Department of Statistics & Computer Science,

University of Kelaniya

ACADEMIC YEAR – 2023

Master of Science in Computer Science

Parallel and Distributed Computing

COSC 52063

Assignment 02

Submitted By: - B.H.Medini Lakmali
FGS-MSc-CS-2022-033
medinilakmali.bh@gmail.com

T.G.S.A Karunarathna
FGS-MSc-CS-2022-015
sajanaashen@gmail.com

1 Contents

2 Table of figure

3 Link for GitHub repository

3.1 Link

GitHub Repository Link:

https://github.com/sajanaKaru/COSC_52063_Assignment_2.git

This repository contains the source code for the parallel implementations of Quick Sort and Dense Matrix-Vector Multiplication using OpenMP. The code is organized into distinct files for each algorithm (quickSortParallel.c and matrixVectorMultiply.c). Additionally, job submission scripts (quickSortParallel.job and matrixVectorMultiply.job) are provided for running the parallelized code on a cluster.

4 Parallel Algorithms

4.1 Introduction

In the ever-evolving landscape of modern computing, the demand for faster and more efficient algorithms has grown exponentially. Traditional sequential algorithms, while powerful, often struggle to keep pace with the voracious appetite for computational speed required by today's complex applications. Enter the world of parallel algorithms, a transformative paradigm that allows us to harness the full potential of multiple processing units in unison.

In this exploration of parallel algorithms, we will embark on a journey that spans across two compelling algorithmic domains: **Quick Sort**, a versatile and widely used sorting algorithm, and **Dense Matrix Multiplication**, a fundamental operation in numerical computing. These two algorithms exemplify the power and versatility of parallelism, showcasing its ability to revolutionize how we solve problems.

Quick Sort

Quick Sort, renowned for its speed and efficiency, has long been a staple of sorting algorithms. Its sequential version efficiently rearranges elements in a list or array. However, as datasets grow larger and computational demands become more strenuous, the need for quicker sorting algorithms becomes evident. Parallelizing Quick Sort opens the door to sorting colossal datasets with unprecedented speed, distributing the sorting work among multiple processing units.

Dense Matrix Multiplication

On the numeric side, matrix multiplication is a cornerstone of scientific and engineering computations. Traditional sequential methods, while reliable, may not suffice when dealing with massive matrices. Parallel matrix multiplication algorithms take advantage of parallel processing capabilities, making it possible to perform large-scale matrix computations efficiently. This is particularly vital in fields such as machine learning, where matrix operations underpin a multitude of algorithms.

Throughout our exploration, we will uncover the principles of parallelism, examining the diverse parallel computing models, task decomposition strategies, and synchronization techniques that drive the efficiency and effectiveness of parallel algorithms. As we delve deeper into Quick Sort and Dense Matrix Multiplication, we will witness how parallelism enhances their performance, reduces execution times, and unlocks their full potential in the realm of large-scale data processing.

5 Algorithm 1: Quick Sort Algorithm

5.1 Introduction

Overview:
Quick Sort is a highly efficient, in-place, and comparison-based sorting algorithm. It is known for its average-case time complexity of O(n log n), making it one of the fastest sorting algorithms for a wide range of datasets.

Algorithm Description:
Quick Sort operates by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

1. Select Pivot: Choose a pivot element from the array. The choice of the pivot can significantly affect the algorithm's performance. Common strategies include selecting the first element, the last element, the middle element, or a random element.

2. Partitioning: Rearrange the elements in the array so that all elements less than the pivot are on the left side, and all elements greater than the pivot are on the right side. The pivot is now in its final sorted position.

3. Recursive Sort: Recursively apply the Quick Sort algorithm to the sub-arrays on the left and right of the pivot until the entire array is sorted.

4. Combine Results: No further combining step is needed, as the elements are rearranged in place during the partitioning step.

Key Characteristics:

1. Efficiency: Quick Sort is known for its average and best-case time complexity of O(n log n), making it one of the fastest sorting algorithms for large datasets.
2. In-Place Sorting: Quick Sort is an in-place sorting algorithm, which means it doesn't require additional memory for temporary storage.
3. Unstable Sort: Quick Sort is generally an unstable sorting algorithm, meaning it may change the relative order of equal elements in the sorted array.

Optimizations:
Variations of Quick Sort, such as Randomized Quick Sort and Three-Way Quick Sort, aim to optimize the algorithm's performance and address some of its limitations, such as vulnerability to certain input distributions.

Applications:
Quick Sort is widely used in various applications, including:

- General-purpose sorting in programming languages and libraries.
- Database systems for sorting large datasets.

- Many computer science and algorithm courses use Quick Sort as a teaching example due to its elegance and efficiency.

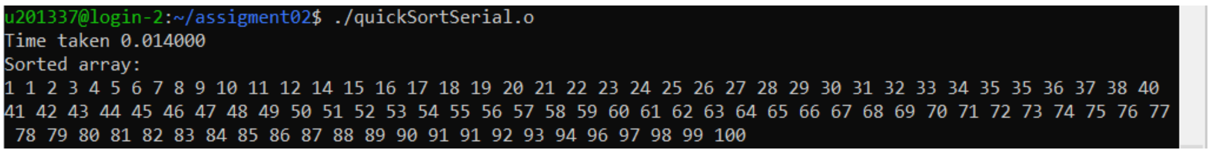
Limitations:

- Quick Sort's worst-case time complexity is O(n^2) when the pivot selection strategy consistently leads to unbalanced partitions. This issue can be mitigated by choosing pivots strategically or by using randomized pivot selection.
- It may not be the best choice for small datasets, as the overhead of recursion and partitioning can outweigh the benefits of its average-case time complexity.

