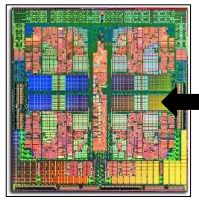
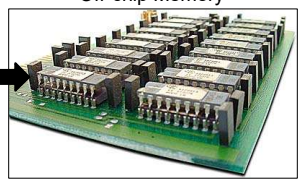
1

Caching Issues in Multicore Performance





Oregon State University

Mike Bailey

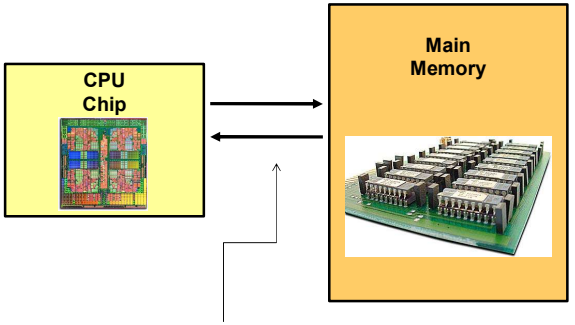
mjb@cs.oregonstate.edu

Oregon State University Computer Graphics

1

2

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.

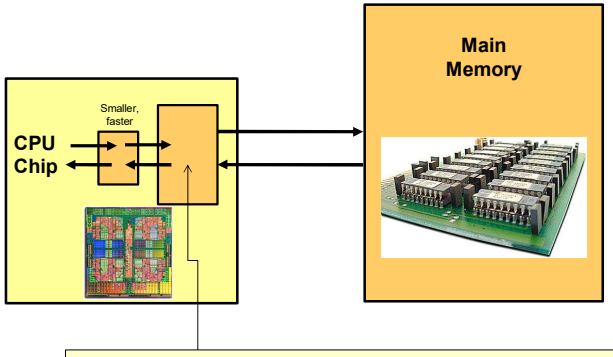
This is a *huge* performance hit!

Oregon State University Computer Graphics

2

3

Solution: Hierarchical Memory Systems, or "Cache"



The solution is to add intermediate memory systems. The one closest to the ALU (L1) is small and fast. The memory systems get slower and larger as they get farther away from the ALU.

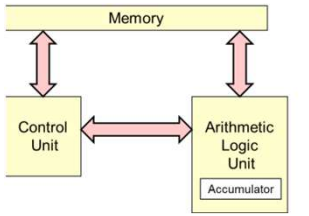
L3 cache also exists on some high-end CPU chips

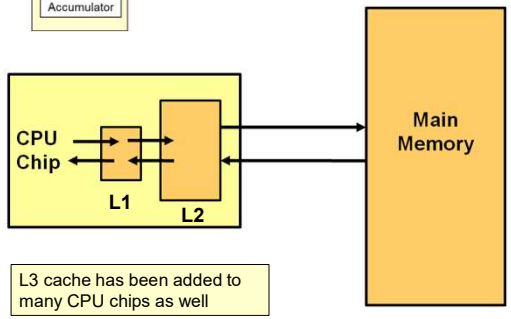
Oregon State University Computer Graphics

3

4

Cache and Memory are Named by "Distance Level" from the ALU





L3 cache has been added to many CPU chips as well

Oregon State University Computer Graphics

4

Storage Level Characteristics

	L1	L2	L3	Memory	Disk
Type of Storage	On-chip	On-chip	On-chip	Off-chip	Disk
Typical Size	100 KB	8 MB	32 MB	32 GB	Many GBs
Typical Access Time (ns)	.25	.50	10.8	50	5,000,000
Scaled Access Time	1 second	2 seconds	43 seconds	3.3 minutes	231 days
Managed by	Hardware	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"

Oregon State University
Computer Graphics

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.

Oregon State University
Computer Graphics

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."

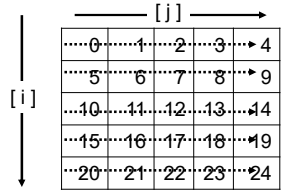
If these assumptions are true, then you will generate a lot of cache hits.

If these assumptions are not true, then you will generate a lot of cache misses, and you end up re-loading the cache a lot.

Oregon State University
Computer Graphics

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]:



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

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

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

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

Oregon State University
Computer Graphics

Demonstrating the Cache-Miss Problem – Across Rows

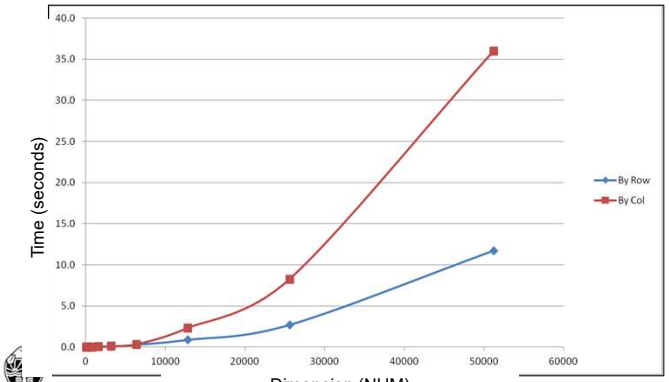

```
#define NUM  10000
float   Array[NUM][NUM];
double  MyTimer( );

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 – Down Columns


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


Demonstrating the Cache-Miss Problem

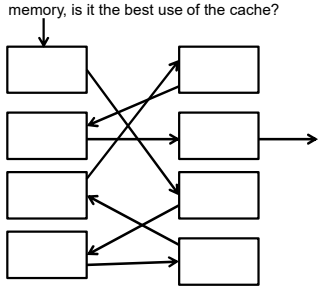
Time, in seconds, to compute the array sums, based on by-row versus by-column order:



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;
};
```

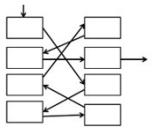
This is good OO style – it encapsulates and isolates the data for this class. Once you have created a linked list whose elements are all over memory, is it the best use of the cache?

