

4 Out-of-order Execution [50 points]

In this problem, we will give you the state of the Register Alias Table (RAT) and Reservation Stations (RS) for a Tomasulo-like out-of-order execution engine. Your job is to determine the original sequence of **five instructions** in program order.

The out-of-order machine in this problem behaves as follows:

- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine dispatches one instruction per cycle into the reservation stations, in program order. Dispatch occurs during the decode stage.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.
- When a value is captured (at a reservation station) or written back (to a register) in this machine, the old tag that was previously at that location is *not cleared*; only the valid bit is set.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- Both the adder and multiplier are fully pipelined. Add instructions take 2 cycles. Multiply instructions take 4 cycles.
- When an instruction completes execution, it broadcasts its result, and dependent instructions can begin execution in the next cycle if they have all operands available.
- When multiple instructions are ready to execute at a functional unit, the *oldest* ready instruction is chosen.

Initially, the machine is empty. Five instructions then are fetched, decoded, and dispatched into reservation stations, before any instruction executes. Then, one instruction completes execution. Here is the state of the machine at this point, after the single instruction completes:

RAT

Reg	V	Tag	Value
R0	1		20
R1	1		50
R2	0	A	37
R3	1	X	500
R4	0	Y	255
R5	1		17
R6	0	Z	73
R7	1		10

	Src 1	Src 2				
	Tag	V	Value	Tag	V	Value
A	X	1	500	Y	0	-
B	-	1	20	-	1	17
C						



	Src 1	Src 2				
	Tag	V	Value	Tag	V	Value
X						
Y	-	1	50	-	1	37
Z	A	0	-	B	0	-



- (a) Give the five instructions that have been dispatched into the machine, in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: “opcode destination \leftarrow source1, source2.”

\leftarrow ,
 \leftarrow ,
 \leftarrow ,
 \leftarrow ,
 \leftarrow ,

- (b) Now assume that the machine flushes all instructions out of the pipeline and restarts execution from the first instruction in the sequence above. Show the full pipeline timing diagram below for the sequence of five instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fifth instruction.

As we saw in class, use “F” for fetch, “D” for decode, “E1,” “E2,” “E3,” and “E4” to signify the first, second, third and fourth cycles of execution for an instruction (as required by the type of instruction), and “W” to signify writeback. You may or may not need all columns shown.

	Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
MUL R3 \leftarrow R1, R7															
MUL R4 \leftarrow R1, R2															
ADD R2 \leftarrow R3, R4															
ADD R6 \leftarrow R0, R5															
MUL R6 \leftarrow R2, R6															

Finally, show the state of the RAT and reservation stations after **8 cycles** in the blank figures below.

RAT

Reg	V	Tag	Value
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
A						
B						
C						



	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
X						
Y						
Z						



4 Out-of-order Execution [50 points]

In this problem, we will give you the state of the Register Alias Table (RAT) and Reservation Stations (RS) for a Tomasulo-like out-of-order execution engine. Your job is to determine the original sequence of **five instructions** in program order.

The out-of-order machine in this problem behaves as follows:

- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine dispatches one instruction per cycle into the reservation stations, in program order. Dispatch occurs during the decode stage.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.
- When a value is captured (at a reservation station) or written back (to a register) in this machine, the old tag that was previously at that location is *not cleared*; only the valid bit is set.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- Both the adder and multiplier are fully pipelined. Add instructions take 2 cycles. Multiply instructions take 4 cycles.
- When an instruction completes execution, it broadcasts its result, and dependent instructions can begin execution in the next cycle if they have all operands available.
- When multiple instructions are ready to execute at a functional unit, the *oldest* ready instruction is chosen.

Initially, the machine is empty. Five instructions then are fetched, decoded, and dispatched into reservation stations, before any instruction executes. Then, one instruction completes execution. Here is the state of the machine at this point, after the single instruction completes:

RAT

Reg	V	Tag	Value
R0	1		20
R1	1		50
R2	0	A	37
R3	1	X	500
R4	0	Y	255
R5	1		17
R6	0	Z	73
R7	1		10

	Src 1	Src 2				
	Tag	V	Value	Tag	V	Value
A	X	1	500	Y	0	-
B	-	1	20	-	1	17
C						



	Src 1	Src 2				
	Tag	V	Value	Tag	V	Value
X						
Y	-	1	50	-	1	37
Z	A	0	-	B	0	-



- (a) Give the five instructions that have been dispatched into the machine, in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: “opcode destination \leftarrow source1, source2.”