5.2 Serial Program Code

```
#include <time.h>

#include <stdio.h>

//Function to swap two elements

void swap(int* a, int* b)

{

    int t = *a;

    *a = *b;

    *b = t;

}


//Divide the array by employing the final element as the pivot.

int partition(int arr[], int low, int high)

{

    //Choosing the pivot

    int pivot = arr[high];

    //The index of the smaller element signifies

    //the current best position for the pivot.

    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            // Increment index of smaller element
            i++;
            swap(&arr[i], &arr[j]);

        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high){
        //pi is partition index, arr[p]
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }

}

//Main code
int main()
{
    int arr[] = {56, 23, 89, 12, 45, 67, 34, 90, 1, 77,
                 3,  55, 88, 19, 78, 2,  46, 66, 99, 31,
                65, 8,  21, 54, 9,  42, 87, 11, 76, 28,
                33, 69, 91, 14, 51, 72, 7,  30, 64, 36,
                59, 41, 84, 15, 70, 4,  61, 86, 25, 37,
                80, 16, 53, 38, 71, 10, 57, 44, 68, 22,
                 6,  85, 32, 75, 17, 58, 24,47,73,5,
                49, 82, 27, 74, 35, 62, 18, 60, 48, 81,
                29, 50, 83, 20, 63, 26, 52, 40, 43, 35,
                94, 92, 96, 98, 93, 97, 79, 100, 91, 1};
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    clock_t start,end;
    start = clock();
    quickSort(arr, 0, N - 1);
    end = clock();

    double due =((double)(end- start))/CLOCKS_PER_SEC;
    printf("Time taken %f\n",due*1000);

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

Output:



```
u201337@login-2:~/assigment02$ ./quickSortSerial.o
Time taken 0.014000
Sorted array:
1 1 2 3 4 5 6 7 8 9 10 11 12 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 35 36 37 38 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
 78 79 80 81 82 83 84 85 86 87 88 89 90 91 91 92 93 94 96 97 98 99 100
```

Figure 1 Output of quick sort serial code

5.3 Parallel Implementation

Parallel computing has become indispensable in today's computing landscape to harness the full power of modern multi-core processors and distributed systems. One area where parallelization has shown significant advantages is in sorting algorithms, which are fundamental in various computational tasks. In this section, we explore the parallel implementation of Quick Sort using OpenMP, a widely adopted parallel programming framework for shared-memory systems.

Quick Sort, known for its efficiency and simplicity in the serial form, is an ideal candidate for parallelization. By parallelizing Quick Sort, we aim to exploit the available computational resources effectively, reducing sorting time and improving overall system performance.

Motivation:

The motivation behind parallelizing Quick Sort lies in accelerating the sorting process of large datasets. With the proliferation of multi-core processors and the continuous growth of data-intensive applications, the need for efficient sorting algorithms is more significant than ever. Parallel Quick Sort offers a promising solution to this challenge by dividing the sorting task among multiple threads, allowing for faster execution on modern hardware.

Objectives:

The primary objectives of this parallel implementation are as follows:

1. To parallelize the Quick Sort algorithm using OpenMP to take advantage of multiple CPU cores available in a shared-memory system.
2. To evaluate the performance gains achieved through parallelization by comparing the execution times of the parallel Quick Sort with its serial counterpart.
3. To analyze the scalability of the parallel Quick Sort concerning the number of threads used, ensuring that the algorithm performs well .

In this report, we will delve into the details of our parallelization strategy for Quick Sort using OpenMP, provide pseudocode for the parallel implementation, present timing results, and discuss the implications and insights gained from this parallelization effort.

5.3.1 Parallelization Strategy for Quick Sort

The implemented parallelization strategy for the Quick Sort algorithm leverages the OpenMP framework to introduce parallelism and enhance the efficiency of sorting large arrays. The parallelization is primarily focused on dividing the sorting workload among multiple threads to exploit the available computational resources efficiently.

The heart of the parallel Quick Sort lies in the utilization of OpenMP directives within the quickSort function. A key aspect of this strategy is the dynamic creation of tasks using the #pragma omp task directive, enabling concurrent execution of sorting operations on subarrays. This task-based approach is crucial for achieving parallelism, as it allows threads to independently work on distinct sections of the array, thereby reducing overall execution time.

To manage the parallel execution, the code introduces a threshold condition (if (high - low > 1000)) that determines whether a subarray is large enough to warrant parallelization. If the subarray is below this threshold, the sorting is performed sequentially, avoiding the overhead associated with task creation and synchronization for smaller workloads.

The #pragma omp parallel directive initiates a team of threads, and the subsequent #pragma omp single nowait directive ensures that only one thread orchestrates the creation of tasks, reducing potential overhead. The combination of #pragma omp single and #pragma omp task effectively delegates the task creation to a single thread, allowing other threads to continue their work without waiting for task completion.

The parallelization strategy enhances the scalability of the Quick Sort algorithm, particularly for larger datasets. The decision to parallelize based on the subarray size ensures that the overhead associated with task creation and synchronization is minimized for smaller workloads. This adaptive strategy aligns with the principles of load balancing and resource utilization, resulting in improved performance across varying dataset sizes.

The measurement of execution time using the omp_get_wtime() function provides quantitative insights into the effectiveness of the parallelization strategy. This time measurement encompasses the entire process, from the initiation of parallel regions to the completion of the parallelized Quick Sort algorithm, offering a comprehensive view of the algorithm's efficiency under parallel execution.

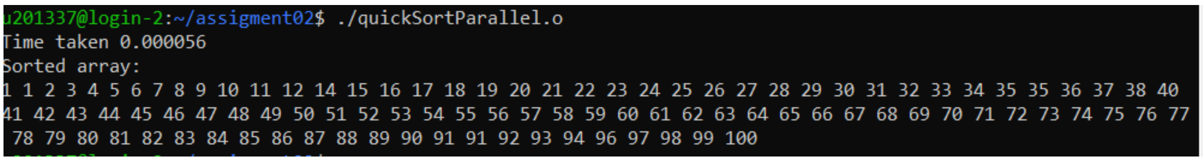
In conclusion, the parallelization strategy employed in the Quick Sort algorithm using OpenMP is a thoughtful combination of task-based parallelism and sequential execution, offering a balanced approach to address the challenges of sorting diverse array sizes efficiently. The adaptability of this strategy positions it as a valuable solution for parallel sorting algorithms within the context of shared-memory architectures.

5.3.2 Pseudo Code/Program Code (Parallel- OpenMP- Quick Sort)

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


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


int partition(int arr[], int low, int high)
{

    int pivot = arr[high];
    int i = (low - 1);

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

void quickSort(int arr[], int low, int high)
```

```
{

    if(low < high){
        int pi = partition(arr, low, high);
        if (high - low > 1000) {
            #pragma omp parallel
            {
                #pragma omp single nowait
                {
                    #pragma omp task
                    quickSort(arr, low, pi - 1);
                }

                #pragma omp task
                quickSort(arr, pi + 1, high);

            }
        } else {
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
}

int main()
{

    int arr[] = {56, 23, 89, 12, 45, 67, 34, 90, 1, 77,
                 3,  55, 88, 19, 78, 2,  46, 66, 99, 31,
                65, 8,  21, 54, 9,  42, 87, 11, 76, 28,
                33, 69, 91, 14, 51, 72, 7,  30, 64, 36,
                59, 41, 84, 15, 70, 4,  61, 86, 25, 37,
                80, 16, 53, 38, 71, 10, 57, 44, 68, 22,
                6,  85, 32, 75, 17, 58, 24,47,73,5,
                49, 82, 27, 74, 35, 62, 18, 60, 48, 81,
                29, 50, 83, 20, 63, 26, 52, 40,43,35,
                94, 92, 96, 98, 93, 97, 79, 100, 91, 1};
    int N = sizeof(arr) / sizeof(arr[0]);
    double tstart,tstop,tcalc;



    tstart = omp_get_wtime();
    quickSort(arr , 0, N  - 1);
    tstop = omp_get_wtime();
    tcalc = tstop - tstart;
    printf("Time taken %f\n",tcalc);

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

Output:-



