InfoLab

IC621: Distributed and Parallel Computing

Lecture 05 : Shared-Memory Programming with Pthreads (I)

Min-Soo Kim

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Processes, Threads, and Pthreads

- **Thread: an instance of a program running on a processor (lighter-weight than a process in MPI)**
  - thread is often called light-weight process
  - the term “thread” comes from the concept of “thread of control”
  - a thread of control is just a sequence of statements in a program
  - a single process may have multiple threads of control

- **Pthread : POSIX thread**
  - POSIX: a standard for Unix-like operating systems (e.g., Linux and Mac OS X)
  - POSIX specifies an API for multithreaded programming
    - Pthread is not a programming language
    - Pthread specifies a library that can be linked with C programs
  - Pthread API is only available on POSIX systems
    - e.g., Linux, Mac OS X, Solaris, HPUX, and so on

- **There are a number of other widely used specifications for multithreaded programming (unlike MPI)**
  - e.g., Java threads, Windows threads, Solaris threads
  - c.f., new C++ standard (C++0x) for shared memory programming
- **Thread consists of**
  - a block of memory for the stack
  - a block of memory for the heap
  - descriptors of resources that the system has allocated for the process
    - e.g., file descriptors
  - security information
    - e.g., information about which hardware and software resources the process can access
  - information about the state of the process
    - e.g., whether the process is ready to run or is waiting on a resource
    - e.g., the content of the registers including the program counter

InfoLab

HELLO, WORLD

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Hello world example using Pthreads

- **Execution**

```
$ gcc -g -Wall -o pth_hello pth_hello.c -lpthread

$ ./pth_hello 4

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

