OpenMP

Jan 7, 2020

Course outline (Pacheco; GGKK; Quinn)

- Motivation (1;1;1)
- How to quantify performance improvement (2.6; 5; 7)
- Parallel hardware architecture (2.2-2.3; 2,4; 2)
- Parallel programming frameworks
  - Pthreads for shared memory (4; 7; -)
  - **OpenMP for shared memory (5; 7.10; 17)**
  - MPI for distributed memory (3; 6; 4)
  - CUDA/OpenCL for GPU,
  - Hadoop/Spark/Mapreduce for distributed systems
- Parallel program verification
- Parallel algorithm design
- Some case studies

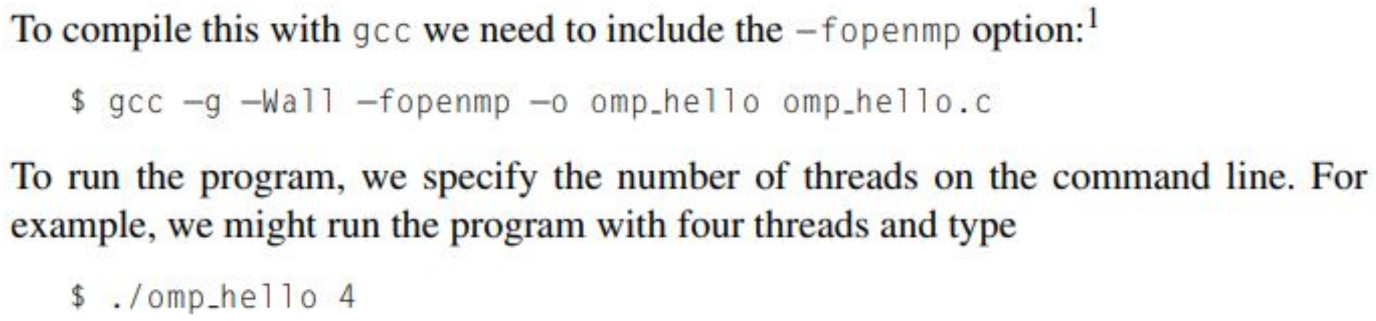
Discussion points

- Hello world program, compile, run
- Synchronization
  - For mutual exclusion on shared data - critical, atomic, lock
  - For work coordination - barrier
  - Prevent data dependencies
- Scope of variables
- Sharing work among threads
- Thread safety
- Task Parallelism
- Cache coherence, false sharing (during architecture discussion)
- Non parallelizable algorithms (during algorithm design discussion)

Hello World

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Compiling and running

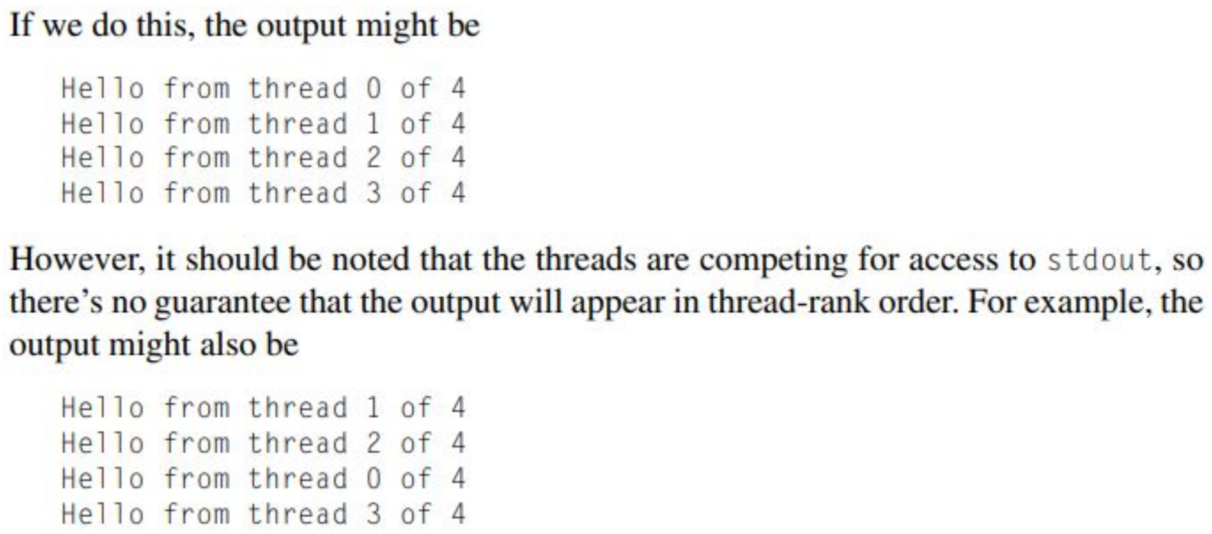


To compile this with `gcc` we need to include the `−fopenmp` option:[1]

```
$ gcc −g −Wall −fopenmp −o omp_hello omp_hello.c
```

To run the program, we specify the number of threads on the command line. For example, we might run the program with four threads and type

```
$ ./omp_hello 4
```



