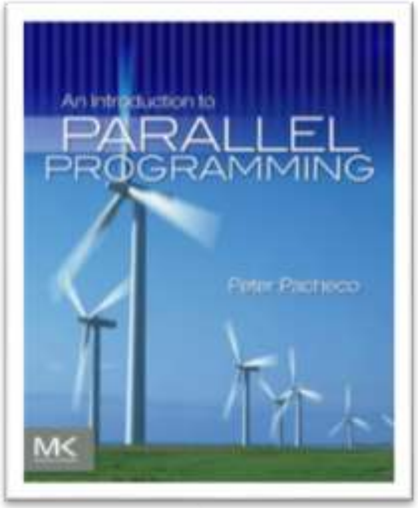
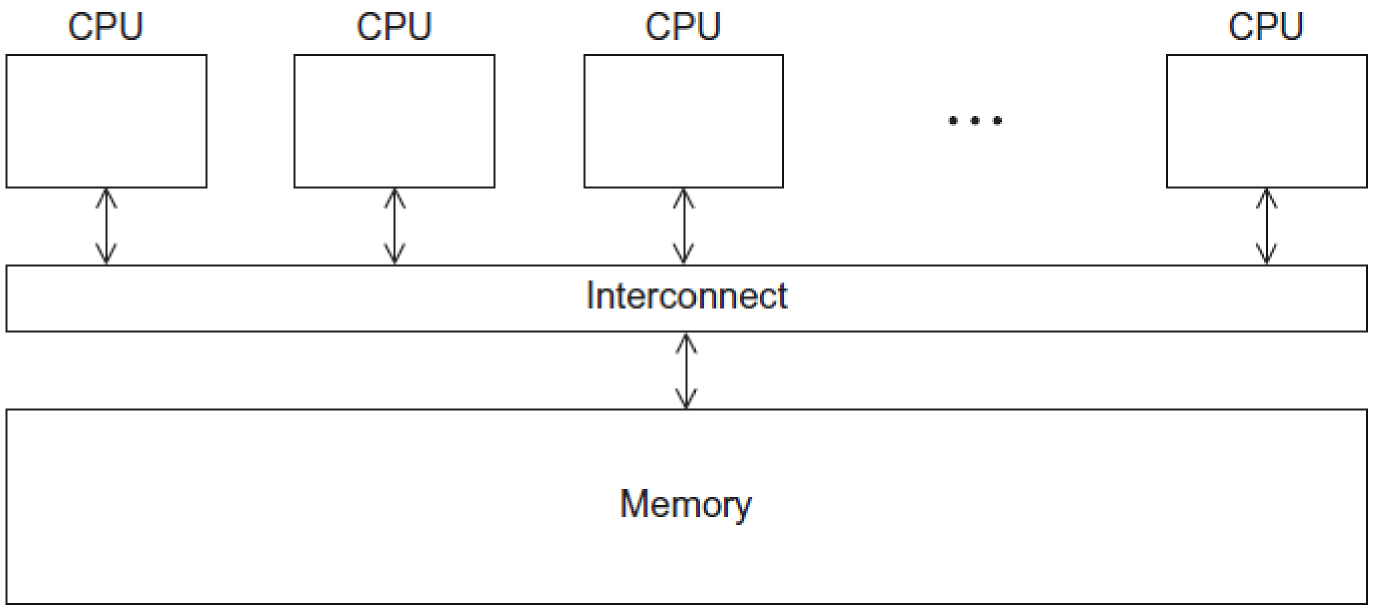


Shared Memory Programming with Pthreads & OpenMP

Adeesha Wijayasiri

Slides extended from
An Introduction to Parallel Programming by
Peter Pacheco & Dr.Dilum Bandara

Shared Memory System



POSIX® Threads

- Also known as Pthreads
- Standard for Unix-like operating systems
- Library that can be linked with C programs
- Specifies an API for multi-threaded programming

Hello World!

```
#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));
```

Declares various Pthreads functions, constants, types, etc.

