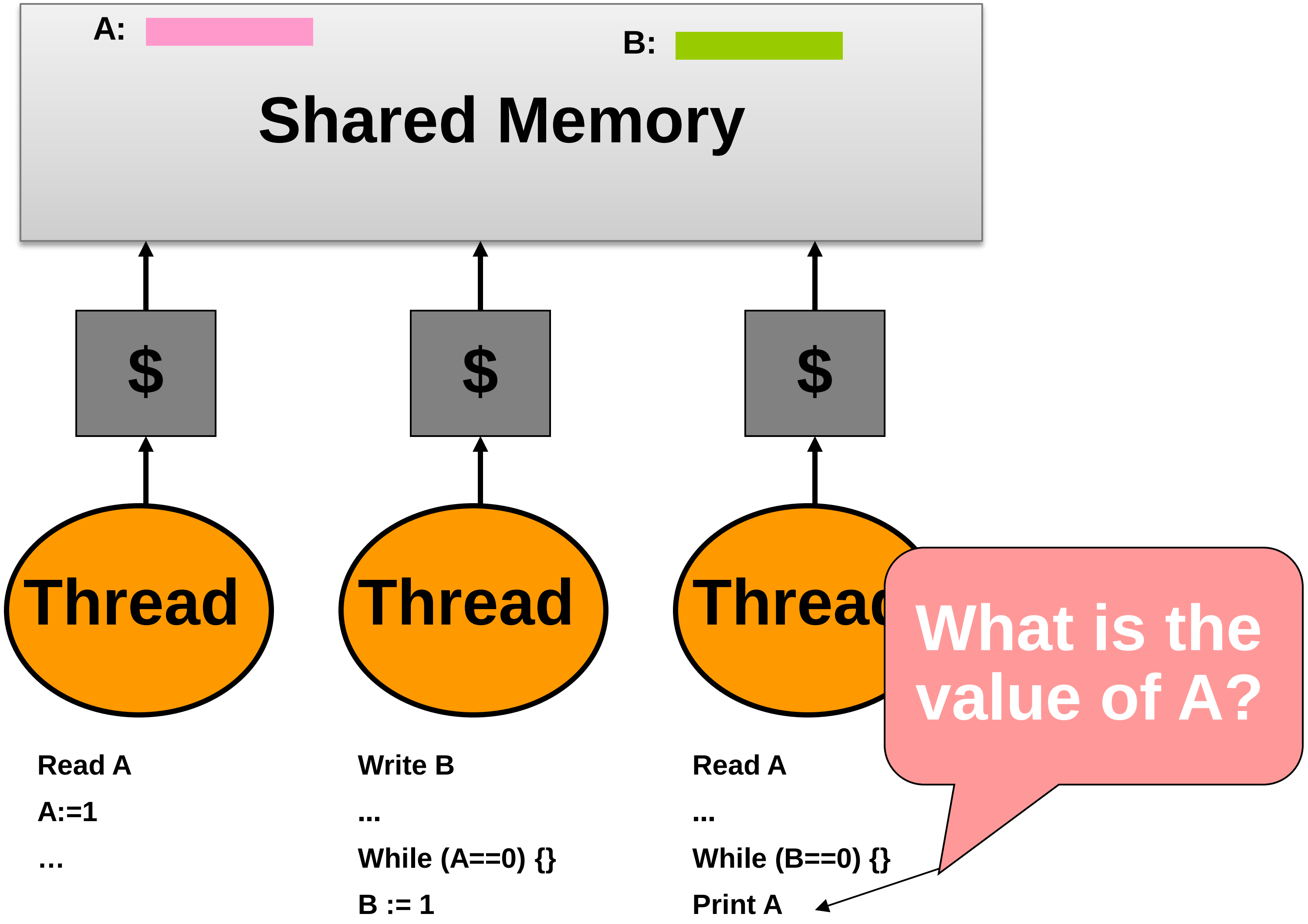
Avoiding cache misses?