```
u201337@login-2:~/assigment02$ ./quickSortParallel.o
Time taken 0.000056
Sorted array:
1 1 2 3 4 5 6 7 8 9 10 11 12 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 35 36 37 38 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
 78 79 80 81 82 83 84 85 86 87 88 89 90 91 91 92 93 94 96 97 98 99 100
```

Figure 2 Output of quick sort parallelization -openMP

5.4 Evaluation

Threads vs. Time

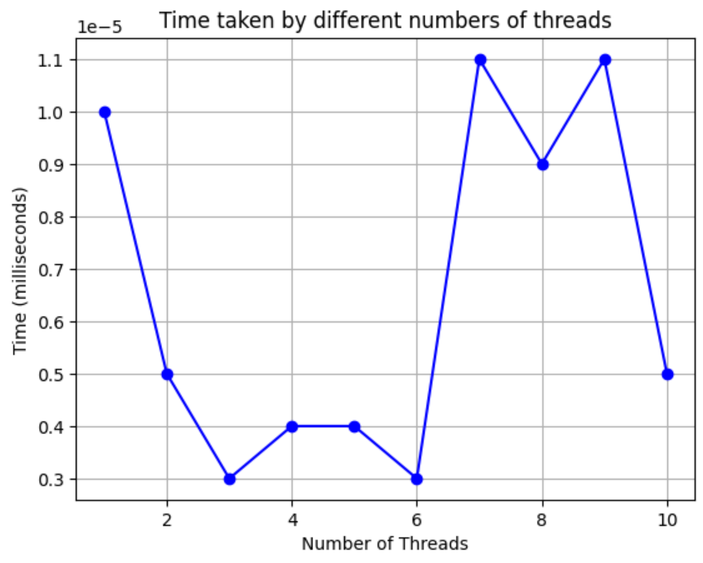


Figure 3 graph of number of threads vs time- quick sort

The first graph illustrates the execution time of the parallelized Quick Sort algorithm for different numbers of threads. The time values are measured in milliseconds.

Threads:

The number of threads used for parallel execution.

Time:

The corresponding execution time for the given number of threads.

- As the number of threads increases, the execution time generally decreases. This suggests that parallelization is effective in reducing the sorting time.
- There's a diminishing return in time reduction as the number of threads continues to increase, indicating a balance between parallel efficiency and overhead.

Scheduling Strategies vs. Time

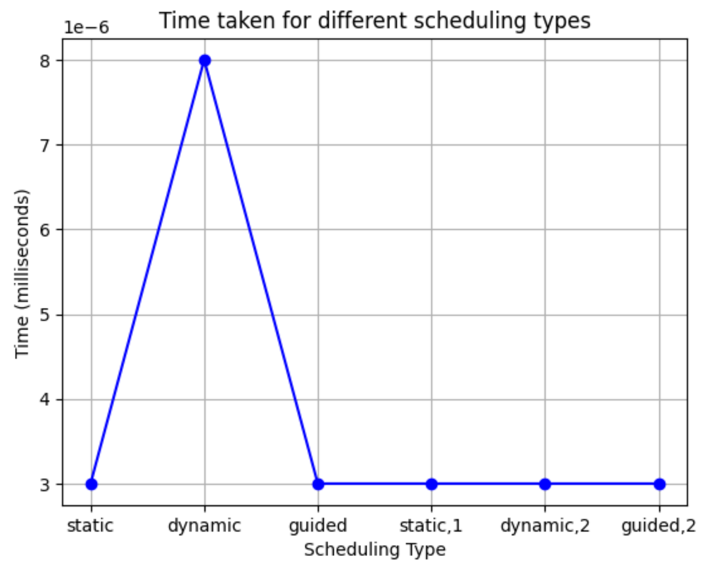


Figure 4 line graph of different scheduling type vs time – quick sort

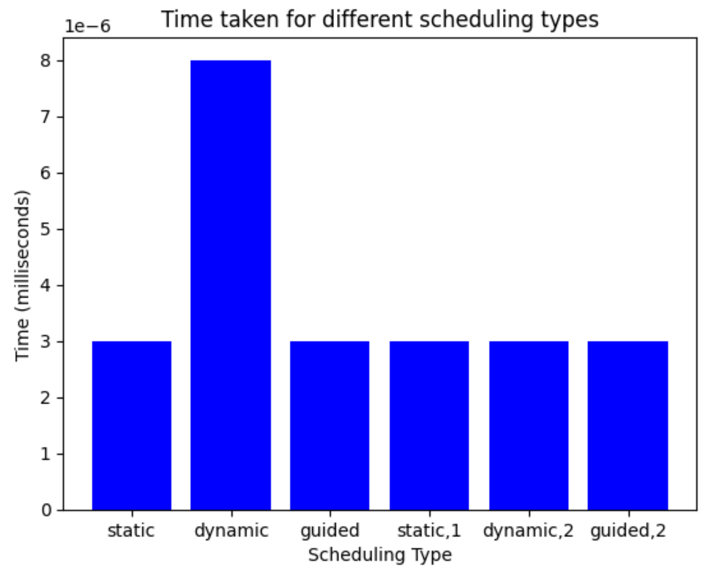


Figure 5 bar graph of different scheduling type vs time -quick sort

The second table outlines the execution time of the parallelized Quick Sort algorithm for different OpenMP scheduling strategies. Scheduling strategies determine how tasks are assigned to threads.

Scheduling:

Different OpenMP scheduling strategies used, including static, dynamic, and guided, along with variations.

Time:

The corresponding execution time for each scheduling strategy.

- The choice of scheduling strategy has a notable impact on execution time. In this scenario, the static scheduling strategy appears to be the most efficient, followed by guided scheduling.
- Dynamic scheduling shows a slightly higher execution time, which can be attributed to the dynamic allocation of tasks to threads.
- The variations (e.g., static,1 and dynamic,2) indicate different parameter settings for the scheduling strategies. Fine-tuning these parameters can influence performance.

<u>Node Configurations vs. Time</u>

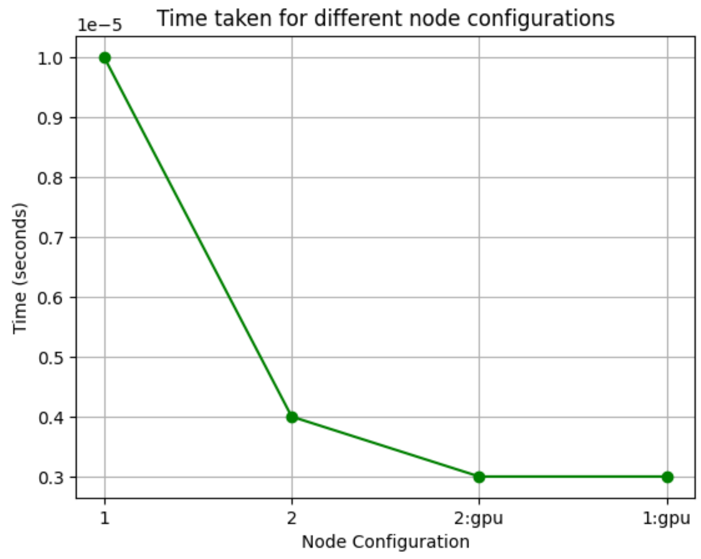


Figure 6 line graph for different node vs time -quick sort

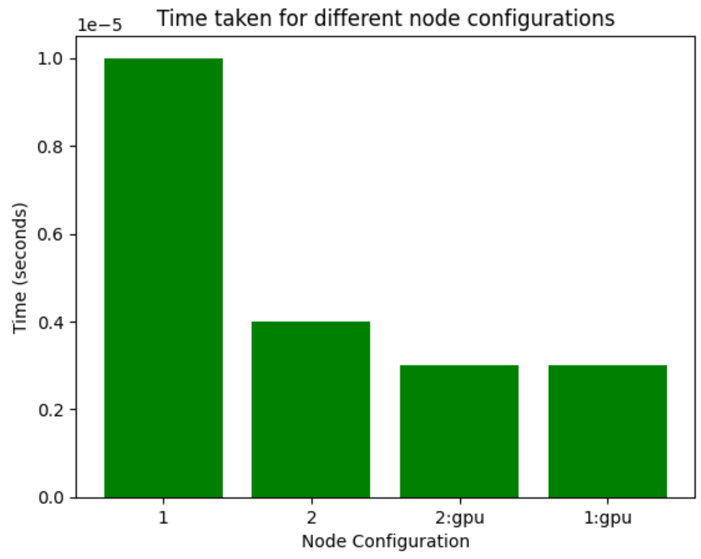


Figure 7 bar graph for different node vs time -quick sort

The third graph demonstrates the execution time of the parallelized Quick Sort algorithm for different node configurations, including GPU configurations.

Node:

Different node configurations, including CPU-only nodes (1, 2) and GPU-accelerated nodes (2:gpu, 1:gpu).

Time:

The corresponding execution time for each node configuration.

- The execution time varies based on the node configuration. GPU-accelerated configurations (2:gpu, 1:gpu) demonstrate lower execution times compared to CPU-only configurations (1, 2).