Hello World! (Cont.)

```
   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 */
```

Hello World! (Cont.)

```
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 */
```

Compiling a Pthread program

```
gcc −g −Wall −o pth_hello pth_hello.c −lpthread
```

Link Pthreads library

Running a Pthreads program

. /pth_hello <number of threads>

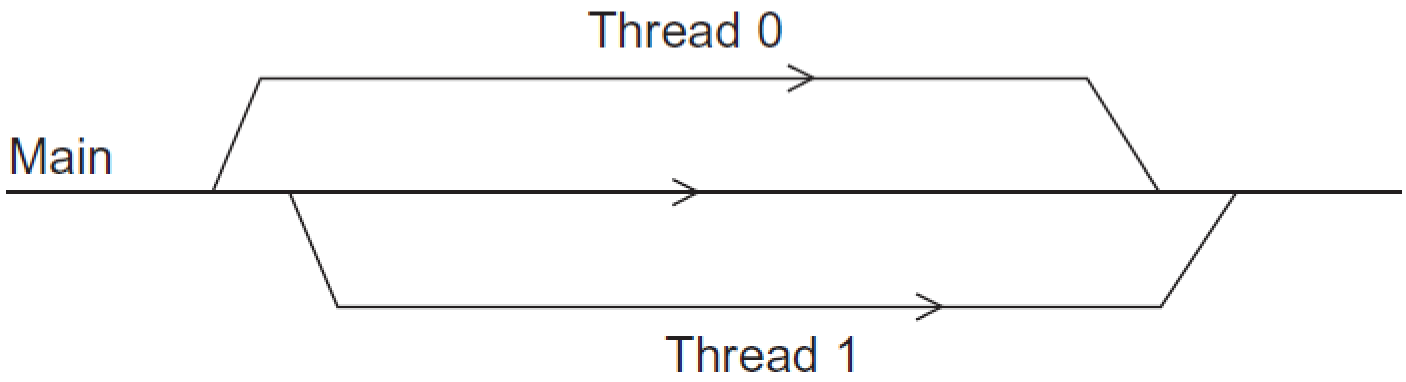
. /pth_hello 1

Hello from the main thread
Hello from thread 0 of 1

. /pth_hello 4

Hello from the main thread
Hello from thread 0 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 1 of 4

Running the Threads



Main thread forks & joins 2 threads

Global Variables

- Can introduce subtle & confusing bugs!
- Use them only when they are essential

Starting Threads

pthread.h

pthread_t

One object for each thread

We ignore return value from pthread_create

```
int pthread_create (
        pthread_t*  thread_p,                    /* out */
        const pthread_attr_t*  attr_p,           /* in */
        void*  (*start_routine) (void),          /* in */
        void*  arg_p);                           /* in */
```

Function Started by pthread_create

- Function start by pthread_create should have following prototype
  void* thread_function (void* args_p) ;

- Void* can be cast to any pointer type in C
  - So args_p can point to a list containing one or more values needed by thread_function
- Similarly, return value of thread_function can point to a list of one or more values

Stopping Threads

- Single call to pthread_join will wait for thread associated with pthread_t object to complete
  - Suspend execution of calling thread until target thread terminates, unless it has already terminated
  - Call pthread_join once for each thread

```
int pthread_join(
        pthread_t*  thread          /* in */ ,
        void**  ret_val_p           /* out */ ) ;
```

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

Matrix-Vector Multiplication in Pthreads

Serial Pseudo-code

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$

```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

Using 3 Pthreads

- Assign each row to a separate thread
- Suppose 6x6 matrix & 3 threads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

Thread 0

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

General case

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

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 */
```

Estimating π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

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

Thread Function for Computing π

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

Using a dual core processor

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

As we increase *n*, estimate with 1 thread gets better & better

2 thread case produce different answers in different runs

Why?

