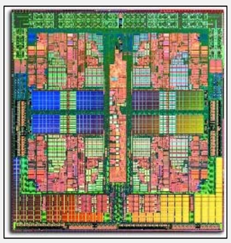
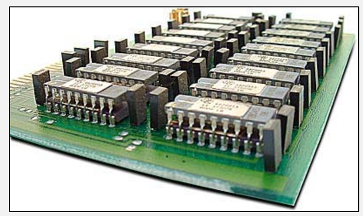
Caching Issues in Multicore Performance

Mike Bailey

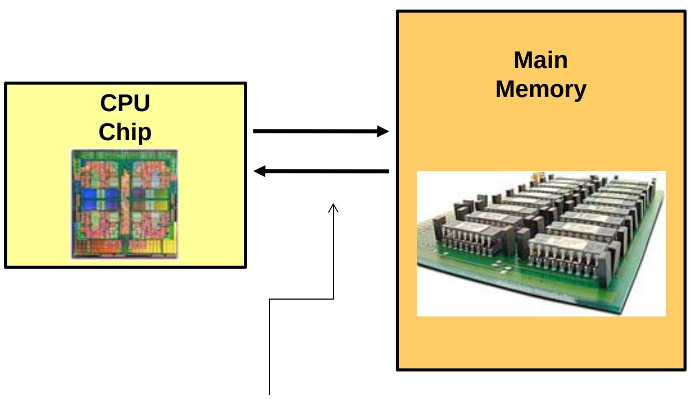
mjb@cs.oregonstate.edu

Oregon State University





Problem: The Path Between a CPU Chip and Off-chip Memory is Slow

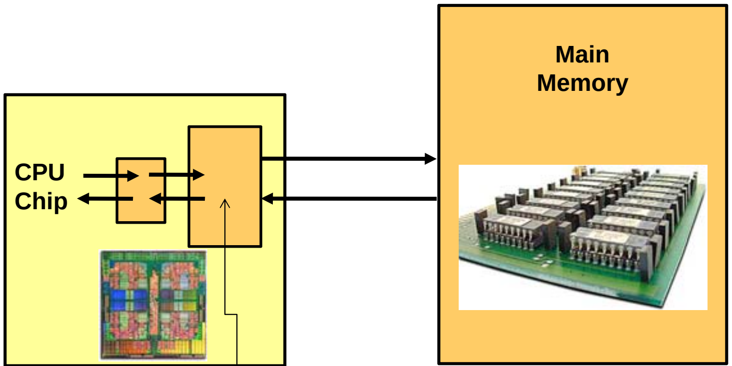


This path is relatively slow, forcing the CPU to wait for up to 200 clock cycles just to do a store to, or a load from, memory.

Depending on your CPU's ability to process instructions out-of-order, it might go idle during this time.

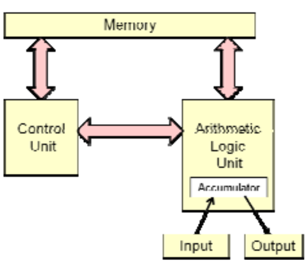
This is a *huge* performance hit!

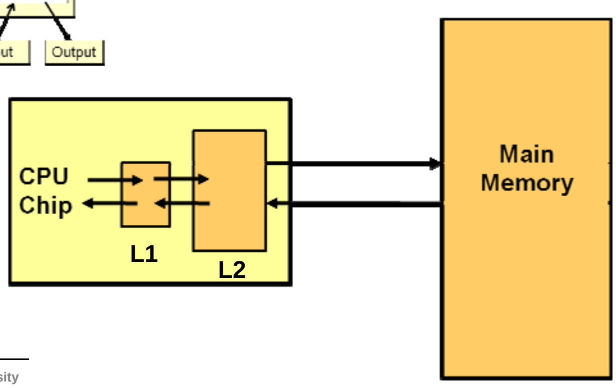
Solution: Hierarchical Memory Systems, or “Cache”



The solution is to add intermediate memory systems. The one closest to the CPU is small and fast. The memory systems get slower and larger as they get farther away from the CPU.

Cache and Memory are Named by “Distance Level” from the ALU





Storage Level Characteristics

	L1	L2	Memory	Disk
Type of Storage	On-chip	On-chip	Off-chip	Disk
Typical Size	< 100 KB	< 8 MB	< 10 GB	Many GBs
Typical Access Time (ns)	.25 - .50	.5 – 25.0	50 - 250	5,000,000
Scaled Access Time	1 second	33 seconds	7 minutes	154 days
Bandwidth (MB/sec)	50,000 – 500,000	5,000 – 20,000	2,500 – 10,000	50 - 500
Managed by	Hardware	Hardware	OS	OS

Adapted from: John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, 2007. (4th Edition)

Usually there are two L1 caches – one for Instructions and one for Data. You will often see this referred to in data sheets as: "L1 cache: 32KB + 32KB" or "I and D cache"

Cache Hits and Misses

When the CPU asks for a value from memory, and that value is already in the cache, it can get it quickly.
This is called a ***cache hit***

When the CPU asks for a value from memory, and that value is not already in the cache, it will have to go off the chip to get it.
This is called a ***cache miss***

While cache might be multiple kilo- or megabytes, the bytes are transferred in much smaller quantities, each called a **cache line.** The size of a cache line is typically just **64 bytes**.

Performance programming should strive to avoid as many cache misses as possible. That's why it is very helpful to know the cache structure of your CPU.

Spatial and Temporal Coherence

Successful use of the cache depends on **Spatial Coherence**:

"If you need one memory address's contents now, then you will probably also need the contents of some of the memory locations around it soon."

Successful use of the cache depends on **Temporal Coherence**:

"If you need one memory address's contents now, then you will probably also need its contents again soon."

If these assumptions are true, then you will generate a lot of cache hits.

If these assumptions are not true, then you will generate a lot of cache misses, or you re-load the cache a lot.

How Bad Is It? -- Demonstrating the Cache-Miss Problem

C and C++ store 2D arrays a row-at-a-time, like this, A[i][j]:

[j] →

[i] ↓

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

```
sum = 0.;
for( int i = 0; i < NUM; i++ )
{
        for( int j = 0; j < NUM; j++ )
        {
                float  f = ???
                sum += f;
        }
}
```

For large arrays, would it be better to add the elements by row, or by column? Which will avoid the most cache misses?

Sequential memory order → `float f = Array[ i ][ j ] ;`

Jump-around-in-memory order → `float f = Array[ j][ i ] ;`

Demonstrating the Cache-Miss Problem

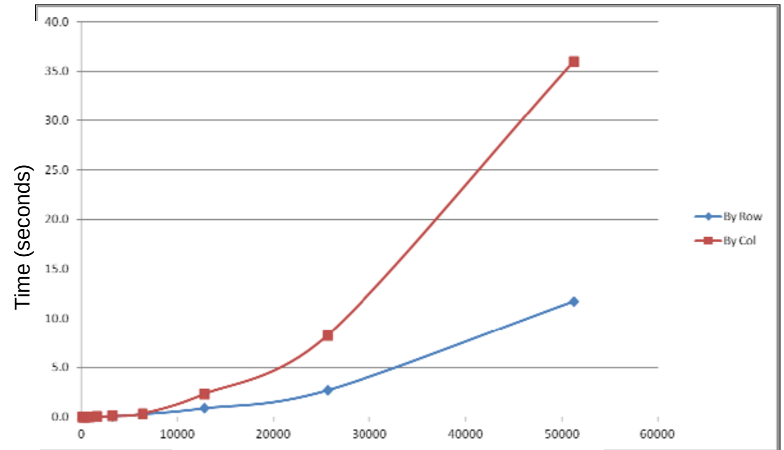
```
#include <stdio.h>
#include <ctime>
#include <cstdlib>

#define NUM  10000

float   Array[NUM][NUM];

double  MyTimer( );

int
main( int argc, char *argv[ ] )
{
        float sum = 0.;
        double start  = MyTimer( );
        for( int i = 0; i < NUM; i++ )
        {
                for( int j = 0; j < NUM; j++ )
                {
                        sum += Array[ i ][ j ];              // access across a row
                }
        }
        double finish = MyTimer( );

        double row_secs = finish – start;
```