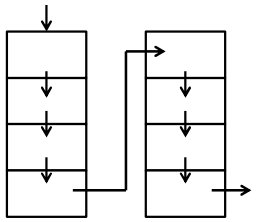


But, Here Is a Compromise:

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.

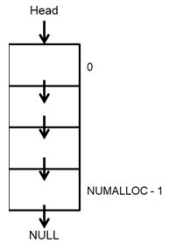




Oregon State University
Computer Graphics

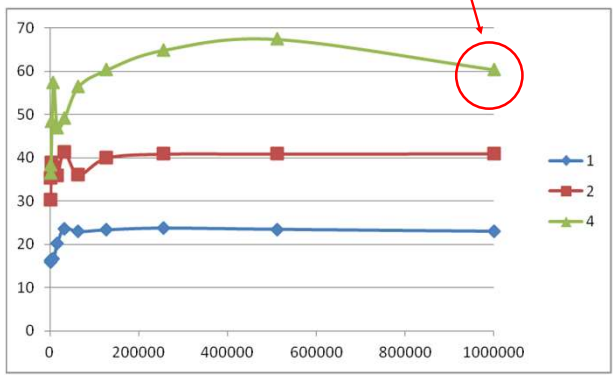
But, Here Is a Compromise:


```
#include <cstdio>
#define NUMALLOC        1024

struct node
{
        float data;
        bool canBeDeleted;
        struct node *next;
};
struct node *Head = NULL;


struct node *
GetNewNode( )
{
        if( Head == NULL )
        {
                struct node *array = new struct node[NUMALLOC];
                Head = &array[0];
                for( int i = 0; i < NUMALLOC - 1; i++ )
                {
                        array[i].canBeDeleted = false;
                        array[i].next = &array[i+1];
                }
                array[NUMALLOC-1].next = NULL;
        }
        struct node *p = Head;
        Head = Head->next;
        return p;
}

void
DeleteNode( struct node *n )
{
        n->canBeDeleted = true;
}
```



Remember: in this scheme, you cannot delete an individual node because it was allocated as part of an array. The best you can do is track which nodes can be deleted and then when all of an array's nodes are flagged, delete the whole array.

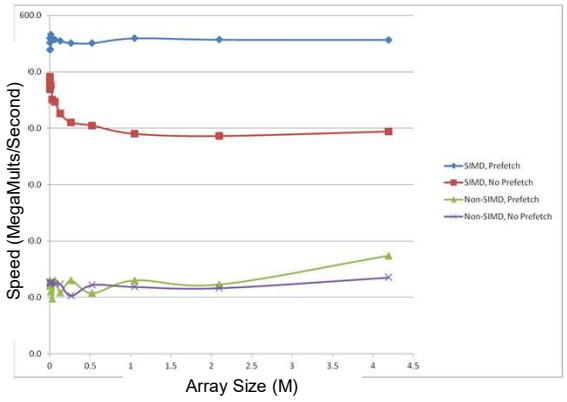
Why Can We Get This Kind of Performance Decrease as Data Sets Get Larger?



We are violating Temporal Coherence

Oregon State University
Computer Graphics

We Can Help the Temporal Problem with Pre-Fetching



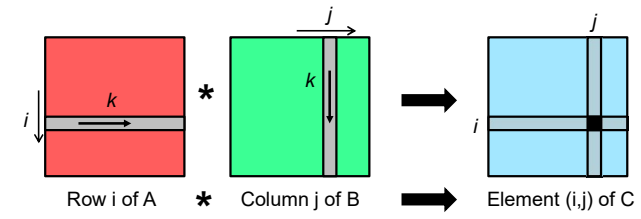
We will cover this in further detail when we discuss SIMD

Oregon State University
Computer Graphics

An Example of Where Cache Coherence Really Matters: Matrix Multiply

17

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




```
for( i = 0; i < SIZE; i++ )
    for( j = 0; j < SIZE; j++ )
        for( k = 0; k < SIZE; k++ )
            Σ A[ i ][ k ]  *  B[ k ][ j ]   Sum and store →  C[ i ][ j ]
```

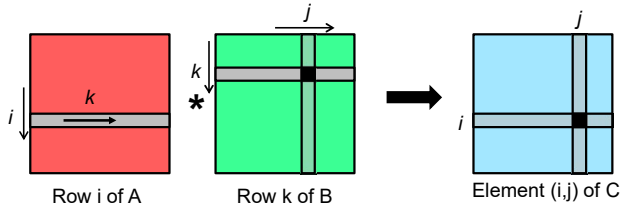
Problem: Column j of the B matrix is not doing a unit stride