Pthreads Global Sum with Busy-Waiting

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

Shared variable

Mutexes

- Make sure only 1 thread in critical region
- Pthreads standard includes a special type for mutexes: pthread_mutex_t

```
int pthread_mutex_init(
      pthread_mutex_t*             mutex_p   /* out */
      const pthread_mutexattr_t*   attr_p    /* in  */);
```

Mutexes

- Lock
  - To gain access to a critical section

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p  /* in/out */);
```

- Unlock
  - When a thread is finished executing code in a critical section

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

- Termination
  - When a program finishes using a mutex

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```

Global Sum Function Using a Mutex

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;
```

Global Sum Function Using a Mutex (Cont.)

```
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

Busy-Waiting vs. Mutex

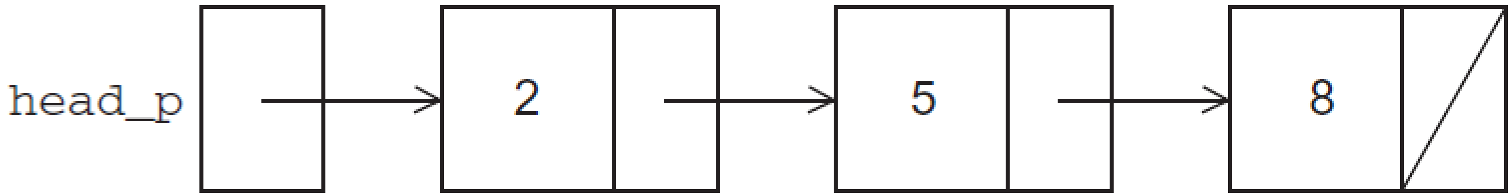
Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

Run-times (in seconds) of π programs using $n = 108$ terms on a system with 2x4-core processors

Read-Write Locks

- While controlling access to a large, shared data structure
- Example
  - Suppose shared data structure is a sorted linked list of ints, & operations of interest are Member, Insert, & Delete

Linked Lists

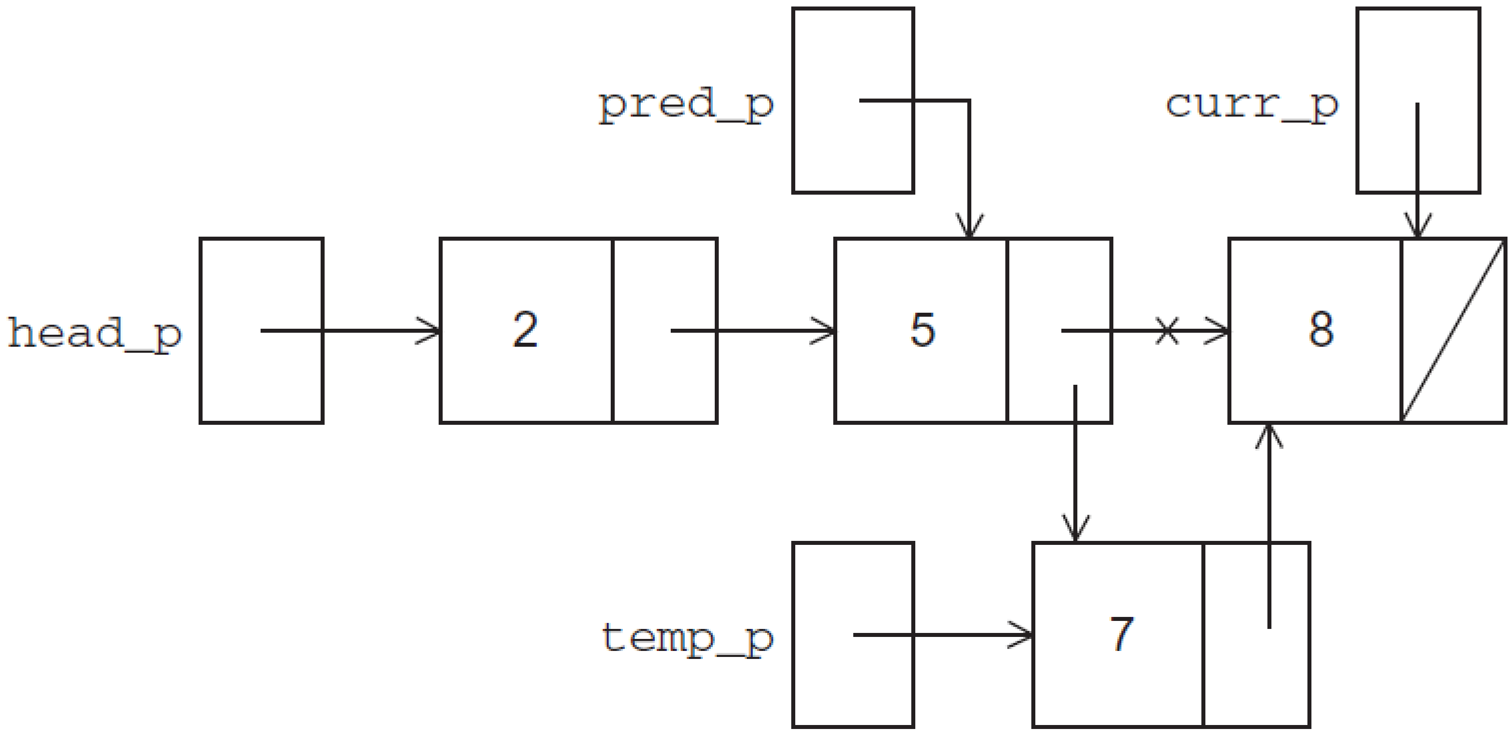



