



RAMAIAH
Institute of Technology

***DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING (CYBER
SECURITY)***

LAB REPORT

***for the course
Multicore Architecture and Programming***

CY71

***for the academic year
2025-2026***

***Submitted by:
Rachit N A - IMS22CY056***

Certificate

This is to certify that ***Rachit N A(1MS22CY056)*** has satisfactorily completed the course of experiments in the practical of ***Multicore Architecture and Programming (CY71)*** prescribed by the ***Department of CSE (Cyber Security)***, Ramaiah Institute of Technology for the year ***2025-2026***.

<i>Component</i>	<i>Max.Marks</i>	<i>Marks Obtained</i>
<i>Record</i>		
<i>Lab Test</i>		
<i>Total Marks</i>		

Signature of Student with Date

Signature of Faculty with Date

Program 1: Basic Multi-Core Matrix Addition Implement parallel processing to calculate the sum of a 2D array.

Code: #include <stdio.h>

#include <pthread.h>

#define SIZE 3 int A[SIZE][SIZE] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

int B[SIZE][SIZE] = { {9, 8, 7}, {6, 5, 4}, {3, 2, 1} }; int C[SIZE][SIZE];

void *add_matrices(void* arg){ int i = *(int*)arg; for(int j = 0; j < SIZE; j++)

C[i][j] = A[i][j] + B[i][j];

return NULL;}

int main(){

pthread_t threads[SIZE];

int indices[SIZE];

for(int i = 0; i < SIZE; i++) {

indices[i] = i;

pthread_create(&threads[i], NULL, add_matrices, (void*)&indices[i]);

for(int i = 0; i < SIZE; i++)

pthread_join(threads[i], NULL);

printf("Resultant matrix:\n"); }

for(int i = 0; i < SIZE; i++) {

for(int j = 0; j < SIZE; j++)

printf("%d ", C[i][j]);

printf("\n"); }

return 0;}

Output:

```
rit@rit:~/rachit_multicore$ vi array_sum.c
rit@rit:~/rachit_multicore$ mpicc -o array_sum array_sum.c
rit@rit:~/rachit_multicore$ mpirun -np 5 ./array_sum
Resultant matrix:
10 10 10
10 10 10
10 10 10
Resultant matrix:
10 10 10
10 10 10
10 10 10
Resultant matrix:
10 10 10
10 10 10
10 10 10
Resultant matrix:
10 10 10
10 10 10
10 10 10
Resultant matrix:
10 10 10
10 10 10
10 10 10
```

Program 2: Simple Multi-Core Factorial Calculation

Implement parallel processing to calculate the factorials of large numbers.

Code:

```
#include <stdio.h>

#include <pthread.h>

#define NUM_THREADS 4

long long results[NUM_THREADS];

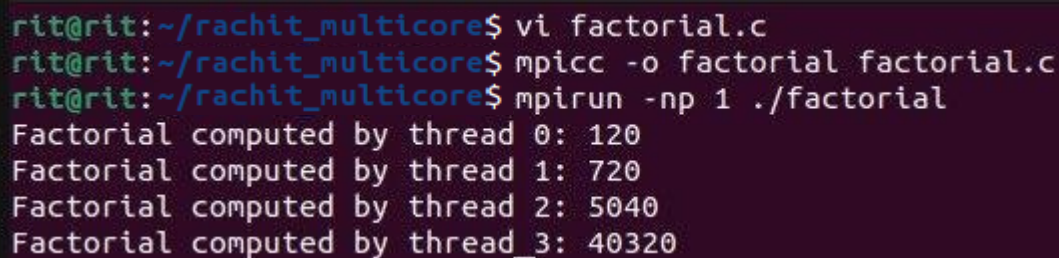
void* factorial(void* arg) {

    int thread_id = *(int*)arg;

    long long fact = 1;
```

```
for (int i = 1; i <= 5 + thread_id; i++) {  
    fact *= i; }  
results[thread_id] = fact;  
return NULL;}  
  
int main() {  
    pthread_t threads[NUM_THREADS];  
    int thread_ids[NUM_THREADS];  
    for (int i = 0; i < NUM_THREADS; i++) {  
        thread_ids[i] = i;  
        pthread_create(&threads[i], NULL, factorial, (void*)&thread_ids[i]); }  
    for (int i = 0; i < NUM_THREADS; i++) {  
        pthread_join(threads[i], NULL);  
        printf("Factorial computed by thread %d: %lld\n", i, results[i]); }  
    return 0;}
```

