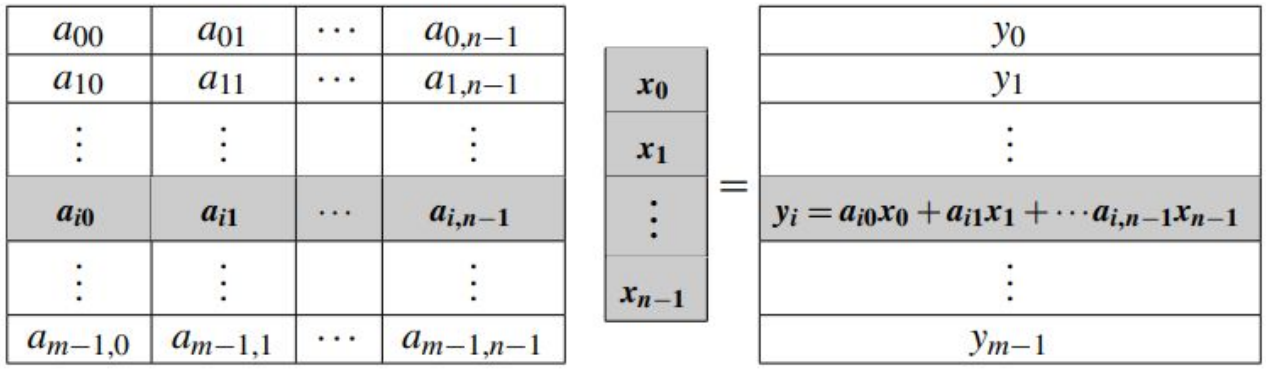
Matrix vector multiplication




```
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

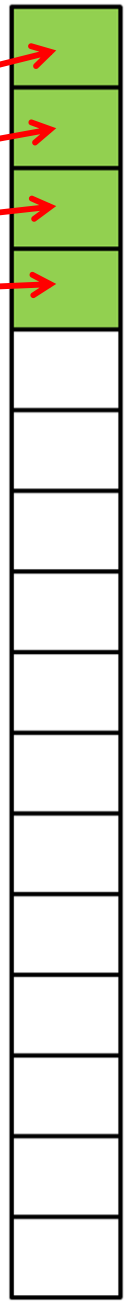
```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

- 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( );
                }

        }
```

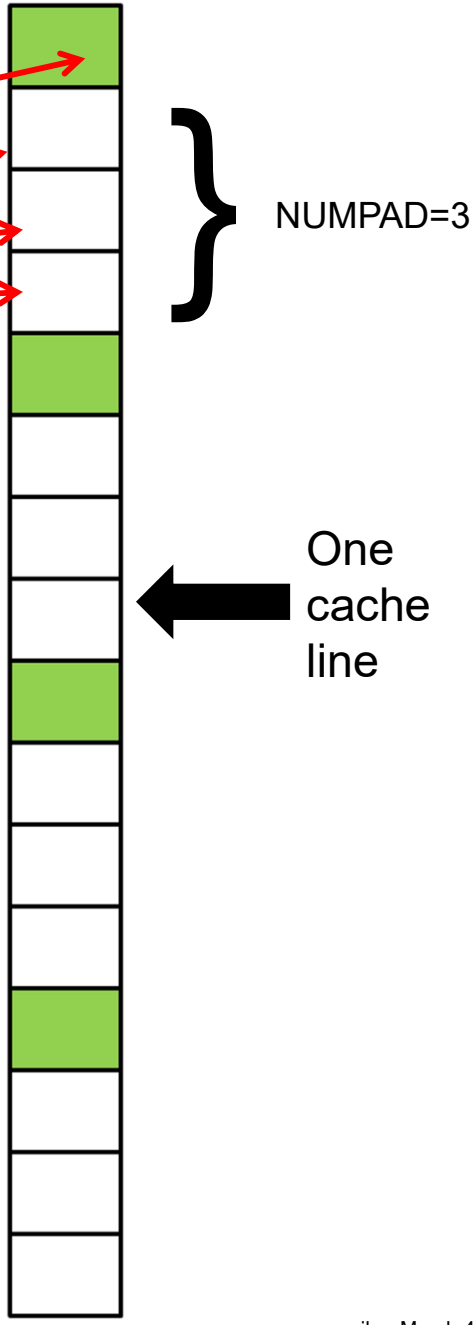
Some unpredictable function so the compiler doesn't try to optimize the j-for-loop away.



One cache line

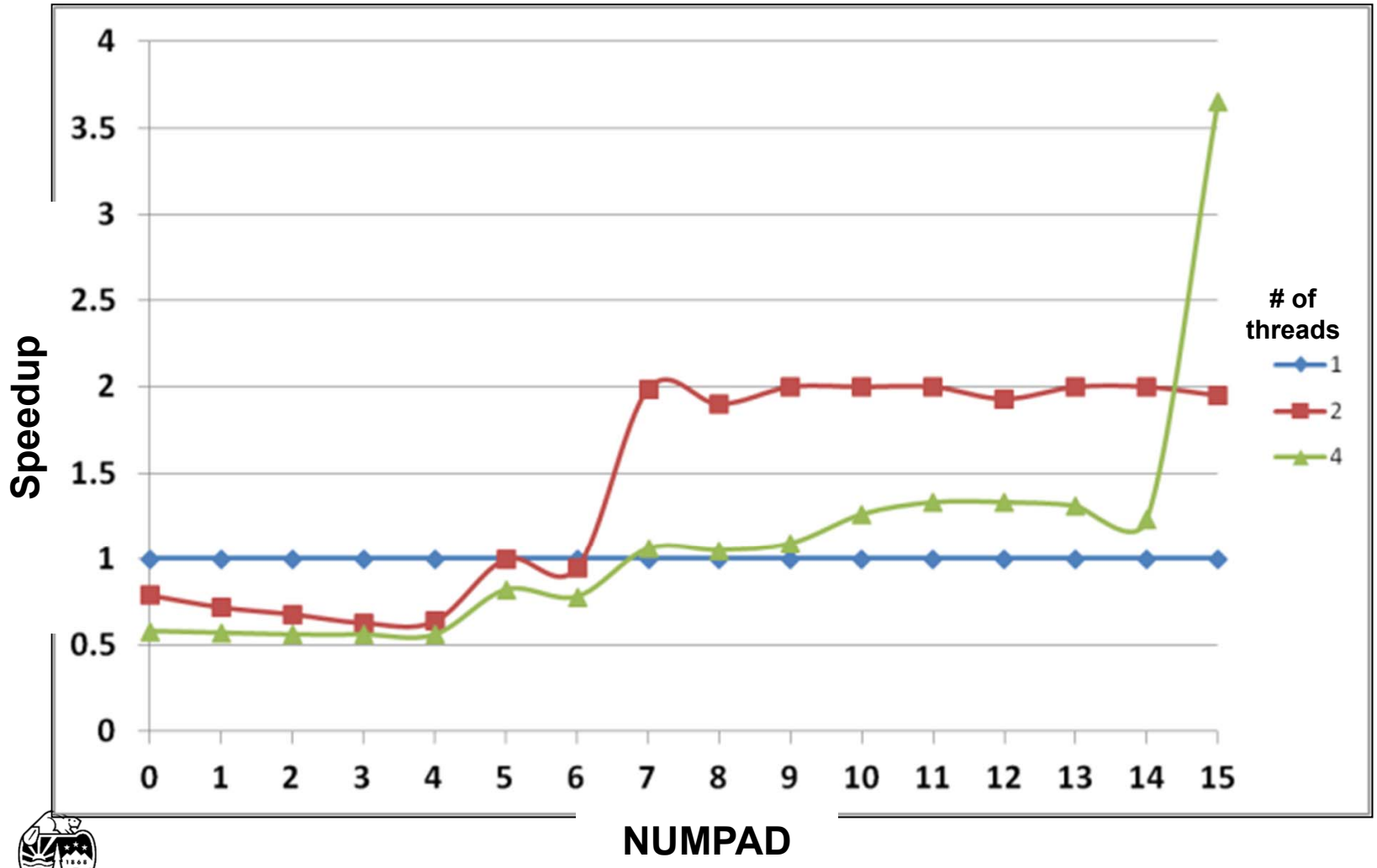
False Sharing – Fix #1
Adding some padding

```
#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

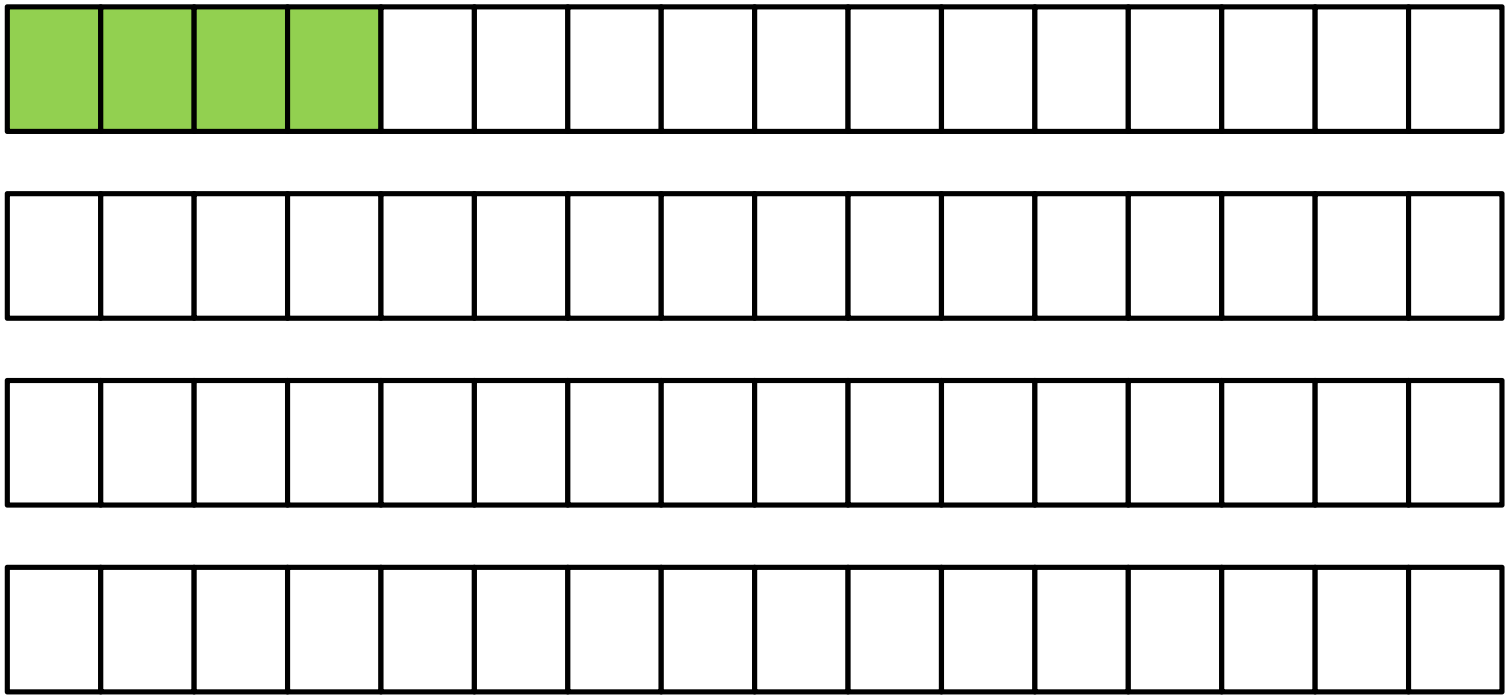


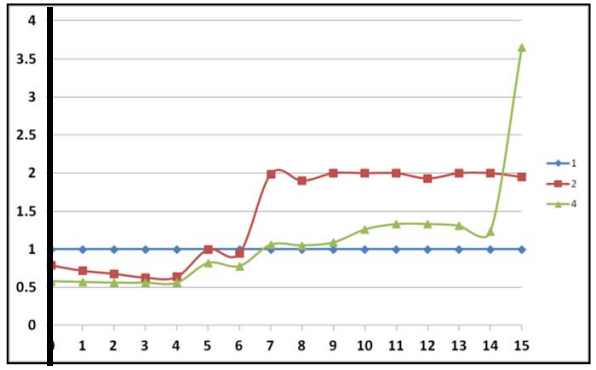
Oregon State University
Computer Graphics

Why do these curves look this way?