17

An Example of Where Cache Coherence Really Matters: Matrix Multiply

18

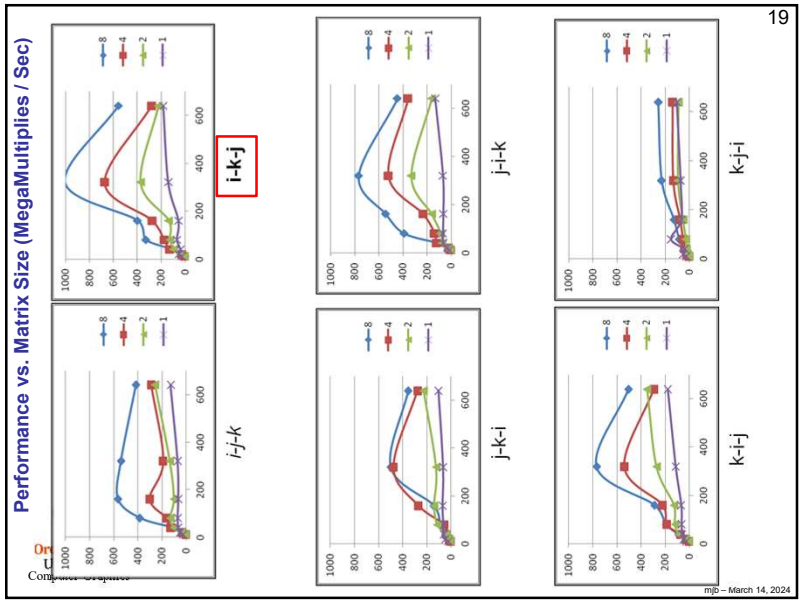
Scalable Universal Matrix Multiply Algorithm (SUMMA)
Entire A row * one element of B row
Equivalent to computing one item in many separate dot products




```
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 ]
```

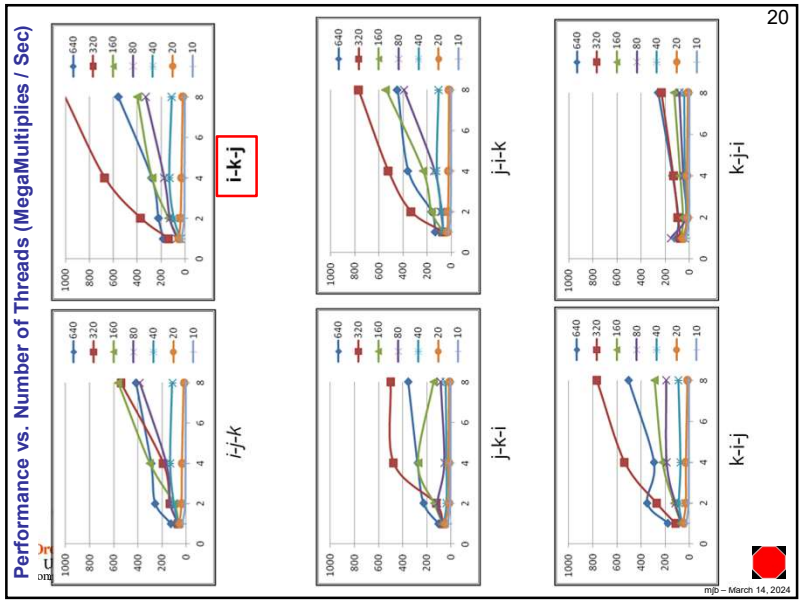

18

19



19

20



20

Cache Architectures

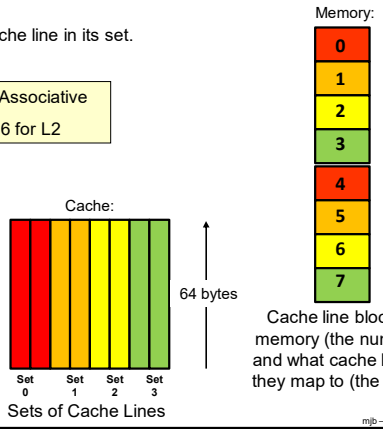
21

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"

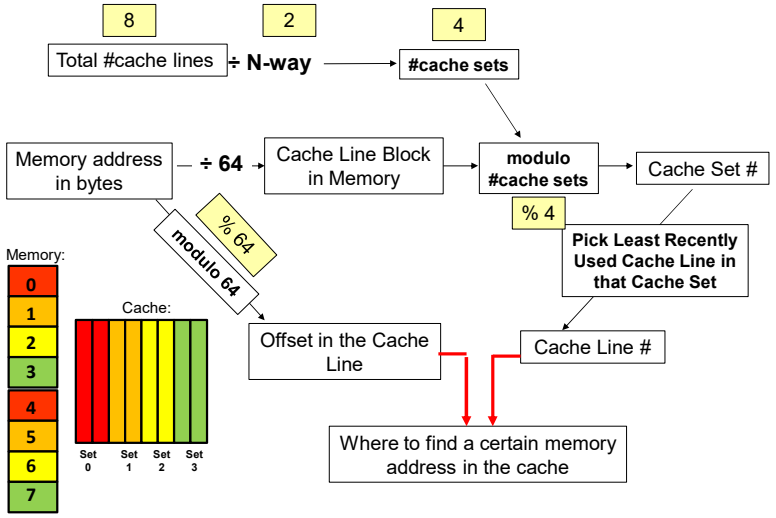
Oregon State University Computer Graphics

