EDA284 – Lab 1

In this lab we will learn how to parallelize programs with pthreads and openmp. You will do a series of instructive exercises for each programming model and then use what you have learned to study Amdahl's law in practice.

Part 1: Pthreads

What is a Thread?

A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.

In general:
- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.
- Threads have their own independent flow of control

Important information about threads:
- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
- Two pointers (in two threads within the same process) having the same value point to the same data
- Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

On modern, multi-core machines, pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel pthreads programs

Thread-safeness:
Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.

The Pthreads API:
- Constists of a series of functions that can create and manage threads among others (around 100 functions in total)

Creating Threads:

- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

- pthread_create creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

pthread_create arguments:
pthread_create (thread, attr, start_routine, arg)
- **thread**: An opaque, unique identifier for the new thread returned by the subroutine.

- **attr**: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
- **start_routine**: the C routine that the thread will execute once it is created.
- **arg**: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Terminating Threads:
Threads terminate when:
- They returns normally from its starting routine. Its work is done.
- make a call to the pthread_exit subroutine - whether its work is done or not.
- The entire process is terminated due to making a call to either the exec() or exit()
- If main() finishes first, without calling pthread_exit explicitly itself

The pthread_exit() routine allows the programmer to specify an optional termination status parameter. This optional parameter is typically returned to threads "joining" the terminated thread (covered later).

Calling pthread_exit() from main():
- There is a definite problem if main() finishes before the threads it spawned if you don't call pthread_exit() explicitly. All of the threads it created will terminate because main() is done and no longer exists to support the threads.
- By having main() explicitly call pthread_exit() as the last thing it does, main() will block and be kept alive to support the threads it created until they are done.

Passing Arguments to Threads:
- The pthread_create() routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create() routine
- All arguments must be passed by reference and cast to (void *).

Example 1, creating and exiting threads (pthreads_ex1.c) :

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

void *thread_func(void *arg){
    printf("Hello World! this is a thread\n");
    pthread_exit(NULL);
}

int main (int argc, char *argv[]){
    printf("Hello World! this is main\n");
    pthread_t thread_handler;
    pthread_create(&thread_handler, NULL, thread_func, NULL);
    pthread_exit(NULL);
}
```

Example2, passing arguments to threads:

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

void *thread_func(void *arg){
      long int id = (long int) arg;
      printf("Hello World! this is a thread #%lu\n",id);
      pthread_exit(NULL);
}

int main (int argc, char *argv[]){
      long int id = 1;
      printf("Hello World! this is main\n");
      pthread_t thread_handler;
      pthread_create(&thread_handler, NULL, thread_func, (void*)id);
      pthread_exit(NULL);
}
```

Example 3, creating multiple threads and passing more arguments:

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 4

struct thread_data{
      int  id;
      int value;
};

void *thread_func(void *arg){
      struct thread_data *my_data;
      my_data = (struct thread_data*) arg;
      printf("Hello World! this is a thread #%d and value is: %d\n",
                                          my_data->id, my_data->value);
      pthread_exit(NULL);
}

int main (int argc, char *argv[]){
      int i = 0;
      struct thread_data data[NUM_THREADS];
      printf("Hello World! this is main\n");

      for(i = 0; i < NUM_THREADS; ++i){
            data[i].id = i;
            data[i].value = i*10;
      }

      pthread_t thread_handler[NUM_THREADS];

      for(i = 0; i < NUM_THREADS; ++i){
            pthread_create(&thread_handler[i], NULL,
                              thread_func, (void*)&data[i]);
      }
      pthread_exit(NULL);
}
```

You will find these examples at pingpong as files pthreads_ex1.c pthreads_ex2.c and pthreads_ex3.c
To compile the use: **gcc -pthread pthreads_ex1.c**

Study the examples and make sure you are comfortable with the code, feel free to play around and change anything you like.

Joining threads:
- "Joining" is one way to accomplish synchronization between threads.
- The pthread_join() function blocks the calling thread until the specified threadid thread terminates.

Example 4, Joining threads:

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 4

struct thread_data{
      long int  id;
      long int value;
};
void *thread_func(void *arg){
      struct thread_data *my_data;
      my_data = (struct thread_data*) arg;
      printf("Hello World! this is a thread #%ld and value is:
                                    %ld\n",my_data->id, my_data->value);
      pthread_exit((void*) my_data->value);
}
int main (int argc, char *argv[]){
      int i = 0;
      struct thread_data data[NUM_THREADS];

      printf("Hello World! this is main\n");

      for(i = 0; i < NUM_THREADS; ++i){
      data[i].id = i;
      data[i].value = i*10;
      }

      pthread_t thread_handler[NUM_THREADS];

      for(i = 0; i < NUM_THREADS; ++i){
            pthread_create(&thread_handler[i], NULL,
                        thread_func, (void*)&data[i]);
      }
      void *status;
      for(i = 0; i < NUM_THREADS; ++i){
      pthread_join(thread_handler[i], &status);
      printf("Main: completed join with thread %d having
                        a status of %ld\n",i,(long)status);
      }
      pthread_exit(NULL);
}
```

Study the join example pthreads_ex4.c as you did the others, compile and run it and study the outcome.

