Padc lab

```
#include<stdio.h>
#include<omp.h>
void main(){
printf("1.Parallel Section (hello world)\n");
#pragma omp parallel num_threads(4)
   {
      int t_id = omp_get_thread_num();
      printf("Hello World from thread %d\n", t_id);
   }

printf("\n2.Single Section\n");
#pragma omp parallel num_threads(4)
{
   #pragma omp single
   {
      printf("This is the single section executed by the thread
%d\n",omp_get_thread_num());
   }
}
printf("\n3.Master Construct\n");
#pragma omp parallel
{
   #pragma omp master
   {
      printf("This section is executed by the master thread only(Thread:%d)
\n",omp_get_thread_num());
   }
}
printf("\n4.Critical Section\n");
#pragma omp parallel num_threads(4)
{
   #pragma omp critical
   {
      printf("Currently, Thread %d is in the critical
section\n",omp_get_thread_num());
   }
}
int n=10;
printf("\n5.Worksharing for Construct\n");
#pragma omp parallel num_threads(4)
```

```
{
   #pragma omp for

      for(int i=0;i<n;i++){
      printf("Iteration:%d by thread:%d\n",i,omp_get_thread_num());
      }

}
printf("\n6.Explicit Barrier (usage with master thread)\n");
#pragma omp parallel num_threads(4)
{
   printf("Hi by thread %d\n",omp_get_thread_num());
   #pragma omp barrier
   #pragma omp master
   printf("HI BY MASTER THREAD %d\n",omp_get_thread_num());
   #pragma omp barrier
   printf("Hi by thread %d (after executing master thread)
\n",omp_get_thread_num());
}
}
```

———————————————————

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char** argv )
{
MPI_Init( &argc, &argv );
int size,rank;
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
printf( "Hello world. Size:%d, Rank:%d\n" ,size,rank);
MPI_Finalize();
return 0;
```

```
}
```

Merge sort

```
#include <stdio.h>

void merge(int arr[], int low, int mid, int high) {
    int temp[high - low + 1]; // temporary array
    int left = low;           // starting index of left half
    int right = mid + 1;      // starting index of right half
    int index = 0;            // index for the temporary array

    // Merging the two halves in sorted order
    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp[index++] = arr[left++];
        } else {
            temp[index++] = arr[right++];
        }
    }

    // Copying remaining elements of the left half, if any
    while (left <= mid) {
        temp[index++] = arr[left++];
    }

    // Copying remaining elements of the right half, if any
    while (right <= high) {
        temp[index++] = arr[right++];
    }

    // Copying the sorted elements back to the original array
    for (int i = 0; i < index; i++) {
        arr[low + i] = temp[i];
    }
}

void mergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);      // Sort the left half
```

```
        mergeSort(arr, mid + 1, high); // Sort the right half
        merge(arr, low, mid, high);     // Merge the sorted halves
    }
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Open mp

```
#include <stdio.h>
#include <omp.h> // OpenMP library

void merge(int arr[], int low, int mid, int high) {
    int temp[high - low + 1]; // temporary array
    int left = low;           // starting index of left half
    int right = mid + 1;      // starting index of right half
    int index = 0;            // index for the temporary array

    // Merging the two halves in sorted order
    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp[index++] = arr[left++];
        } else {
            temp[index++] = arr[right++];
        }
    }

    // Copying remaining elements of the left half, if any
    while (left <= mid) {
        temp[index++] = arr[left++];
    }
```

```
    // Copying remaining elements of the right half, if any
    while (right <= high) {
        temp[index++] = arr[right++];
    }

    // Copying the sorted elements back to the original array
    for (int i = 0; i < index; i++) {
        arr[low + i] = temp[i];
    }
}

void mergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;

        // Parallel region for recursive merge sort calls
        #pragma omp parallel sections
        {
            #pragma omp section
            {
                mergeSort(arr, low, mid);      // Sort the left half
            }
            #pragma omp section
            {
                mergeSort(arr, mid + 1, high); // Sort the right half
            }
        }

        merge(arr, low, mid, high); // Merge the sorted halves
    }
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Set the number of threads
    omp_set_num_threads(4);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
```

```
    return 0;
}
```

MPI

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

void merge(int arr[], int low, int mid, int high) {
    int temp[high - low + 1];
    int left = low;
    int right = mid + 1;
    int index = 0;

    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp[index++] = arr[left++];
        } else {
            temp[index++] = arr[right++];
        }
    }

    while (left <= mid) {
        temp[index++] = arr[left++];
    }

    while (right <= high) {
        temp[index++] = arr[right++];
    }

    for (int i = 0; i < index; i++) {
        arr[low + i] = temp[i];
    }
}

void mergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
```

```
    }
}

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

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    int local_n = n / size; // Size of subarray each process will handle
    int *local_arr = malloc(local_n * sizeof(int));

    // Scatter data across processes
    MPI_Scatter(arr, local_n, MPI_INT, local_arr, local_n, MPI_INT, 0,
MPI_COMM_WORLD);

    // Each process sorts its local array
    mergeSort(local_arr, 0, local_n - 1);

    // Gather sorted subarrays back to the root process
    MPI_Gather(local_arr, local_n, MPI_INT, arr, local_n, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        // Root process merges the sorted sections
        for (int i = 1; i < size; i++) {
            int mid = (i * local_n) - 1;
            int high = (i == size - 1) ? n - 1 : ((i + 1) * local_n) - 1;
            merge(arr, 0, mid, high);
        }

        // Print sorted array
        printf("Sorted array:\n");
        for (int i = 0; i < n; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
    }

    // Clean up
    free(local_arr);
    MPI_Finalize();
```

```
    return 0;
}
```

1. MPI-Based Bubble Sort

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

void bubble_sort(int *arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    int rank, size, n;
    int *data = NULL, *local_data = NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        n = atoi(argv[1]);  // Total elements
        data = (int *)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) data[i] = rand() % 100;

        int chunk_size = n / size;
        MPI_Scatter(data, chunk_size, MPI_INT, local_data, chunk_size, MPI_INT,
0, MPI_COMM_WORLD);
    } else {
        int chunk_size = n / size;
        local_data = (int *)malloc(chunk_size * sizeof(int));
        MPI_Scatter(NULL, chunk_size, MPI_INT, local_data, chunk_size, MPI_INT,
0, MPI_COMM_WORLD);
    }
```

```
    // Each process sorts its segment
    bubble_sort(local_data, n / size);

    // Gather the sorted segments back
    MPI_Gather(local_data, n / size, MPI_INT, data, n / size, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        // Further steps to merge the sorted segments at root process, if needed
        // Display final sorted array
        free(data);
    }
    free(local_data);
    MPI_Finalize();
    return 0;
}
```

=3. MPI-Based Jacobi Method

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void jacobi(int n, double *A, double *b, double *x, int max_iter, double tol) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double *x_new = (double *)malloc(n * sizeof(double));
    double diff = tol + 1;
    int iter = 0;

    while (iter < max_iter && diff > tol) {
        diff = 0.0;

        for (int i = rank; i < n; i += size) {
            double sigma = 0.0;
            for (int j = 0; j < n; j++) {
                if (j != i)
                    sigma += A[i * n + j] * x[j];
            }
            x_new[i] = (b[i] - sigma) / A[i * n + i];

            double local_diff = fabs(x_new[i] - x[i]);
            if (local_diff > diff) diff = local_diff;
```

```
        }

        MPI_Allgather(MPI_IN_PLACE, 1, MPI_DOUBLE, x_new, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
        for (int i = 0; i < n; i++) x[i] = x_new[i];

        MPI_Allreduce(MPI_IN_PLACE, &diff, 1, MPI_DOUBLE, MPI_MAX,
MPI_COMM_WORLD);
        iter++;
    }

    if (rank == 0) {
        printf("Converged after %d iterations with tolerance %f\n", iter, tol);
    }

    free(x_new);
}

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

    int n = 4; // System size (example)
    double A[16] = {4, -1, 0, 0, -1, 4, -1, 0, 0, -1, 4, -1, 0, 0, -1, 3}; // Example
matrix
    double b[4] = {15, 10, 10, 10};  // Example vector
    double x[4] = {0, 0, 0, 0};      // Initial guess

    jacobi(n, A, b, x, 1000, 1e-6);