mjb – March 14, 2024

21

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

22



mjb – March 14, 2024

22

A Specific Example with Numbers

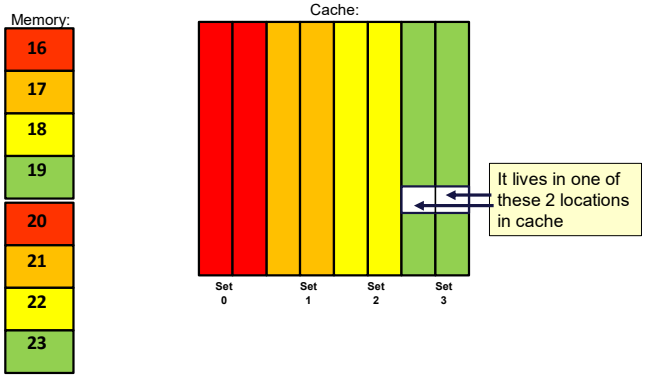
23

Memory address = 1234 bytes

Cache Line Block in Memory = 1234 / 64 =19 Because there are 64 bytes in a cache line

Cache Set # = 19 % 4 = 3 Because there are 4 sets to rotate through

Offset in the Cache Line = 1234 – 19*64 = **18** Because there are 18 bytes left after filling 19 complete cache lines



Oregon State University Computer Graphics

mjb – March 14, 2024

23

How Different Cores' Cache Lines Keep Track of Each Other

24

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

Oregon State University Computer Graphics

mjb – March 14, 2024

24

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**.
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**.

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

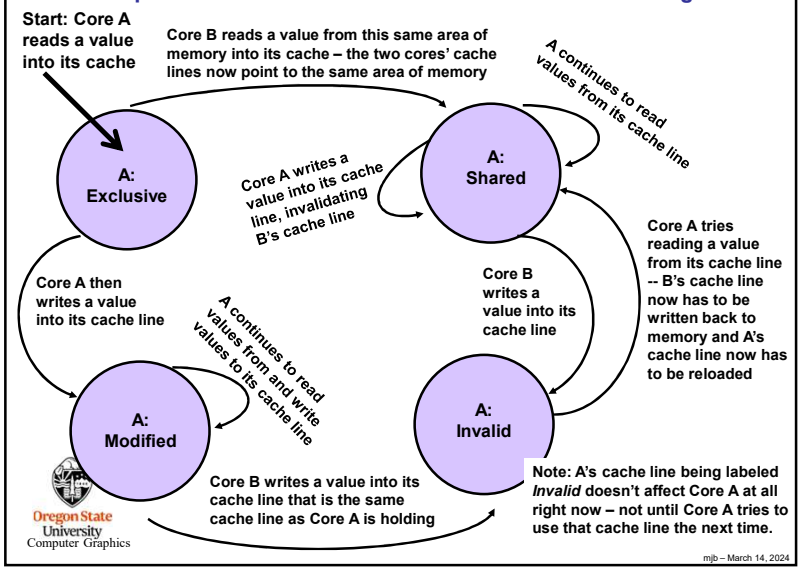
This is a huge performance hit, and is referred to as ***False Sharing***

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

Oregon State University Computer Graphics

mjb – March 14, 2024

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



Oregon State University Computer Graphics

mjb – March 14, 2024

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.

One cache line

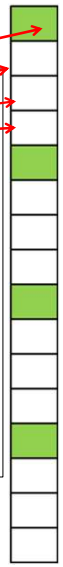
Oregon State University Computer Graphics

mjb – March 14, 2024

False Sharing – Fix #1 Adding some padding

```
#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( );
                }

        }
```

NUMPAD=3



One cache line

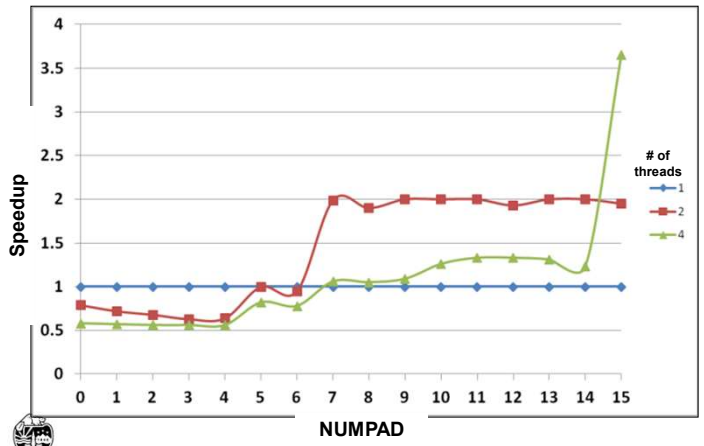
This works because successive Array elements are forced onto different cache lines, so less (or no) cache line conflicts exist

