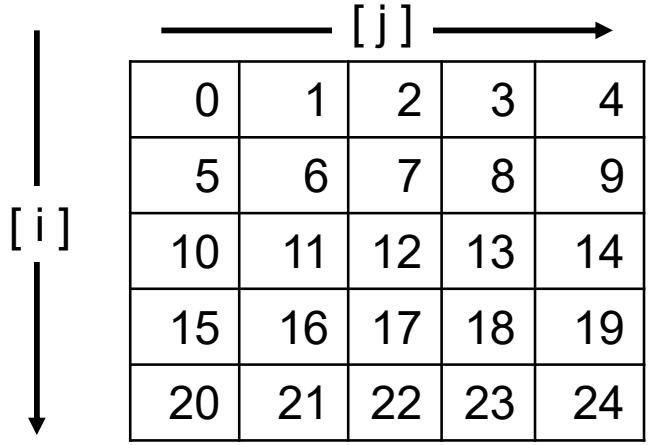
How Bad Is It? -- Demonstrating the Cache-Miss Problem

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 – Across Rows

```
#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;
```

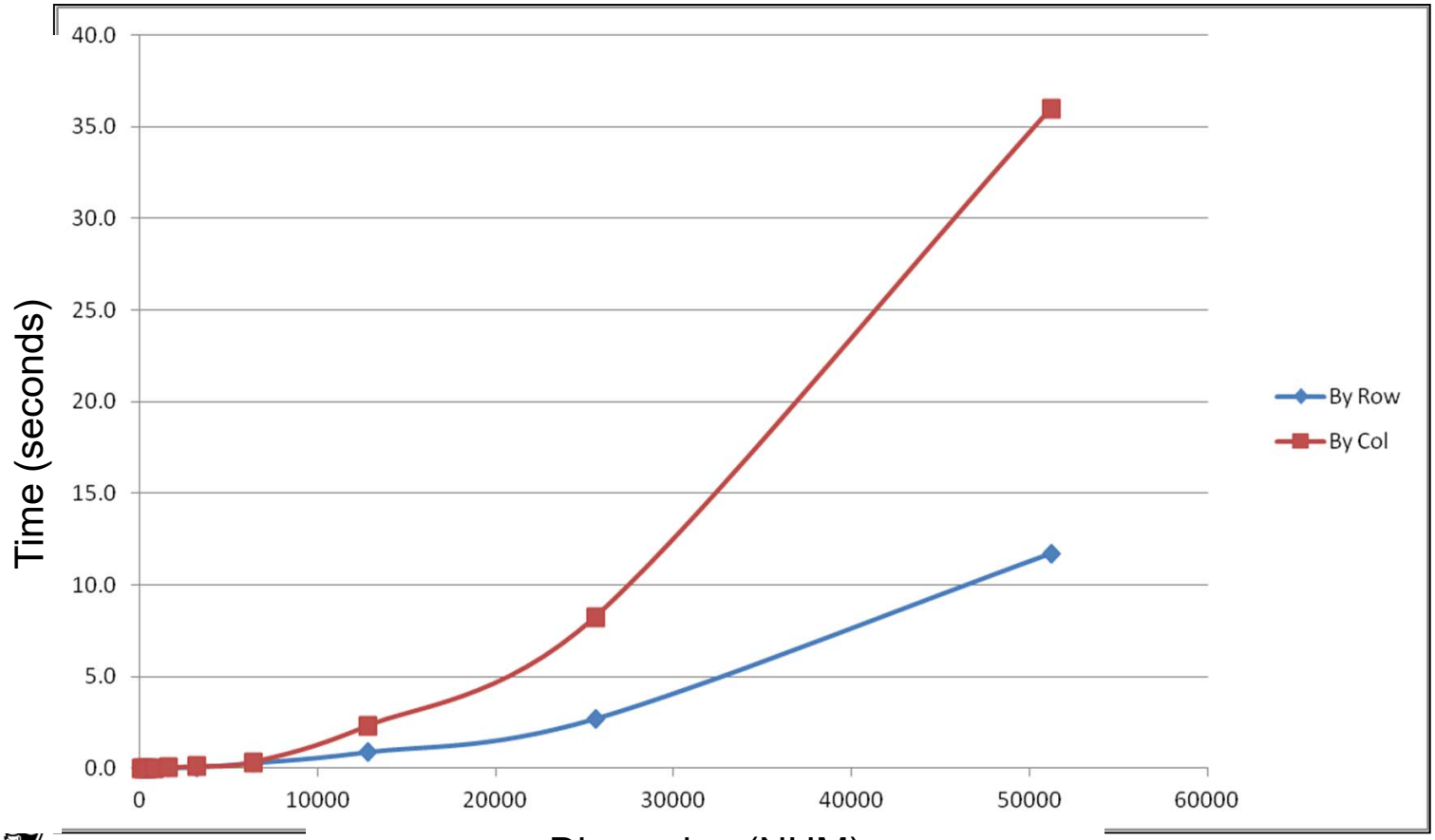
Demonstrating the Cache-Miss Problem – Down Columns

```
	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 );
}
```

Oregon State University
Computer Graphics

Demonstrating the Cache-Miss Problem

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



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];
```

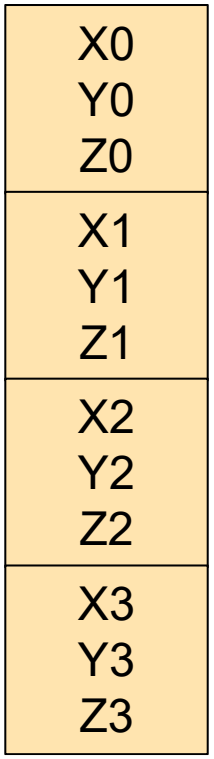
X0 X1 X2 X3 . . .
Y0 Y1 Y2 Y3 . . .
Z0 Z1 Z2 Z3 . . .

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?

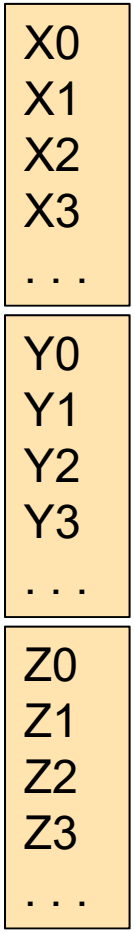
I've seen some programs use a "Shadow Data Structure" to get the advantages of both AOS and SOA

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




```
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( );
```

A Good Use for Structure-of-Arrays:

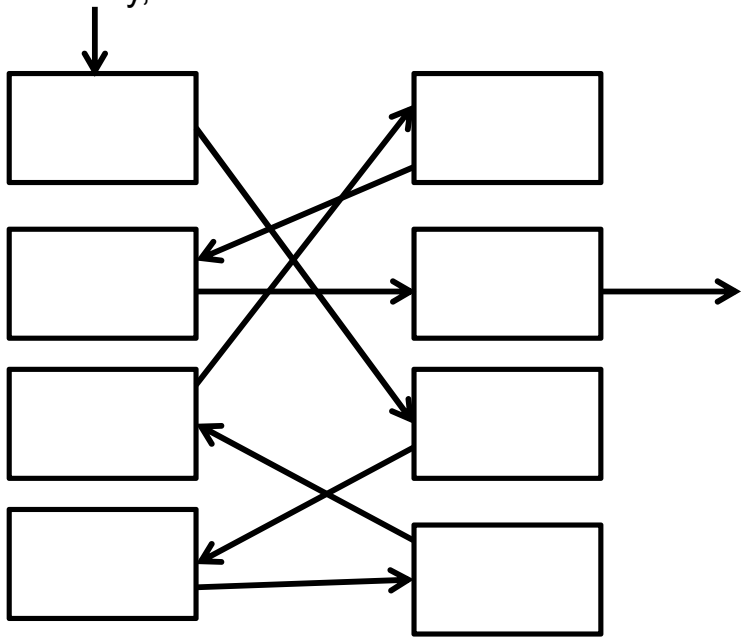



```
float X[N], Y[N], Z[N];
float Dx[N], Dy[N], Dz[N];
. . .

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;
};
```

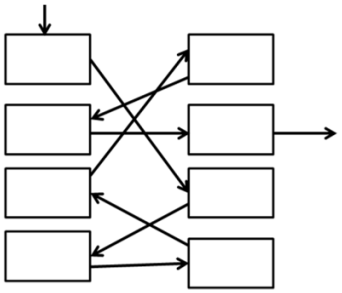
This is good OO style – it encapsulates and isolates the data for this class. Once you have created a linked list whose elements are all over memory, is it the best use of the cache?

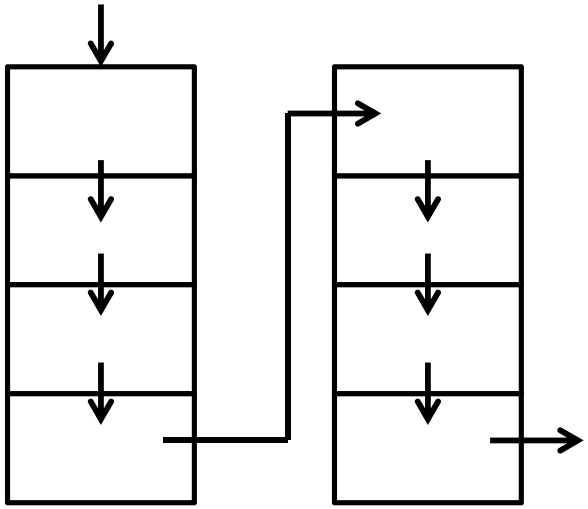


But, Here Is a Compromise:

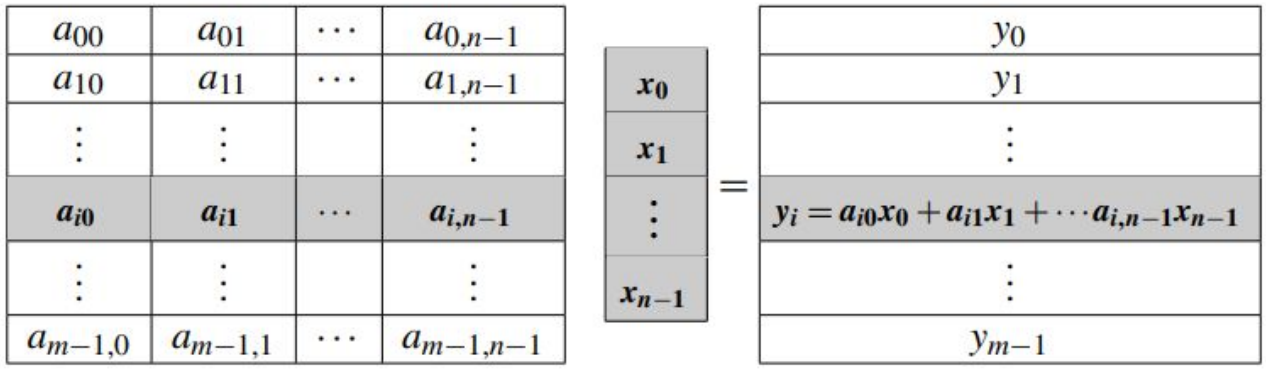
It might be better to create a large array of xyz structures and then have the constructor method pull new ones from that list. That would keep many of the elements close together while preserving the flexibility of the linked list.

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





Matrix vector multiplication




```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

Performance numbers

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

Observation 1

Threads	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Explanation 1

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

- A **write-miss** occurs when a core tries to update a variable that's not in cache, and it has to access the main memory
- 8,000,000 x 8 shows more cache write-misses than either of the other inputs
- Bulk of these occur in Line 4
- Since the number of elements in the vector y is far greater in this case (8,000,000 vs. 8000 or 8), and each element must be initialized, so line 4 slows down the execution of the program with the 8,000,000 × 8 input

Observation 2

Threads	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Explanation 2

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

- A **read-miss** occurs when a core tries to read a variable that's not in cache, and it has to access main memory
- 8 x 8,000,000 shows more cache read-misses than either of the other inputs
- Bulk of these occur in Line 6
- for this matrix dimension, x has 8,000,000 elements, versus only 8000 or 8 for the other inputs

Observation 3

	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Explanation 3

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

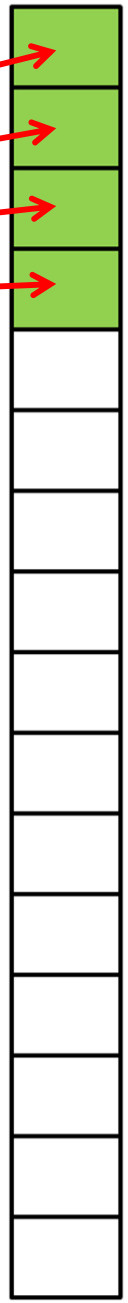
```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

- Cache coherence is enforced at “cache-line level.” Each time any value in a cache line is written, if the line is also stored in another core’s cache, the entire line will be invalidated, not just the value that was written.
- System used has two dual-core processors and each processor has its own cache. Suppose threads 0 and 1 are assigned to one of the processors and threads 2 and 3 are assigned to the other.
- 8,000,000 × 8 input, each thread is assigned 2,000,000 components
  8000 × 8000 input, each thread is assigned 2000 components
  8 × 8,000,000 input, each thread is assigned 2 components
- On system used, cache line is 64 bytes. y is double -> 8 bytes, a single cache line will store 8 doubles
- for 8 × 8,000,000 all of y is stored in a single cache line. Then every write to some element of y will invalidate the line in the other processor’s cache

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( );
                }

        }
```

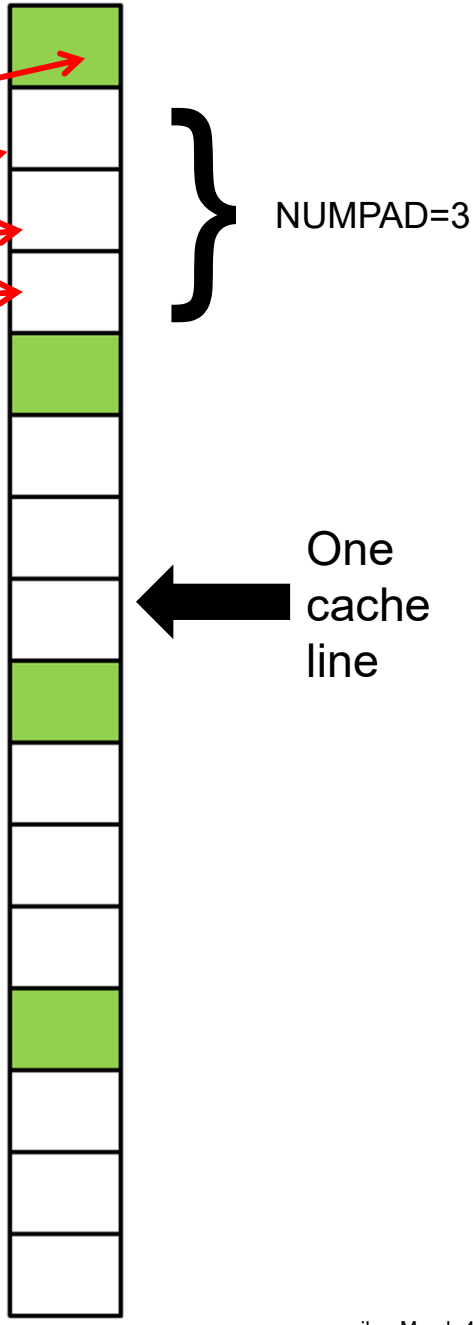
Some unpredictable function so the compiler doesn't try to optimize the j-for-loop away.