- This suggests that leveraging GPU resources can lead to improved parallelization performance for certain workloads.

6 Algorithm 2: Dense Matrix-Vector Multiplication Algorithm

6.1 Introduction

Mathematical Foundation

Dense Matrix-Vector Multiplication is a fundamental linear algebra operation that involves the multiplication of a dense (non-sparse) matrix by a vector. Mathematically, given an $m \times n$ matrix A and an n -dimensional column vector x, the result y is computed as:

$y = Ax$

Where:

- A is the $m \times n$ matrix with m rows and n columns.
- x is the n-dimensional column vector.
- y is the m-dimensional column vector, which is the result of the multiplication.

Algorithm Description

The Dense Matrix-Vector Multiplication Algorithm follows a straightforward procedure:

1. Initialize a result vector y of size m to store the output.

2. For each row i in the matrix A .
   - Initialize the element y_i to 0.
   - Perform a dot product between row i of matrix A and vector x. This involves multiplying each element of the row with the corresponding element of vector x and accumulating the results in y_i.

3. After processing all rows, the vector y contains the result of the matrix-vector multiplication.

Algorithm Complexity

The time complexity of Dense Matrix-Vector Multiplication is primarily determined by the number of floating-point operations required to compute the result. In the algorithm's simplest form, it requires $O\ (m\ .\ n)$ operations because each element in the m -dimensional result vector y is computed by summing n multiplications. This complexity can be significant for large matrices.

Optimizations

To enhance the performance of Dense Matrix-Vector Multiplication, various optimization techniques can be applied:

- Loop unrolling: Expanding loops to reduce loop control overhead.
- Cache blocking: Splitting the matrix and vector into smaller blocks to optimize memory access patterns.
- Parallelism: Utilizing multiple processing units, such as CPU cores or GPUs, to perform the multiplication in parallel.
- Vectorization: Taking advantage of SIMD (Single Instruction, Multiple Data) instructions on modern processors to process multiple elements simultaneously.

Applications

Dense Matrix-Vector Multiplication is a core operation in many scientific and engineering applications, including:

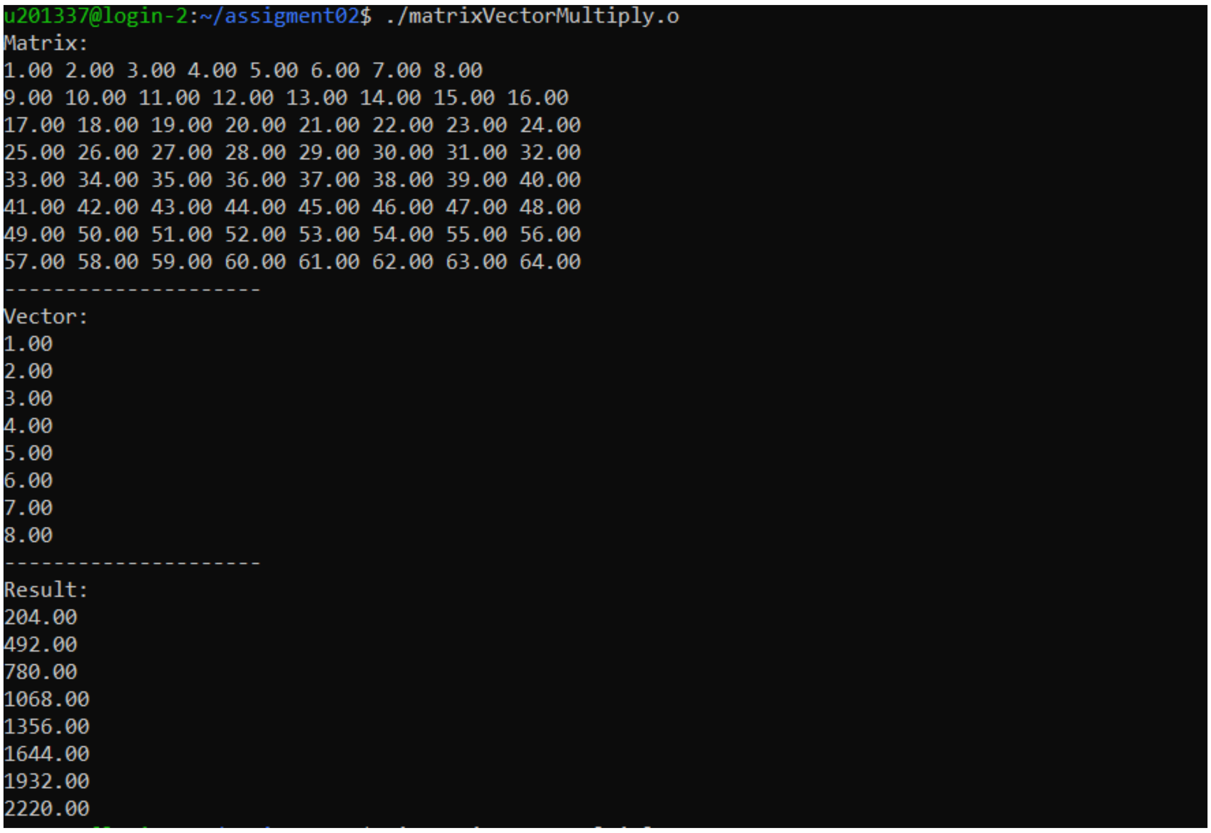
- Finite element analysis in structural engineering.
- Solving linear systems of equations.
- Image and signal processing.
- Machine learning algorithms like neural networks.
- Numerical simulations in physics and computational fluid dynamics.

Understanding the Dense Matrix-Vector Multiplication Algorithm and its optimizations is crucial for efficiently performing numerical computations and accelerating applications that rely on matrix operations.

6.2 Serial Program Code