    MPI_Finalize();
    return 0;
}
```

ask 1: MPI-Based Matrix Multiplication with PAPI

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

#define N 20
#define MAX_NUM 100
```

```
void merge(int* left, int left_size, int* right, int right_size, int* result) {
int i = 0, j = 0, k = 0;
while (i < left_size && j < right_size) {
if (left[i] < right[j]) {
result[k++] = left[i++];
} else {
result[k++] = right[j++];
}
}
while (i < left_size) {
result[k++] = left[i++];
}

while (j < right_size) {
result[k++] = right[j++];
}
}

// Merge sort function
void merge_sort(int* arr, int size) {
if (size < 2) return;

int mid = size / 2;
merge_sort(arr, mid);
merge_sort(arr + mid, size - mid);

int* temp = (int*)malloc(size * sizeof(int));
merge(arr, mid, arr + mid, size - mid, temp);
for (int i = 0; i < size; i++) {
arr[i] = temp[i];
}
free(temp);
}

int main(int argc, char** argv) {
int rank, size;
int arr[N];

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank == 0) {
srand(time(NULL));
for (int i = 0; i < N; i++) {
arr[i] = rand() % MAX_NUM;
}
```

```
printf("Original array: ");
for (int i = 0; i < N; i++) {
printf("%d ", arr[i]);
}
printf("\n");
}
int local_size = N / size;
int* local_arr = (int*)malloc(local_size * sizeof(int));

MPI_Scatter(arr, local_size, MPI_INT, local_arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);
merge_sort(local_arr, local_size);
MPI_Gather(local_arr, local_size, MPI_INT, arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD);
if (rank == 0) {
merge_sort(arr, N);
printf("Sorted array: ");
for (int i = 0; i < N; i++) {
printf("%d ", arr[i]);
}
printf("\n");
}

free(local_arr);
MPI_Finalize();
return 0;
}
```

```
#include <mpi.h>
#include <papi.h>
#include <stdio.h>
#include <stdlib.h>

#define N 4  // Matrix size, assuming N is divisible by the number of processes

void matrix_multiply(double *A, double *B, double *C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i * n + j] = 0.0;
            for (int k = 0; k < n; k++) {
                C[i * n + j] += A[i * n + k] * B[k * n + j];
            }
        }
```

```
    }
}

int main(int argc, char *argv[]) {
    int rank, size;
    double A[N * N], B[N * N], C[N * N];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        for (int i = 0; i < N * N; i++) {
            A[i] = rand() % 10;
            B[i] = rand() % 10;
        }
    }

    // Distribute matrix A and B blocks across processes
    int local_n = N / size;
    double *local_A = malloc(local_n * N * sizeof(double));
    double *local_C = malloc(local_n * N * sizeof(double));
    MPI_Scatter(A, local_n * N, MPI_DOUBLE, local_A, local_n * N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(B, N * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Initialize PAPI
    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
        printf("PAPI library initialization error!\n");
        MPI_Finalize();
        return 1;
    }

    int events[2] = {PAPI_FP_OPS, PAPI_L1_DCM};  // Floating-point operations
and L1 data cache misses
    long long values[2];

    if (PAPI_start_counters(events, 2) != PAPI_OK) {
        printf("PAPI start counters error!\n");
        MPI_Finalize();
        return 1;
    }

    // Perform local matrix multiplication
    matrix_multiply(local_A, B, local_C, local_n);

    // Stop PAPI counters and retrieve values
```

```
    if (PAPI_stop_counters(values, 2) != PAPI_OK) {
        printf("PAPI stop counters error!\n");
    } else if (rank == 0) {
        printf("Floating-point operations: %lld\n", values[0]);
        printf("L1 data cache misses: %lld\n", values[1]);
    }

    // Gather the result matrix C at root
    MPI_Gather(local_C, local_n * N, MPI_DOUBLE, C, local_n * N, MPI_DOUBLE,
0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Result matrix:\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%lf ", C[i * N + j]);
            }
            printf("\n");
        }
    }

    free(local_A);
    free(local_C);
    MPI_Finalize();
    return 0;
}
```

Task 2: 3x3 Cannon’s Algorithm for Matrix-Matrix Multiplication with PAPI

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <math.h>
    #include <mpi.h>
    #define N 16384
    int main(int argc, char *argv[])
    {
    MPI_Comm cannon_comm;
    MPI_Status status;
```

```
int rank,size;
int shift;
int i,j,k;
int dims[2];
int periods[2];
int left,right,up,down;
double *A,*B,*C;
double *buf,*tmp;
double start,end;
unsigned int iseed=0;
int Nl,N;
printf("Matrix size:");
printf("\n");
scanf("%d", &N);
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
srand(iseed);
dims[0]=0; dims[1]=0;
periods[0]=1; periods[1]=1;
MPI_Dims_create(size,2,dims);
if(dims[0]!=dims[1]) {
if(rank==0) printf("The number of processors must be a square.\n");
MPI_Finalize();
return 0;
}
Nl=N/dims[0];
A=(double*)malloc(Nl*Nl*sizeof(double));
B=(double*)malloc(Nl*Nl*sizeof(double));
buf=(double*)malloc(Nl*Nl*sizeof(double));
C=(double*)calloc(Nl*Nl,sizeof(double));
for(i=0;i<Nl;i++)
for(j=0;j<Nl;j++) {
A[i*Nl+j]=5-(int)( 10.0 * rand() / ( RAND_MAX + 1.0 ) );
B[i*Nl+j]=5-(int)( 10.0 * rand() / ( RAND_MAX + 1.0 ) );
C[i*Nl+j]=0.0;
}
MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,1,&cannon_comm);
MPI_Cart_shift(cannon_comm,0,1,&left,&right);
MPI_Cart_shift(cannon_comm,1,1,&up,&down);
start=MPI_Wtime();
for(shift=0;shift<dims[0];shift++) {
// Matrix multiplication
for(i=0;i<Nl;i++)
for(k=0;k<Nl;k++)
for(j=0;j<Nl;j++)
C[i*Nl+j]+=A[i*Nl+k]*B[k*Nl+j];
```

```
    if(shift==dims[0]-1) break;
    // Communication

MPI_Sendrecv(A,Nl*Nl,MPI_DOUBLE,left,1,buf,Nl*Nl,MPI_DOUBLE,right,1,canno
n_comm,&status);
    tmp=buf; buf=A; A=tmp;

MPI_Sendrecv(B,Nl*Nl,MPI_DOUBLE,up,2,buf,Nl*Nl,MPI_DOUBLE,down,2,cann
on_comm,&status);
    tmp=buf; buf=B; B=tmp;
    }
    MPI_Barrier(cannon_comm);
    end=MPI_Wtime();
    if(rank==0) printf("Time: %.4fs\n",end-start);
    free(A); free(B); free(buf); free(C);
    MPI_Finalize();
    return 0;
    }
```

```
#include <mpi.h>
#include <papi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 9  // Assume N=9 for a 3x3 process grid

void matrix_multiply(double *A, double *B, double *C, int block_size) {
    for (int i = 0; i < block_size; i++) {
        for (int j = 0; j < block_size; j++) {
            C[i * block_size + j] = 0.0;
            for (int k = 0; k < block_size; k++) {
                C[i * block_size + j] += A[i * block_size + k] * B[k * block_size + j];
```

```
            }
        }
    }
}

int main(int argc, char *argv[]) {
    int rank, size, block_size;
    double A[N], B[N], C[N] = {0};
    MPI_Comm cart_comm;
    int dims[2] = {3, 3}, periods[2] = {1, 1};

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &cart_comm);

    // Initialize matrices A and B
    for (int i = 0; i < N; i++) {
        A[i] = rand() % 10;
        B[i] = rand() % 10;
    }

    block_size = N / dims[0];
    double *local_A = malloc(block_size * block_size * sizeof(double));
    double *local_B = malloc(block_size * block_size * sizeof(double));
    double *local_C = calloc(block_size * block_size, sizeof(double));

    // Initialize PAPI
    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
        printf("PAPI library initialization error!\n");
        MPI_Finalize();
        return 1;
    }

    int events[2] = {PAPI_FP_OPS, PAPI_L1_DCM};
    long long values[2];

    if (PAPI_start_counters(events, 2) != PAPI_OK) {
        printf("PAPI start counters error!\n");
        MPI_Finalize();
        return 1;
    }