One cache line

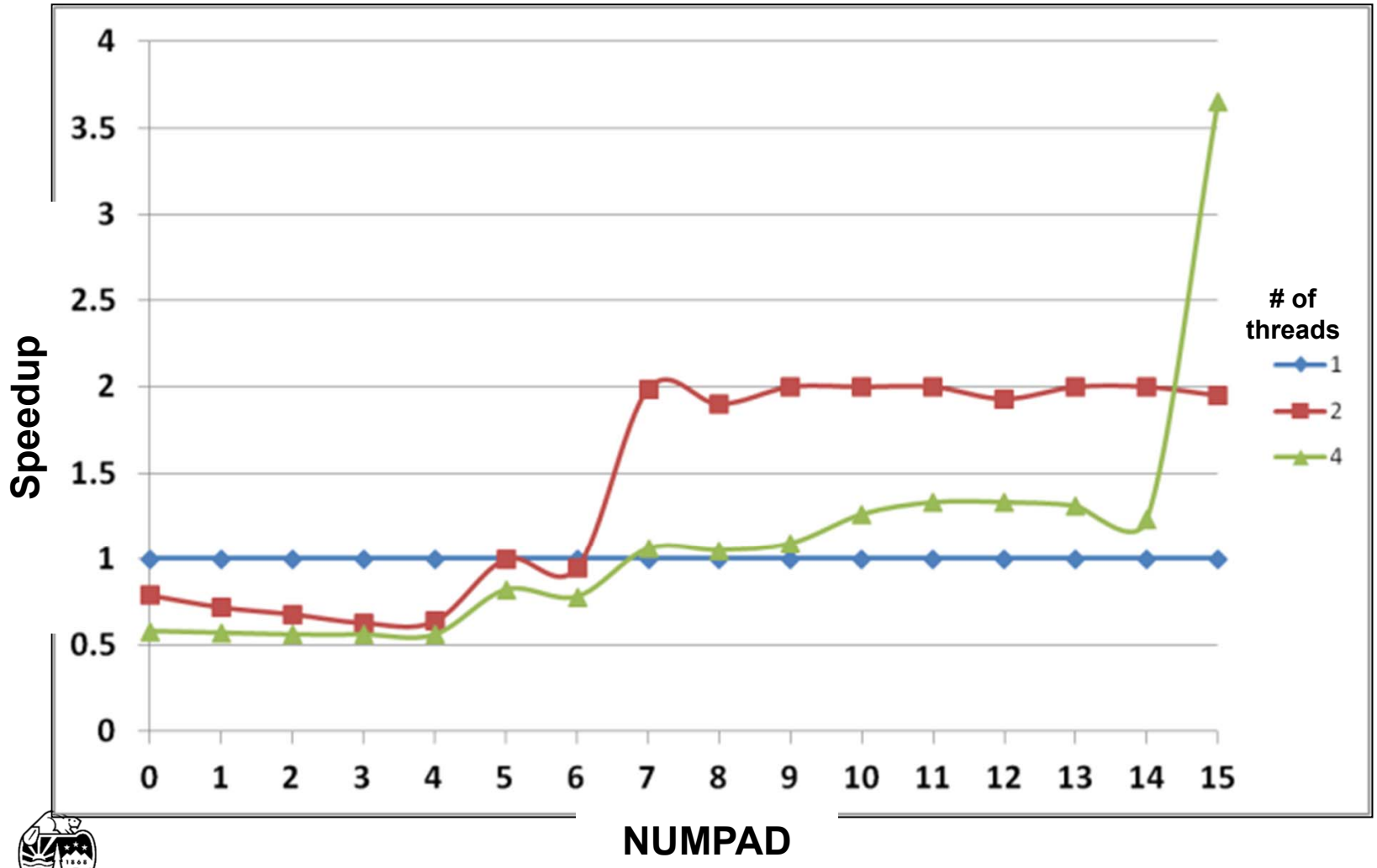
False Sharing – Fix #1
Adding some padding

```
#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( );
                }

        }
```



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

False Sharing – Fix #1

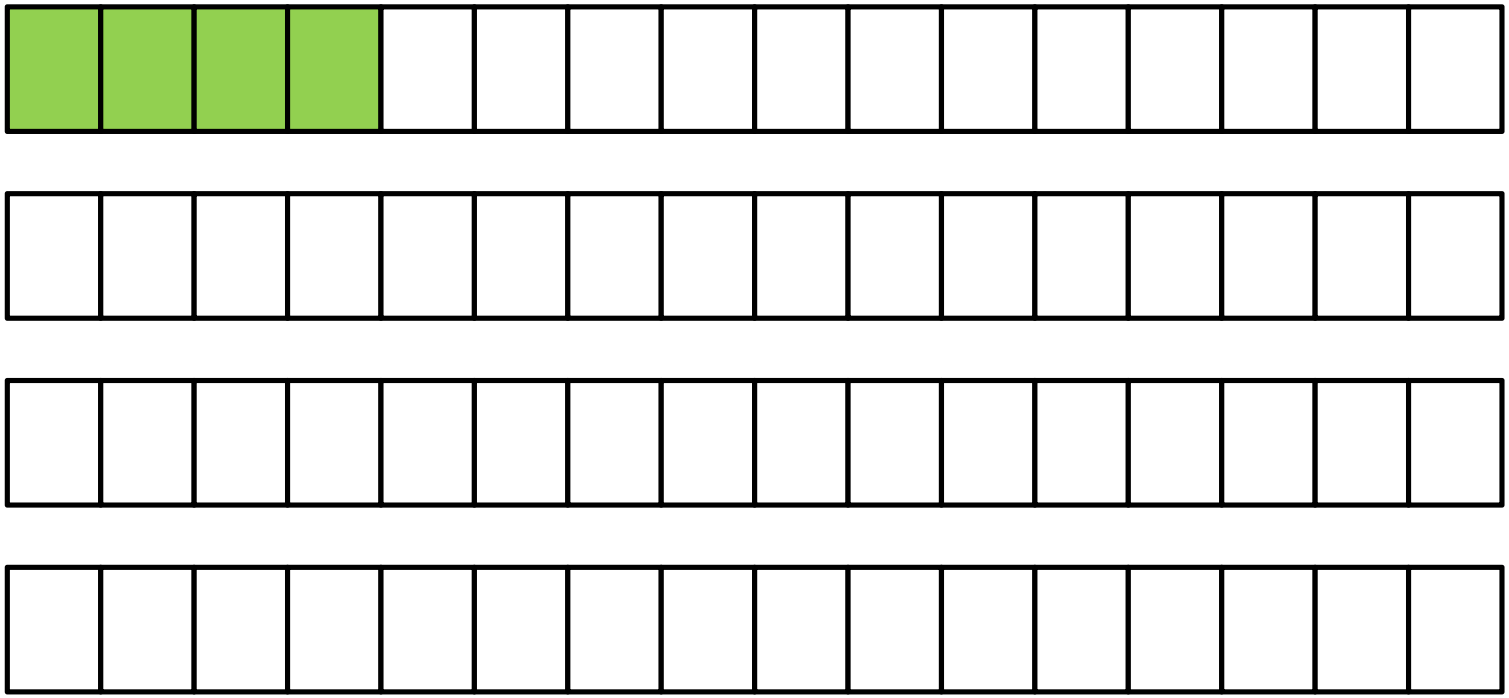


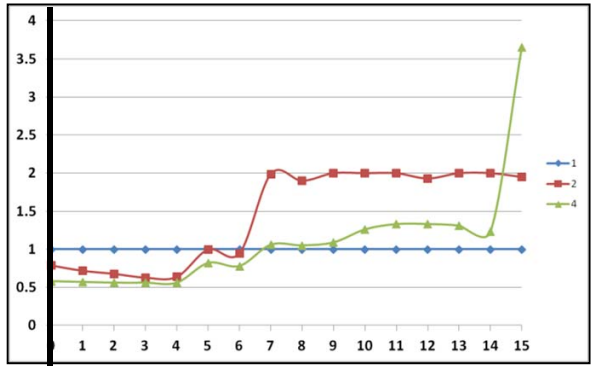
Oregon State University
Computer Graphics

Why do these curves look this way?

