CS 461

Lab Assignment 9

Name: Gandhi Dhruv Vipulkumar

Institute ID: 202151053

Date: 10-11-2024

Q1. Parallel Quicksort Implementation using OpenMP

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

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void parallel_quicksort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

// Parallel regions using OpenMP
```

```
#pragma omp parallel sections
        {
#pragma omp section
            parallel_quicksort(arr, low, pi - 1);

#pragma omp section
            parallel_quicksort(arr, pi + 1, high);
        }
    }
}

int main()
{
    int arr[] = {10, 80, 30, 90, 40, 50, 70};
    int n = sizeof(arr) / sizeof(arr[0]);

// Parallel region with single thread to kick off quicksort
#pragma omp parallel
    {
#pragma omp single
        parallel_quicksort(arr, 0, n - 1);
    }

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

Code Explanation:

1. Swap Function: Swaps two elements in the array.

2. Partition Function: Selects a pivot and rearranges the array so that all elements less than the pivot are on the left and those greater are on the right. It returns the pivot index.

3. Parallel Quicksort Function:

- Recursively sorts the array.

- Uses OpenMP to run the recursive quicksort on the left and right subarrays **in parallel**.
- OpenMP directives (#pragma omp parallel sections and #pragma omp section) parallelize the recursion.

4. **Main Function**:

- Initializes the array, calls parallel quicksort, and prints the sorted array.

Key Features:

- Parallelizes the recursive sorting of the subarrays.
- Improves performance by using multiple threads for different parts of the array.

Testing Phase:

Input: arr[] = {10, 80, 30, 90, 40, 50, 70};

```
PS D:\LAB\Sem 7\CS 401 Parallel Computi
rt_omp.c -o parallel_quicksort_omp } ;
Sorted array: 10 30 40 50 70 80 90
```

Q2. Hyperquicksort Implementation using MPI

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

void quicksort(int *arr, int low, int high);
int partition(int *arr, int low, int high);

int main(int argc, char *argv[])
{
    int rank, size, n = 16, local_n;
    int *arr = NULL, *local_arr = NULL;
    int pivot;

    MPI_Init(&argc, &argv);
```

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

    if (rank == 0)
    {
        // Master process initializes the array
        arr = (int *)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++)
        {
            arr[i] = rand() % 100;
        }
        printf("Unsorted array: ");
        for (int i = 0; i < n; i++)
        {
            printf("%d ", arr[i]);
        }
        printf("\n");
    }
    // Scatter data to all processes
    local_n = n / size;
    local_arr = (int *)malloc(local_n * sizeof(int));
    MPI_Scatter(arr, local_n, MPI_INT, local_arr, local_n, MPI_INT,
0, MPI_COMM_WORLD);

    // Local quicksort
    quicksort(local_arr, 0, local_n - 1);

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

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

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

```
int partition(int *arr, int low, int high)
{
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}
void quicksort(int *arr, int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}
```

Code Explanation:

1. **MPI Initialization**: Initialize MPI and get the rank (ID) and size (total processes).
2. **Data Distribution**: The master process (rank 0) generates an array and distributes portions to all processes using MPI_Scatter.
3. **Local Sorting**: Each process sorts its local subarray using the quicksort() function.
4. **Data Gathering**: After local sorting, the sorted subarrays are gathered back into the master process using MPI_Gather.
5. **Final Output**: The master process prints the partially sorted array.

Testing Phase:

```
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$ mpicc -o hyperquicksort_mpi hyperquicksort_mpi.c
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$ mpirun -np 4 ./hyperquicksort_mpi
Unsorted array: 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26
Partially sorted array: 15 77 83 86 35 86 92 93 21 27 49 62 26 59 63 90
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$
```

Q3. Parallel Sorting by Regular Sampling (PSRS)

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

void quicksort(int *arr, int low, int high);
int partition(int *arr, int low, int high);

int main(int argc, char *argv[]) {
    int rank, size, n = 50;
    int *arr = NULL, *local_arr = NULL;

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

    if (rank == 0) {
        arr = (int *)malloc(n * sizeof(int));
        srand(time(NULL));
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 1000;
        }
    }

    // Distribute data among processes
    int local_n = n / size;
    local_arr = (int *)malloc(local_n * sizeof(int));
    MPI_Scatter(arr, local_n, MPI_INT, local_arr, local_n, MPI_INT,
0, MPI_COMM_WORLD);

    // Sort locally using OpenMP
    #pragma omp parallel
    {
        #pragma omp single
        quicksort(local_arr, 0, local_n - 1);
```

```
    }

    // Gather results and continue sampling and redistribution (full
PSRS logic omitted)
    MPI_Gather(local_arr, local_n, MPI_INT, arr, local_n, MPI_INT,
0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Sorted (partial PSRS logic): ");
        for (int i = 0; i < n; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
        free(arr);
    }

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

void quicksort(int *arr, int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        #pragma omp task shared(arr)
        quicksort(arr, low, pivot - 1);
        #pragma omp task shared(arr)
        quicksort(arr, pivot + 1, high);
        #pragma omp taskwait
    }
}

int partition(int *arr, int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
```

```
    return i + 1;
}
```