- ***Capacity misses*** – the cache is too small
  - larger caches, selective cacheing, prefetch cache, ...
- ***Conflict misses*** – the cache organization is not perfect
  - more associativity, skewed caches, victim caches, ...
- ***Compulsory misses*** – touching the data for the first time
  - larger cache lines, SW prefetching, HW prefetching, ...
- ***Coherence misses*** – caused by communication
  - update-based coherence, migr. optimization,...

So Far…

- Coherent shared memory
- “Snoopy-based coherence protocol”
  - All global accesses are broadcasted to all caches
  - Cache lines are automatically invalidated/fetched
- Ensure ordering and serialization for a single cache line

Example1: Causal Correctness Issues

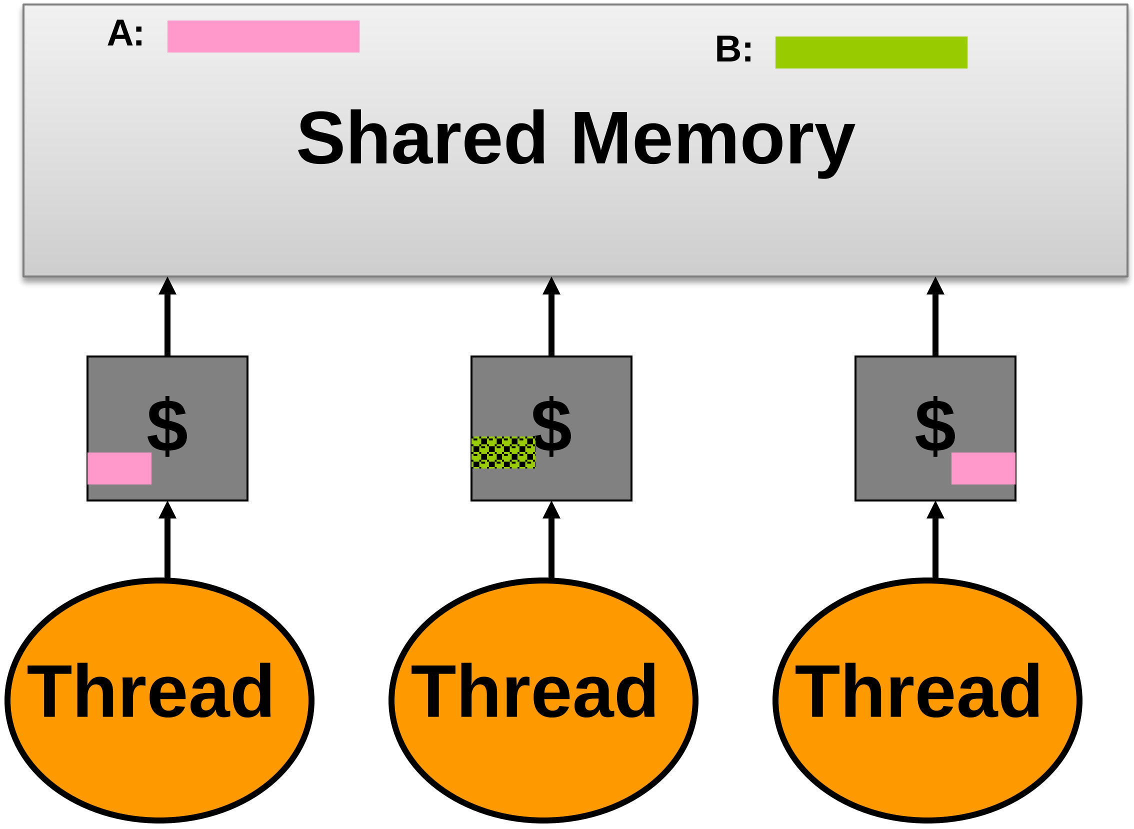


```
Read A
A:=1
…
```

```
Write B
...
While (A==0) {}
B := 1
```

