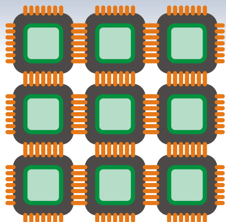
Multicore Application Optimization

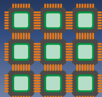
How not to shoot your own foot



Sven Stork

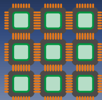
Carnegie Mellon University
&
University of Coimbra

IBERGRID, June 10, 2011



Outline

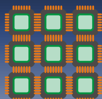
1. Overview
2. Performance Optimizations
3. Why are the current systems broken?
4. Future



Why Multi-Cores?

- Computer chips reached physical limitations (e.g., heat)
- Moore's Law is still correct (i.e., doubling the amount of transistors per areas unit)
- Instead of making chips faster just duplicate core.
- "The free lunch is over" (~2004, Herbert Sutter)

paradigm shift

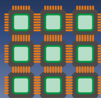


multi-core vs. many-core

We differentiate between **multi-core** and **many-core**:

multi-core
: A computing component with a moderate amount of cores (up to ∼16).

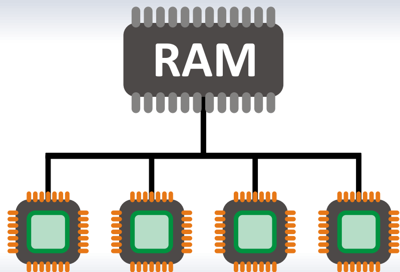
many-core
: Sometimes called **massively multi-core** describes systems with 10s and 100s of cores in a single computing component.

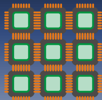


SMP vs NUMA

Symmetric Multi-Processing (SMP)

- all processors are directly connected to main memory
- all processors have the same access time to main memory
- limited scalability beyond a few cores

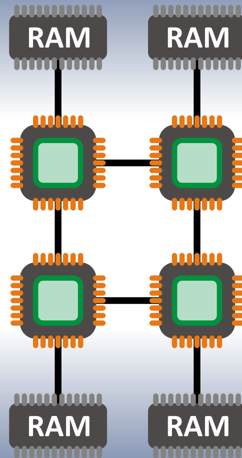


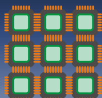


SMP vs NUMA

Non-Uniform Memory Access
NUMA

- "network" of processors
- every processor has local memory
- accessing memory of remote processors via message passing
- (more) scalable





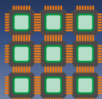
Programming Models

Message Passing

Communicating between concurrent entities is make via explicit message exchange. Examples of message passing are Actors and MPI.

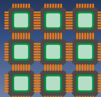
Shared Memory

In a shared memory system concurrent entities communicate by exchanging data via shared memory (implicit message passing).



Open MP Primer

- Examples in this Open MP for brevity
- **But showed concepts and techniques are generally applicable.**
- Open MP is a compiler extension for C,C++ and Fortran
- User **annotates** sequential program and compiler will **automatically parallelize** it

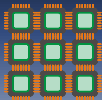


Open MP Primer

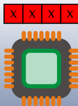
- Open MP uses **pragmas** to specify concurrency: #pragma omp
- Open MP support parallel **sections** and **for-loops**

```
#pragma omp parallel for
for ( int x = 0; x < N; x++ ) {
    ...
}
```

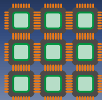
- Note: we omit extra clauses for readability and because the examples are small enough for the compiler to figure the correct defaults



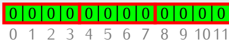
Cache Primer

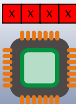


- A cache is a fast memory **between** CPU and RAM that exploit **locality in space**.
- A cache **buffers** values from RAM.
- Cache always loads/stores **fixed** N bytes **blocks** from/to memory (called a **cache line**)
- Multi-Core systems often use **cache coherency protocol** to keep caches and RAM synchronized.

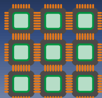


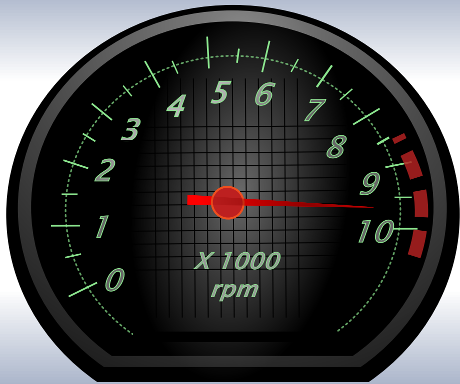
Cache Primer

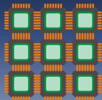




- A cache is a fast memory **between** CPU and RAM that exploit **locality in space**.
- A cache **buffers** values from RAM.
- Cache always loads/stores **fixed** N bytes **blocks** from/to memory (called a **cache line**)
- Multi-Core systems often use **cache coherency protocol** to keep caches and RAM synchronized.

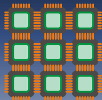



0
1
2
3
4
5
6
7
8
9
10
X 1000
rpm



Premature optimization is the root of all evil!

(Donald Knuth)

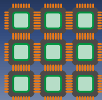


Basic Rules

- First get it **right** then get it fast.
- Make sure you understand **Amdahl's Law**

$$\textbf{Speedup} = \frac{1}{(1-P)+\frac{P}{S}}$$

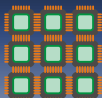
- **Benchmark** and **profile** your application.
- **Test** your application regularly to **avoid regressions**.



Common Mistakes

Optimizing by avoid common mistakes:

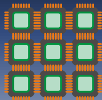
- Cache Optimizations (sequential programming)
- Load Imbalance
- Over Synchronization
- False Sharing



Cache Optimization

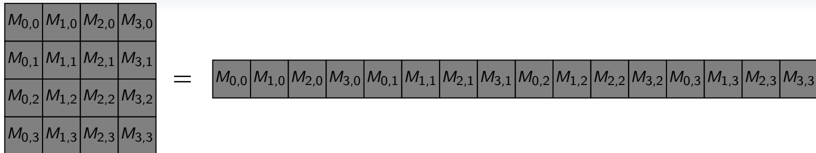
Example: Find Minimum Entry in Matrix

```
int findMin(int *matrix, int dimX, int dimY) {
    int x,y,min = INT_MAX;

    for (x = 0; x < dimX; ++x) {
        for (y = 0; y < dimY; ++y) {
            if ( matrix[y*dimX + x] < min ) {
                min = matrix[y*dimX + x];
            }
        }
    }

    return min;
}
```



Cache Optimization

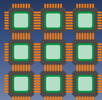
- matrix is stored in **contiguous** memory block (in C,C++, etc)



- Layout depends in programming language

row-major: rows first (e.g., C, C++)

column-major: columns first (e.g., Fortran, Matlab)



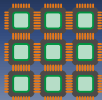
Cache Optimization

RAM

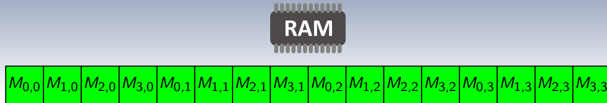
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

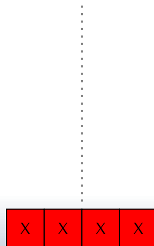
X	X	X	X

matrix[x,y]