Demonstrating the Cache-Miss Problem

```
        sum = 0.;
        start  = MyTimer( );
        for( int i = 0; i < NUM; i++ )
        {
                for( int j = 0; j < NUM; j++ )
                {
                        sum += Array[ j ][ i ];          // access down a column
                }
        }
        finish = MyTimer( );

        double col_secs = finish - start;
        fprintf( stderr, "NUM = %5d ; By rows = %lf ; By cols = %lf\n",
                                        NUM, row_secs, col_secs );
}
```

Demonstrating the Cache-Miss Problem

Time, in seconds, to compute the array sums, based on by-row versus by-column order:



Dimension (NUM)
(Total array size = NUMxNUM)

Array-of-Structures vs. Structure-of-Arrays:

```
struct xyz
{
		float x, y, z;
}  Array[N];
```

X0 Y0 Z0
X1 Y1 Z1
X2 Y2 Z2
X3 Y3 Z3

```
float X[N], Y[N], Z[N];
```

X0 X1 X2 X3 . . .
Y0 Y1 Y2 Y3 . . .
Z0 Z1 Z2 Z3 . . .

1. **Which is a better use of the cache if we are going to be using X-Y-Z triples a lot?**
2. **Which is a better use of the cache if we are going to be looking at all X's, then all Y's, then all Z's?**

I've seen some programs use a "Shadow Data Structure" to get the advantages of both AOS and SOA

Computer Graphics is often a Good Use for Array-of-Structures:

X0 Y0 Z0
X1 Y1 Z1
X2 Y2 Z2
X3 Y3 Z3

```
struct xyz
{
        float x, y, z;
}  Array[N];

. . .

glBegin( GL_LINE_STRIP );
for( int i = 0; i < N; i++ )
{
        glVertex3f( Array[ i ].x, Array[ i ].y, Array[ i ].z );
}
glEnd( );
```


Good Use for Structure-of-Arrays:

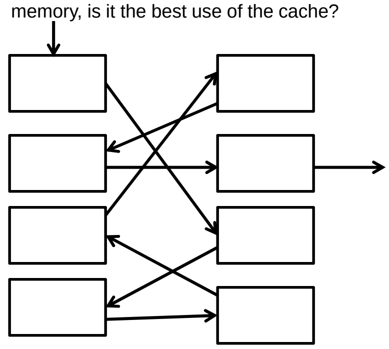
X0 X1 X2 X3 . . .
Y0 Y1 Y2 Y3 . . .
Z0 Z1 Z2 Z3 . . .

```
float X[N], Y[N], Z[N];
float Dx[N], Dy[N], Dz[N];
. . .

Dx[0:N] = X[0:N]  -  Xnow;
Dy[0:N] = Y[0:N]  -  Ynow;
Dz[0:N] = Z[0:N]  -  Znow;
```

Good Object-Oriented Programming Style can sometimes be Inconsistent with Good Cache Use:

```
class xyz
{
   public:
         float x, y, z;
         xyz *next;
         xyz( );
         static xyz *Head = NULL;
};

xyz::xyz( )
{
         xyz * n = new xyz;
         n->next = Head;
         Head = n;
};
```

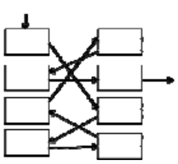
This is good OO style – it encapsulates and isolates the data for this class. Once you have created a linked list whose elements are all over memory, is it the best use of the cache?

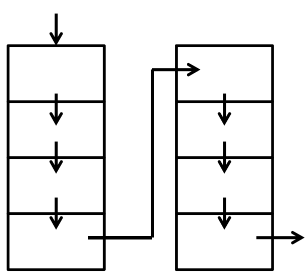


But, Here Is a Compromise:

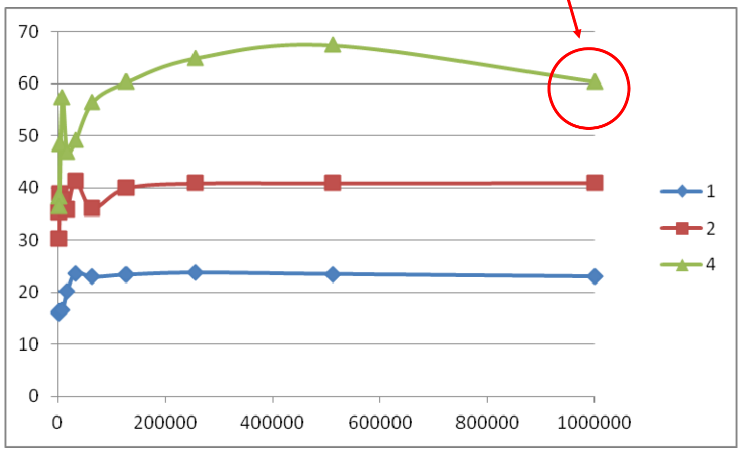
It might be better to create a large array of xyz structures and then have the constructor method pull new ones from that list. That would keep many of the elements close together while preserving the flexibility of the linked list.

When you need more, allocate another large array and link to it.



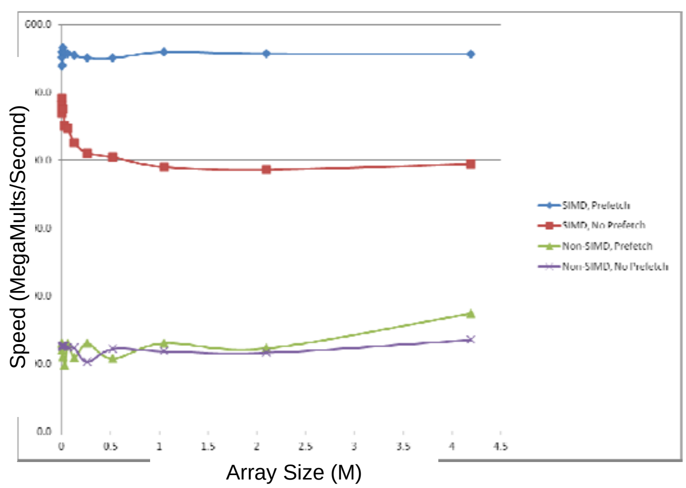


Why Can We Get This Kind of Performance Decrease as Data Sets Get Larger?