    // Perform Cannon's Algorithm
    matrix_multiply(local_A, local_B, local_C, block_size);

    if (PAPI_stop_counters(values, 2) != PAPI_OK) {
```

```
        printf("PAPI stop counters error!\n");
    } else if (rank == 0) {
        printf("Floating-point operations: %lld\n", values[0]);
        printf("L1 data cache misses: %lld\n", values[1]);
    }

    free(local_A);
    free(local_B);
    free(local_C);
    MPI_Finalize();
    return 0;
}
```

Task 1: MPI Point-to-Point Communication Using Bubble Sort

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

#define N 16  // Total number of elements

void bubble_sort(int *arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    int rank, size, local_n;
    int *data = NULL, *local_data = NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    local_n = N / size;  // Number of elements per process
    local_data = (int *)malloc(local_n * sizeof(int));

    if (rank == 0) {
```

```
        data = (int *)malloc(N * sizeof(int));
        for (int i = 0; i < N; i++) {
            data[i] = rand() % 100;  // Random numbers for demonstration
        }
        printf("Unsorted data:\n");
        for (int i = 0; i < N; i++) printf("%d ", data[i]);
        printf("\n");
    }

    // Scatter data to each process
    MPI_Scatter(data, local_n, MPI_INT, local_data, local_n, MPI_INT, 0,
MPI_COMM_WORLD);

    // Perform local Bubble Sort
    bubble_sort(local_data, local_n);

    // Neighbor exchange using point-to-point communication
    for (int step = 0; step < size; step++) {
        if (rank % 2 == 0) {
            if (rank + 1 < size) {
                MPI_Send(local_data + local_n - 1, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD);
                int recv_val;
                MPI_Recv(&recv_val, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                if (recv_val < local_data[local_n - 1]) {
                    local_data[local_n - 1] = recv_val;
                }
            }
        } else {
            if (rank - 1 >= 0) {
                int recv_val;
                MPI_Recv(&recv_val, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                MPI_Send(local_data, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);
                if (recv_val > local_data[0]) {
                    local_data[0] = recv_val;
                }
            }
        }
        MPI_Barrier(MPI_COMM_WORLD);  // Sync before next step
    }

    // Gather sorted data at root process
    MPI_Gather(local_data, local_n, MPI_INT, data, local_n, MPI_INT, 0,
MPI_COMM_WORLD);
```

```
    if (rank == 0) {
        printf("Sorted data:\n");
        for (int i = 0; i < N; i++) printf("%d ", data[i]);
        printf("\n");
        free(data);
    }

    free(local_data);
    MPI_Finalize();
    return 0;
}
```

Task 2: MPI Collective Communication Using Merge Sort

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

#define N 16  // Total number of elements

void merge(int *arr, int *temp, int left, int mid, int right) {
    int i = left, j = mid, k = left;
    while (i < mid && j < right) {
        if (arr[i] < arr[j]) temp[k++] = arr[i++];
        else temp[k++] = arr[j++];
    }
    while (i < mid) temp[k++] = arr[i++];
    while (j < right) temp[k++] = arr[j++];
    for (i = left; i < right; i++) arr[i] = temp[i];
}

void merge_sort(int *arr, int *temp, int left, int right) {
    if (right - left <= 1) return;
    int mid = (left + right) / 2;
    merge_sort(arr, temp, left, mid);
    merge_sort(arr, temp, mid, right);
    merge(arr, temp, left, mid, right);
}

int main(int argc, char *argv[]) {
    int rank, size, local_n;
```

```
    int *data = NULL, *local_data = NULL, *temp = NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    local_n = N / size;  // Number of elements per process
    local_data = (int *)malloc(local_n * sizeof(int));
    temp = (int *)malloc(local_n * sizeof(int));

    if (rank == 0) {
        data = (int *)malloc(N * sizeof(int));
        for (int i = 0; i < N; i++) {
            data[i] = rand() % 100;  // Random numbers for demonstration
        }
        printf("Unsorted data:\n");
        for (int i = 0; i < N; i++) printf("%d ", data[i]);
        printf("\n");
    }

    // Scatter data to each process
    MPI_Scatter(data, local_n, MPI_INT, local_data, local_n, MPI_INT, 0,
MPI_COMM_WORLD);

    // Perform local Merge Sort
    merge_sort(local_data, temp, 0, local_n);

    // Gather sorted segments back to root process
    MPI_Gather(local_data, local_n, MPI_INT, data, local_n, MPI_INT, 0,
MPI_COMM_WORLD);

    // Final merge at the root process
    if (rank == 0) {
        int *final_temp = (int *)malloc(N * sizeof(int));
        for (int i = 1; i < size; i++) {
            merge(data, final_temp, 0, i * local_n, (i + 1) * local_n);
        }
        printf("Sorted data:\n");
        for (int i = 0; i < N; i++) printf("%d ", data[i]);
        printf("\n");
        free(final_temp);
        free(data);
    }

    free(local_data);
    free(temp);
    MPI_Finalize();
```

```
    return 0;
}
```

ii) Matrix Multiplication with MPI\

In this example, we'll divide matrix multiplication across multiple processes. Each process will calculate a portion of the resulting matrix and send it back to the root process for assembly.
c
Copy code

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

#define N 4 // Define the size of the matrices (NxN)

int main(int argc, char *argv[]) {
    int rank, size;
    int A[N][N], B[N][N], C[N][N] = {0}; // Matrices
    int local_A[N][N], local_C[N][N] = {0};

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Initialize matrices on the root process
    if (rank == 0) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                A[i][j] = i + j; // Example values, customize as needed
                B[i][j] = i - j;
            }
        }
    }

    // Broadcast matrix B to all processes
    MPI_Bcast(B, N * N, MPI_INT, 0, MPI_COMM_WORLD);

    // Scatter rows of matrix A to all processes
    MPI_Scatter(A, N * N / size, MPI_INT, local_A, N * N / size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Perform multiplication on the assigned rows
    for (int i = 0; i < N / size; i++) {
        for (int j = 0; j < N; j++) {
```

```
            local_C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                local_C[i][j] += local_A[i][k] * B[k][j];
            }
        }
    }

    // Gather the result back to the root process
    MPI_Gather(local_C, N * N / size, MPI_INT, C, N * N / size, MPI_INT, 0,
MPI_COMM_WORLD);

    // Display result on the root process
    if (rank == 0) {
        printf("Result Matrix C:\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                printf("%d ", C[i][j]);
            }
            printf("\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

iii) Finding Prime Numbers with MPI

In this example, we'll divide the range of numbers to check for primality across processes, each of which will independently identify primes in its segment.

c

Copy code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

int is_prime(int num) {
    if (num <= 1) return 0;
    if (num <= 3) return 1;
    if (num % 2 == 0 || num % 3 == 0) return 0;
    for (int i = 5; i <= sqrt(num); i += 6) {
        if (num % i == 0 || num % (i + 2) == 0) return 0;
    }
    return 1;
}
```

```
int main(int argc, char *argv[]) {
    int rank, size, start, end;
    int n = 100; // Define the upper limit for finding primes
    int local_prime_count = 0, global_prime_count = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Divide range across processes
    int range = n / size;
    start = rank * range + 1;
    end = (rank + 1) * range;

    // Adjust the last process's range to go up to n
    if (rank == size - 1) end = n;

    // Find primes in the assigned range
    for (int i = start; i <= end; i++) {
        if (is_prime(i)) {
            local_prime_count++;
        }
    }

    // Sum all local prime counts to get the global prime count
    MPI_Reduce(&local_prime_count, &global_prime_count, 1, MPI_INT,
MPI_SUM, 0, MPI_COMM_WORLD);

    // Display result on the root process
    if (rank == 0) {
        printf("Total number of primes up to %d is %d\n", n, global_prime_count);
    }

    MPI_Finalize();
    return 0;
}
```

```
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
```

```
bool is_prime(long num) {
if (num <= 1) return false;
for (long i = 2; i * i <= num; i++) {
if (num % i == 0) return false;
}
return true;
}

long nth_prime(long n) {
long count = 0;
long num = 1;
while (count < n) {
num++;
if (is_prime(num)) {
count++;
}
}
return num;
}

int main() {
long n;
clock_t start, end;

printf("Enter the value of n: ");
scanf("%ld", &n);

start = clock();
long prime = nth_prime(n);
end = clock();

printf("The %ldth prime number is: %ld\n", n, prime);
double time= ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Execution Time: %f seconds\n", time);

return 0;
}
```

ii) Matrix Multiplication with OpenMP

For matrix multiplication, we can parallelize the loop that computes the elements of the resulting matrix.
c
Copy code

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

#define N 4 // Define the size of the matrices (NxN)

int main() {
    int A[N][N], B[N][N], C[N][N] = {0};

    // Initialize matrices A and B
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = i + j; // Example values, customize as needed
            B[i][j] = i - j;
        }
    }

    // Parallel matrix multiplication
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    // Display result
    printf("Result Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

iii) Finding Prime Numbers with OpenMP

In this example, we'll find primes up to a given limit, parallelizing the loop where we check each number for primality.

c

Copy code

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

int is_prime(int num) {
    if (num <= 1) return 0;
    if (num <= 3) return 1;
    if (num % 2 == 0 || num % 3 == 0) return 0;
    for (int i = 5; i <= sqrt(num); i += 6) {
        if (num % i == 0 || num % (i + 2) == 0) return 0;
    }
    return 1;
}

int main() {
    int n = 100; // Define the upper limit for finding primes
    int prime_count = 0;