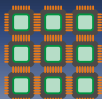
Cache Optimization



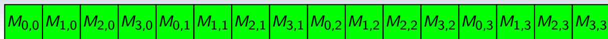


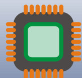
matrix[x,y]

```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```



Cache Optimization

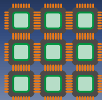




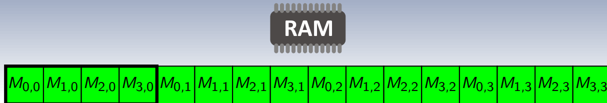
matrix[x,y]

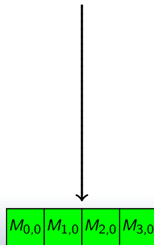
```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

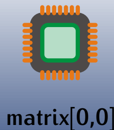
```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization



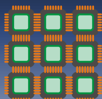




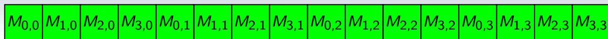
matrix[0,0]

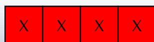
```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

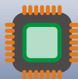
```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization



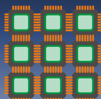




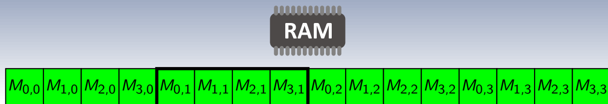
matrix[0][1]

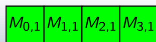
```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

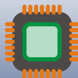
```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization



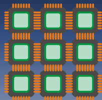




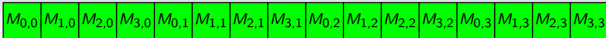
matrix[0,1]

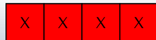
```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

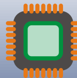
```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization



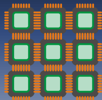




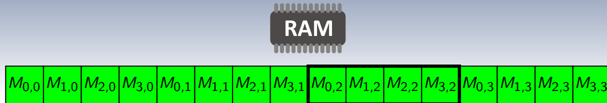
matrix[0,2]

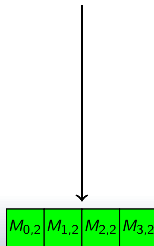
```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

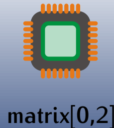
```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization





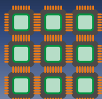


matrix[0,2]

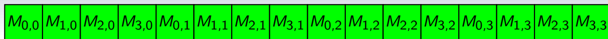
```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

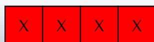
$\Downarrow$

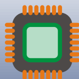
```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization



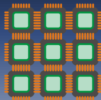




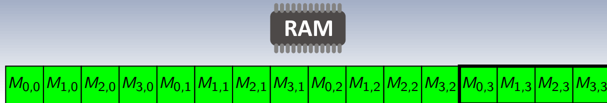
matrix[0,3]

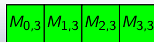
```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

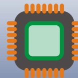
```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization



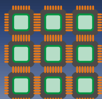




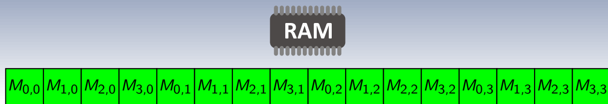
matrix[0,3]

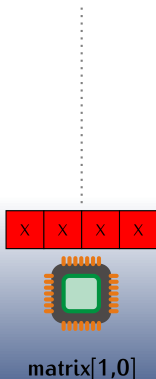
```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization

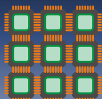




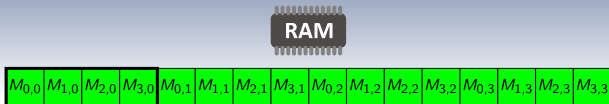

```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

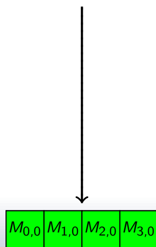
⇓

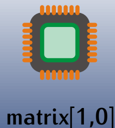
```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization



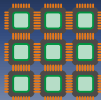




matrix[1,0]

```
for (x = 0; x < dimX; ++x) {
    for (y = 0; y < dimY; ++y) {...
    }
}
```

```
matrix[0,0]
matrix[0,1]
matrix[0,2]
matrix[0,3]
matrix[1,0]
```



Cache Optimization

RAM

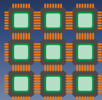
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$

matrix[1,0]

Problem

- loading repeatedly same cache line from memory
- use only a single element out of N elements of cache-line
- causes lots of unnecessary memory traffic



Cache Optimization

RAM

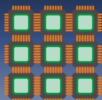
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$

matrix[1,0]

Solution

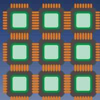
- Better usage of cached data.
- Load data only once (if possible)



Cache Optimization

Example: Find Minimum Entry in Matrix

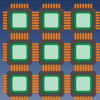
```
int findMin(int *matrix, int dimX, int dimY) {
    int x,y,min = INT_MIN;

    for (x = 0; x < dimX; ++x) {
        for (y = 0; y < dimY; ++y) {
            if ( matrix[y*dimX + x] < min ) {
                min = matrix[y*dimX + x];
            }
        }
    }

    return min;
}
```



Cache Optimization

Example: Find Minimum Entry in Matrix

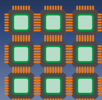
```
int findMin(int *matrix, int dimX, int dimY) {
    int x,y,min = INT_MIN;

    for (x = 0; x < dimX; ++x) {

            if ( matrix[y*dimX + x] < min ) {
                min = matrix[y*dimX + x];
            }
        }
    }

    return min;
}
```



Cache Optimization

Example: Find Minimum Entry in Matrix

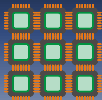
```
int findMin(int *matrix, int dimX, int dimY) {
    int x,y,min = INT_MIN;


        for (x = 0; x < dimX; ++x) {
            if ( matrix[y*dimX + x] < min ) {
                min = matrix[y*dimX + x];
            }
        }
    }

    return min;
}
```



Cache Optimization

Example: Find Minimum Entry in Matrix

```
int findMin(int *matrix, int dimX, int dimY) {
    int x,y,min = INT_MIN;

    for (y = 0; y < dimY; ++y) {
        for (x = 0; x < dimX; ++x) {
            if ( matrix[y*dimX + x] < min ) {
                min = matrix[y*dimX + x];
            }
        }
    }

    return min;
}
```



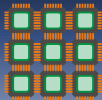
Cache Optimization

RAM

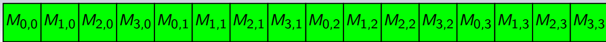
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

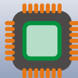
X	X	X	X

matrix[x,y]



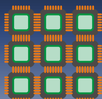
Cache Optimization



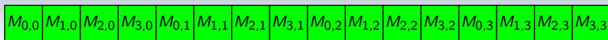


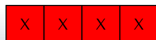
matrix[x,y]

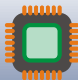
```
for (y = 0; y < dimY; ++y) {
    for (x = 0; x < dimX; ++x) {...
    }
}
```



Cache Optimization



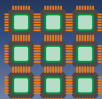




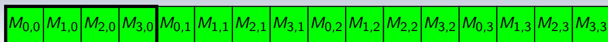
matrix[x,y]