We are violating Temporal Coherence

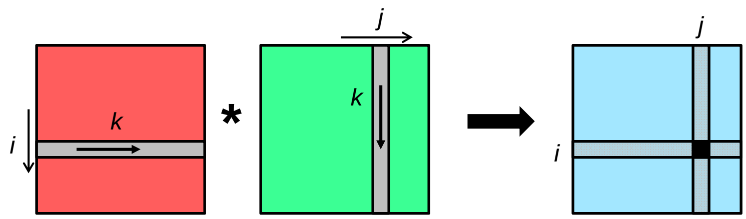
We Can Help the Temporal Problem with Pre-Fetching



We will cover this in further detail when we discuss SIMD

An Example of Where Cache Coherence Really Matters: Matrix Multiply

The usual approach is multiplying the entire A row * entire B column
This is equivalent to computing a single dot product



Row i of A * Column j of B → Element (i,j) of C

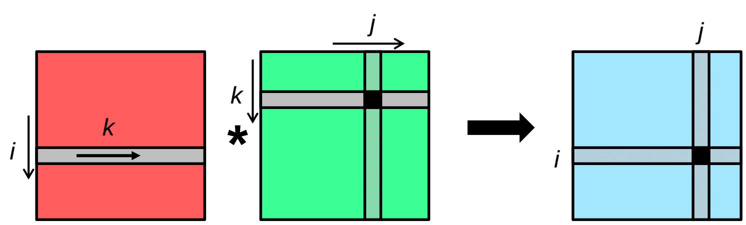
```
for( i = 0; i < SIZE; i++ )
    for( j = 0; j < SIZE; j++ )
        for( k = 0; k < SIZE; k++ )
```

∑ A[i][k] * B[k][j] —Sum and store→ C[i][j]

Problem: Column j of the B matrix is not doing a unit stride

An Example of Where Cache Coherence Really Matters: Matrix Multiply

Scalable Universal Matrix Multiply Algorithm (SUMMA)
Entire A row * one element of B row
Equivalent to computing one item in many separate dot products

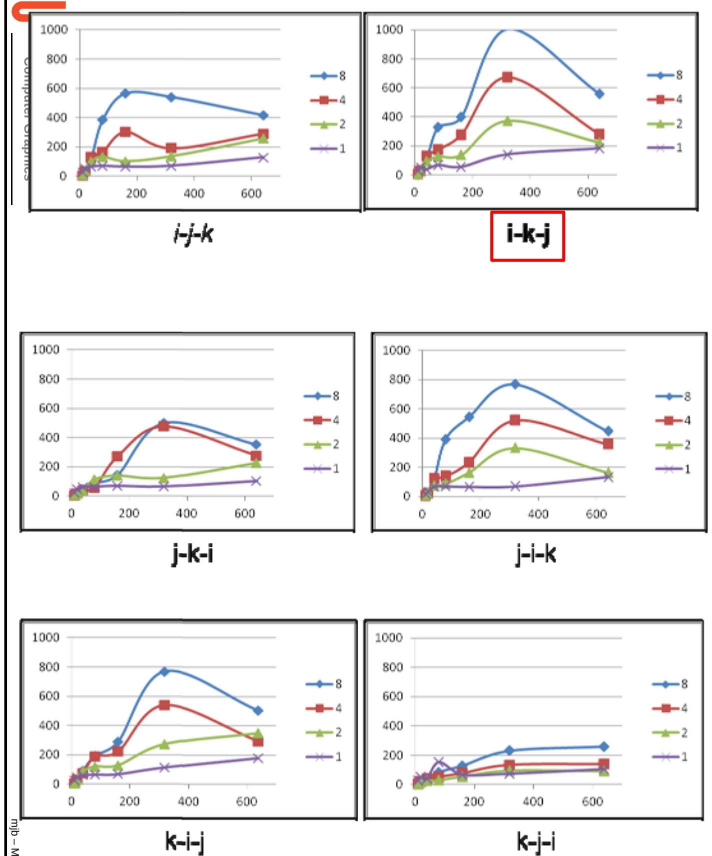


Row i of A   Row k of B   Element (i,j) of C

```
for( i = 0; i < SIZE; i++ )
    for( k = 0; k < SIZE; k++ )
        for( j = 0; j < SIZE; j++ )
```

A[i][k] * B[k][j] —Add to→ C[i][j]

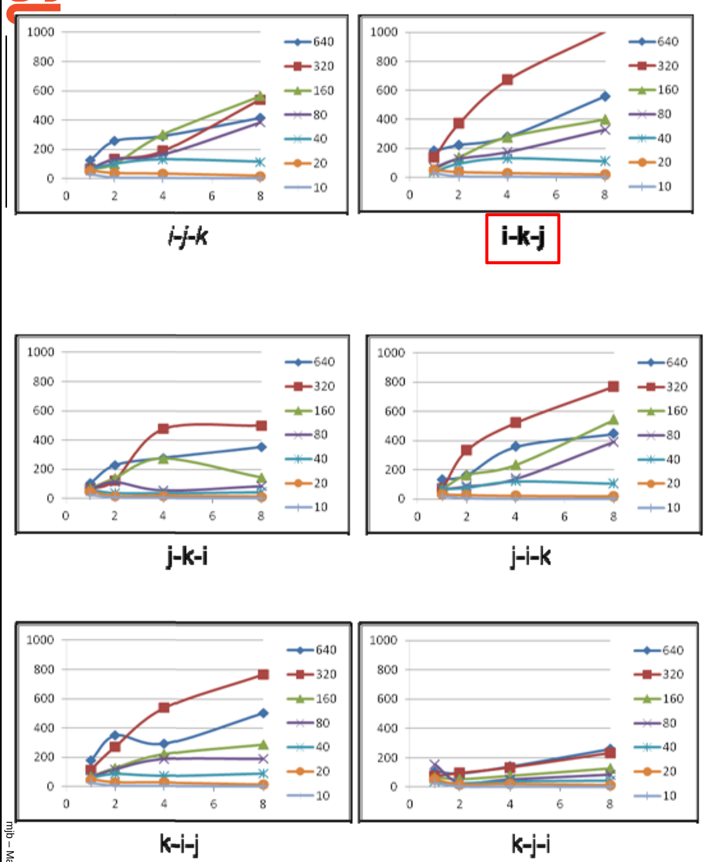
Performance vs. Matrix Size (MegaMultiplies / Sec)



mjb – May 2, 2017

21

Performance vs. Number of Threads (MegaMultiplies / Sec)



mjb – May 2, 2017

22

Cache Architectures

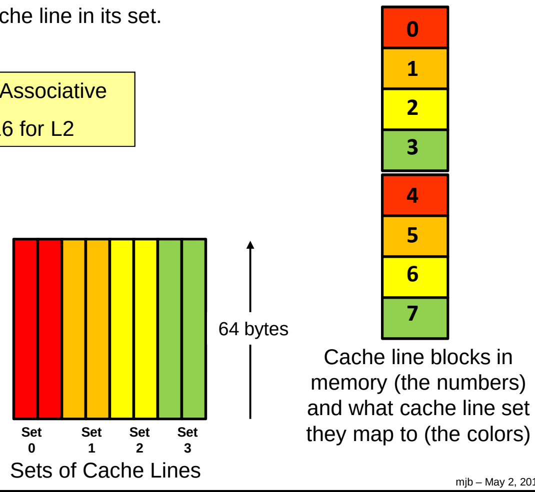
N-way Set Associative – a cache line from a particular block of memory can appear in a limited number of places in cache. Each "limited place" is called a **set** of cache lines. A set contains **N** cache lines.

The memory block can appear in any cache line in its set.

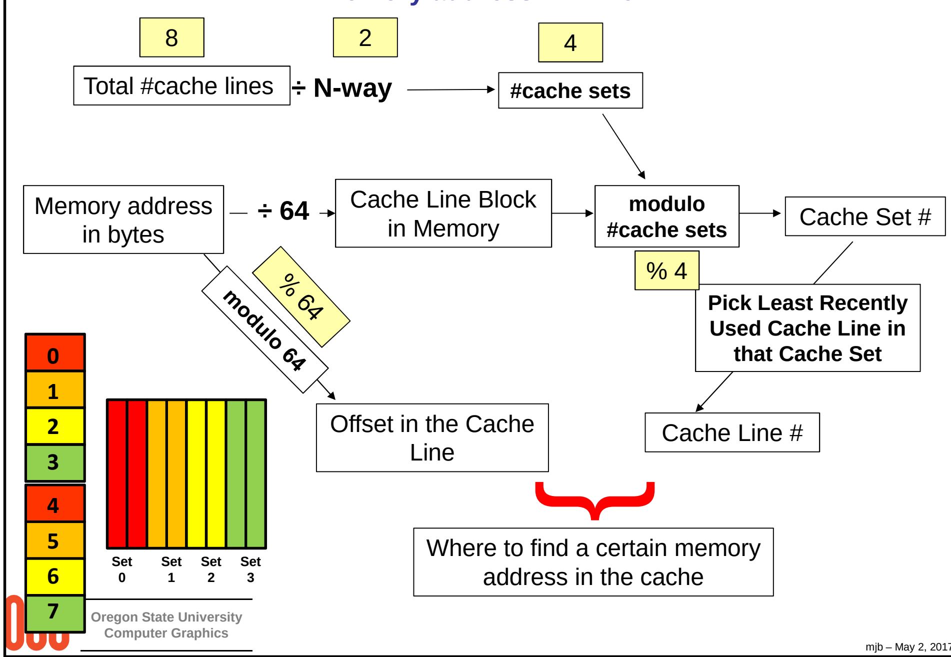
Most Caches today are N-way Set Associative

N is typically 4 for L1 and 8 or 16 for L2

This would be called "2-way"



How do you figure out where in cache a specific memory address will live?