Output:



```
rit@rit:~/rachit_multicore$ vi factorial.c  
rit@rit:~/rachit_multicore$ mpicc -o factorial factorial.c  
rit@rit:~/rachit_multicore$ mpirun -np 1 ./factorial  
Factorial computed by thread 0: 120  
Factorial computed by thread 1: 720  
Factorial computed by thread 2: 5040  
Factorial computed by thread 3: 40320
```

Program 3: Parallel Search in an Array

Implement multithreading to simulate searching large datasets, such as user records in a database.

Code:

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

```
#define SIZE 100

int data[SIZE];

int found_index = -1;

pthread_mutex_t lock;

void* search(void* arg) {

    int start = *(int*)arg;

    for (int i = start; i < start + SIZE / 4; i++) {

        if (data[i] == 50) {

            pthread_mutex_lock(&lock);

            found_index = i;

            pthread_mutex_unlock(&lock);

            return NULL; } }

    return NULL;}

int main() {

    pthread_t threads[4];

    int thread_ids[4];

    for (int i = 0; i < SIZE; i++) {

        data[i] = i + 1; }

    pthread_mutex_init(&lock, NULL);

    for (int i = 0; i < 4; i++) {

        thread_ids[i] = i * (SIZE / 4);

        pthread_create(&threads[i], NULL, search, (void*)&thread_ids[i]); }

    for (int i = 0; i < 4; i++) {

        pthread_join(threads[i], NULL); }

    if (found_index != -1) {

        printf("Number 50 found at index: %d\n", found_index);

    } else {
```

```
printf("Number 50 not found.\n"); }  
  
pthread_mutex_destroy(&lock);  
  
return 0;}
```

Output:



```
rit@rit: ~/rachit_multicore$ vi bank.c  
rit@rit: ~/rachit_multicore$ mpicc -o bank bank.c  
rit@rit: ~/rachit_multicore$ mpirun -np 5 ./bank  
Number 50 found at index: 49  
Number 50 found at index: 49  
Number 50 found at index: 49  
Number 50 found at index: 49  
Number 50 found at index: 49  
rit@rit: ~/rachit_multicore$ mpirun -np 1 ./bank  
Number 50 found at index: 49
```

Program 4: Parallel Sorting with Merge Sort

Implement multithreading to simulate sorting large datasets in common applications, such as database management systems.

Code: #include <stdio.h>

<pthread.h> #include

#include <stdlib.h> #define SIZE 100

int array[SIZE]; int temp[SIZE]; void

merge(int left, int mid, int right) {

int i = left, j = mid + 1, k = left;

while (i <= mid && j <= right) {

if (array[i] < array[j]) {

temp[k++] = array[i++];

```
        } else {  
            temp[k++] = array[j++]; } }  
while (i <= mid) {  
    temp[k++] = array[i++]; }  
while (j <= right) {  
    temp[k++] = array[j++]; }  
for (i = left; i <= right; i++) {  
    array[i] = temp[i]; } }  
void merge_sort_range(int left, int right) {  
    if (left < right) {  
        int mid = (left + right) / 2;  
        merge_sort_range(left, mid);  
        merge_sort_range(mid + 1, right);  
        merge(left, mid, right); } }  
void* merge_sort(void* arg) {  
    int left = *(int*)arg;  
    int right = left + (SIZE / 4) - 1;  
    merge_sort_range(left, right);  
    return NULL; }  
int main() {  
    pthread_t threads[4];  
    int thread_ids[4];  
    for (int i = 0; i < SIZE; i++) {  
        array[i] = rand() % 1000; }  
    for (int i = 0; i < 4; i++) {  
        thread_ids[i] = i * (SIZE / 4);  
        pthread_create(&threads[i], NULL, merge_sort, (void*)&thread_ids[i]);}
```



```

for(int i = 0; i < 4; i++) {

    pthread_join(threads[i], NULL); }

intquarter = SIZE / 4;

merge(0, quarter - 1, 2 * quarter - 1);

merge(2 * quarter, 3 * quarter - 1, SIZE - 1);

merge(0, 2 * quarter - 1, SIZE - 1);

printf("Sorted Array:\n");

for(int i = 0; i < SIZE; i++) {

    printf("%d ", array[i]); }

printf("\n");

return 0; }