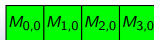
```
for (y = 0; y < dimY; ++y) {
    for (x = 0; x < dimX; ++x) {...
    }
}
```

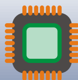
```
matrix[0,0]
matrix[1,0]
matrix[2,0]
matrix[3.0]
matrix[0,1]
```



Cache Optimization



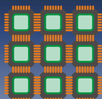




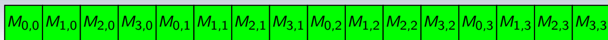
matrix[0,0]

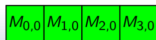
```
for (y = 0; y < dimY; ++y) {
    for (x = 0; x < dimX; ++x) {...
    }
}
```

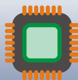
```
matrix[0,0]
matrix[1,0]
matrix[2,0]
matrix[3.0]
matrix[0,1]
```



Cache Optimization



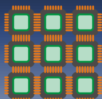




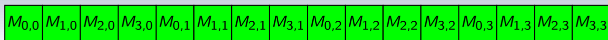
matrix[1,0]

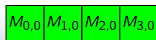
```
for (y = 0; y < dimY; ++y) {
    for (x = 0; x < dimX; ++x) {...
    }
}
```

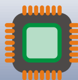
```
matrix[0,0]
matrix[1,0]
matrix[2,0]
matrix[3.0]
matrix[0,1]
```



Cache Optimization



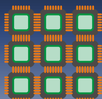




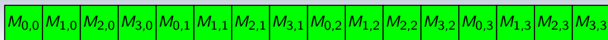
matrix[2,0]

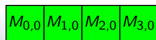
```
for (y = 0; y < dimY; ++y) {
    for (x = 0; x < dimX; ++x) {...
    }
}
```

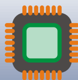
```
matrix[0,0]
matrix[1,0]
matrix[2,0]
matrix[3.0]
matrix[0,1]
```



Cache Optimization



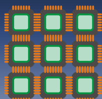




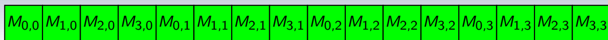
matrix[3,0]

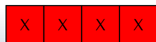
```
for (y = 0; y < dimY; ++y) {
    for (x = 0; x < dimX; ++x) {...
    }
}
```

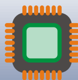
```
matrix[0,0]
matrix[1,0]
matrix[2,0]
matrix[3.0]
matrix[0,1]
```



Cache Optimization



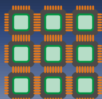




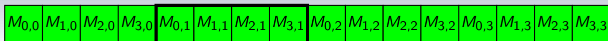
matrix[0,1]

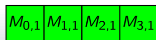
```
for (y = 0; y < dimY; ++y) {
    for (x = 0; x < dimX; ++x) {...
    }
}
```

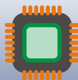
```
matrix[0,0]
matrix[1,0]
matrix[2,0]
matrix[3.0]
matrix[0,1]
```



Cache Optimization



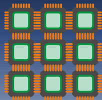




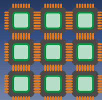
matrix[0,1]

```
for (y = 0; y < dimY; ++y) {
    for (x = 0; x < dimX; ++x) {...
    }
}
```

```
matrix[0,0]
matrix[1,0]
matrix[2,0]
matrix[3.0]
matrix[0,1]
```

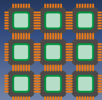


Demo



Cache Optimization

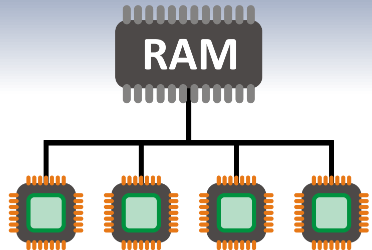
- Program is faster even in the sequential case!



Cache Optimization

- Program is faster even in the sequential case!
- Import to reduce because memory bandwidth:

SMP Single memory bus (bottleneck)

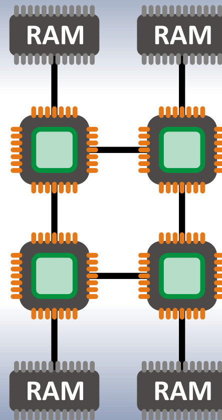


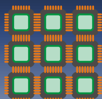
Cache Optimization

- Program is faster even in the sequential case!
- Import to reduce because memory bandwidth:

SMP
: Single memory bus (bottleneck)

NUMA
: Transfer from remote “memory” has higher latency.

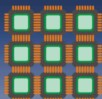




Load Imbalance

Example: Fibonacci

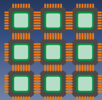
```
int main(int argc, char *argv[])
{
    const int COUNT = 8;
    int i = 0;


    for ( i = 0; i < COUNT; i++ ) {
        if ( i < COUNT/2 ) {
            printf("fib = %i\n", fib(30));
        } else {
            printf("fib = %i\n", fib(42));
        }
    }
    return 0;
}
```



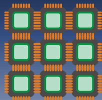
Load Imbalance

Example: Fibonacci

```
int main(int argc, char *argv[])
{
    const int COUNT = 8;
    int i = 0;

    #pragma omp parallel for
    for ( i = 0; i < COUNT; i++ ) {
        if ( i < COUNT/2 ) {
            printf("fib = %i\n", fib(30));
        } else {
            printf("fib = %i\n", fib(42));
        }
    }
    return 0;
}
```

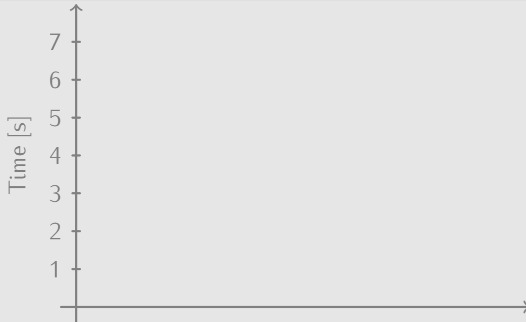


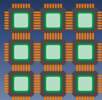
Demo



Load Imbalance

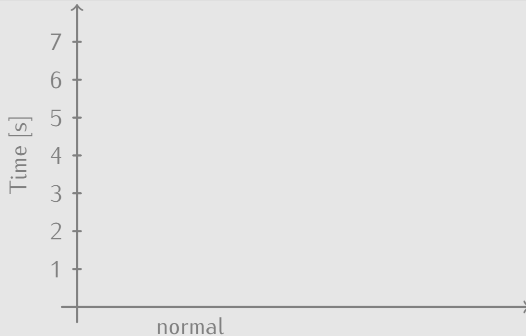
Performance (2-Cores)

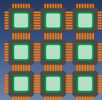




Load Imbalance

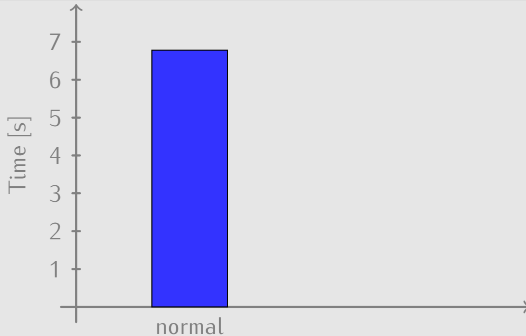
Performance (2-Cores)