Task 1: Using Pthreads

Your first task is to parallelize a C program using pthreads, open file serial.c and study the code.

Compile with gcc serial.c -o serial
and run with ./serial [Array size] [Times to sum]
e.g: ./serial 100000 4 will sum an array of 100000 numbers 4 times.

The program you are going to parallelize is a simple program that accepts an array size and a number that specifies how many times it should sum the array.
1. Initializes an array of integers (init_array function)
2. Calculates the sum of the array a number of times (sum_array function) and
3. Checks if the sum was correct.
4. Prints the result.

Your task is to parallelize the part of the program that calculates the sum. The initialization and check should not be parallelized.

To parallelize the program you are going to use the functions we saw in the examples earlier. The steps you must follow are:
1. Start by adding a third argument to main for the number of threads.
2. Create a struct which will hold the for each thread a pointer to the array as well as the start and end index of the part of the array it should work on, you can optionally add a thread_id to help you debug.
2. write a thread_func which will be the start function of your threads, this function will produce the sum of the part of the array for each thread by calling sum_array(...) and pass the sum back to main though thread_exit().
3. write code to create threads, make sure you pass the proper arguments and initialize the accordingly.
4. join the threads in main and add up all the sums returned from the threads.
5. use check_sum(…) after all threads have finished to check if the sum was correct.

After you have successfully parallelized your program, pick an array size that runs for at least 10 seconds when calculating the sum 5 times (time ./serial you_array_size 5) in the serial version and calculate the speedup you get when using 1, 2, 4, 8 threads.
Make a graphs of the speedups for your report.
Use Amdahl's law to calculate what percentage of your program is serial and what is parallel and **include the answer in your report.**

What happens to the speedups for the same array size for 1,2,4,8 threads when you vary the number of times the sum must be calculated to 5, and 20?
Make a graphs of the speedups for your report.
Use Amdahl's law to calculate what percentage of your program is serial and what is parallel for every **case and include the answer in your report.**

Part 2: OpenMP

The purpose of this part is to acquaint you with the openMP API for automatic parallelization of programs. OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming. We are not going to go in depth to all the constructs that openMP provides. We are going to limit our experiments to parallel for loops.

To compile your programs with openMP you have to **#include <omp.h>** in your .c file and compile with **gcc -fopenmp file.c.**

To set the number of threads that you want openMP to use you should call
`omp_set_num_threads(NUM_THREADS);`

To parallelize a for loop with openMP all you have to do is use a **pragma** before the loop.
When a loop is parallelized with openMP the iterations of the loop are distributed to different threads to be executed in parallel. For example:

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

int main (int argc, char *argv[]){

      if(argc != 2){
            printf("usage: %s [num threads] \n",argv[0]);
            return 0;
      }

      int i,a = 0;
      int num_threads = atoi(argv[1]);
      printf("num threads: %d\n",num_threads);

      omp_set_num_threads(num_threads);

      #pragma omp parallel for
      for(i = 0; i < 1000; ++i){
            a++;
      }
      printf("a is %d\n",a);
}
```

The iterations of this for-loop will be executed in parallel.

You will find this code in file omp_ex1.c
compile and run it, what do you notice as you try different number of threads? Is the result the same?

When a variable is written in more than one loop iteration it is possible that the variable ends up with a corrupted value like in the example above. OpenMP provides special clauses for such cases.
Change the pragma in the previous program with:

```
#pragma omp parallel for reduction(+ : a)
```

The “reduction” clause is a safe way to join work from all threads. By using this we actually instruct the openMP runtime to allocate a different “a” variable to each thread and add the per-thread values to the original “a” variable in the end of the parallel for.

What is the output for one thread? What is the output for two threads?
Why do you think the values are different even if we have specified the reduction clause?

The answer is that although openMP can successfully parallelize the outer loop and assign a different “i” variable to each thread, it cannot do the same for the “j” variable of the inner loop. The reason that the outcome of the program is not correct is that all threads use and update the same “j”.
In order to have openMP create a different “j” variable for every thread we must add a “private” clause to the pragma like:

```
#pragma omp parallel for reduction(+ : a) private(j)
```

Run your program with the above pragma and see if the result is the same for one and two threads.

Task 2: Using OpenMP

parallelize the serial.c program using openMP. Parallelize the outer loop of the sum_array() function by using the right pragma, for which variables should you use private and for which should you use reduction?

Rememer:
- #include <omp.c>
- add a second argument to main for the number of threads.
- set the number of threads using: omp_set_num_threads(num_threads);
- compile with -fopenmp flag

Make sure your program runs correctly and calculate the speedup for 1,2,4, and 8 threads.
Make a graphs of the speedups for your report.
Compare the OpenMP implementation with pthreads performance-wise which one is faster, which took the most work to code?
Write your conclusions in your report.