JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Performance analysis of parallel processing using OpenMP on Raspberry PI

Sai Pavan Peddagoni
M. Tech, ES,
ECE Department
JNTUH, Hyderabad, India.

Abstract: Packing multiple CPU cores into a single unit can theoretically boost computational power. However, to fully realize this potential, the software must be specifically designed to take advantage of the multi-core setup. This involves dividing tasks into smaller sub-tasks that can run concurrently on different cores, a process known as parallel programming. By doing so, the workload is spread across multiple cores, improving performance. But this requires careful planning, as developers must ensure proper management of threads and avoid issues like race conditions and deadlocks. This project explores the implementation of multi-core programming using OpenMP directives on the Raspberry Pi platform. The objective is to harness the computational power of multiple cores to enhance the performance of parallelizable algorithms. The project involves the parallelization of key computational tasks through OpenMP directives, including thread management. The core components of the project include the compilation and execution of code with OpenMP support on the Raspberry Pi, facilitating effective thread management. The performance gains achieved through parallelization are evaluated and compared with the sequential execution of the same code. In summary, this project contributes to the field of parallel programming by showcasing the effective utilization of multi-core architectures on the Raspberry Pi using OPENMP.

***Index Terms* – Parallelization, Raspberry pi, OpenMP, Thread management.**

I. Introduction

A multi-core processor, also known as a chip multiprocessor, is a single integrated circuit that contains multiple processing units, or cores. Currently, the CPUs are made with several cores. We will think of cores as separate CPUs. In reality, they are not because they use the same power supply and other peripherals. The CPU cores often have smart algorithms that make them very efficient in executing multiple programs simultaneously. While one program is waiting for data to arrive from RAM, the other program is utilizing the ALU. These individual programs are called threads. In this project, we aim to execute highly parallelized workloads and assess the performance of the design through multiprocessing. By using runtime as the primary metric to measure device performance.

II. Proposed system

Designing an efficient parallel algorithm is quite challenging because most algorithms are structured in a way that requires certain stages to be executed in a specific, predefined order. This creates a significant hurdle when dividing the algorithm into parallel tasks.

The challenge lies in arranging the different parts of the algorithm to run efficiently in parallel while ensuring that the final results can be properly combined in the correct order. Effective parallelization requires careful attention to how dependencies between tasks are managed and how data is shared and synchronized across different threads or cores, all while maintaining the intended flow of the algorithm. Multiprocessing, as applied in multi-core systems like the Raspberry Pi, plays a key role in achieving parallel computation. When using a multi-core processor, such as the one on a Raspberry Pi 4, you can divide tasks into several processes, each running on a separate core. This leads to a significant improvement in performance for computationally heavy applications.
Hence, it is proposed to design a system to perform the analysis of parallel processing by designing an efficient parallel algorithm using OpenMP. The major components in realizing the design are:

i. Raspberry PI.
In this project, the Raspberry Pi 4 was chosen as the target platform. The key reasons for selecting Raspberry Pi include: Raspberry Pi 4's Multi-core CPU, Shared Memory, and Caches. Multi-core Performance on the Raspberry Pi. As of the time of writing, the GCC compiler version 4.6 included in the Raspberry Pi 4's "Raspbian" operating system distribution supports OpenMP 3.0 specifications, which is recent enough for this project.

ii. OpenMP.
We also use OpenMP for this project. OpenMP (Open Multi-Processing) is a widely used API (Application Programming Interface) that enables parallel programming in shared-memory environments, specifically for systems with multi-core

processors. It simplifies the process of writing code that can run across multiple CPU cores, making it ideal for leveraging the full potential of multi-core architectures, like those found in modern computers and embedded systems such as the Raspberry Pi. OpenMP primarily supports C, C++, and Fortran languages and provides an efficient way to write parallel programs using compiler directives (or pragmas), which tell the compiler how to parallelize the code without drastically altering its structure. OpenMP provides compiler directives to specify which sections should be executed in parallel and handles Thread management, Synchronization, and data scoping during the algorithm's parallelization. It provides a simple, flexible way to write parallel code for shared memory systems. OpenMP is a powerful tool for parallel programming in shared-memory environments, offering a simple way to add parallelism to existing code and achieve significant performance improvements with minimal effort.

