Pthreads and OpenMP

Pacheko Chapters 4 and 5

Pthreads vs. OpenMP helloworld.c

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

/* Global variable:  accessible to all threads */
int thread_count;

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

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

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

   thread_handles = malloc (thread_count*sizeof(pthread_t));

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Hello, (void*) thread);

   printf("Hello from the main thread\n");

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   free(thread_handles);
   return 0;
}  /* main */

void* Hello(void* rank) {
   long my_rank = (long) rank
         /* Use long in case of 64-bit system */

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

   return NULL;
}  /* Hello */
```

```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 */
```

$gcc -g -Wall -o pth_helloworld pth_helloworld.c -lpthread

$gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

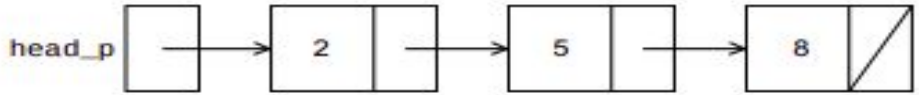
Maintaining correctness vs. performance

1. Histogram computation in CUDA, trapezoidal area computation in Pacheko, pi computation in Pacheko, are good candidates for parallelization, as summation is commutative and does not depend on order of threads. But manipulating the global sum forms **critical section**
2. **Long busy wait** with a single global sum variable accessed by all threads for all additions, **short busy wait** with private sum variables for each thread and only as many accesses to the global sum variable as number of threads, **mutex** to free CPU cores while waiting for the lock — all maintain correctness, with increasing performance
3. Message passing program mutual exclusion not enough, source thread should complete writing before destination thread reads — accomplished with **binary semaphore** protecting per thread's array entry

OpenMP has multiple primitives for thread synchronization - critical, atomic, locks, barrier ……

Concurrent Data Structures

```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```

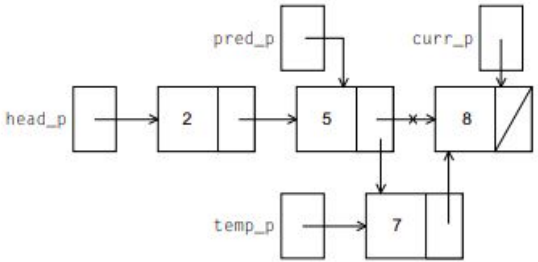



```
1  int  Member(int value, struct list_node_s* head_p) {
2     struct list_node_s* curr_p = head_p;
3
4     while (curr_p != NULL && curr_p->data < value)
5        curr_p = curr_p->next;
6
7     if (curr_p == NULL || curr_p->data > value) {
8        return 0;
9     } else {
10       return 1;
11    }
12 }  /* Member */
```

Insertion and deletion operations

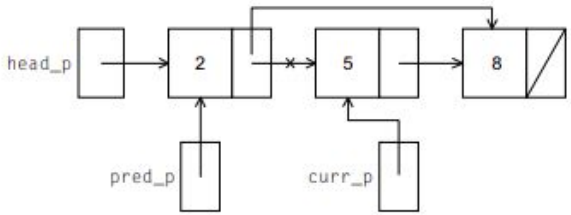
```
1  int Insert(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4     struct list_node_s* temp_p;
5
6     while (curr_p != NULL && curr_p->data < value) {
7        pred_p = curr_p;
8        curr_p = curr_p->next;
9     }
10
11    if (curr_p == NULL || curr_p->data > value) {
12       temp_p = malloc(sizeof(struct list_node_s));
13       temp_p->data = value;
14       temp_p->next = curr_p;
15       if (pred_p == NULL)  /* New first node */
16          *head_p = temp_p;
17       else
18          pred_p->next = temp_p;
19       return 1;
20    } else { /* Value already in list */
21       return 0;
22    }
23 }  /* Insert */
```

Program 4.10: The `Insert` function




```
1  int Delete(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4
5     while (curr_p != NULL && curr_p->data < value) {
6        pred_p = curr_p;
7        curr_p = curr_p->next;
8     }
9
10    if (curr_p != NULL && curr_p->data == value) {
11       if (pred_p == NULL) { /* Deleting first node in list */
12          *head_p = curr_p->next;
13          free(curr_p);
14       } else {
15          pred_p->next = curr_p->next;
16          free(curr_p);
17       }
18       return 1;
19    } else {  /* Value isn't in list */
20       return 0;
21    }
22 }  /* Delete */
```

Program 4.11: The `Delete` function



Multiple correctness alternatives vs. performance

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops, 99.9% `Member`, 0.05% `Insert`, 0.05% `Delete`

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

Potential thread issues: fairness, deadlocks, mixed primitives

1. Is fairness necessary?
   a. Not for pi, trapezoid, histogram as summation is commutative and order of threads doesn't matter
   b. For client server programs, fairness among threads might be needed to improve client satisfaction

   The busy-wait programs ensured fairness among threads, as they passed control from thread to thread by design, mutex doesn't ensure fairness as some threads might starve