If we do this, the output might be

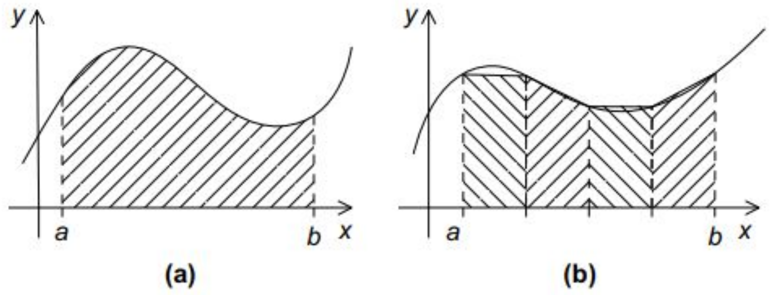
```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

However, it should be noted that the threads are competing for access to `stdout`, so there’s no guarantee that the output will appear in thread-rank order. For example, the output might also be

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

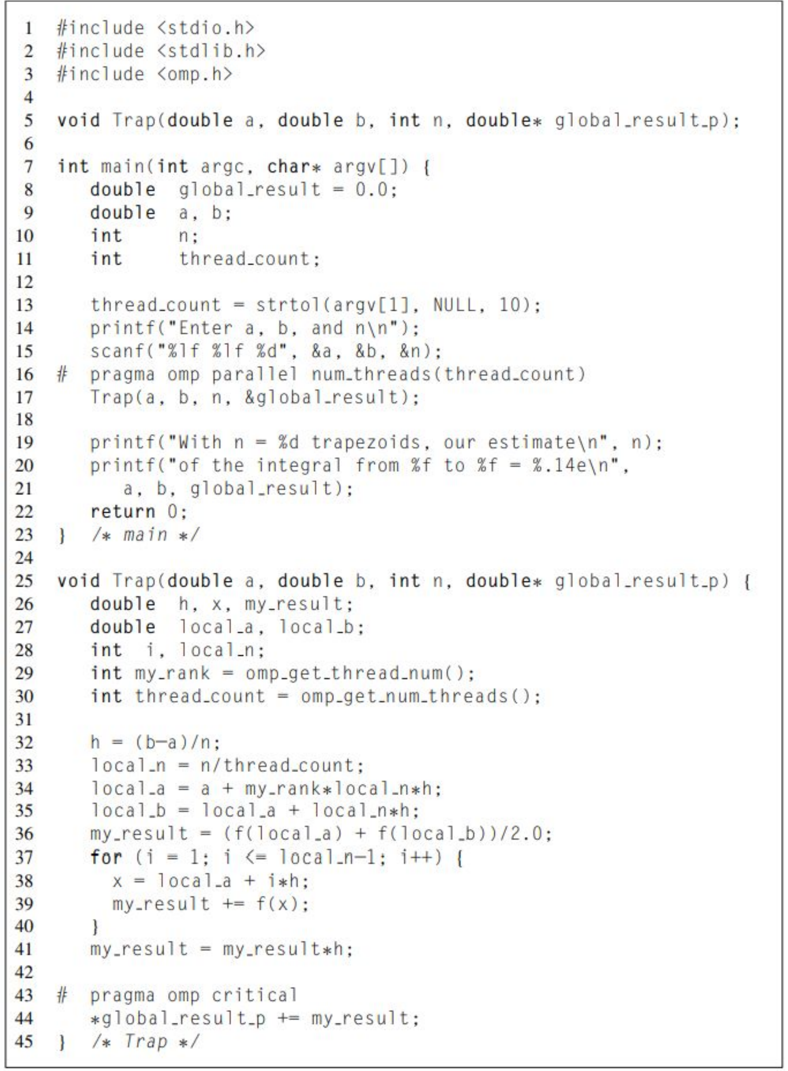
Race condition example: Trapezoidal rule example

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```



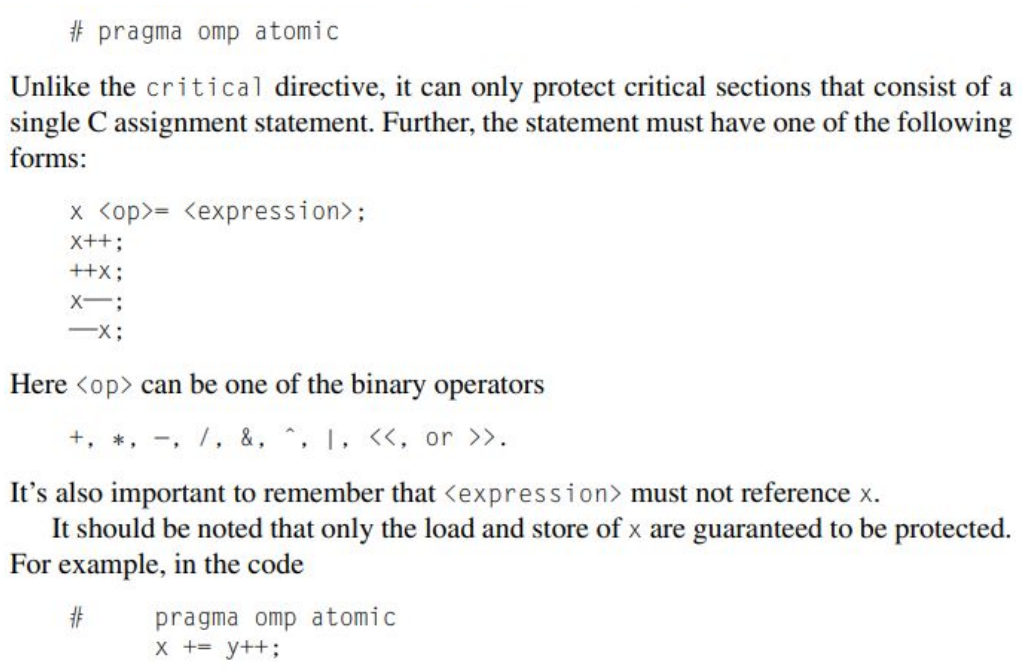
Time	Thread 0	Thread 1
0	global_result = 0 to register	finish my_result
1	my_result = 1 to register	global_result = 0 to register
2	add my_result to global_result	my_result = 2 to register
3	store global_result = 1	add my_result to global_result
4		store global_result = 2

Critical directive



```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;
   double  a, b;
   int     n;
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */

void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

Atomic directive



```
# pragma omp atomic
```

Unlike the `critical` directive, it can only protect critical sections that consist of a single C assignment statement. Further, the statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x—;
—x;
```

Here `<op>` can be one of the binary operators