IMPLEMENTING OPENMP ON RASPBERRY PI.

The Raspberry Pi’s multi-core architecture, especially in models like the Raspberry Pi 4, makes it an excellent platform for experimenting with OpenMP. To get started with OpenMP on the Raspberry Pi:

1. Install GCC (GNU Compiler Collection): The GCC compiler supports OpenMP out of the box, so ensure it's installed on your Raspberry Pi.
2. Write OpenMP Code: Write your C or C++ code with OpenMP pragmas to parallelize sections.
3. Compile with OpenMP Support: When compiling, use the -fopenmp flag to enable OpenMP.
4. Run the Program: After compiling, run the program to see how it leverages multiple cores on the Raspberry Pi. We observe the performance of the parallel processing by comparing the time.

III. Methodology

3.1 Algorithm Selection and Code Design.

The core concept of the implementation is to take a computational algorithm, run it sequentially, and then parallelize it using OpenMP to compare the performance gains.

APPROACH:

a) Choose an Algorithm:
An iterative computational function (e.g., calculating an array of gamma values) was chosen for performance evaluation. This algorithm performs repetitive calculations, making it a good candidate for parallelization.

b) Sequential Execution:
First, the algorithm is run sequentially (without parallelism) to obtain a baseline performance measurement.

c) Parallelization Using OpenMP:
The algorithm is parallelized using OpenMP directives.
To parallelize, the #pragma omp parallel directive is added above the for-loop to allow multiple threads to execute parts of the loop simultaneously.

d) Execution of Parallelized Code:
The parallelized code is executed on the Raspberry Pi with varying numbers of threads. The performance is then measured in terms of execution time and CPU utilization. The header file #include "SimpleTimer.h" is used to measure the runtime of the function.

e) Comparison of Sequential and Parallel Execution:
The runtime of the parallelized code is compared against the runtime of the sequential code. A key point of analysis is how the number of threads affects performance. For example, tests are conducted using 2, 4, and more threads, and the results are recorded. Based on the run time, the performance of the device is measured.

IV. VERIFYING THE DESIGN OF DIFFERENT FUNCTIONS

4.1. Matrix Exponentiation Using Iterative Multiplication.

This code raises a square matrix to a specified power using iterative matrix multiplication and measures the time taken. The program prompts the user to input the size of the square matrix (n) and the power to which the matrix should be raised. The matrix A is initialized with all elements set to 1.0. Multiplying two matrices A and B to produce the result matrix C. computing A^power, which is matrix A raised to a given power using matrix multiplication.

The matrix function is defined as:

```
for (int i = 0; i < n; ++i) {
for (int j = 0; j < n; ++j) {
for (int k = 0; k < n; ++k) {
C[i][j] += A[i][k] * B[k][j];.
} } }
// Function to raise a matrix to a power using iterative multiplication
vector<vector> matrixPower(const vector<vector>& A, int power) {
int n = A.size();
vector<vector> result(n, vector(n, 0.0));
vector<vector> base = A;
// Initialize the result as the identity matrix
for (int i = 0; i < n; ++i) {
result[i][i] = 1.0; }
// Multiply the matrix by itself power times
for (int p = 0; p < power; ++p) {
 result = multiplyMatrices(result, base); }
```

The parallel code is an improved version of matrix exponentiation using OpenMP to parallelize the matrix multiplication. Parallelizing the outermost loop using OpenMP helps distribute the work of matrix multiplication across multiple threads, potentially reducing the time required for large matrices. The use of #pragma omp atomic ensures thread safety when updating elements of the matrix. However, atomic operations can introduce some overhead. In this case, since matrix multiplications are independent, you could refactor the code to avoid atomic operations by dividing the work more carefully between threads (for example, by giving each thread exclusive responsibility for certain rows).