```

Output:

```

rit@rit:~/rachit_multicore$ vi sort.c
rit@rit:~/rachit_multicore$ mpicc -o sort sort.c
rit@rit:~/rachit_multicore$ mpirun -np 5 ./sort
Sorted Array:
11 12 22 27 42 43 58 59 60 67 69 84 87 91 94 123 124 135 167 170 172 178 198 211 226 229 276 281 305 313 315 324 327 335 336 362 364 367 368 370
373 383 386 393 399 403 413 421 421 426 429 434 456 492 505 526 530 537 539 540 545 567 582 584 586 649 651 676 690 729 736 739 750 754 763 777 782 7
84 788 793 802 808 814 846 857 862 862 873 886 895 915 919 925 926 929 932 956 980 996
Sorted Array:
11 12 22 27 42 43 58 59 60 67 69 84 87 91 94 123 124 135 167 170 172 178 198 211 226 229 276 281 305 313 315 324 327 335 336 362 364 367 368 370
373 383 386 393 399 403 413 421 421 426 429 434 456 492 505 526 530 537 539 540 545 567 582 584 586 649 651 676 690 729 736 739 750 754 763 777 782 7
84 788 793 802 808 814 846 857 862 862 873 886 895 915 919 925 926 929 932 956 980 996
Sorted Array:
11 12 22 27 42 43 58 59 60 67 69 84 87 91 94 123 124 135 167 170 172 178 198 211 226 229 276 281 305 313 315 324 327 335 336 362 364 367 368 370
373 383 386 393 399 403 413 421 421 426 429 434 456 492 505 526 530 537 539 540 545 567 582 584 586 649 651 676 690 729 736 739 750 754 763 777 782 7
84 788 793 802 808 814 846 857 862 862 873 886 895 915 919 925 926 929 932 956 980 996
Sorted Array:
11 12 22 27 42 43 58 59 60 67 69 84 87 91 94 123 124 135 167 170 172 178 198 211 226 229 276 281 305 313 315 324 327 335 336 362 364 367 368 370
373 383 386 393 399 403 413 421 421 426 429 434 456 492 505 526 530 537 539 540 545 567 582 584 586 649 651 676 690 729 736 739 750 754 763 777 782 7
84 788 793 802 808 814 846 857 862 862 873 886 895 915 919 925 926 929 932 956 980 996

```

Program 5: Mutex for Synchronization

Implement multi-threading to simulate concurrent bank transactions using mutex locks.

Code:

```

#include <stdio.h>

#include <pthread.h>

#define NUM_THREADS 5

```

```
int account_balance = 1000;

pthread_mutex_t lock;

void* perform_transaction(void* arg) {

    int amount = *((int*)arg);

    pthread_mutex_lock(&lock);

    if (account_balance + amount >= 0) {

        account_balance += amount;

        printf("Transaction successful. New balance: %d\n", account_balance);

    } else {

        printf("Transaction denied. Insufficient funds.\n"); }

    pthread_mutex_unlock(&lock);

    return NULL; }

int main() {

    pthread_t threads[NUM_THREADS];

    int transactions[NUM_THREADS] = {-200, 100, -300, 150, -400};

    pthread_mutex_init(&lock, NULL);

    for (int i = 0; i < NUM_THREADS; i++) {

        pthread_create(&threads[i], NULL, perform_transaction, (void*)&transactions[i]); }

    for (int i = 0; i < NUM_THREADS; i++) {

        pthread_join(threads[i], NULL); }

    printf("Final account balance: %d\n", account_balance);

    pthread_mutex_destroy(&lock);

    return 0; }
```