```
+, *, −, /, &, ^, |, <<, or >>.
```

It's also important to remember that `<expression>` must not reference `x`.

It should be noted that only the load and store of `x` are guaranteed to be protected. For example, in the code

```
#     pragma omp atomic
      x += y++;
```

The idea behind the atomic directive is that many processors provide a special **load-modify-store instruction**. A critical section that only does a load-modify-store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

Fine grained synchronization: message queue example

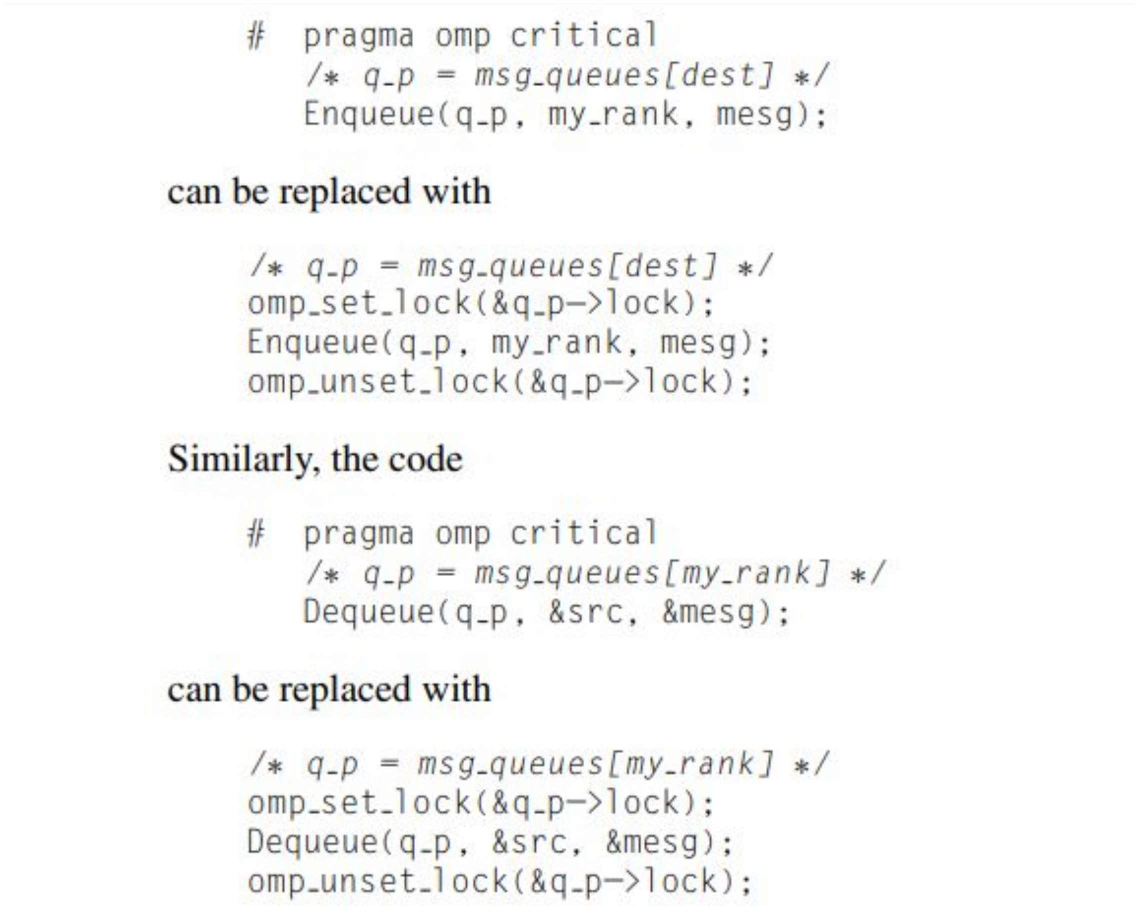
```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
   Send_msg();
   Try_receive();
}

while (!Done())
   Try_receive();
```

```
   mesg = random();
   dest = random() % thread_count;
#  pragma omp critical
   Enqueue(queue, dest, my_rank, mesg);
```

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```

```
queue_size = enqueued - dequeued;
if (queue_size == 0) return;

else if (queue_size == 1)
   pragma omp critical
   Dequeue(queue, &src, &mesg);
else
   Dequeue(queue, &src, &mesg);
Print_message(src, mesg);
```

Lock primitive



```
#  pragma omp critical
   /* q_p = msg_queues[dest] */
   Enqueue(q_p, my_rank, mesg);
```

can be replaced with

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

Similarly, the code

```
#  pragma omp critical
   /* q_p = msg_queues[my_rank] */
   Dequeue(q_p, &src, &mesg);
```

can be replaced with

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

Synchronization caveats

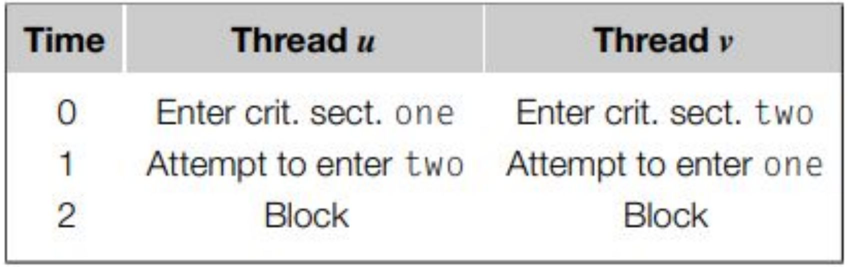
- Mixing different synchronization primitives

```
#  pragma omp atomic        #  pragma omp critical
   x += f(y);                  x = g(x);
```

- Issue of fairness

