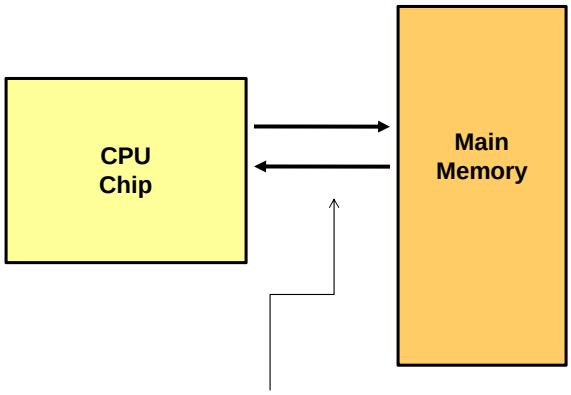
Caching Issues in Multicore Performance

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

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

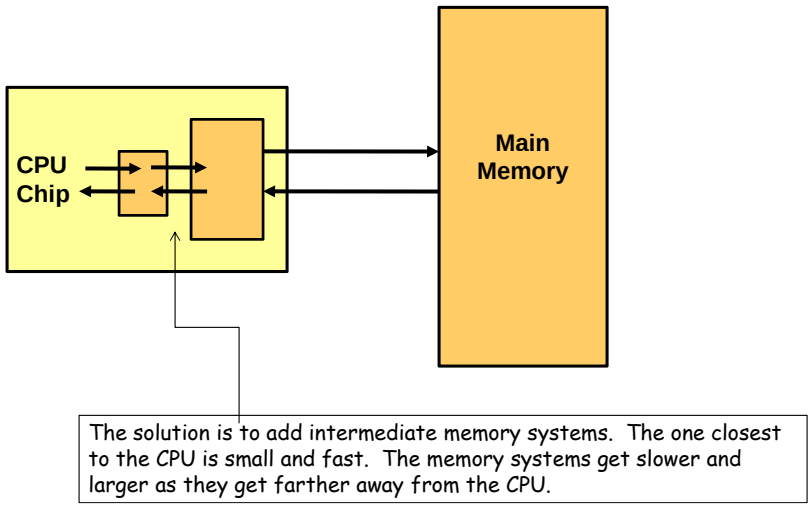


This path is relatively slow, forcing the CPU to wait for up to 200 clock cycles just to do a store to, or a load from, memory.

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"

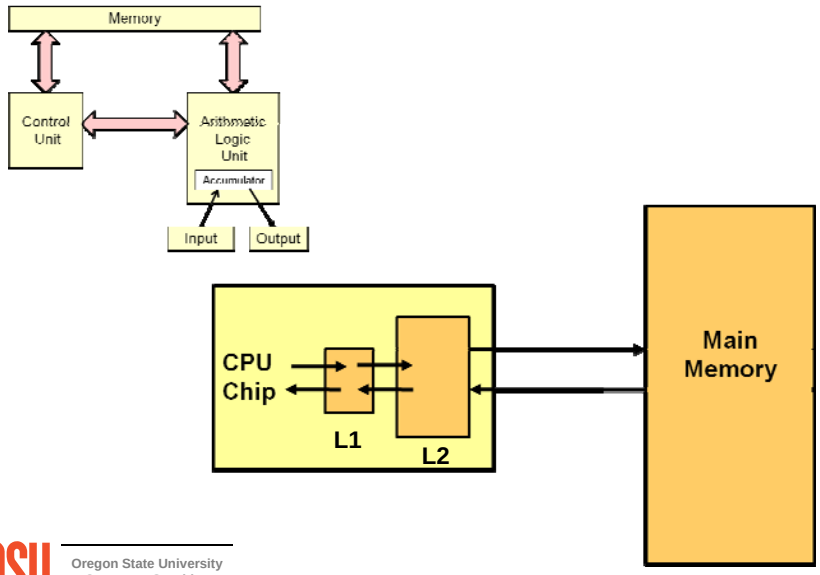


How is Cache Memory Actually Defined?

*In computer science, a **cache** is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (due to longer access time) or to compute, compared to the cost of reading the cache. In other words, a cache is a temporary storage area where frequently accessed data can be stored for rapid access. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or recomputing the original data, so that the average access time is shorter. Cache, therefore, helps expedite data access that the CPU would otherwise need to fetch from main memory.*

-- Wikipedia

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.

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 it again soon."

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

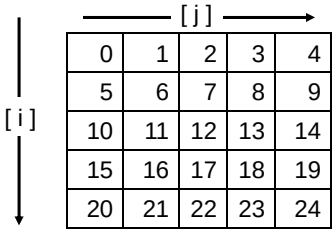
If these assumptions are false, then you will generate a lot of cache misses.

What Happens When the Cache is Full and a New Piece of Memory Needs to Come In?