```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 */
```

- **`pthread.h` (Pthreads header file)**
  - declares the various Pthreads functions, constants, types, …
- **`thread_count`**
  - a global variable shared by all the threads
- **multiple threads are executing the same function Hello**
  - each thread will have its own private copies of the local variables and function arguments (e.g., `rank, my_rank`)
- **`pthread_t` (opaque object)**
  - actual data that they store is system specific
  - data members aren't directly accessible to user code

pthread_create

- 1st argument must be allocated before the call
- 3rd argument : the function that the thread is to run
- 4th argument : a pointer to the argument that should be passed to the function `start_routine`
- return value : indicates if there's been an error in the function call

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

void* thread_function(void* args_p)

- `args_p` can point to a list containing one or more values needed by `thread_function`
- return value of `thread_function` can point to a list of one or more values

Passing rank to `thread_function`

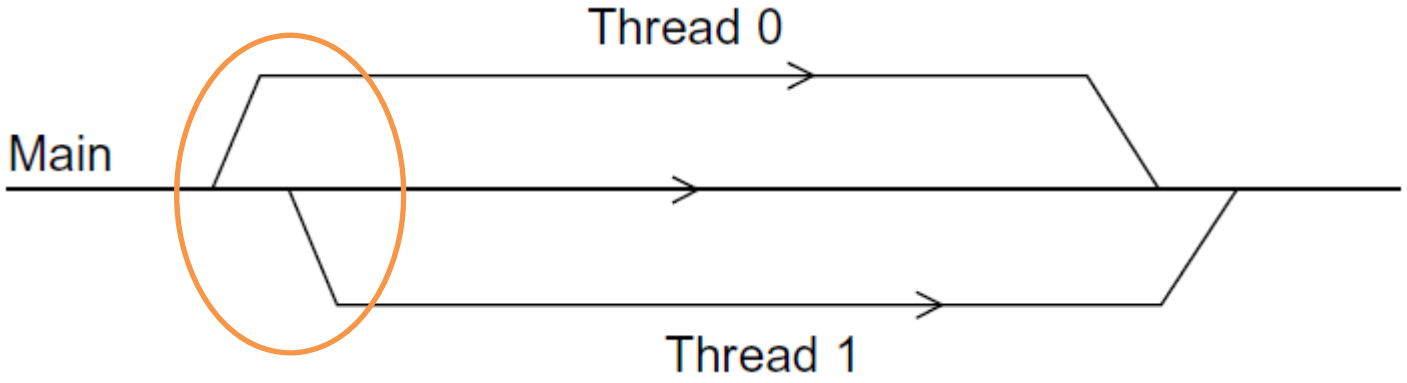
- we assign the first thread rank 0, and the second thread rank 1, …
  - casting `int` to `void*`, and then, `void*` to `int`
  - “system-defined” casting, but most C compilers do allow that
- we can easily determine which thread ran into trouble by just including the thread’s rank in the error message
- note: we can’t just print out the `pthread_t` object since it’s opaque

No reason for each thread to run the same function

- we could have one thread run `hello`, another run `goodbye`
- c.f., typically “single program, multiple data” style parallelism in MPI

Running the threads

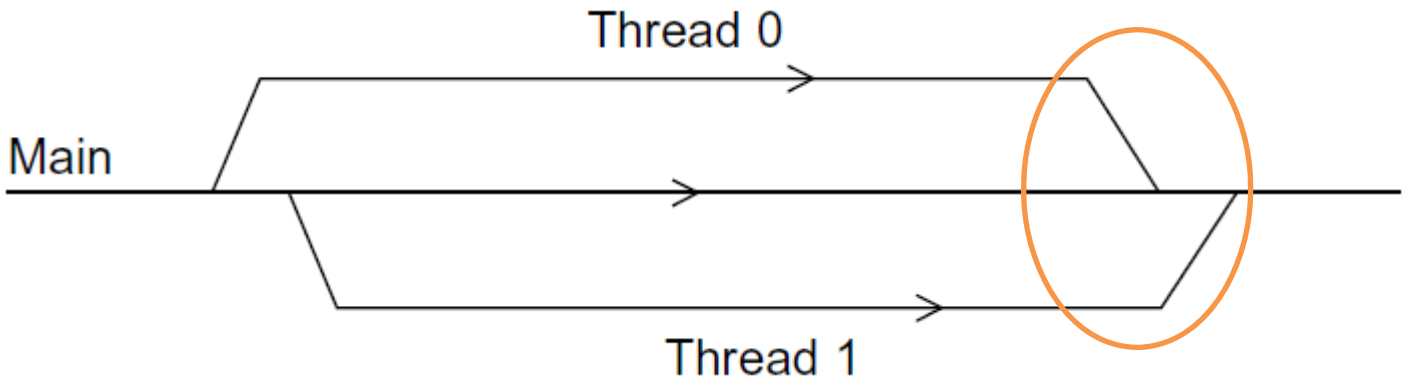
- **Programmer doesn't directly control where the threads are run**
  - there's no argument in `pthread_create` saying which core should run which thread
  - thread placement is controlled by OS
  - however, if there is a core that isn't being used, OS typically places a new thread on such a core



Stopping the threads

- **`pthread_join`**
  - a single call to it waits for the thread associated with the `pthread_t` object to complete
  - 2nd argument can be used to receive any return value computed by the thread

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

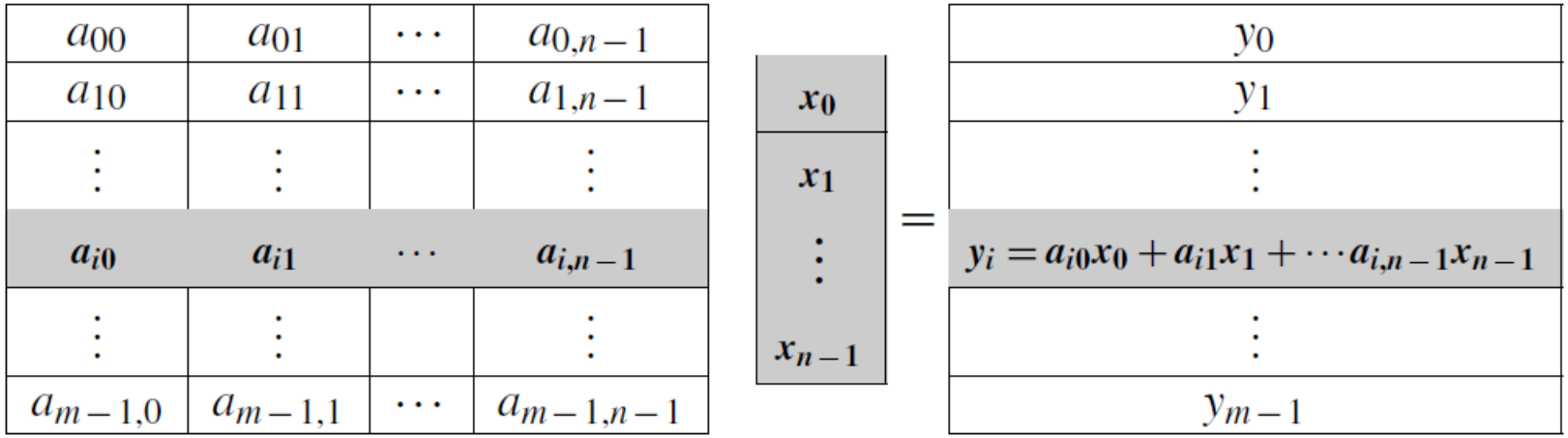


InfoLab

MATRIX-VECTOR MULTIPLICATION

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Matrix-vector multiplication




```
/* 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];
}
```

Suppose that m = n = 6 and thread_count = 3