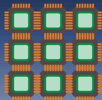




Load Imbalance

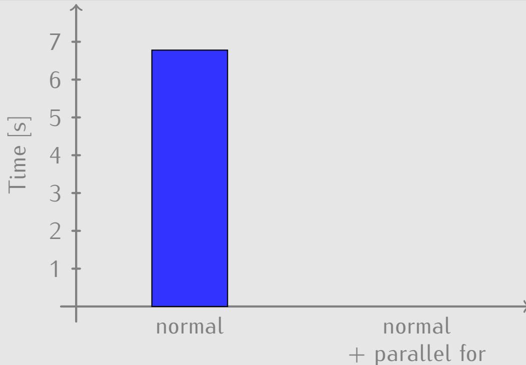
Performance (2-Cores)

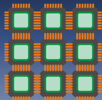




Load Imbalance

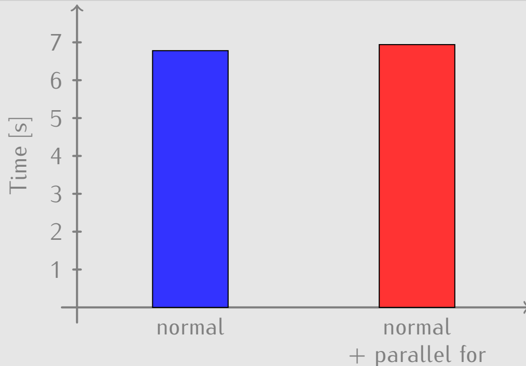
Performance (2-Cores)

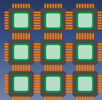




Load Imbalance

Performance (2-Cores)

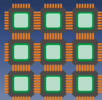




Load Imbalance

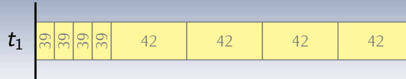
Why does the parallel version is not getting faster?

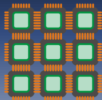
- Body computes first computes $COUNT/2$ times $fib(30)$ then compute $COUNT/2$ times $fib(42)$
- Fibonacci computation is **exponential** complex $\therefore$ $fib(30)$ **computes faster** than $fib(42)$
- Open MP uses by default **static scheduler** i.e.,
    - 1st thread computes $0 \rightarrow (\#I/\#T) - 1$
    - 2nd thread computes $(\#I/\#T) \rightarrow 2 \times (\#I/\#T) - 1$
    - ...



Load Imbalance

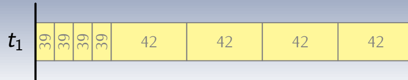
normal



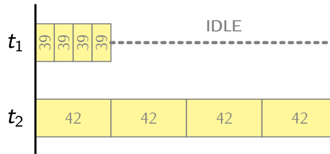


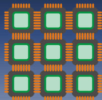
Load Imbalance

normal



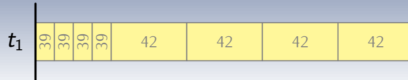
normal
+ parallel for



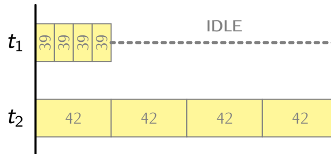


Load Imbalance

normal

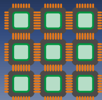


normal
+ parallel for



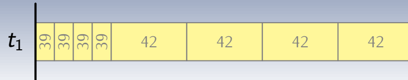
Need to balance both threads

1. Make each iteration the same duration.
2. Distribute work more evenly.

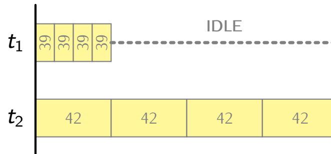


Load Imbalance

normal

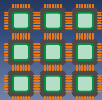


normal
+ parallel for



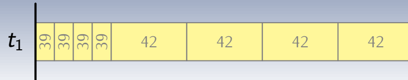
Need to balance both threads

1. Make each iteration the same duration.
2. Distribute work more evenly.

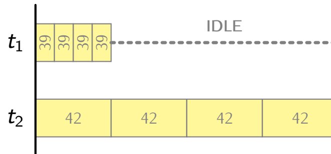


Load Imbalance

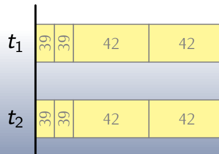
normal



normal
+ parallel for



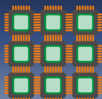
normal
+ parallel for
+ dynamic



Load Imbalance

Example: Fibonacci

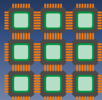
```
int main(int argc, char *argv[])
{
    const int COUNT = 8;
    int i = 0;

    #pragma omp parallel for
    for ( i = 0; i < COUNT; i++ ) {
        if ( i < COUNT/2 ) {
            printf("fib = %i\n", fib(30));
        } else {
            printf("fib = %i\n", fib(42));
        }
    }
    return 0;
}
```



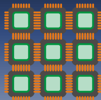
Load Imbalance

Example: Fibonacci [FIXED]

```
int main(int argc, char *argv[])
{
    const int COUNT = 8;
    int i = 0;

    #pragma omp parallel for schedule(dynamic)
    for ( i = 0; i < COUNT; i++ ) {
        if ( i < COUNT/2 ) {
            printf("fib = %i\n", fib(30));
        } else {
            printf("fib = %i\n", fib(42));
        }
    }
    return 0;
}
```

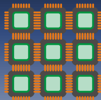


Demo



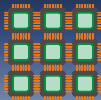
Load Imbalance

Performance (Dual-Core)



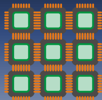
Load Imbalance

Performance (Dual-Core)



Load Imbalance

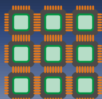
Performance (Dual-Core)



Over Synchronization

Example: Counting

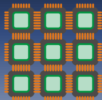
```
void counting() {
    int cur = 0;
    int counter = 0;


    for (cur = 0; cur < COUNT; cur++ ) {

        counter++;
    }
    printf(" %i \n", counter);
}
```



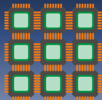
Over Synchronization

Example: Counting

```
void counting() {
    int cur = 0;
    int counter = 0;

    #pragma omp parallel for
    for (cur = 0; cur < COUNT; cur++ ) {

        counter++;
    }
    printf(" %i \n", counter);
}
```



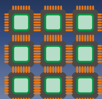
Demo



Over Synchronization

Example: Counting

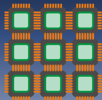
```
void counting() {
    int cur = 0;
    int counter = 0;

    #pragma omp parallel for
    for (cur = 0; cur < COUNT; cur++ ) {

        counter++;
    }
    printf(" %i \n", counter);
}
```



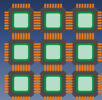
Over Synchronization

Example: Counting

```
void counting() {
    int cur = 0;
    int counter = 0;

    #pragma omp parallel for
    for (cur = 0; cur < COUNT; cur++ ) {
        #pragma omp atomic
        counter++;
    }
    printf(" %i \n", counter);
}
```

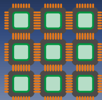


Demo



Over Synchronization

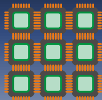
Performance (2-Cores)



Over Synchronization

Problem

- Use too much synchronization.



