ECSE 420 - Parallel Computing
Student: Daniel Macario
ID: 260503662

Assignment 1

1. (15%) Describe the following terms, their cause, and the work-around the industry has undertaken to overcome their consequences: (i) memory wall, (ii) frequency wall, and (iii) power wall.

Memory Wall:

The term memory wall was introduce in the 1980s to describe the growing disparity between CPU clock rates and off-chip memory and disk drive I/O (input/output) rates. From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory speed only improved at annual rate of 10%. Because of this slow growth, memory latency became a significant bottleneck in the performance of modern computers.

The industry has updated direct memory access (DMA) techniques to reduce I/O bottlenecks. A number of approaches have been used with different degrees of success; among these one can find the following: having application threads perform I/O independently of computation, and building multicore architectures with cores exclusively dedicate for I/O. There has also been significant research in creating new parallel programming models through projects like Map/Reduce, and Apache Hadoop.

Frequency Wall:

The term frequency wall refers to the challenges that result from further increasing the frequency of modern processors. The industry has reached a plateau were increases in frequency do not result in faster processors - this is because current architectures have instructions that take longer to execute than a single clock cycle of these high-frequency CPUs. In other words, increasing the frequency of CPUs results in no gains in speed if the longest operation of the architecture takes more time to execute than a single clock cycle.

There has been significant research done in how to tackle the challenges posed by the frequency wall. The industry has tried to divide long operations into various steps, however, this means redesigning existing architectures. A second approach to reducing the longest operations of a given architecture consists of reducing the physical size of the components in a processor - this results in electrical impulses having to travel shorter distances, shorter transistor switch times, and other hardware implications which can make operations speed up uniformly, thus overcoming the frequency wall.

Power Wall:

For many decades, the design goal behind CPUs centred on increasing their operating frequency. In order to do this, transistors were continuously made smaller so that more could fit on a chip. The problem with this design choice lies in the fact that as transistors have approached atomic sizes, their gates have become so thin that electrons leak from them. This produces heat, which current consumer-level cooling technologies cannot dissipate effectively.

How to build faster processors that can operate reliably when faced with these issues is the challenge presented by the power wall.

The industry has worked around this problem by developing transistors with materials that reduce the leakage - these materials are know as high-k, for high dielectric constant. Another advance consists of adjusting the internal voltage and frequency of different sections of the logic depending on the operation being performed. This is known as dynamic voltage scaling and dynamic frequency scaling, respectively. Finally, another work-around consists in developing better and more cost-effective cooling technologies.

2. (40%) Suppose we have a program that can be divided into 5 tasks which do not depend on each other. 4 of these tasks take T seconds to run, and one task takes 2T seconds to run. We redesign this program to run on a multicore computer using threads.

(a) What is the speedup obtained by running the 5 tasks as 5 threads on 5 separate cores? (Assume there is no OS overhead, and all 5 threads start at the same time.)

$T_{single\_core} = T + T + T + T + 2T = 6T$

$T_{five\_cores} = 2T$

Speedup = $T_{single\_core} / T_{five\_cores} = 6T / 2T = 3$

(b) What is the maximum speedup obtained when there are 5 threads but only 4 cores? (Assume there is no OS overhead, and 4 of the threads start at the same time.)

The maximum speedup is obtained by executing the task that takes Depends on whether the thread that takes 2T to execute is in the first batch of 4 threads, or if it is in the second one.

First Batch

$T_{four\_cores} = 2T$

Speedup = $6T / 2T = 4$

Second Batch

$T_{four\_cores} = 3T$

Speedup = $6T / 3T = 2$

(c) Now assume that one of the 5 threads is a "master thread" which spawns the other threads. The master thread requires X seconds of OS overhead to spawn a worker thread. After those X seconds, the spawned thread begins to run. Once the master thread is finished spawning the 4 other threads, it runs the final task. What is the maximum speedup as a function of X and T? Make a diagram of the sequence of threads to back up your answer.