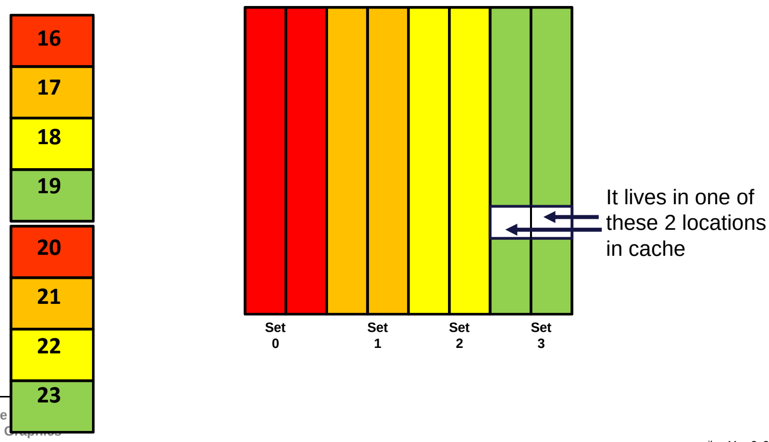
A Specific Example with Numbers

Memory address = 1234 bytes

Cache Line Block in Memory = 19

Cache Set # = 3

Offset in the Cache Line = 18



Cache Can Interact with Cores in Unexpected Ways

Each core has its own separate L2 cache, but a write by one can impact the state of the others.

For example, if one core writes a value into one of its own cache lines, any other core using a copy of that same cache line can no longer count on its values being up-to-date. In order to regain that confidence, the core that wrote must flush that cache line back to memory and the other core must then reload its copy of that cache line.

To maintain this organization, each core's L2 cache has 4 states (***MESI***):

1. **M**odified
2. **E**xclusive
3. **S**hared
4. **I**nvalid

A Simplified View of How MESI Works

1. Core A reads a value. Those values are brought into its cache. That cache line is now tagged **Exclusive**.
2. Core B reads a value from the same area of memory. Those values are brought into its cache, and now both cache lines are re-tagged **Shared**.
3. If Core B writes into that value. Its cache line is re-tagged **Modified** and Core A's cache line is re-tagged **Invalid**.

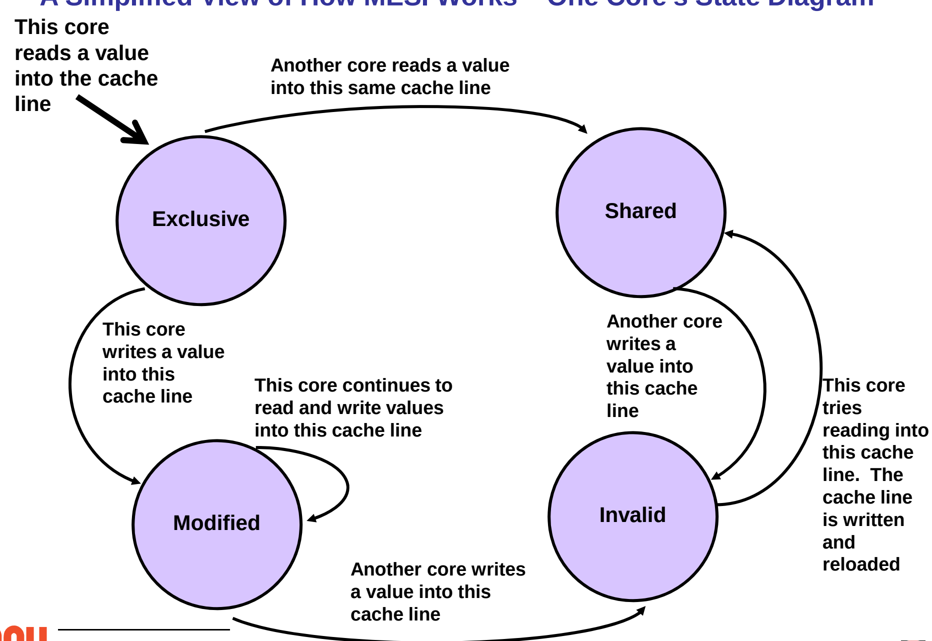
Step	Cache Line A	Cache Line B
1	Exclusive	-----
2	Shared	Shared
3	Invalid	Modified
4	Shared	Shared

4. Core A tries to read a value from that same part of memory. But its cache line is tagged **Invalid**. So, *Core B's cache line is flushed back to memory and then Core A's cache line is re-loaded from memory*. Both cache lines are now tagged **Shared**.

This is a huge performance hit, and is referred to as ***False Sharing***

Note that False Sharing doesn't create incorrect results – just a performance hit.
If anything, False Sharing *prevents* getting incorrect results.

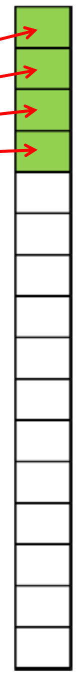
A Simplified View of How MESI Works – One Core's State Diagram



False Sharing – An Example Problem

```
struct s
{
        float value;
}   Array[4];

omp_set_num_threads(  4 );

#pragma omp parallel for
        for( int i = 0; i < 4; i++ )
        {
                for( int j = 0; j < SomeBigNumber; j++ )
                {
                        Array[ i ].value  =  Array[ i ].value  +  (float)rand( );
                }

        }
```

Some unpredictable function so the compiler doesn't try to optimize the j-for-loop away.

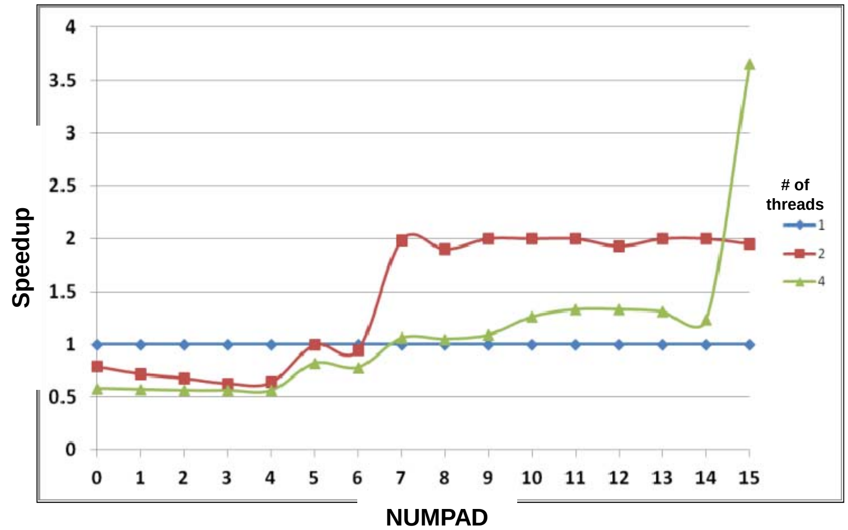


False Sharing – Fix #1

```
#include <stdlib.h>
struct s
{
        float value;
        int pad[NUMPAD];
}   Array[4];

const  int  SomeBigNumber = 100000000;  // keep less than 2B

omp_set_num_threads(  4 );

#pragma omp parallel for
        for( int i = 0; i < 4; i++ )
        {
                for( int j = 0; j < SomeBigNumber; j++ )
                {
                                Array[ i ].value  =  Array[ i ].value + (float)rand( );
                }

        }
```