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

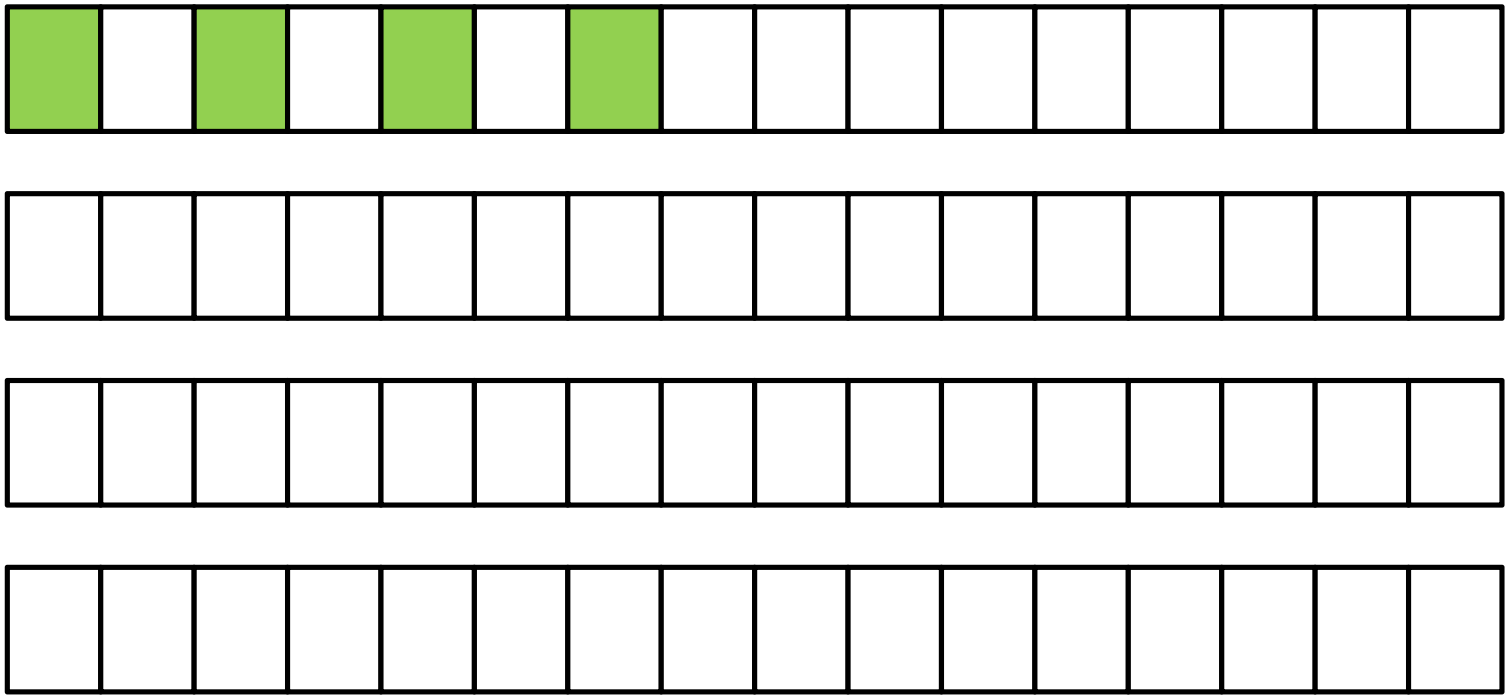
NUMPAD = 0

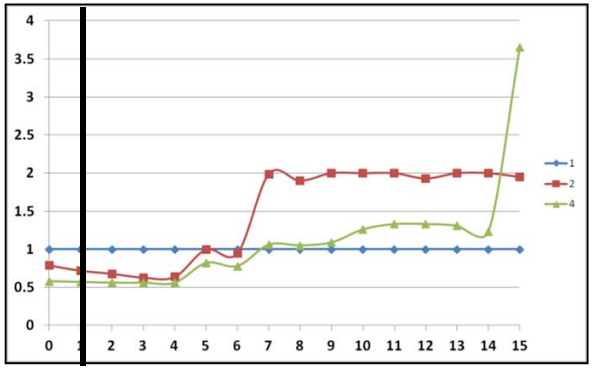




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

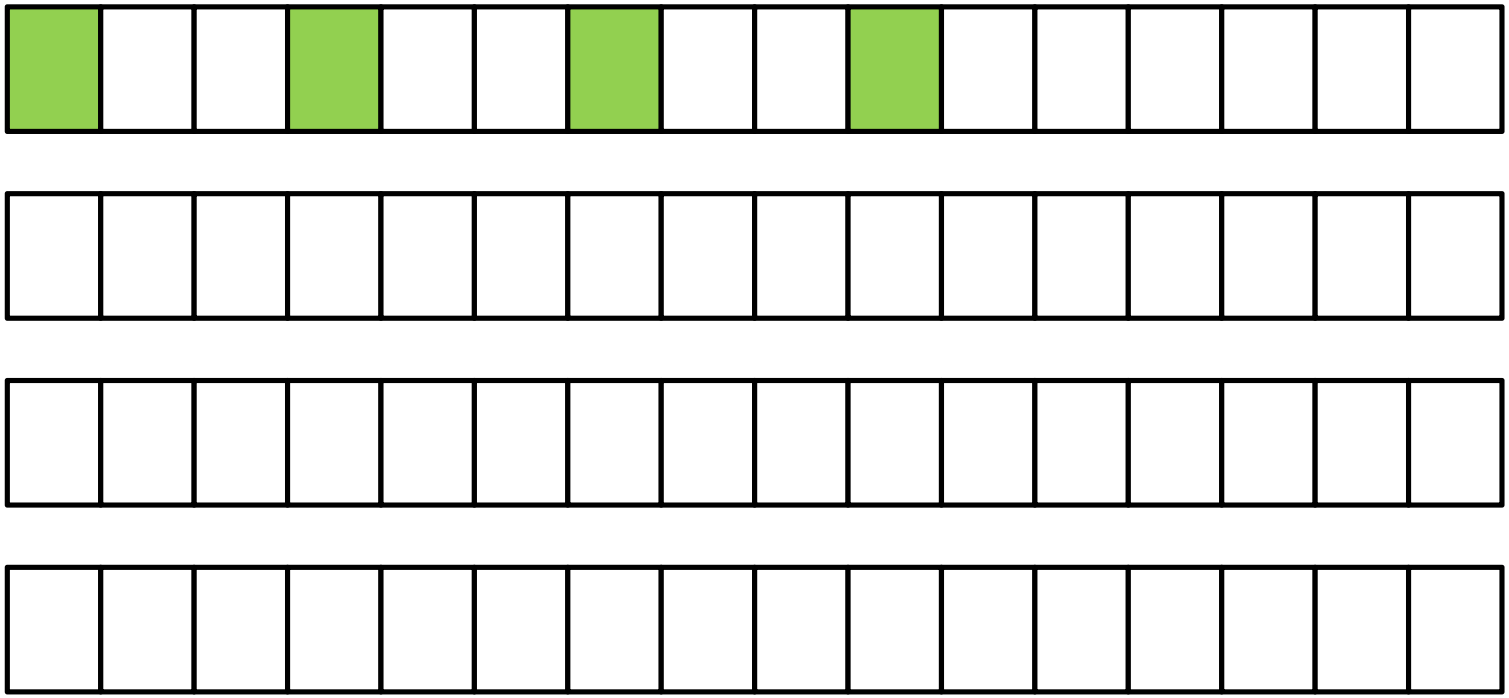
NUMPAD = 1

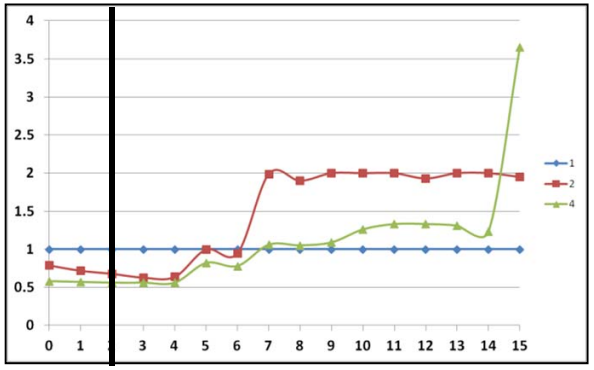




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

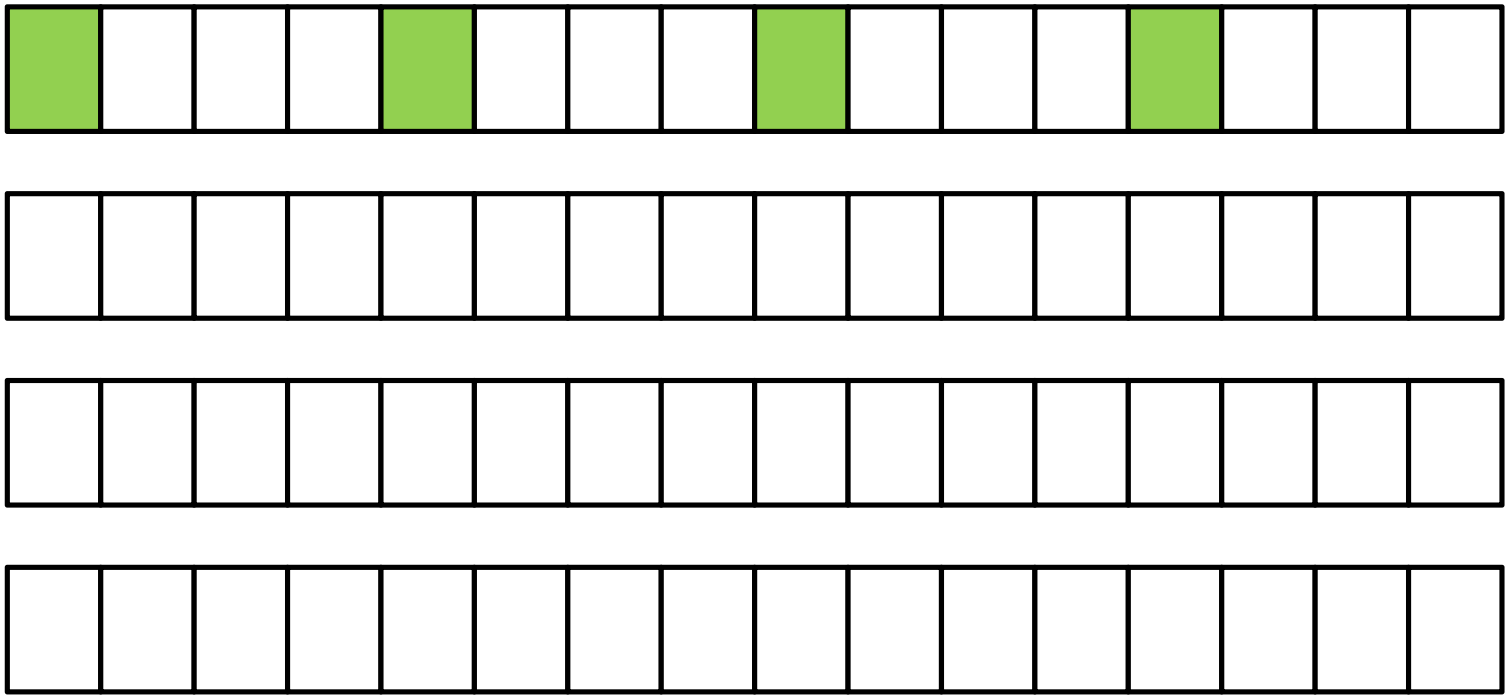
NUMPAD = 2

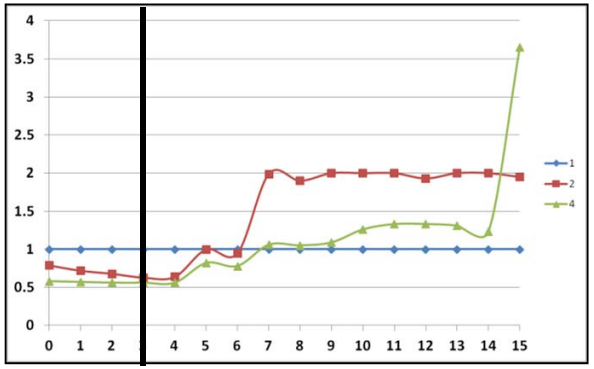




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

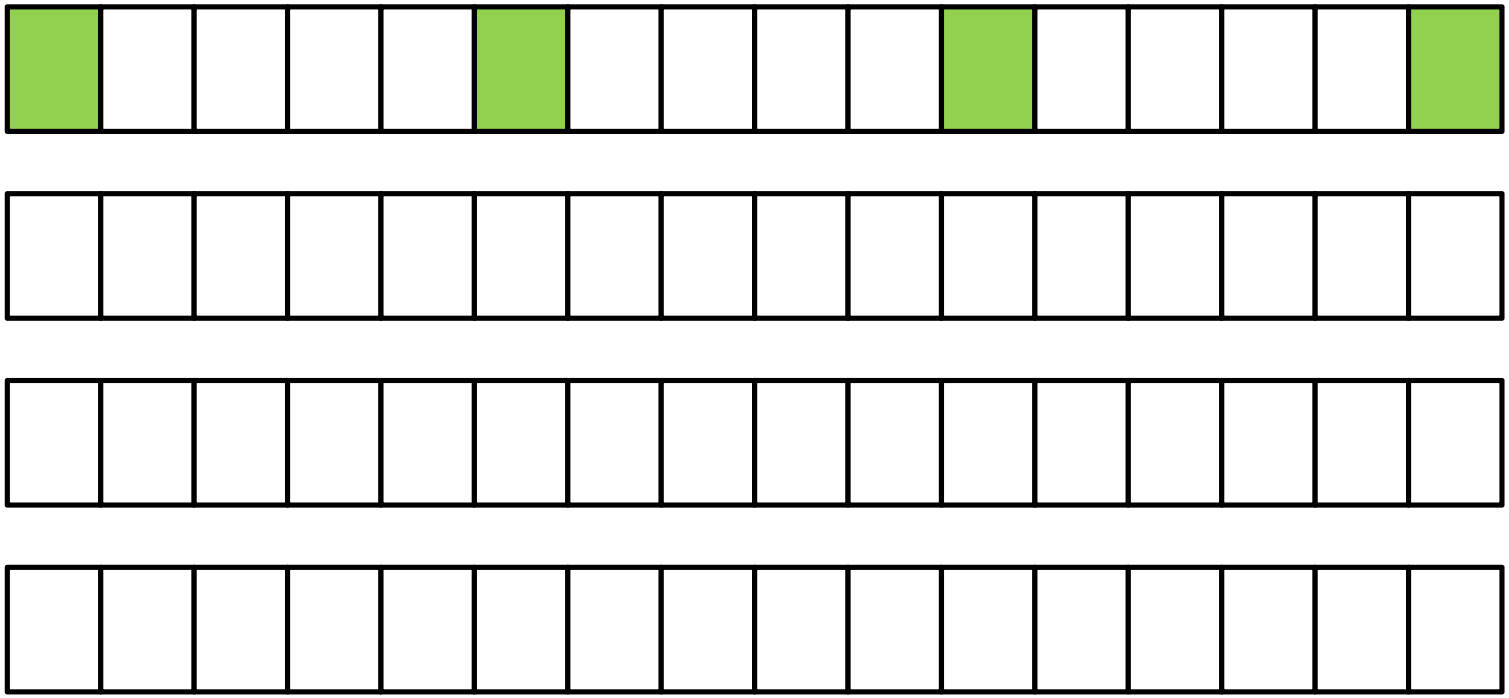
NUMPAD = 3

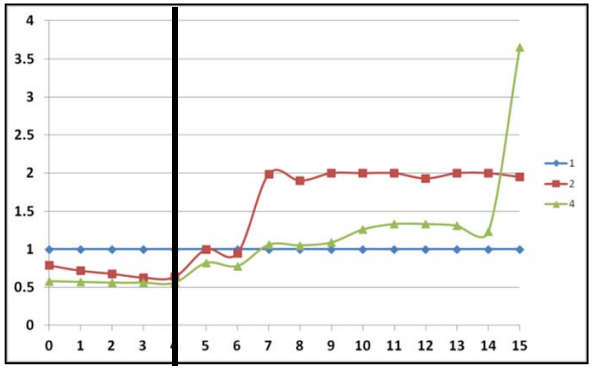




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

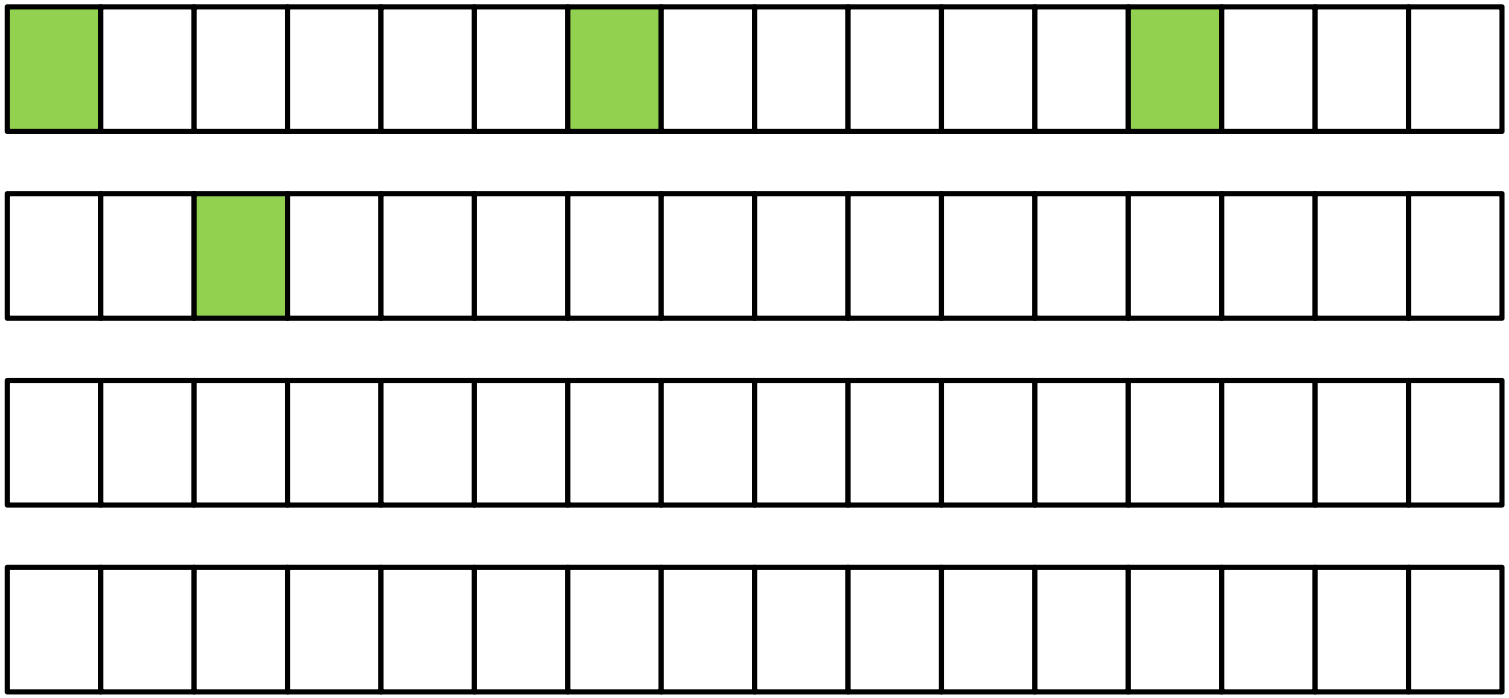
NUMPAD = 4

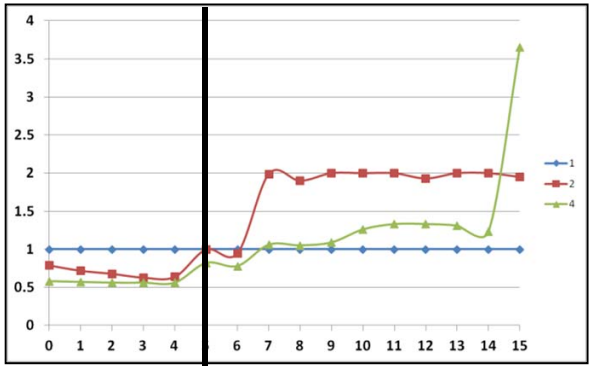




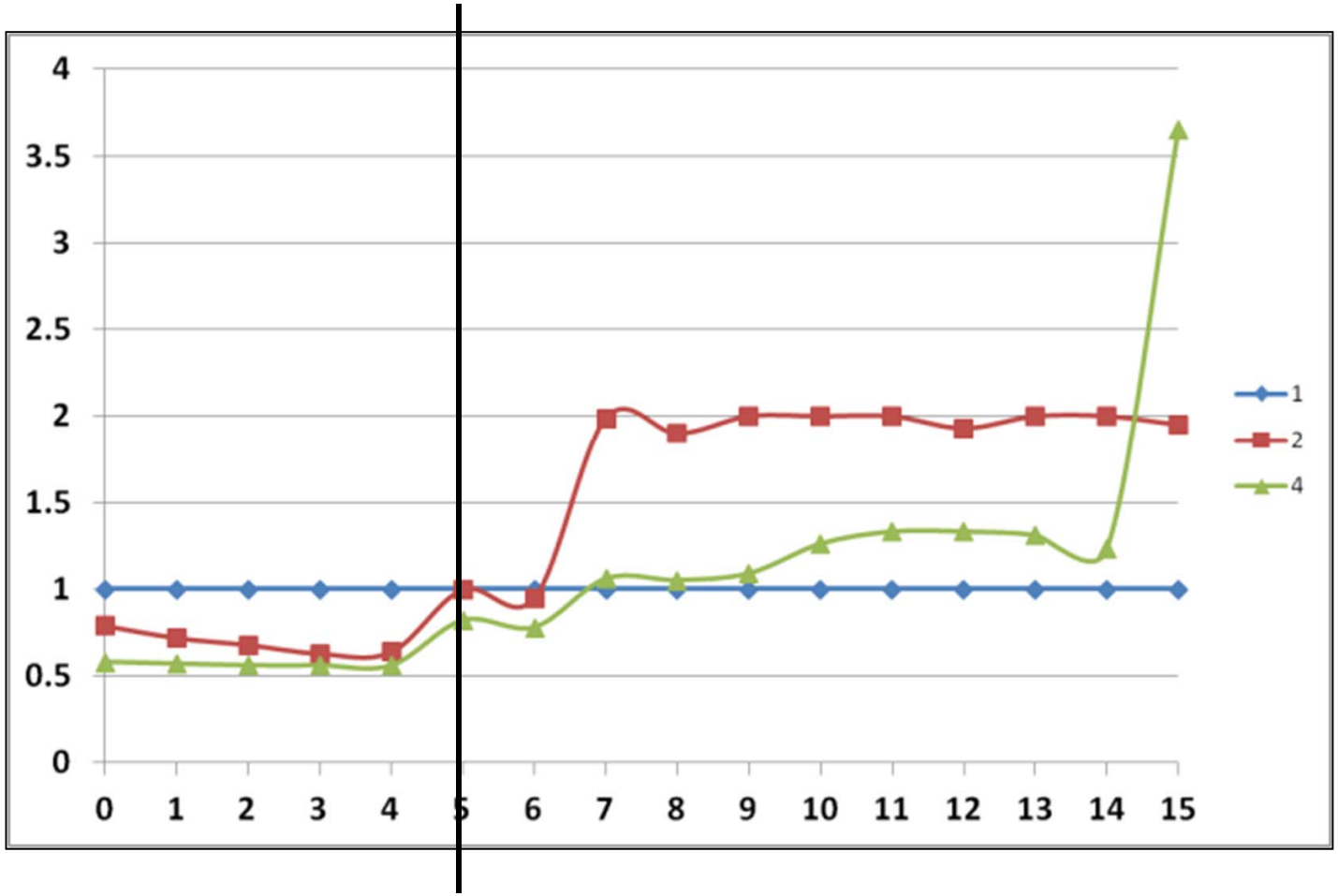
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 5