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

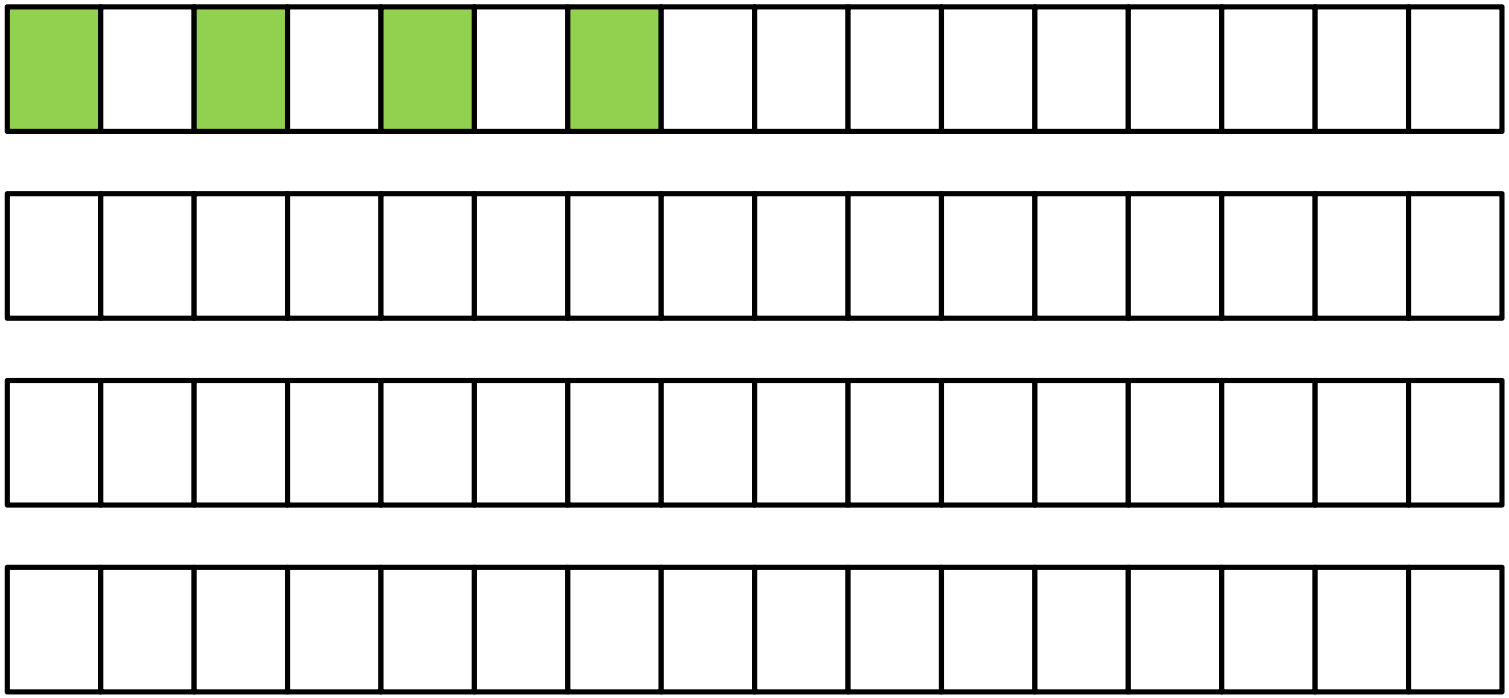
NUMPAD = 0

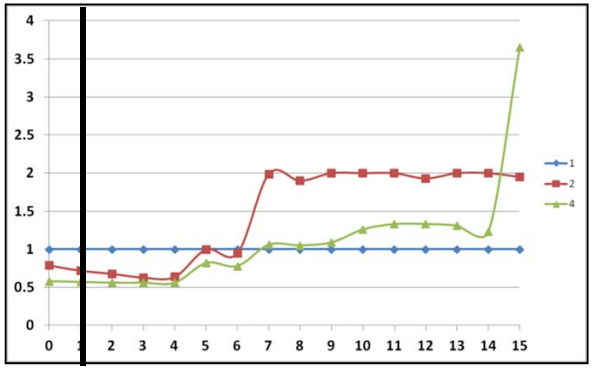




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

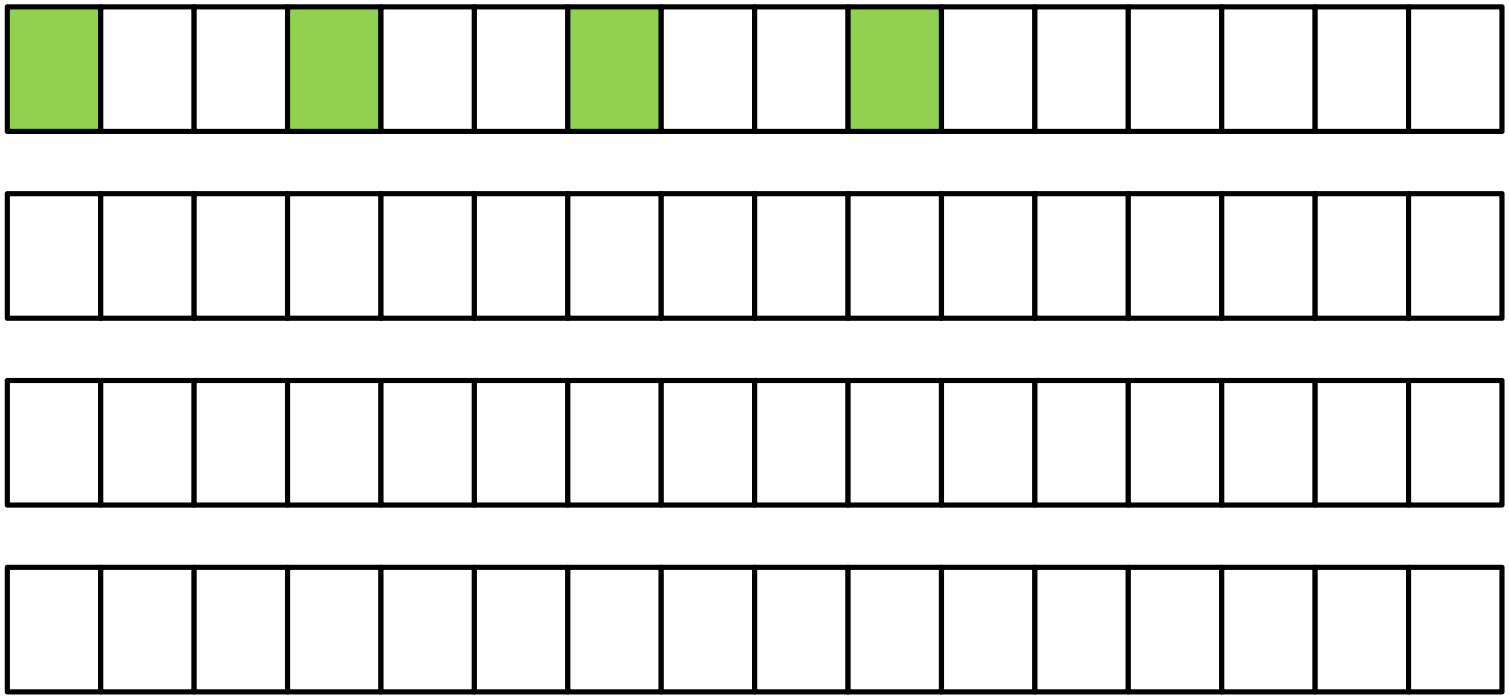
NUMPAD = 1

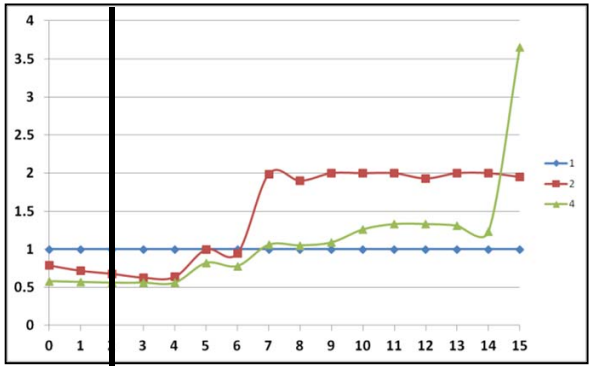




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

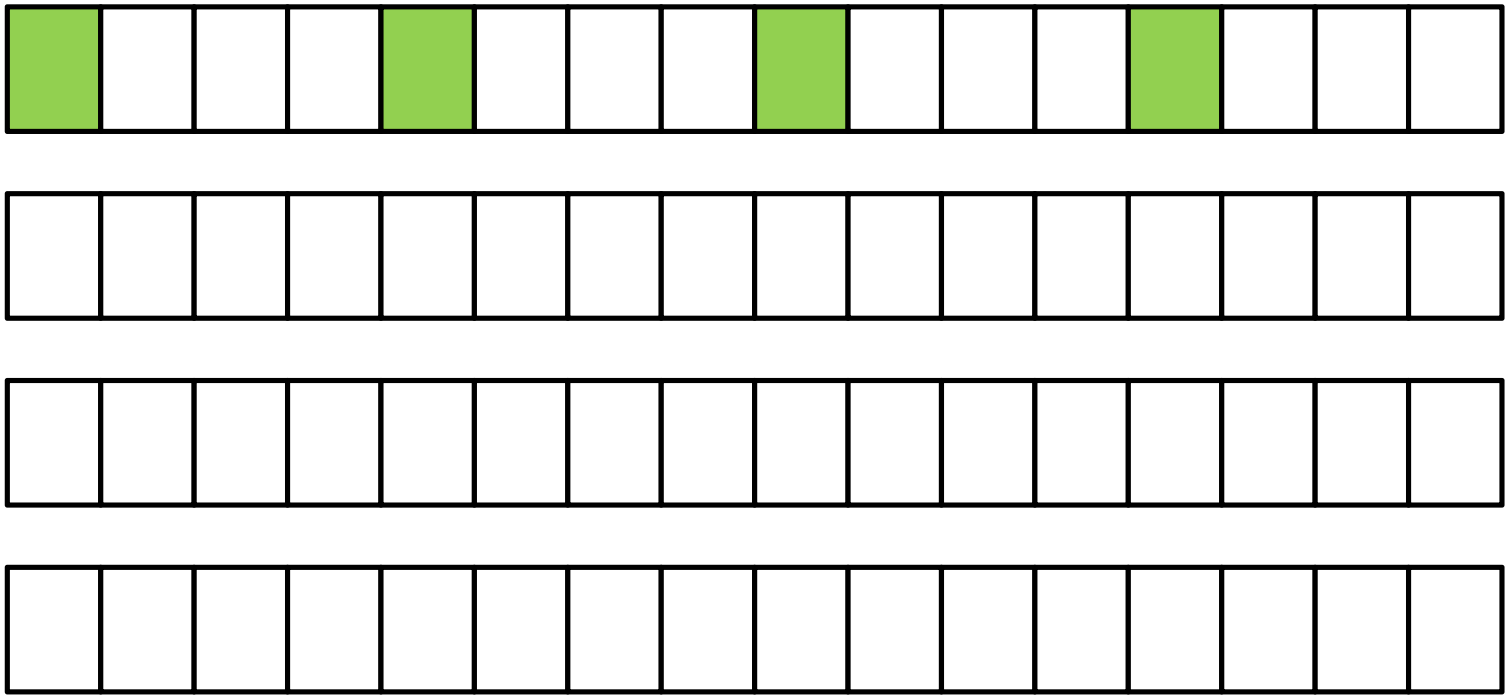
NUMPAD = 2