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
for (int j = 0; j < n; ++j) {
for (int k = 0; k < n; ++k) {
#pragma omp atomic
C[i][j] += A[i][k] * B[k][j];
} } }
return C; }
```

// Function to raise a matrix to a power using iterative multiplication and OpenMP

```
vector<vector> matrixPower(const vector<vector>& A, int power) {
int n = A.size();
vector<vector> result(n, vector(n, 0.0));
vector<vector> base = A;
// Initialize result as the identity matrix
for (int i = 0; i < n; ++i) {
result[i][i] = 1.0; }
// Multiply the matrix by itself power times
for (int p = 0; p < power; ++p) {
result = multiplyMatrices(result, base); }
```

Run time is measured for both cases using SimplerTimer and performance is analysed by comparing the runtimes.

```
SimpleTimer tm;
 tm.start();
 main function
tm.end();.
```

4.2. Factorial Calculation.

Create a program that calculates the factorial of a given number and measures the time required for the computation using the SimpleTimer class. The factorial function follows an iterative approach, multiplying all integers from 1 up to n to determine the factorial value. The goal is to calculate the factorial of a number sequentially and measure the time taken to perform the calculation.

```
// Define the factorial function
unsigned long long factorial(int n) {
unsigned long long result = 1;
for (int i = 1; i <= n; ++i) {
result *= i;}
return result;
```

Using OpenMP to parallelize the factorial calculation speeds up the process, especially for large values of n.

// Define the parallel factorial function using OpenMP

```
unsigned long long para factorial(int n) {
if (n == 0 || n == 1) return 1;

unsigned long long result = 1;
#pragma omp parallel for reduction(*:result)
for (int i = 2; i <= n; ++i) {
result *= i; }
return result; }
```

The para factorial function uses OpenMP to parallelize the multiplication of numbers from 2 to n. The #pragma omp parallel for reduction (*: result) directive enables parallel computation of the factorial. The reduction (*: result) clause ensures that each thread computes part of the factorial and then safely multiplies the partial results. The reduction operation ensures that the multiplication (*) is performed in a way that avoids race conditions, where multiple threads might try to update the result variable at the same time. OpenMP automatically handles the combination of the partial results from each thread.

Run time is measured for both cases using SimplerTimer and performance is analysed by comparing the runtimes.

```
SimpleTimer tm;
 tm.start();
main function
tm.end();.
```

4.3. Iterative Fibonacci Calculation.

Calculate the n-th Fibonacci number using an iterative approach, and measure the time it takes to perform the calculation using the SimpleTimer class. The Fibonacci function computes the Fibonacci number for a given n using an iterative approach. This method is more efficient than the recursive one, which can have exponential time complexity due to repeated calculations. The iterative method runs in O(n) time. The SimpleTimer class is used to measure the time taken for the Fibonacci calculation.
The program prompts the user to enter a number (num), which represents the position in the Fibonacci sequence and then calculates the corresponding Fibonacci number.
Calculating the Fibonacci number iteratively, using a loop to compute the sequence.