    #pragma omp parallel for reduction(+:prime_count)
    for (int i = 2; i <= n; i++) {
        if (is_prime(i)) {
            prime_count++;
        }
    }

    printf("Total number of primes up to %d is %d\n", n, prime_count);
    return 0;
}
```

trapezoidal

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
```

```
double f(double x) {
    return x * x;  // Example function: f(x) = x^2
}

double trapezoidal_rule(double a, double b, int n) {
    double h = (b - a) / n;
    double sum = (f(a) + f(b)) / 2.0;

    for (int i = 1; i < n; i++) {
        sum += f(a + i * h);
    }

    return sum * h;
}

int main(int argc, char **argv) {
    int rank, size;
    double a = 0.0;    // Lower limit
    double b = 2.0;    // Upper limit
    int n = 1000000;   // Number of subdivisions
    double local_a, local_b, local_n;
    double local_sum, total_sum;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Determine the local range for each process
    local_n = n / size; // Number of trapezoids for each process
    local_a = a + rank * local_n * (b - a) / n;
    local_b = a + (rank + 1) * local_n * (b - a) / n;

    // Calculate local sum using the trapezoidal rule
    local_sum = trapezoidal_rule(local_a, local_b, local_n);

    // Reduce all local sums to total sum at rank 0
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    // Rank 0 prints the result
    if (rank == 0) {
        printf("Estimated integral from %.2f to %.2f is: %.10f\n", a, b, total_sum);
    }

    MPI_Finalize();
    return 0;
```

```
}
```

Jacobian

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void function(double *x, double *f) {
    f[0] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] + x[3] * x[3];
    f[1] = sin(x[0]);
    f[2] = cos(x[1]);
    f[3] = exp(x[2]);
}

void compute_jacobian_for_row(double *x, double *jacobian_row, int row, int n,
double eps) {
    double f1[4], f2[4];
    function(x, f1);

    for (int j = 0; j < n; j++) {
        double x_temp = x[j];

        x[j] = x_temp + eps;
        function(x, f2);

        jacobian_row[j] = (f2[row] - f1[row]) / eps;

        x[j] = x_temp;
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int n = 4;
    int m = 4;

    double x[4] = {1.0, 2.0, 3.0, 4.0};
```

```
    double jacobian_row[n];

    double *global_jacobian = NULL;
    if (rank == 0) {
        global_jacobian = (double*)malloc(m * n * sizeof(double));
    }

    int rows_per_process = (m + size - 1) / size;

    for (int i = 0; i < rows_per_process; i++) {
        int row = rank * rows_per_process + i;
        if (row < m) {
            compute_jacobian_for_row(x, jacobian_row, row, n, 1e-6);
            MPI_Gather(jacobian_row, n, MPI_DOUBLE,
                       global_jacobian + row * n, n, MPI_DOUBLE,
                       0, MPI_COMM_WORLD);
        }
    }

    if (rank == 0) {
        printf("Jacobian Matrix:\n");
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                printf("%f ", global_jacobian[i * n + j]);
            }
            printf("\n");
        }
    }
    MPI_Finalize();
    return 0;
}
```

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void function(double *x, double *f) {
    f[0] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] + x[3] * x[3];
    f[1] = sin(x[0]);
    f[2] = cos(x[1]);
    f[3] = exp(x[2]);
```

```
}

void compute_jacobian_for_row(double *x, double *jacobian_row, int row, int n,
double eps) {
    double f1[4], f2[4];
    function(x, f1);

    for (int j = 0; j < n; j++) {
        double x_temp = x[j];

        x[j] = x_temp + eps;
        function(x, f2);

        jacobian_row[j] = (f2[row] - f1[row]) / eps;

        x[j] = x_temp;
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int n = 4;
    int m = 4;

    double x[4] = {1.0, 2.0, 3.0, 4.0};
    double jacobian_row[n];

    double *global_jacobian = NULL;
    if (rank == 0) {
        global_jacobian = (double*)malloc(m * n * sizeof(double));
    }

    int rows_per_process = (m + size - 1) / size;

    for (int i = 0; i < rows_per_process; i++) {
        int row = rank * rows_per_process + i;
        if (row < m) {
            compute_jacobian_for_row(x, jacobian_row, row, n, 1e-6);
            MPI_Gather(jacobian_row, n, MPI_DOUBLE,
                       global_jacobian + row * n, n, MPI_DOUBLE,
                       0, MPI_COMM_WORLD);
        }
```

```
    }

    if (rank == 0) {
        printf("Jacobian Matrix:\n");
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                printf("%f ", global_jacobian[i * n + j]);
            }
            printf("\n");
        }
    }
    MPI_Finalize();
    return 0;
}
```

Fibonacci Sequence

```
/*****************************************************************************
* This is an example to show how to use low level function PAPI_get_real_cyc *
* and PAPI_get_real_usec. *
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

int your_slow_code()
{
int i,tmp;

for(i=1; i<20000; i++)
{
tmp=(tmp+100)/i;
}
return 0;
}
long double fib(long double n){
long double a=0,b=1,c,k;
```

```
for(long double i=0;i<=n;i++){
k=a;
c=a+b;
a=b;
b=c;
}
return k;
}

int main()
{
long long s,s1, e, e1;
int retval;

/*************************************************************************
* This part initializes the library and compares the version number of the *
* header file, to the version of the library, if these don't match then it *
* is likely that PAPI won't work correctly.If there is an error, retval *
* keeps track of the version number. *
*************************************************************************/

if((retval = PAPI_library_init(PAPI_VER_CURRENT)) != PAPI_VER_CURRENT )
{
printf("Library initialization error! \n");
exit(1);
}
/* Here you get initial cycles and time */
/* No error checking is done here because this function call is always
successful */
for(int i=10;i<=10000;i*=10){
s = PAPI_get_real_cyc();
s1= PAPI_get_real_usec();
// printf("\n Here, I need to include the fibonaci code ");
int x=fib(i);

/*Here you get final cycles and time */
e = PAPI_get_real_cyc();


// your_slow_code();

e1= PAPI_get_real_usec();

printf("For n=%d\nWallclock cycles : %lld\nWallclock time(ms): %lld\n",i,e-s,e1-
s1);
}
/* clean up */
```

