CS 3410 Fall 2025

Lab 12 Worksheet: Parallelism

1. Amdhal's Law

Part 1A

Suppose we have a program such that 20% of its original runtime is taken up by unparallelizable (or sequential) portions of the code. What is the theoretical maximum speedup we could obtain in the situations where we have 2, 4, 8, or 50 cores? You should have a different answer for each number of cores.

Part 1B

Explain why adding more cores results in diminishing speedup.

Part 1C

In the same situation as above, what is the theoretical maximum speedup even if we have infinite cores/workers?

2. Circular Buffer

Part 2A

*In lecture, you were shown how to implement the **push** function in a circular buffer. We will now implement the **pop** function to allow consumers to retrieve data from the producer.*

```
typedef struct {
    int* data;
    int capacity;   // length of data[]
    int head;       // index of next item to pop
    int tail;       // index of next free slot to push

    // used only in thread-safe versions:
    pthread_mutex_t* mutex;
    pthread_cond_t*  full_cv;   // signaled when there is space
    pthread_cond_t*  empty_cv;  // signaled when there is data
} bounded_buffer_t;
```

CS 3410 Fall 2025

Lab 12 Worksheet: Parallelism

```
bool bb_empty(bounded_buffer_t* bb); // returns true iff buffer is empty
bool bb_full(bounded_buffer_t* bb);  // returns true iff buffer is full
```

*Fill in the blanks to implement an unprotected version of **bb_pop**.*

```
int bb_pop(bounded_buffer_t* bb) {
    assert( ! ________(bb) );

    int value = bb->data[ ________ ];

    bb->head = ( ________ + 1 ) % ________;

    return value;
}
```

Part 2B

*Now implement **bb_block_pop_busy** which makes **bb_pop** thread safe through only the use of mutexes (not conditional variables). Hint: It is okay for your implementation to use busy waiting.*

```
int bb_block_pop_busy(bounded_buffer_t* bb) {
    pthread_mutex_lock( ________ );

    while ( ________(bb) ) {

    }

    int value = ________( ________ );

    pthread_mutex_unlock( ________ );

    return value;
}
```

Part 2C

*Now, implement **bb_block_pop,** which removes busy waiting by using condition variables. Make sure that your implementation correctly signals to producers that there is now empty space in the buffer after you execute the pop.*

```
int bb_block_pop(bounded_buffer_t* bb) {
    pthread_mutex_lock( ________ );

    while ( bb_empty(bb) ) {
        pthread_cond_wait( ________ , ________ );
    }

    int value = ________( ________ );

    pthread_mutex_unlock( ________ );

    pthread_cond_signal( ________ );
```

Lab 12 Worksheet: Parallelism

```
    return value;
}
```

2. MatVec

Part 3A

*Recall from linear algebra the operation of matrix-vector multiplication. Namely, for a vector **x** of dimension **N_COLS** and a matrix **A** of dimension **N_ROWS** x **N_COLS**, we can say **y = Ax** where **y** is a vector of dimension **N_ROWS**. Importantly, **y[i] = A[i] · x**, where **y[i]** is the ith entry of the result vector y and **A[i]** is the ith row of A (**A[i] · x** here is the dot product between **A[i]** and **x**). We can use this fact to parallelize matrix-vector multiplication. We give an outline of the code to execute this below.*

```c
#define N_ROWS 4
#define N_COLS 3

// Global matrix A and vector x
double A[N_ROWS][N_COLS] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
    {1, 0, 1}
};

double x[N_COLS] = {1, 2, 1};

// Result vector y = A * x
double y[N_ROWS];

// Thread argument type (to be completed in part (a))
typedef struct {
    /* blanks in part (a) */
} row_task_t;

// Worker function (to be completed in part (b))
void* row_worker(void* arg);

// main will create N_ROWS threads, one per row (part (c))
int main(void) {
    /* blanks in part (c) */
}
```

Fill in the blanks below. Assume each thread has access to all the global variables above.

```c
// Each thread needs to know which row of A to multiply with x.
typedef struct {
    _________;        // (i) what does each thread need to know?
} row_task_t;

// We will use one thread per row.
pthread_t threads[_________];   // (ii) number of threads
row_task_t tasks[_________];    // (iii) one task per thread
```

CS 3410 Fall 2025

Lab 12 Worksheet: Parallelism

Part 3B

Now implement the task that each worker will perform.

```
// Each thread computes one entry y[row] = sum_c A[row][c] * x[c].
void* row_worker(void* arg) {
    // (i) get typed pointer to our task
    row_task_t* task = _____________;

    // (ii) which row do we compute?
    int r = _____________;

    // (iii) compute the dot product
    double sum = 0.0;
    for (int c = 0; c < _____________ ; ++c) {
        sum += _____________ * _____________;
    }

    // (iv) write result
    _____________ = sum;

    return NULL;
}
```

Part 3C

Let's put it all together in our main function.

```
int main(void) {
    // 1. Create tasks and spawn one thread per row.
    for (int r = 0; r < N_ROWS; ++r) {
        // (i) initialize task for row r
        tasks[r].row = _____________;

        // (ii) create thread r
        pthread_create(&threads[r],
                       NULL,
                       _____________,
                       _____________);
    }

    // 2. Wait for all threads to finish.
    for (int r = 0; r < N_ROWS; ++r) {
        _____________(threads[r], NULL);
    }

    return 0;
}
```