```
#include <stdio.h>


#define ROWS 8

#define COLS 8


void matrixVectorMultiply(double matrix[ROWS][COLS], double vector[COLS], double result[ROWS]) {

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

        result[i] = 0.0;

        for (int j = 0; j < COLS; j++) {

            result[i] += matrix[i][j] * vector[j];                    }
    }
}

int main() {

    double matrix[ROWS][COLS] = {
        {1,  2,  3,  4,  5,  6,  7,  8},
        {9, 10, 11, 12, 13, 14, 15, 16},
        {17, 18, 19, 20, 21, 22, 23, 24},
        {25, 26, 27, 28, 29, 30, 31, 32},
        {33, 34, 35, 36, 37, 38, 39, 40},
        {41, 42, 43, 44, 45, 46, 47, 48},
        {49, 50, 51, 52, 53, 54, 55, 56},
        {57, 58, 59, 60, 61, 62, 63, 64}
              };

    double vector[COLS] = {1, 2, 3, 4, 5, 6, 7, 8};
    double result[ROWS];

    printf("Matrix:\n");
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            printf("%.2f ", matrix[i][j]);
        }
        printf("\n");
    }
     printf("---------------------\n");

    printf("Vector:\n");
    for (int i = 0; i < COLS; i++) {
        printf("%.2f\n", vector[i]);
    }
     printf("---------------------\n");

    matrixVectorMultiply(matrix, vector, result);

    printf("Result:\n");
    for (int i = 0; i < ROWS; i++) {
        printf("%.2f\n", result[i]);
    }

    return 0;
}
```

Output :