```
struct list_node_s {
    int data;
    struct list_node_s* next;
}
```

Linked List Membership

```
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* curr_p = head_p;

   while (curr_p != NULL && curr_p->data < value)
      curr_p = curr_p->next;

   if (curr_p == NULL || curr_p->data > value) {
      return 0;
   } else {
      return 1;
   }
}  /* Member */
```

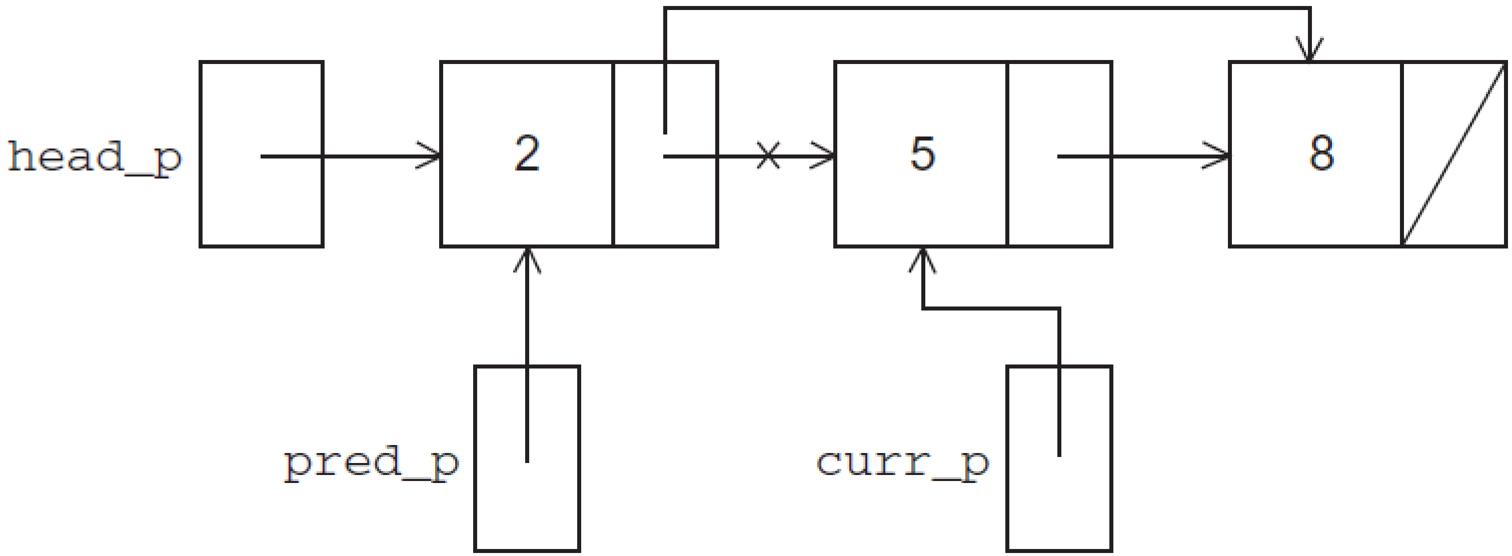
Inserting New Node Into a List



Inserting New Node Into a List (Cont.)