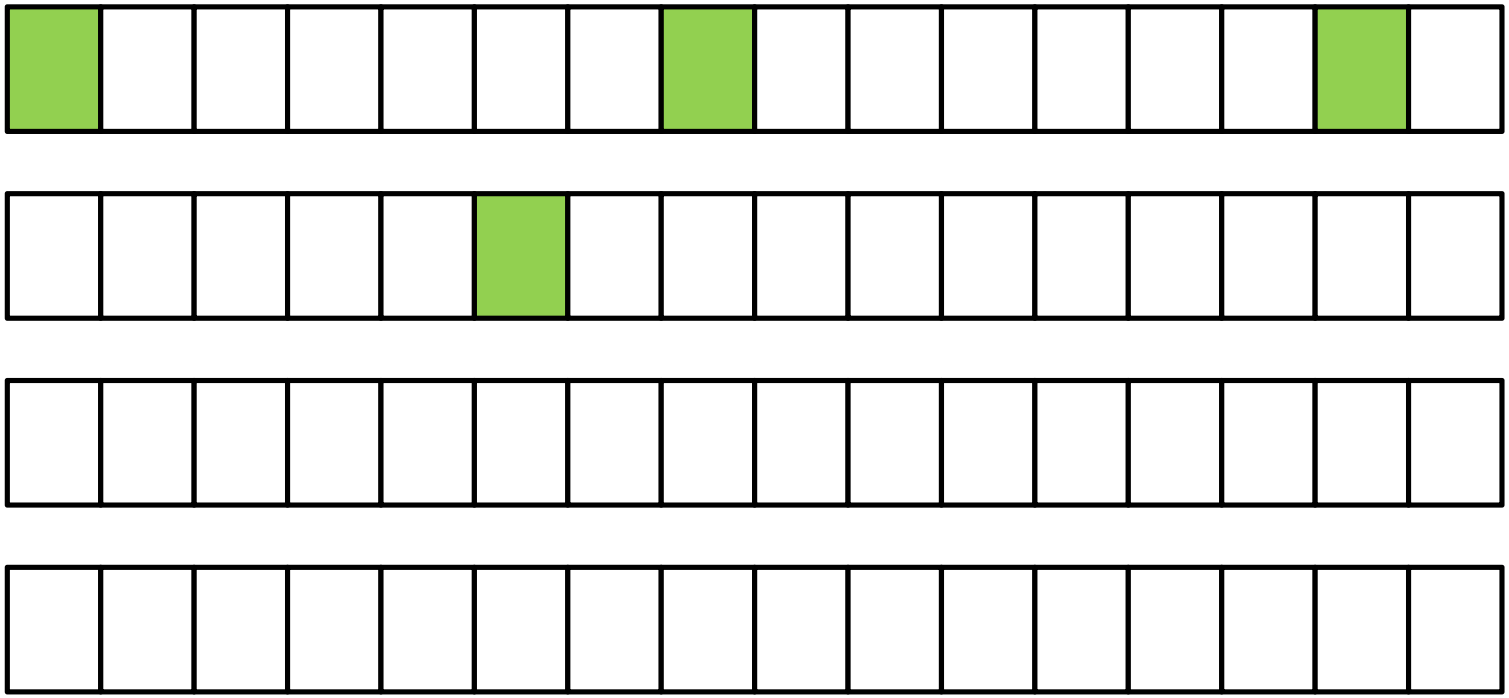


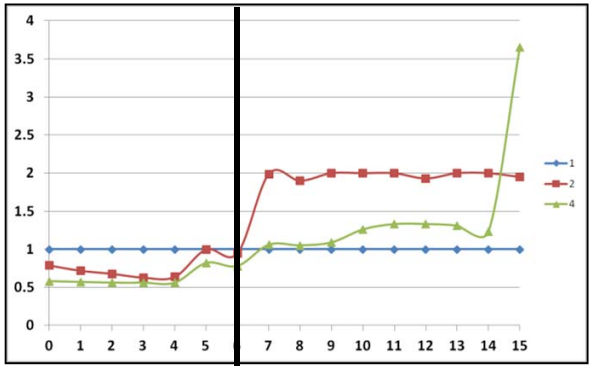
False Sharing – Fix #1



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

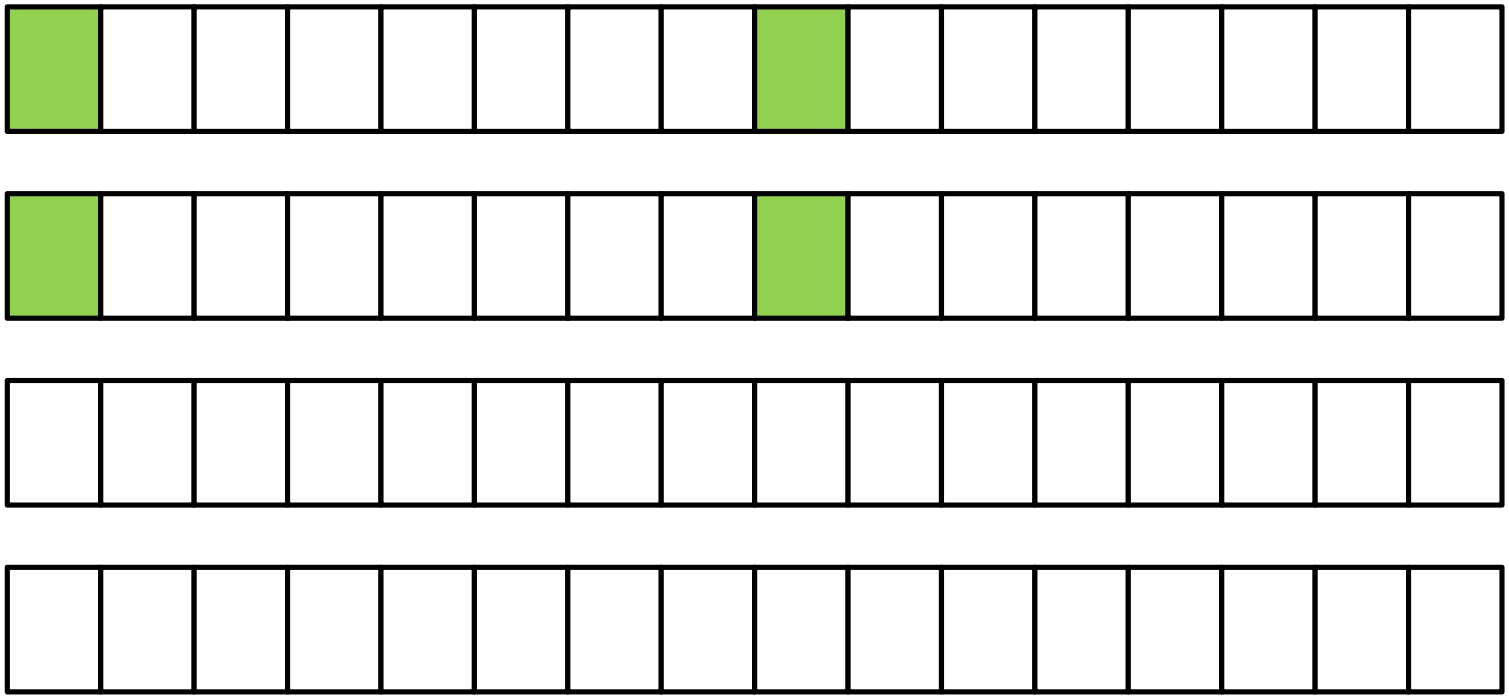
NUMPAD = 6

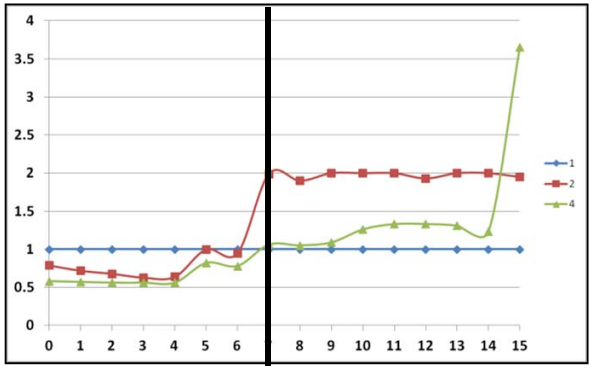




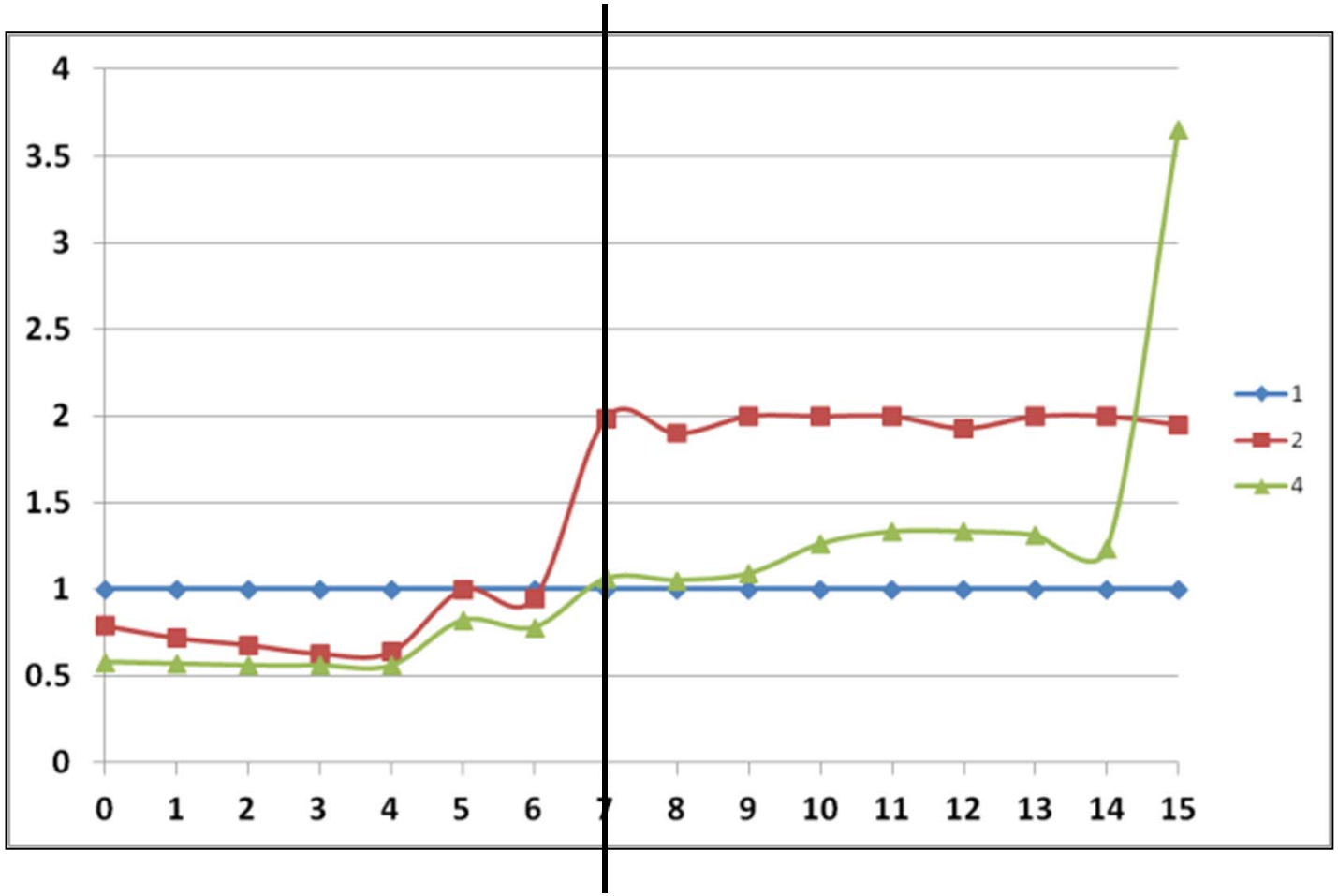
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 7



