CACHES,CACHE COHERENCE, & FALSE SHARING

- Processors will be able to execute operations much faster than they can access data in main memory so if a processor must read data from main memory for each operation, it will spend most of its time simply waiting for the data from memory to arrive. chip designers have added blocks of relatively fast memory to processors. This faster memory is called ***cache memory.***

- Suppose ***x*** is a shared variable with the value ***5***, and both thread 0 and thread 1 read x from memory into their (separate) caches, because both want to execute the statement

```
my_y = x;
```

- Here, my_ y is a private variable defined by both threads. Now suppose thread 0 executes the statement

```
x++;
```

  Finally, suppose that thread 1 now executes

```
my_z = x;
```

  wheremy_z is another private variable.

- There are (at least) three copies of x: the one in main memory, the one in thread 0's cache, and the one in thread 1's cache.This is the ***cache coherence***problem,
- The use of cache coherence can have a dramatic effect on the performance of shared-memory systems. To illustrate this, let's take a look at matrix-vector multiplication.
  If A =(aij) is an mxn matrix and x is a vector with ***n*** components,
  then their product y = Ax is a vector with ***m*** components, and its i^{th} component yi isfound by forming the dot product of the i^{th} row of A with x:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}.$$

- So if we store A as a two-dimensional array and x and y as one-dimensional arrays, we can implement serial matrix-vector multiplication with the following code:

```
for (i = 0; i < m; i++) {
        y[i] = 0.0;
for (j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

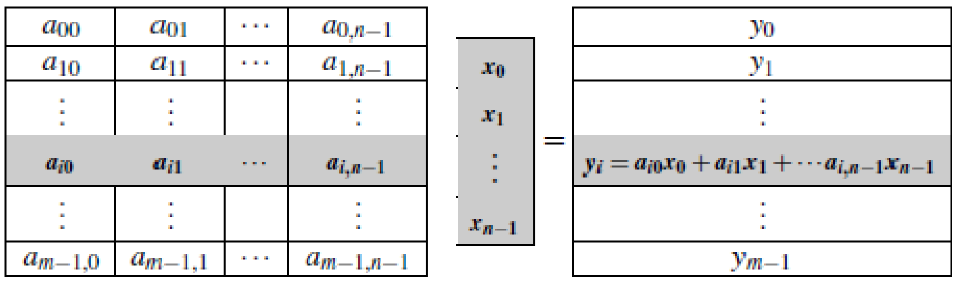


Fig: Matrix-vector multiplication

- There are no loop-carried dependences in the outer loop, since A and x are never updated and iteration i only updates y[i]. Thus, we can parallelize this by dividing the iterations in the outer loop among the threads:

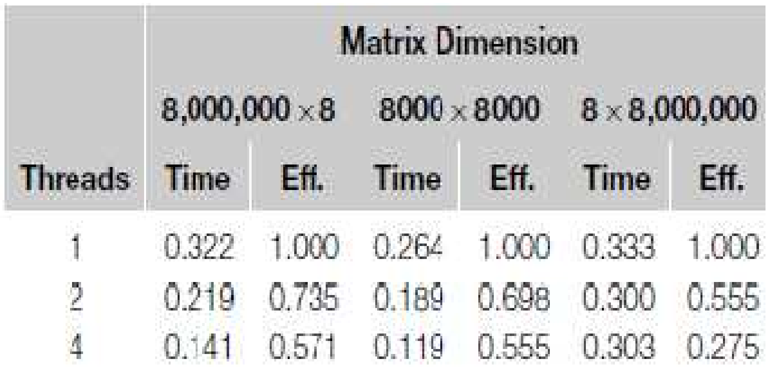
```
 # pragma omp parallel for numthreads(thread count) \
default(none) private(i, j) shared(A, x, y, m, n)
for (i = 0; i < m; i++) {
        y[i] = 0.0;
 for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

- If ***Tserial***is the run-time of the serial program and ***Tparallel*** is the run-time of the parallel program, recall that the efficiency E of the parallel program is the speedup Sdivided by the number of threads, t:

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

Where S ≤ t, E ≤ 1.