```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL)  /* New first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```

Deleting a Node From a Linked List



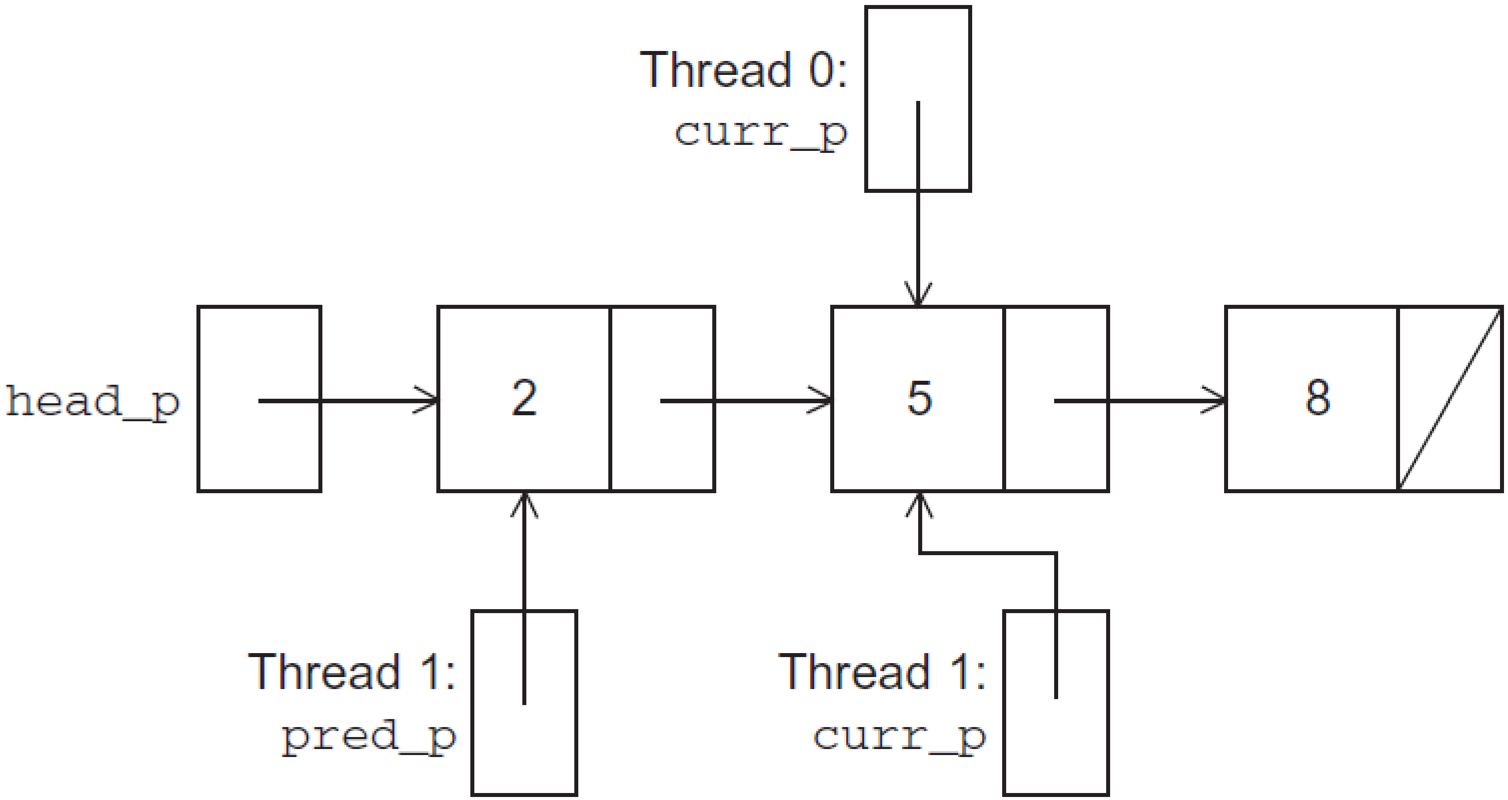
Deleting a Node From a Linked List (Cont.)

```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* Value isn't in list */
      return 0;
   }
} /* Delete */
```