```
   while(1) {
      . . .
#     pragma omp critical
      x = g(my_rank);
      . . .
   }
```

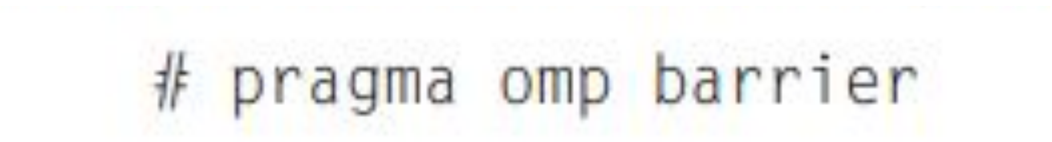
- Issue of deadlock, especially if threads enter different critical sections in different orders



Time	Thread *u*	Thread *v*
0	Enter crit. sect. one	Enter crit. sect. two
1	Attempt to enter two	Attempt to enter one
2	Block	Block

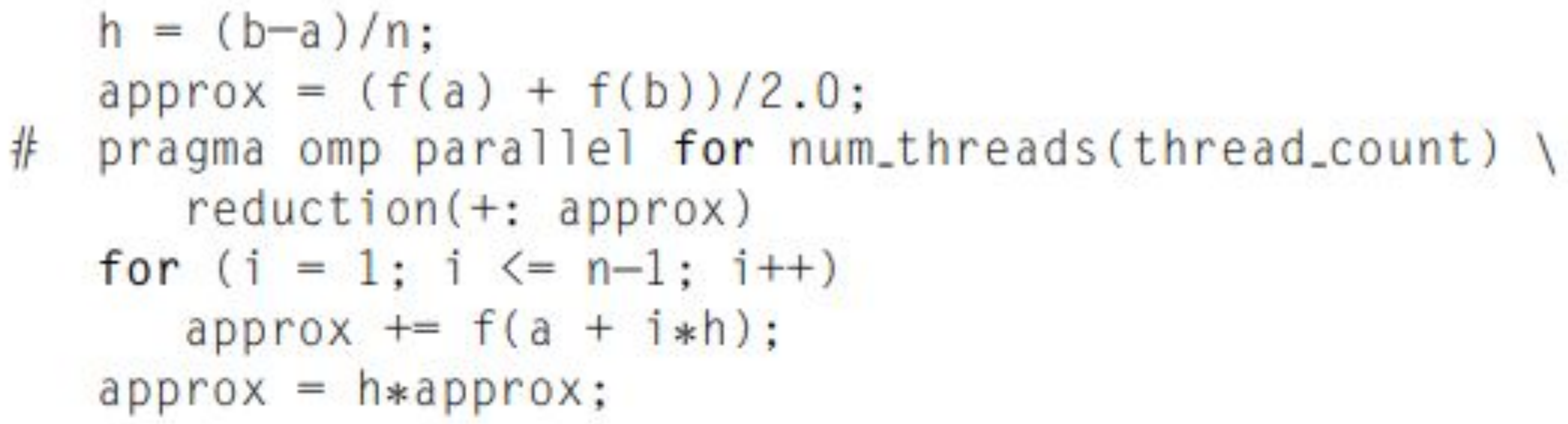
Work synchronization: Barrier primitive

- One or more threads might finish allocating their queues before some other threads
- If this happens, the threads that finish first could start trying to enqueue messages in a queue that hasn’t been allocated
- Program will crash
- In middle of parallel block, so implicit barriers will not work
- Use explicit barrier to make sure none of the threads start sending messages until all the queues are allocated.



```
# pragma omp barrier
```

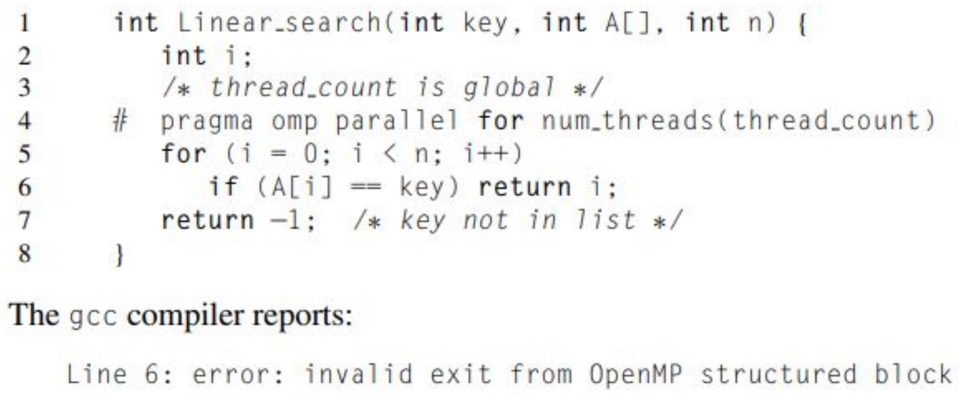
Parallel for