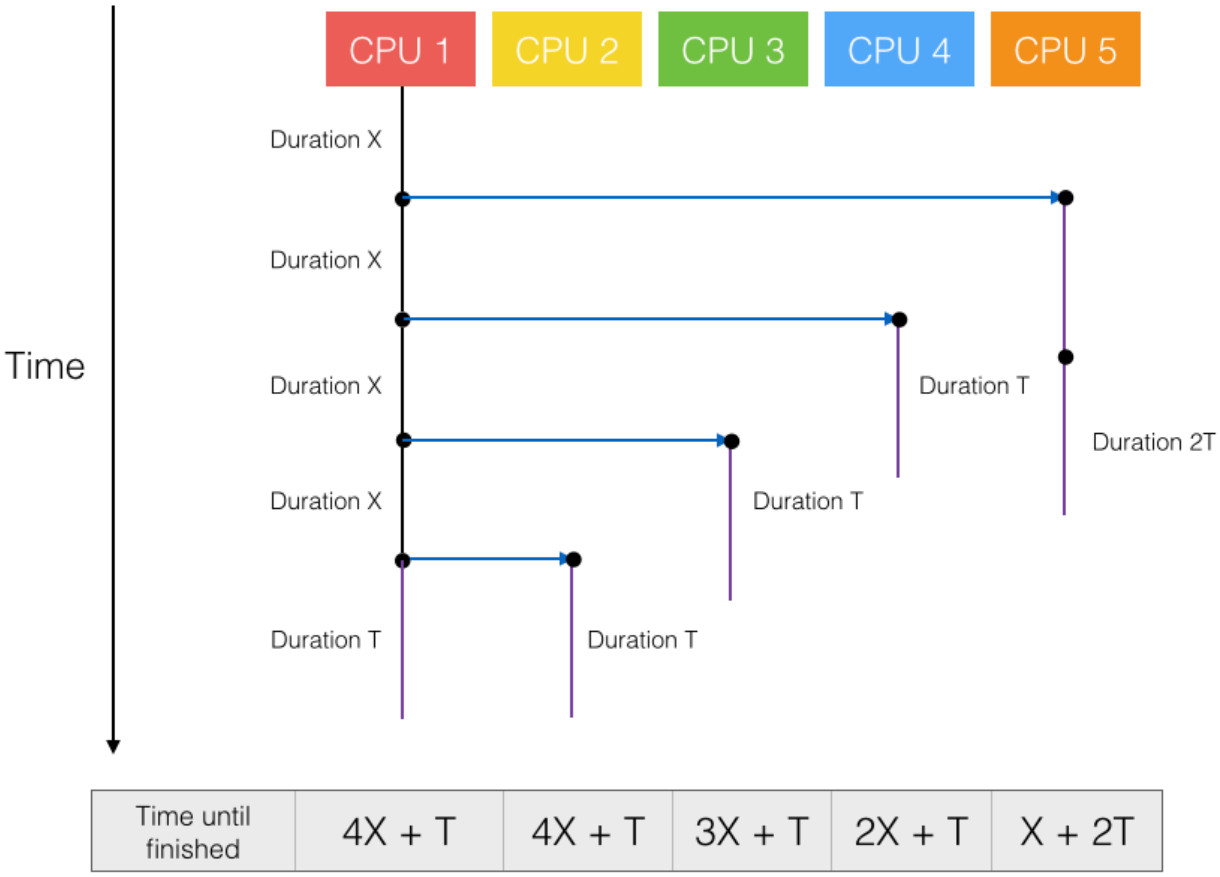

Diagram 1. Thread diagram

The thread sequence demonstrated in diagram 1 indicates the optimal execution of the 5 threads. Note that the master thread first spans a thread to execute the longest task (the one that has an execution time of 2T) on CPU 5. Starting work on this task at the beginning prevents having it become a bottleneck. There are two possible solutions to the problem depending on the values of X and T, below we present both of them.