MUL R3 \leftarrow R1, R7
 MUL R4 \leftarrow R1, R2
 ADD R2 \leftarrow R3, R4
 ADD R6 \leftarrow R0, R5
 MUL R6 \leftarrow R2, R6

- (b) Now assume that the machine flushes all instructions out of the pipeline and restarts execution from the first instruction in the sequence above. Show the full pipeline timing diagram below for the sequence of five instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fifth instruction.

As we saw in class, use “F” for fetch, “D” for decode, “E1,” “E2,” “E3,” and “E4” to signify the first, second, third and fourth cycles of execution for an instruction (as required by the type of instruction), and “W” to signify writeback. You may or may not need all columns shown.

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
MUL R3 \leftarrow R1, R7	F	D	E1	E2	E3	E4	W							
MUL R4 \leftarrow R1, R2		F	D	E1	E2	E3	E4	W						
ADD R2 \leftarrow R3, R4			F	D				E1	E2	W				
ADD R6 \leftarrow R0, R5				F	D	E1	E2	W						
MUL R6 \leftarrow R2, R6					F	D				E1	E2	E3	E4	W

Finally, show the state of the RAT and reservation stations after **8 cycles** in the blank figures below.

RAT

Reg	V	Tag	Value
R0	1		20
R1	1		50
R2	0	A	37
R3	1	X	500
R4	1	Y	1850
R5	1		17
R6	0	Z	73
R7	1		10

	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
A	X	1	500	Y	1	1850
B						
C						



	Src 1			Src 2		
	Tag	V	Value	Tag	V	Value
X						
Y						
Z	A	0	-	B	1	37



4 Caching [15 points]

Below, we have given you four different sequences of addresses generated by a program running on a processor with a data cache. Cache hit ratio for each sequence is also shown below. Assuming that the cache is initially empty at the beginning of each sequence, find out the following parameters of the processor's data cache:

- Associativity (1, 2 or 4 ways)
- Block size (1, 2, 4, 8, 16, or 32 bytes)
- Total cache size (256 B, or 512 B)
- Replacement policy (LRU or FIFO)

Assumptions: all memory accesses are one byte accesses. All addresses are byte addresses.

Sequence No.	Address Sequence	Hit Ratio
1	0, 2, 4, 8, 16, 32	0.33
2	0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

4 Caching [15 points]

Below, we have given you four different sequences of addresses generated by a program running on a processor with a data cache. Cache hit ratio for each sequence is also shown below. Assuming that the cache is initially empty at the beginning of each sequence, find out the following parameters of the processor's data cache:

- Associativity (1, 2 or 4 ways)
- Block size (1, 2, 4, 8, 16, or 32 bytes)
- Total cache size (256 B, or 512 B)
- Replacement policy (LRU or FIFO)

Assumptions: all memory accesses are one byte accesses. All addresses are byte addresses.

Sequence No.	Address Sequence	Hit Ratio
1	0, 2, 4, 8, 16, 32	0.33
2	0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

Solution:

Cache block size - 8 bytes

For sequence 1, only 2 out of the 6 accesses (specifically those to addresses 2 and 4) can hit in the cache, as the hit ratio is 0.33. With any other cache block size but 8 bytes, the hit ratio is either smaller or larger than 0.33.

Therefore, the cache block size is 8 bytes.

Associativity - 4 For sequence 2, blocks 0, 512, 1024 and 1536 are the only ones that are reused and could potentially result in cache hits when they are accessed the second time. Three of these four blocks should hit in the cache when accessed for the second time to give a hit rate of 0.33 (3/9).

Given that the block size is 8 and for either cache size (256B or 512B), all of these blocks map to set 0. Hence, an associativity of 1 or 2 would cause at most one or two of these four blocks to be present in the cache when they are accessed for the second time, resulting in a maximum possible hit rate of less than 3/9. However, the hit rate for this sequence is 3/9. Therefore, an associativity of 4 is the only one that could potentially give a hit rate of 0.33 (3/9).

Total cache size - 256 B

For sequence 3, a total cache size of 512 B will give a hit rate of 4/9 with a 4-way associative cache and 8 byte blocks regardless of the replacement policy, which is higher than 0.33. Therefore, the total cache size is 256 bytes.