Multi-Threaded Linked List

- To share access to the list, we can define head_p to be a global variable
  - This will simplify function headers for Member, Insert, & Delete
  - Because we won't need to pass in either head_p or a pointer to head_p: we'll only need to pass in the value of interest

Simultaneous Access by 2 Threads



Solution #1

- Simply lock the list any time that a thread attempts to access it
- Call to each of the 3 functions can be protected by a mutex

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

In place of calling Member(value).

Issues

- Serializing access to the list
- If vast majority of our operations are calls to Member
  - We fail to exploit opportunity for parallelism
- If most of our operations are calls to Insert & Delete
  - This may be the best solution

Solution #2

- Instead of locking entire list, we could try to lock individual nodes
- A “finer-grained” approach

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

Issues

- Much more complex than original Member function
- Much slower
  - Because each time a node is accessed, a mutex must be locked & unlocked
  - Addition of a mutex field to each node substantially increase memory needed for the list

Pthreads Read-Write Locks

- Neither multi-threaded linked lists exploits potential for simultaneous access to any node by threads that are executing Member
  - 1st solution only allows 1 thread to access the entire list at any instant
  - 2nd only allows 1 thread to access any given node at any instant
- Read-write lock is somewhat like a mutex except that it provides 2 lock functions
- 1st locks the read-write lock for reading
- 2nd locks it for writing

Pthreads Read-Write Locks (Cont.)

- Multiple threads can simultaneously obtain lock by calling read-lock function
- While only 1 thread can obtain lock by calling write-lock function
- Thus
  - If any thread owns lock for reading, any thread that wants to obtain a lock for writing will be blocked
  - If any thread owns lock for writing, any threads that want to obtain lock for reading or writing will be blocked

Protecting Our Linked List Functions

```
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);
```

Linked List Performance

100,000 ops/thread
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

100,000 ops/thread
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

OpenMP

OpenMP

- High-level API for shared-memory parallel programming
  - MP = multiprocessing
- Use Pragmas
  - Special preprocessor instructions
    - #pragma
    - Typically added to support behaviors that aren't part of the basic C specification

```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 & Running

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

compiling