1. **Random** – randomly pick a cache line to remove
2. **Least Recently Used** (LRU) – remove the cache line which has gone unaccessed the longest
3. **Oldest** (FIFO, First-In-First-Out) – remove the cache line that has been there the longest

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 ] ;`

Demonstrating the Cache-Miss Problem

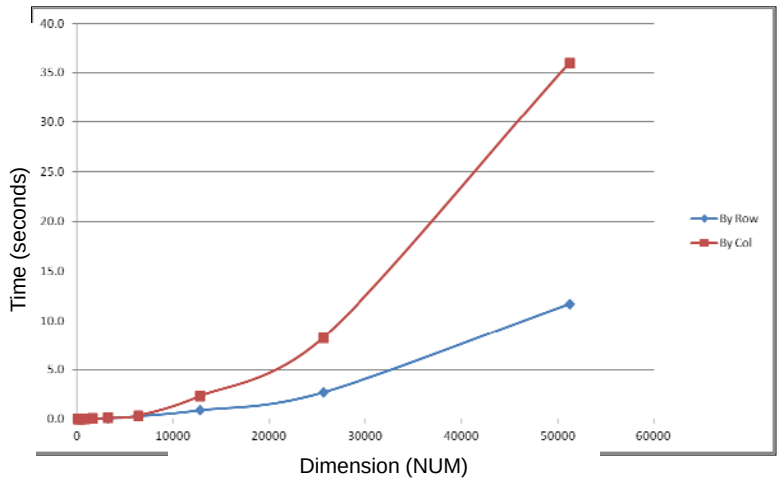
```
#include <stdio.h>
#include <ctime>
#include <cstdlib>

#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)

Welcome to the Hotel Cachifornia

There once was a country whose population was 1G people (1,073,741,824). That country was divided into 16M villages (16,777,216), each of which had 64 people.

People → Bytes of Memory
Villages → Cache Lines in Memory

The country decided to build a gambling center with a single huge hotel. The hotel would have a total of 32K rooms (32,768) arranged on some number of floors. They decided that a floor's rooms would not be individual rooms, but instead would be a suite with enough sleeping rooms to hold an entire village (64 people). This meant that there would have to be 512 floors ($\frac{32768}{64}$). It was also agreed that each suite could only be occupied by people from the *same* village at a time. A side effect of this was, though, that even if only 8 people from a village showed up, they would be given the entire floor.

Rooms → Bytes of L1 Cache Memory
Floor → A Cache Line in L1 Cache

This sounded great. Travel Agents started sending people to the hotel.

Travel Agents → Threads on cores

Welcome to the Hotel Cachifornia

At first, it was decided that any village could occupy any floor. This seemed fair.

Fully-Associative Cache strategy

Also, it was decided that when a village (or part of one) showed up at the hotel, and the hotel was full, that the floor that had been occupied the longest would have its occupants kicked out and would be given to the arriving village.

Oldest (FIFO) strategy

Finally, it was noticed that checking in any number of guests took a fair amount of time, so that this was something one didn't want to do any more often than necessary.

Check-in time → Latency of moving memory into L1 cache

Welcome to the Hotel Cachifornia

After a while, it became apparent that things weren't working as well as they could be. Guests from some villages wanted to be in town much longer than others. But, they got kicked off their floor when enough short-term guests arrived, and had to go outside and get back in line to check-in again.

So, a new strategy was devised. Certain floors would be reserved only for certain villages. That way, long-term villages could keep their room longer. This meant that with 16M villages and 512 floors, each floor would have to be time-shared by 32,768 villages ($\frac{16M}{512}$).

Direct-mapping Cache strategy

Floor # 0 would only be for Villages 0, 512, 1024, …
Floor # 1 would only be for Villages 1, 513, 1025, …
Floor #511 would only be for Villages 511, 1023, 1535, …

← Note: "Village Stride" = 512

An alert student noticed that you could figure this out by:

Floor # = (Village #) % 512 = (Village #) & 0x1ff

While it was recognized that conflicts could arise, it was thought that they wouldn't come up very often.

Welcome to the Hotel Cachifornia

This plan worked a lot of the time. However, conflicts did arise because of "Village Stride". One day, guests from villages 0, 512, and 1024 all showed up. By rule, each of those villages could only be placed in the suite on Floor #0, even though there were other floors unoccupied. Village 0 arrived in the morning and were placed in Floor #0. Village 512 arrived at noon, and by rule, Village 0 was kicked out off the floor and the floor was given to Village 512. Village 1024 arrived in the afternoon, and, well, you can guess what happened.

So, another strategy was devised. Certain floors were still reserved only for certain villages. But, each group of villages was given *4* floors reserved for them instead of one. This meant that the 512 floors were being grouped into 128 sets ($\frac{512}{4}$) of 4 floors each. With only 1/4 the number of floor sets available, each set had to be shared by 4 times as many villages. Instead of being time-shared by 32K villages (32,768), each floor set would be time-shared by 128K villages (131,072).

4-Way Set Associative Cache strategy

Floor Sets → Cache Sets

Note: "Village Stride" = 128

Floor Set # 0 = Floors # 0- 3 would now be for Villages 0, 128, 256, 384, …
Floor Set # 1 = Floors # 4- 7 would now be for Villages 1, 129, 257, 385, …
Floor Set #127 = Floors #507-511 would now be for Villages 127, 255, 383, 511, …

An alert student noticed that you could figure this out by:

Floor Set # = (Village #) % 128 = (Village #) & 0x7f

Welcome to the Hotel Cachifornia

Notice that this hotel scheme makes a good use of resources in two circumstances:

1. When you come to the hotel, you bring your whole village — Spatial Coherence
2. When you come to the hotel, you stay a while — Temporal Coherence

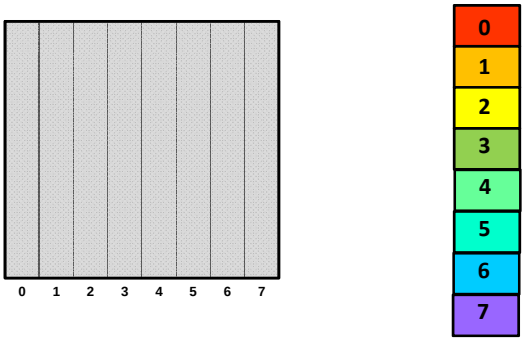
If you come by yourself, or come with a friend from another village, you are wasting hotel resources.

If your village needs to come the same day as several other groups whose villages map to the same floor set as yours, you will have a conflict for the floor. — Cache Stride, Conflict Misses

If you come for a few minutes, admire the view, and then leave, you are wasting hotel resources. — Capacity Misses

Possible Cache Architectures

1. Fully Associative – cache lines from any block of memory can appear anywhere in cache.



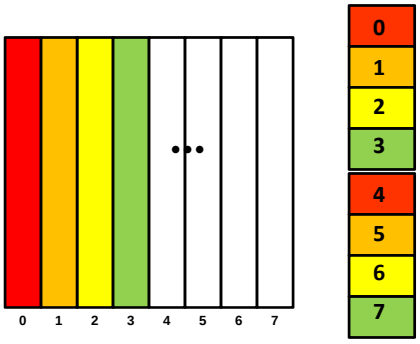
Possible Cache Architectures

2. Direct Mapped – a cache line from a particular block of memory has only one place it could appear in cache. A memory block's cache line is:

Cache line # = Memory block # % # lines the cache has

In the hotel-story case:

Cache line # = (Memory block #) % 512



Possible Cache Architectures

3. 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. A memory block's set number is:

Set # = (Memory block #) % (# sets the cache has)

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

Note that Direct Mapped can be thought of as 1-Way Set Associative.
Note that Fully Associative can be thought of as 1-Set Set-Associative.

What is the Complaint About Cache in Some Gaming Systems?

A function jump table that *you* setup requires two instruction cache lines:

1. The for-loop statements looping through the data
2. The function that really gets called

and one data cache line:

1. Your *funcptr* array

A **virtual function table**, or **vtable**, that the *compiler* sets up requires three instruction cache lines:

1. The for-loop statements looping through the data
2. The f( ) function that really gets called
3. The virtual function table

The Sony PS2 and PSP have **2-way** set associative caches. If all of these 3 items are in the same memory block, they will attempt to use the same cache set. But, there are only two cache lines in each cache set. So, every pass through the for-loop will cause a cache miss.

(Other gaming consoles have 8-way set associative cache, so there is no similar problem there.)

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"

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
. . .

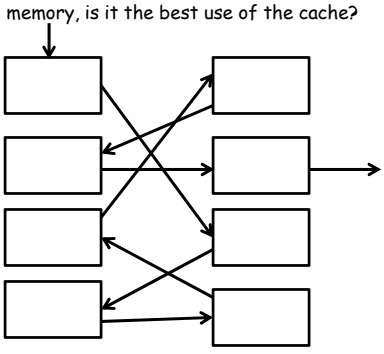
Z0
Z1
Z2
Z3
. . .