Over Synchronization

Problem

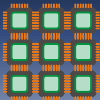
- Use too much synchronization.

Solutions

- Avoid sharing of mutable data.
- Find better approach/algorithm (e.g., map-reduce style)

  map Compute partial results locally (reduce data volume).

  reduce Combine/merge local results to final result.



Over Synchronization

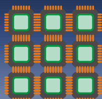
How to fix the example?

```
...
for (cur = 0; cur < COUNT; cur++ ) {
    counter++;
}
...
```

map
: Every threads counts a local variable up.

reduce
: The local counts of every thread are added to *counter*.

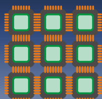
Open MP has (limited) builtin support for this via the **reduction** clause.



Over Synchronization

Example: Counting

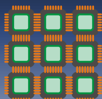
```
void counting() {
    int cur = 0;
    int counter = 0;

    #pragma omp parallel for
    for (cur = 0; cur < COUNT; cur++ ) {
        #pragma omp atomic
        counter++;
    }
    printf(" %i \n", counter);
}
```



Over Synchronization

Example: Counting

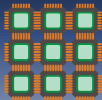
```
void counting() {
    int cur = 0;
    int counter = 0;

    #pragma omp parallel for
    for (cur = 0; cur < COUNT; cur++ ) {

        counter++;
    }
    printf(" %i \n", counter);
}
```



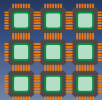
Over Synchronization

Example: Counting

```
void counting() {
    int cur = 0;
    int counter = 0;

    #pragma omp parallel for reduction(+:counter)
    for (cur = 0; cur < COUNT; cur++ ) {

        counter++;
    }
    printf(" %i \n", counter);
}
```

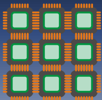


Demo



Over Synchronization

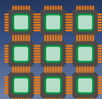
Performance (2-Cores)



False-Sharing

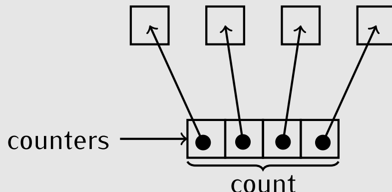
Counting Function [Sequential]

```
void *increment(int count, int **counters) {
    int cur = 0;


    for (cur = 0; cur < count; cur++ ) {
        int *counter = counters[cur];

        int i;
        for ( i = 0; i < COUNT ; i++ ) {
            (*counter)++;
        }
    }
}
```

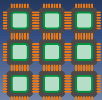


False-Sharing

Counting Function [Sequential]



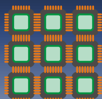
Input	array of pointers to integers in memory.
Operation	Interment each integer cell COUNT times.
Assumption	All pointers point to different integers.



False-Sharing

Counting Function [Sequential]

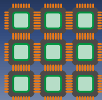
```
void *increment(int count, int **counters) {
    int cur = 0;


    for (cur = 0; cur < count; cur++ ) {
        int *counter = counters[cur];

        int i;
        for ( i = 0; i < COUNT ; i++ ) {
            (*counter)++;
        }
    }
}
```



False-Sharing

Counting Function [Parallel]

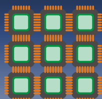
```
void *increment(int count, int **counters) {
    int cur = 0;

    #pragma omp parallel for
    for (cur = 0; cur < count; cur++ ) {
        int *counter = counters[cur];

        int i;
        for ( i = 0; i < COUNT ; i++ ) {
            (*counter)++;
        }
    }
}
```



False-Sharing

Counting Example

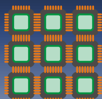
```
#define COUNT 8

void main(int argc, char *argv[]) {
    int cnt[COUNT] = {0,0,0,0,0,0,0,0};
    int *counters[COUNT] = {&cnt[0], &cnt[1], &cnt[2], &cnt[3],
                            &cnt[4], &cnt[5], &cnt[6], &cnt[7]};

    increment(COUNT, counters);
}
```



False-Sharing

Counting Example

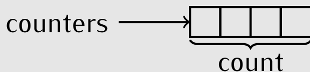
```
#define COUNT 8

void main(int argc, char *argv[]) {
    int cnt[COUNT] = {0,0,0,0,0,0,0,0};
    int *counters[COUNT] = {&cnt[0], &cnt[1], &cnt[2], &cnt[3],
                            &cnt[4], &cnt[5], &cnt[6], &cnt[7]};

    increment(COUNT, counters);
}
```

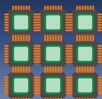


False-Sharing

Counting Example

```
#define COUNT 8

void main(int argc, char *argv[]) {
    int cnt[COUNT] = {0,0,0,0,0,0,0,0};
    int *counters[COUNT] = {&cnt[0], &cnt[1], &cnt[2], &cnt[3],
                            &cnt[4], &cnt[5], &cnt[6], &cnt[7]};

    increment(COUNT, counters);
}
```

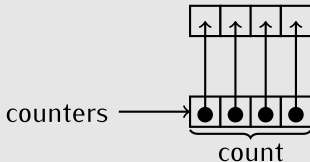


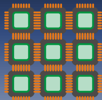


False-Sharing

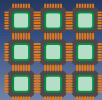
Counting Example

```
#define COUNT 8

void main(int argc, char *argv[]) {
    int cnt[COUNT] = {0,0,0,0,0,0,0,0};
    int *counters[COUNT] = {&cnt[0], &cnt[1], &cnt[2], &cnt[3],
                            &cnt[4], &cnt[5], &cnt[6], &cnt[7]};

    increment(COUNT, counters);
}
```





Demo



False-Sharing

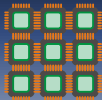
Performance

Time [%]

200%
150%
100%
50%

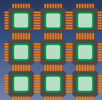
1 2 4 8

#Threads



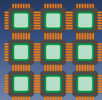
False-Sharing

Performance



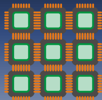
False-Sharing

Performance



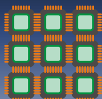
False-Sharing

Performance



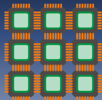
False-Sharing

Performance



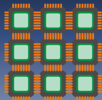
False-Sharing

Performance



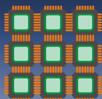
False-Sharing

Performance



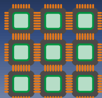
False-Sharing

Performance



False-Sharing

Performance



False-Sharing

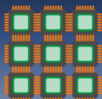
Performance

Time [%]

200%
150%
100%
50%

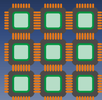
1 2 4 8

#Threads



False-Sharing

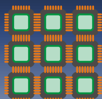
Performance



False-Sharing

What is wrong?

- The problem is trivial parallel ?!?
- We use more threads that work in parallel ?!?
- We do NOT use any synchronization ?!?



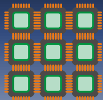
False-Sharing

What is wrong?

- The problem is trivial parallel ?!?
- We use more threads that work in parallel ?!?
- We do NOT use any synchronization ?!?

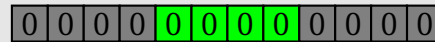
Problem

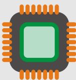
All integers **share** the same **cache line**. Caused by **cache coherency** this cache line "ping-pong" between CPUs. This is called **false sharing**.