This works because successive Array elements are forced onto different cache lines, so less (or no) cache line conflicts exist

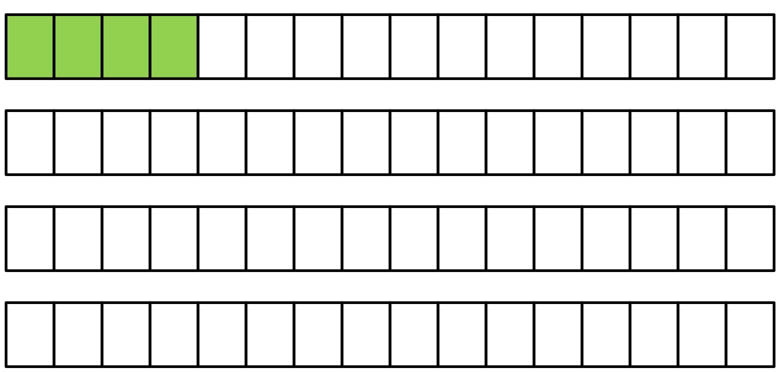
False Sharing – Fix #1

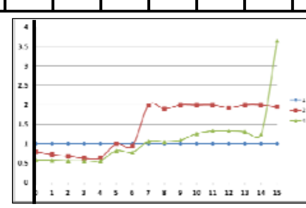


Why do these curves look this way?

