ITP 30002 Operating System

Lock-based Concurrent Data Structures

OSTEP Chapter 29

Shin Hong

Performance of Concurrent Algorithm

- Amdahl's law
  - Upper bound of performance gains by adding additional cores to an application that has both serial and parallel components
    - *S* is serial portion, *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

    - That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
    - As *N* approaches infinity, speedup approaches 1 / *S*
  - Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- Cost of context-switching and synchronization operation

Concurrent Counter – Naïve Thread Safe Version

```
1   typedef struct __counter_t {
2       int value;
3   } counter_t;
4
5   void init(counter_t *c) {
6       c->value = 0;
7   }
8
9   void increment(counter_t *c) {
10      c->value++;
11  }
12
13  void decrement(counter_t *c) {
14      c->value--;
15  }
16
17  int get(counter_t *c) {
18      return c->value;
19  }
```

⟶

```
1   typedef struct __counter_t {
2       int                value;
3       pthread_mutex_t lock;
4   } counter_t;
5
6   void init(counter_t *c) {
7       c->value = 0;
8       Pthread_mutex_init(&c->lock, NULL);
9   }
10
11  void increment(counter_t *c) {
12      Pthread_mutex_lock(&c->lock);
13      c->value++;
14      Pthread_mutex_unlock(&c->lock);
15  }
16
17  void decrement(counter_t *c) {
18      Pthread_mutex_lock(&c->lock);
19      c->value--;
20      Pthread_mutex_unlock(&c->lock);
21  }
22
23  int get(counter_t *c) {
24      Pthread_mutex_lock(&c->lock);
25      int rc = c->value;
26      Pthread_mutex_unlock(&c->lock);
27      return rc;
28  }
```
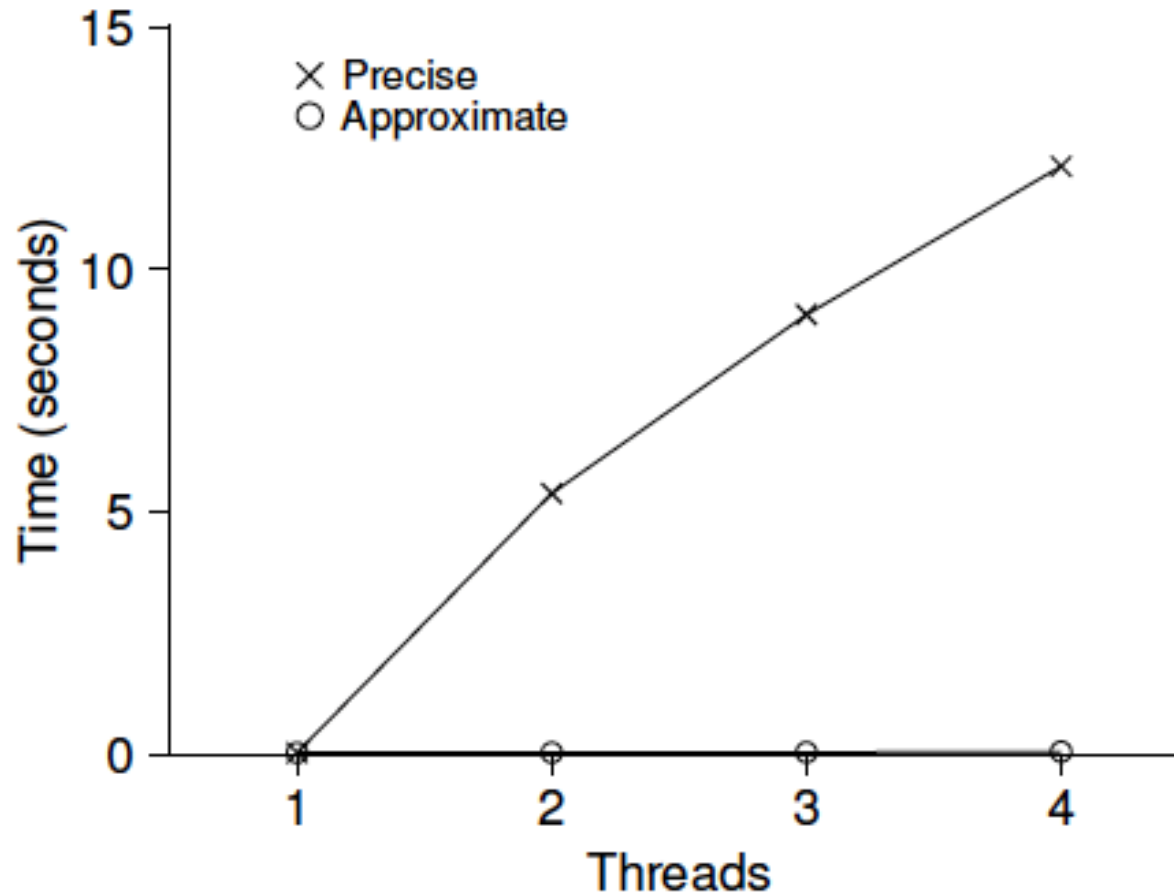
Lock-based Concurrent Data Structures