```
    h = (b-a)/n;
    approx = (f(a) + f(b))/2.0;
#   pragma omp parallel for num_threads(thread_count) \
       reduction(+: approx)
    for (i = 1; i <= n-1; i++)
       approx += f(a + i*h);
    approx = h*approx;
```

For loop restrictions

- Only loops for which the number of iterations can be determined . from the for statement itself and prior to execution of the loop.
- The variable index must have integer or pointer type (e.g., it can’t be a float).
- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, then incr must have integer type.
- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the “increment expression” in the for statement.



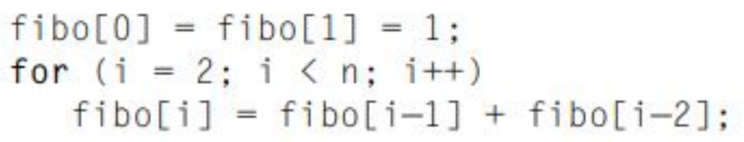
```
1     int Linear_search(int key, int A[], int n) {
2        int i;
3        /* thread_count is global */
4     #  pragma omp parallel for num_threads(thread_count)
5        for (i = 0; i < n; i++)
6           if (A[i] == key) return i;
7        return -1;  /* key not in list */
8     }
```

The gcc compiler reports:

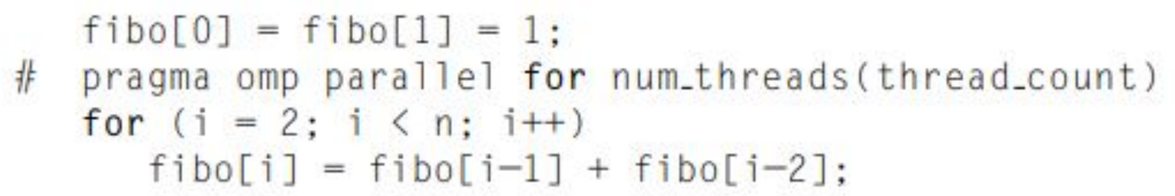
```
Line 6: error: invalid exit from OpenMP structured block
```

Loop carried dependencies

- OpenMP compilers don’t check for dependences among iterations in a loop that’s being parallelized with a parallel for directive. It’s up to us, the programmers, to identify these dependencies.
- A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.
- Example: 1 1 2 3 5 8 13 21 34 55 or 1 1 2 3 5 8 0 0 0 0 can both be output from parallelizing the Fibonacci for loop

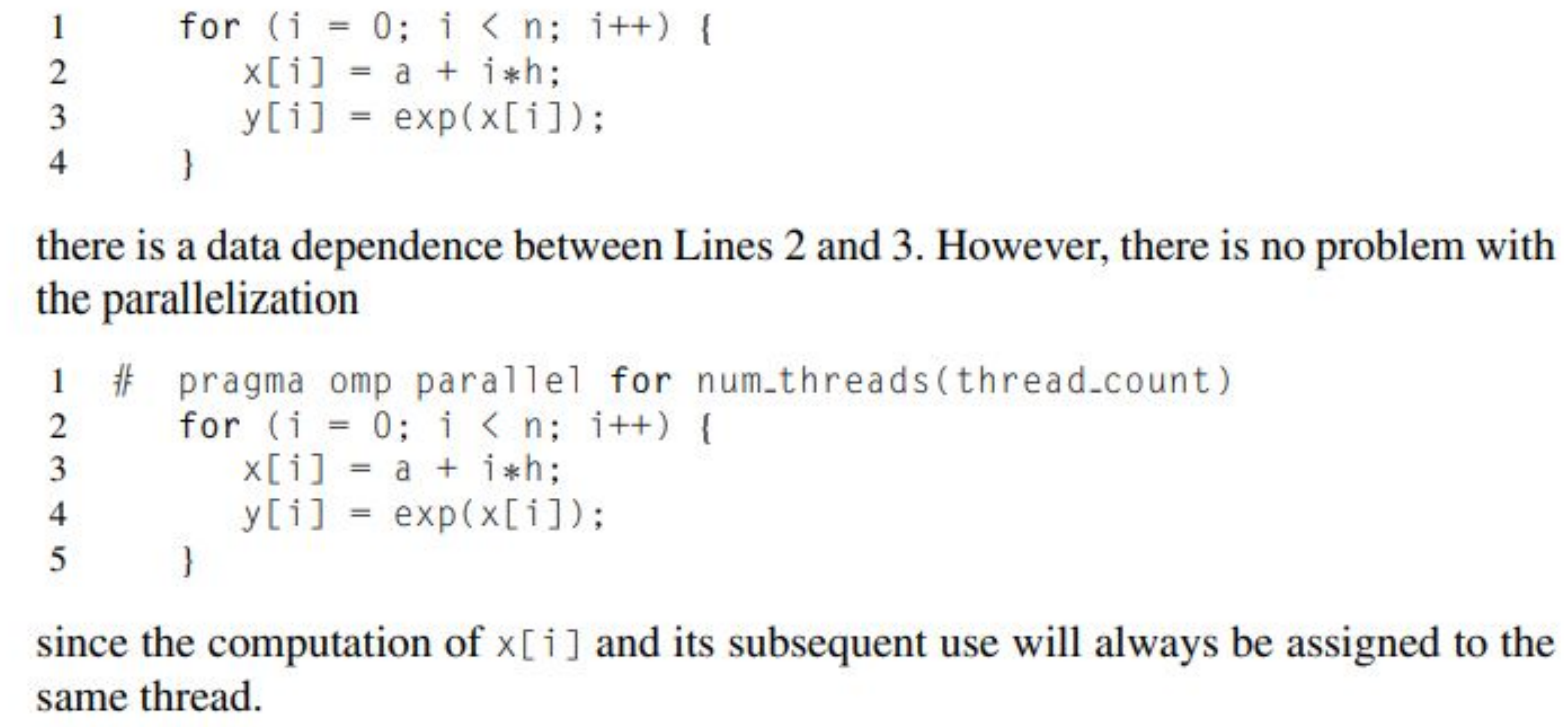


```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
   fibo[i] = fibo[i-1] + fibo[i-2];
```