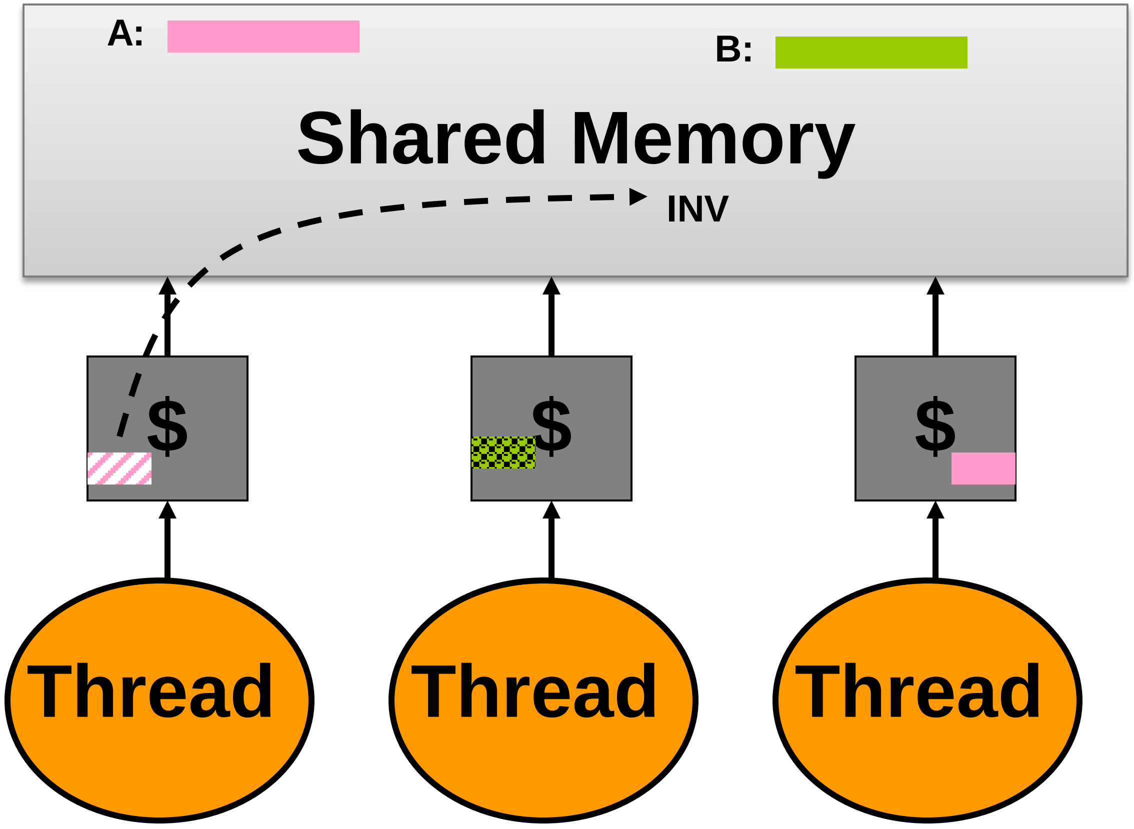
```
Read A
...
While (B==0) {}
Print A
```