False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

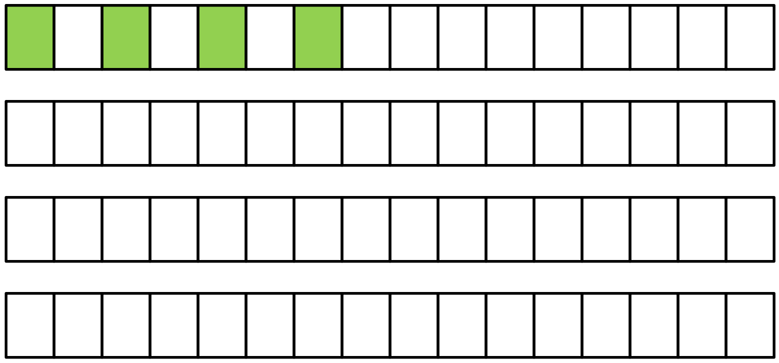
NUMPAD = 0

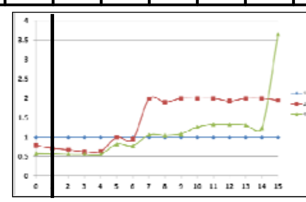




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

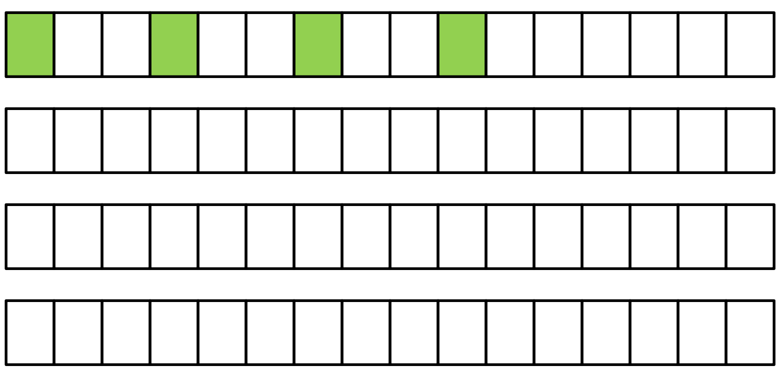
NUMPAD = 1

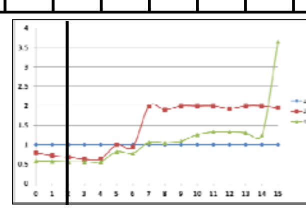




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

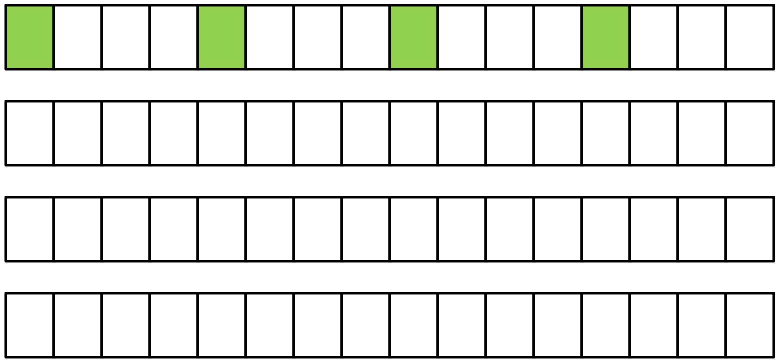
NUMPAD = 2

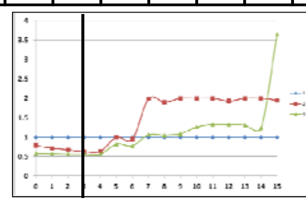




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

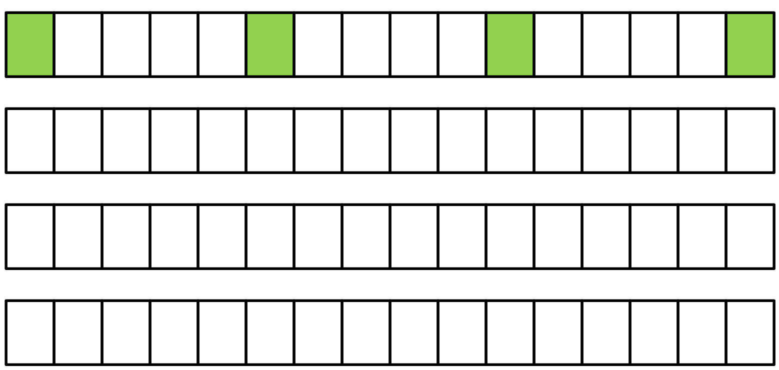
NUMPAD = 3

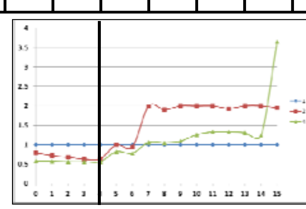




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

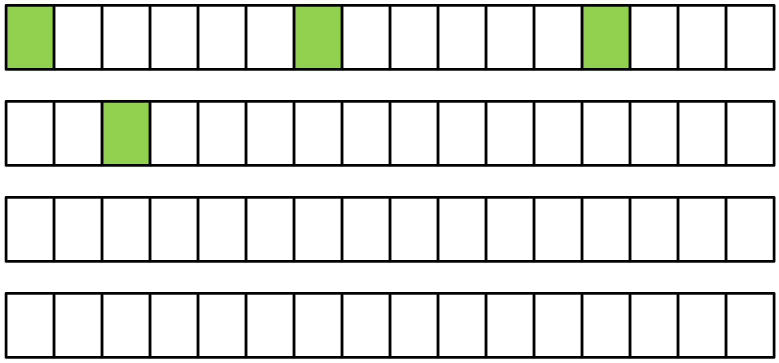
NUMPAD = 4

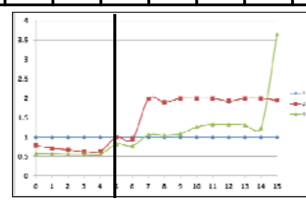




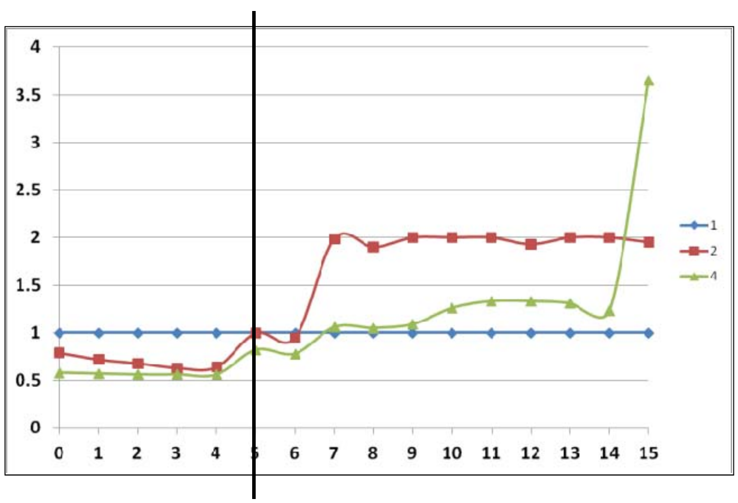
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 5





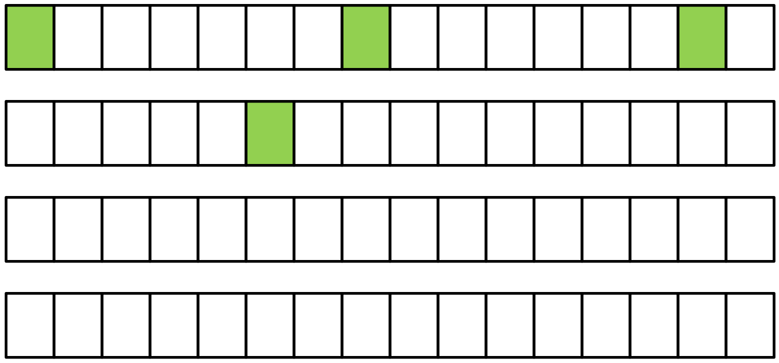
False Sharing – Fix #1

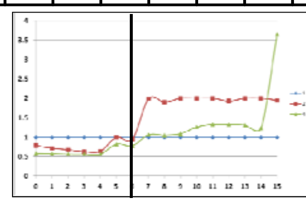


39

False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 6





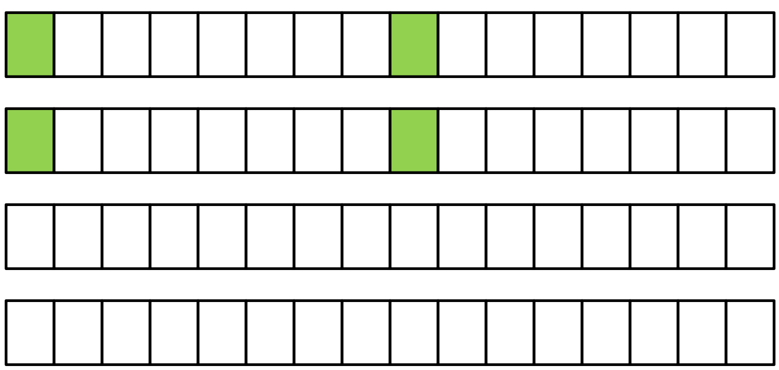
Oregon State University
Computer Graphics