```
   fibo[0] = fibo[1] = 1;
#  pragma omp parallel for num_threads(thread_count)
   for (i = 2; i < n; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];
```

General data dependencies are fine



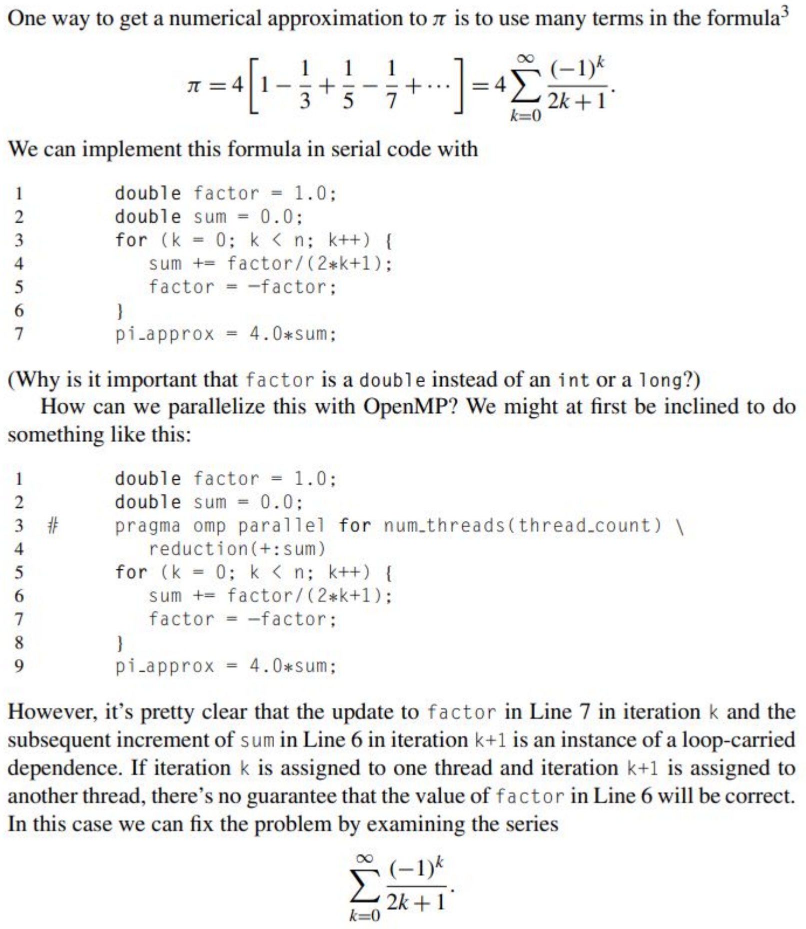
```
1     for (i = 0; i < n; i++) {
2        x[i] = a + i*h;
3        y[i] = exp(x[i]);
4     }
```

there is a data dependence between Lines 2 and 3. However, there is no problem with the parallelization

```
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < n; i++) {
      x[i] = a + i*h;
      y[i] = exp(x[i]);
   }
```

since the computation of `x[i]` and its subsequent use will always be assigned to the same thread.

Dealing with loop carried dependencies



One way to get a numerical approximation to π is to use many terms in the formula[3]

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

We can implement this formula in serial code with

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
   sum += factor/(2*k+1);
   factor = -factor;
}
pi_approx = 4.0*sum;
```

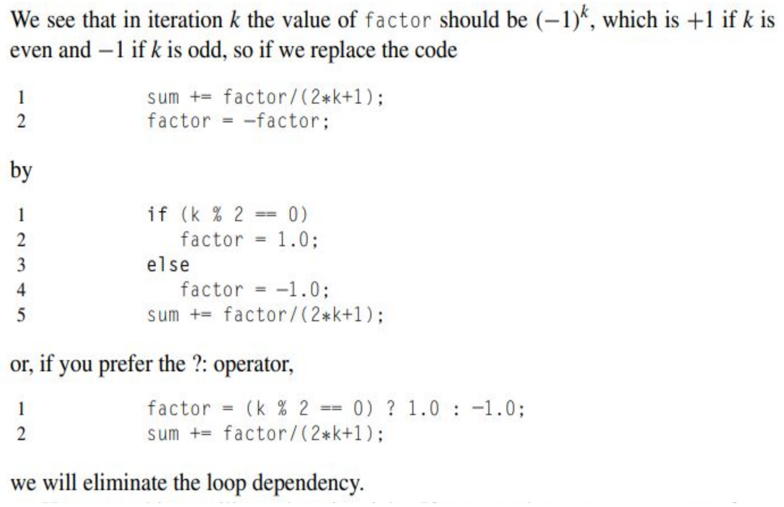
(Why is it important that `factor` is a `double` instead of an `int` or a `long`?)

How can we parallelize this with OpenMP? We might at first be inclined to do something like this:

```
1        double factor = 1.0;
2        double sum = 0.0;
3  #     pragma omp parallel for num_threads(thread_count) \
4           reduction(+:sum)
5        for (k = 0; k < n; k++) {
6           sum += factor/(2*k+1);
7           factor = -factor;
8        }
9        pi_approx = 4.0*sum;
```

However, it's pretty clear that the update to `factor` in Line 7 in iteration `k` and the subsequent increment of `sum` in Line 6 in iteration `k+1` is an instance of a loop-carried dependence. If iteration `k` is assigned to one thread and iteration `k+1` is assigned to another thread, there's no guarantee that the value of `factor` in Line 6 will be correct. In this case we can fix the problem by examining the series

$$\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$



We see that in iteration k the value of `factor` should be $(-1)^k$, which is $+1$ if k is even and -1 if k is odd, so if we replace the code

```
sum += factor/(2*k+1);
factor = -factor;
```

by

```
if (k % 2 == 0)
   factor = 1.0;
else
   factor = -1.0;
sum += factor/(2*k+1);
```