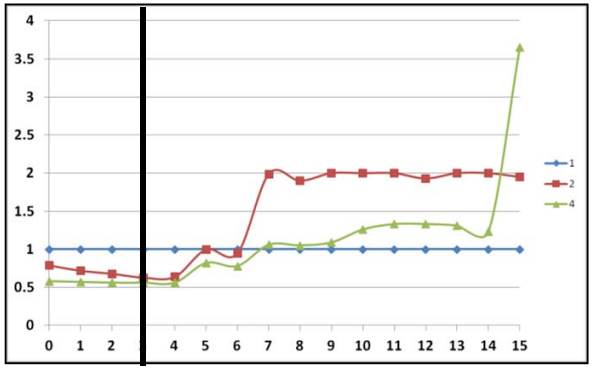




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

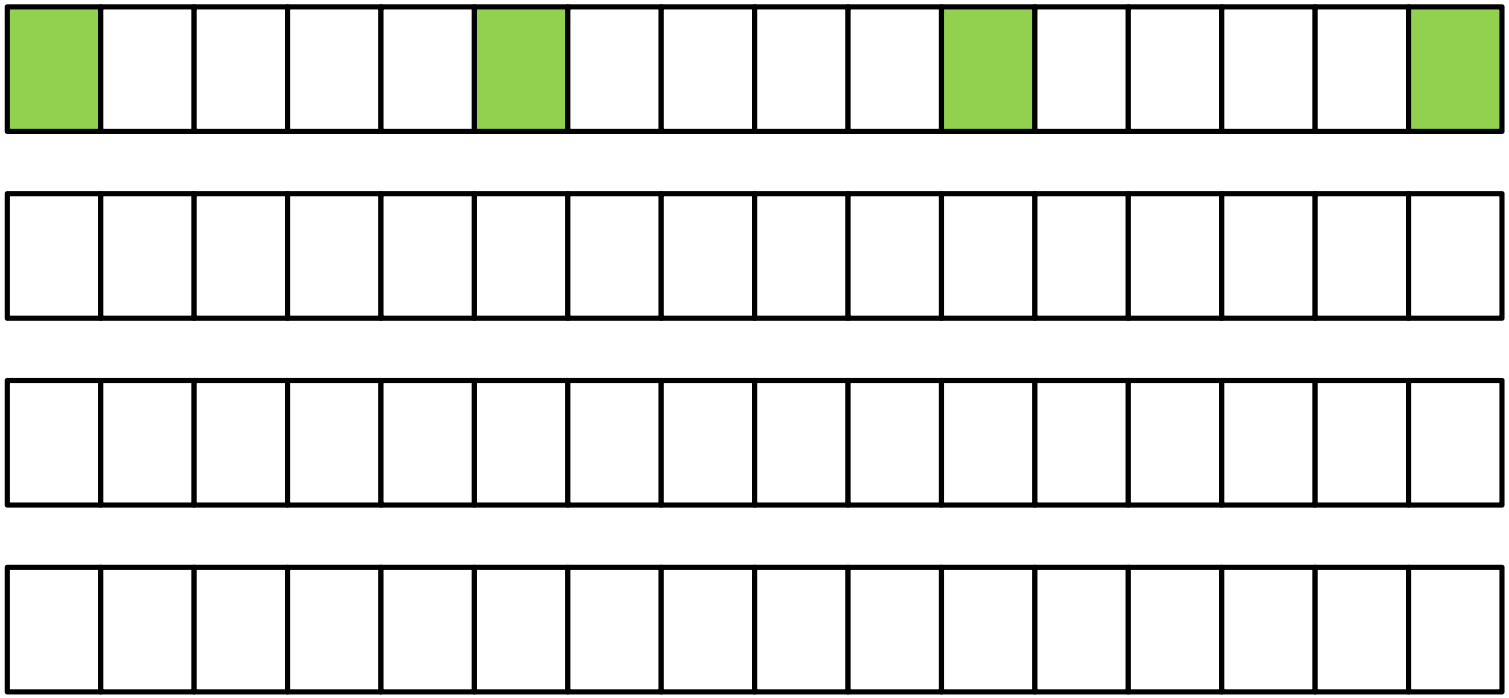
NUMPAD = 3

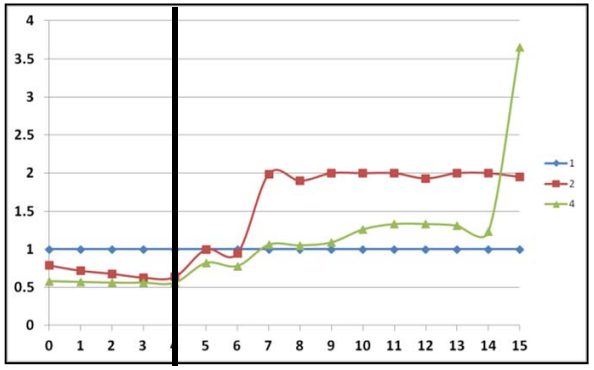




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

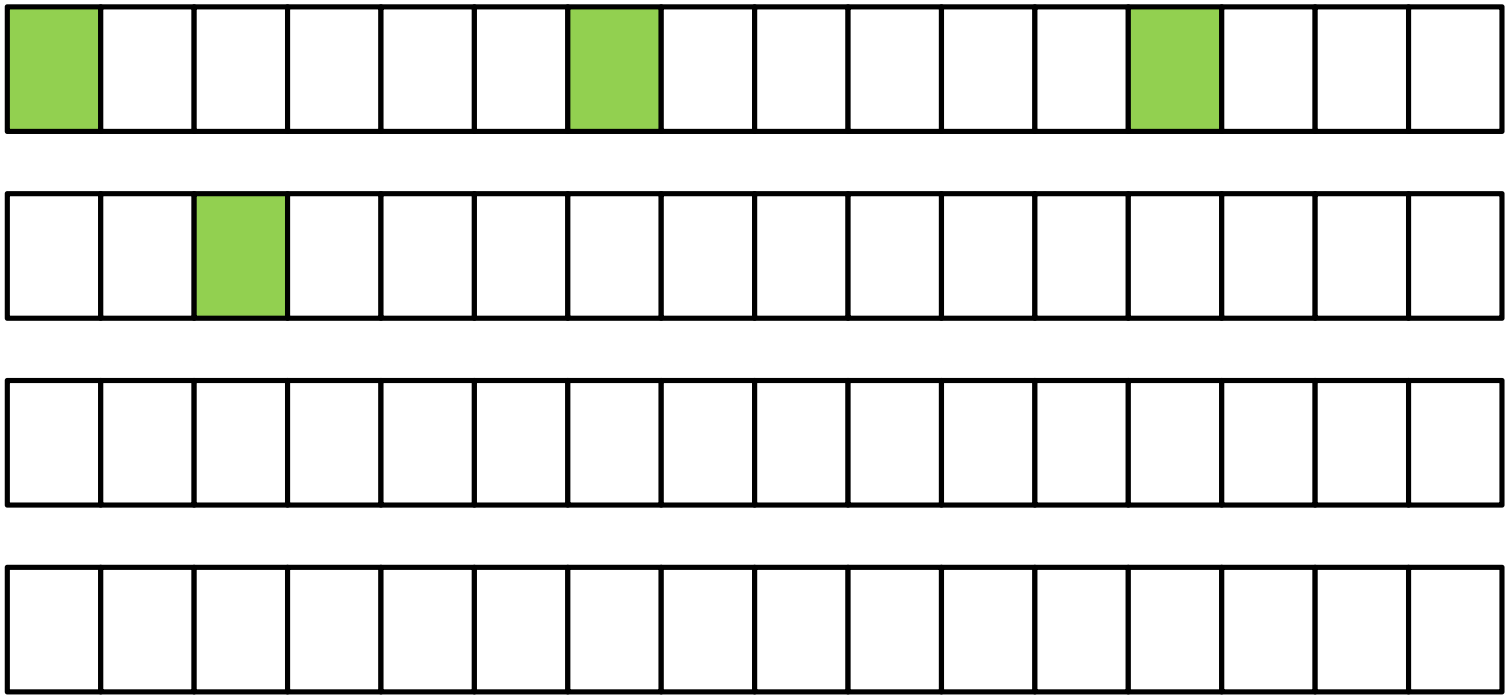
NUMPAD = 4

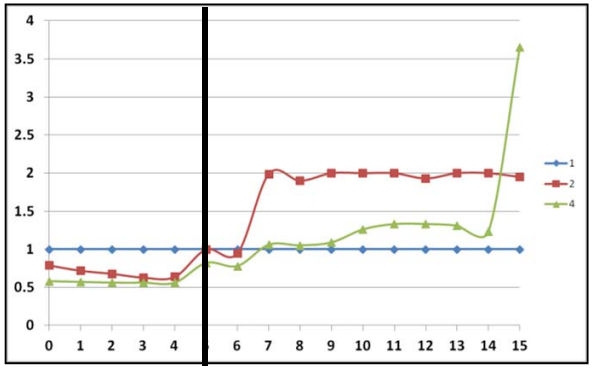




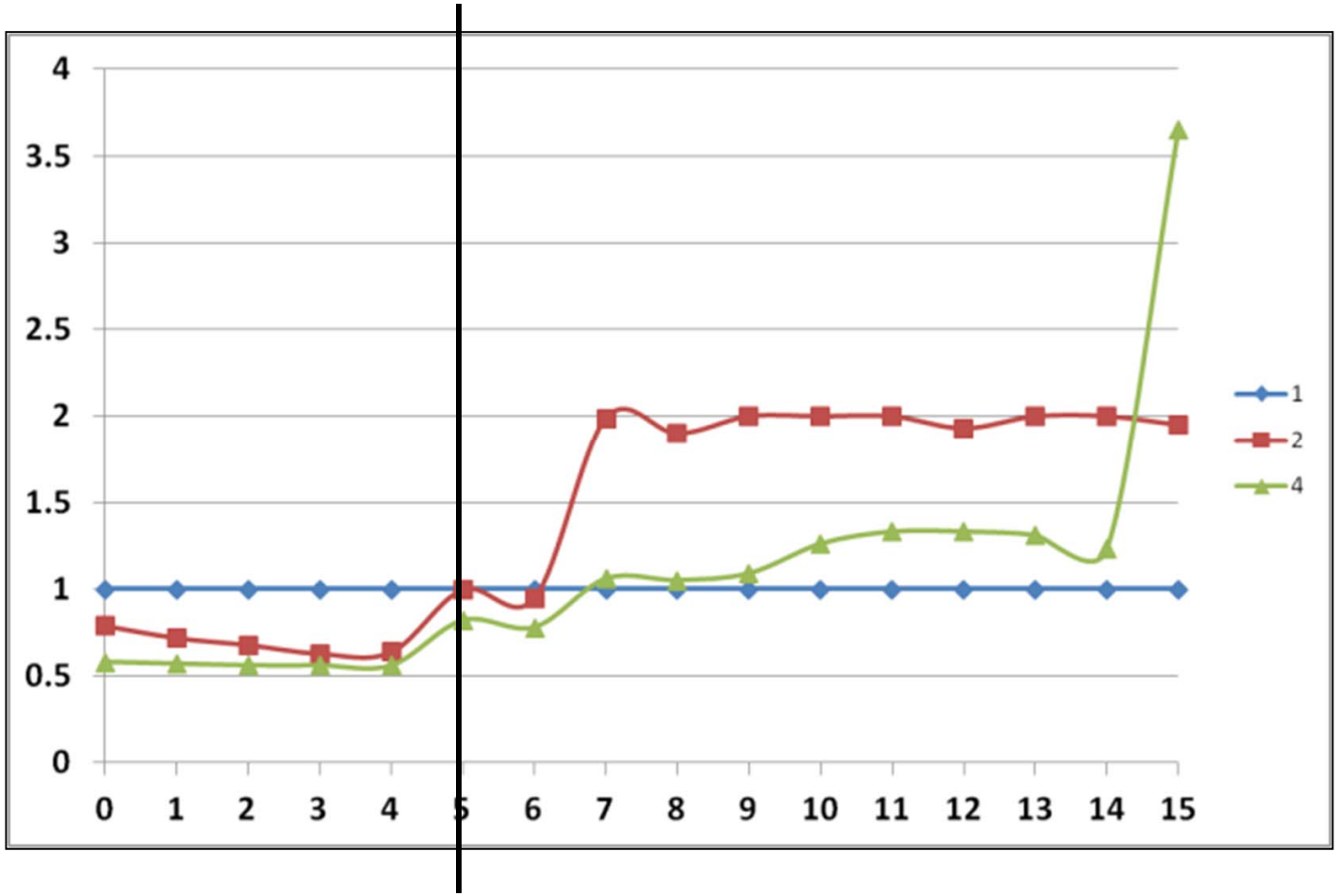
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 5