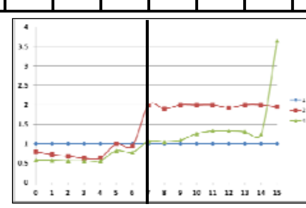
mjb – May 2, 2017

40

False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 7

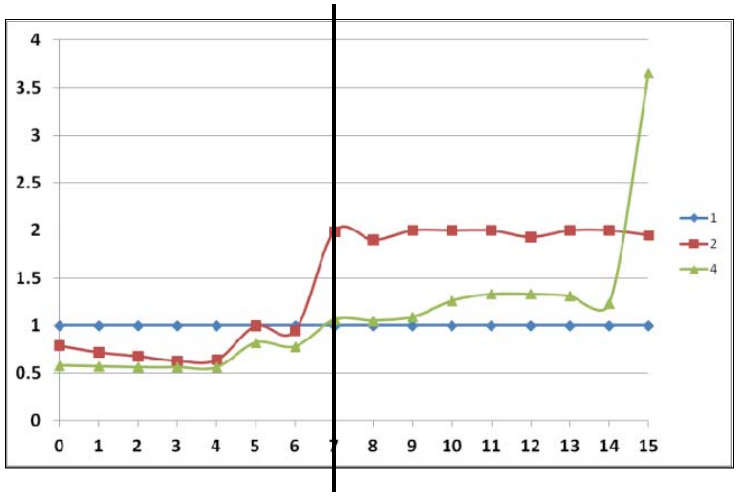




Oregon State University
Computer Graphics

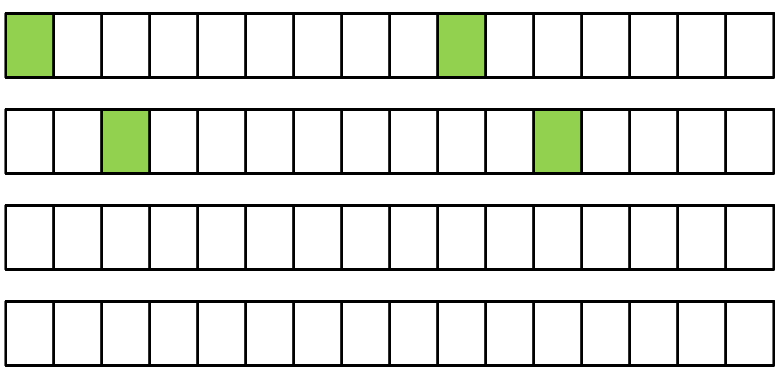
mjb – May 2, 2017

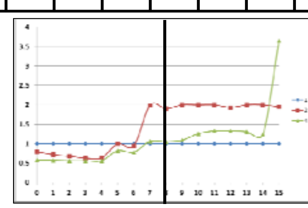
False Sharing – Fix #1



False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

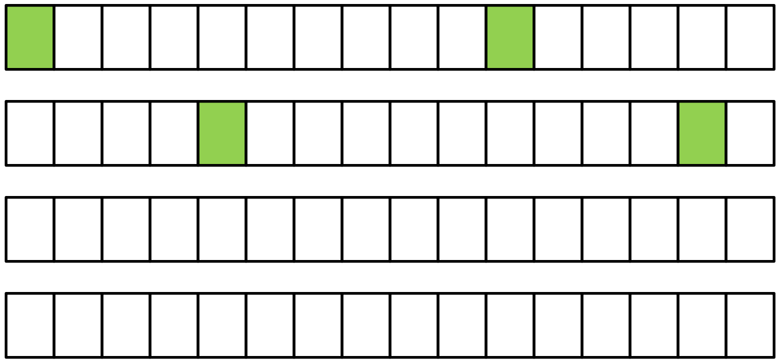
NUMPAD = 8

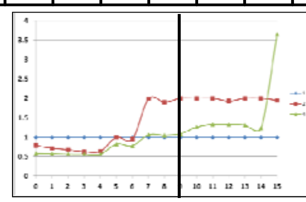




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

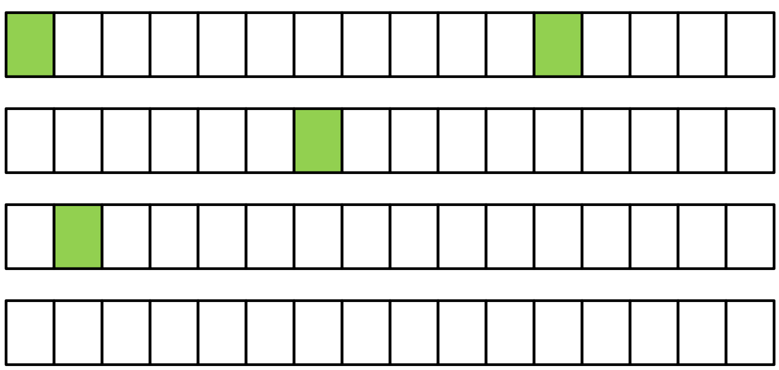
NUMPAD = 9

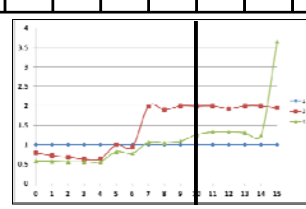




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

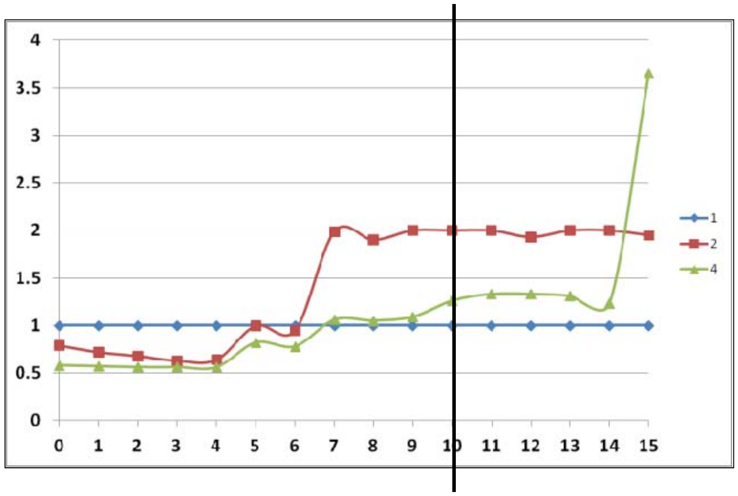
NUMPAD = 10





45

False Sharing – Fix #1



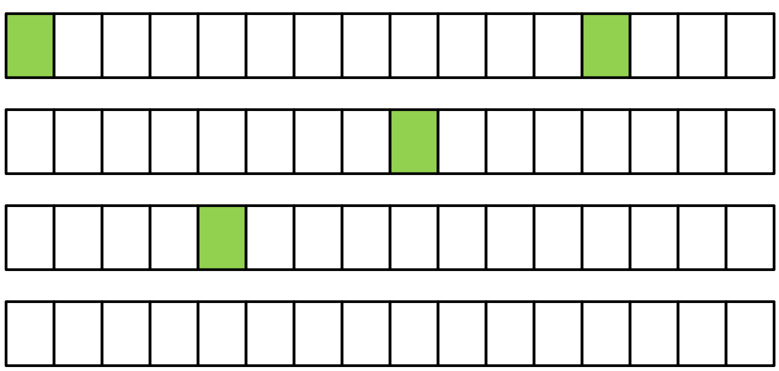
Oregon State University
Computer Graphics

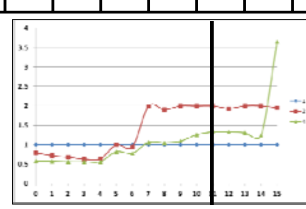
mjb – May 2, 2017

46

False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 11



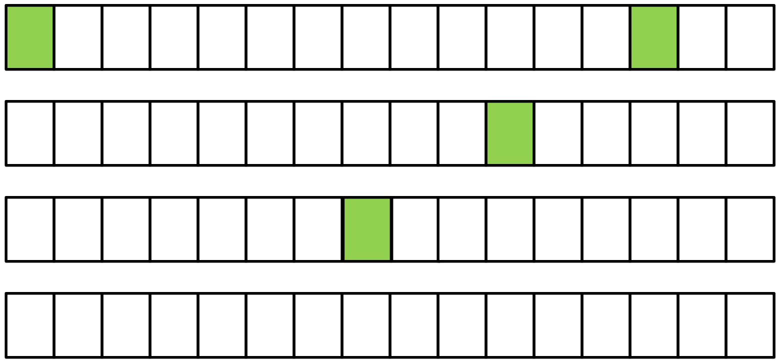


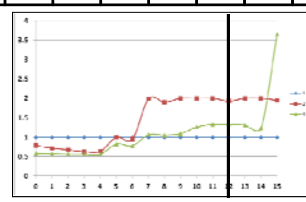
Oregon State University
Computer Graphics

mjb – May 2, 2017

False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

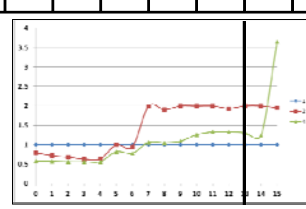
NUMPAD = 12





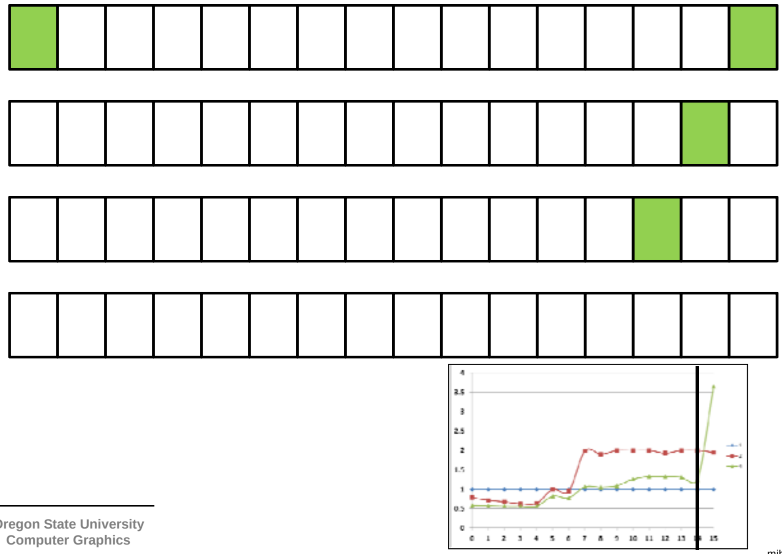
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 13



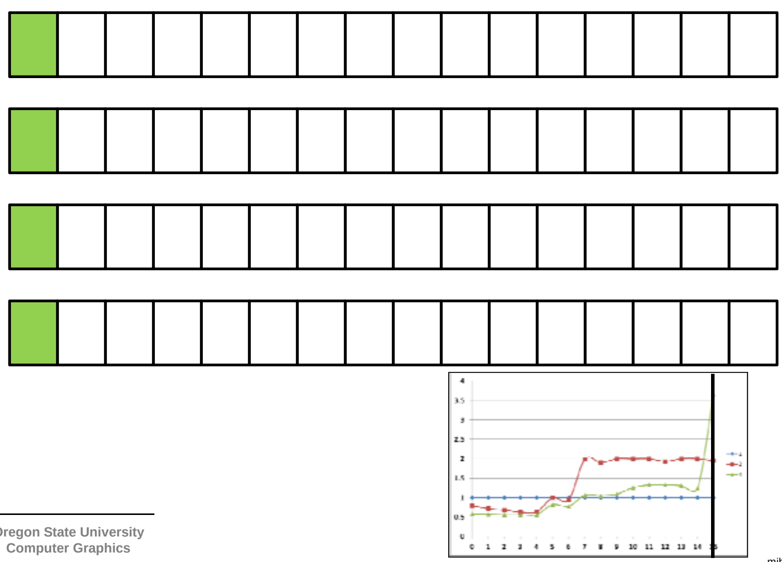
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 14