Output:

```
rit@rit:~/rachit_multicore$ vi mutex.c
rit@rit:~/rachit_multicore$ mpicc -o mutex mutex.c
rit@rit:~/rachit_multicore$ mpirun -np 5 ./mutex
Transaction successful. New balance: 800
Transaction successful. New balance: 950
Transaction successful. New balance: 1050
Transaction successful. New balance: 650
Transaction successful. New balance: 350
Final account balance: 350
Transaction successful. New balance: 800
Transaction successful. New balance: 900
Transaction successful. New balance: 600
Transaction successful. New balance: 750
Transaction successful. New balance: 350
Final account balance: 350
Transaction successful. New balance: 800
Transaction successful. New balance: 900
Transaction successful. New balance: 600
Transaction successful. New balance: 750
Transaction successful. New balance: 350
Final account balance: 350
Transaction successful. New balance: 800
Transaction successful. New balance: 900
Transaction successful. New balance: 600
Transaction successful. New balance: 750
Transaction successful. New balance: 350
Final account balance: 350
Transaction successful. New balance: 800
Transaction successful. New balance: 900
Transaction successful. New balance: 600
Transaction successful. New balance: 750
Transaction successful. New balance: 350
Final account balance: 350
```

Program 6: Condition Variables for Thread Communication

Implement a producer-consumer model using threads for task management.

Here, we use the concept of a bakery, where bakers (producers) need to signal when bread is ready for delivery (consumers).

Code:

```
#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define MAX_BREAD 10

int bread_count = 0;

pthread_mutex_t lock;

pthread_cond_t cond;

void* producer(void* arg) {

    while (1) {

        pthread_mutex_lock(&lock);

        while (bread_count >= MAX_BREAD) {
```

```
        pthread_cond_wait(&cond, &lock); }

    bread_count++;

    printf("Produced bread. Total: %d\n", bread_count);

    pthread_cond_signal(&cond);

    pthread_mutex_unlock(&lock);

    sleep(1); }

return NULL;}

void* consumer(void* arg) {

    while (1) {

        pthread_mutex_lock(&lock);

        while (bread_count <= 0) {

            pthread_cond_wait(&cond, &lock); }

        bread_count--;

        printf("Consumed bread. Total: %d\n", bread_count);

        pthread_cond_signal(&cond);

        pthread_mutex_unlock(&lock);

        sleep(2); }

    return NULL;}

int main() {

    pthread_t prod_thread, cons_thread;

    pthread_mutex_init(&lock, NULL);

    pthread_cond_init(&cond, NULL);

    pthread_create(&prod_thread, NULL, producer, NULL);

    pthread_create(&cons_thread, NULL, consumer, NULL);

    pthread_join(prod_thread, NULL);

    pthread_join(cons_thread, NULL);

    pthread_mutex_destroy(&lock);
```

```
pthread_cond_destroy(&cond);  
  
return 0;}
```

Output:

```
rit@rit:~/rachit_multicore$ vi producer_consumer.c  
rit@rit:~/rachit_multicore$ mpicc -o producer_consumer producer_consumer.c  
rit@rit:~/rachit_multicore$ mpirun -np 1 ./producer_consumer  
Produced bread. Total: 1  
Consumed bread. Total: 0  
Produced bread. Total: 1  
Consumed bread. Total: 0  
Produced bread. Total: 1  
Produced bread. Total: 2  
Consumed bread. Total: 1  
Produced bread. Total: 2  
Produced bread. Total: 3  
Consumed bread. Total: 2  
Produced bread. Total: 3  
Produced bread. Total: 4  
Consumed bread. Total: 3  
Produced bread. Total: 4  
Produced bread. Total: 5  
Consumed bread. Total: 4  
Produced bread. Total: 5  
Produced bread. Total: 6  
Consumed bread. Total: 5  
Produced bread. Total: 6  
Produced bread. Total: 7  
Consumed bread. Total: 6  
Produced bread. Total: 7  
Produced bread. Total: 8  
Consumed bread. Total: 7  
Produced bread. Total: 8  
Produced bread. Total: 9  
Consumed bread. Total: 8  
Produced bread. Total: 9  
Produced bread. Total: 10  
Consumed bread. Total: 9  
Produced bread. Total: 10  
Consumed bread. Total: 9  
Produced bread. Total: 10
```


Program 7: OpenMP Parallel Loop Implement parallel array

summation using OpenMP for performance optimisation.