```
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?



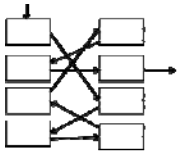
It might be better to create a large array of xyz structures and then have the constructor method pull new ones from that list. That would keep many of the elements close together while preserving the flexibility of the linked list.

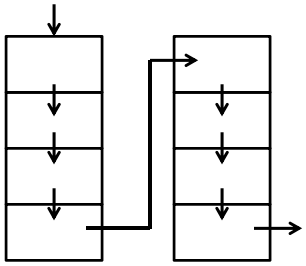
When you need more, allocate another large array and link to it.

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

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.





Cache Can Interact with Cores in Unexpected Ways

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

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

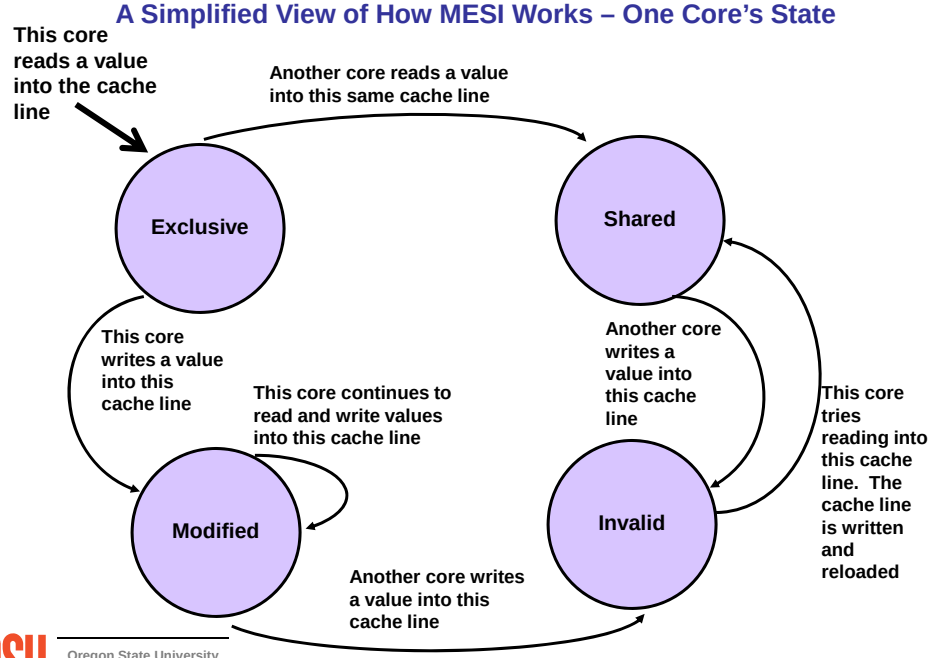
1. Core A reads a value. Those values are brought into its cache. That cache line is now tagged **Exclusive**.
2. Core B reads a value from the same area of memory. Those values are brought into its cache, and now both cache lines are re-tagged **Shared**.
3. If Core B writes into that value. Its cache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

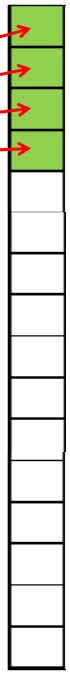
Note that False Sharing doesn't cause incorrect results – just a performance hit.