or, if you prefer the ?: operator,

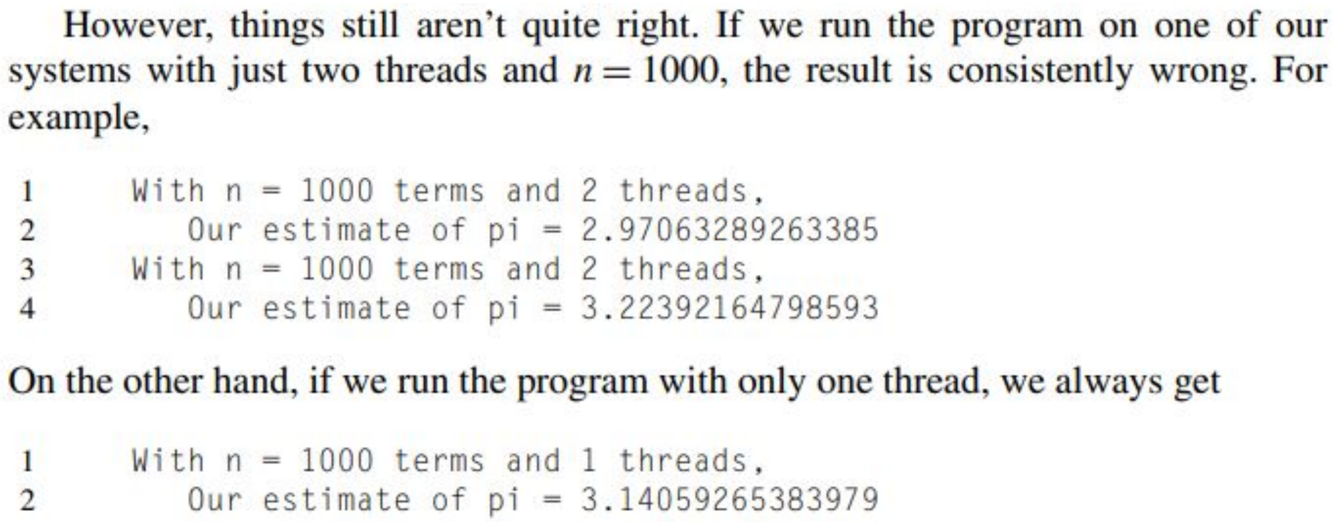
```
factor = (k % 2 == 0) ? 1.0 : -1.0;
sum += factor/(2*k+1);
```

we will eliminate the loop dependency.

Discussion points

- **Hello world program, compile, run**
- **Synchronization**
  - **For mutual exclusion on shared data - critical, atomic, lock**
  - **For work coordination - barrier**
  - **Prevent data dependencies**
- Scope of variables
- Sharing work among threads
- Thread safety
- Task Parallelism
- Cache coherence, false sharing (during architecture discussion)
- Non parallelizable algorithms (during algorithm design discussion)

Scope of variables



However, things still aren't quite right. If we run the program on one of our systems with just two threads and $n = 1000$, the result is consistently wrong. For example,

```
With n = 1000 terms and 2 threads,
   Our estimate of pi = 2.97063289263385
With n = 1000 terms and 2 threads,
   Our estimate of pi = 3.22392164798593
```

On the other hand, if we run the program with only one thread, we always get

```
With n = 1000 terms and 1 threads,
   Our estimate of pi = 3.14059265383979
```

Use default(none) and private for correctness

```
      double sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         default(none) reduction(+:sum) private(k, factor) \
         shared(n)
      for (k = 0; k < n; k++) {
         if (k % 2 == 0)
            factor = 1.0;
         else
            factor = -1.0;
         sum += factor/(2*k+1);
      }
```

Reduction clause

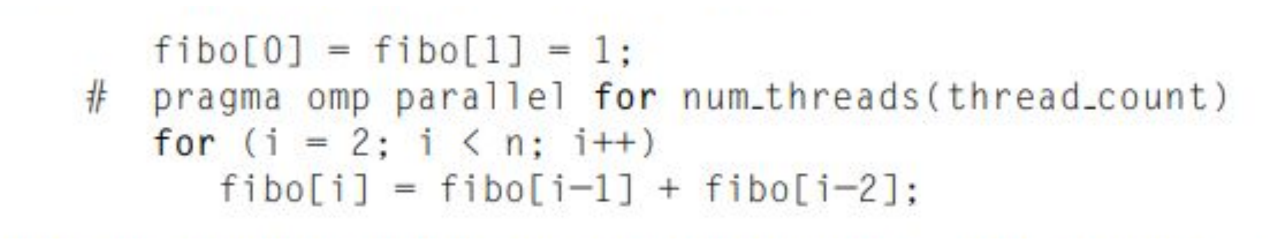
```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
#     pragma omp critical
      global_result += Local_trap(double a, double b, int n);
   }
```

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

How is work divided among threads?

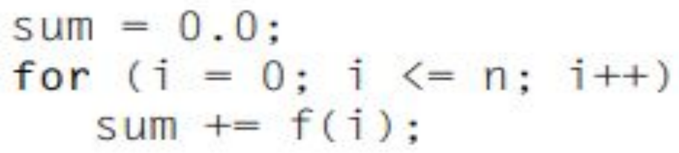
Most OpenMP implementations use roughly a block partitioning: if there are n iterations in the serial loop, then in the parallel loop the first n/thread count are assigned to thread 0, the next n/thread count are assigned to thread 1, and so on.