```
// Function to calculate the n-th Fibonacci number iteratively
        unsigned long long fibonacci(int n) {
        if (n <= 0) return 0;
        if (n == 1) return 1;
        unsigned long long fib0 = 0;
        unsigned long long fib1 = 1;
        unsigned long long fibN = 0;
        for (int i = 2; i <= n; ++i) {
        fibN = fib0 + fib1;
         fib0 = fib1;
        fib1 = fibN; }
        return fibN; }
         int main() {
        int num;
// Prompt the user to enter a number
        cout << "Enter a number to calculate the Fibonacci sequence: ";
        cin >> num;
// Calculate the Fibonacci number
        SimpleTimer tm;
        tm.start(); // Start the timer
        unsigned long long result = fibonacci(num); // Calculate Fibonacci number
// Output the result
        cout << "The " << num << "-th Fibonacci number is " << result << "." << endl;
        tm.end(); // End the timer
        std::cout << "The parallel function running time is: " << tm.getTime() << "\n"; // Output time taken
        return 0; }
```

This function calculates the n-th Fibonacci number using an iterative approach.It initializes two variables fib0 (0) and fib1 (1) as the first two Fibonacci numbers. loop (starting from index 2), it calculates the next Fibonacci number by adding fib0 and fib1, updating fib0 and fib1 in each iteration. The result is returned after the loop ends. Prompts the user to input a number. Starts the timer using SimpleTimer and calls the Fibonacci function to calculate the n-th Fibonacci number. The result is printed, and the time taken is output using tm.getTime(). tm.start() begins the timer before the Fibonacci calculation. tm.end() stops the timer and calculates the elapsed time. The time is displayed using tm.getTime(), which likely returns the time in seconds.
The parallel version attempts to parallelize the Fibonacci calculation using OpenMP, which allows multiple threads to compute parts of the Fibonacci sequence concurrently.

```
// Function to calculate the n-th Fibonacci number iteratively using OpenMP
        long long parallelFibonacci(int n) {
        if (n <= 0) return 0;
        if (n == 1) return 1;
// Initialize the first two Fibonacci numbers
        long long fib0 = 0;
        long long fib1 = 1;
        long long fibN = 0;
        #pragma omp parallel for
         for (int i = 2; i <= n; ++i) {
        #pragma omp critical {
        fibN = fib0 + fib1;
         fib0 = fib1;
        fib1 = fibN; } }
        return fibN; }
        int main() {
        int num;
// Prompt the user to enter a number
        cout << "Enter a number to calculate the Fibonacci sequence: ";
        cin >> num;
// Calculate Fibonacci number
        SimpleTimer tm;
        tm.start(); // Start the timer
        long long result = parallelFibonacci(num); // Calculate Fibonacci number using parallel method
        tm.end(); // End the timer
        std::cout << "The function auto running time is: " << tm.getTime() << "\n"; // Output time taken
// Output the result
        cout << "The " << num << "-th Fibonacci number is " << result << "." << endl;
        return 0; }
```

#pragma omp parallel for: This directive tells the compiler to parallelize the for loop. However, the Fibonacci calculation is inherently sequential because each Fibonacci number depends on the previous two, so parallelizing it in this way may not yield a performance boost.
#pragma omp critical: This ensures that the critical section of updating fib0, fib1, and fibN is executed by only one thread at a time, preventing race conditions. This is necessary because these variables are being updated by multiple threads, which could lead to inconsistent results if not properly synchronized. Similar to the non-parallel version, it prompts the user to input a number. The Fibonacci number is calculated in parallel using parallel Fibonacci, and the time taken is measured and displayed. SimpleTimer class is used the same way as in the non-parallel version to measure the execution time.

4.4. Jacobi Iterative method for solving linear systems.

The Jacobi method is an iterative algorithm that repeatedly improves an initial guess of the solution vector x using the formula:

$$x_i^{(k+1)} = \frac{1}{A_{ii}}\left(b_i - \sum_{j\neq i} A_{ij}x_j^{(k)}\right)$$

Figure 4.1

Where:
• xi(k+1) is the value of the i-th variable after the (k+1)-th iteration.
• Aij are the elements of matrix A.
• bi is the i-th element of vector b.
• xj(k) is the value of the j-th variable in the current iteration
The provided code examples demonstrate the Jacobi method for solving a system of linear equations Ax=b using both non-parallel and parallel versions. The Jacobi method is an iterative algorithm used to solve the equation system where A is a matrix and b is a vector.