Oregon State University Computer Graphics

mjb – March 14, 2024

29

False Sharing – Fix #1

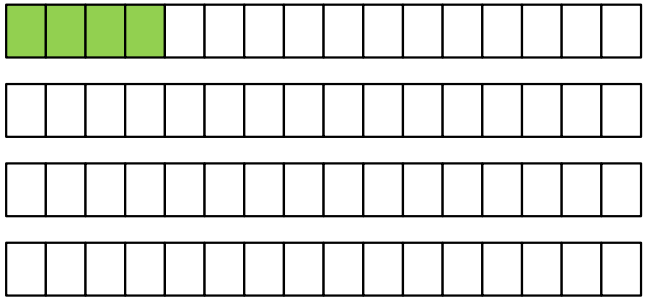


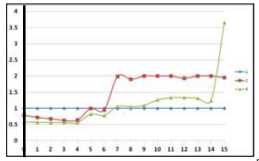
Why do these curves look this way?

30

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

NUMPAD = 0

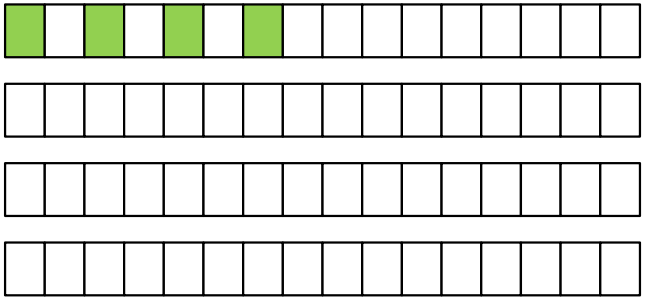


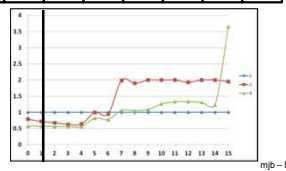


31

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

NUMPAD = 1

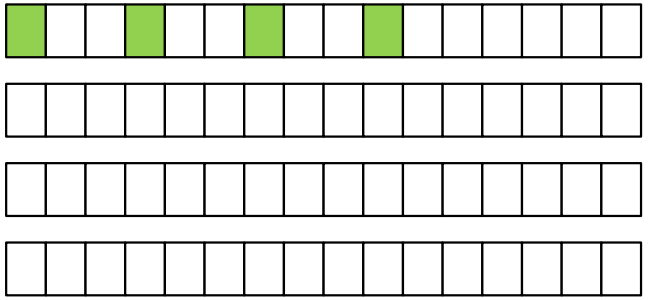


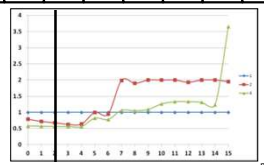


32

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

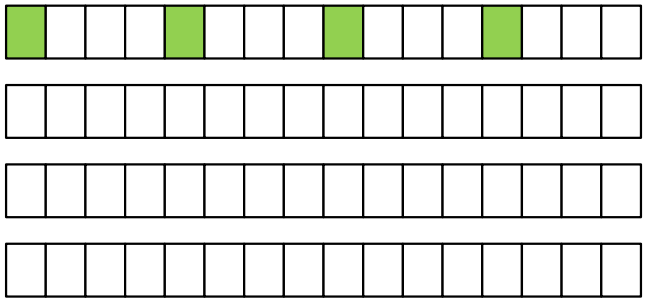
NUMPAD = 2

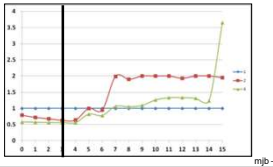




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 3

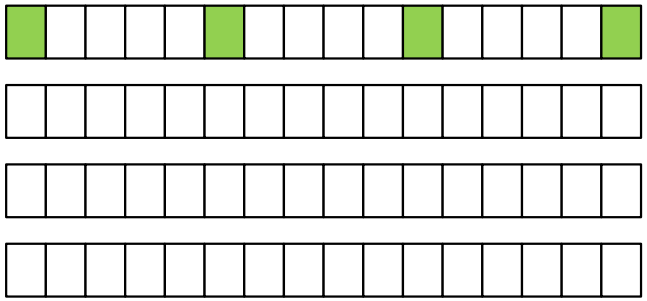


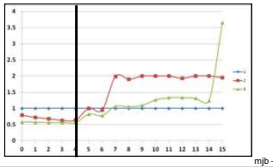


Oregon State University
Computer Graphics