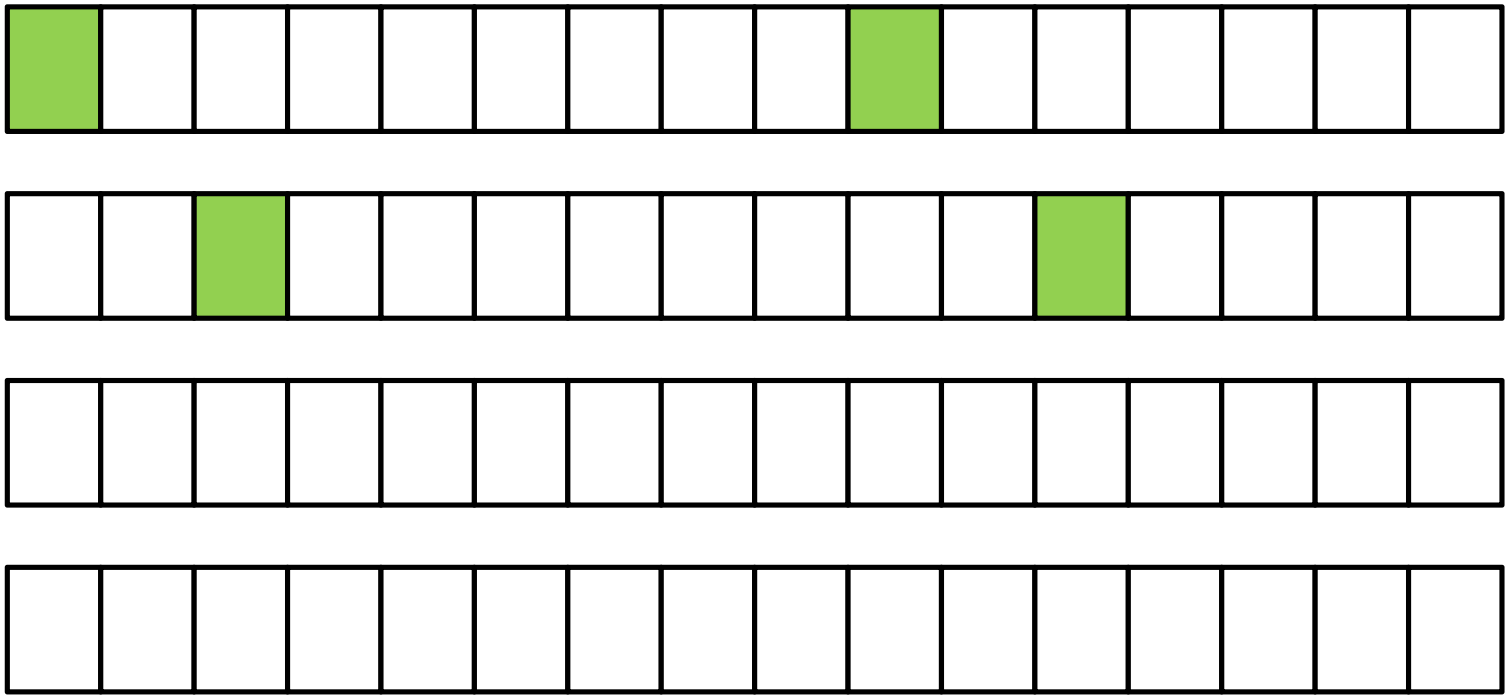


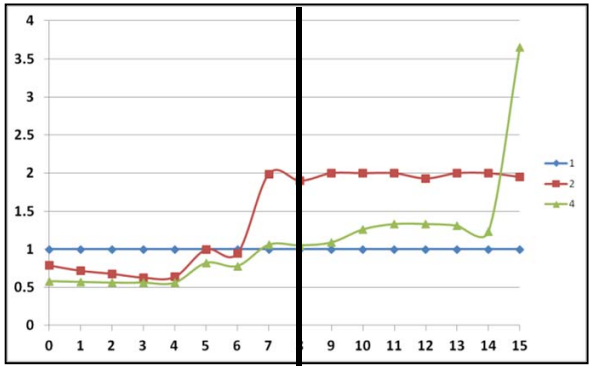
False Sharing – Fix #1



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

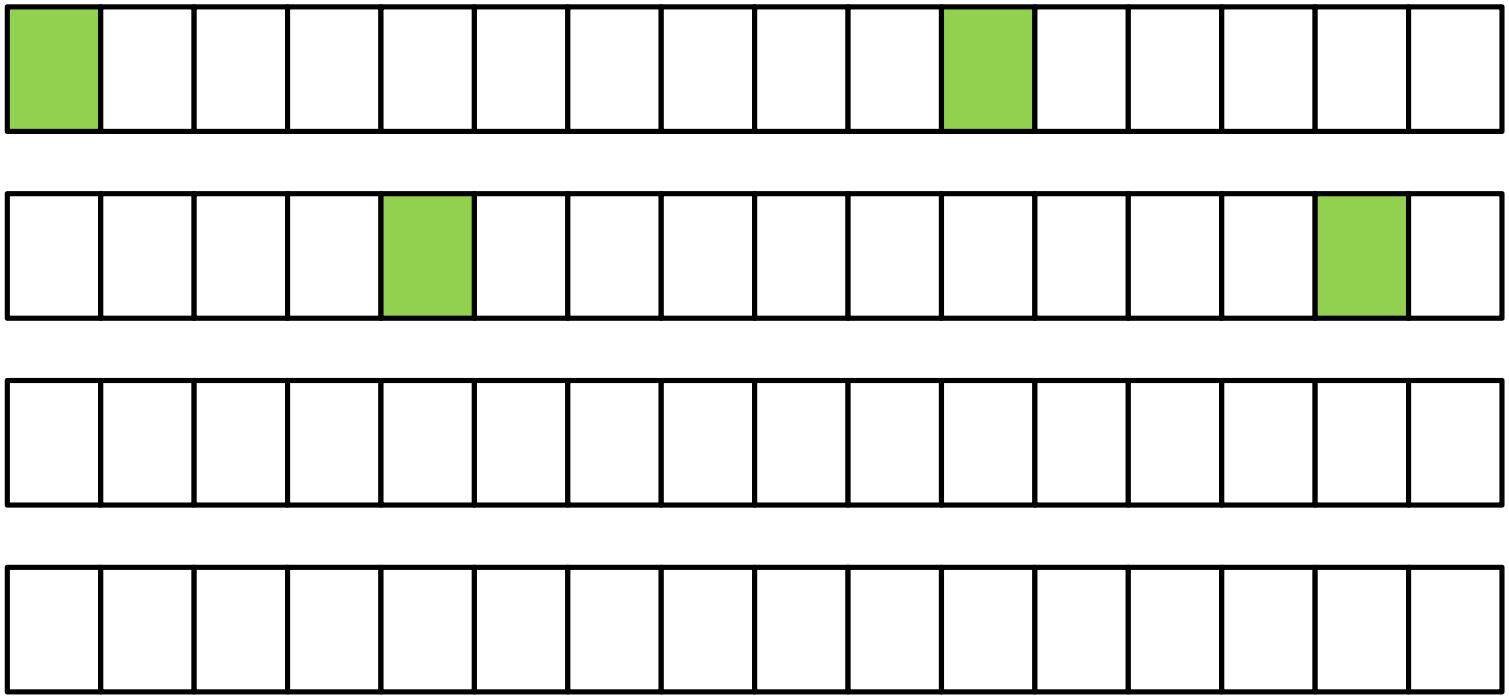
NUMPAD = 8

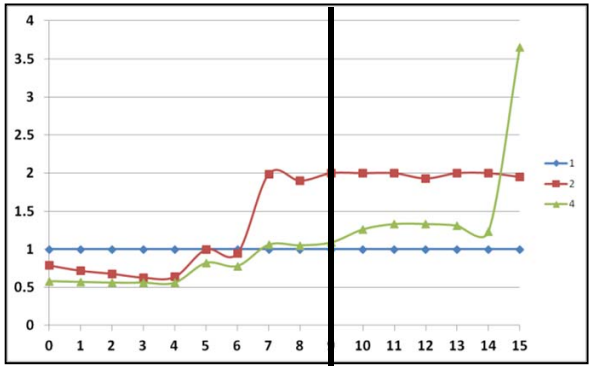




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

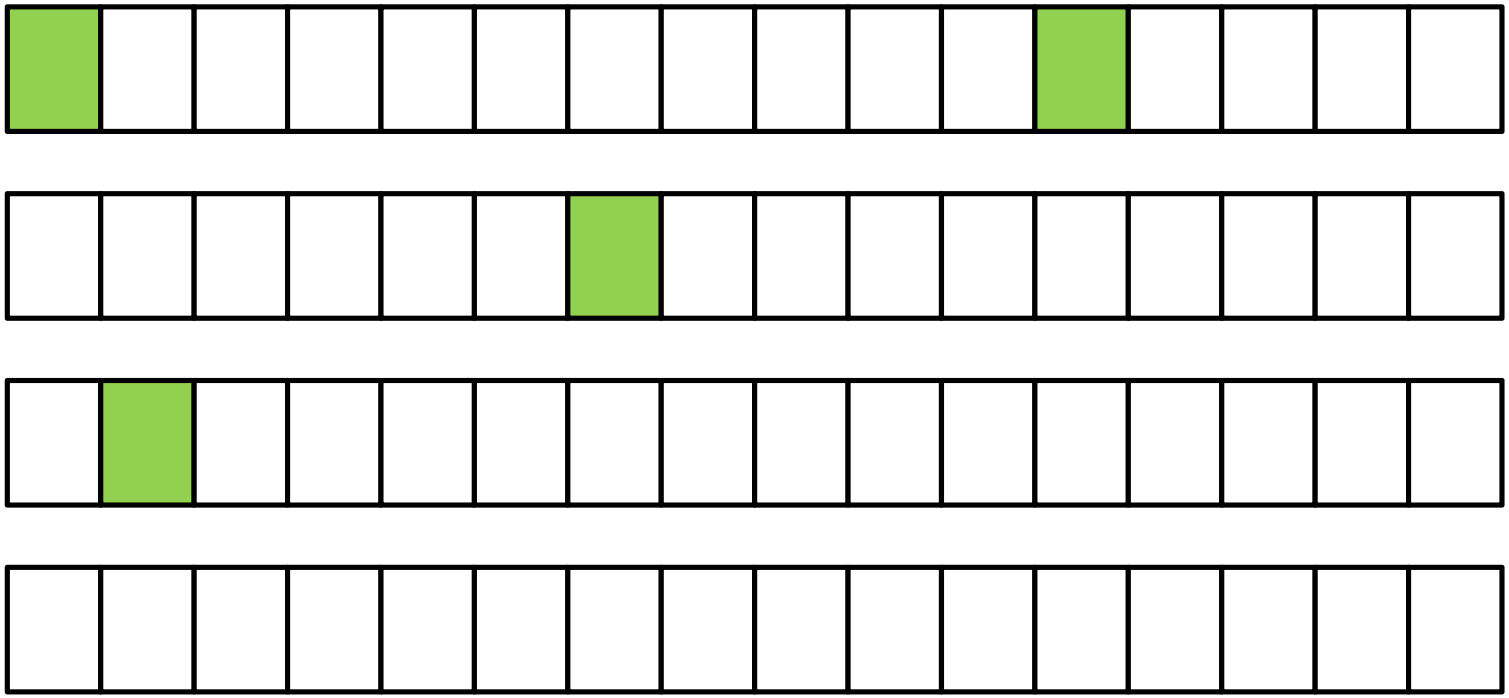
NUMPAD = 9

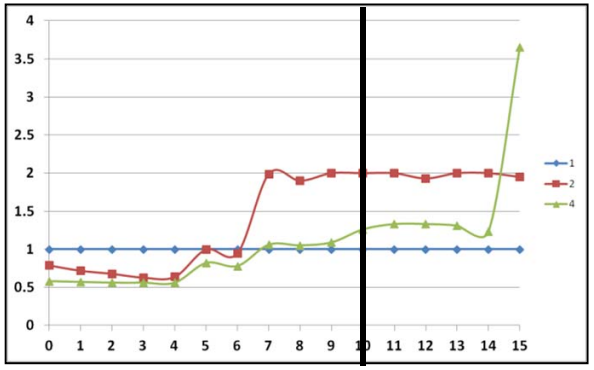




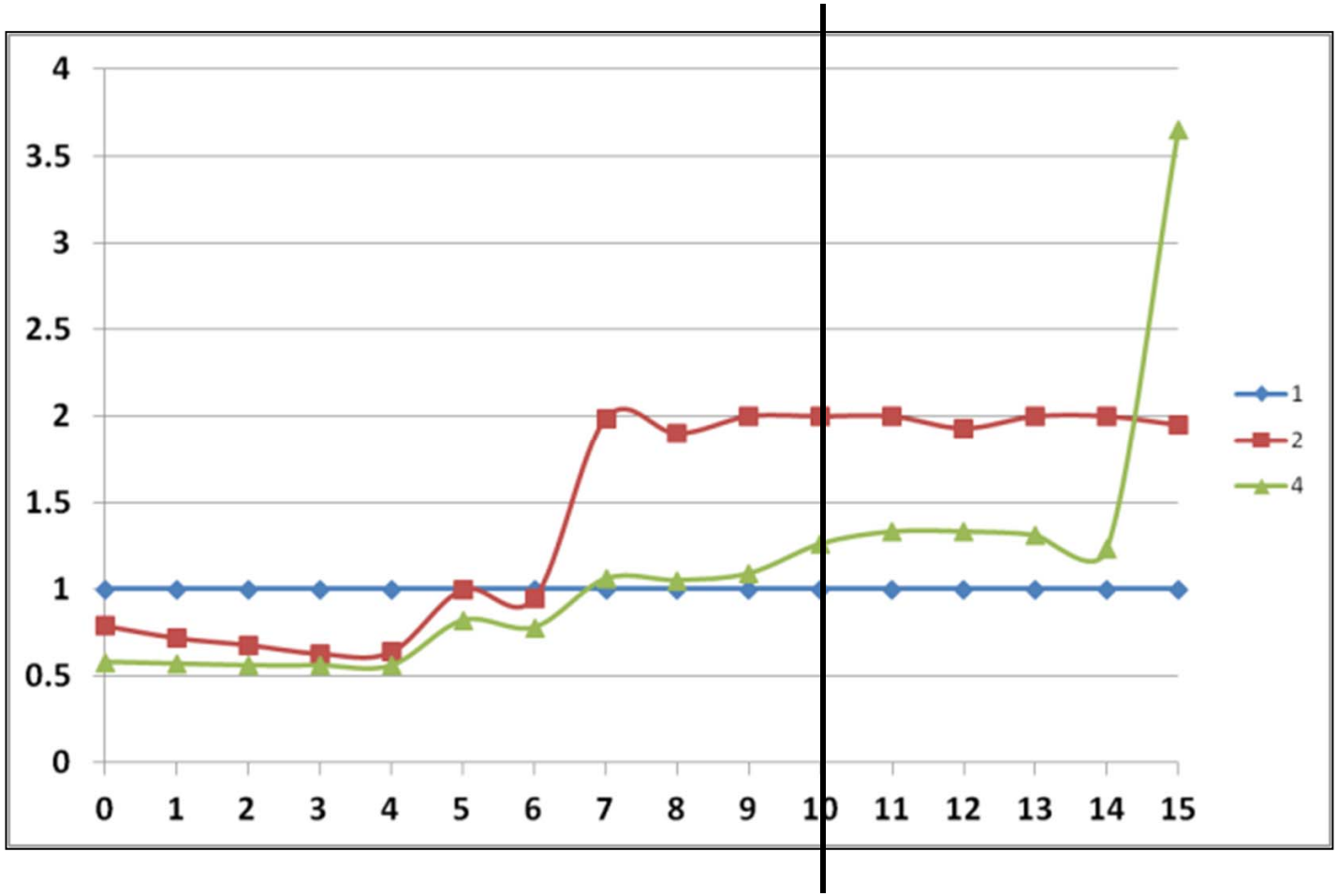
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 10