- In each case, the total number of floating point additions and multiplications is 64, 000, 000. An analysis that only considers arithmetic operations would predict thata single thread running the code would take the same amount of time for all threeinputs.
- However, it's clear that this is not the case. The 8, 000,000 x 8 system requires about 22% more time than the 8000 x 8000 system, and the 8 x 8, 000,000 systemrequires about 26% more time than the 8000 x 8000 system. Both of thesedifferencesare at least partially attributable to cache performance



	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

Fig :Run-Times and Efficiencies of Matrix-Vector Multiplication (times in seconds)

- A write-miss occurs when a core tries to update a variable that's notin cache, and it has to access main memory. A cache profiler (such as Valgrind [49]) shows that when the program is run with the 8, 000,000 x 8 input, it has far more cache write-misses than either of the other inputs. The 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, it's not surprising that this line slows down the execution of the program with the 8,000,000 x 8 input.

- Also recall that a read-miss occurs when a core tries to read a variable that's not in cache, and it has to access main memory. A cache profiler shows that when the program is run with the 8 x 8,000,000 input, it has far more cache read-misses than either of the other inputs. These occur in Line 6.

- The two-thread efficiency of the program with the 8x8,000,000 input is more than 20% less than the efficiency of the program with the 8,000,000x8 and the 8000x8000 inputs. The four-thread efficiency of the programwith the 8x8,000,000 input is more than 50% less than the program's efficiencywith the 8,000,000x8 and the 8000x8000 inputs.

- Cache coherence is enforced at the "cache-line level." That is, 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.
- The system we're using has two dual-core processors and each processor has its own cache. Suppose for the moment that threads 0 and 1 are assigned to one of the processors and threads 2 and 3 are assigned to the other. Also suppose that for the 8x8,000,000 problem 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. For example, each time thread 0 updates y[0] in the statement

```
y[i] += A[i][j] * x[j];
```

- If thread 2 or 3 is executing this code, it will have to reload ***y***. Each thread will update each of its components 8,000,000 times. We see that with this assignment of threads to processors and components of ***y*** to cache lines, all the threads will have to reload y many times. This is going to happen in spite of the fact that only one thread accesses any one component of ***y***—for example, only thread 0 accesses y[0].

- Each thread will update its assigned components of y a total of 16,000,000 times. It appears that many, if not most, of these updates are forcing the threads to access main memory. This is called ***false sharing.***

- Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable. Even though neither thread has written to a shared variable, the cache controller invalidates the entire cache line and forces the other threads to get the values of the variables from main memory. The threads aren't sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable. Hence the name ***false sharing***.

- For the 8000x8000 input, suppose thread 2 is assigned to one of the processors and thread 3 is assigned to another. Thread 2 is responsible for computing

  y[4000], y[4001], . . . , y[5999],

  and thread 3 is responsible for computing

  y[6000], y[6001], . . . , y[7999]

- If a cache line contains eight consecutive doubles, the only possibility for false sharing is on the interface between their assigned elements. If, for example, a single cache line contains

  y[5996], y[5997], y[5998], y[5999], y[6000], y[6001], y[6002], y[6003], then it’s conceivable that there might be false sharing of this cache line. However,thread 2 will access

y[5996], y[5997], y[5998], y[5999]

at the end of its iterations of the for i loop, while thread 3 will access

y[6000], y[6001], y[6002], y[6003]

at the beginning of its iterations. So it’s very likely that when thread 2 accesses, say,

y[5996], thread 3 will be long done with all four of

y[6000], y[6001], y[6002], y[6003].

Similarly, when thread 3 accesses, say, y[6003], it’s very likely that thread 2 won’tbe anywhere near starting to access

y[5996], y[5997], y[5998], y[5999].

- It’s therefore unlikely that false sharing of the elements of y will be a significant problem with the 8000_8000 input