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



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  +  2.;
                }

        }
```



One cache line

Array[0].value, Array[1].value, Array[2].value, and Array[3].value are likely all in the same cache line

False Sharing – Fix #1

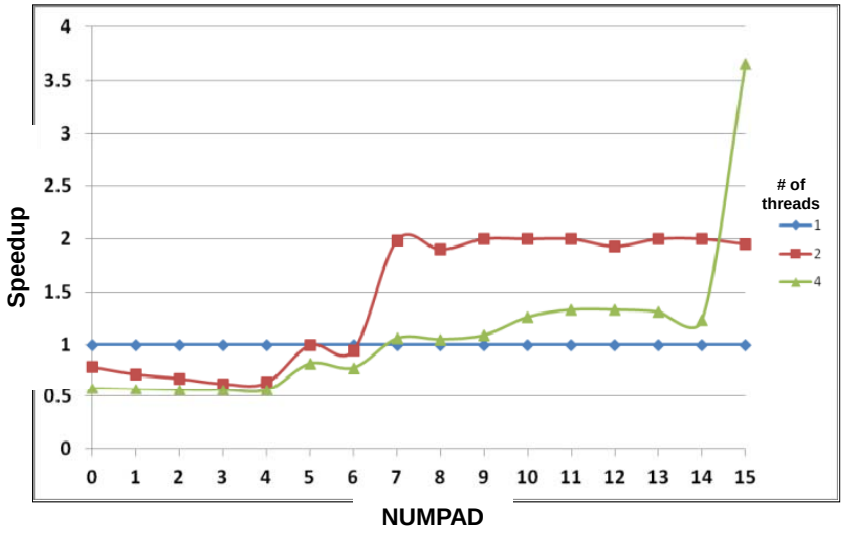
```
struct s
{
        float value;
        int pad[NUMPAD];
}   Array[4];

