

PROJECT PROBLEM DESCRIPTION

Amisha Gunderia-201071042

Srushti Gujjanwar-201071073

Problem:

Comparison between hyperquicksort and parallel quick sort using OpenMP and CUDA.

Differences

- **Algorithmic Approach:**

Hyperquicksort: Hyperquicksort is an algorithmic improvement over traditional quicksort that uses a multi-pivot approach. Instead of selecting a single pivot element, it selects multiple pivot elements (typically three or five) to partition the array. This allows for more efficient partitioning and reduces the number of comparisons needed.

Parallel Quicksort with OpenMP: Parallel quicksort using OpenMP, as shown in the code example provided, applies parallelism to the traditional quicksort algorithm. It divides the array into smaller subarrays and sorts them independently in parallel using tasks. The parallelism is achieved by creating tasks for sorting subarrays and letting multiple threads execute these tasks simultaneously.

- **Parallelism:**

Hyperquicksort: Hyperquicksort can be implemented in a parallel fashion by distributing the partitioning and sorting tasks among multiple threads. Each thread can work on a different pivot and sort its corresponding partition concurrently.

Parallel Quicksort with OpenMP: Parallel quicksort using OpenMP explicitly utilizes OpenMP directives and task-based parallelism to distribute the sorting tasks among multiple threads. The algorithm creates tasks for sorting subarrays, and the OpenMP runtime system schedules these tasks across the available threads.

- Dependencies and Communication:

Hyperquicksort: Hyperquicksort has inherent dependencies between the pivot selection and partitioning steps. If the algorithm is parallelized, it requires careful synchronization and communication between threads to ensure correct partitioning and sorting.

Parallel Quicksort with OpenMP: The parallel quicksort using OpenMP in the provided code example does not have explicit dependencies between tasks because each task operates on a separate subarray. However, data dependencies may arise if there are shared variables that need to be accessed and modified concurrently. In such cases, appropriate synchronization mechanisms like `#pragma omp critical` or `#pragma omp atomic` can be used to ensure thread safety.

- Implementation Complexity:

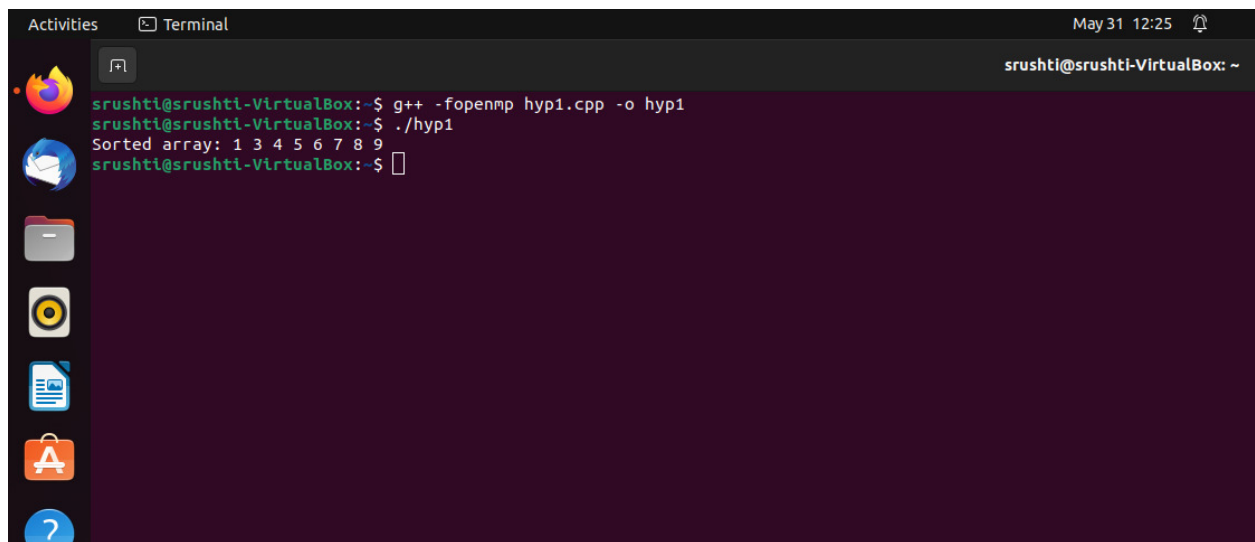
Hyperquicksort: Implementing hyperquicksort and handling the complexities of multi-pivot partitioning can be more challenging compared to the traditional quicksort algorithm. It requires careful consideration of pivot selection, partitioning logic, and handling dependencies in a parallel setting.

Parallel Quicksort with OpenMP: Implementing parallel quicksort using OpenMP is relatively simpler as it leverages the task-based

parallelism provided by OpenMP. The code can be structured using OpenMP directives like `#pragma omp parallel` and `#pragma omp task` to distribute the sorting tasks among threads. However, ensuring proper synchronization and avoiding data races may require additional attention

OUTPUT

Parallel Quicksort

A screenshot of a Linux terminal window titled 'Terminal' with a date and time of 'May 31 12:25'. The terminal shows the user 'srushti@srushti-VirtualBox' in the home directory. The user enters the command 'g++ -fopenmp hyp1.cpp -o hyp1' to compile a C++ program. Then, they enter './hyp1' to run the program. The output of the program is 'Sorted array: 1 3 4 5 6 7 8 9'. The terminal window has a sidebar on the left with various application icons like Firefox, Files, and the Dash icon.

```
srushti@srushti-VirtualBox:~$ g++ -fopenmp hyp1.cpp -o hyp1
srushti@srushti-VirtualBox:~$ ./hyp1
Sorted array: 1 3 4 5 6 7 8 9
srushti@srushti-VirtualBox:~$
```

Hyper Quicksort

Parallel Qui x pc_mini_pro x Parallel-K-h x Parallel Qui x hyperquick: x R (PDF) Parali x Upload you x WhatsApp x Google Doc x Untitled do x +

colab.research.google.com/drive/1sccuYd0SXmKRQWA4_tVRf3c0WqRok-sy

Maps YouTube Gmail

hyperquicksort (new).ipynb

Comment Share

File Edit View Insert Runtime Tools Help Last saved at 11:54

+ Code + Text

Connect ^

{x}

```
cuda
{
    n_t = 1024;
    n_b = answer/n_t + (answer%n_t==0?0:1);
}
n_i = answer;
cudaMemcpy(arr, d_d, (h-1+1)*sizeof(int), cudaMemcpyDeviceToHost);
}

int main()
{
    int arr[10];
    srand(time(NULL));
    for (int i = 0; i<10; i++)
    {
        arr[i] = rand ()%10000;
    }

    int n = sizeof( arr ) / sizeof( *arr );
    printf("The array is:\n");
    printArr( arr, n );
    quickSortIterative( arr, 0, n - 1 );
    printf("\nThe sorted array is:\n");
    printArr( arr, n );
    return 0;
}
```

<>

The array is:

1746 1580 6872 7491 6026 7924 1803 942 3388 8779

The sorted array is:

942 1580 1746 1803 3388 6026 6872 7491 7924 8779