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

NUMPAD = 4

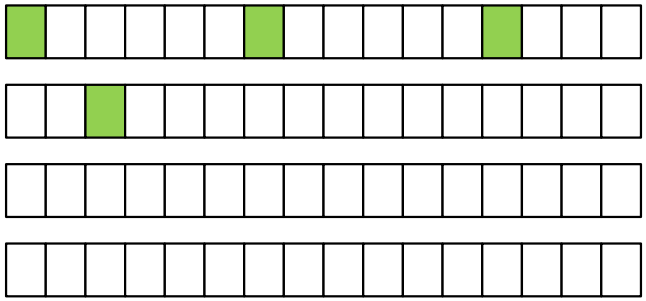


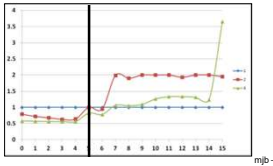


Oregon State University
Computer Graphics

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

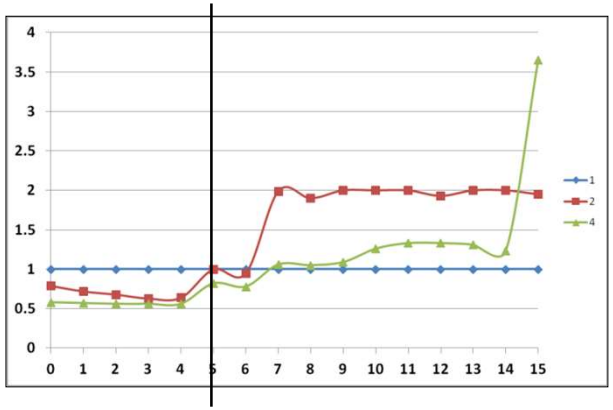
NUMPAD = 5





Oregon State University
Computer Graphics

False Sharing – Fix #1

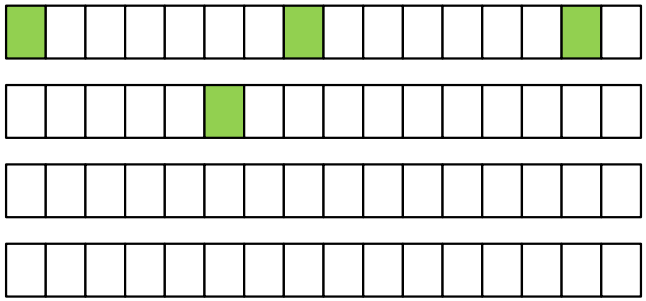


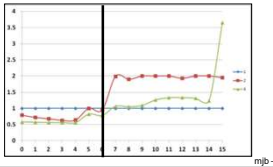
Oregon State University
Computer Graphics

37

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

NUMPAD = 6





Oregon State University
Computer Graphics

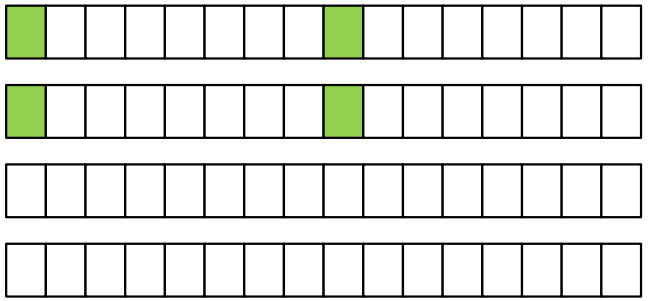
mjb – March 14, 2024

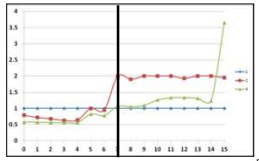
37

38

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

NUMPAD = 7





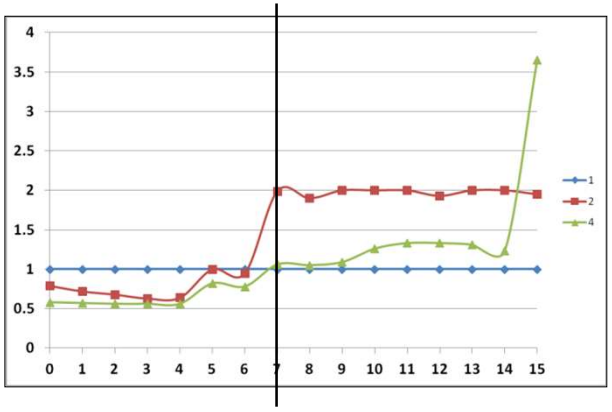
Oregon State University
Computer Graphics