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  +  2.;
                }

        }
```

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

False Sharing – Fix #1



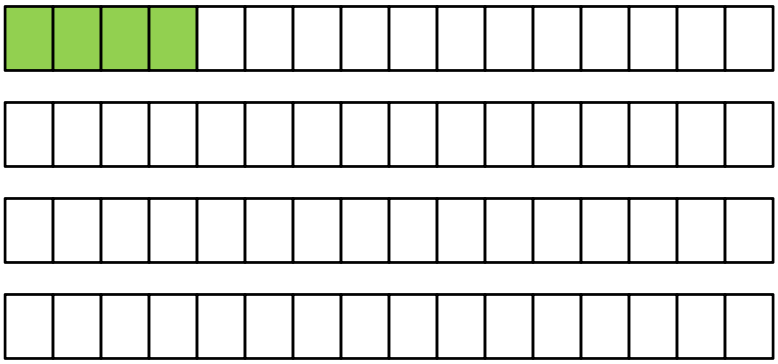
Oregon State University
Computer Graphics

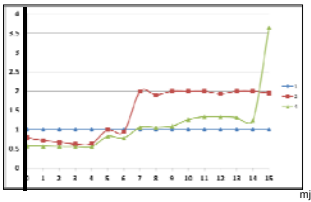
Why do these curves look this way?

mjb – February 9, 2015

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

NUMPAD = 0



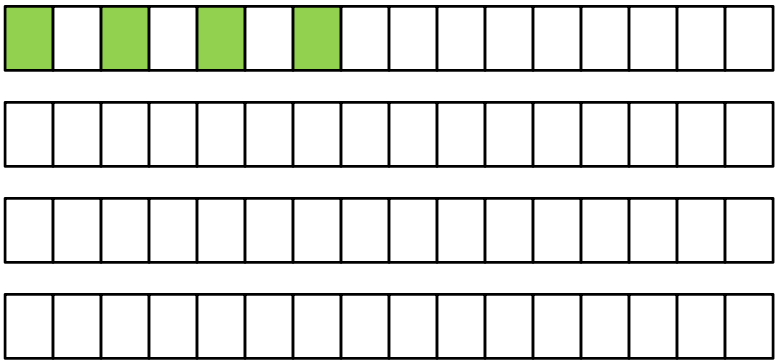


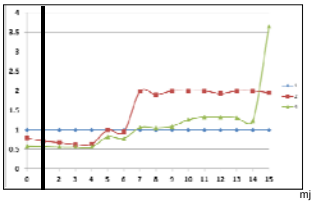
Oregon State University
Computer Graphics

mjb – February 9, 2015

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

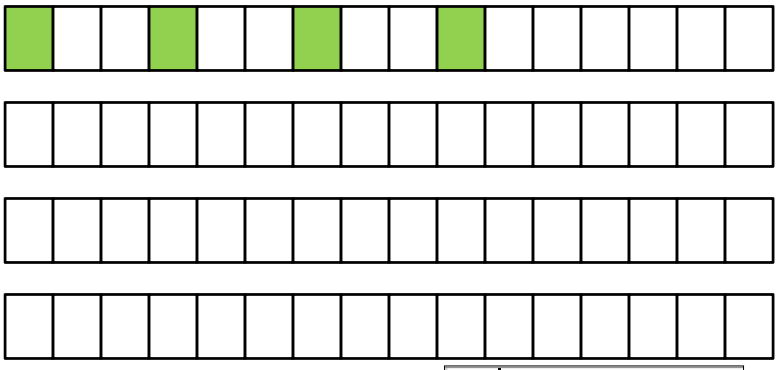
NUMPAD = 1

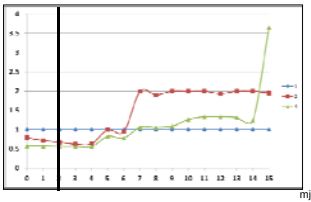




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

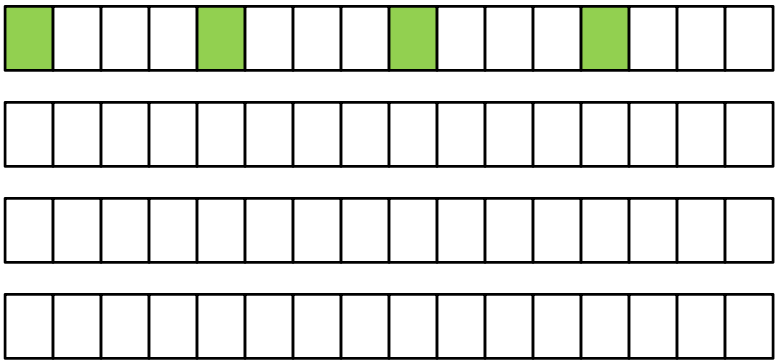
NUMPAD = 2

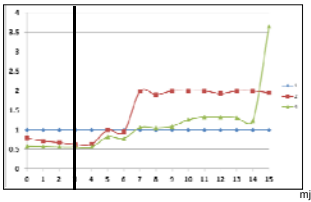