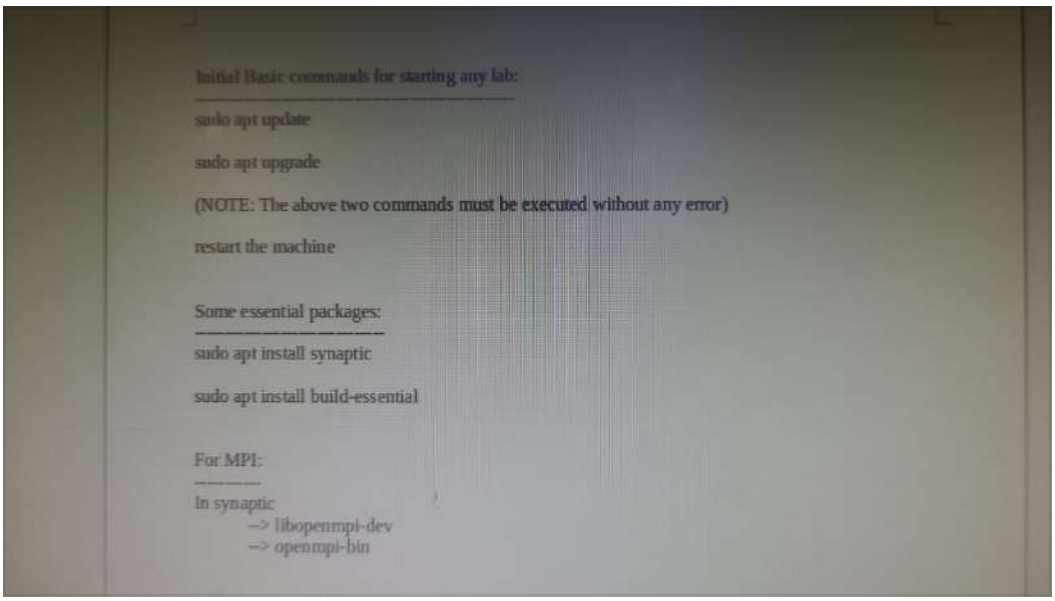
```
PAPI_shutdown();

exit(0);
}
```

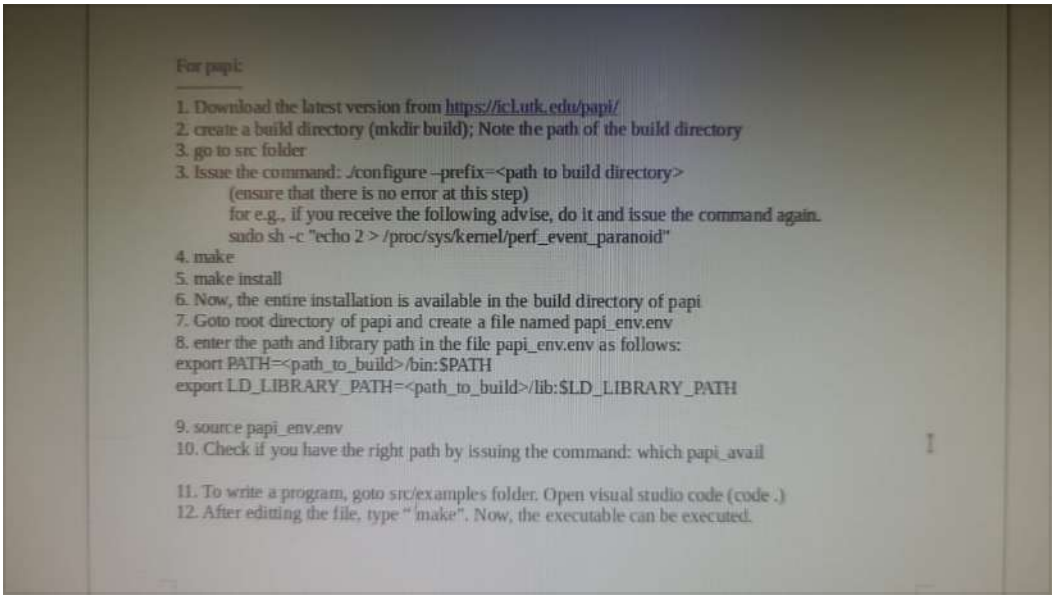
```
#include <stdio.h>
#include <time.h>
long double fib(long double n){
long double a=0,b=1,c,k;
for(long double i=0;i<=n;i++){
k=a;
c=a+b;
a=b;
b=c;
}
return k;
}
void main(){
clock_t start,end;
start=clock();
printf("Enter the term to display of the Fibonacci Sequence:");
long double a=0;
scanf("%Lf",&a);
printf("The Following is the %Lfth term of the Fibonacci Sequence:
%Lf",a,fib(a));
end=clock();
double time=(double)(end-start)/CLOCKS_PER_SEC;
printf("\nExecution Time: %f seconds\n",time);
//print(fib(a));
}
```

```
procedure MatrixMultiplication(A, B)
  input A, B n*n matrix
  output C, n*n matrix

begin
  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      C[i,j] = 0;
    end for
  end for

  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      for( k = 0; k < n; k++)
       C[i,j] = C[i,j] + A[i,k] * B[k,j]
      end for
    end for
  end for
end MatrixMultiplication
```

```
#include<stdio.h>
#include<time.h>
void main(){
clock_t start,end;
start=clock();
int m,n,p;
printf("Enter no of rows for first matrix:");
scanf("%d",&m);
printf("Enter no of columns for first matrix:");
scanf("%d",&n);
printf("Enter no of columns for second matrix:");
scanf("%d",&p);
int a[m][n],b[n][p],C[m][p];
int i,j,k;
printf("Enter elements for the first matrix(%d x %d):",m,n);
for(i=0;i<m;i++){
   for(j=0;j<n;j++){
   scanf("%d",&a[i][j]);
   }
}
```

```
printf("Enter elements for the first matrix(%d x %d):",n,p);
for(i=0;i<n;i++){
   for(j=0;j<p;j++){
   scanf("%d",&b[i][j]);
}
}
for(i=0;i<m;i++){
   for(j=0;j<p;j++){
   C[i][j]=0;
   for(k=0;k<n;k++){
      C[i][j]+=a[i][k]*b[k][j];
      }
   }
}
printf("Product Matrix is:\n");
for(i=0;i<m;i++){
   for(j=0;j<p;j++){
      printf("%d ",C[i][j]);
   }
printf("\n");

}
end=clock();
double time=(double)(end-start)/CLOCKS_PER_SEC;
printf("\nExecution Time: %f seconds\n",time);
}
```



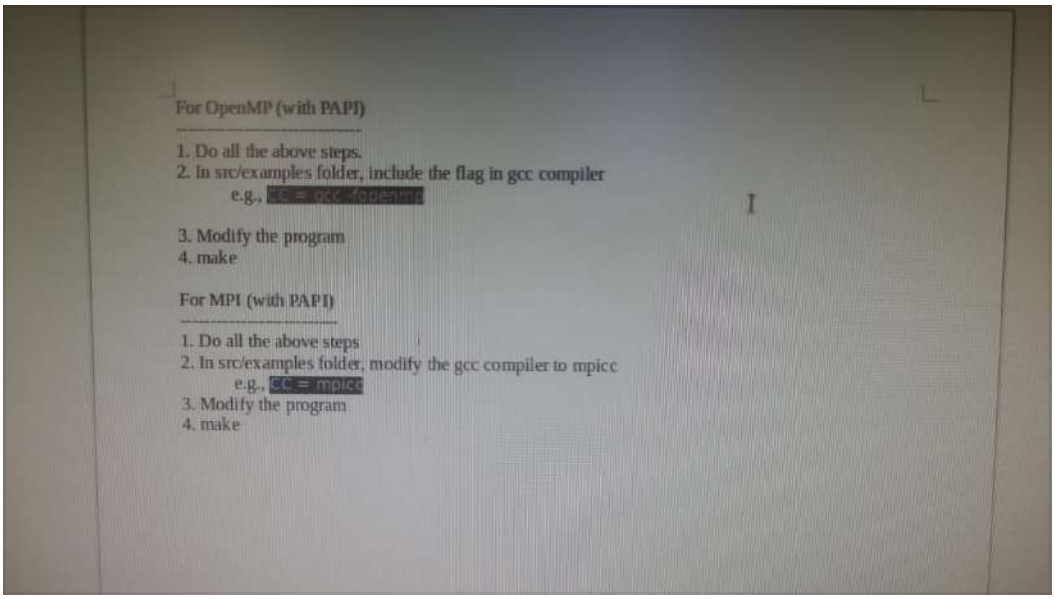


For papi:

1. Download the latest version from https://icl.utk.edu/papi/
2. create a build directory (mkdir build); Note the path of the build directory
3. go to src folder
3. Issue the command: ./configure –prefix=<path to build directory>
   (ensure that there is no error at this step)
   for e.g., if you receive the following advise, do it and issue the command again.
   sudo sh -c "echo 2 > /proc/sys/kernel/perf_event_paranoid"
4. make
5. make install
6. Now, the entire installation is available in the build directory of papi
7. Goto root directory of papi and create a file named papi_env.env
8. enter the path and library path in the file papi_env.env as follows:
export PATH=<path_to_build>/bin:$PATH
export LD_LIBRARY_PATH=<path_to_build>/lib:$LD_LIBRARY_PATH

9. source papi_env.env
10. Check if you have the right path by issuing the command: which papi_avail

11. To write a program, goto src/examples folder. Open visual studio code (code .)
12. After editting the file, type " make". Now, the executable can be executed.



For OpenMP (with PAPI)

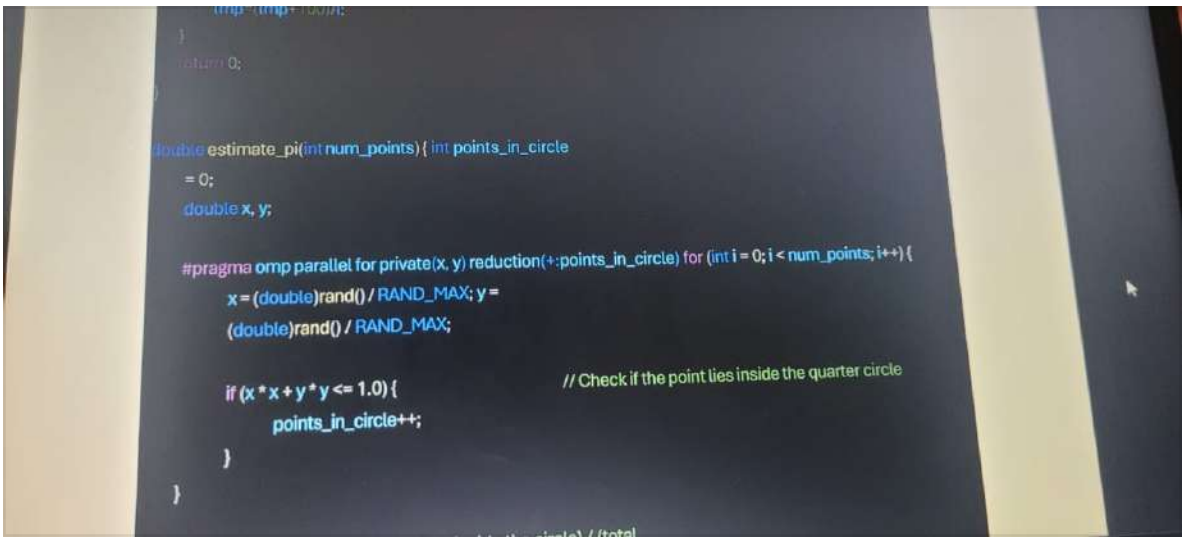
1. Do all the above steps.
2. In src/examples folder, include the flag in gcc compiler
   e.g., CC = gcc -fopenmp

3. Modify the program
4. make

For MPI (with PAPI)

1. Do all the above steps
2. In src/examples folder, modify the gcc compiler to mpicc
   e.g., CC = mpicc
3. Modify the program
4. make

```
for i from 1 to N
    for j from 0 to N - 1
      if a[j] > a[j + 1]
         swap ( a[j] , a[j + 1] )
```




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

#define NUM_SAMPLES 1000000000  // Total number of random samples

int main() {
    long long count = 0;  // Count of points inside the quarter circle

    // OpenMP parallel region
    #pragma omp parallel
    {
        long long private_count = 0; // Private count for each thread
        unsigned int seed = omp_get_thread_num(); // Seed for random number
generation

        #pragma omp for
        for (long long i = 0; i < NUM_SAMPLES; i++) {
            double x = (double)rand_r(&seed) / RAND_MAX; // Random x in [0, 1]
            double y = (double)rand_r(&seed) / RAND_MAX; // Random y in [0, 1]
            if (x * x + y * y <= 1.0) {
                private_count++; // Inside the circle
            }
        }

        // Combine the counts from all threads
        #pragma omp atomic
```

```
        count += private_count;
    }

    // Calculate the estimated value of π
    double pi_estimate = (double)count / NUM_SAMPLES * 4.0;

    printf("Estimated value of π = %f\n", pi_estimate);
    return 0;
}
```

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 8

static long steps = 1000000000;
double step;

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

    int i,j;
    double x;
    double pi, sum = 0.0;
    double start, delta;

    step = 1.0/(double) steps;

    // Compute parallel compute times for 1-MAX_THREADS
    for (j=1; j<= MAX_THREADS; j++) {

        printf(" running on %d threads: ", j);

        // This is the beginning of a single PI computation
        omp_set_num_threads(j);

        sum = 0.0;
        double start = omp_get_wtime();
```

```
        #pragma omp parallel for reduction(+:sum) private(x)
        for (i=0; i < steps; i++) {
            x = (i+0.5)*step;
            sum += 4.0 / (1.0+x*x);
        }

        // Out of the parallel region, finialize computation
        pi = step * sum;
        delta = omp_get_wtime() - start;
        printf("PI = %.16g computed in %.4g seconds\n", pi, delta);

    }


}
```

Task 1: Sieve of Eratosthenes Algorithm using MPI and OpenMP

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

#define N 1000

void sieve(int start, int end, int *prime) {
    #pragma omp parallel for schedule(dynamic)
    for (int i = 2; i * i <= end; i++) {
        if (prime[i]) {
            for (int j = i * i; j <= end; j += i) {
                prime[j] = 0;
            }
        }
    }
```

```
}

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int prime[N + 1];
    for (int i = 0; i <= N; i++) prime[i] = 1;

    int start = (rank * N) / size + 1;
    int end = ((rank + 1) * N) / size;

    sieve(start, end, prime);

    if (rank == 0) {
        for (int i = 2; i <= N; i++) {
            if (prime[i]) {
                printf("%d ", i);
            }
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

———————————

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
#include "papi.h"
// Function to print a matrix (for debugging)
```

```
void printMatrix(int *matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d\t", matrix[i * cols + j]);
        }
        printf("\n");
    }
    printf("\n");
}

// Function to perform matrix multiplication using Cannon's algorithm
void cannonMatrixMultiply(int *localA, int *localB, int *localC, int local_n) {
    for (int i = 0; i < local_n; i++) {
        for (int j = 0; j < local_n; j++) {
            for (int k = 0; k < local_n; k++) {
                localC[i * local_n + j] += localA[i * local_n + k] * localB[k * local_n + j];
            }
        }
    }
}

int main(int argc, char *argv[]) {
    int n;              // Matrix dimension
    int sqrt_p;         // Square root of the number of processes
    int myrank, p;      // Rank and size of the MPI communicator
    int *A, *B, *C;     // Matrices A, B, and C
    int *localA, *localB, *localC; // Local matrices for each process
    int local_n;        // Dimension of local matrices
    long long s, s1, e, e1;
    int retval;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

if ((retval = PAPI_library_init(PAPI_VER_CURRENT)) != PAPI_VER_CURRENT) {
        printf("Library initialization error! \n");
        MPI_Finalize();
        exit(1);
    }

    if (argc != 2) {
        if (myrank == 0) {
            printf("Usage: %s <matrix_dimension>\n", argv[0]);
        }
        MPI_Finalize();
        return 1;
    }
```

```
    n = 3;
    sqrt_p = (int)sqrt(p);

    if (n % sqrt_p != 0) {
        if (myrank == 0) {
            printf("Matrix dimension must be divisible by the square root of the
number of processes.\n");
        }
        MPI_Finalize();
        return 1;
    }

    // Initialize matrices A, B, and C on all processes
    A = (int *)malloc(n * n * sizeof(int));
    B = (int *)malloc(n * n * sizeof(int));
    C = (int *)calloc(n * n, sizeof(int));

    // Initialize matrices A and B with random values on process 0
    if (myrank == 0) {
        srand(42); // Seed for reproducibility
        for (int i = 0; i < n * n; i++) {
            A[i] = rand() % 10000;
            B[i] = rand() % 10000;
        }

        // Print the input matrices (for debugging)
        printf("Matrix A:\n");
        printMatrix(A, n, n);
        printf("Matrix B:\n");
        printMatrix(B, n, n);
    }

    // Broadcast the matrix dimension to all processes
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Calculate the dimension of local matrices
    local_n = n / sqrt_p;

    // Allocate memory for local matrices
    localA = (int *)malloc(local_n * local_n * sizeof(int));
    localB = (int *)malloc(local_n * local_n * sizeof(int));
    localC = (int *)calloc(local_n * local_n, sizeof(int));
s = PAPI_get_real_cyc();
    s1 = PAPI_get_real_usec();

    // Scatter blocks of A and B to all processes
```

```
    MPI_Scatter(A, local_n * local_n, MPI_INT, localA, local_n * local_n, MPI_INT,
0, MPI_COMM_WORLD);
    MPI_Scatter(B, local_n * local_n, MPI_INT, localB, local_n * local_n, MPI_INT,
0, MPI_COMM_WORLD);

    // Perform matrix multiplication using Cannon's algorithm
    cannonMatrixMultiply(localA, localB, localC, local_n);

    // Gather the results from all processes
    MPI_Gather(localC, local_n * local_n, MPI_INT, C, local_n * local_n, MPI_INT,
0, MPI_COMM_WORLD);
e = PAPI_get_real_cyc();
    e1 = PAPI_get_real_usec();

    // Print the result matrix (only by process 0 for simplicity)
    if (myrank == 0) {
        printf("Matrix C (Result):\n");
        printMatrix(C, n, n);
        printf("\nWallclock cycles : %lld\nWallclock time(ms): %lld\n", e - s, e1 -
s1);
    }

    free(A);
    free(B);
    free(C);
    free(localA);
    free(localB);
    free(localC);
    PAPI_shutdown();
    MPI_Finalize();

    return 0;
}
```

Code for Amdahl’s Law with OpenMP Parallelism

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

double calculate_speedup(double P, int N) {
    return 1.0 / ((1.0 - P) + (P / N));
```

```
}

int main() {
    double P;  // Parallelizable portion of the program
    int max_processors; // Maximum number of processors to simulate

    // User input for parallelizable fraction and maximum processors
    printf("Enter the parallelizable fraction of the program (P) (0 <= P <= 1): ");
    scanf("%lf", &P);
    printf("Enter the maximum number of processors to simulate: ");
    scanf("%d", &max_processors);

    // Ensure valid input
    if (P < 0 || P > 1) {
        printf("Error: Parallelizable fraction must be between 0 and 1.\n");
        return 1;
    }
    if (max_processors <= 0) {
        printf("Error: Number of processors must be greater than 0.\n");
        return 1;
    }

    // Array to store speedup results
    double speedup_results[max_processors + 1];

    // Parallel loop to calculate speedup for each number of processors
    #pragma omp parallel for
    for (int N = 1; N <= max_processors; N++) {
        speedup_results[N] = calculate_speedup(P, N);
    }

    // Print results
    printf("Processors\tSpeedup\n");
    for (int N = 1; N <= max_processors; N++) {
        printf("%d\t\t%f\n", N, speedup_results[N]);
    }

    return 0;
}
```

———————————————————————

Task 1: Applying MPI_Wtime() to Measure Execution Time

The function MPI_Wtime() returns the elapsed wall-clock time in seconds. You can use it at the start and end of your program to measure the total execution time.

Example: Adding MPI_Wtime() to Measure Execution Time

In the previous example with the Sieve of Eratosthenes, you can modify the code to include MPI_Wtime() as follows:

cpp

Copy code

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

#define N 1000

void sieve(int start, int end, int *prime) {
    #pragma omp parallel for schedule(dynamic)
    for (int i = 2; i * i <= end; i++) {
        if (prime[i]) {
            for (int j = i * i; j <= end; j += i) {
                prime[j] = 0;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double start_time = MPI_Wtime(); // Start timing

    int prime[N + 1];
    for (int i = 0; i <= N; i++) prime[i] = 1;
```

```
    int start = (rank * N) / size + 1;
    int end = ((rank + 1) * N) / size;

    sieve(start, end, prime);

    if (rank == 0) {
        for (int i = 2; i <= N; i++) {
            if (prime[i]) {
                printf("%d ", i);
            }
        }
        printf("\n");
    }

    double end_time = MPI_Wtime(); // End timing

    if (rank == 0) {
        printf("Execution Time: %f seconds\n", end_time - start_time);
    }

    MPI_Finalize();
    return 0;
}
```

In this code:

- start_time records the time at the beginning of the computation.
- end_time records the time after the computation.
- The difference end_time - start_time provides the elapsed execution time, which is printed by the root process (rank 0).

Task 2: Demonstrate the Efficiency of MPI_Bcast vs. MPI_Send

In many cases, MPI_Bcast is more efficient than using multiple MPI_Send calls because it is designed for broadcasting data from one process to all others in a single collective operation. This is useful for distributing data from one process to all other processes without having to loop through MPI_Send.

Example: Using MPI_Bcast and MPI_Send for Comparison

Here's an example where a root process sends an integer to all other processes. We'll first use MPI_Send in a loop and then demonstrate the same functionality with MPI_Bcast.

cpp

Copy code

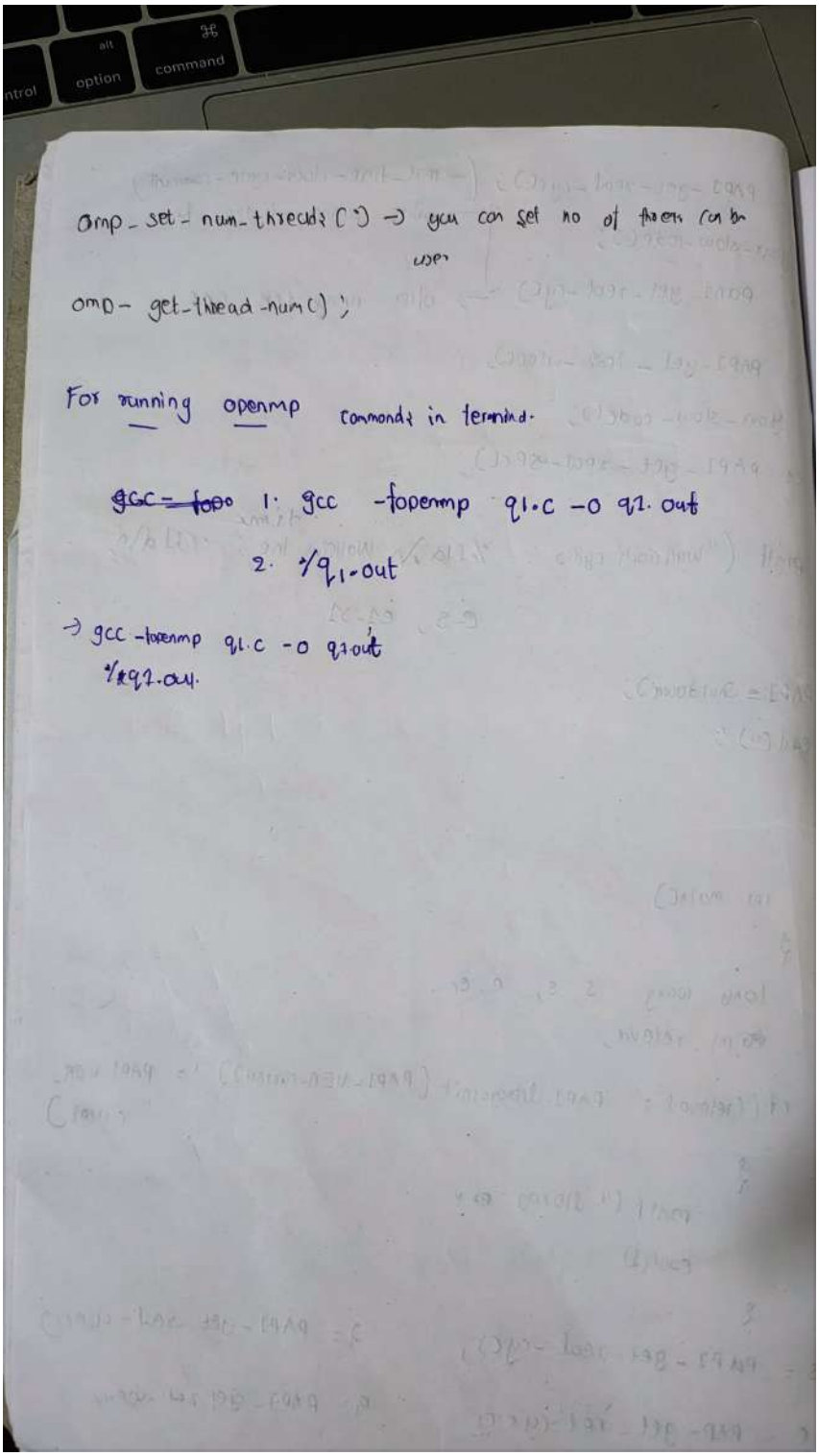
```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank, size, data;
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        data = 42; // Root process sets data
        double start_time = MPI_Wtime();

        // Using MPI_Send to send data to all other processes
        for (int i = 1; i < size; i++) {
            MPI_Send(&data, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        double end_time = MPI_Wtime();
        printf("MPI_Send time: %f seconds\n", end_time - start_time);
    } else {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process %d received data %d using MPI_Send\n", rank, data);
    }

    // Synchronize before starting MPI_Bcast
    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0) {
        data = 42; // Reset data in root process
        double start_time = MPI_Wtime();

        // Using MPI_Bcast to broadcast data from root to all processes
        MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

        double end_time = MPI_Wtime();
        printf("MPI_Bcast time: %f seconds\n", end_time - start_time);
    } else {
        MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf("Process %d received data %d using MPI_Bcast\n", rank, data);
    }

    MPI_Finalize();
    return 0;
}
```

Explanation of the Code

- **MPI_Send**: The root process (rank 0) sends the integer data to each other process individually in a loop. This can become time-consuming as the number of processes increases.
- **MPI_Bcast**: The root process broadcasts the integer data to all other processes in one call, which is typically faster as it’s optimized for broadcasting data.

By comparing the execution time printed for MPI_Send and MPI_Bcast, you should see that MPI_Bcast is more efficient for distributing the same data to multiple processes.

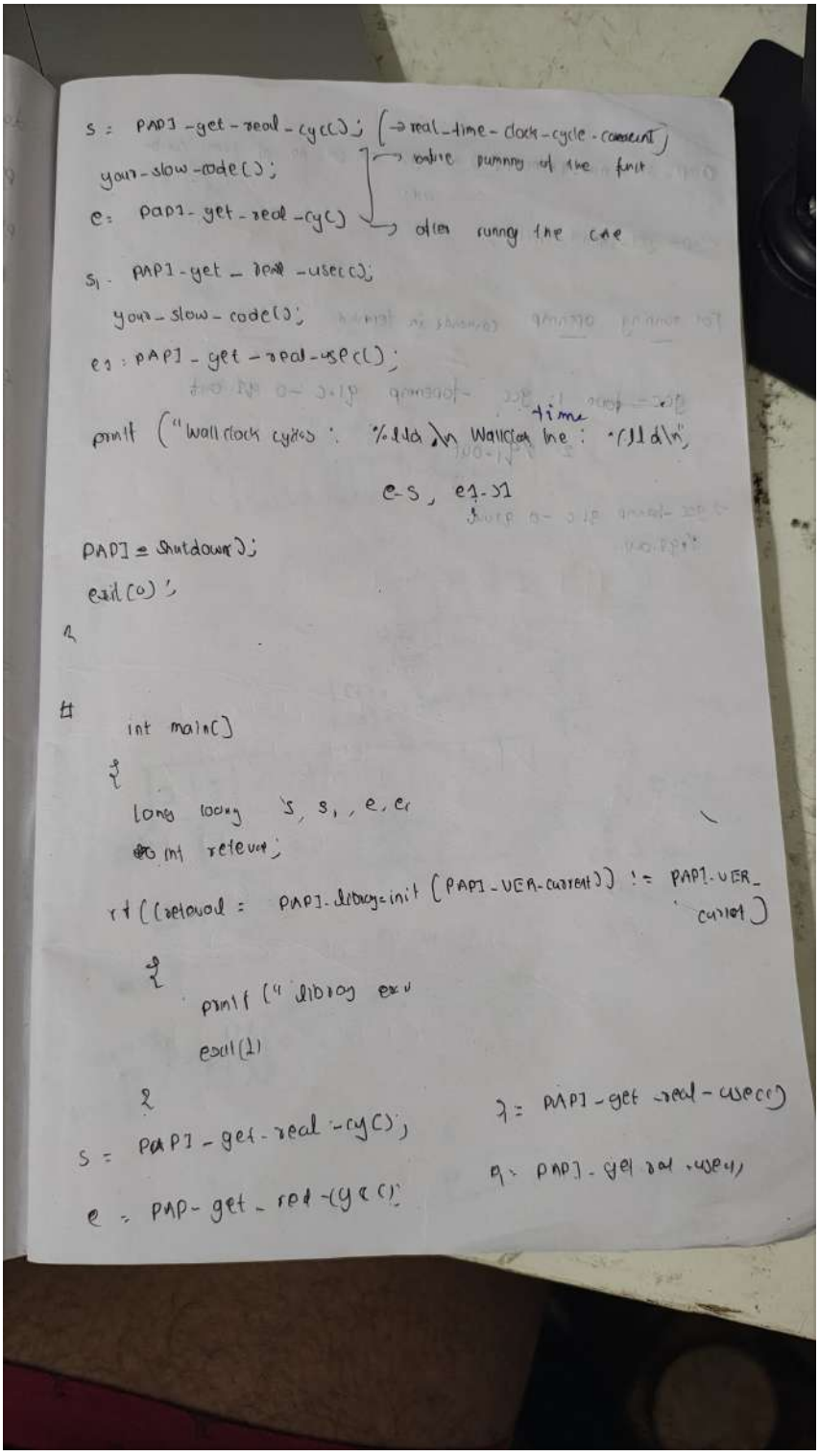


```
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
printf ("Elapsed %.4f", time_spent);
printf ("clock cycle : %.4d\n", (long int)(end - begin))
```

For prime no:

For papi

```
#include <stdio.h>
#include <stdlib.h>
#include "papi.h"

int you_slow_code()
{
}

int main()
{
    long long s, s1, e,
    int retval;
    printf ("hello world")

    if ( (retval = PAPI_library_init (PAPI_VER_CURRENT))
                != PAPI_VER_CURR
    {
        perror ("library error")
        exit(1)
```



```
s = PAPI_get_real_cyc();   (→ real_time_clock_cycle_counter)
your_slow_code();          → entire running of the first
e = PAPI_get_real_cyc()    → after running the one

s1 = PAPI_get_real_usec();
your_slow_code();
e1 = PAPI_get_real_usec();

printf("Wall clock cycles: %lld \n Wall clock time: %lld\n",
                                   e-s, e1-s1

PAPI_shutdown();
exit(0);
}

#
int main()
{
long long s, s1, e, e1
int retval;
if((retval = PAPI_library_init(PAPI_VER_CURRENT)) != PAPI_VER_
                                                    CURRENT)
{
printf("library ex
exit(1)
}
s = PAPI_get_real_cyc();        s1 = PAPI_get_real_usec()
e = PAP_get_real_cyc();         e1 = PAPI_get_real_usec();
```

```
double (time_spent) = double(end - begin) / clocks_per_sec
printf ("%.2f \n", a)
printf ("Elapsed : %f seconds\n", time_spent)
printf ("clock cycle : %ld\n", (long int)(end - begin));

time (&rawtime)
time info = local time (& rawtime)
printf ("Current time %.s", asctime (time info))
return(0)
```

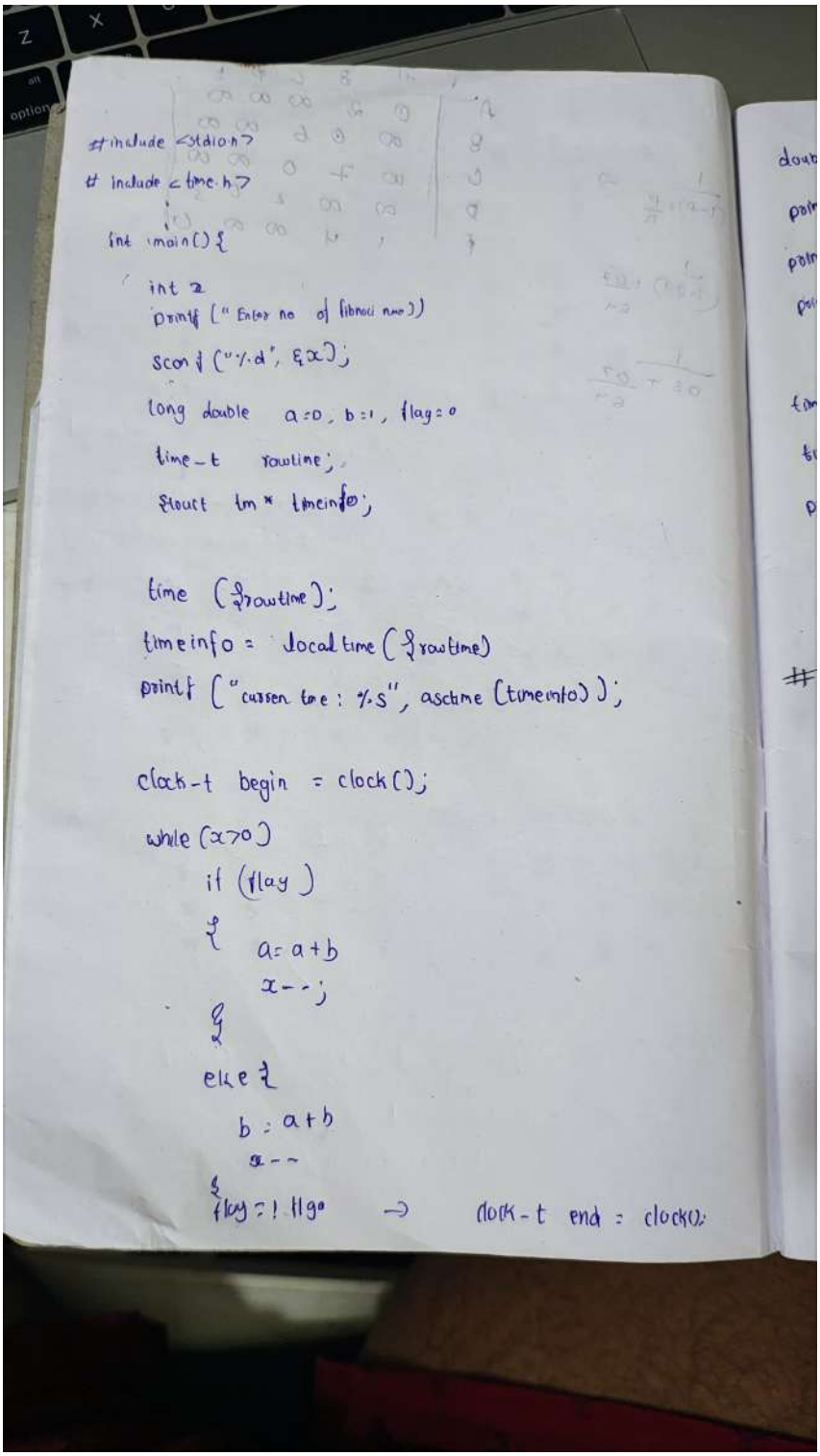
%.ld -> long int
%.LF -> long double
%.lf -> long float
%.s -> asctime

For big Fibonacci
=>

%.lld -> long long int

#:

```
time_t rawtime;
struct tm * timeinfo;

time (&rawtime)
time_info = local time (&rawtime)
printf ("current time : %.s", asctime (timeinfo))

clock_t begin = clock ()
{
}
clock_t end = clock ()
```



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

int main(){

   int x
   Printf ("Enter no of fibnoci num"))
   scanf ("%d", &x);
   long double  a=0, b=1, flag=0
   time_t   rawtime;
   struct tm * timeinfo;

   time (&rawtime);
   timeinfo = localtime (&rawtime)
   printf ("current time: %s", asctime (timeinfo));

   clock_t begin = clock();
   while (x>0)
      if (flag)
      {  a= a+b
         x--;
      }
      else {
         b = a+b
         x--
      }
      flag = ! flag      →      clock_t end = clock();
```