mjb – March 14, 2024

38

39

False Sharing – Fix #1



Oregon State University
Computer Graphics

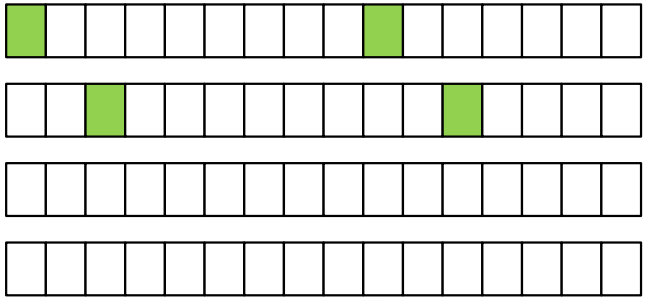
mjb – March 14, 2024

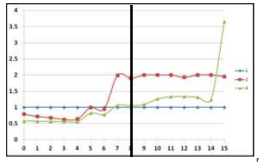
39

40

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

NUMPAD = 8





Oregon State University
Computer Graphics

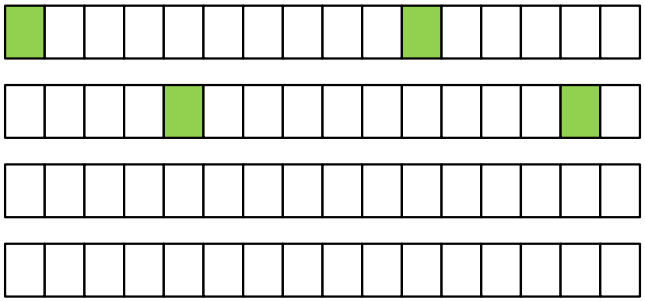
mjb – March 14, 2024

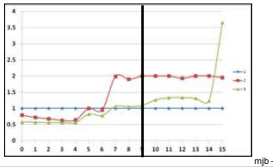
40

41

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

NUMPAD = 9





Oregon State University
Computer Graphics

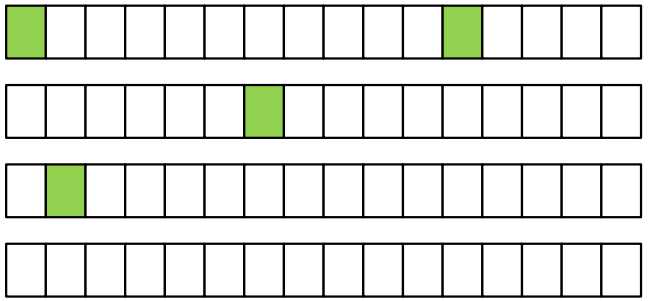
mjb – March 14, 2024

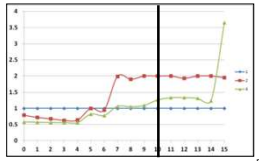
41

42

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

NUMPAD = 10





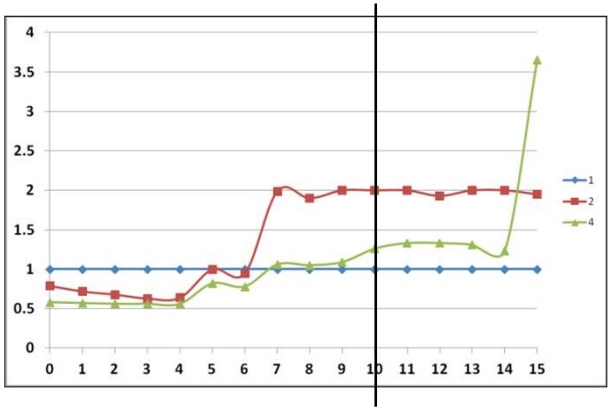
Oregon State University
Computer Graphics