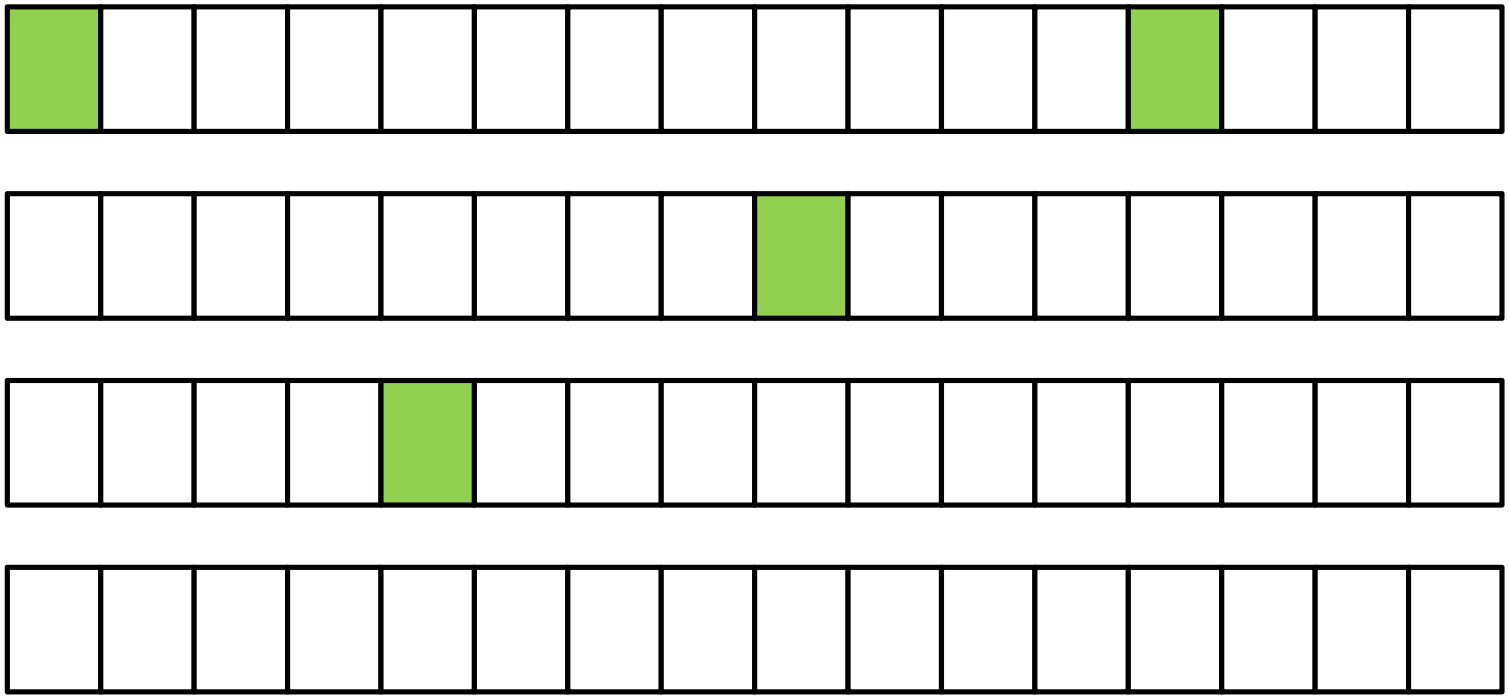


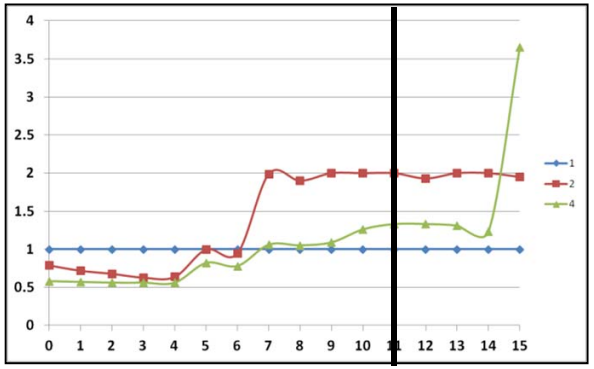
False Sharing – Fix #1



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

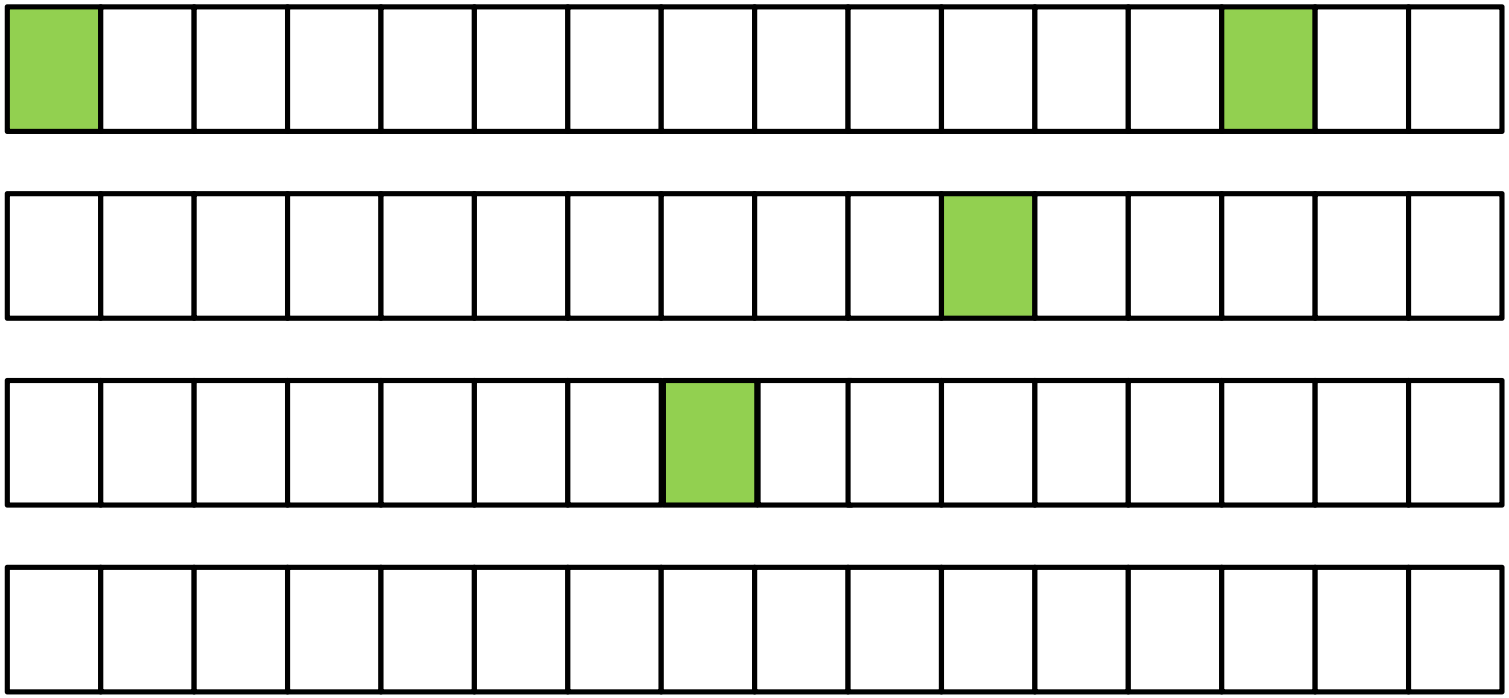
NUMPAD = 11

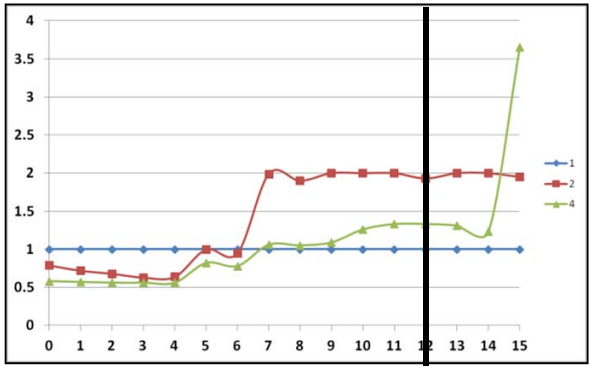




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

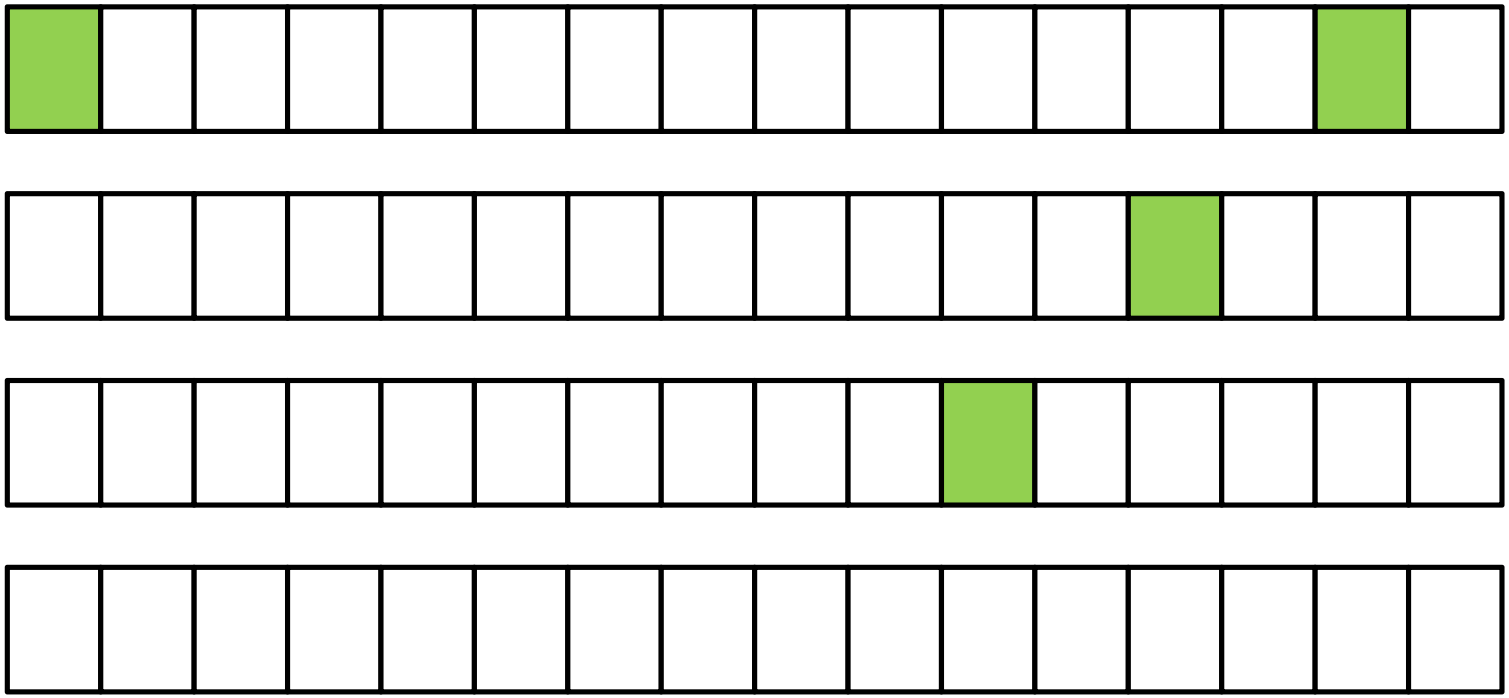
NUMPAD = 12

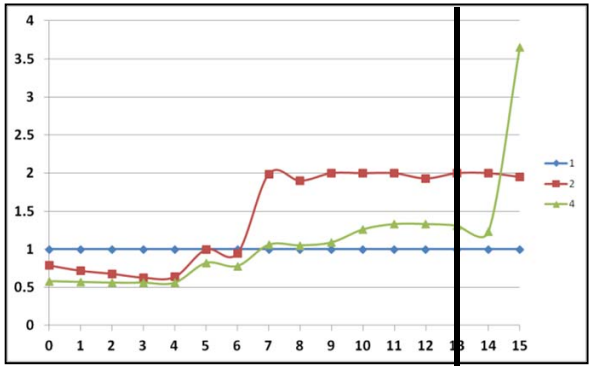




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

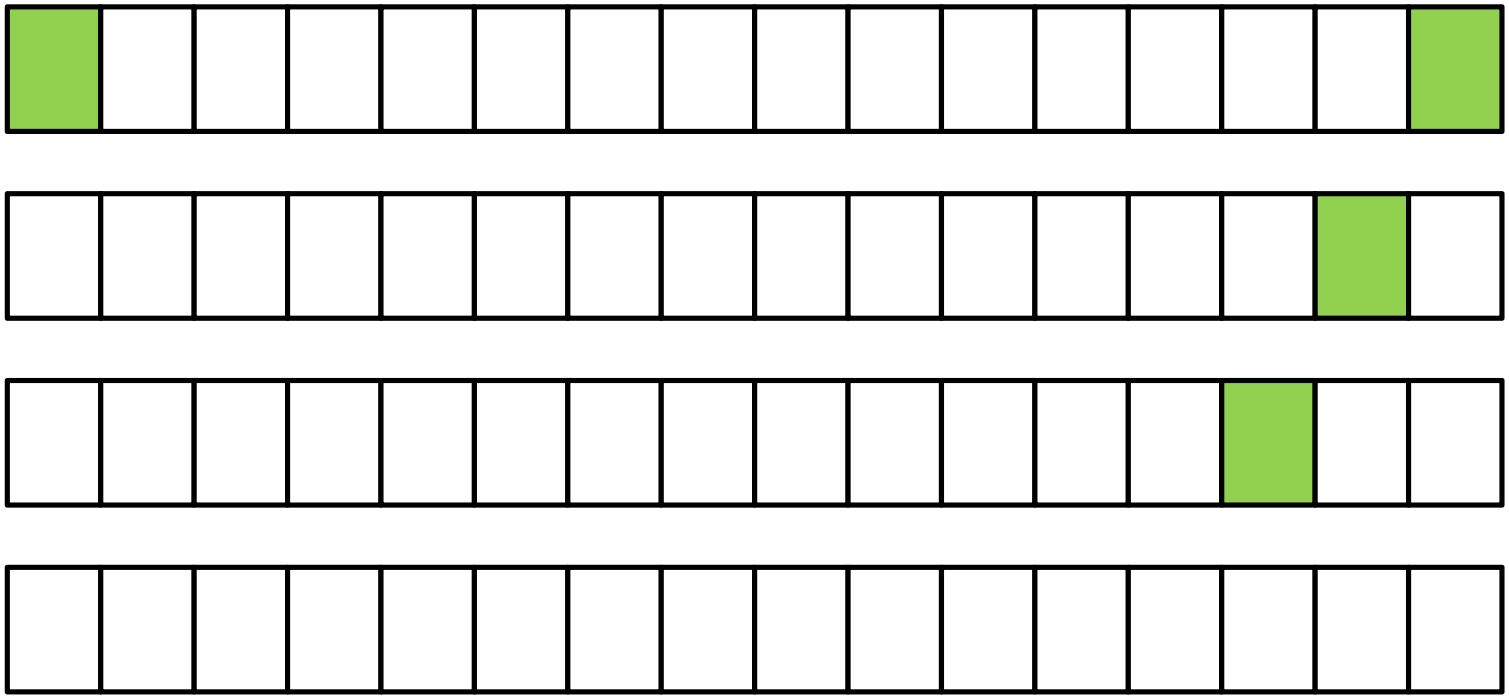
NUMPAD = 13

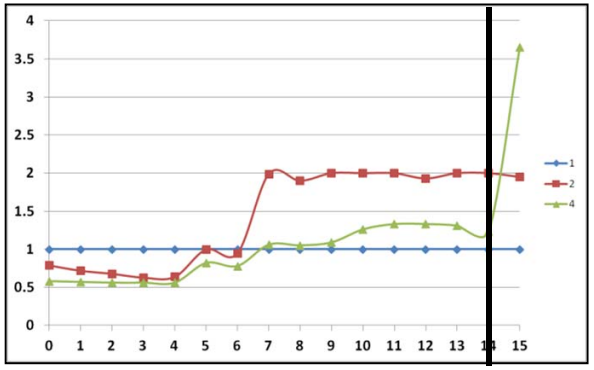