Code Explanation:

1. **MPI Initialization**: Initializes MPI and gets the rank and size of the processes.
2. **Array Distribution**: The master process generates an array of random numbers and distributes parts of the array to each process using MPI_Scatter.
3. **Local Sorting**: Each process sorts its local subarray using the quicksort() function, which is parallelized using OpenMP. OpenMP tasks allow parallel execution of subarray sorting.
4. **Gathering Results**: After local sorting, the sorted subarrays are gathered back into the master process using MPI_Gather.
5. **Partial PSRS Logic**: While the code demonstrates partial PSRS logic (local sorting and gathering), the full PSRS process would also involve sampling and redistributing data to ensure global ordering.

Testing Phase:

```
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$ mpicc -o psrs psrs.c
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$ mpirun -np 4 ./psrs
Sorted (partial PSRS logic): 188 341 374 403 434 460 569 605 637 682 684 686 14 345 354 428 441 465 474 479 575 666 820
826 67 89 96 188 385 527 620 644 747 815 901 951 81 104 288 292 465 501 507 508 636 672 686 747 867 565
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$
```

Q4. Calculate the Sum of Large Array

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

#define ARRAY_SIZE 1000000

int main(int argc, char *argv[]) {
    int rank, size, i;
    long long local_sum = 0, global_sum = 0;
    int *array = NULL;
    int local_n;
```

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

    // Allocate array and distribute data among processes
    if (rank == 0) {
        array = (int *)malloc(ARRAY_SIZE * sizeof(int));
        for (i = 0; i < ARRAY_SIZE; i++) {
            array[i] = i + 1;  // Fill array with values 1 to
ARRAY_SIZE
        }
    }

    // Divide the array among processes
    local_n = ARRAY_SIZE / size;
    int *local_array = (int *)malloc(local_n * sizeof(int));

    // Scatter the array to all processes
    MPI_Scatter(array, local_n, MPI_INT, local_array, local_n,
MPI_INT, 0, MPI_COMM_WORLD);

    // Each process computes its partial sum using OpenMP
    #pragma omp parallel for reduction(+:local_sum)
    for (i = 0; i < local_n; i++) {
        local_sum += local_array[i];
    }

    // Reduce all local sums to a global sum
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_LONG_LONG, MPI_SUM,
0, MPI_COMM_WORLD);

    // Display result
    if (rank == 0) {
        printf("Total sum = %lld\n", global_sum);
        free(array);
    }

    // Clean up
    free(local_array);
    MPI_Finalize();

    return 0;
}
```

Code Explanation:

1. **MPI Initialization**:
   - Initializes MPI with MPI_Init, gets the rank and size of the processes using MPI_Comm_rank and MPI_Comm_size.
2. **Array Initialization and Distribution**:
   - The master process (rank == 0) creates an array of size ARRAY_SIZE and fills it with values from 1 to ARRAY_SIZE.
   - The array is divided among all processes. Each process gets a part of the array, and this is done using MPI_Scatter.
3. **Local Sum Computation**:
   - Each process computes the sum of its local portion of the array using a parallelized for loop with **OpenMP**. The reduction(+:local_sum) clause ensures that each thread's local sum is added to the local_sum variable safely.
4. **Global Sum Calculation**:
   - After computing the local sum, the process uses MPI_Reduce to gather all local sums from the processes and compute the global sum on the master process (rank == 0).
5. **Final Output**:
   - The master process prints the total sum of the array.
6. **Cleanup**:
   - Memory allocated for the arrays is freed, and MPI_Finalize is called to end the MPI session.

Testing Phase:

```
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$ mpicc -o large_arr_sum large_arr_sum.c
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$ mpirun -np 4 ./large_arr_sum
Total sum = 500000500000
dhruv@LAPTOP-LB6QPGHC:~/Parallel Computing/lab 9$
```

Conclusion: The above four codes cover real-life implementation on OpenMP and MPI libraries.