2. Deadlocks with multiple critical sections that threads enter in opposite order.
3. Mixed primitives give no synchronization

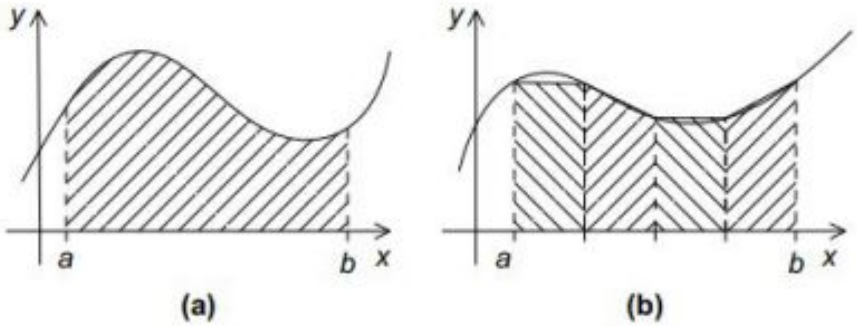
```
# pragma omp atomic
  x += f(y);
```

```
# pragma omp critical
  x = g(x);
```

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

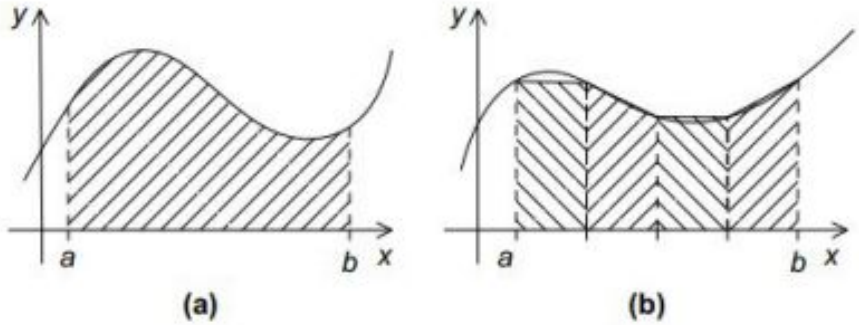
OpenMP automated parallelization of for loops

```
/* 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;
```



OpenMP automated parallelization of for loops

```
/* 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;
```




```
   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;
```

Restrictions on for loops for automated parallelization

- Only loops for which the number of iterations can be determined from the for statement itself and prior to execution of the loop.

```
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
```

- 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.

Fair work division among threads for automated for loop

```
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, …

Static schedule with chunksizes 1, 2, 4

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

Different schedules in OpenMP

- 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.

Runtime bugs with 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];
```

Dependencies for the same thread 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

```
1  #  pragma omp parallel for num_threads(thread_count)
2      for (i = 0; i < n; i++) {
3         x[i] = a + i*h;
4         y[i] = exp(x[i]);
5      }
```

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

Program order is maintained for the same thread.

Handling loop dependencies across threads

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

```
1       double factor = 1.0;
2       double sum = 0.0;
3       for (k = 0; k < n; k++) {
4          sum += factor/(2*k+1);
5          factor = -factor;
6       }
7       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

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

by

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

or, if you prefer the ?: operator,

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

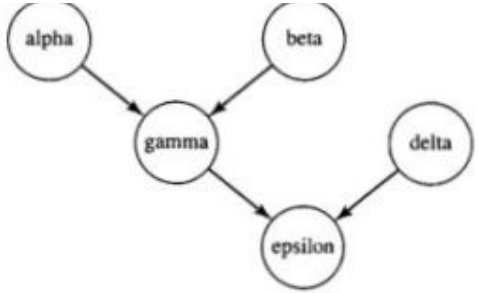
we will eliminate the loop dependency.

Still buggy due to variable scope issues

```
        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);
        }
```

Non for loop parallelization: dependencies and latencies

```
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));
```

Causes of incorrectness/issues in parallel programs

1. Global variables form critical sections requiring mutual exclusion — busy-wait, mutex
2. Ordering might be needed across threads -> produce before consume — semaphore
3. All threads might need to wait for an event before proceeding — barrier
4. Mixing synchronization primitives — stick to one primitive
5. Deadlocks with multiple critical sections — acquire multiple locks in order
6. Fairness issues across threads — programmatically create orders to acquire locks, load balance
7. Loop carried dependencies — break with code change
8. Incorrect scope of variables for different threads — default(none) private(var1, var2 ….)

Causes of incorrectness/issues in parallel programs

1. Global variables form critical sections requiring mutual exclusion — busy-wait, mutex
2. Ordering might be needed across threads -> produce before consume — semaphore
3. All threads might need to wait for an event before proceeding — barrier
4. Mixing synchronization primitives — stick to one primitive
5. Deadlocks with multiple critical sections — acquire multiple locks in order
6. Fairness issues across threads — programmatically create orders to acquire locks, load balance
7. Loop carried dependencies — break with code change
8. Incorrect scope of variables for different threads — default(none) private(var1, var2 ….)

Measure performance and see if the trouble is worth the gain.