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

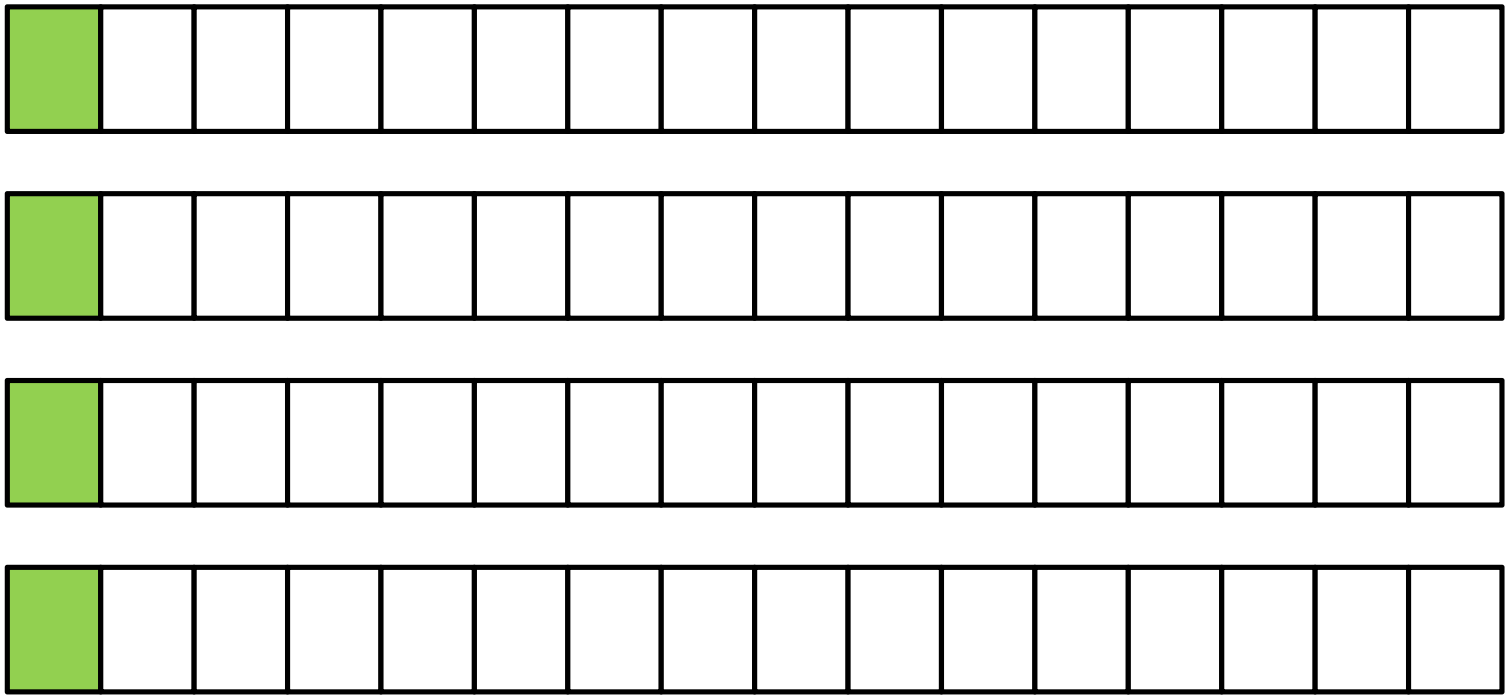
NUMPAD = 14

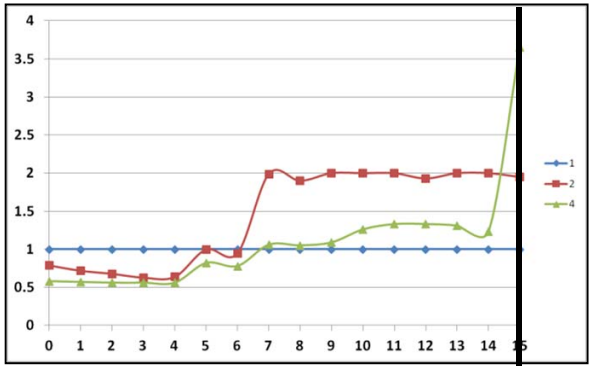




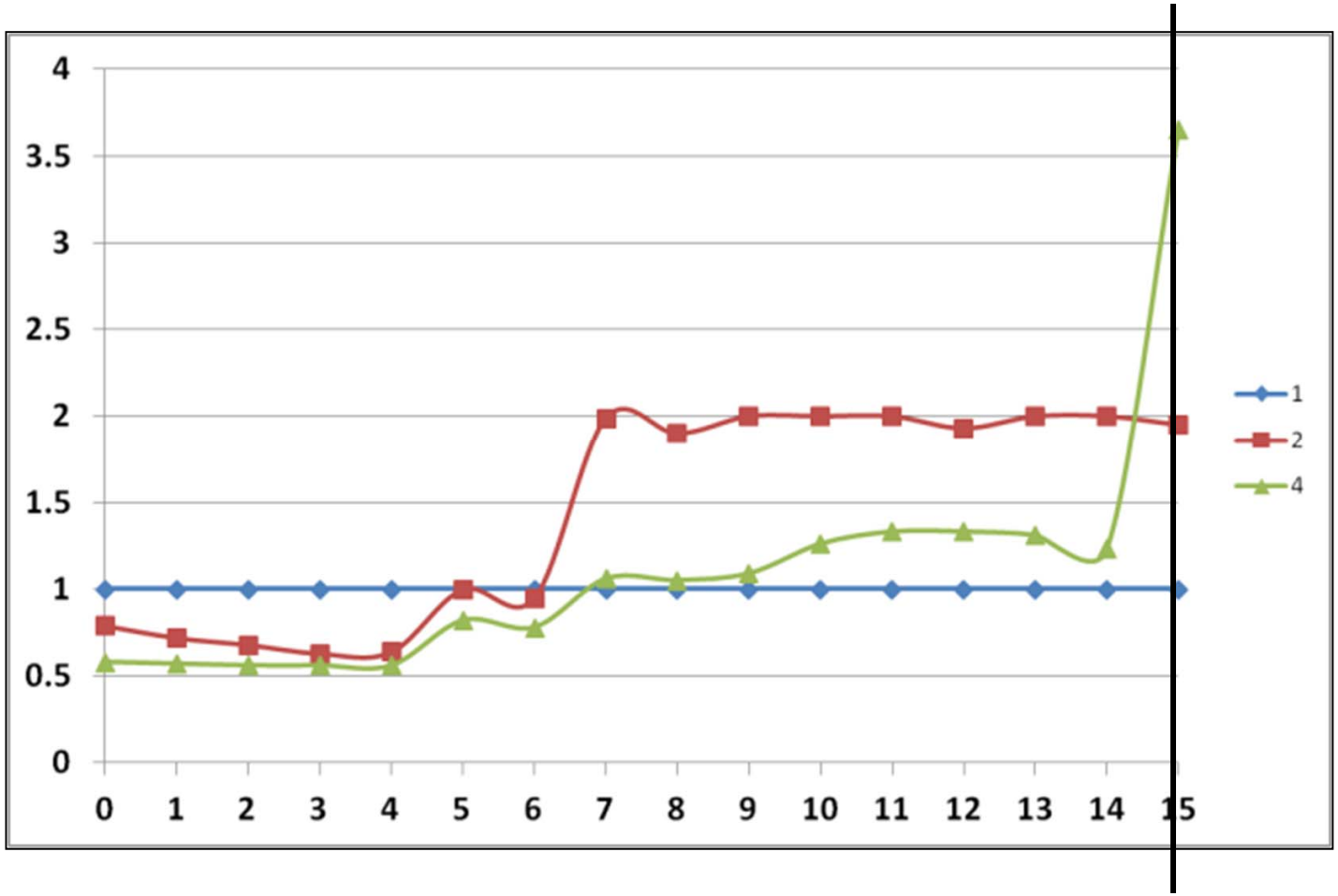
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 15





False Sharing – Fix #1

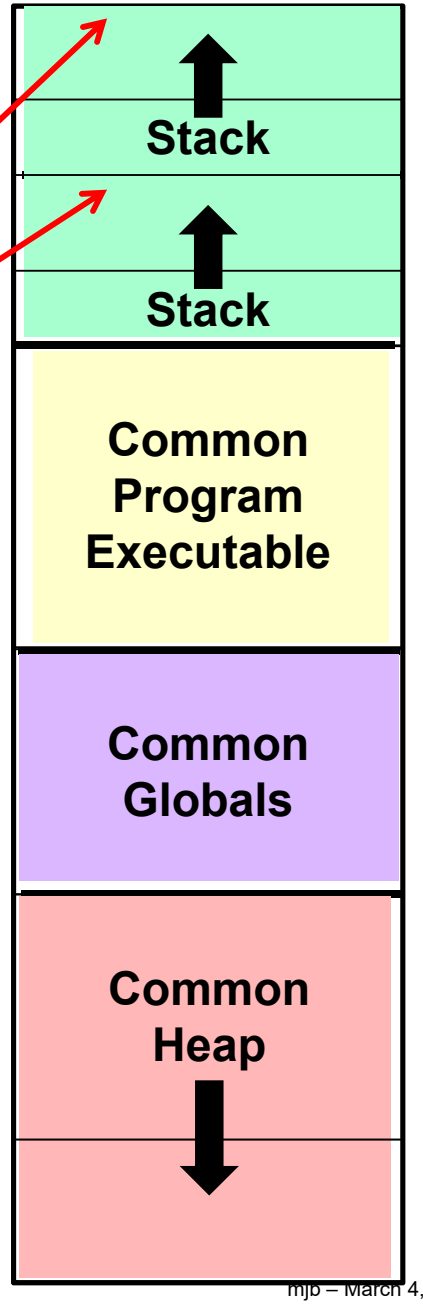


False Sharing – Fix #2: Using local (private) variables

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?

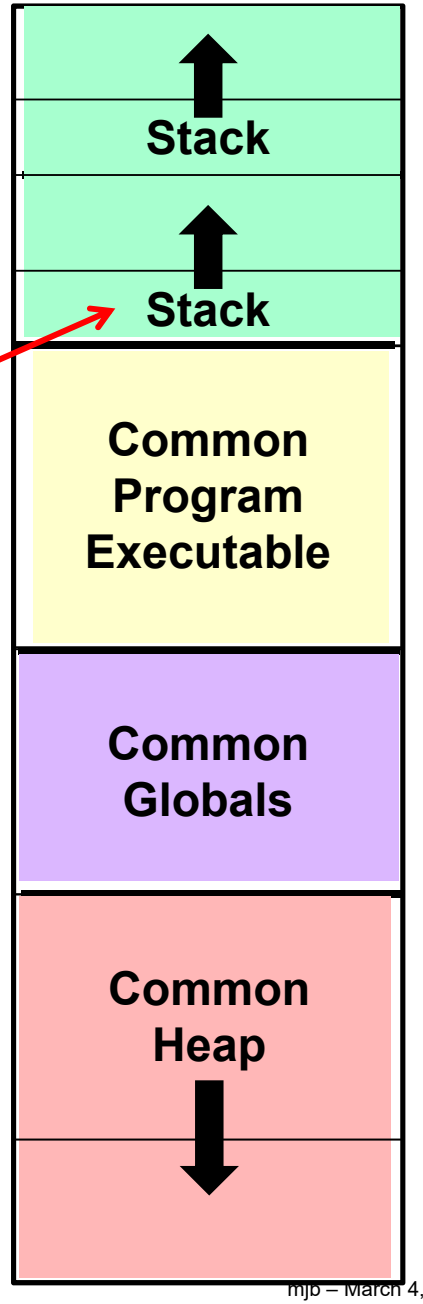
If we use local variables, instead of contiguous array locations, that will spread our writes out in memory, and to different cache lines.