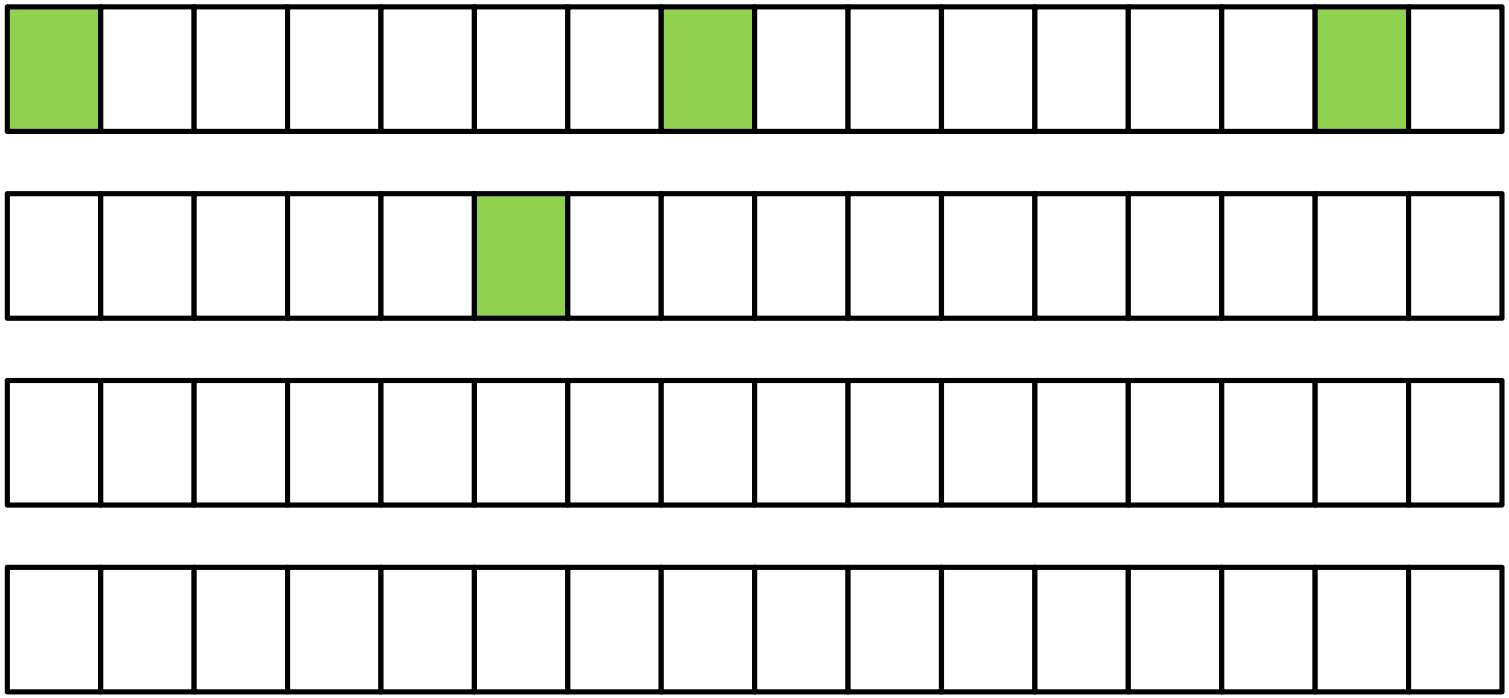


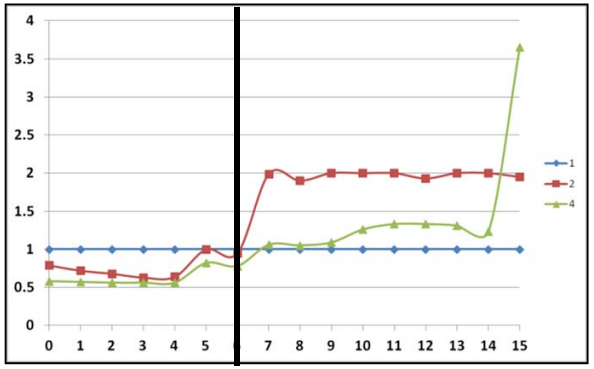
False Sharing – Fix #1



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

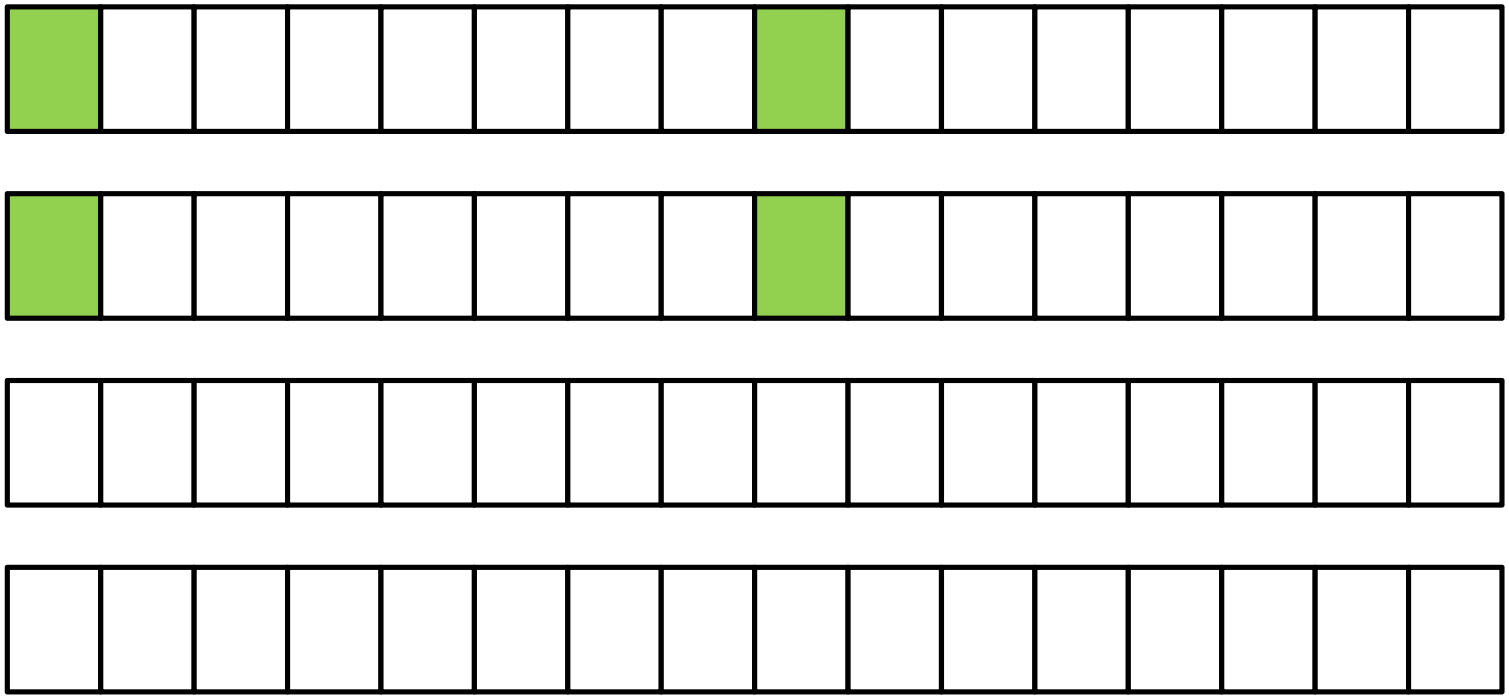
NUMPAD = 6

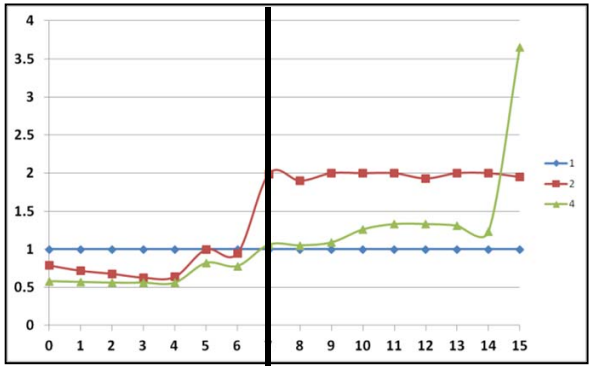




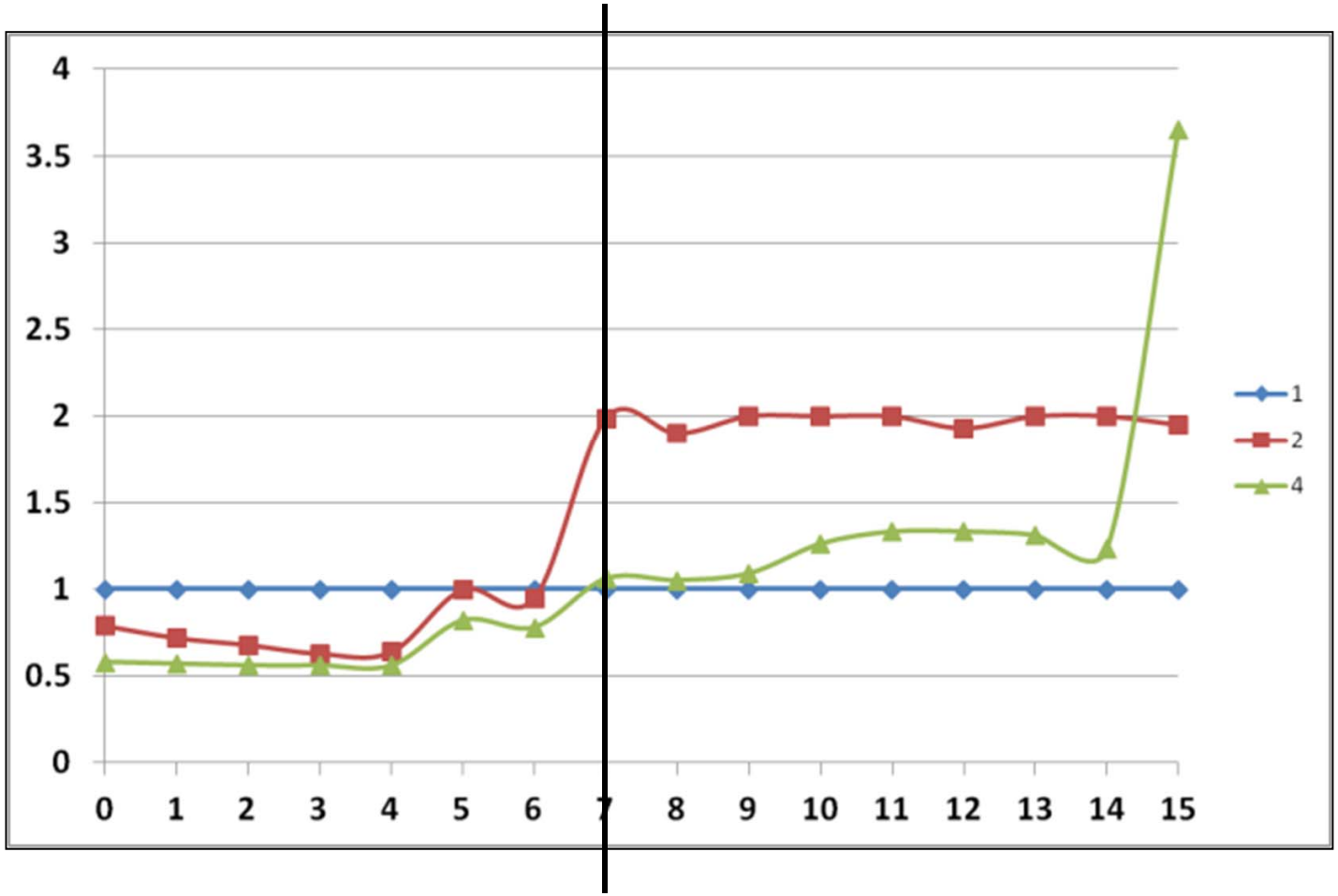
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 7