ITP 30002 Operating System

2023-06-06

Concurrent Counter – Scalability Issue

Lock-based
Concurrent Data
Structures

ITP 30002
Operating System

2023-06-06

Approximate Counter

- Each thread updates on a local counter and periodically synchronize it with the global counter

```
1   typedef struct __counter_t {
2       int             global;
3       pthread_mutex_t glock;
4       int             local[NUMCPUS];
5       pthread_mutex_t llock[NUMCPUS];
6       int             threshold;
7   } counter_t;


11  void init(counter_t *c, int threshold) {
12      c->threshold = threshold;
13      c->global = 0;
14      pthread_mutex_init(&c->glock, NULL);
15      int i;
16      for (i = 0; i < NUMCPUS; i++) {
17          c->local[i] = 0;
18          pthread_mutex_init(&c->llock[i], NULL);
19      }
20  }
```

```
25  void update(counter_t *c, int threadID, int amt) {
26      int cpu = threadID % NUMCPUS;
27      pthread_mutex_lock(&c->llock[cpu]);
28      c->local[cpu] += amt;
29      if (c->local[cpu] >= c->threshold) {
30          // transfer to global (assumes amt>0)
31          pthread_mutex_lock(&c->glock);
32          c->global += c->local[cpu];
33          pthread_mutex_unlock(&c->glock);
34          c->local[cpu] = 0;
35      }
36      pthread_mutex_unlock(&c->llock[cpu]);
```

```
40  int get(counter_t *c) {
41      pthread_mutex_lock(&c->glock);
42      int val = c->global;
43      pthread_mutex_unlock(&c->glock);
44      return val; // only approximate!
45  }
```

Scaling Approximate Counter
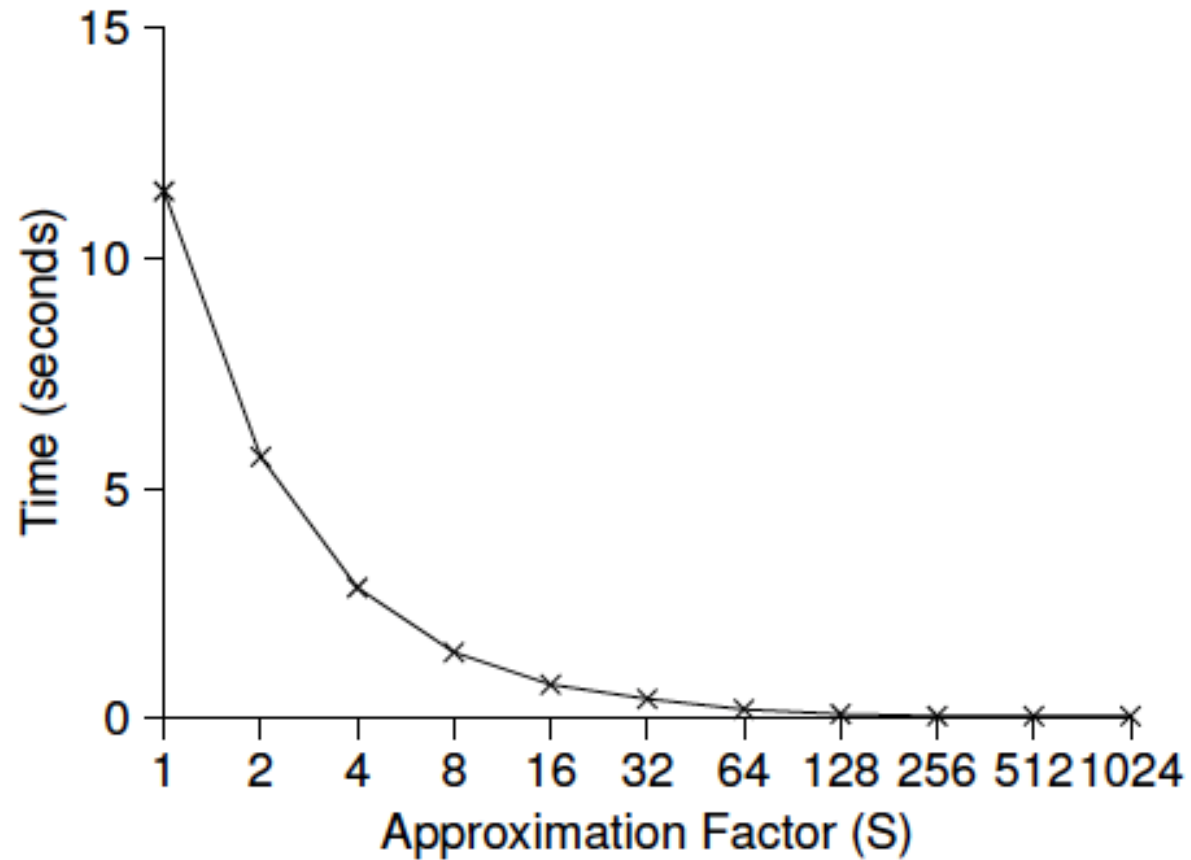
```
1   // basic node structure
2   typedef struct __node_t {
3       int                 key;
4       struct __node_t         *next;
5   } node_t;
6
7   // basic list structure (one used per list)
8   typedef struct __list_t {
9       node_t              *head;
10      pthread_mutex_t     lock;
11  } list_t;
12
13  void List_Init(list_t *L) {
14      L->head = NULL;
15      pthread_mutex_init(&L->lock, NULL);
16  }
```

```
18  int List_Insert(list_t *L, int key) {
19      pthread_mutex_lock(&L->lock);
20      node_t *new = malloc(sizeof(node_t));
21      if (new == NULL) {
22          perror("malloc");
23          pthread_mutex_unlock(&L->lock);
24          return -1; // fail
25      }
26      new->key  = key;
27      new->next = L->head;
28      L->head   = new;
29      pthread_mutex_unlock(&L->lock);
30      return 0; // success
31  }
32
33  int List_Lookup(list_t *L, int key) {
34      pthread_mutex_lock(&L->lock);
35      node_t *curr = L->head;
36      while (curr) {
37          if (curr->key == key) {
38              pthread_mutex_unlock(&L->lock);
39              return 0; // success
40          }
41          curr = curr->next;
42      }
43      pthread_mutex_unlock(&L->lock);
44      return -1; // failure
45  }
```