```
// Function to perform the Jacobi iteration
        vector jacobiMethod(const vector<vector>& A, const vector& b, int maxIterations, double tolerance) {
        int n = A.size();
        vector <double>x(n, 0.0); // Initial guess (x = 0 for all elements)
        vector<double> x_new(n, 0.0); // New solution to update x
// Jacobi iteration loop
        for (int iteration = 0; iteration < maxIterations; ++iteration) {
        for (int i = 0; i < n; ++i) {
        double sigma = 0.0;
// Calculate sigma (sum of A[i][j] * x[j] for j != i)
        for (int j = 0; j < n; ++j) {
        if (j != i) {
        sigma += A[i][j] * x[j]; } }
// Update x_new[i]
        x_new[i] = (b[i] - sigma) / A[i][i]; }
// Check for convergence (error calculation)
        double error = 0.0;
        for (int i = 0; i < n; ++i) {
        error += fabs(x_new[i] - x[i]);
        x[i] = x_new[i]; // Update solution vector }
// If the error is below the tolerance, stop the iterations
        if (error < tolerance) {
        break; } }
        return x; }
        int main() {
// Define the system of equations Ax = b
        vector<vector> A = {
        {4, -1, 0, 0},
        {-1, 4, -1, 0},
        {0, -1, 4, -1},
        {0, 0, -1, 3} };
        vector b = {15, 10, 10, 10}; // The right-hand side vector
        int maxIterations = 1000;
        double tolerance = 1e-6;
// Perform Jacobi iteration
        SimpleTimer tm;
        tm.start();
        vector solution = jacobiMethod(A, b, maxIterations, tolerance);
// Print the solution
        cout << "Solution:" << endl;
        printVector(solution);
// End the timer and print the execution time
        tm.end();
        std::cout << "The non-parallel function running time is: " << tm.getTime() << "\n";
        return 0; }
```

The core of the method is the iterative update of the solution vector x. At each iteration, the solution for each unknown x[i] is computed using the other values x[j] (for j≠ij \neq ij =i). The new values of x are stored in x_new, and after each iteration, x is updated with x_new. The loop continues until the change between consecutive x values (error) is smaller than the given tolerance. SimpleTimer is used to measure and output the execution time for the Jacobi method. The loops that compute the new values for x_new and the error calculation are parallelized using OpenMP. This enables the concurrent computation of multiple elements of the solution vector.