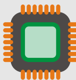


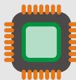
False-Sharing

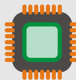
Example

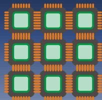






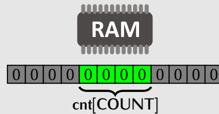




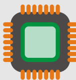


False-Sharing

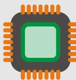
Example



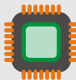
X X X X



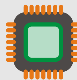
X X X X

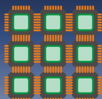


X X X X



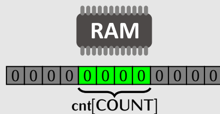
X X X X



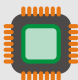


False-Sharing

Example

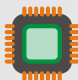


X X X X



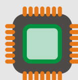
(*counter)++

X X X X



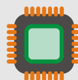
(*counter)++

X X X X

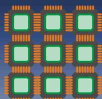


(*counter)++

X X X X

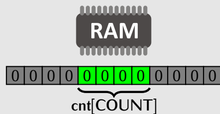


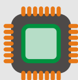
(*counter)++



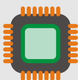
False-Sharing

Example

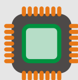




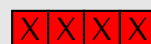
cnt[0]++

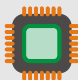


cnt[1]++

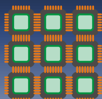


cnt[2]++



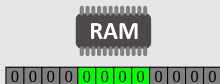


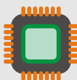
cnt[3]++



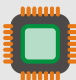
False-Sharing

Example

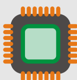




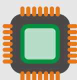
cnt[0]++



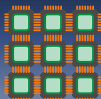
cnt[1]++



cnt[2]++

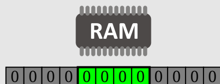


cnt[3]++

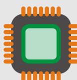


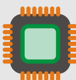
False-Sharing

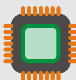
Example

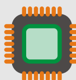


0 0 0 0 | X X X X | X X X X | X X X X

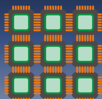








cnt[0]++ cnt[1]++ cnt[2]++ cnt[3]++



False-Sharing

Example

RAM

0	0	0	0	0	0	0	0	0	0	0	0

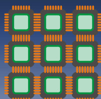
1	0	0	0

X	X	X	X

X	X	X	X

X	X	X	X

cnt[0]++ cnt[1]++ cnt[2]++ cnt[3]++



False-Sharing

Example

RAM

0	0	0	0	1	0	0	0	0	0	0	0

X	X	X	X

cnt[0]++

X	X	X	X

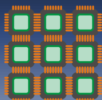
cnt[1]++

X	X	X	X

cnt[2]++

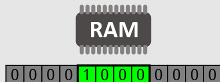
X	X	X	X

cnt[3]++

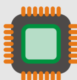


False-Sharing

Example

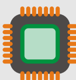


X X X X



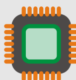
cnt[0]++

X X X X



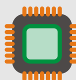
cnt[1]++

1 0 0 0

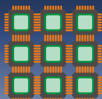


cnt[2]++

X X X X

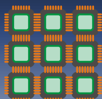


cnt[3]++



False-Sharing

Example



False-Sharing

Example

False-Sharing

Example

RAM

0	0	0	0	1	0	1	0	0	0	0	0

X	X	X	X

cnt[0]++

1	0	1	0

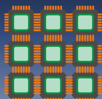
cnt[1]++

X	X	X	X

cnt[2]++

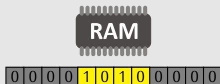
X	X	X	X

cnt[3]++

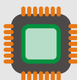


False-Sharing

Example

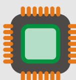


X	X	X	X



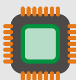
cnt[0]++

1	1	1	0



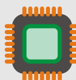
cnt[1]++

X	X	X	X

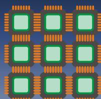


cnt[2]++

X	X	X	X



cnt[3]++



False-Sharing

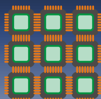
Example

RAM

0 0 0 0 1 1 1 0 0 0 0 0

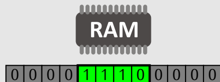
X X X X | X X X X | X X X X | X X X X

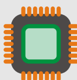
cnt[0]++ cnt[1]++ cnt[2]++ cnt[3]++



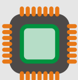
False-Sharing

Example

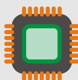




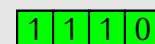
cnt[0]++

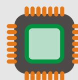


cnt[1]++



cnt[2]++

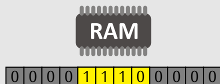


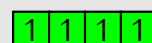


cnt[3]++

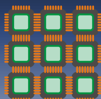
False-Sharing

Example





cnt[0]++ cnt[1]++ cnt[2]++ cnt[3]++



False-Sharing

Example

RAM

0 0 0 0 1 1 1 1 0 0 0 0

X X X X

X X X X

X X X X

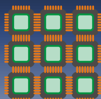
X X X X

cnt[0]++

cnt[1]++

cnt[2]++

cnt[3]++



False-Sharing

Example

RAM

0	0	0	0	1	1	1	1	0	0	0	0

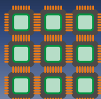
1	1	1	1

X	X	X	X

X	X	X	X

X	X	X	X

cnt[0]++ cnt[1]++ cnt[2]++ cnt[3]++



False-Sharing

Example

RAM

0	0	0	0	1	1	1	1	0	0	0	0

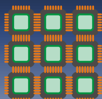
2	1	1	1

X	X	X	X

X	X	X	X

X	X	X	X

cnt[0]++ cnt[1]++ cnt[2]++ cnt[3]++



False-Sharing

Example

RAM

0	0	0	0	2	1	1	1	0	0	0	0

X	X	X	X

cnt[0]++

X	X	X	X

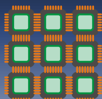
cnt[1]++

X	X	X	X

cnt[2]++

X	X	X	X

cnt[3]++



False-Sharing

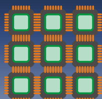
How to fix/avoid False-Sharing?

- Make sure non-shared data structures share the same cache line.
  - **padding** of data structures.

0	0	0	0

0	X	X	X	0	X	X	X	0	X	X	X	0	X	X	X

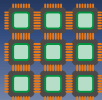
  - **aligned** allocation (e.g., posix_memalign).