```
. / omp_hello 4
```

running with 4 threads

possible outcomes

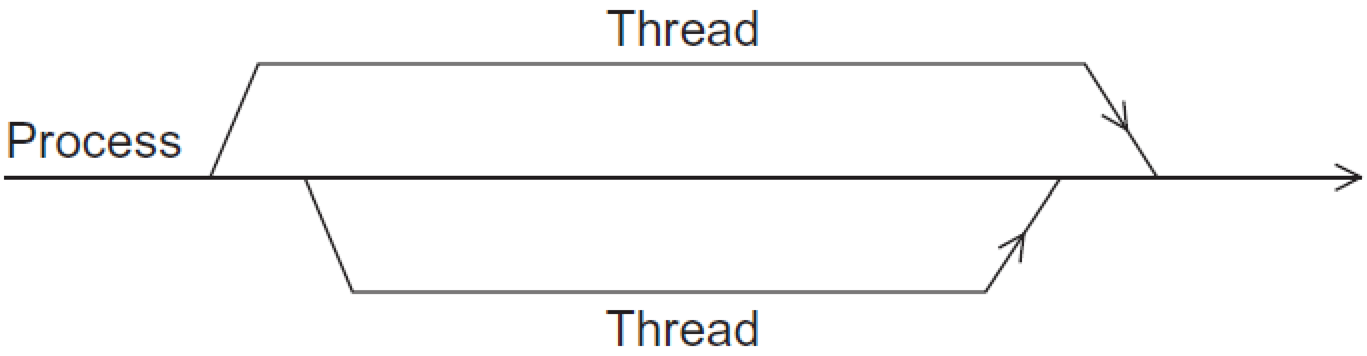
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

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

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

OpenMp pragmas

- # pragma omp parallel
  - Most basic parallel directive
  - Original thread is called master
  - Additional threads are called slaves
  - Original thread & new threads called a team



Clause

- Text that modifies a directive
- *num_threads* clause can be added to a parallel directive
- Allows programmer to specify no of threads that should execute following block

```
# pragma omp parallel num_threads ( thread_count )
```

Be Aware…

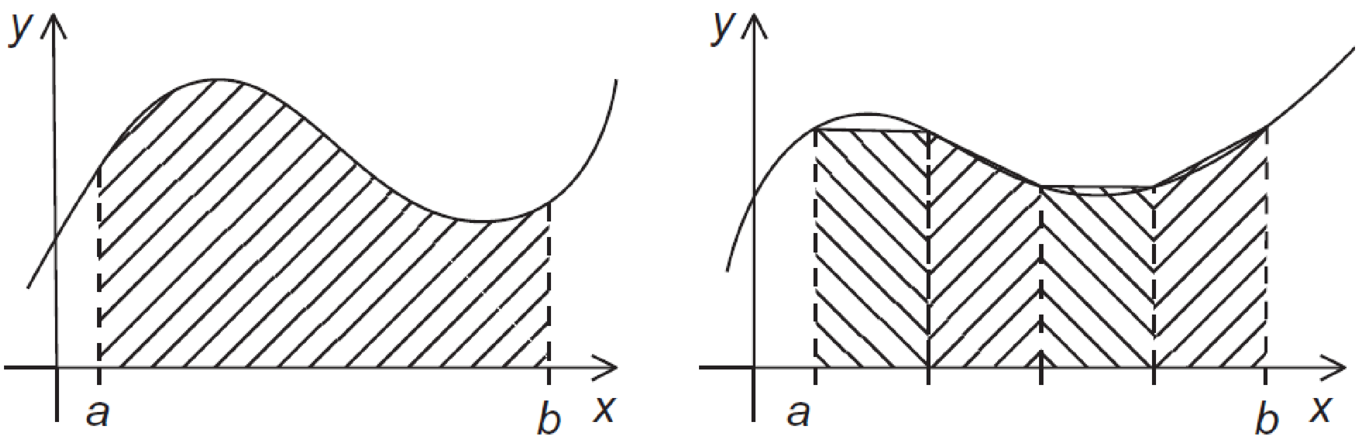
- There may be system-defined limitations on number of threads that a program can start
- OpenMP standard doesn’t guarantee that this will actually start *thread_count* threads
- Most current systems can start hundreds or even 1,000s of threads
- Unless we’re trying to start a lot of threads, we will almost always get desired no of threads

Mutual Exclusion