```
// Function to perform the Jacobi iteration using OpenMP for parallelization
        vector jacobiMethod(const vector<vector>& A, const vector& b, int maxIterations, double tolerance) {
        int n = A.size();
        vector<double> x(n, 0.0); // Initial guess (x = 0 for all elements)
        vector<double> x_new(n, 0.0); // New solution to update x
// Jacobi iteration loop with OpenMP parallelization
        for (int iteration = 0; iteration < maxIterations; ++iteration) {
// Parallelize the loop using OpenMP
        #pragma omp parallel for
        for (int i = 0; i < n; ++i) {
        double sigma = 0.0;
// Calculate sigma (sum of A[i][j] * x[j] for j != i)
        for (int j = 0; j < n; ++j) {
        if (j != i) {
        sigma += A[i][j] * x[j]; } }
// Update x_new[i]
        x_new[i] = (b[i] - sigma) / A[i][i]; }
// Check for convergence (error calculation)
        double error = 0.0;
        #pragma omp parallel for reduction(+:error)
        for (int i = 0; i < n; ++i) {
        error += fabs(x_new[i] - x[i]);
        x[i] = x_new[i]; // Update solution vector
        }
// If the error is below the tolerance, stop the iterations
        if (error < tolerance) {
        break;
        } }
        return x;
        }
        int main() {
// Define the system of equations Ax = b
        vector<vector> A = {
        {4, -1, 0, 0},
        {-1, 4, -1, 0},
        {0, -1, 4, -1},
        {0, 0, -1, 3} };
        vector b = {15, 10, 10, 10}; // The right-hand side vector
        int maxIterations = 1000;
        double tolerance = 1e-6; // Perform Jacobi iteration using OpenMP for parallel computation
        SimpleTimer tm;
        tm.start();
        vector solution = jacobiMethod(A, b, maxIterations, tolerance);
// Print the solution
        cout << "Solution:" << endl;
        printVector(solution);
// End the timer and print the execution time
        tm.end();
        std::cout << "The parallel function running time is: " << tm.getTime() << "\n";
        return 0; }.
```

The loop that updates the x_new values is parallelized with #pragma omp parallel. The calculation of error in the convergence check is also parallelized using #pragma omp parallel for reduction(+:error) to ensure that the error is computed correctly across all threads. No critical section is required in the parallel version because each thread computes an independent part of the solution. However, it's important to synchronize the updates and the error calculation.

4.5. Mandelbrot set generation.

The Mandelbrot set, a famous fractal, and saved as a Portable Pixmap (PPM) image file. The Mandelbrot set is a set of complex numbers where the iterative function z_{n+1} = z_n^2 + c does not diverge (i.e., remains bounded). The code iteratively computes this function for each pixel, with the number of iterations indicating how quickly it escapes the bounded region. The image is saved in the PPM format, which is a simple text-based format. P3 specifies that the image is in plain-text format. The dimensions and color depth (255) are included, followed by RGB values for each pixel. The user is prompted to input the width, height, and the maximum number of iterations for the image. The calculate Mandelbrot() function is called, which generates the Mandelbrot set and saves the image to a file called Mandelbrot. ppm. The image is created by iterating over each pixel, calculating the corresponding complex number, and determining how many iterations it takes for that number to escape the bounded region. The pixel color is written to the output file based on the iteration count. Finally, the elapsed time to generate the Mandelbrot set is printed.

```
// Function to calculate the Mandelbrot set and save the result to an image file
        void calculateMandelbrot(int width, int height, int maxIterations, const string& filename) {
        ofstream imageFile(filename);
        imageFile << "P3\n" << width << " " << height << "\n255\n"; // PPM header
        for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
// Mapping pixel (x, y) to the
        complex plane complex<double> c((double)x / width * 3.5 - 2.5, (double)y / height * 2.0 - 1.0);
        complex <double>z(0.0, 0.0);
        int iterations = 0;
// Mandelbrot iteration
while (abs(z) <= 2.0 && iterations < maxIterations) {
        z = z * z + c;
        iterations++; }
// Map the number of iterations to a color
        int color = (iterations == maxIterations) ? 0 : 255 * iterations / maxIterations;
        imageFile << color << " " << color << " " << color << " "; // Write color to file }
        imageFile << "\n"; // New line for the next row of pixels
        }
        imageFile.close();
        cout << "Mandelbrot set image saved to " << filename << endl; }
        int main() {
        int width, height, maxIterations;
. // Prompt the user to enter the image dimensions and max iterations
        cout << "Enter the width of the image: ";
        cin >> width;

        cout << "Enter the height of the image: ";
        cin >> height;
        cout << "Enter the maximum number of iterations: ";
        cin >> maxIterations;
// Calculate the Mandelbrot set
        SimpleTimer tm;
        tm.start();
        calculateMandelbrot(width, height, maxIterations, "mandelbrot.ppm");
        tm.end();
        std::cout << "The non-parallel function running time is: " << tm.getTime() << "\n";
        return 0; }.
```

The calculation for each pixel is done sequentially, and the results are written directly to a file in PPM format. Mandelbrot Set is computed by iterating on each pixel, mapping it to a point in the complex plane, and performing iterations until either the point escapes or the maximum number of iterations is reached. The output is a PPM file with the P3 format, which is a plain text format where each pixel is represented by RGB values. The outer loop (over rows of pixels) is parallelized using OpenMP, allowing multiple rows to be processed concurrently.