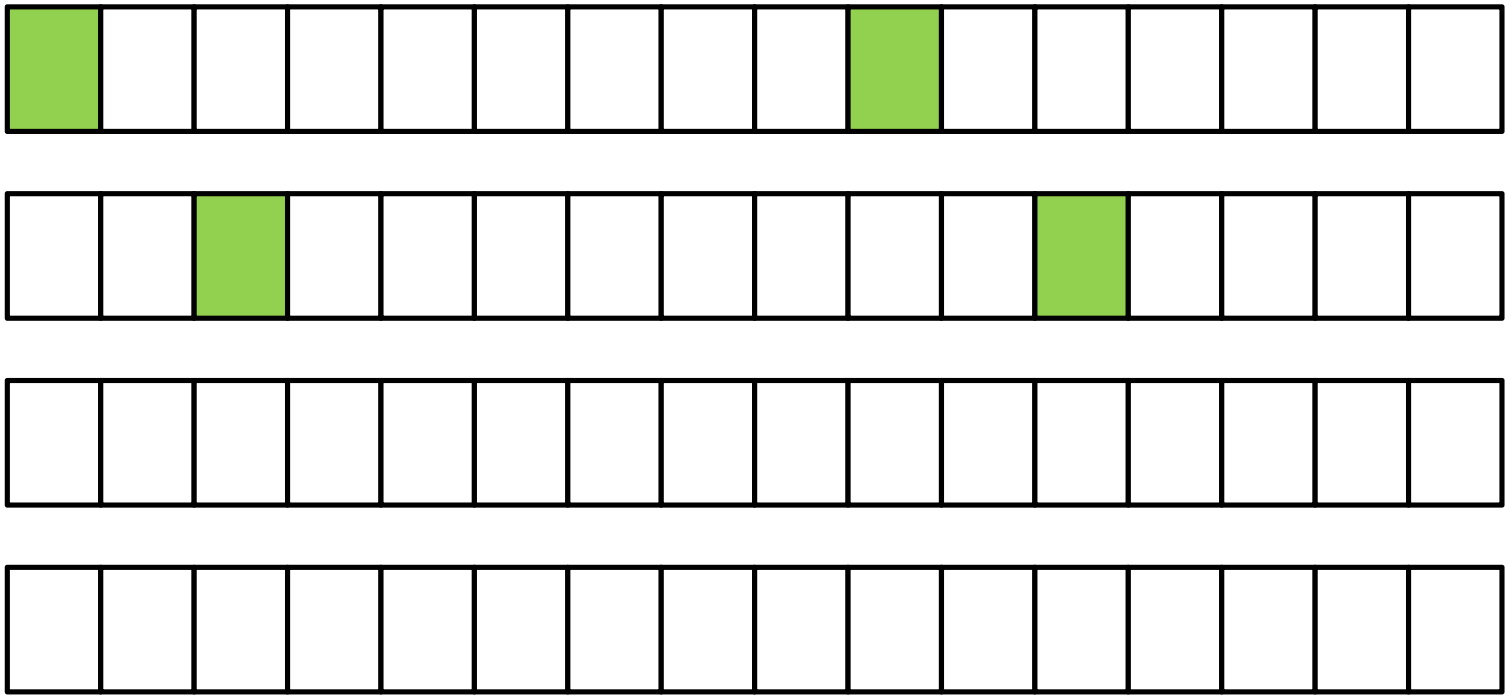


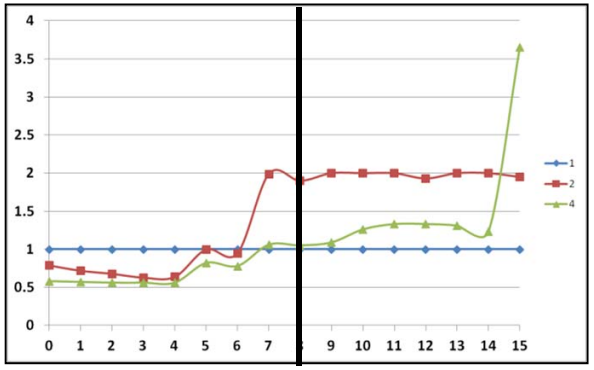
False Sharing – Fix #1



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

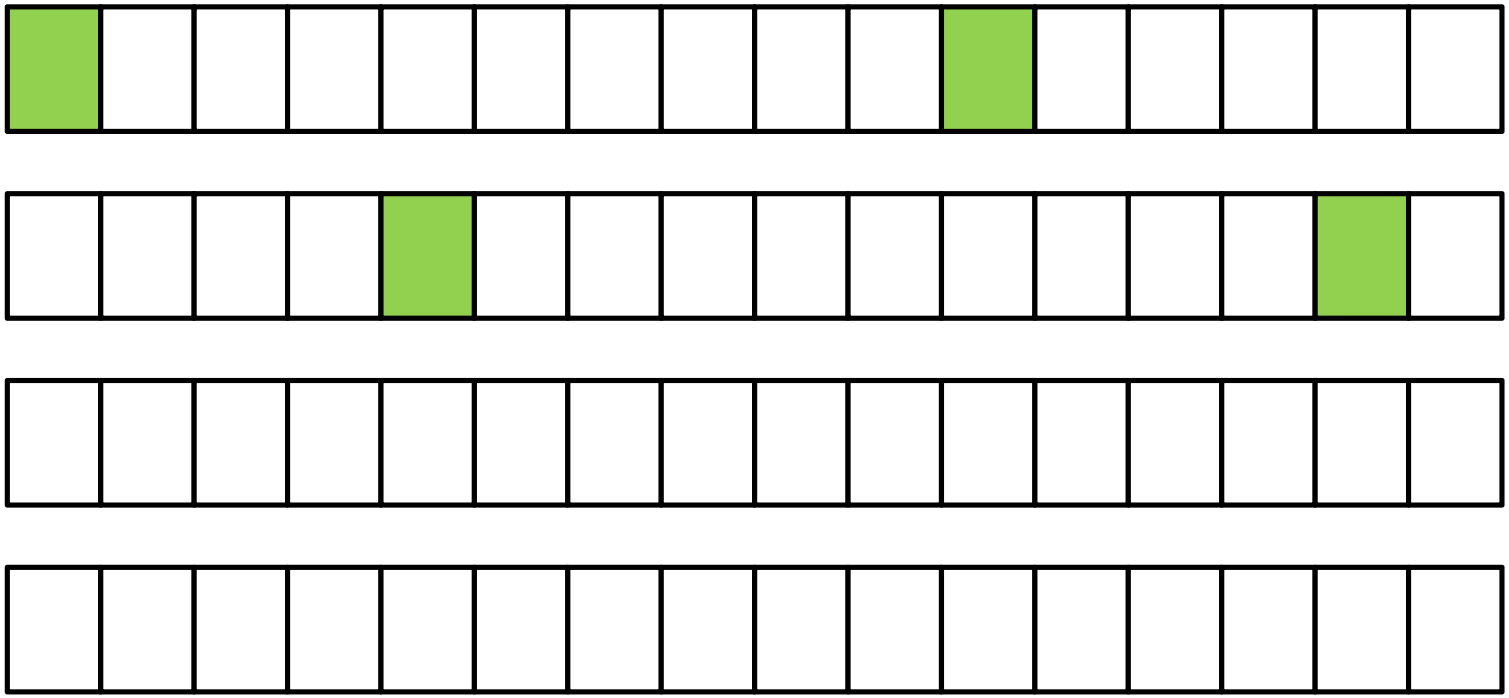
NUMPAD = 8

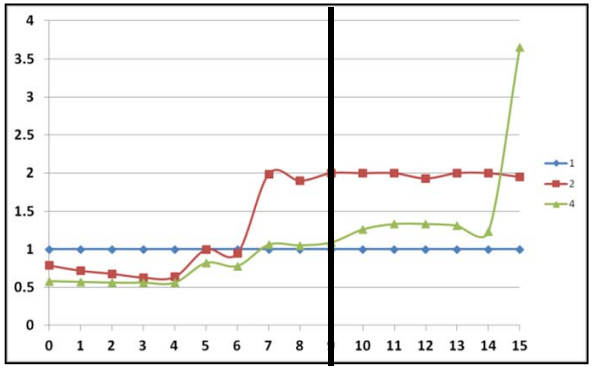




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

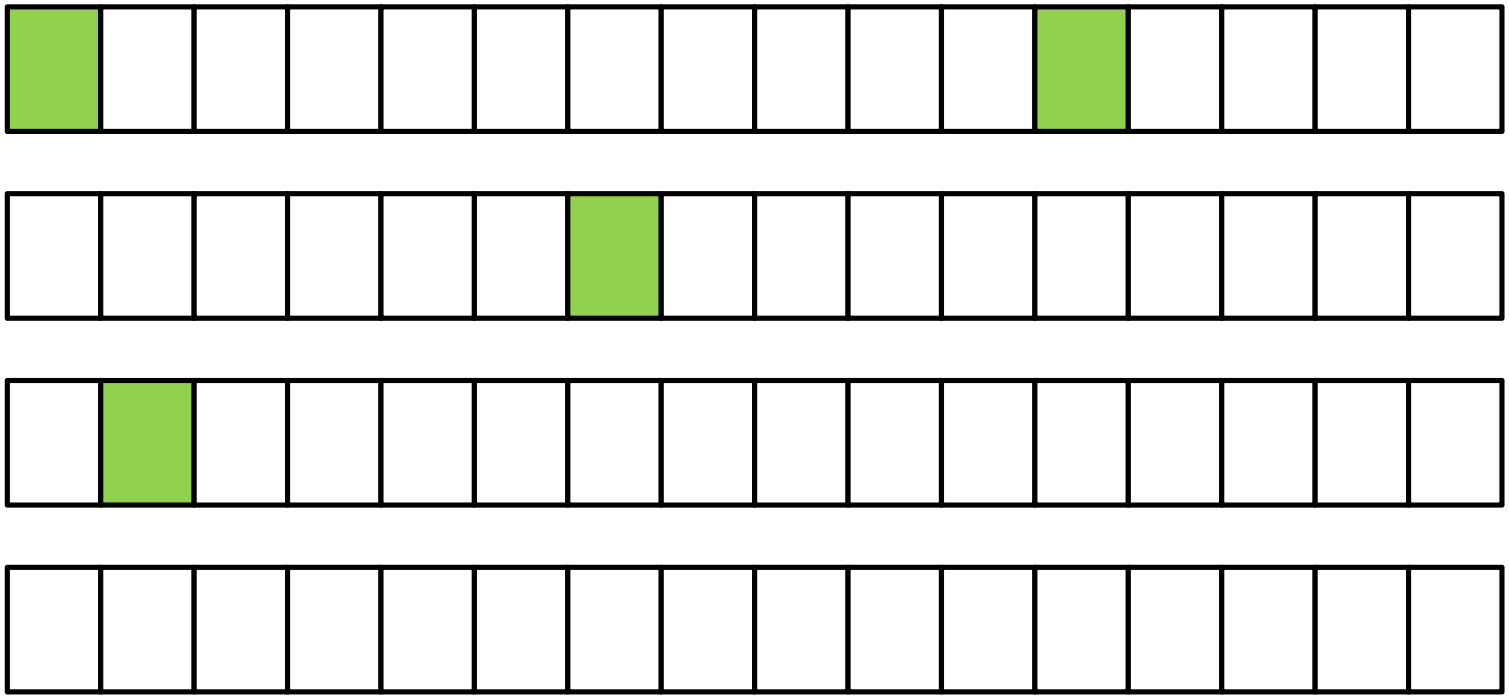
NUMPAD = 9

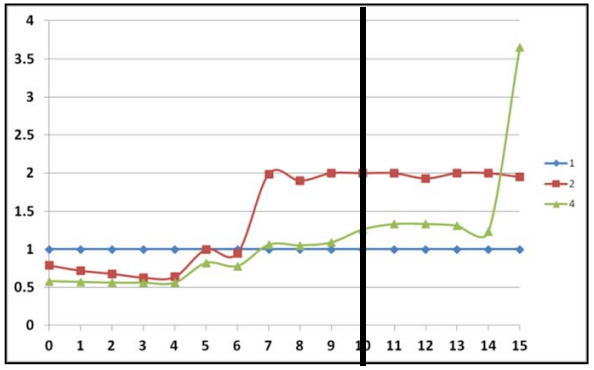




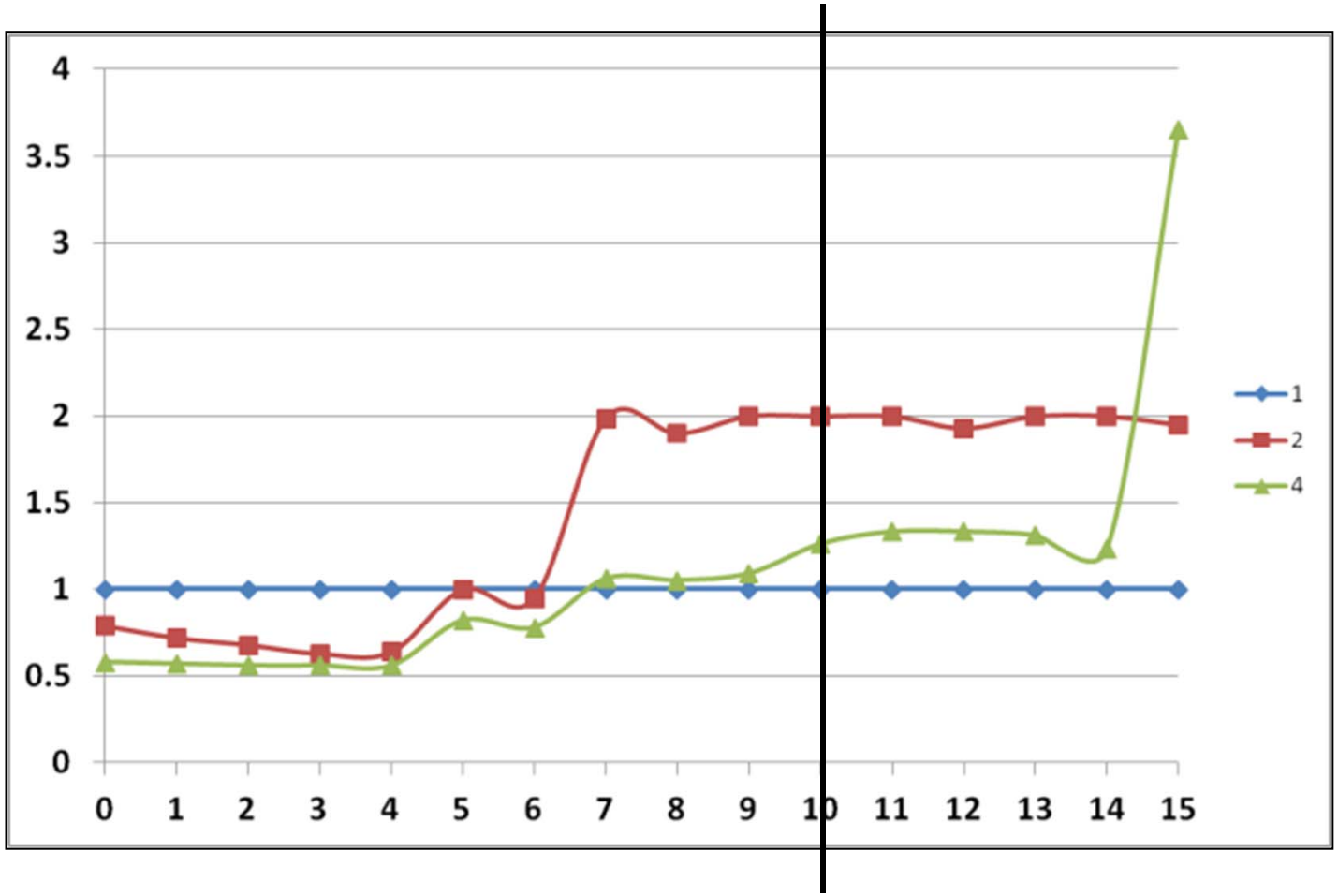
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 10