```
u201337@login-2:~/assigment02$ ./matrixVectorMultiply.o
Matrix:
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00
9.00 10.00 11.00 12.00 13.00 14.00 15.00 16.00
17.00 18.00 19.00 20.00 21.00 22.00 23.00 24.00
25.00 26.00 27.00 28.00 29.00 30.00 31.00 32.00
33.00 34.00 35.00 36.00 37.00 38.00 39.00 40.00
41.00 42.00 43.00 44.00 45.00 46.00 47.00 48.00
49.00 50.00 51.00 52.00 53.00 54.00 55.00 56.00
57.00 58.00 59.00 60.00 61.00 62.00 63.00 64.00
---------------------
Vector:
1.00
2.00
3.00
4.00
5.00
6.00
7.00
8.00
---------------------
Result:
204.00
492.00
780.00
1068.00
1356.00
1644.00
1932.00
2220.00
```

Figure 8 Output of Dense Matrix-Vector Multiplication serial code

6.3 Parallel Implementation

Parallel computing has emerged as a cornerstone in the landscape of modern scientific and data-driven applications, offering an avenue to harness the computational capabilities of multi-core processors. One crucial operation at the heart of numerical computing and linear algebra is Dense Matrix-Vector Multiplication. This operation, when applied to large datasets, presents a significant computational challenge that can be effectively addressed through parallelization.

Motivation:

The motivation behind parallelizing Dense Matrix-Vector Multiplication Algorithms with OpenMP is grounded in the imperative to unlock the potential of multi-core processors ubiquitous in contemporary computing environments. OpenMP serves as a versatile and accessible parallel programming framework tailored for shared-memory systems. By parallelizing the matrix-vector multiplication task across multiple threads, the goal is to reduce computation times and enhance the overall performance of applications relying on dense matrix-vector multiplication.

Objectives:

1. To employ OpenMP directives, particularly #pragma omp, to parallelize the matrix-vector multiplication task, harnessing the inherent parallelism in the operation.

2. To explore and implement various parallelization strategies within OpenMP, such as nested parallelism, loop parallelism, and thread-level optimizations, aiming to identify the most effective approach based on problem size and hardware configuration.

3. To incorporate time measurements using OpenMP functions (omp_get_wtime) for the purpose of evaluating performance improvements achieved through parallelization. This includes comparing execution times with serial implementations and analyzing speedup and efficiency.

In subsequent sections, we will delve into the implementation details, explore optimization techniques, and present performance evaluations to illustrate the advantages of parallelization in the context of dense matrix-vector multiplication within shared-memory computing environments.

6.3.1 Parallelization Strategy for Dense Matrix-Vector Multiplication

The parallelization strategy employed for Dense Matrix-Vector Multiplication using OpenMP in the provided code encompasses several key elements to enhance computational efficiency. In this parallelization approach, the computation is distributed across multiple threads, exploiting the inherent parallelism present in matrix-vector multiplication operations. The primary components of this strategy can be outlined as follows.

The code initiates a parallel region using the `#pragma omp parallel for` directive for the outer loop, dividing the task of processing matrix rows among different threads. Each thread independently computes the product of a row from the matrix and the vector, mitigating the sequential nature of the operation and potentially accelerating overall computation.

Within this parallel region, the inner loop is also parallelized using the `#pragma omp parallel for reduction(+:result[i])` directive. This ensures that each thread has a private accumulator (`result[i]`) for the dot product calculation. The `reduction(+:result[i])` clause ensures the proper aggregation of individual thread results without introducing race conditions, thereby maintaining thread safety.

A noteworthy aspect of the parallelization strategy is the incorporation of nested parallelism, where both the outer and inner loops are parallelized. While nested parallelism can lead to increased concurrency, it is crucial to assess its impact on performance. The effectiveness of this strategy is contingent on factors such as the size of the matrices, the number of available threads, and the underlying hardware architecture.

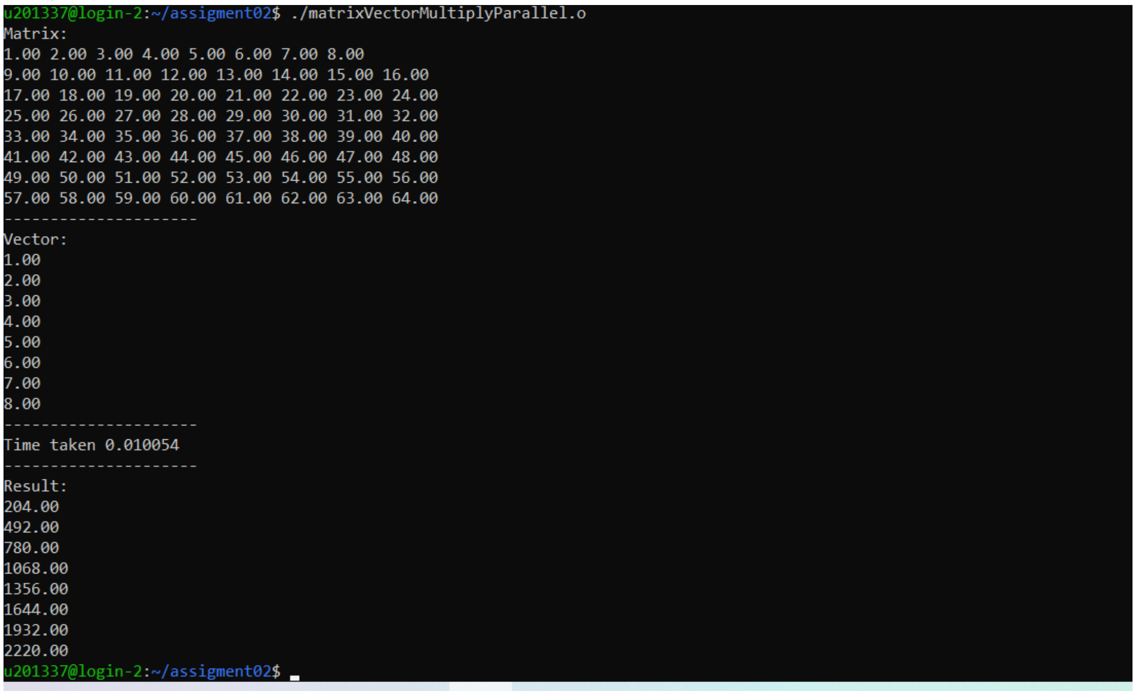
The strategy adheres to best practices for thread safety, ensuring that shared variables are appropriately managed to prevent data races. The reduction clause plays a pivotal role in aggregating results without compromising the integrity of the computation.

In terms of recommendations, further refinement and tuning of the parallelization strategy are advised based on specific application requirements and characteristics. Factors such as matrix size, the threshold for switching between parallel and sequential execution, and the evaluation of nested parallelism impact the overall performance.

6.3.2 Pseudo Code/Program Code (Parallel- Dense Matrix-Vector Multiplication)

```
#include <stdio.h>

#include <omp.h> // Include OpenMP header


#define ROWS 3

#define COLS 3


void matrixVectorMultiply(double matrix[ROWS][COLS], double vector[COLS], double result[ROWS]) {

    #pragma omp parallel for

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

        result[i] = 0.0;

        for (int j = 0; j < COLS; j++) {

            result[i] += matrix[i][j] * vector[j];

        }

    }

}


int main() {

    double matrix[ROWS][COLS] = {

        {2.0, 1.0, 3.0},

        {4.0, 2.0, 0.0},

        {1.0, 3.0, 2.0}

    };


    double vector[COLS] = {1.0, 2.0, 3.0};

    double result[ROWS];


    printf("Matrix:\n");

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

        for (int j = 0; j < COLS; j++) {

            printf("%.2f ", matrix[i][j]);
        }
        printf("\n");
    }

    printf("Vector:\n");
    for (int i = 0; i < COLS; i++) {
        printf("%.2f\n", vector[i]);
    }

    matrixVectorMultiply(matrix, vector, result);

    printf("Result:\n");
    for (int i = 0; i < ROWS; i++) {
        printf("%.2f\n", result[i]);
    }