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

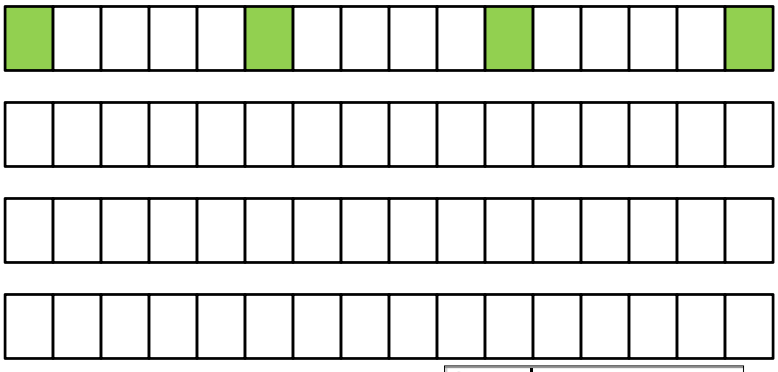
NUMPAD = 3

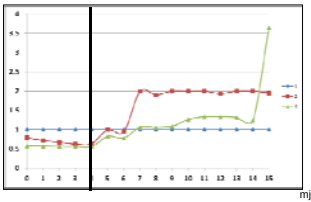




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

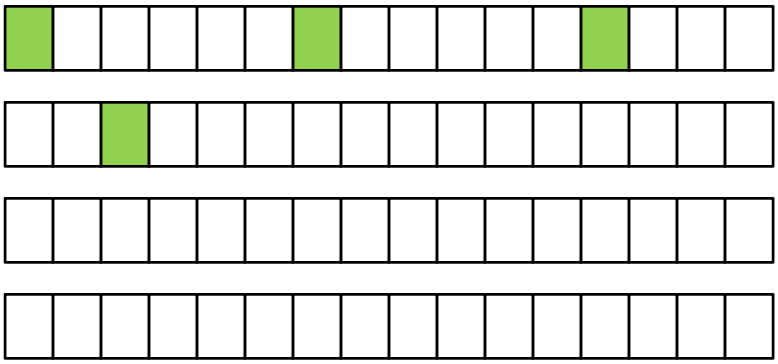
NUMPAD = 4

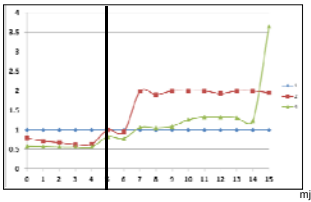




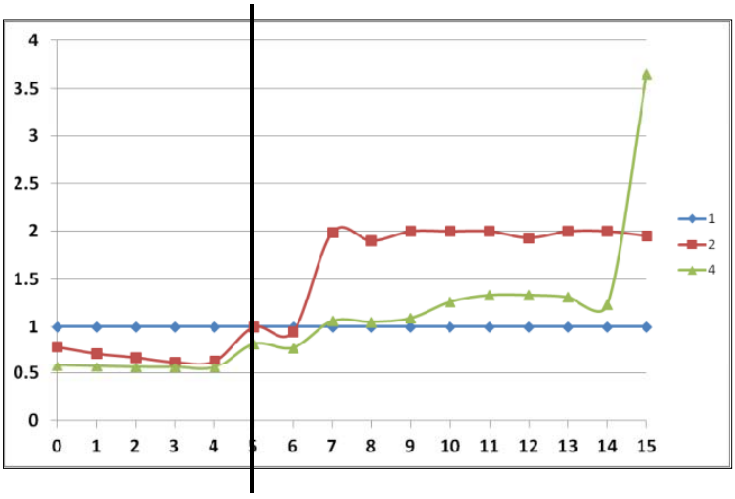
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 5



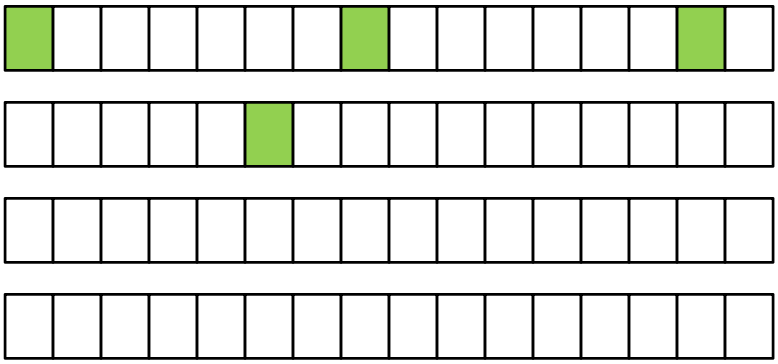


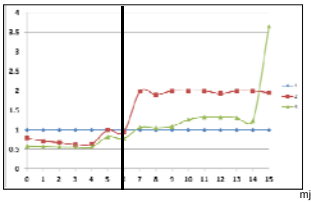
False Sharing – Fix #1



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

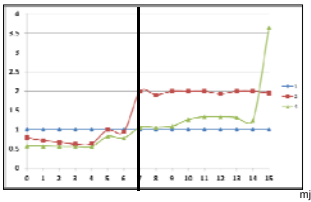
NUMPAD = 6



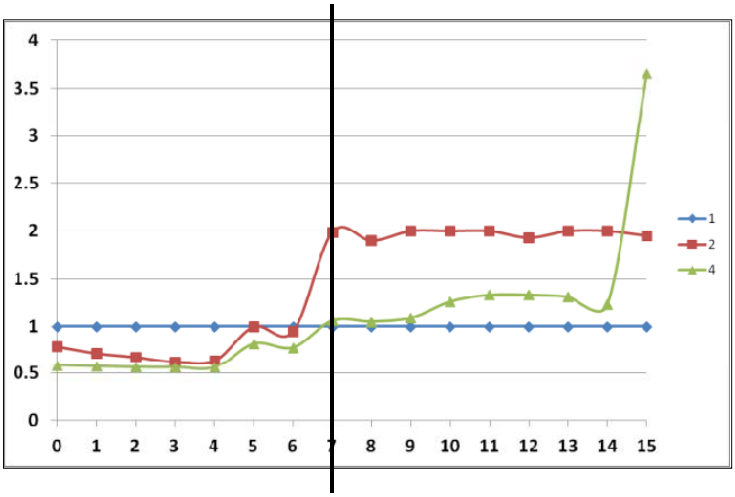


False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 7

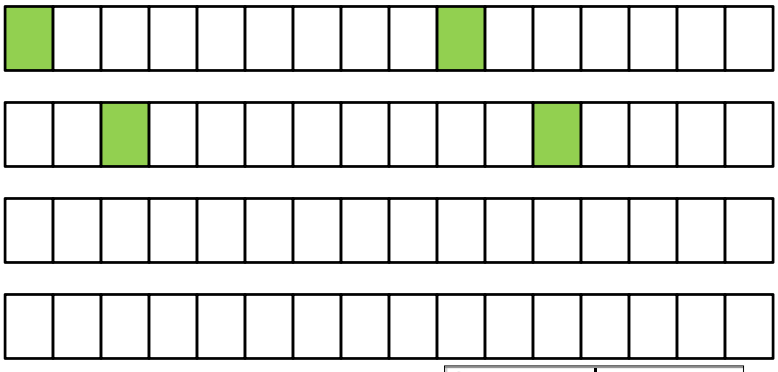


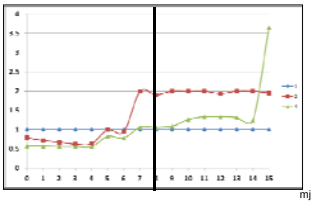
False Sharing – Fix #1



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

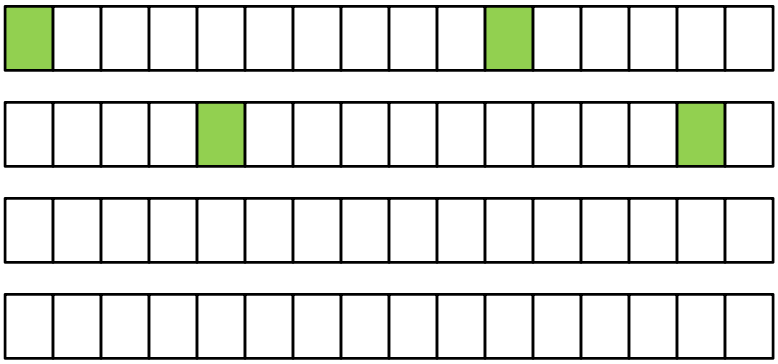
NUMPAD = 8

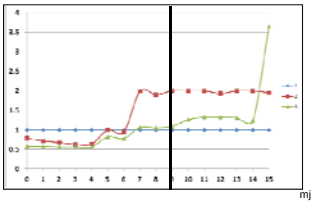




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

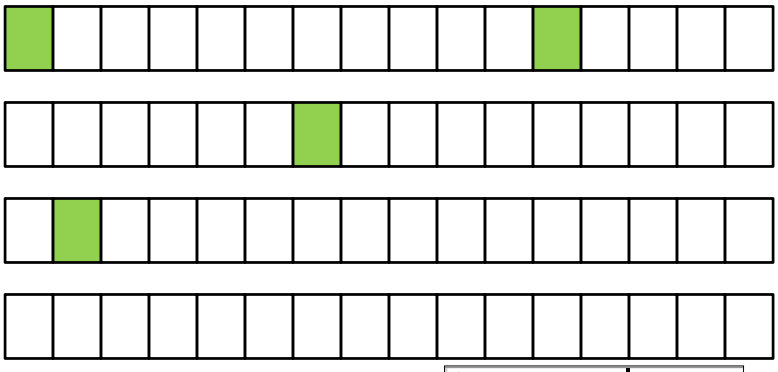
NUMPAD = 9

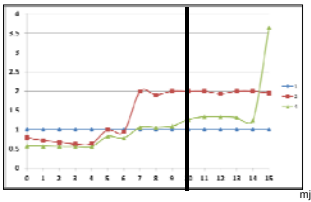




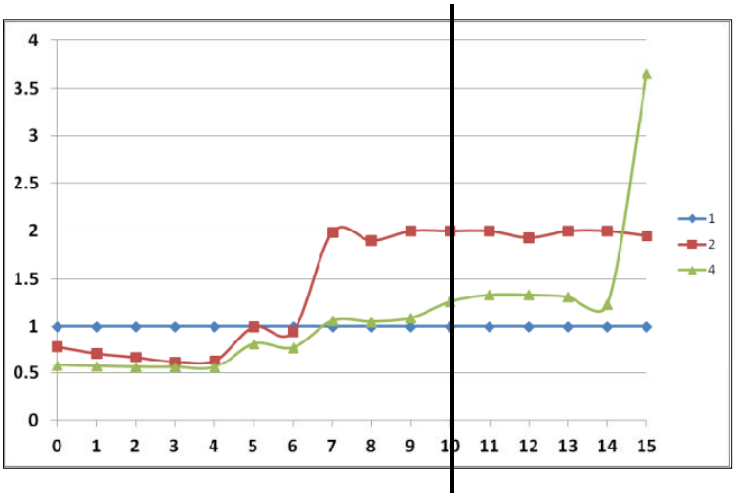
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 10



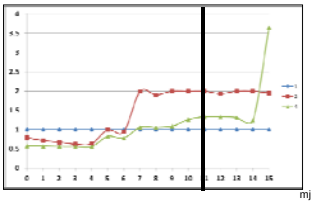


False Sharing – Fix #1



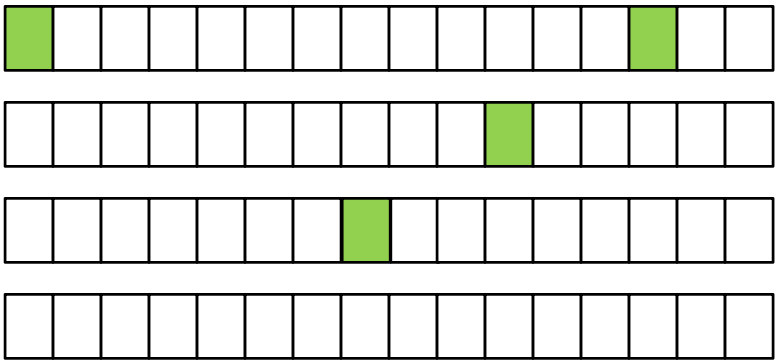
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

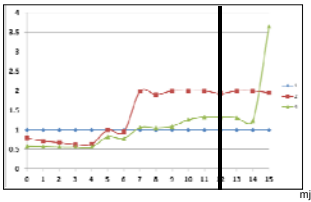
NUMPAD = 11



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

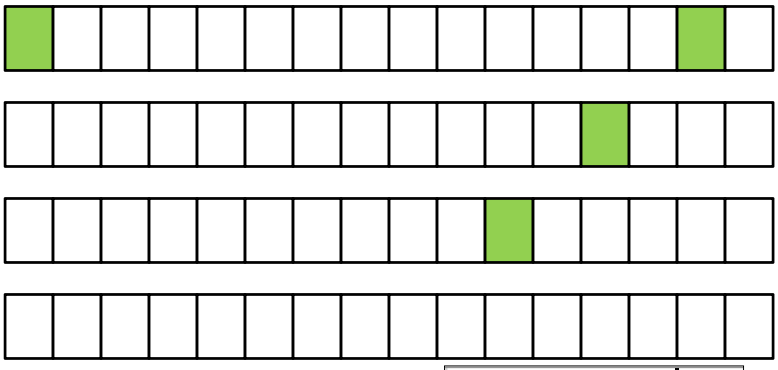
NUMPAD = 12

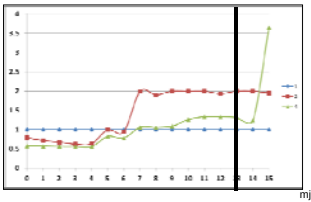




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

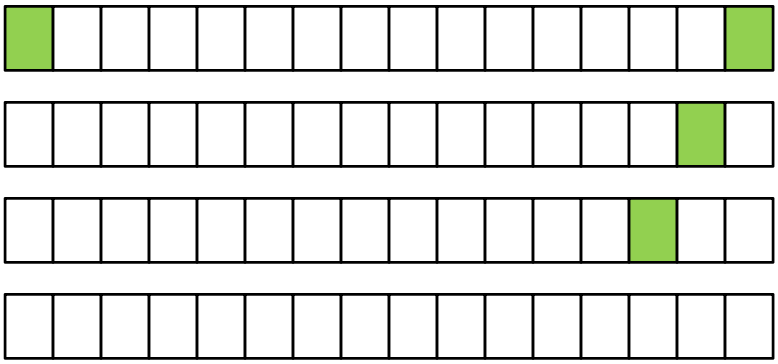
NUMPAD = 13

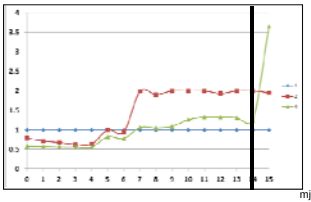