False-Sharing

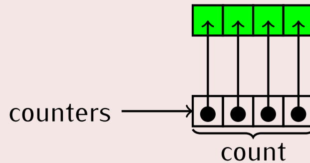
Counting Example [FIXED]

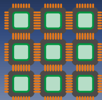
```
#define COUNT 8
#define CL ... /* system dependent */

void main(int argc, char *argv[]) {
    int cnt[COUNT * CL/sizeof(int)];
    memset(&cnt, 0, sizeof(cnt));
    int *counters[COUNT] = {
      &cnt[0*(CL/sizeof(int))], &cnt[1*(CL/sizeof(int))],
      &cnt[2*(CL/sizeof(int))], &cnt[3*(CL/sizeof(int))],
      &cnt[4*(CL/sizeof(int))], &cnt[5*(CL/sizeof(int))],
      &cnt[6*(CL/sizeof(int))], &cnt[7*(CL/sizeof(int))]
    };

    increment(COUNT, counters);
}
```



False-Sharing

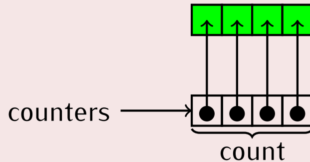
OLD apprach



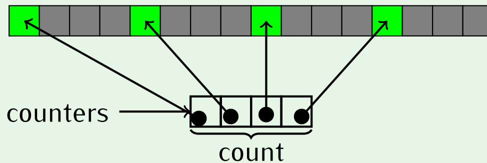


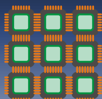
False-Sharing

OLD apprach



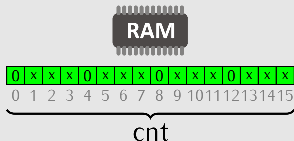
NEW approach

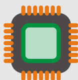


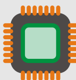


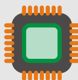
False-Sharing

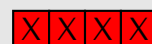
Example

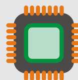


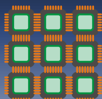












False-Sharing

Example

RAM

0	x	x	x	0	x	x	x	0	x	x	x	0	x	x	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

X	X	X	X

(∗counter)++

X	X	X	X

(∗counter)++

X	X	X	X

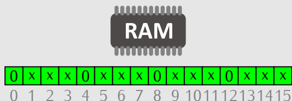
(∗counter)++

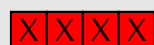
X	X	X	X

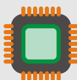
(∗counter)++

False-Sharing

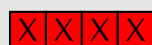
Example

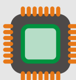




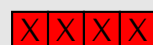


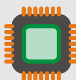
cnt[0]++



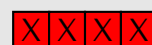


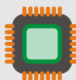
cnt[4]++



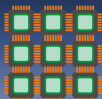


cnt[8]++





cnt[12]++



False-Sharing

Example

RAM

0	x	x	x	0	x	x	x	0	x	x	x	0	x	x	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

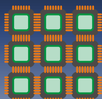
0	x	x	x

X	X	X	X

X	X	X	X

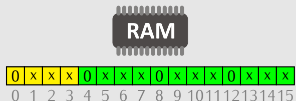
X	X	X	X

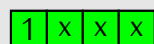
cnt[0]++ cnt[4]++ cnt[8]++ cnt[12]++

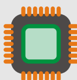


False-Sharing

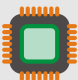
Example



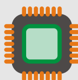




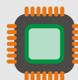
cnt[0]++



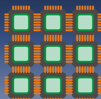
cnt[4]++



cnt[8]++



cnt[12]++



False-Sharing

Example

RAM

0	x	x	x	0	x	x	x	0	x	x	x	0	x	x	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

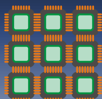
2	x	x	x

0	x	x	x

X	X	X	X

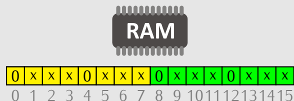
X	X	X	X

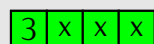
cnt[0]++ cnt[4]++ cnt[8]++ cnt[12]++

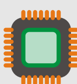


False-Sharing

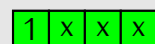
Example

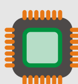




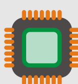


cnt[0]++

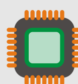




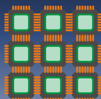
cnt[4]++



cnt[8]++

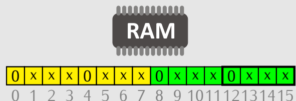


cnt[12]++



False-Sharing

Example

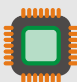


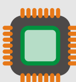
4 x x x

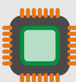
2 x x x

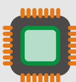
X X X X

0 x x x





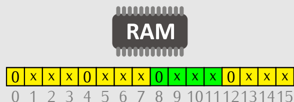




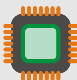
cnt[0]++ cnt[4]++ cnt[8]++ cnt[12]++

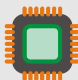
False-Sharing

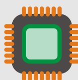
Example

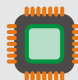


5 x x x | 3 x x x | X X X X | 1 x x x

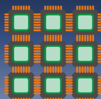








cnt[0]++ | cnt[4]++ | cnt[8]++ | cnt[12]++



False-Sharing

Example

RAM

0	x	x	x	0	x	x	x	0	x	x	x	0	x	x	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

6	x	x	x

cnt[0]++

4	x	x	x

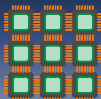
cnt[4]++

0	x	x	x

cnt[8]++

2	x	x	x

cnt[12]++



False-Sharing

Example

RAM

0	x	x	x	0	x	x	x	0	x	x	x	0	x	x	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

7	x	x	x

cnt[0]++

5	x	x	x

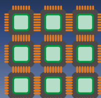
cnt[4]++

1	x	x	x

cnt[8]++

3	x	x	x

cnt[12]++



False-Sharing

Example

RAM

0	x	x	x	0	x	x	x	0	x	x	x	0	x	x	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

8	x	x	x

cnt[0]++

6	x	x	x

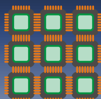
cnt[4]++

2	x	x	x

cnt[8]++

4	x	x	x

cnt[12]++



False-Sharing

Example

RAM

0	x	x	x	0	x	x	x	0	x	x	x	0	x	x	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

9	x	x	x

cnt[0]++

7	x	x	x

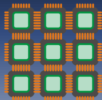
cnt[4]++

3	x	x	x

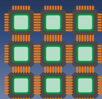
cnt[8]++

5	x	x	x

cnt[12]++

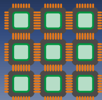


Demo



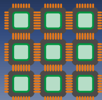
False-Sharing

Performance



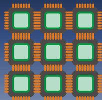
False-Sharing

Performance



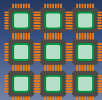
False-Sharing

Performance



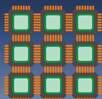
False-Sharing

Performance



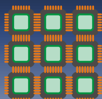
False-Sharing

Performance



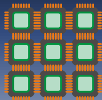
False-Sharing

Performance



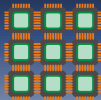
False-Sharing

Performance



How to find those problems?

- Using the right **tools**
  - profiler
  - hardware performance counters
  - software cache emulation (e.g., valgrind)
- Specific **behavior**
- **Experience**



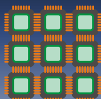
More Advanced Optimization Topics

- **Locality**
  - Avoid thread migration and force memory locality

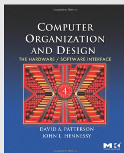
    Memory Allocate memory at specific NUMA nodes

    Threads pin threads to specific CPUs/cores

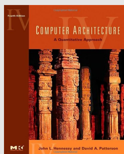
- **Translation Lookaside Buffer (TLB)**
  - cache for memory translations
  - similar to "normal" cache optimization just for the virtual memory system



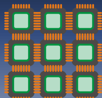
Recommendations



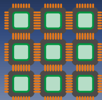
Computer Organization and Design
The Hardware/Software Interface
David A. Patterson, John L. Hennessy



Computer Architecture
A Quantitative Approach
John L. Hennessy, David A. Patterson



Why are the current systems broken?