```
// Function to calculate the Mandelbrot set and save the result to an image file
        void calculateMandelbrot(int width, int height, int maxIterations, const string& filename) {
        ofstream imageFile(filename);
        imageFile << "P3\n" << width << " " << height << "\n255\n";
// PPM header // Parallelize the outer loop with OpenMP
        #pragma omp parallel for schedule(dynamic)
        for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
// Mapping pixel (x, y) to the complex plane
        Complex<double> c((double)x / width * 3.5 - 2.5, (double)y / height * 2.0 - 1.0);
        Complex<double> z(0.0, 0.0);
        int iterations = 0;
// Mandelbrot iteration
        while (abs(z) <= 2.0 && iterations < maxIterations) {
        z = z * z + c;
        iterations++; }
// Map the number of iterations to a color
        int color = (iterations == maxIterations) ? 0 : 255 * iterations / maxIterations;
        #pragma omp critical
        {
        imageFile << color << " " << color << " " << color << " "; // Write color to file
        } }
        #pragma omp critical
        {
        imageFile << "\n"; // New line for the next row of pixels } }
        imageFile.close();
        cout << "Mandelbrot set image saved to " << filename << endl; }
        int main() {
        int width, height, maxIterations;
// Prompt the user to enter the image dimensions and max iterations
        cout << "Enter the width of the image: ";
        cin >> width;
        cout << "Enter the height of the image: ";
        cin >> height;
        cout << "Enter the maximum number of iterations: ";
        cin >> maxIterations;
// Calculate the Mandelbrot set
SimpleTimer tm;
        tm.start();
        calculateMandelbrot(width, height, maxIterations, "mandelbrot.ppm");
        tm.end();
        std::cout << "The parallel function running time is: " << tm.getTime() << "\n";
        return 0; }
```

The outer loop (over the height of the image) is parallelized using the #pragma omp parallel for the directive, which divides the work across multiple threads. The schedule(dynamic) clause helps in distributing the work dynamically, which can improve load balancing, especially when some parts of the image take longer to compute than others. Since of stream is not thread-safe, a #pragma omp critical directive is used to ensure that writing to the image file happens sequentially within each thread. This prevents race conditions when multiple threads try to write to the file at the same time. Mandelbrot Calculation for each pixel is the same as in the non-parallel version, but now it's performed in parallel across different rows of pixels. SimpleTimer is used to measure and output the execution time.

4.6. Calculation of Euler's number using series expansion.

calculates the mathematical constant e (Euler's number) using a series expansion and measures the execution time of the calculation. The user provides the number of iterations, which determines the accuracy of the approximation of e. Factorial Function: Calculates the factorial of a number iteratively. The factorial function is used to compute each term in the series. The calculated function sums up the series up to the specified number of iterations. The user inputs the number of iterations, and the program outputs the calculated value of e and the time taken for the computation.

```
// Function to calculate the factorial of a number iteratively
            unsigned long long factorial(int n) {
            unsigned long long result = 1;
            for (int i = 1; i <= n; ++i) {
            result *= i; }
            return result;}
// Function to calculate the value of e using the series expansion
            double calculateE(int iterations) {
            double e = 1.0; // Start with the first term of the series
            for (int i = 1; i <= iterations; ++i) {
            e += 1.0 / factorial(i); // Add the next term of the series
            }
            return e; }
            int main() {
            int numIterations;
// Prompt the user to enter the number of iterations
            cout << "Enter the number of iterations to calculate e: ";
            cin >> numIterations;
// Calculate e
            SimpleTimer tm;
            tm.start();
            double result = calculateE(numIterations); // Calculate e using the series expansion
// Output the result
            cout << fixed << setprecision(15);
            cout << "The value of e calculated with " << numIterations << " iterations is " << result << "." << endl;
```

```
        tm.end();
        std::cout << "The non-parallel function running time is: " << tm.getTime() << "\n"; // Display the time taken
        return 0; }.
```

Factorial Calculation is computed iteratively. The value of e is calculated by summing the terms of the series expansion up to the specified number of iterations. The program measures and prints the execution time using the SimpleTimer.