```
# pragma omp critical
   global_result += my_result ;
```

only 1 thread can execute following structured block at a time

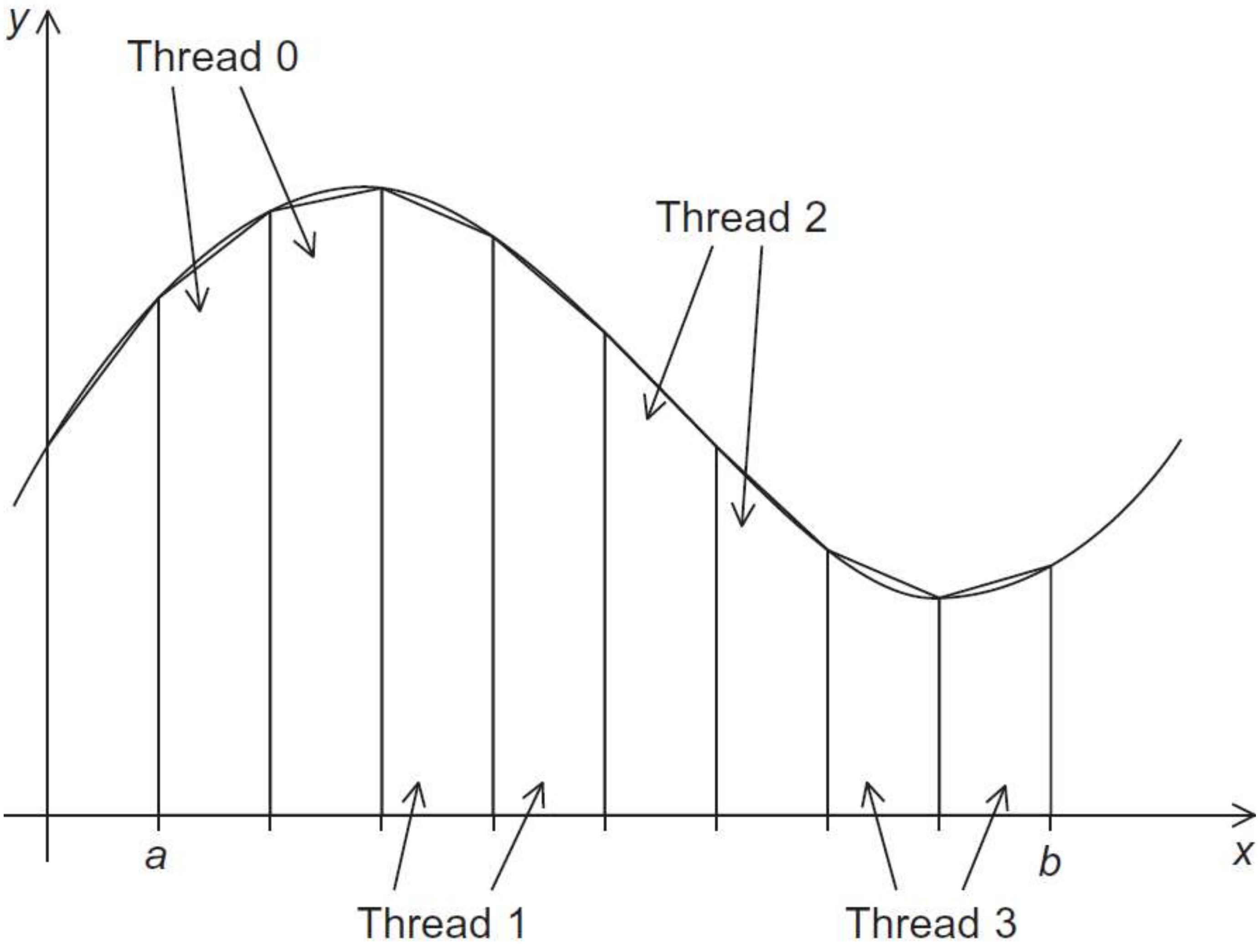
Trapezoidal Rule



Serial algorithm

```
/* 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;
```

Assignment of Trapezoids to Threads



```
#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;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   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 */
```