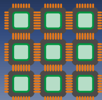


Common errors not checked!

Many approach and systems are used, but most of them do not provide an protection against concurrency bugs:

data race Multiple concurrent accesses to the same resource with at least one access being a modification.

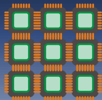
deadlock Two or more processes depend in a **circular** way on each other to release resources.



Common errors not checked!

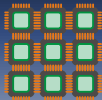
Example: Race Condition

```
void counting() {
    int cur = 0;
    int counter = 0;

    #pragma omp parallel for
    for (cur = 0; cur < COUNT; cur++ ) {
        counter++;
    }
    printf(" %i \n", counter);
}
```



Languages are broken!

- Most programming languages have **sequential semantics**.
- Concurrency is added via libraries.
- Compilers are **not** aware of concurrency and may introduce errors.



Languages are broken!

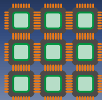
Assume $x = 0 \land x = 0$

Core 1

```
if ( y == 1 )
    x++
```

Core 2

```
if ( x == 1 )
    y++
```



Languages are broken!

Assume $x = 0 \wedge x = 0$

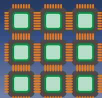
Core 1

```
if ( y == 1 )
    x++
```

Core 2

```
if ( x == 1 )
    y++
```

NO data race



Languages are broken!

Assume $x = 0 \wedge x = 0$

Core 1

```
if ( y == 1 )
    x++
```

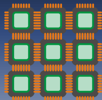
Core 2

```
if ( x == 1 )
    y++
```

NO data race

```
x++
if ( y != 1 )
    x--
```

```
y++
if ( x != 1 )
    y--
```



Languages are broken!

Assume $x = 0 \wedge x = 0$

Core 1

```
if ( y == 1 )
    x++
```

Core 2

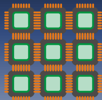
```
if ( x == 1 )
    y++
```

NO data race

```
x++
if ( y != 1 )
    x--
```

```
y++
if ( x != 1 )
    y--
```

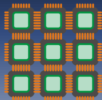
DATA RACE



Languages are broken!

- Avoiding compiler optimizations?
  $\Longrightarrow$ performance penalty.
- Sarita et al.[1] show **correct** and **efficient** concurrency support requires programming language support.
- programming language must **guarantee absence race-conditions**.

[1] Memory Models: A Case for Rethinking Parallel Languages and Hardware

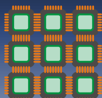


Future

The Future

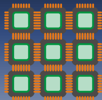
NEXT EXIT

http://worldcricketwatch.com/stories/opinion/what-is-the-future-of-cricket-in-australia/



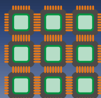
Kilim - Safe Actors

- Kilim is an shared memory Actor system for Java.
- zero-copy messaging support.
- provides strong isolation guarantees between actors.
- compiler rejects any program that violates isolation guarantees.
- similar to Microsoft's Singularity OS.



Deterministic Parallel Java (DPJ)

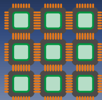
- Java extensions with parallel **sections**, **for-loop** and **regions** (similar to Open MP)
- DPJ uses **effects** and **regions** to guarantee
  - parallel computation do not interfere
    $\Longrightarrow$ **deterministic**
  - parallel computations that do interfere properly protect access to shared memory
    $\Longrightarrow$ **non-deterministic**



Deterministic Parallel Java (DPJ)

Example: DPJ Class

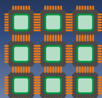
```
class Body<region P> {
    region Link, M, F;
    double mass in P:M;
    double force in P:F;
    Body<*> link in Link;

    void computeForce() reads Link, *:M writes P:F {
        force = (mass * link.mass) * R_GRAV;
    }
}
```



Deterministic Parallel Java (DPJ)

Example: DPJ Class

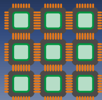
```
final Body<[_]>[]<[_] > bodies = new Body<[_]>[N]<[_]>;

foreach (int i in 0, N) {
    /* writes [i] */
    bodies[i] = new Body<[i]> ();
}

foreach (int i in 0, N) {
    /* reads [i], Link, *:M writes [i]:F */
    bodies[i].computeForce();
}
```



ÆMINIUM[2]

- Push the envelop further: Let the compiler figure out how execute code in parallel.
- Uses **permission flow** to determine potential parallelism
- uses **data group** to partition shared memory

[2] This work was partially supported by the Portuguese Research Agency – FCT, through a scholarship (SFRH / BD / 33522 / 2008), CISUC (R&D Unit 326/97), the CMU—Portugal program.

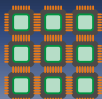


Access Permissions

unique
- there is only one reference to the object
- **exclusive access**
- no synchronization required

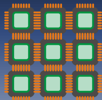
immutable
- there might be several alias reference to the object, but **all** of them are **immutable**
- the object **cannot** be modified through an immutable reference
- no synchronization required

shared
- there might be several alias reference to the object, but **all** of them are **shared**
- the object **can** be modified through an shared reference
- **access** to shared objects requires synchronization



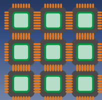
Inferring concurrency with permissions

- **automatically splitting/joining** of permissions
  e.g., unique $\Longleftrightarrow$ immutable $\otimes$ immutable
  e.g., unique $\Longleftrightarrow$ shared $\otimes$ shared
  e.g., shared $\Longleftrightarrow$ shared $\otimes$ shared
- use **linear logic** and **fractions** for management access permissions
- **“reverse”** this approach and **infer** concurrency from **permission flow**
  1. infer **permission flow** base on **lexical order**
  2. DEFINE that **operations** can **run concurrently** iff they depend on:
     immutable permissions $\rightsquigarrow$ only read operations
     shared permissions $\rightsquigarrow$ access must synchronized
  3. generate **dataflow graph**



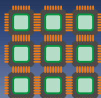
Example: Transfer

```
public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
    withdraw(from, amount)
    deposit(to, amount);
}
```



Example: Transfer

```
public void withdraw(unique Account account,
                     immutable Amount amount) {...}


public void deposit(unique Account account,
                    immutable Amount amount) {...}


public void transfer(unique Account from,
                     unique Account to,
                     immutable Amount amount) {
    withdraw(from, amount)
    deposit(to, amount);
}
```



ÆMINIUM

transfer(to, from, amount) {

to : unique,
from : unique,
amount : immutable

split

amount : immutable,
to : unique

amount : immutable,
from : unique

deposit(to, amount)

withdraw(from, amount)

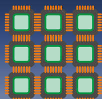
to : unique,
amount : immutable

from : unique,
amount : immutable

join

to : unique,
from : unique,
amount : immutable

}

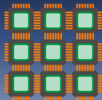


Conclusion

- Getting things **right** (i.e., correct **and** performance) is **hard**.
- Manage your **expectations** and **goals**.
- Analyze your problem (i.e., **benchmark** and **profile**).
- **Know your ENEMY!!!**

Thanks for the Attention!

Questions?



What does the name ÆMINIUM come from?

- ÆMINIUM was the **ancient roman city** on which **Coimbra** was established.