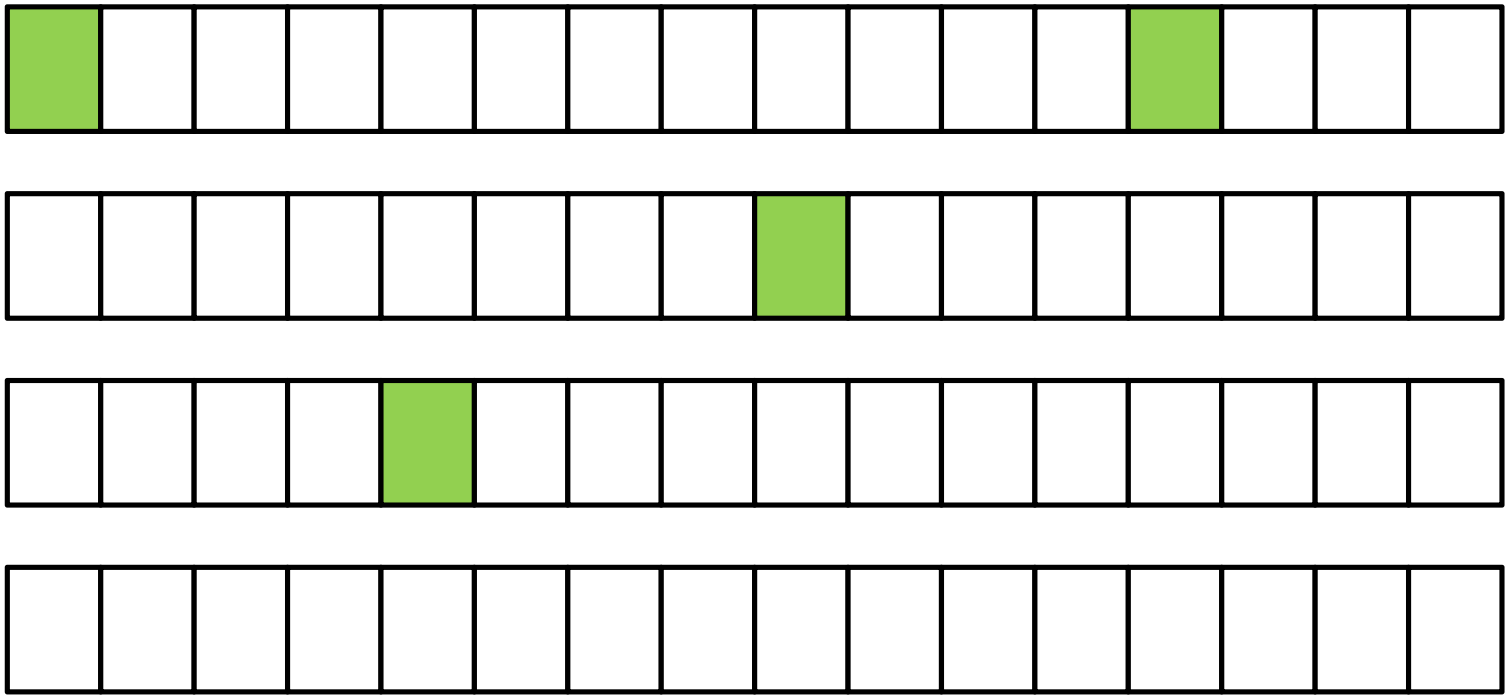


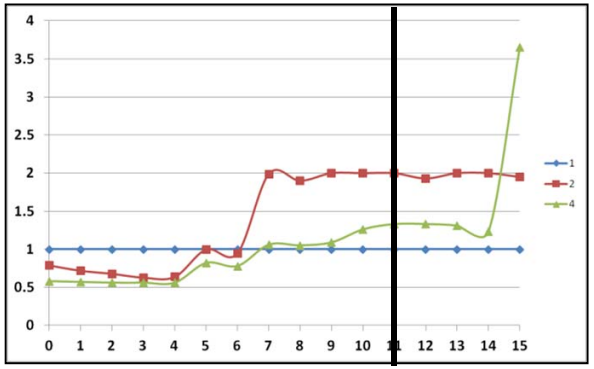
False Sharing – Fix #1



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

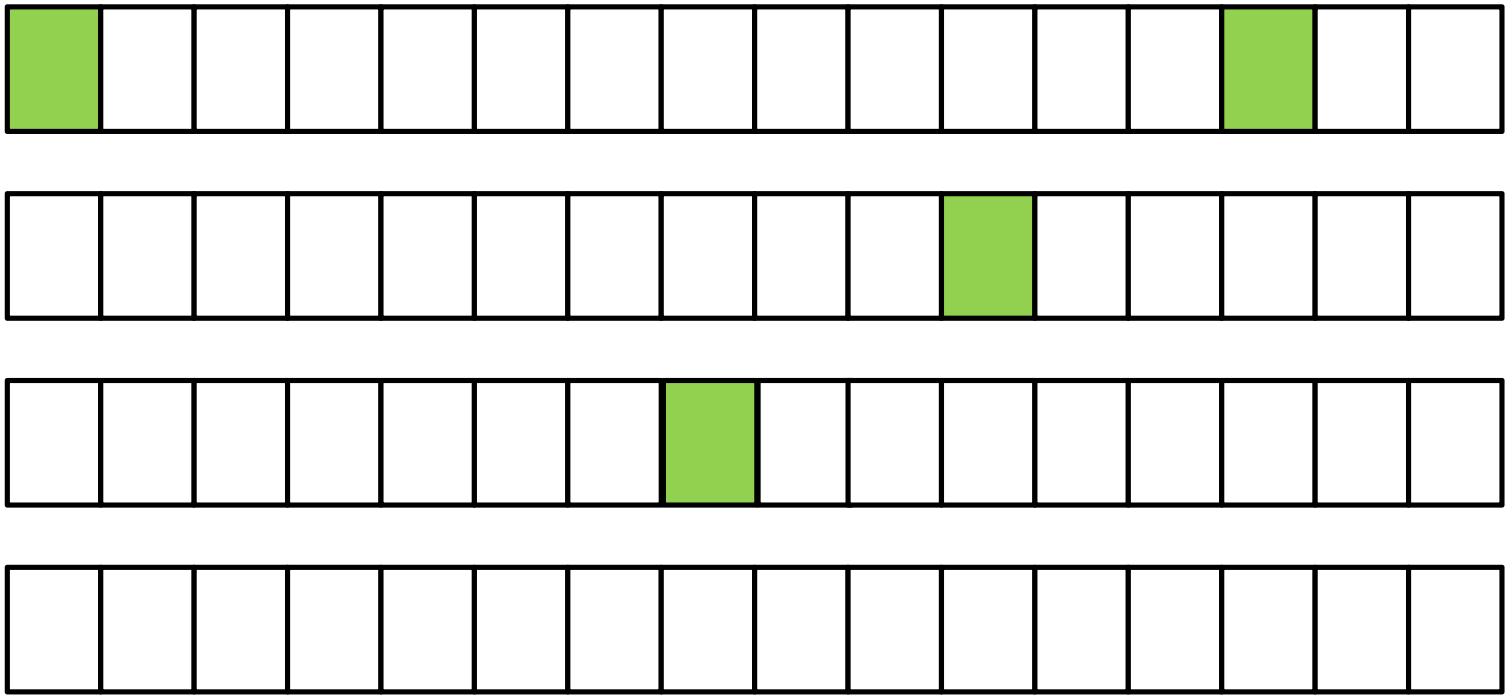
NUMPAD = 11

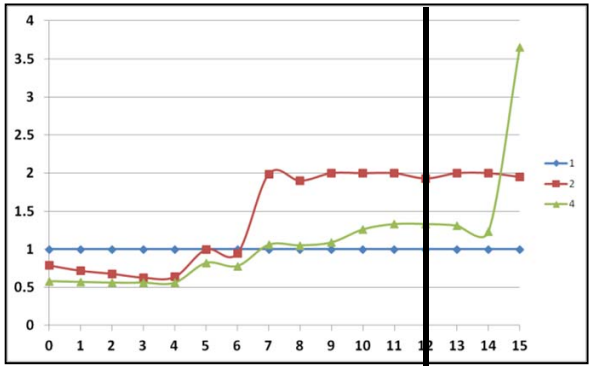




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

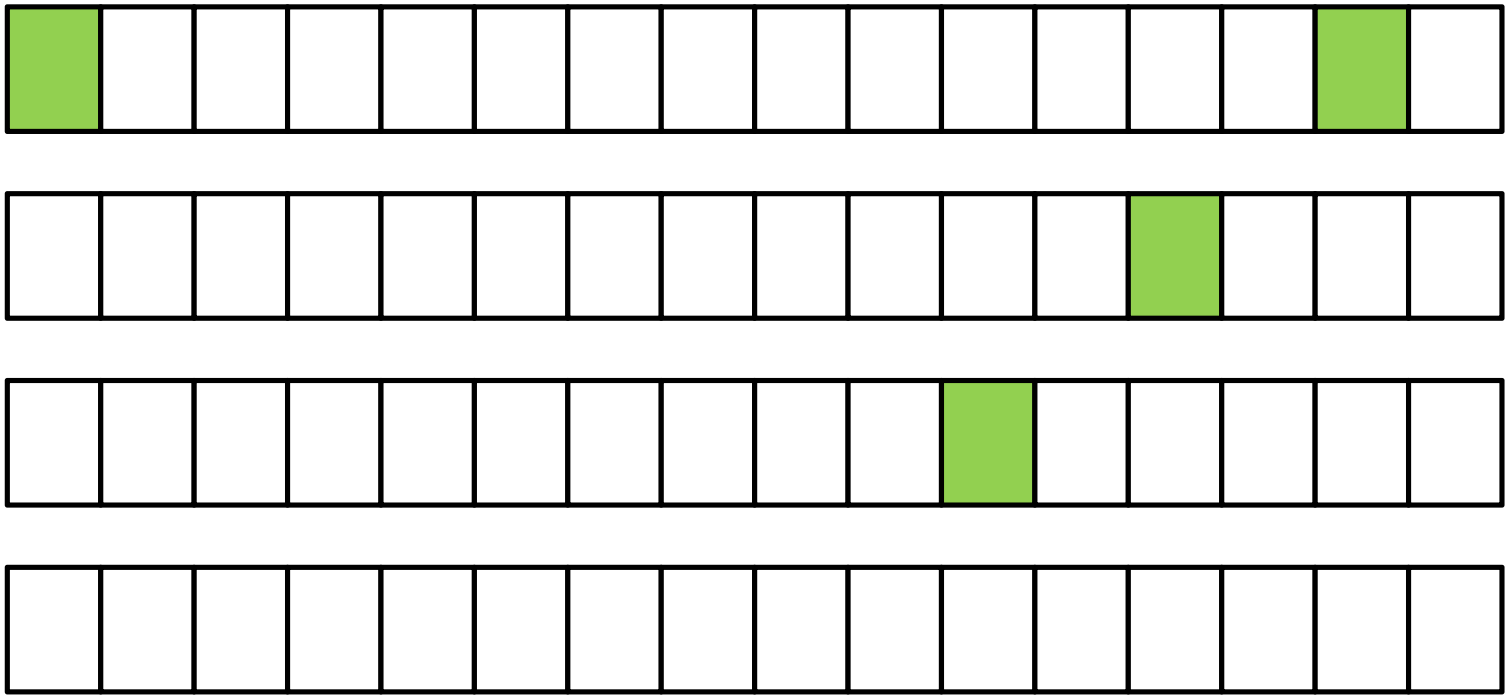
NUMPAD = 12

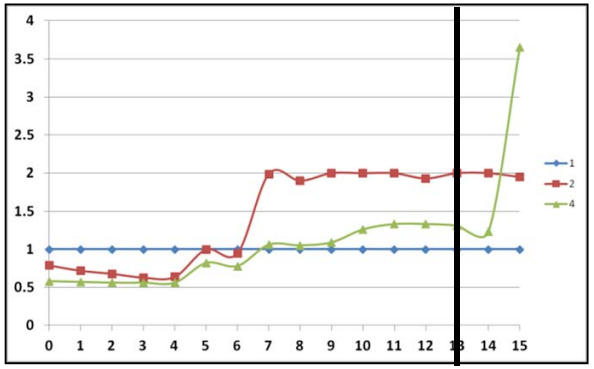




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