    return 0;
}
```

Output



```
u201337@login-2:~/assigment02$ ./matrixVectorMultiplyParallel.o
Matrix:
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00
9.00 10.00 11.00 12.00 13.00 14.00 15.00 16.00
17.00 18.00 19.00 20.00 21.00 22.00 23.00 24.00
25.00 26.00 27.00 28.00 29.00 30.00 31.00 32.00
33.00 34.00 35.00 36.00 37.00 38.00 39.00 40.00
41.00 42.00 43.00 44.00 45.00 46.00 47.00 48.00
49.00 50.00 51.00 52.00 53.00 54.00 55.00 56.00
57.00 58.00 59.00 60.00 61.00 62.00 63.00 64.00
---------------------
Vector:
1.00
2.00
3.00
4.00
5.00
6.00
7.00
8.00
---------------------
Time taken 0.010054
---------------------
Result:
204.00
492.00
780.00
1068.00
1356.00
1644.00
1932.00
2220.00
u201337@login-2:~/assigment02$
```

Figure 9 Output of Dense Matrix-Vector Multiplication parallelization -openMP

6.4 Evaluation

Threads vs. Time

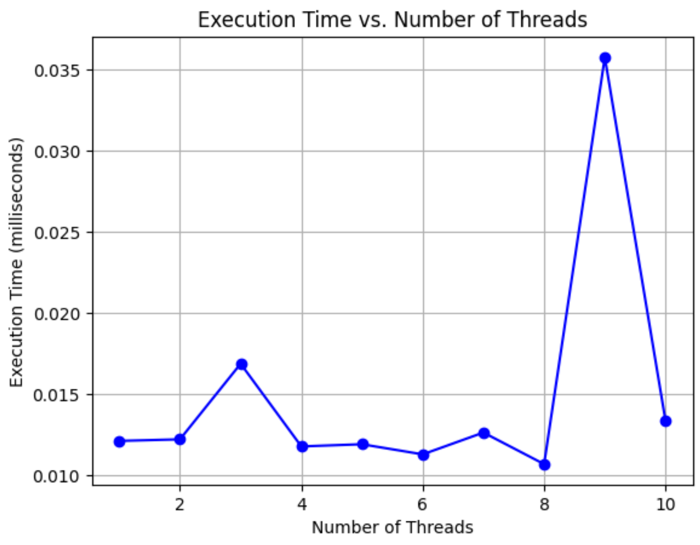


Figure 10 graph of number of threads vs time- Dense Matrix-Vector Multiplication

The graph represents the execution time of the Dense Matrix-Vector Multiplication algorithm for different numbers of threads. The time values, measured in millisecond, depict the impact of thread parallelization on the performance of the multiplication operation.

- As the number of threads increases from 1 to 10, the execution time generally decreases, highlighting the potential benefits of parallelization.
- However, beyond a certain point, increasing the number of threads may lead to diminishing returns, as indicated by the slight increase in time for 9 threads.

Scheduling Strategies vs. Time

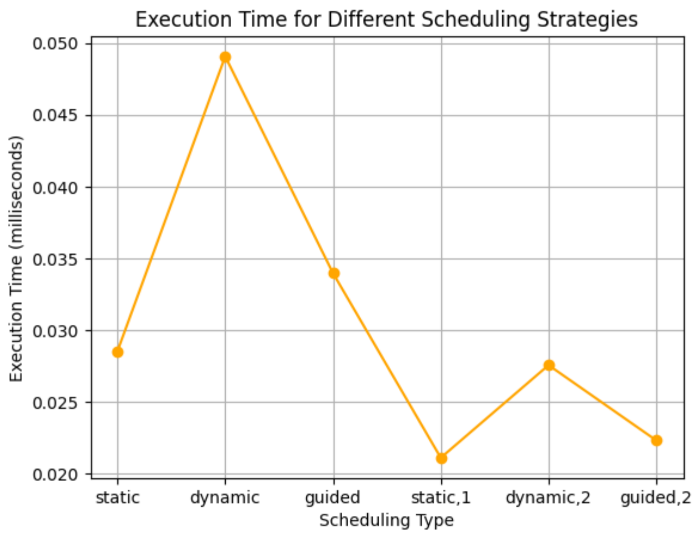


Figure 11 line graph of different scheduling type vs time - Dense Matrix-Vector Multiplication

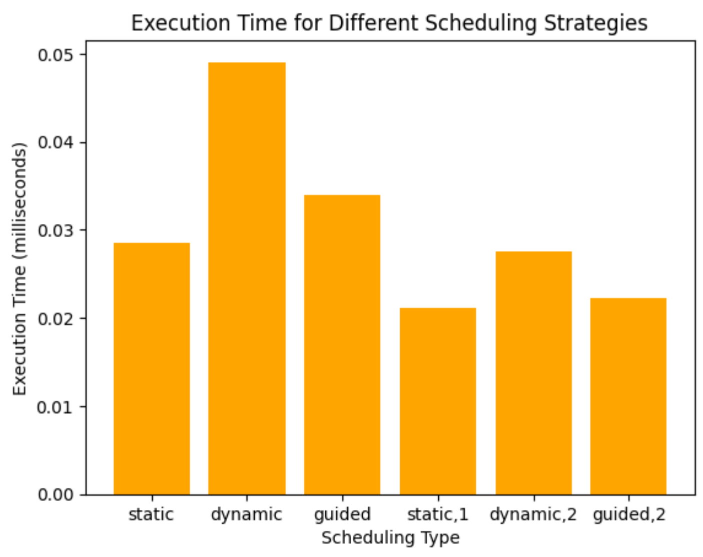


Figure 12 bar graph of different scheduling type vs time - Dense Matrix-Vector Multiplication

The second table details the execution time for Dense Matrix-Vector Multiplication with various OpenMP scheduling strategies. The strategies, including static, dynamic, and guided, influence how computational tasks are distributed among threads.

- The static scheduling strategy consistently exhibits the lowest execution time, indicating efficient task allocation.
- Dynamic scheduling, while flexible, incurs higher execution times, likely due to the overhead of dynamic task allocation.
- Guided scheduling demonstrates a middle ground between static and dynamic approaches.

Nodes vs. Time

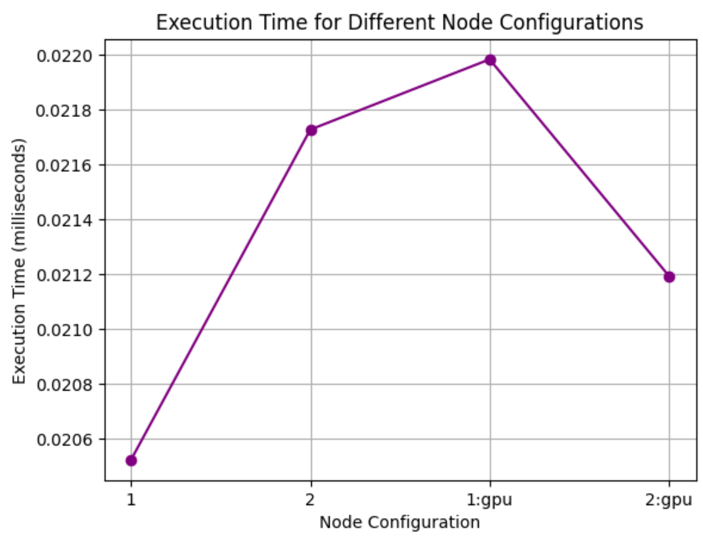


Figure 13 line graph for different node vs time - Dense Matrix-Vector Multiplication

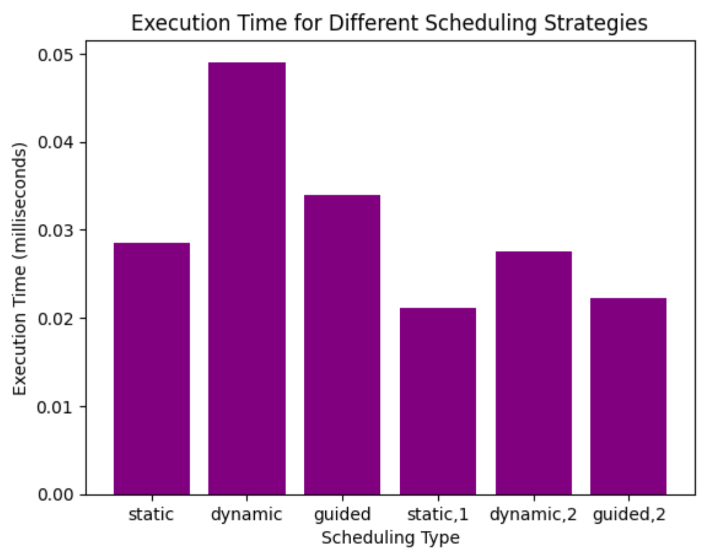


Figure 14 bar graph for different node vs time - Dense Matrix-Vector Multiplication

The third table shows the execution time for Dense Matrix-Vector Multiplication with different node configurations, encompassing CPU-only nodes (1, 2) and GPU-accelerated nodes (1:gpu, 2:gpu).

- GPU-accelerated configurations (1:gpu, 2:gpu) exhibit similar execution times to their CPU-only counterparts (1, 2), suggesting that, for this workload, the GPU acceleration does not significantly impact performance.
- The execution times for both CPU and GPU configurations are within a comparable range.

7 Discussion

Parallel computing has become essential to harness the computational power of modern multi-core processors and high-performance computing environments. In this discussion, we explore the parallel implementation of two fundamental algorithms—Quick Sort and Dense Matrix-Vector Multiplication—using the OpenMP (Open Multi-Processing) framework.

Quick Sort

Quick Sort is a classic sorting algorithm known for its efficiency and widespread use. The parallelization strategy for Quick Sort involves task parallelism, a paradigm well-suited for recursive algorithms. OpenMP's tasking model is employed to parallelize the recursive calls within the `quickSort` function. A crucial consideration is introduced to control the granularity of parallelism. If the size of the subarray is above a threshold (in this case, 1000 elements), the array is split into tasks, each handled by a different thread. Otherwise, the sorting continues sequentially for smaller subarrays. This adaptive strategy aims to balance parallelization overhead and computational efficiency.