Code:

```
#include <stdio.h> #include <omp.h> #include <stdlib.h> #define SIZE
1000000 int array[SIZE]; int main() {
for (int i = 0; i < SIZE; i++) {
array[i] = rand() % 100;}
#pragma omp parallel{
    if (omp_get_thread_num() == 0) {
        printf("Running with %d threads\n", omp_get_num_threads()); }}
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < SIZE; i++) {
sum += array[i];}
printf("Total Sum: %d\n", sum);
return 0;}
```

Output:

```
rit@rit:~/rachit_multicore$ vi parallel_loop.c
rit@rit:~/rachit_multicore$ gcc -fopenmp parallel_loop.c -o parallel_loop
rit@rit:~/rachit_multicore$ ./parallel_loop
Running with 16 threads
Total Sum: 49498583
```



```
rit@rit:~/rachit_multicore$ mpicc -fopenmp parallel_loop.c -o parallel_loop
rit@rit:~/rachit_multicore$ mpirun -np 4 ./parallel_loop
Running with 16 threads
Total Sum: 49498583
Running with 16 threads
Total Sum: 49498583
Running with 16 threads
Total Sum: 49498583
Running with 16 threads
Total Sum: 49498583
```

Program 8: OpenMP Task Parallelism

Implement task parallelism in image processing applications using OpenMP.

Code:

```
#include <stdio.h>

#include <omp.h>

#define NUM_TASKS 4

void blur() {

    printf("Blurring the image...\n");}

void sharpen() {

    printf("Sharpening the image...\n");}

void contrast() {

    printf("Adjusting contrast...\n");}

void resize() {

    printf("Resizing the image...\n");}

int main() {

    #pragma omp parallel {

        #pragma omp single {

            int num_threads = omp_get_num_threads();

            printf("Running with %d OpenMP threads\n", num_threads);

            #pragma omp task
```

```
        blur();  
  
        #pragma omp task  
  
        sharpen();  
  
        #pragma omp task  
  
        contrast();  
  
        #pragma omp task  
  
        resize(); } }  
  
return 0; }
```

Output:

```
rit@rit:~/rachit_multicore$ gcc -fopenmp image_processing.c -o image_processing  
rit@rit:~/rachit_multicore$ ./image_processing  
Running with 16 OpenMP threads  
Blurring the image...  
Adjusting contrast...  
Resizing the image...  
Sharpening the image...
```

```
rit@rit:~/rachit_multicore$ vi image_processing.c  
rit@rit:~/rachit_multicore$ mpicc -fopenmp image_processing.c -o image_processing  
rit@rit:~/rachit_multicore$ mpirun -np 4 ./image_processing  
Running with 16 OpenMP threads  
Blurring the image...  
Sharpening the image...  
Resizing the image...  
Adjusting contrast...  
Running with 16 OpenMP threads  
Blurring the image...  
Sharpening the image...  
Adjusting contrast...  
Resizing the image...  
Running with 16 OpenMP threads  
Blurring the image...  
Sharpening the image...  
Adjusting contrast...  
Resizing the image...  
Running with 16 OpenMP threads  
Blurring the image...  
Sharpening the image...  
Adjusting contrast...  
Resizing the image...
```


Program 9: Handling Deadlocks with Timeouts Implement timeout mechanisms to handle deadlocks effectively.