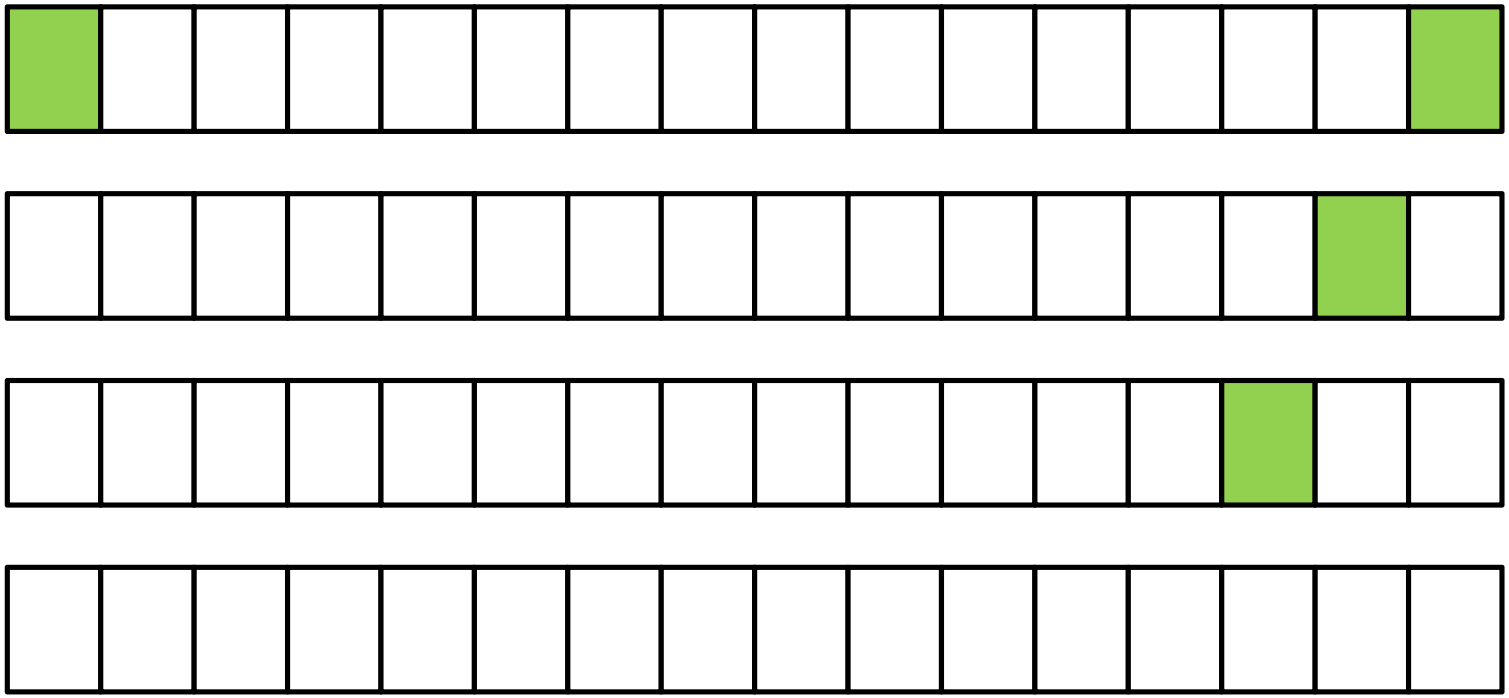
NUMPAD = 13

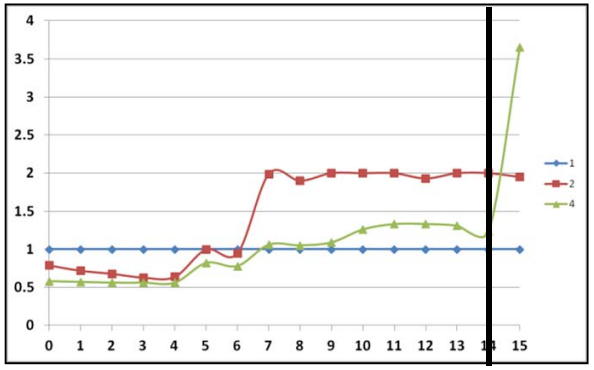




False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

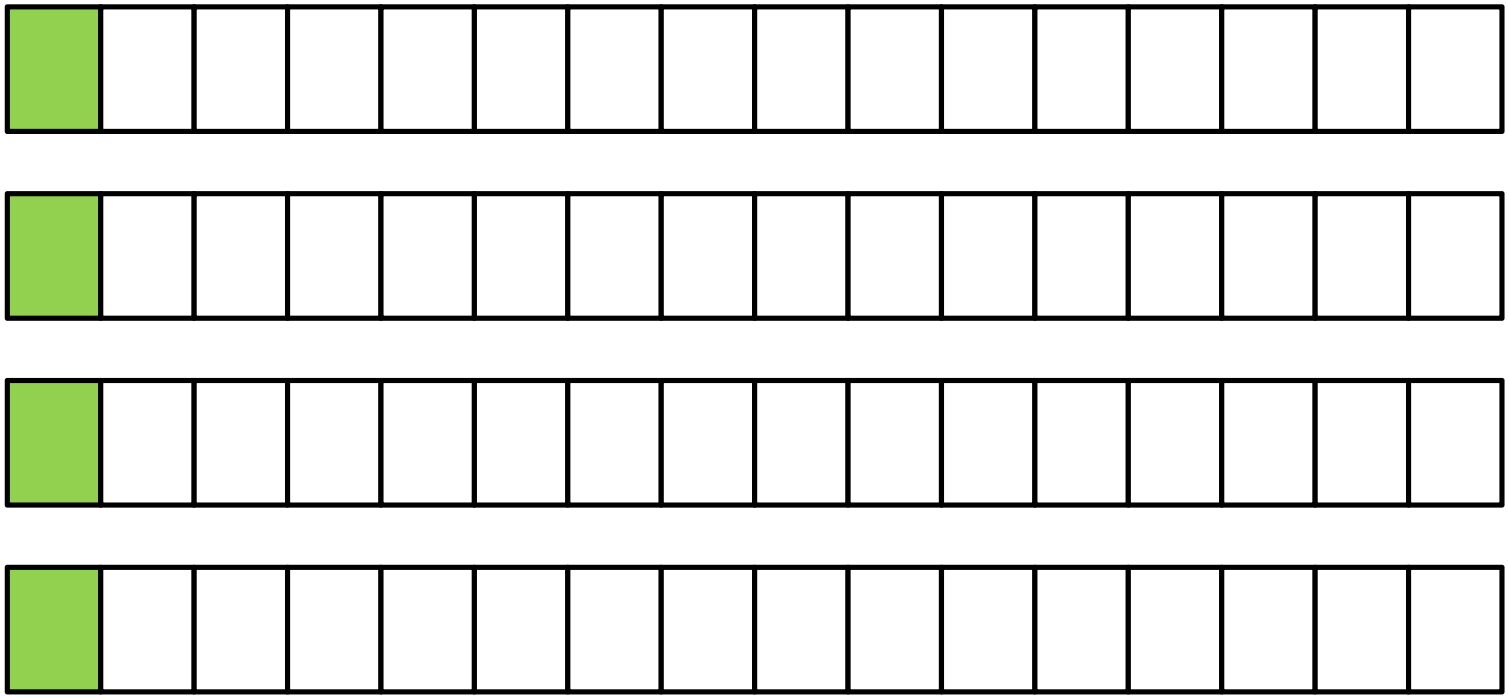
NUMPAD = 14

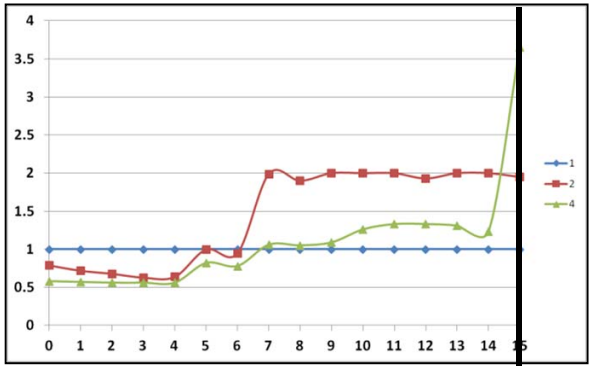




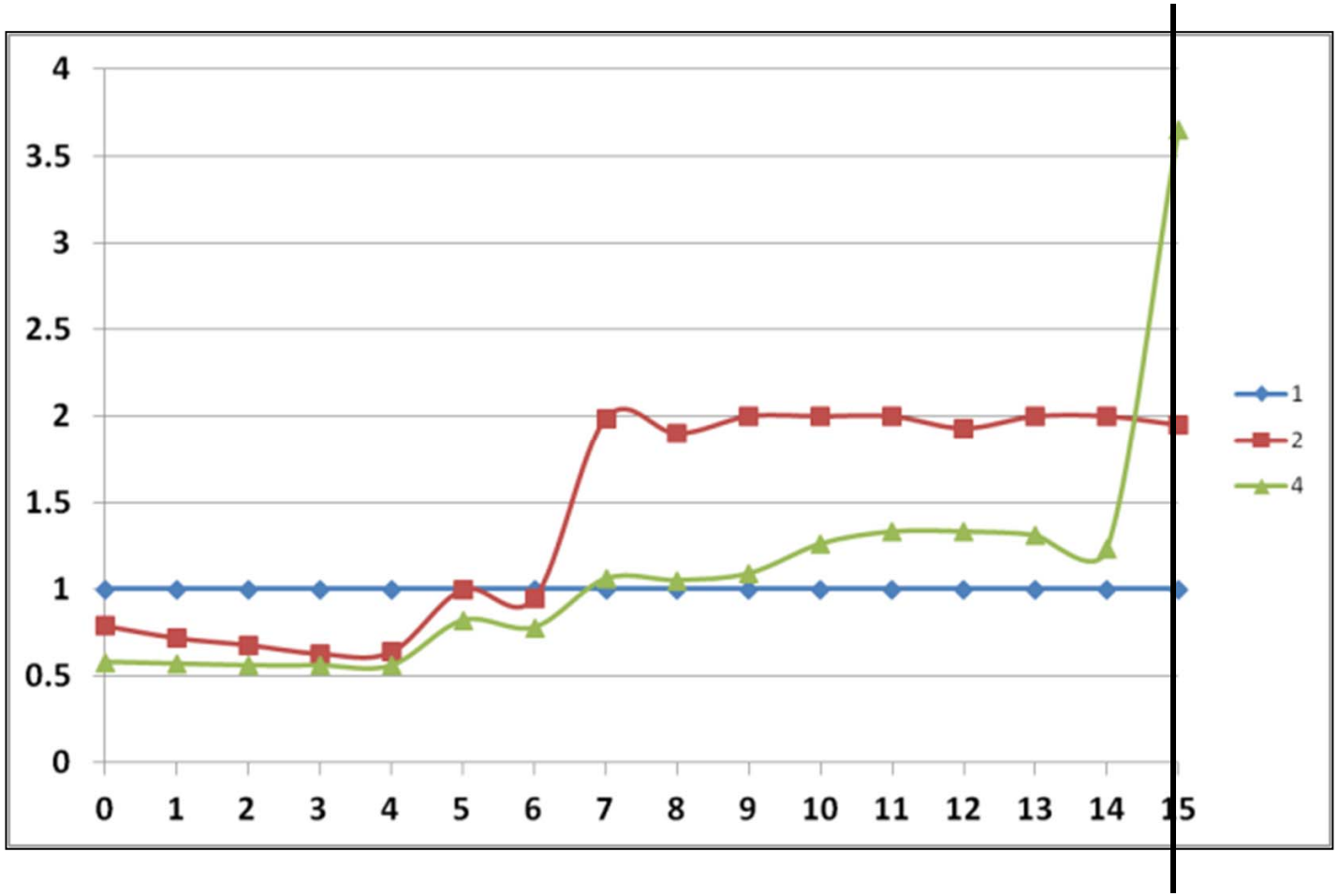
False Sharing – the Effect of Spreading Your Data to Multiple Cache Lines

NUMPAD = 15





False Sharing – Fix #1

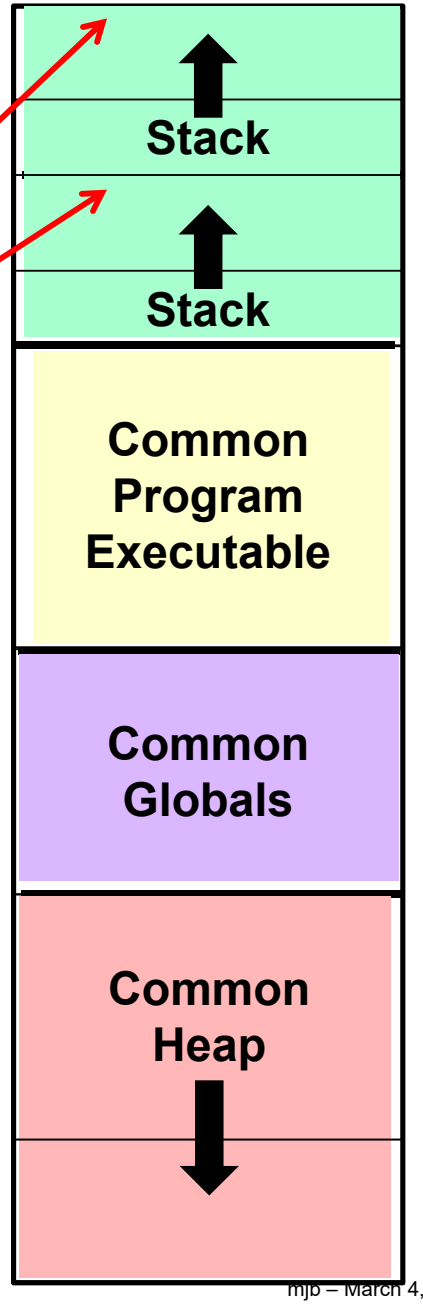