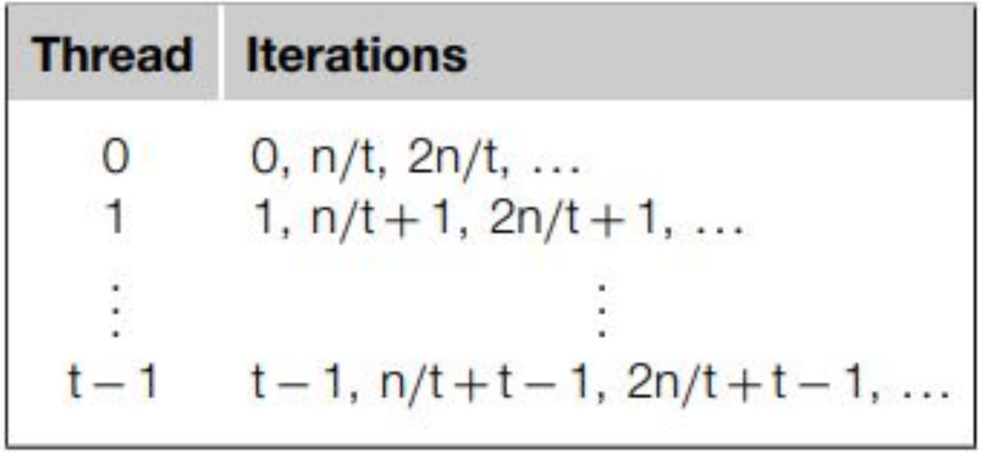
```
   fibo[0] = fibo[1] = 1;
#  pragma omp parallel for num_threads(thread_count)
   for (i = 2; i < n; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];
```

In addition to correctness issues due to loop carried dependencies, there can be load balancing issues.

Load balancing issue



```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```



Thread	Iterations
0	0, n/t, 2n/t, ...
1	1, n/t+1, 2n/t+1, ...
⋮	⋮
t−1	t−1, n/t+t−1, 2n/t+t−1, ...

Schedule clause

Schedule clause has the form schedule(<type>[,<chunksize>])

- Type can be any one of the following:
  - **static:** The iterations can be assigned to the threads before the loop is executed.
  - **dynamic or guided:** The iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system.
  - **auto:** The compiler and/or the run-time system determine the schedule.
  - **runtime:** The schedule is determined at run-time.
- Chunksize is a positive integer.
  - A chunk of iterations is a block of iterations that would be executed consecutively in the serial loop. The number of iterations in the block is the chunksize.
  - Only static, dynamic, and guided schedules can have a chunksize. This determines the details of the schedule, but its exact interpretation depends on the type.

Static schedule with chunksizes 1, 2, 4

Thread 0:	0, 3, 6, 9
Thread 1:	1, 4, 7, 10
Thread 2:	2, 5, 8, 11

Thread 0:	0, 1, 6, 7
Thread 1:	2, 3, 8, 9
Thread 2:	4, 5, 10, 11

Thread 0:	0, 1, 2, 3
Thread 1:	4, 5, 6, 7
Thread 2:	8, 9, 10, 11

Guided schedule

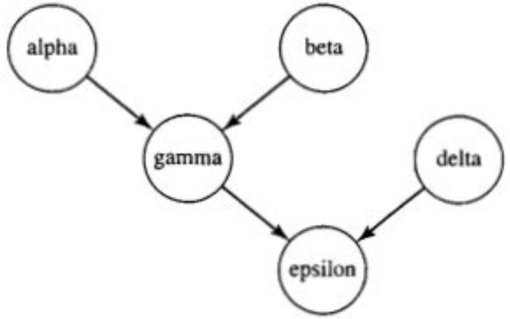
Thread	Chunk	Size of Chunk	Remaining Iterations
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

Discussion points

- **Hello world program, compile, run**
- **Synchronization**
  - **For mutual exclusion on shared data - critical, atomic, lock**
  - **For work coordination - barrier**
  - **Prevent data dependencies**
- **Scope of variables**
- **Sharing work among threads**
- Thread safety
- Task Parallelism
- Cache coherence, false sharing (during architecture discussion)
- Non parallelizable algorithms (during algorithm design discussion)

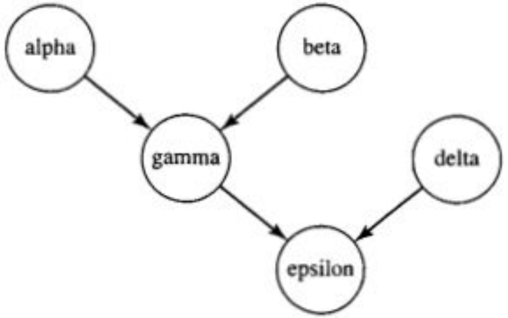
Task parallelism

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```



Task parallelism

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```



```
#pragma omp parallel sections
    {
#pragma omp section       /* This pragma optional */
       v = alpha();
#pragma omp section
       w = beta();
#pragma omp section
       y = delta();
    }
   x = gamma(v, w);
   printf ("%6.2f\n", epsilon(x,y));
```

```
#pragma omp parallel
   {
   #pragma omp sections
      {
      #pragma omp section      /* This pragma optional */
         v = alpha();
      #pragma omp section
         w = beta();
      }
   #pragma omp sections
      {
      #pragma omp section     /* This pragma optional */
         x = gamma(v, w);
      #pragma omp section
         y = delta();
      }
   }
   printf ("%6.2f\n", epsilon(x,y));
```

Discussion points

- **Hello world program, compile, run**
- **Synchronization**
  - **For mutual exclusion on shared data - critical, atomic, lock**
  - **For work coordination - barrier**
  - **Prevent data dependencies**
- **Scope of variables**
- **Sharing work among threads**
- **Thread safety**
- **Task Parallelism**
- Cache coherence, false sharing (during architecture discussion)
- Non parallelizable algorithms (during algorithm design discussion)

Course outline (Pacheco; GGKK; Quinn)

- Motivation (1;1;1)
- How to quantify performance improvement (2.6; 5; 7)
- **Parallel hardware architecture (2.2-2.3; 2,4; 2)**
- Parallel programming frameworks
  - Pthreads for shared memory (4; 7; -)
  - OpenMP for shared memory (5; 7.10; 17)
  - MPI for distributed memory (3; 6; 4)
  - CUDA/OpenCL for GPU,
  - Hadoop/Spark/Mapreduce for distributed systems
- Parallel program verification
- Parallel algorithm design
- Some case studies