```
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 */
```

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

Compare with MPI version

- each MPI process only has direct access to its own local memory
- we need to explicitly gather all of x into each process' memory

InfoLab

CRITICAL SECTIONS

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Estimating the value of π

- **One of the simplest formulas**

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

- **Serial code**

```
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;
```

- **Parallel code (using `t` threads)**
  - evenly divides the number of terms in the sum, n
  - thread q the loop variable will range over (`n' = n / t`)

    `qn', qn'+1, qn'+2, … , (q + 1)n' - 1`

Thread function

```
1   void* Thread_sum(void* rank) {
2      long my_rank = (long) rank;
3      double factor;
4      long long i;
5      long long my_n = n/thread_count;
6      long long my_first_i = my_n*my_rank;
7      long long my_last_i = my_first_i + my_n;
8
9      if (my_first_i % 2 == 0)  /* my_first_i is even */
10        factor = 1.0;
11     else  /* my_first_i is odd */
12        factor = -1.0;
13
14     for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15        sum += factor/(2*i+1);
16     }
17
18     return NULL;
19  }  /* Thread_sum */
```

// should be a critical section

Results of computation

- as we increase n, the estimate with one thread gets better and better
- however, for larger values of n, the result computed by two threads actually gets worse

	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

Cause of the problem

- multiple threads try to update a single shared variable, `sum`
- race condition : fundamental problem in shared-memory programming

InfoLab

BUSY-WAITING

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Simple example

- **Problematic code**

```
y = Compute(my_rank);
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location `x`	Add 0 and 2
7		Store 2 in memory location `x`

- **Busy-waiting code**
  - initially, `flag` is set to 0
  - thread 1 cannot enter the critical section until thread 0 has completed it

```
1  y = Compute(my_rank);
2  while (flag != my_rank);
3  x = x + y;
4  flag++;
```

Compiler optimization

- the compiler might make changes that will affect the correctness of busy-waiting
- it doesn't "know" that the variables `x` and `flag` can be modified by another thread

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

```
y = Compute(my_rank);
x = x + y;
while (flag != my_rank);
flag++;
```

- the changes might defeat the purpose of the busy-wait loop
- the simplest solution : turn it off when using busy-waiting

Problem about simple increment of flag

- in π calculation
  - eventually flag will be greater than t, the number of threads
  - none of the threads will be able to return to the critical section
  - all the threads will be stuck forever in the busy-wait loop

Correct busy-waiting

- the last thread, thread `t-1`, reset `flag` to zero (instead of simply increasing `flag`)

```
14      for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15         while (flag != my_rank);
16         sum += factor/(2*i+1);
17         flag = (flag+1) % thread_count;
18      }
```

- now, the code is computing the correct results

However, still performance problem

- thread 1 must wait until thread 0 executes the critical section and increments flag
- thread 0 must wait until thread 1 executes and increments
- the serial sum (e.g., 2.8 seconds) is consistently faster than the parallel sum (e.g., 19.5 seconds)

One solution

- each thread use a private variable to store its total contribution to the sum
- now, the elapsed time is reduced to 1.5 seconds

```
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;
}
```

```
for (i = my_first_i; i < my_last_i; i++, factor = -factor)
   my_sum += factor/(2*i+1);

while (flag != my_rank);
sum += my_sum;
flag = (flag+1) % thread_count;
```

InfoLab

MUTEXES

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Mutual exclusion (Mutex)

- **A special type of variable (together with a couple of special functions)**
  - provide restrict access to a critical section to a single thread at a time
  - guarantee that one thread "excludes" all other threads while it executes the critical section
- **`pthread_mutex_t`**
  - needs to be initialized by the system before it's used

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

  - needs to be destroyed after finishing using a mutex

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

Making critical section

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

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

```
for (i = my_first_i; i < my_last_i; i++, factor = -factor)
   my_sum += factor/(2*i+1);

while (flag != my_rank);
sum += my_sum;
flag = (flag+1) % thread_count;
```

```
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);
```

- **The order in which the threads execute the code in the critical section is more or less random**
  - Pthreads doesn't guarantee that the threads will obtain the lock in the order in which they called `Pthread_mutex_lock`
  - it is unlike the busy-waiting solution
  - however, eventually each thread will obtain the lock

- **Performance comparison with busy-waiting**

  - each thread only enters the critical section once
  - so, threads are not delayed very much by waiting to enter the critical section
  - busy-waiting performance can degrade if there are more threads than cores

Table 4.1 Run-Times (in Seconds) of π Programs Using $n = 10^8$ Terms on a System with Two Four-Core Processors

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

Table 4.2 Possible Sequence of Events with Busy-Waiting and More Threads than Cores

Time	flag	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
0	0	crit sect	busy-wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy-wait	susp
2	2	—	terminate	susp	busy-wait	busy-wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy-wait

InfoLab

Thank you!

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology