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("\nG.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, U, 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, U, 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, U, 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[1G] = {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-G);

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 1G384

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 U // Assume N=U 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 1G // 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 1G // 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;

}

# 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;

}

# 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 += G) {

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;

}

# 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;

}

# 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 += G) {

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-G); 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-G); 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);

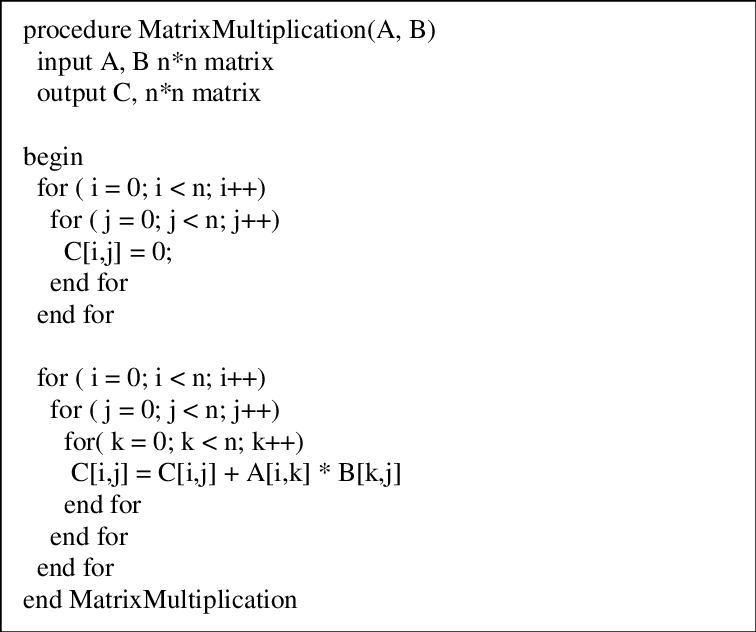
printf("The Following is the %Lfth term of the Fibonacci Sequence:

%Lf",a,fib(a)); end=clock();

double time=(double)(end-start)/CLOCKS\_PER\_SEC; printf("\nExecution Time: %f seconds\n",time);

//print(fib(a));

}



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]);

}

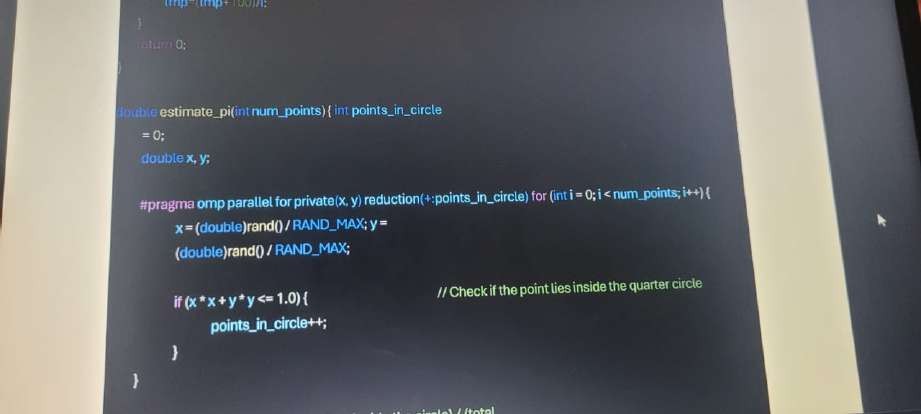
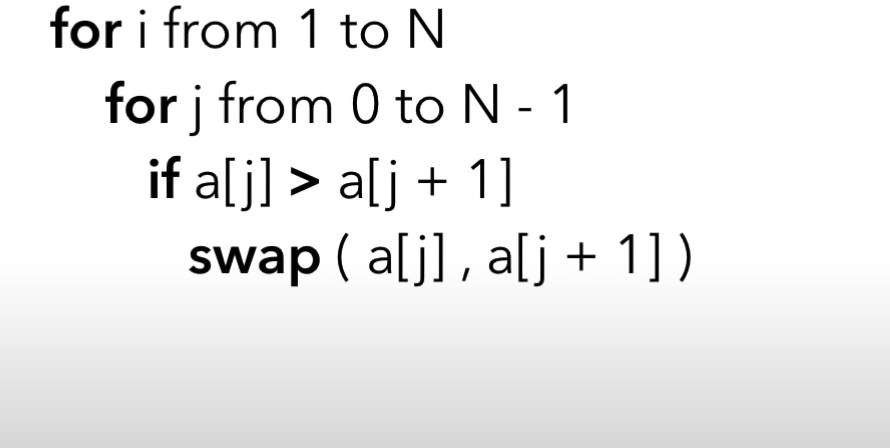
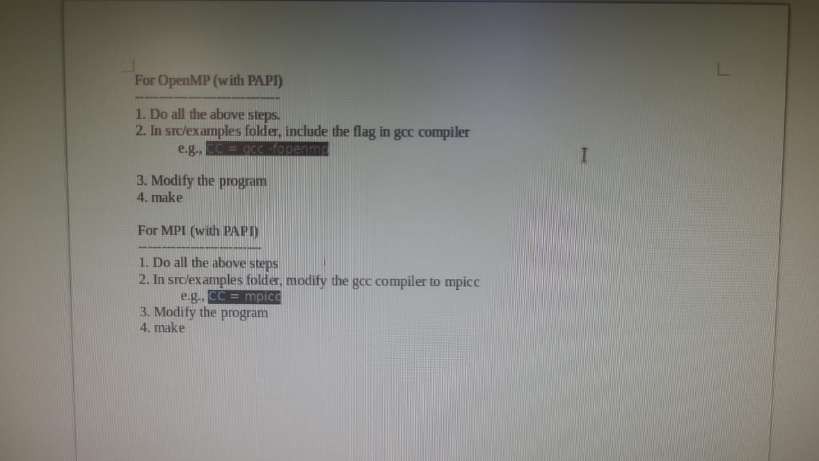
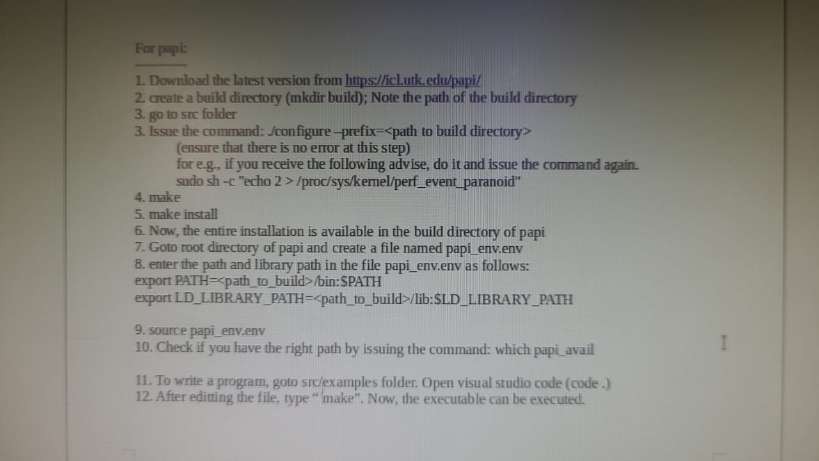
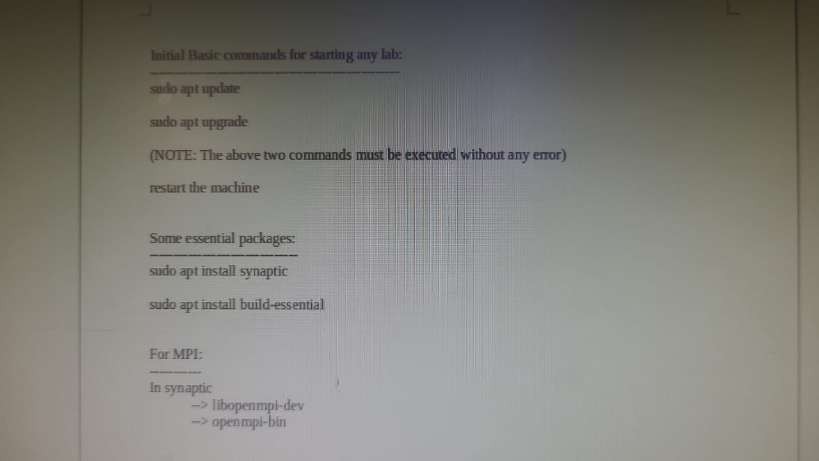
printf("\n");

}

end=clock();

double time=(double)(end-start)/CLOCKS\_PER\_SEC; printf("\nExecution Time: %f seconds\n",time);

}



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 = %.1Gg 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.

