ECE 432/532
Programming for Parallel Processors

Caches, Cache-Coherence, and False Sharing

- Recall that chip designers have added blocks of relatively fast memory to processors called cache memory.

- The use of cache memory can have a huge impact on shared-memory.

- A write-miss occurs when a core tries to update a variable that's not in cache, and it has to access main memory.

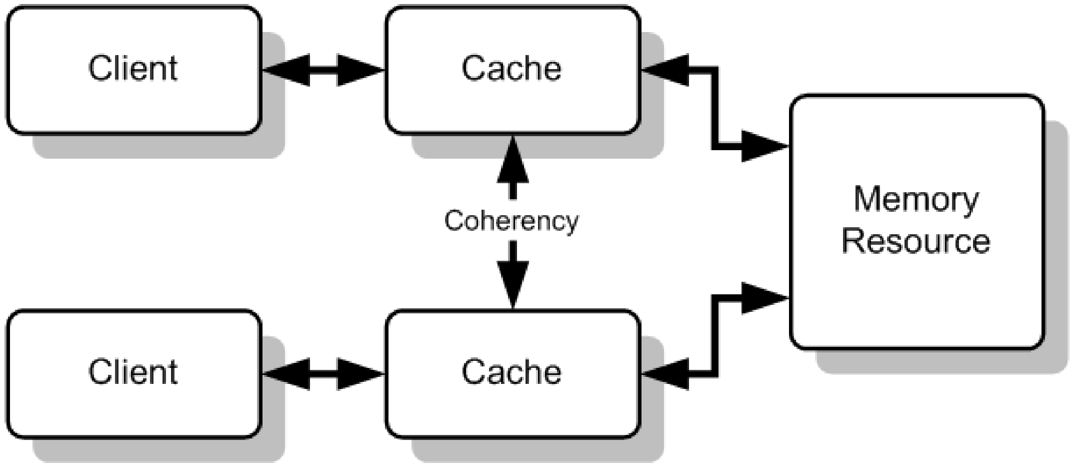
Caches, Cache-Coherence, and False Sharing

- The design of cache memory takes into consideration the principles of **temporal and spatial locality**
  - if a processor accesses main memory location *x* at time *t*, then it is likely that at times close to *t* it will access main memory locations close to *x*.

- Thus, when a processor accesses main memory location *x*, it transfers a block of memory
  - **cache line** or **cache block**.

Caches, Cache-Coherence, and False Sharing

- In computer science, **cache coherence** is the consistency of shared resource data that ends up stored in multiple local caches.



Caches, Cache-Coherence, and False Sharing

- In a **shared memory** system with a separate cache memory for each processor, it is possible to have many copies of data

- Cache coherence is the discipline that ensures that changes in the values of shared data are propagated throughout the system in a timely fashion

Pthreads matrix-vector multiplication

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

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

   return NULL;
}  /* Pth_mat_vect */
```

Run-times and efficiencies of matrix-vector multiplication

Threads	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

(times are in seconds)

In each case, the total number of floating point additions and multiplications is 64, 000, 000

Run-times and efficiencies of matrix-vector multiplication

- A *write-miss* occurs when a core tries to update a variable that's not in the cache, and it has to access main memory

- A *read-miss* occurs when a core tries to read a variable that's not in the cache, and it has to access main memory

- With the 8, 000,0008 input, y has 8,000,000 components, so each thread is assigned 2,000,000 components.
- With the 8000x8000 input, each thread is assigned 2000 components of y
- With the 8 x 8, 000,000 input, each thread is assigned 2 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 can store 8 **double**s.

Run-times and efficiencies of matrix-vector multiplication

- Cache coherence is enforced at the “cache-line level.”

- Each time any value in a cache line is written, if the line is also stored in another processor’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.
  - threads 0 and 1 are assigned to one of the processors and threads
  - 2 and 3 are assigned to the other
  - all of y is stored in a single cache line (8 x 8, 000,000)

Run-times and efficiencies of matrix-vector multiplication

- What will happen here?

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

- Every write to some element of y will invalidate the line in the other processor's cache

- Each time thread 0 updates y[0], thread 2 or 3, it will have to reload y.
  - Each thread will update each of its components 8, 000,000 times → False sharing

- What happens in other cases?

Example

- Suppose we want to use multiple threads to "tokenize" a file that consists of ordinary English text.
- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.

Simple approach

- 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 tth goes to thread t, the t +1st goes to thread 0, etc.

Simple approach

- We can serialize access to the lines of input using semaphores.
- After a thread has read a single line of input, it can tokenize the line using the strtok function.

The strtok function

- The first time it's called the string argument should be the text to be tokenized.
  - Our line of input.
- For subsequent calls, the first argument should be NULL.

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

The strtok function

- 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.

Multi-threaded tokenizer (1)

```
void *Tokenize(void* rank) {
   long my_rank = (long) rank;
   int count;
   int next = (my_rank + 1) % thread_count;
   char *fg_rv;
   char my_line[MAX];
   char *my_string;

   sem_wait(&sems[my_rank]);
   fg_rv = fgets(my_line, MAX, stdin);
   sem_post(&sems[next]);
   while (fg_rv != NULL) {
      printf("Thread %ld > my line = %s", my_rank, my_line);
```

Multi-threaded tokenizer (2)

```
      count = 0;
      my_string = strtok(my_line, " \t\n");
      while ( my_string != NULL ) {
         count++;
         printf("Thread %ld > string %d = %s\n", my_rank, count,
               my_string);
         my_string = strtok(NULL, " \t\n");
      }

      sem_wait(&sems[my_rank]);
      fg_rv = fgets(my_line, MAX, stdin);
      sem_post(&sems[next]);
   }

   return NULL;
}  /* Tokenize */
```

Running with one thread

- It correctly tokenizes the input stream.

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

Running with two threads

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

Oops!

What happened?

- strtok caches the input line by declaring a variable to have static storage class.
- This causes the value stored in this variable to persist from one call to the next.

- Unfortunately for us, this cached string is shared, not private.

What happened?

- Thus, thread 0's call to strtok with the third line of the input has apparently <u>overwritten</u> the contents of thread 1's call with the second line.

- So the strtok function is not thread-safe. If multiple threads call it simultaneously, the output may not be correct.

Other unsafe C library functions

- Regrettably, it's not uncommon for C library functions to fail to be thread-safe.
- The random number generator random in stdlib.h.
- The time conversion function localtime in time.h.