Case 1:

$4X + T > X + 2T$

$T < 3X$

$T_{T<3X} = 4X + T$

Speedup = $6T / (4X + T)$

Case 2:

$4X + T < X + 2T$

$T > 3X$

$T_{T>3X} = X + 2T$

Speedup = $6T / (X + 2T)$

3. (20%) Write a C program using Pthreads which takes as command line input an integer N. The program should spawn N threads and report the order in which the threads finish their "task" (the task is to wait a random number of microseconds). For example, for N = 3, if thread 0, thread 1, and thread 2 finish their task in the order {2,0,1}, the program should print "2, 0, 1".

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

static sem_t thread_finish_count_semaphore;
int thread_finished_count = 0;
int num_threads;

// Struct used to associate threads with an ID corresponding
// to the poisition in the sequence of creation.
typedef struct PositionData {
    int position;
} PositionData;

// Sleep the thread for a random number between 0 and 10000 us
void sleepMicroseconds(int us) {
    // usleep sleeps for a specified amount of microseconds.
    if (usleep(us) == -1) {
        printf("Error sleeping thread\n");
        exit(2);
    }
}

// Executing thread sleeps a random number of miliseconds and prints finishing position
void *thread_function( void *arg ) {
    PositionData *pos = (PositionData *) arg;
    int position = pos->position;
    sleepMicroseconds(rand() % 10001);

    // The semaphore is used for the sole purpose of eliminating the comma
    // at the end of the output string. E.g instead of getting "0,1,2," the program
    // outputs "0,1,2"
    if (sem_wait(&thread_finish_count_semaphore) == -1) exit(2);
```

```
    thread_finished_count += 1;
    printf("%d", position);
    if (thread_finished_count != num_threads) {
        printf(",");
    }
    if (sem_post(&thread_finish_count_semaphore) == -1) exit(2);
}

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

    // Function call to be able to generate random numbers
    srand(time(NULL));

    if (argc == 1) {
        printf("Please specify the number of threads as a parameter to the program\n");
        exit(1);
    }

    num_threads = atoi(argv[1]);
    pthread_t threads[num_threads];

    if (sem_init(&thread_finish_count_semaphore, 0, 1) == -1) {
        printf("Error, could initialize semaphore\n");
        exit(1);
    }

    // Span the number of threads specified by the user
    PositionData *posData;
    int i;
    for (i = 0; i < num_threads; i++) {
        posData = malloc(sizeof(PositionData));
        (*posData).position = i;
        rc = pthread_create(&threads[i], NULL, thread_function, (void*) posData);
        if (rc != 0) {
            printf("Error creating threads\n");
            exit(1);
        }
    }

    for (i = 0; i < num_threads; i++) {
        rc = pthread_join(threads[i], NULL);
        if (rc != 0) {
            printf("Error joining threads\n");
            exit(1);
        }
    }

    printf("\n");
```

```
    exit(0);
}
```

Sample usage:

```
$ gcc program.c -pthreads
$ ./a.out 4
$ 4,3,1,2
```

4. (25%) A uniprocessor application is parallelized for 4 processors, yielding a 3.8× speedup. Given the time breakdown of the various functions seen in the graph, what is the minimum total time that the uniprocessor application spends while busy and in performing data access?

First we calculate the execution time of each processor by adding their corresponding time of synchronization, busy useful time, data local time, data remote time, and busy overhead time.

T_{P0} = 145 ms

T_{P1} = 160 ms

T_{P2} = 145 ms

T_{P3} = 160 ms

Then we consider the expression for calculating the speedup that results from parallelizing a serial application:

Speedup = T_{serial} / $T_{parallel}$

$T_{parallel}$ corresponds to the time of the processor that had the longest execution time. We can represent this expression as Max(145, 160, 145, 160). Finally, we plug these values in the speedup formula to calculate the minimum total time the uniprocessor application spent while busy and in performing data access:

3.8 <= T_{serial} / Max(145, 160, 145, 160)

T_{serial} >= 3.8 x 160

T_{serial} >= 608 ms