Replacement policy - LRU

For the aforementioned cache parameters, all cache lines in sequence 4 map to set 0. If a FIFO replacement policy were used, the hit ratio would be 3/8, whereas if an LRU replacement policy were used, the hit ratio would be 1/4. Therefore, the replacement policy is LRU.

4 Angry Students

Your friend is developing a game that features a horde of angry students chasing after professors for making long exams. Simulating students is expensive, so your friend decides to parallelize the computation using one thread to compute and update the student's positions, and another thread to simulate the student's angriness. The state of the game's N students is stored in the global array `students` in the code below).

```
struct Student {
    float position; // assume 1D position for simplicity
    float angriness;
};

Student students[N];

////////////////////////////////////

void update_positions() {
    for (int i=0; i<N; i++) {
        students[i].position = compute_new_position(i);
    }
}

void update_angriness() {
    for (int i=0; i<N; i++) {
        students[i].angriness = compute_new_angriness(i);
    }
}

////////////////////////////////////

// ... initialize students here

pthread_t t0, t1;
pthread_create(&t0, NULL, updatePositions, NULL);
pthread_create(&t1, NULL, updateAngriness, NULL);
pthread_join(t0, NULL);
pthread_join(t1, NULL);
```

- A. Since there is no synchronization between thread 0 and thread 1, your friend expects near a perfect $2\times$ speedup when running on two-core processor that implements invalidation-based cache coherence. She is shocked when she doesn't obtain it. Why is this the case? (For this problem assume that there is sufficient bandwidth to keep two cores busy – “the code is bandwidth bound” is not an answer we are looking for.)

Solution: This is a classic false-sharing situation. Assuming the threads iterate through the array at equal rates (that is, they are on the same loop iteration at about the same time), both threads will be writing to elements on the same cache line at about the same time. The cache line will bounce back and forth between the caches of the two processors. In the worst case, every write is a miss.

- B. Modify the program to correct the performance problem. You are allowed to modify the code and data structures as you wish, **but you are not allowed to change what computations are performed by each thread and your solution should not substantially increase the amount of memory used by the program.** You only need to describe your solution in pseudocode (compilable code is not required).

A simple solution is to change the data structure from an array of `Student` structures to two arrays, one for each field. As a result, each thread works on its own array and scans over it contiguously.

```
float position[N];  
float angriness[N];
```

Some students mentioned that an alternative solution was to offset the position of the threads in the arrays to ensure that, at any one moment, each thread operating in distant parts of the array. One example was to have one thread iterate from $i=0$ to N , and the other iterate backwards from $N-1$ to 0 . This solution eliminates the false sharing and was given full credit. It should be noted that the spatial locality of data access is not as good (by a factor of 2) in this scenario than for the solution described above since each thread only makes use of 1/2 of the data in each cache line it loads.

6 Parallel Histogram Generation

Your friend implements the following parallel code for generating a histogram from the values in a large input array `input`. For each element of the input array, the code uses the function `bin_func` to compute a “bin” the element belongs to (`bin_func` always returns an integer between 0 and `NUM_BINS-1`), and increments a count of elements in that bin. His port targets a small parallel machine with only two processors. *This machine features 64-byte cache lines and uses an invalidation-based cache coherence protocol.* Your friend’s implementation is given below.

```
float input[N];           // assume input is initialized and N is a very large
int  histogram_bins[NUM_BINS]; // output bins
int  partial_bins[2][NUM_BINS]; // assume bins are initialized to 0
                                   // assume partial_bins is 64-byte aligned

//////////////////////////////////// Code executed by thread 0 //////////////////////////////////////
for (int i=0; i<N/2; i++)
    partial_bins[0][bin_func(input[i])]++;

barrier(); // wait for both threads to reach this point

for (int i=0; i<NUM_BINS; i++)
    histogram_bins[i] = partial_bins[0][i] + partial_bins[1][i];

//////////////////////////////////// Code executed by thread 1 //////////////////////////////////////
for (int i=N/2; i<N; i++)
    partial_bins[1][bin_func(input[i])]++;

barrier(); // wait for both threads to reach this point
```

- A. Your friend runs this code on an input of 1 million elements ($N=1,000,000$) to create a histogram with eight bins ($NUM_BINS=8$). He is shocked when his program obtains far less than a linear speedup, and glumly asserts believe he needs to completely restructure the code to eliminate load imbalance. You take a look and recommend that he not do any coding at all, and just create a histogram with 16 bins instead. Explain why.