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

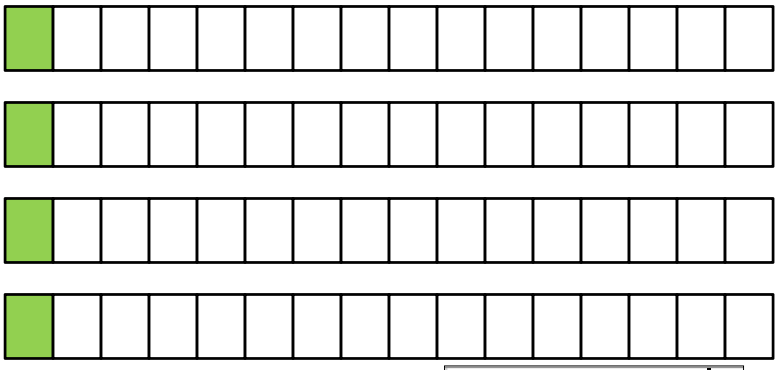
NUMPAD = 14

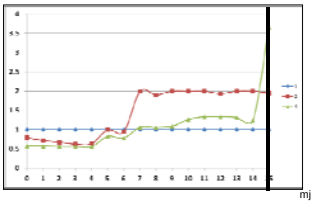




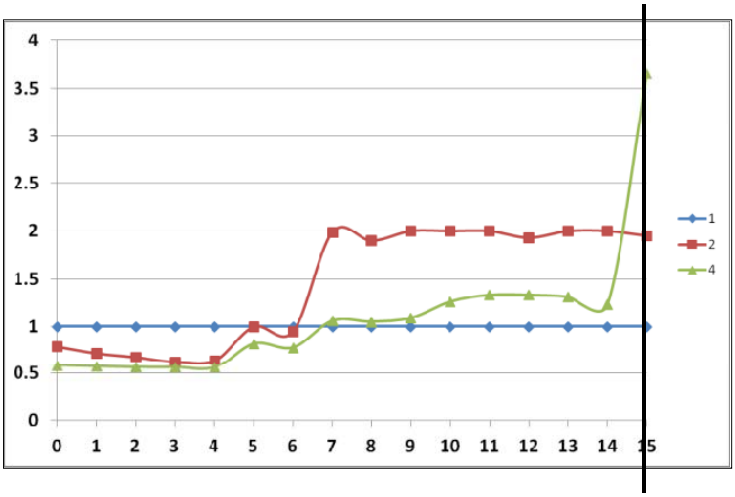
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 15





False Sharing – Fix #1

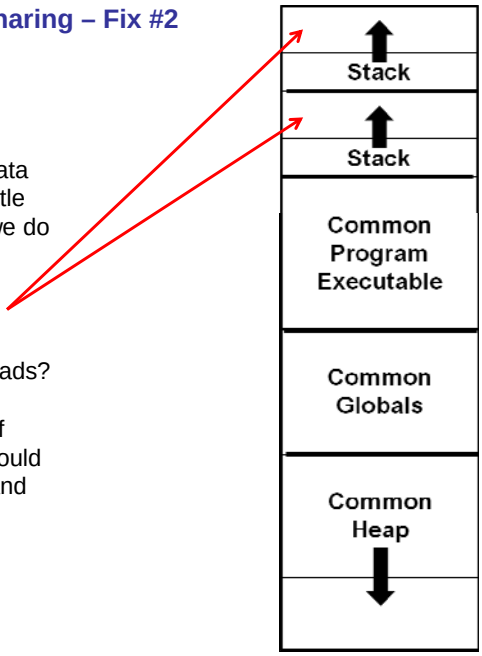


False Sharing – Fix #2

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 should spread our writes out in memory, and thus to different cache lines.



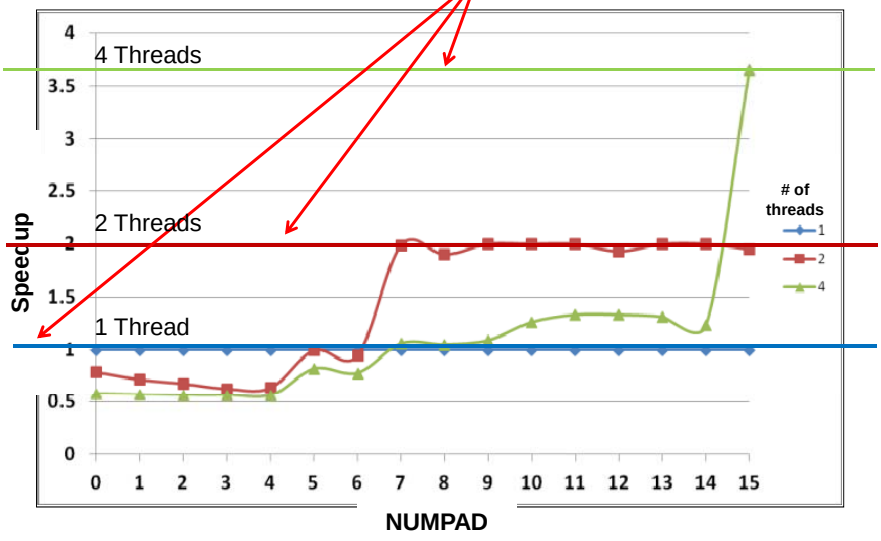
False Sharing – Fix #2

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

omp_set_num_threads(  4 );

#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 +  2.;
                }
                Array[ i ].value = tmp;
        }
```

Makes this a private variable

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



Fix #2 gives the same performance as NUMPAD=15