False Sharing – Fix #2: Using local (private) variables

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?

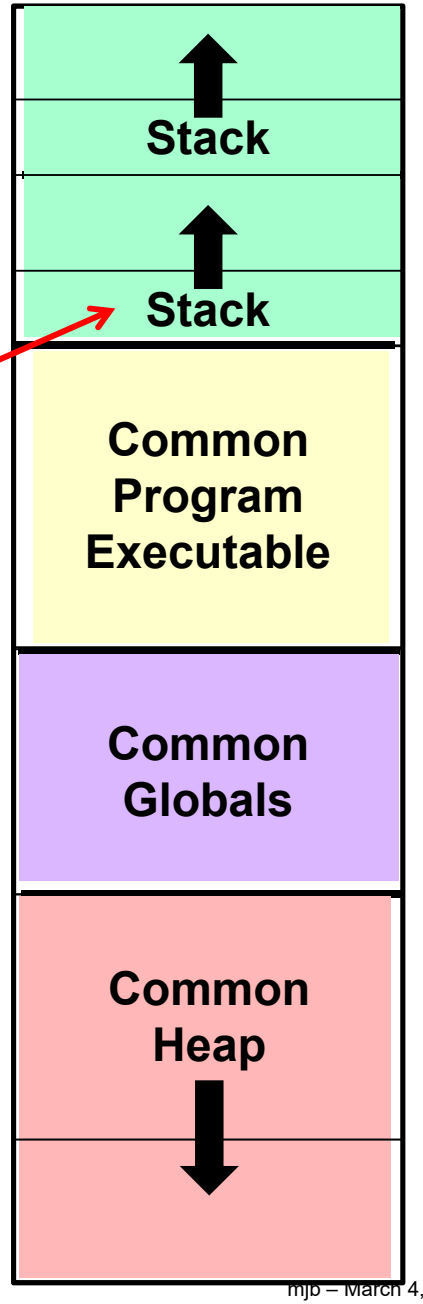
If we use local variables, instead of contiguous array locations, that will spread our writes out in memory, and to different cache lines.



False Sharing – Fix #2

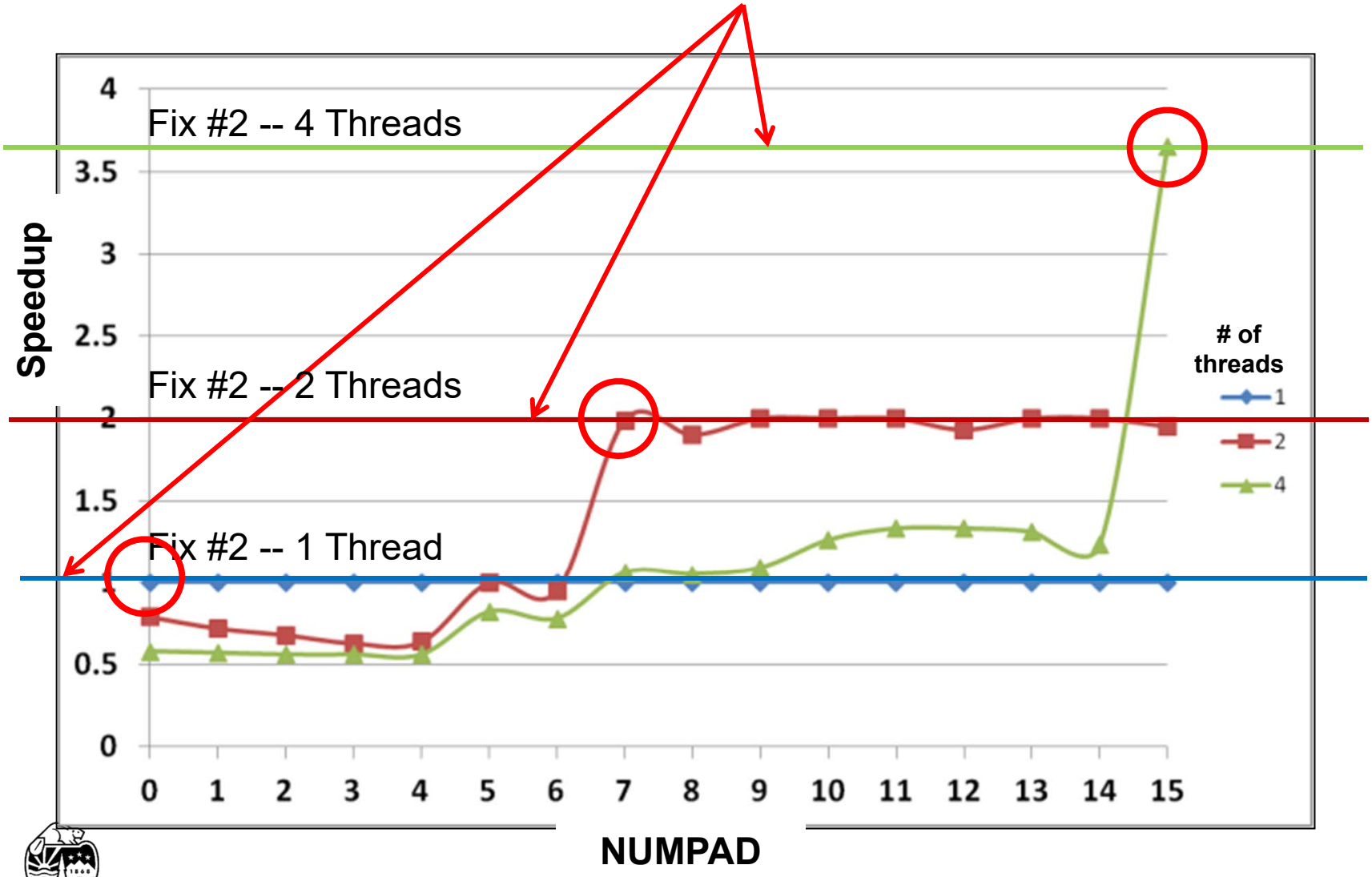
```
#include <stdlib.h>
struct s
{
        float value;
}   Array[4];

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}