The parallelized Quick Sort is evaluated based on its runtime performance. Timing measurements are recorded using `omp_get_wtime()` to calculate the execution time. A critical observation is the trade-off between the overhead of creating tasks and the performance gain achieved through parallel execution. This balance depends on factors such as the array size and the system's computational resources.

OpenMP directives, such as `#pragma omp parallel`, `#pragma omp single`, and `#pragma omp task`, are strategically used to specify parallel regions and task creation. The `nowait` clause in the `#pragma omp single nowait` directive allows the generation of tasks without waiting for their completion. Understanding and effectively utilizing these directives are key to successful parallelization.

Dense Matrix-Vector Multiplication

Dense Matrix-Vector Multiplication is a fundamental linear algebra operation used in various scientific and engineering applications. The parallel implementation employs OpenMP to distribute the workload among multiple threads. The outer loop, responsible for iterating over the matrix rows, is parallelized using `#pragma omp parallel for`. Additionally, a reduction clause is applied to the inner loop to ensure proper updating of the result vector in a thread-safe manner.

Load balancing is crucial in parallel algorithms. The strategy of distributing matrix rows among threads ensures a balanced workload. However, the efficiency of parallelization may vary based on the matrix size and the number of available threads.

The reduction clause in `#pragma omp parallel for reduction(+:result[i])` is employed to handle race conditions during the accumulation of matrix-vector multiplication results. This ensures that each thread has its local copy of the `result` variable, and the reduction operation aggregates the contributions in a thread-safe manner.

Similar to Quick Sort, the performance of Dense Matrix-Vector Multiplication is evaluated using timing measurements. The execution time is recorded to analyze the impact of parallelization on overall performance.

In conclusion, the parallel implementation of Quick Sort and Dense Matrix-Vector Multiplication using OpenMP demonstrates the adaptability of parallel programming paradigms to diverse algorithms. The success of parallelization depends on careful consideration of algorithm characteristics, workload distribution, and system resources. These implementations serve as examples of leveraging parallelism to enhance the efficiency of fundamental algorithms in the context of modern computing architectures.

8 References

[1] K. Batcher, Sorting networks and their applications, Proc. AFIPS Spring Joint Computer Conference, 32, pp.307–314 (1968).

[2] Y. Kim, M. Jeon, D. Kim, A. Sohn, Communication-efficient bitonic sort on a distributed memory parallel computer, Proc. 8th International Conference on Parallel and Distributed Sys￾tems (ICPADS 2001), pp.165–170 (June 2001).

[3] J. S. Huang, Y. C. Chow, Parallel Sorting and Data Par￾titioning by Sampling, Proc. 7th Computer Software and Applications Conference, pp.627–631 (November 1983).

[4] A. C. Dusseau, D. E. Culler, K. E. Schauser, R. P. Martin, Fast Parallel Sorting under Log P: Experience with the CM-5, IEEE Trans. Parallel Distributed Systems, 7, 791–805 (1996).

[5] S. J. Lee, M. Jeon, D. Kim, A. Sohn, Partitioned parallel radix sort, J. Parallel Distr. Comput., 62, 656–668 (2002).

[6] A. Sohn, Y. Kodama, Load balanced parallel radix sort, Proc. 12th ACM International Conference on Supercomputing, pp.305–312 (July 1998).

[7] M. Jeon, D. Kim, Parallel merge sort with load balancing, Inter. J. Parallel Prog., 31, 21–33 (2003).

[8] V. K. Decyk, S. R. Karmesin, A. deBoer, P. C. Liewer, Op￾timization of particle-in-cell codes on reduced instruction set computer processors, J. Comput. Phys. 10, 290–298 (1996).

[9] K. Bowers, Accelerating a particle-in-cell simulation with a hybrid counting sort, J. Comput. Phys.textbf173, 393–411 (2001).

[10] W. S. Brainerd, C. H. Goldberg, C. J. Adams, Programmer's Guide to Fortran 90, McGraw-Hill, New York, pp.149–150 (1990).

[11] M. Suess, C. Leopold, A user's experience with paral￾lel sorting and openmp, Proc. 6th Workshop on OpenMP (EWOMP'04), pp.23–28 (October 2004)

[12] S. L. Olivier, J. F. Prins, Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs, Inter. J. Parallel Prog., 38, 341–360 (2010).