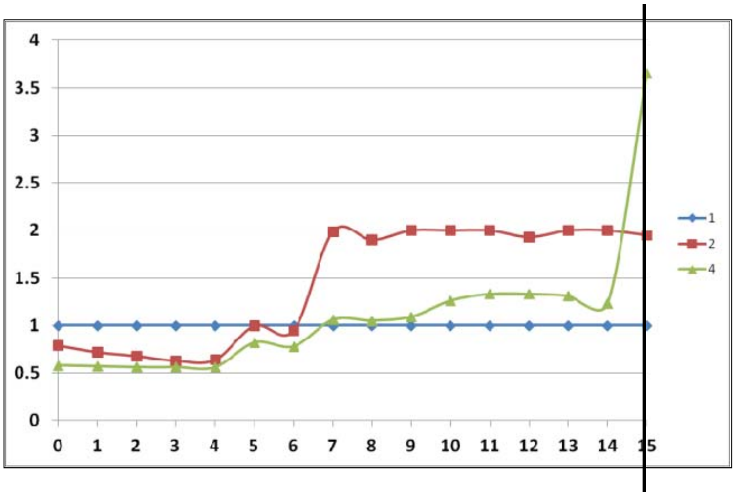
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 15



51

False Sharing – Fix #1



Oregon State University
Computer Graphics

mjb – May 2, 2017

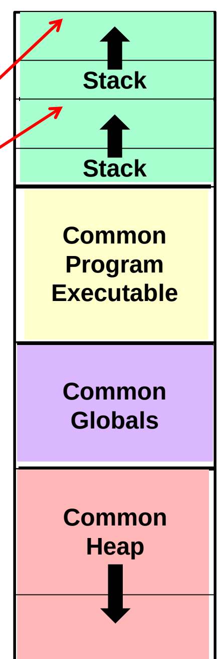
52

False Sharing – Fix #2

OK, wasting memory to put your data on different cache lines seems a little silly (even though it works). Can we do something else?

Remember our discussion in the OpenMP section about how stack space is allocated for different threads?

If we use local variables, instead of contiguous array locations, that will spread our writes out in memory, and to different cache lines.



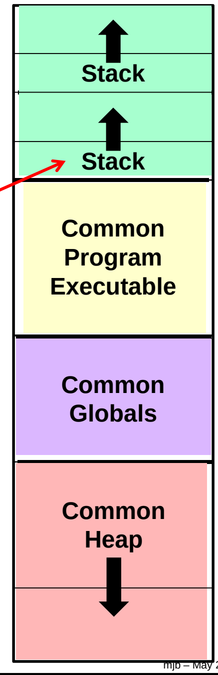
Oregon State University
Computer Graphics

mjb – May 2, 2017

False Sharing – Fix #2

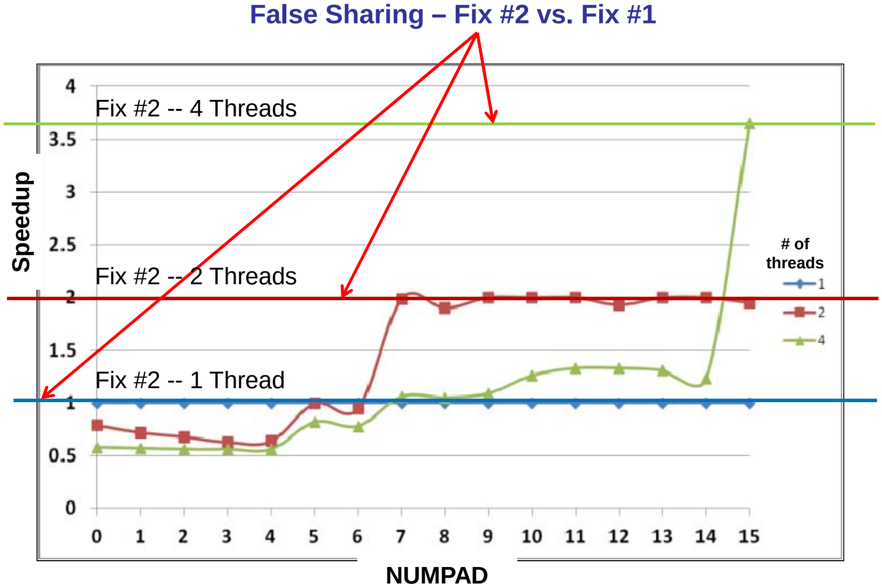
```
#include <stdlib.h>
struct s
{
        float value;
}   Array[4];

omp_set_num_threads( 4 );

const int SomeBigNumber = 100000000;

#pragma omp parallel for
        for( int i = 0; i < 4; i++ )
        {
                float  tmp = Array[ i ].value;
                for( int j = 0; j < SomeBigNumber; j++ )
                {
                        tmp =  tmp + (float)rand( );
                }
                Array[ i ].value = tmp;
        }
```

Makes this a private variable that lives in each thread's individual stack



This works because a localized temporary variable is created in each core's stack area, so little or no cache line conflict exists

False Sharing – Fix #2 vs. Fix #1



Note that Fix #2 gives the same performance as NUMPAD=15

malloc'ing on a cache line

What if you are malloc'ing, and want to be sure your data structure starts on a cache line?

Knowing that cache lines start on fixed 64-byte boundaries lets you do this. Consider a memory address. The top N-6 bits tell you what cache line number this address is a part of. The bottom 6 bits tell you what offset that address has within that cache line. So, for example, on a 32-bit memory system:

32-6 = 26 bits	6 bits
Cache line number	Offset in that cache line

So, if you see a memory address whose bottom 6 bits are 000000, then you know that that memory location begins a cache line.

malloc'ing on a cache line

Let's say that you have a structure and you want to malloc an ARRAYSIZE array of them. Normally, you would do this:

```
struct xyzw *p = (struct xyzw *) malloc(  (ARRAYSIZE)*sizeof(struct xyzw)  );
struct xyzw *Array = &p[0];
. . .
Array[ i ].x = 10. ;
```

If you wanted to make sure that array of structures started on a cache line boundary, you would do this:

```
unsigned char *p = (unsigned char *) malloc(  64 + (ARRAYSIZE)*sizeof(struct xyzw)  );
int offset = (long int)p  &  0x3f;          // 0x3f = bottom 6 bits are all 1's
struct xyzw *Array = (struct xyzw *) &p[64-offset];
. . .
Array[ i ].x = 10. ;
```

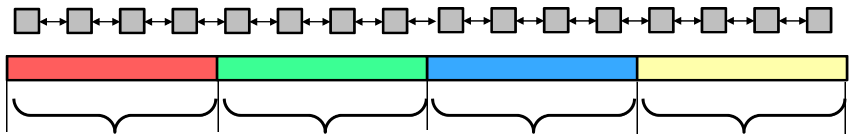
Remember that when you want to free this malloc'ed space, be sure to say:

```
free( p );
```

not:

```
free( Array );
```

Now, Consider This Type of Computation



Should you allocate the data as one large global-memory block (i.e., shared)?
Or, should you allocate it as separate blocks, each local to its own core (i.e., private)?
Does it matter? Yes!

If you allocate the data as one large global-memory block, there is a risk that you will get False Sharing at the individual-block boundaries. Solution: make sure that each individual-block starts and ends on a cache boundary, even if you have to pad it. **(Fix #1!)**

If you allocate the data as separate blocks, then you don't have to worry about False Sharing **(Fix #2!)**, but you do have to worry about the logic of your program remembering where to find each Node #i-1 and Node #i+1.