False Sharing – Fix #2

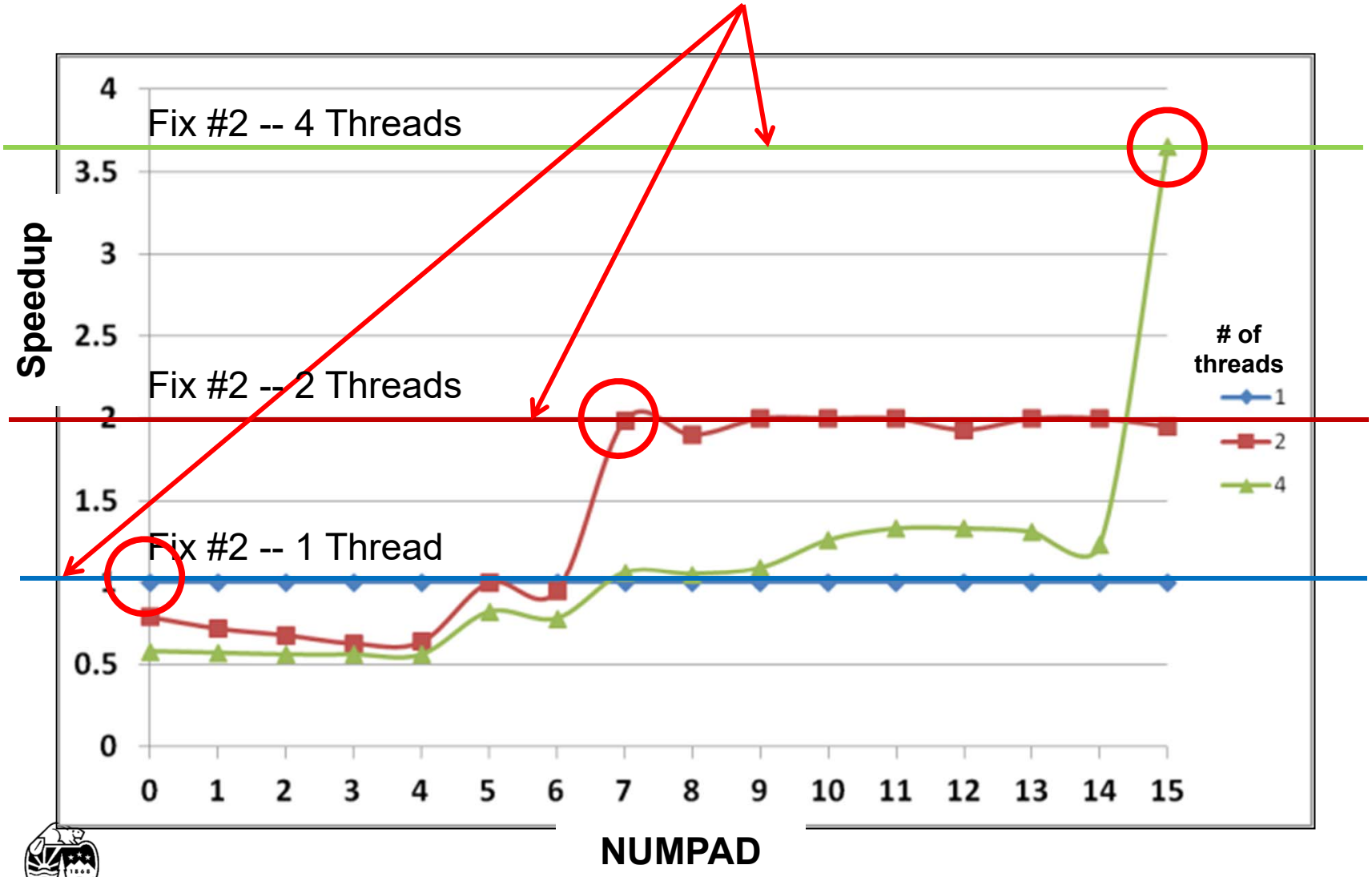
```
#include <stdlib.h>
struct s
{
        float value;
}   Array[4];

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

Cache Coherence

A memory system is coherent if:

- Write propagation: P1 writes to X, sufficient time elapses, P2 reads X and gets the value written by P1
- Write serialization: Two writes to the same location by two processors are seen in the same order by all processors
- The memory *consistency* model defines “time elapsed” before the effect of a processor is seen by others and the ordering with R/W to other locations (loosely speaking – more later)

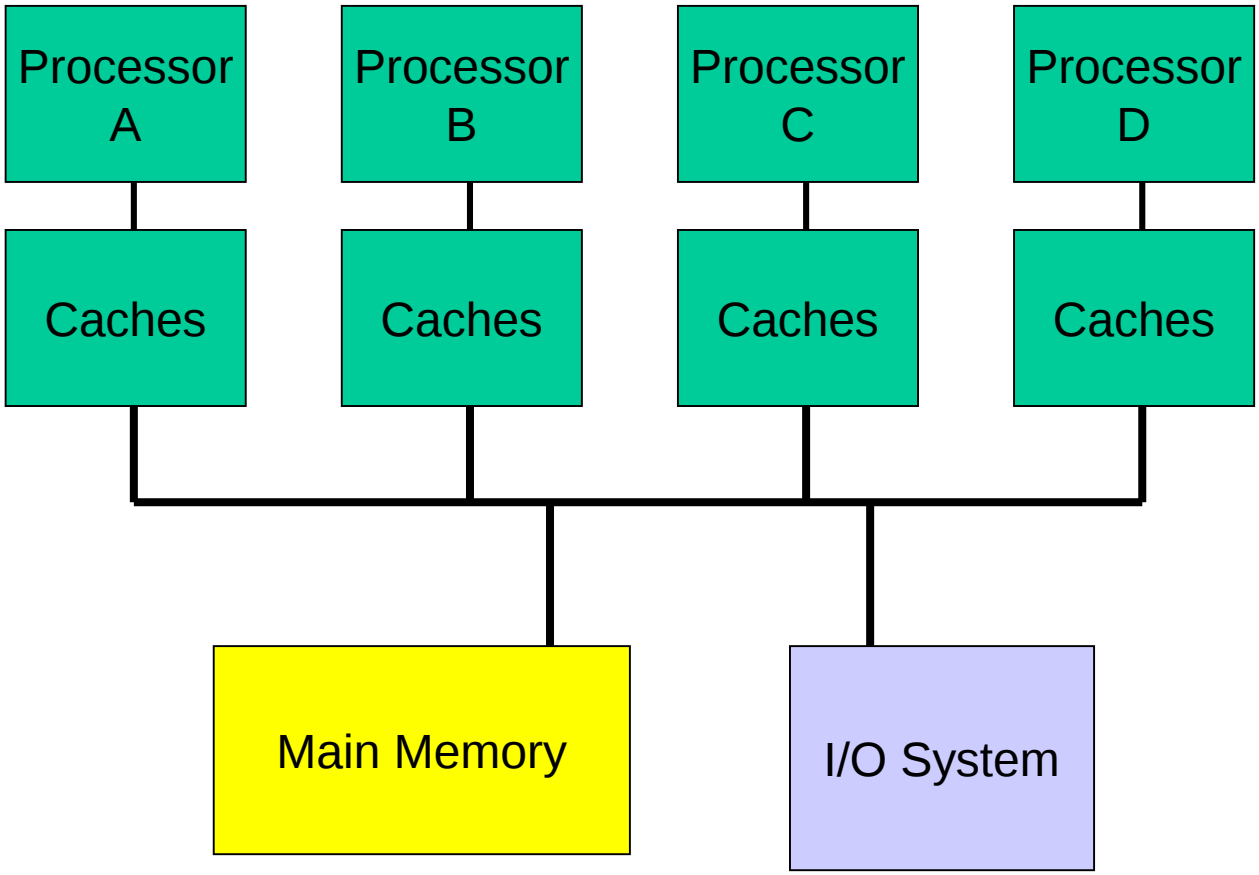
Cache Coherence Protocols

- Directory-based: A single location (directory) keeps track of the sharing status of a block of memory

- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

  - Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
  - Write-update: when a processor writes, it updates other shared copies of that block

SMP Example



A: Rd X
B: Rd X
C: Rd X
A: Wr X
A: Wr X
C: Wr X
B: Rd X
A: Rd X
A: Rd Y
B: Wr X
B: Rd Y
B: Wr X
B: Wr Y

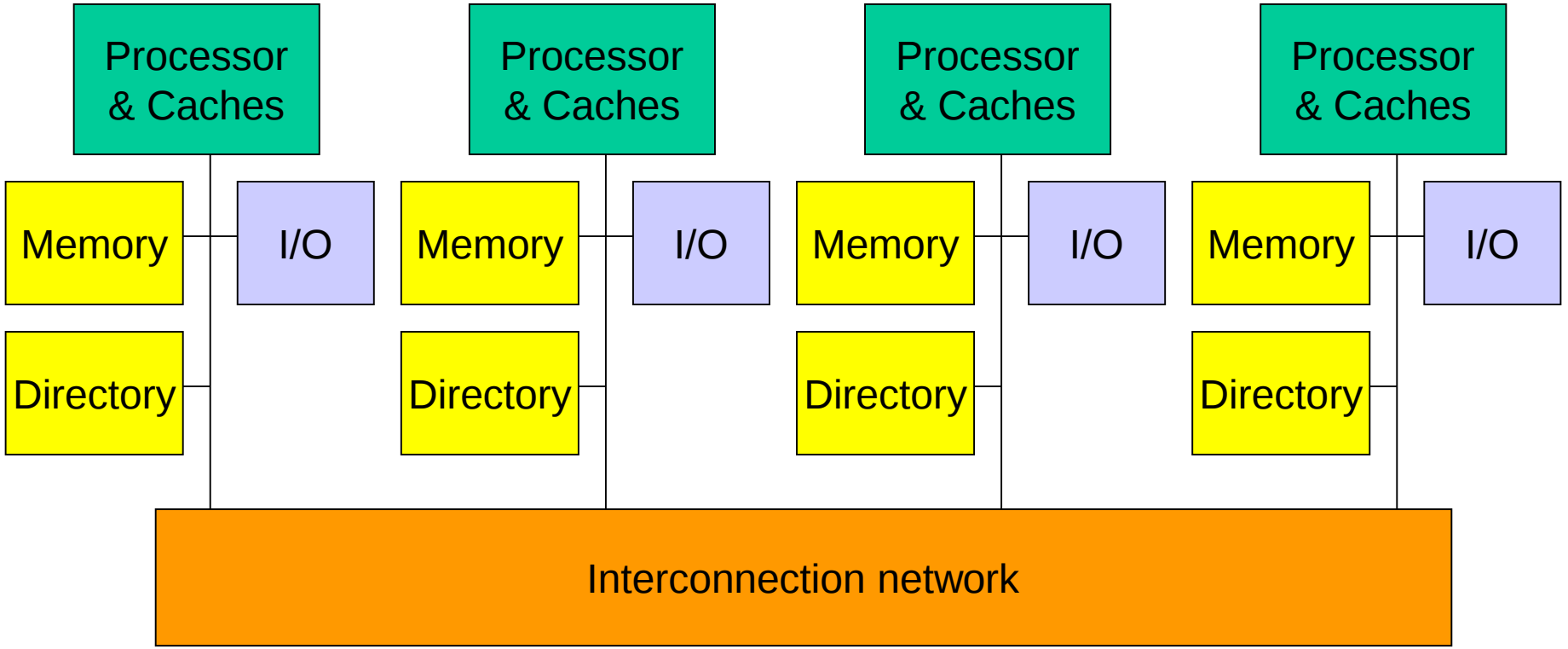
SMP Example

	A	B	C	
A: Rd X	S			Rd-miss req; mem responds
B: Rd X	S	S		Rd-miss req; mem responds
C: Rd X	S	S	S	Rd-miss req; mem responds
A: Wr X	M	I	I	Upgrade req; no resp; others inv
A: Wr X	M	I	I	Cache hit
C: Wr X	I	I	M	Wr-miss req; A resp & inv; no wrtbk
B: Rd X	I	S	S	Rd-miss req; C resp; wrtbk to mem
A: Rd X	S	S	S	Rd-miss req; mem responds
A: Rd Y	S (Y)	S (X)	S (X)	Rd-miss req; X evicted; mem resp
B: Wr X	S (Y)	M (X)	I	Upgrade req; no resp; others inv
B: Rd Y	S (Y)	S (Y)	I	Rd-miss req; mem resp; X wrtbk
B: Wr X	S (Y)	M (X)	I	Wr-miss req; mem resp; Y evicted
B: Wr Y	I	M (Y)	I	Wr-miss req; mem resp; others inv; X wrtbk

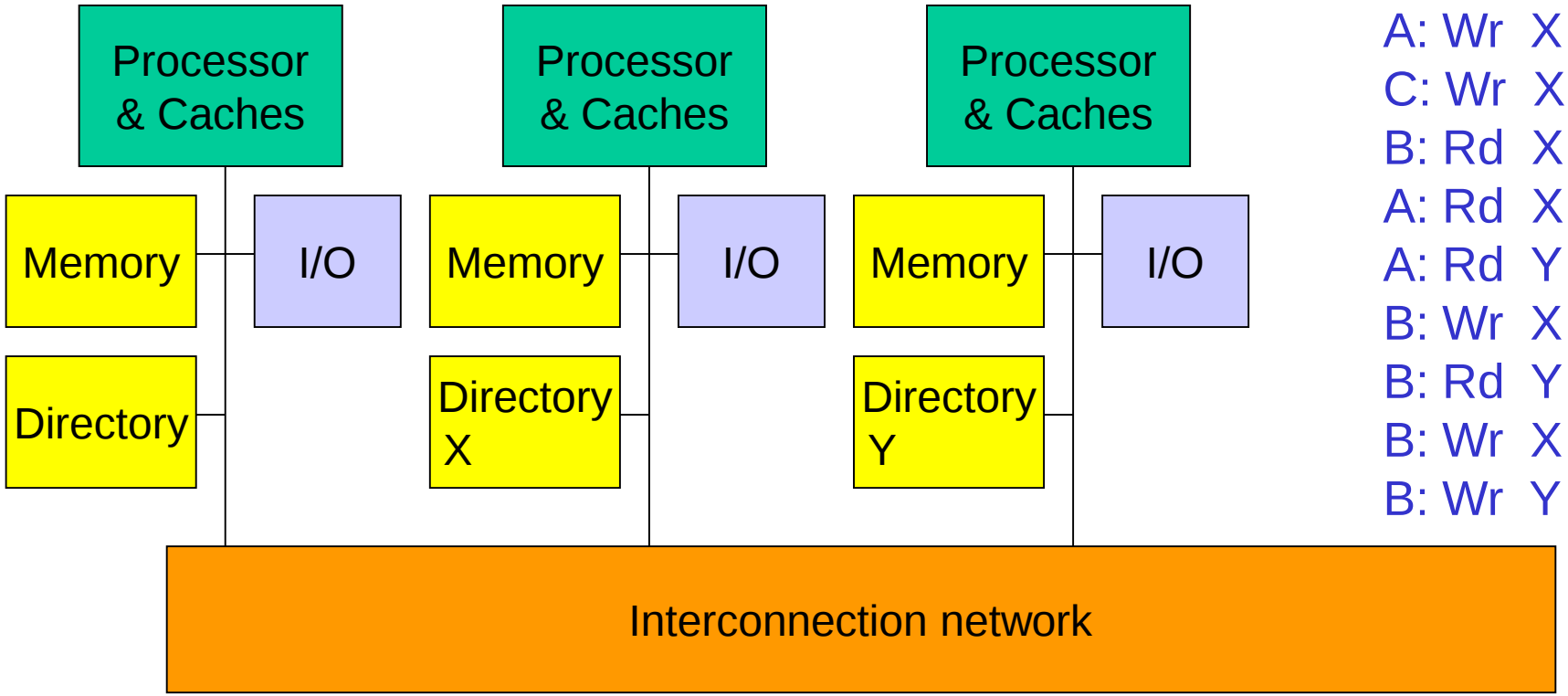
Directory-Based Cache Coherence

- The physical memory is distributed among all processors
- The directory is also distributed along with the corresponding memory
- The physical address is enough to determine the location of memory
- The (many) processing nodes are connected with a scalable interconnect (not a bus) – hence, messages are no longer broadcast, but routed from sender to receiver – since the processing nodes can no longer snoop, the directory keeps track of sharing state

Distributed Memory Multiprocessors



Directory-Based Example



A: Rd X
B: Rd X
C: Rd X
A: Wr X
A: Wr X
C: Wr X
B: Rd X
A: Rd X
A: Rd Y
B: Wr X
B: Rd Y
B: Wr X
B: Wr Y

Directory Example

	A	B	C	Dir	Comments
A: Rd X	S			S: A	Req to dir; data to A
B: Rd X	S	S		S: A, B	Req to dir; data to B
C: Rd X	S	S	S	S: A,B,C	Req to dir; data to C
A: Wr X	M	I	I	M: A	Req to dir;inv to B,C;dir recv ACKs;perms to A
A: Wr X	M	I	I	M: A	Cache hit
C: Wr X	I	I	M	M: C	Req to dir;fwd to A; sends data to dir; dir to C
B: Rd X	I	S	S	S: B, C	Req to dir;fwd to C;data to dir;dir to B; wrtbk
A: Rd X	S	S	S	S:A,B,C	Req to dir; data to A
A: Rd Y	S(Y)	S	S	X:S: A,B,C (Y:S:A)	Req to dir; data to A
B: Wr X	S(Y)	M	I	X:M:B	Req to dir; inv to A,C;dir recv ACK;perms to B
B: Rd Y	S(Y)	S(Y)	I	X: - Y:S:A,B	Req to dir; data to B; wrtbk of X
B: Wr X	S(Y)	M(X)	I	X:M:B Y:S:A,B	Req to dir; data to B
B: Wr Y	I	M(Y)	I	X: - Y:M:B	Req to dir;inv to A;dir recv ACK; perms and data to B;wrtbk of X

Performance Improvements

- What determines performance on a multiprocessor:
  - What fraction of the program is parallelizable?
  - How does memory hierarchy performance change?

- New form of cache miss: coherence miss – such a miss would not have happened if another processor did not write to the same cache line

- False coherence miss: the second processor writes to a different word in the same cache line – this miss would not have happened if the line size equaled one word