mjb – March 14, 2024

42

43

False Sharing – Fix #1



Oregon State University
Computer Graphics

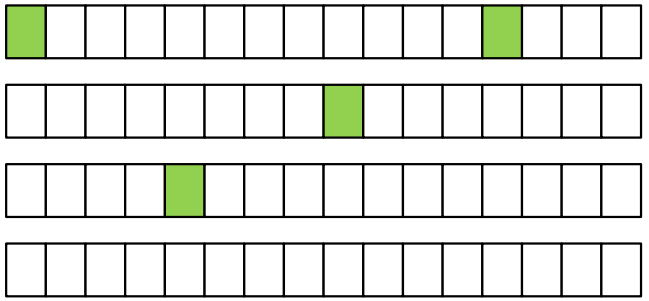
mjb – March 14, 2024

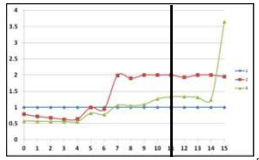
43

44

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

NUMPAD = 11





Oregon State University
Computer Graphics

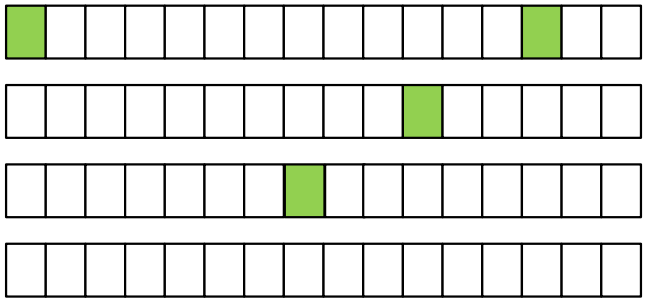
mjb – March 14, 2024

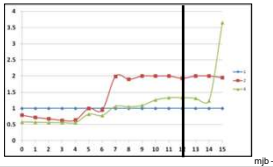
44

45

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

NUMPAD = 12





Oregon State University
Computer Graphics

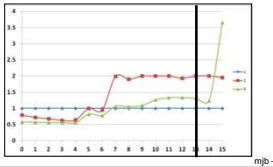
mjb – March 14, 2024

45

46

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

NUMPAD = 13



Oregon State University
Computer Graphics

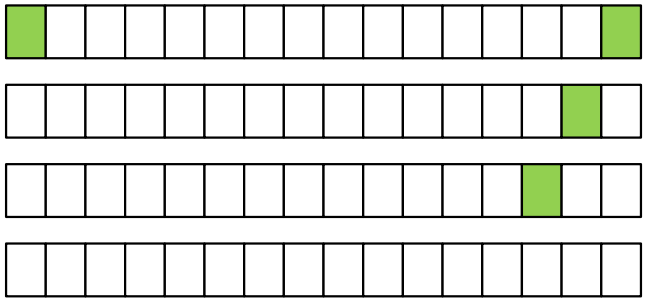
mjb – March 14, 2024

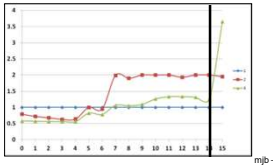
46

47

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

NUMPAD = 14





Oregon State University
Computer Graphics

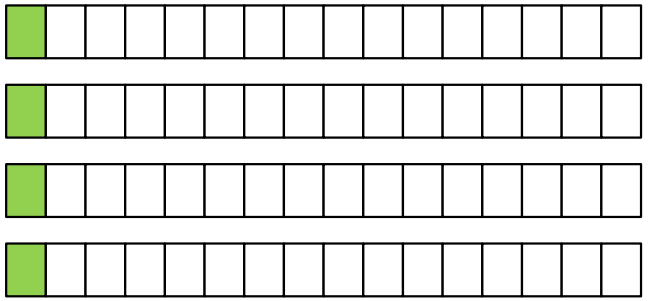
mjb – March 14, 2024

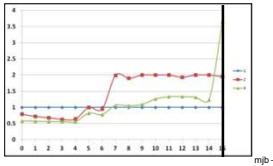
47

48

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

NUMPAD = 15





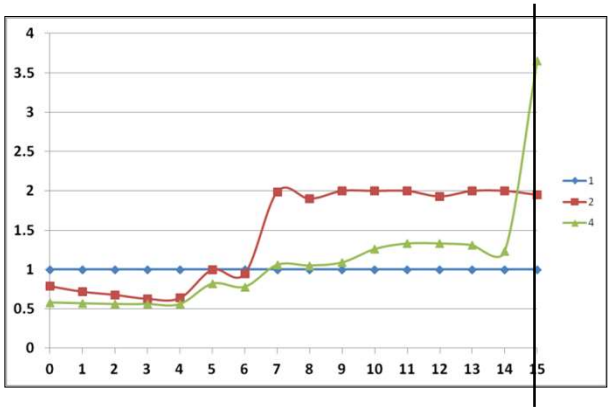
Oregon State University
Computer Graphics

mjb – March 14, 2024

48

49

False Sharing – Fix #1



Oregon State University Computer Graphics

mjb – March 14, 2024

49

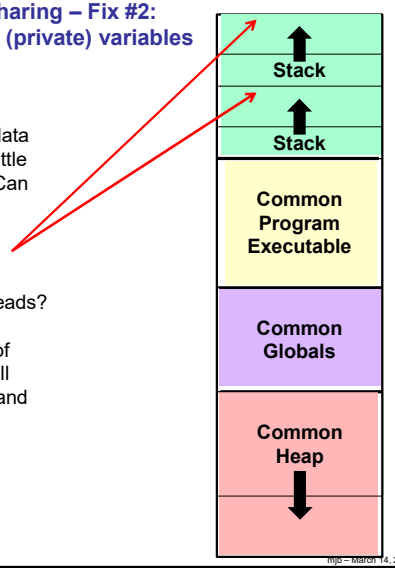
50

False Sharing – Fix #2: Using local (private) variables

OK, wasting memory to put your data on different cache lines seems a little silly (even though it works well). 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 – March 14, 2024

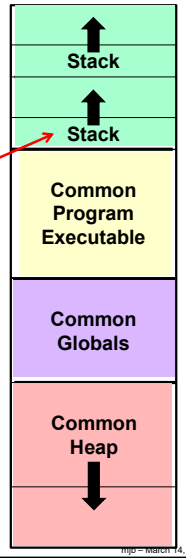
50

51

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

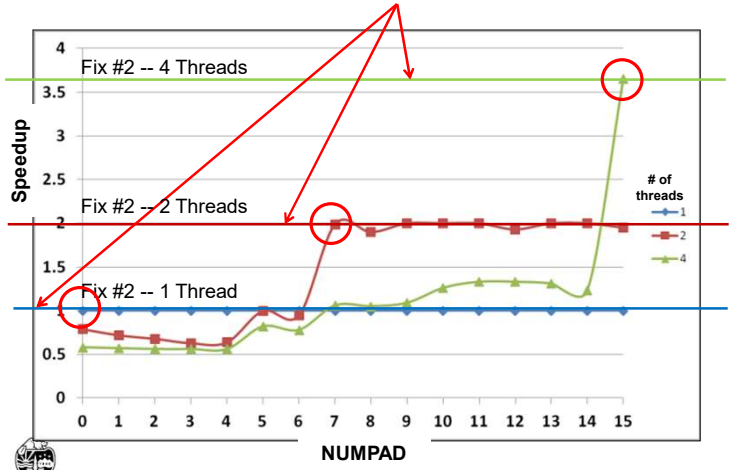
Oregon State University Computer Graphics

mjb – March 14, 2024

51

52

False Sharing – Fix #2 vs. Fix #1



Note that Fix #2 with {1, 2, 4} threads gives the same performance as NUMPAD= {0,7,15}

Oregon State University Computer Graphics

mjb – March 14, 2024

52

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 boundary?

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:

Cache line number	Offset in that cache line
32 - 6 = 26 bits	6 bits: 0-63

For example $\mathbf{101010_b}$ = 42

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

Oregon State University
Computer Graphics

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)  );
unsigned int offset = (unsigned int)p  &  0x3f;                    // 0x3f = bottom 6 bits are all 1's
struct xyzw *Array;
If( offset == 0 )
        Array = p;
Else
        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);

Oregon State University
Computer Graphics