*Solution: With a 64-bit-wide cache line and 8 bins, the `partial_bins` arrays for each thread lie on the same cache line (8 integers = 32 bytes). As a result, although there is no data sharing between the two threads when computing the partial results, significant **false sharing** will occur. Increasing the number of bins to 16 causes, `partial_bins[0]` and `partial_bins[1]` to reside on a separate cache lines, and false sharing is eliminated. It could also be noted that there is very little load imbalance in the current solution. The threads each process 500,000 elements in parallel. Then the serial part of the code is a simple summation of eight numbers.*

- B. Inspired by his new-found great performance, your friend concludes that more bins is better. He tries to use the provided code from part A to compute a histogram of 10,000 elements with 2,000 bins. He is shocked when the speedup obtained by the code drops. Improve the existing code to scale near linearly with the larger number of bins. (Please provide pseudocode as part of your answer – it need not be compilable C code.)

Now, with a large number of bins (and fewer total elements), the serial combination of the partial results is a significant fraction (20%) of execution time, significantly limiting speedup! Correct solutions sought to parallelize the combination step. Example code is given below:

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;
```

where `my_z` is another private variable.

What's the value in `my_z`? Is it 5? Or is it 6? The problem is that 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. When thread 0 executed `x++`, what happened to the values in main memory and thread 1's cache? This is the **cache coherence** problem, which we discussed in Chapter 2. We saw there that most systems insist that the caches be made aware that changes have been made to data they are caching. The line in the cache of thread 1 would have been marked *invalid* when thread 0 executed `x++`, and before assigning `my_z = x`, the core running thread 1 would see that its value of `x` was out of date. Thus, the core running thread 0 would have to update the copy of `x` in main memory (either now or earlier), and the core running thread 1 would get the line with the updated value of `x` from main memory. For further details, see Chapter 2.

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. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and \mathbf{x} is a vector with n components, then their product $\mathbf{y} = \mathbf{A}\mathbf{x}$ is a vector with m components, and its i th component y_i is found by forming the dot product of the i th row of A with \mathbf{x} :

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

See Figure 5.5.

So if we store A as a two-dimensional array and \mathbf{x} and \mathbf{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;
```

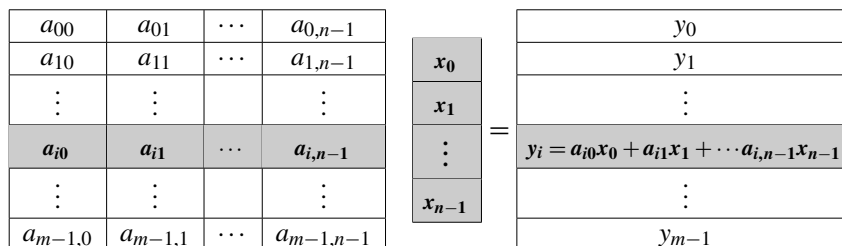


FIGURE 5.5

Matrix-vector multiplication

```

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

```

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:

```

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   }

```

If T_{serial} is the run-time of the serial program and T_{parallel} is the run-time of the parallel program, recall that the *efficiency* E of the parallel program is the speedup S divided by the number of threads, t :

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

Since $S \leq t$, $E \leq 1$. Table 5.4 shows the run-times and efficiencies of our matrix-vector multiplication with different sets of data and differing numbers of threads.

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 that a single thread running the code would take the same amount of time for all three inputs. However, it's clear that this is *not* the case. The $8,000,000 \times 8$ system requires about 22% more time than the 8000×8000 system, and the $8 \times 8,000,000$ system requires about 26% more time than the 8000×8000 system. Both of these differences are at least partially attributable to cache performance.

Recall that a **write-miss** occurs when a core tries to update a variable that's not in cache, and it has to access main memory. A cache profiler (such as Valgrind [49]) shows that when the program is run with the $8,000,000 \times 8$ input, it has far more

Table 5.4 Run-Times and Efficiencies of Matrix-Vector Multiplication (times in seconds)

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

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 $8,000,000 \times 8$ 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 $8 \times 8,000,000$ input, it has far more cache read-misses than either of the other inputs. These occur in Line 6, and a careful study of this program (see Exercise 5.12) shows that the main source of the differences is due to the reads of x . Once again, this isn't surprising, since for this input, x has 8,000,000 elements, versus only 8000 or 8 for the other inputs.