Scaling Linked List

- Reduce locking scope

- Use finer-grained lock
  - Add a lock per node, instead of one global lock
  - Hand-over-hand locking

```
6   void List_Insert(list_t *L, int key) {
7       // synchronization not needed
8       node_t *new = malloc(sizeof(node_t));
9       if (new == NULL) {
10          perror("malloc");
11          return;
12      }
13      new->key = key;
14
15      // just lock critical section
16      pthread_mutex_lock(&L->lock);
17      new->next = L->head;
18      L->head    = new;
19      pthread_mutex_unlock(&L->lock);
20  }
```

Lock-based
Concurrent Data
Structures

ITP 30002
Operating System

2023-06-06

Concurrent Queue

```
1  typedef struct __node_t {
2      int                    value;
3      struct __node_t     *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t                *head;
8      node_t                *tail;
9      pthread_mutex_t       head_lock, tail_lock;
10  } queue_t;


12  void Queue_Init(queue_t *q) {
13      node_t *tmp = malloc(sizeof(node_t));
14      tmp->next = NULL;
15      q->head = q->tail = tmp;
16      pthread_mutex_init(&q->head_lock, NULL);
17      pthread_mutex_init(&q->tail_lock, NULL);
18  }
```

```
20  void Queue_Enqueue(queue_t *q, int value) {
21      node_t *tmp = malloc(sizeof(node_t));
22      assert(tmp != NULL);
23      tmp->value = value;
24      tmp->next  = NULL;
25
26      pthread_mutex_lock(&q->tail_lock);
27      q->tail->next = tmp;
28      q->tail = tmp;
29      pthread_mutex_unlock(&q->tail_lock);
30  }

32  int Queue_Dequeue(queue_t *q, int *value) {
33      pthread_mutex_lock(&q->head_lock);
34      node_t *tmp = q->head;
35      node_t *new_head = tmp->next;
36      if (new_head == NULL) {
37          pthread_mutex_unlock(&q->head_lock);
38          return -1; // queue was empty
39      }
40      *value = new_head->value;
41      q->head = new_head;
42      pthread_mutex_unlock(&q->head_lock);
43      free(tmp);
44      return 0;
45  }
```

```
1   int buffer[MAX];
2   int fill_ptr = 0;
3   int use_ptr  = 0;
4   int count    = 0;
5
6   void put(int value) {
7       buffer[fill_ptr] = value;
8       fill_ptr = (fill_ptr + 1) % MAX;
9       count++;
10  }
11
12  int get() {
13      int tmp = buffer[use_ptr];
14      use_ptr = (use_ptr + 1) % MAX;
15      count--;
16      return tmp;
17  }
```

```
1   void *producer(void *arg) {
2       int i;
3       int loops = (int) arg;
4       for (i = 0; i < loops; i++) {
5           put(i);
6       }
7   }
8
9   void *consumer(void *arg) {
10      while (1) {
11          int tmp = get();
12          printf("%d\n", tmp);
13      }
14  }
```

```
1   cond_t empty, fill;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);
8           while (count == MAX)
9               Pthread_cond_wait(&empty, &mutex);
10          put(i);
11          Pthread_cond_signal(&fill);
12          Pthread_mutex_unlock(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);
20          while (count == 0)
21              Pthread_cond_wait(&fill, &mutex);
22          int tmp = get();
23          Pthread_cond_signal(&empty);
24          Pthread_mutex_unlock(&mutex);
25          printf("%d\n", tmp);
26      }
27  }
```