Code: #include <stdio.h> #include <pthread.h> #include <unistd.h>

```
#define NUM_THREADS 2 pthread_mutex_t lock1;

pthread_mutex_t lock2;

void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    if (thread_id == 0) {

        pthread_mutex_lock(&lock1);
        sleep(1);
        printf("Thread 0: Waiting for lock 2...\n");
        if (pthread_mutex_trylock(&lock2) != 0) {

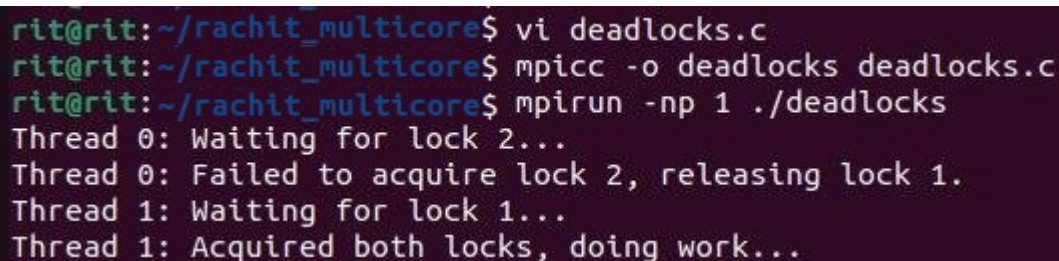
            printf("Thread 0: Failed to acquire lock 2, releasing lock 1.\n");
            pthread_mutex_unlock(&lock1);
        } else {

            printf("Thread 0: Acquired both locks, doing work...\n");
            sleep(1);
            pthread_mutex_unlock(&lock2);
            pthread_mutex_unlock(&lock1); }
    } else {

        pthread_mutex_lock(&lock2);
        sleep(1);
        printf("Thread 1: Waiting for lock 1...\n");
```

```
    if(pthread_mutex_trylock(&lock1) != 0) {  
        printf("Thread 1: Failed to acquire lock 1, releasing lock 2.\n");  
        pthread_mutex_unlock(&lock2);  
    } else {  
        printf("Thread 1: Acquired both locks, doing work...\n");  
        sleep(1);  
        pthread_mutex_unlock(&lock1);  
        pthread_mutex_unlock(&lock2); } }  
    return NULL;}  
  
int main() {  
    pthread_t threads[NUM_THREADS];  
    int thread_ids[NUM_THREADS] = {0, 1};  
    pthread_mutex_init(&lock1, NULL);  
    pthread_mutex_init(&lock2, NULL);  
    for(int i=0; i < NUM_THREADS; i++) {  
        pthread_create(&threads[i], NULL, thread_function, (void*)&thread_ids[i]); }  
    for(int i=0; i < NUM_THREADS; i++) {  
        pthread_join(threads[i], NULL); }  
    pthread_mutex_destroy(&lock1);  
    pthread_mutex_destroy(&lock2);  
    return 0;}
```

Output:



```
rit@rit:~/rachit_multicore$ vi deadlocks.c  
rit@rit:~/rachit_multicore$ mpicc -o deadlocks deadlocks.c  
rit@rit:~/rachit_multicore$ mpirun -np 1 ./deadlocks  
Thread 0: Waiting for lock 2...  
Thread 0: Failed to acquire lock 2, releasing lock 1.  
Thread 1: Waiting for lock 1...  
Thread 1: Acquired both locks, doing work...
```

Program 10: Work-Stealing Scheduler

Implement a work-stealing scheduler to optimize task processing among threads.

Code:

```
#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#define NUM_TASKS 10

#define NUM_THREADS 4

int tasks[NUM_TASKS];

void* process_tasks(void* arg) {

    int thread_id = *(int*)arg;

    for (int i = thread_id; i < NUM_TASKS; i += NUM_THREADS) {

        printf("Thread %d processing task %d\n", thread_id, tasks[i]);

        sleep(1);}

    return NULL;}

int main() {

    pthread_t threads[NUM_THREADS];

    int thread_ids[NUM_THREADS];

    for (int i = 0; i < NUM_TASKS; i++) {

        tasks[i] = i + 1;}

    for (int i = 0; i < NUM_THREADS; i++) {

        thread_ids[i] = i;

        pthread_create(&threads[i], NULL, process_tasks, (void*)&thread_ids[i]);}

    for (int i = 0; i < NUM_THREADS; i++) {

        pthread_join(threads[i], NULL);}

    return 0;}
```



Output:

```
rit@rit:~/rachit_multicore$ vi work_stealer.c
rit@rit:~/rachit_multicore$ mpicc -o work_stealer work_stealer.c
rit@rit:~/rachit_multicore$ mpirun -np 1 ./work_stealer
Thread 0 processing task 1
Thread 1 processing task 2
Thread 3 processing task 4
Thread 2 processing task 3
Thread 1 processing task 6
Thread 0 processing task 5
Thread 2 processing task 7
Thread 3 processing task 8
Thread 1 processing task 10
Thread 0 processing task 9
```