It should be noted that there may be other factors that affect the relative performance of the single-threaded program with differing inputs. For example, we haven't taken into consideration whether virtual memory (see Section 2.2.4) has affected the performance of the program with the different inputs. How frequently does the CPU need to access the page table in main memory?

Of more interest to us, though, are the differences in efficiency as the number of threads is increased. The two-thread efficiency of the program with the $8 \times 8,000,000$ input is more than 20% less than the efficiency of the program with the $8,000,000 \times 8$ and the 8000×8000 inputs. The four-thread efficiency of the program with the $8 \times 8,000,000$ input is more than 50% less than the program's efficiency with the $8,000,000 \times 8$ and the 8000×8000 inputs. Why, then, is the multithreaded performance of the program so much worse with the $8 \times 8,000,000$ input?

In this case, once again, the answer has to do with cache. Let's take a look at the program when we run it with four threads. With the $8,000,000 \times 8$ input, y has 8,000,000 components, so each thread is assigned 2,000,000 components. With the 8000×8000 input, each thread is assigned 2000 components of y , and with the $8 \times 8,000,000$ input, each thread is assigned two components. On the system we used, a cache line is 64 bytes. Since the type of y is `double`, and a `double` is 8 bytes, a single cache line will store eight `doubles`.

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 $8 \times 8,000,000$ 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*.

Why is false sharing not a problem with the other inputs? Let's look at what happens with the 8000×8000 input. Suppose thread 2 is assigned to one of the processors and thread 3 is assigned to another. (We don't actually know which threads are assigned to which processors, but it turns out—see Exercise 5.13—that it doesn't matter.) 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't be 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. Similar reasoning suggests that false sharing of y is unlikely to be a problem with the $8,000,000 \times 8$ input. Also note that we don't need to worry about false sharing of A or x , since their values are never updated by the matrix-vector multiplication code.

This brings up the question of how we might avoid false sharing in our matrix-vector multiplication program. One possible solution is to “pad” the y vector with dummy elements in order to insure that any update by one thread won't affect another thread's cache line. Another alternative is to have each thread use its own private storage during the multiplication loop, and then update the shared storage when they're done (see Exercise 5.15).

5.10 THREAD-SAFETY⁶

Let's look at another potential problem that occurs in shared-memory programming: *thread-safety*. A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

As an example, suppose we want to use multiple threads to “tokenize” a file. Let's suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—spaces, tabs, or newlines. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, ..., the t th goes to thread t , the $t + 1$ st goes to thread 0, and so on.

We'll read the text into an array of strings, with one line of text per string. Then we can use a `parallel for` directive with a `schedule(static,1)` clause to divide the lines among the threads.

One way to tokenize a line is to use the `strtok` function in `string.h`. It has the following prototype:

```
char* strtok(
    char*    string    /* in/out */,
    const char* separators /* in */);
```

Its usage is a little unusual: the first time it's called, the `string` argument should be the text to be tokenized, so in our example it should be the line of input. For subsequent calls, the first argument should be `NULL`. The idea is that in the first call, `strtok` caches a pointer to `string`, and for subsequent calls it returns successive tokens taken from the cached copy. The characters that delimit tokens should be passed in `separators`, so we should pass in the string `" \t\n"` as the `separators` argument.

⁶This material is also covered in Chapter 4, so if you've already read that chapter, you may want to just skim this section.

5 Virtual Memory [10 points]

An ISA supports an 8-bit, byte-addressable virtual address space. The corresponding physical memory has only 128 bytes. Each page contains 16 bytes. A simple, one-level translation scheme is used and the page table resides in physical memory. The initial contents of the frames of physical memory are shown below.

Frame Number	Frame Contents
0	Empty
1	Page 13
2	Page 5
3	Page 2
4	Empty
5	Page 0
6	Empty
7	Page Table

A three-entry translation lookaside buffer that uses Least Recently-Used (LRU) replacement is added to this system. Initially, this TLB contains the entries for pages 0, 2, and 13. For the following sequence of references, put a circle around those that generate a TLB hit and put a rectangle around those that generate a page fault. What is the hit rate of the TLB for this sequence of references? (Note: LRU policy is used to select pages for replacement in physical memory.)

References (to pages): 0, 13, 5, 2, 14, 14, 13, 6, 6, 13, 15, 14, 15, 13, 4, 3.

- At the end of this sequence, what three entries are contained in the TLB?
- What are the contents of the 8 physical frames?