Example1: Causal Correctness Issues



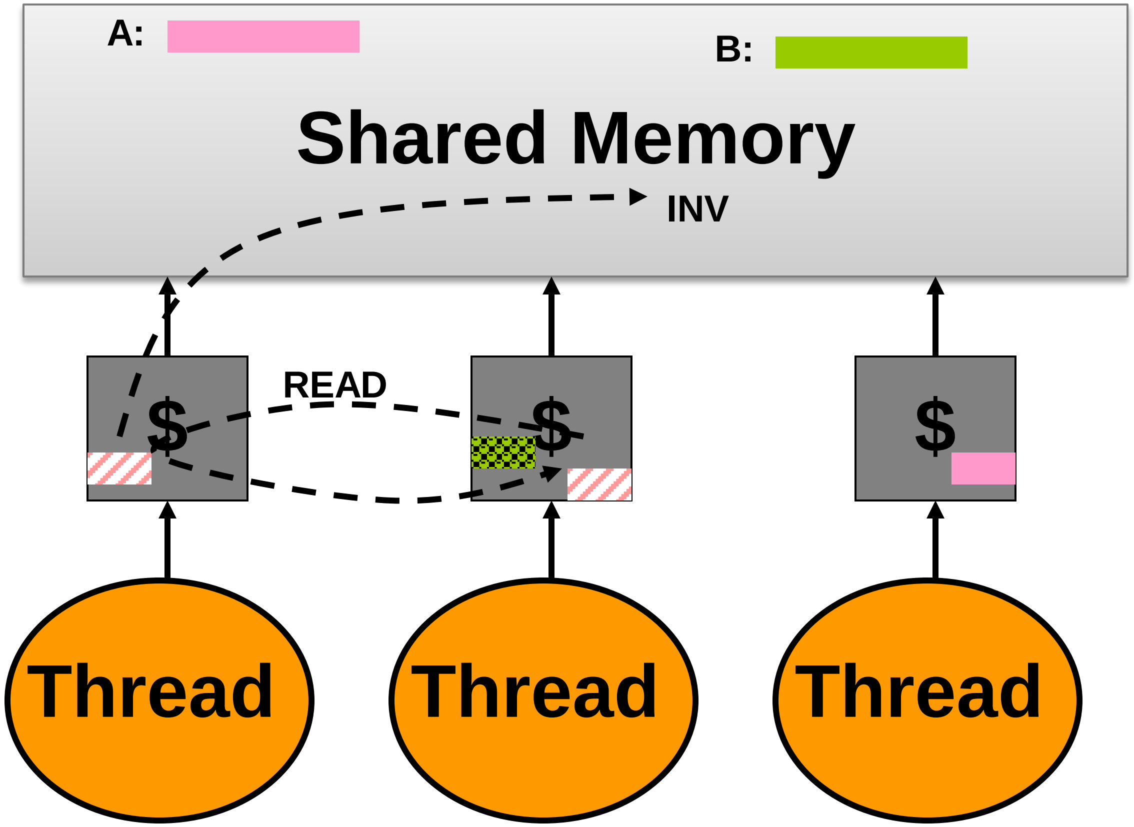
Read A	Write B	Read A
A:=1	…	…
…	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues



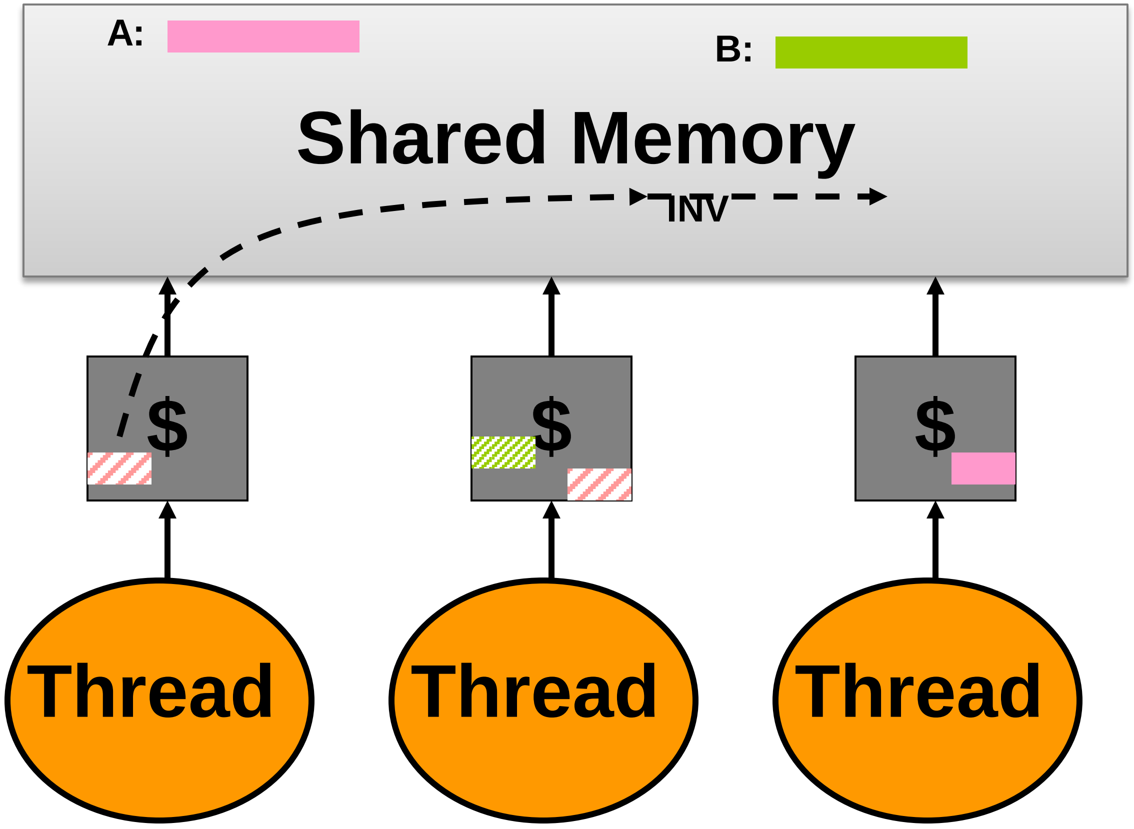
Thread 1	Thread 2	Thread 3
Read A	Write B	Read A
A:=1	…	…
…	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues



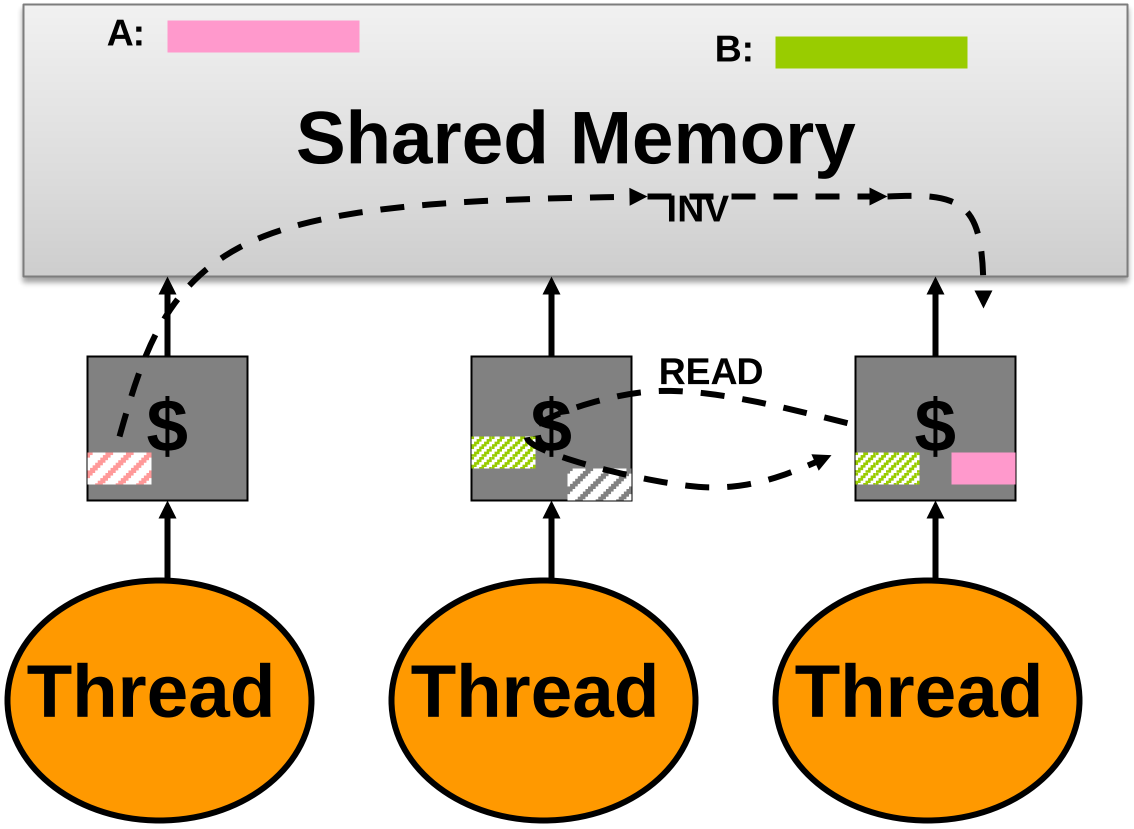
Read A	Write B	Read A
A:=1	…	…
…	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues



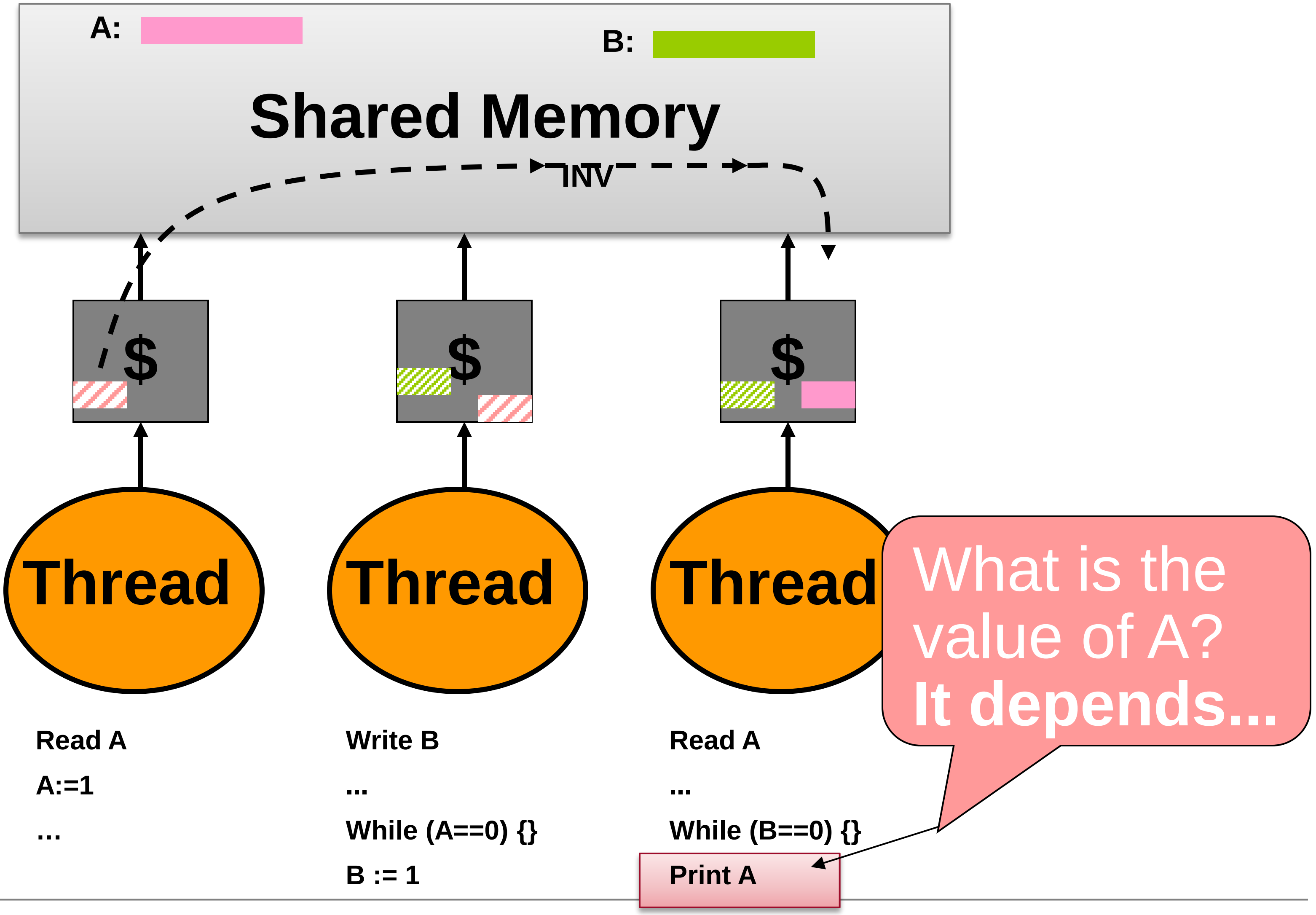
Read A	Write B	Read A
A:=1	...	...
…	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues



Read A	Write B	Read A
A:=1	...	...
…	While (A==0) {}	While (B==0) {}
	B := 1	Print A

Example1: Causal Correctness Issues



Read A	Write B	Read A
A:=1	...	...
…	While (A==0) {}	While (B==0) {}
	B := 1	Print A

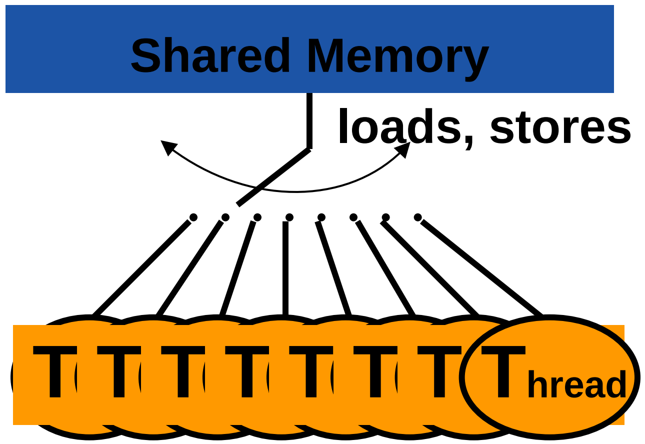
Memory [Consistency] Model

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
- Without it, can't tell much about a program's execution
- A contract between the programmer and the system designer
  - Programmer uses it to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware

Sequential Consistency

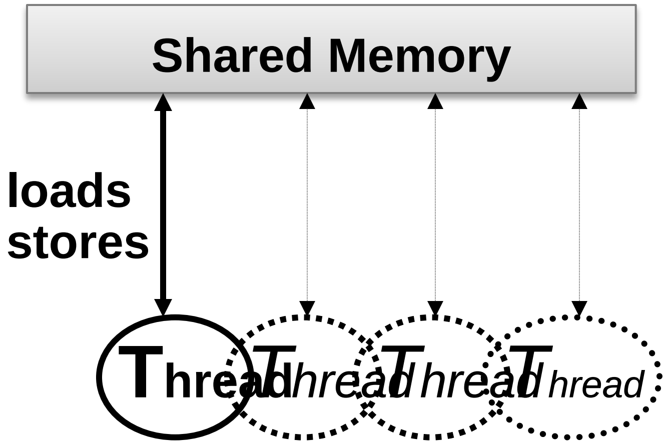
- “A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” ’
- [Lamport, 1979]

Sequential Consistency



- Global order achieved by *interleaving* <u>all</u> memory accesses from different processes
- "Programmer's intuition is maintained"
  - Store causality? Yes
  - Does Dekker work? Yes
- -- Unnecessarily restrictive ==> performance penalty

Weak/release Consistency



- Most accesses are unordered
- “Programmer’s intuition is not maintained”
  - Store causality? No
  - Does Dekker work? No
- Global order only established when the programmer explicitly inserts memory barrier instructions
- ++ Great performance!!
- --- Interesting bugs!!
- But programming models such as OpenMP can help!