```
// Function to calculate the factorial of a number iteratively
        unsigned long long factorial(int n) {
        unsigned long long result = 1;
        for (int i = 1; i <= n; ++i) {
        result *= i; }
        return result; }
// Function to calculate the value of e using the series expansion with OpenMP
        double calculateE(int iterations) {
        double e = 1.0; // Start with the first term of the series
// Parallelize the summation with OpenMP
        #pragma omp parallel for reduction(+:e)
        for (int i = 1; i <= iterations; ++i) {
        e += 1.0 / factorial(i); // Add the next term of the series
         }
        return e; }
        int main() {
        int numIterations;
// Prompt the user to enter the number of iterations
        cout << "Enter the number of iterations to calculate e: ";
        cin >> numIterations;
        // Calculate e
        SimpleTimer tm;
        tm.start();
        double result = calculateE(numIterations); // Calculate e using the series expansion
// Output the result
        cout << fixed << setprecision(15);
        cout << "The value of e calculated with " << numIterations << " iterations is " << result << "." << endl;
        tm.end();
        std::cout << "The parallel function running time is: " << tm.getTime() << "\n"; // Display the time taken
        return 0; }.
```

The #pragma omp parallel for reduction(+:e) directive is used to parallelize the summation of the series. The reduction ensures that each thread gets its own local copy of the result (e), and then the local results are combined at the end. As in the non-parallel version, factorials are calculated sequentially for each term. The program measures and prints the execution time, similar to the non-parallel version using SimpleTimer.

***4.7. Prime Number Calculation Using Sieve of Eratosthenes*.**

Calculate and print all prime numbers up to a user-specified limit using the Sieve of Eratosthenes algorithm. The execution time of the sieve process is measured using a custom SimpleTimer class. Sieve of Eratosthenes: An efficient algorithm for finding all prime numbers up to a given limit n. SimpleTimer for Timing: The total execution time of the sieve function is measured and displayed. Efficient Prime Calculation: The algorithm marks non prime numbers in a Boolean array and prints only the prime numbers.

Program Flow:

1. The user inputs a limit n (up to which prime numbers are to be calculated).
2. The program runs the Sieve of Eratosthenes to calculate all primes up to n.
3. It prints the prime numbers and measures the execution time.

```
// Function to calculate prime numbers using the Sieve of Eratosthenes
        void sieveOfEratosthenes(int n) {
        vector<bool> isPrime(n + 1, true); // Assume all numbers are prime initially
        isPrime[0] = isPrime[1] = false; // 0 and 1 are not prime numbers
        for (int p = 2; p <= sqrt(n); ++p) {
        if (isPrime[p]) { // If the number is prime
        for (int i = p * p; i <= n; i += p) { // Mark multiples of p as non-prime
        isPrime[i] = false; } } }
// Output the prime numbers
        cout << "Prime numbers up to " << n << " are:\n";
        for (int i = 2; i <= n; ++i) {
        if (isPrime[i]) {
        cout << i << " "; // Print the prime number } }
        cout << endl; }
        int main() {
        int limit;
// Prompt the user to enter the limit
        cout << "Enter the limit to calculate prime numbers: ";
        cin >> limit;
// Calculate prime numbers
        SimpleTimer tm;
        tm.start();
        sieveOfEratosthenes(limit); // Call the function to calculate primes
        tm.end(); // Output the running time
        std::cout << "The non-parallel function running time is: " << tm.getTime() << "\n";
        return 0; }.
```

The algorithm marks multiples of each prime number starting from 2 as non-prime. The outer loop runs up to sqrt(n) because for any composite number, at least one of its divisors will be smaller than or equal to sqrt(n). After marking the non-prime numbers, the algorithm prints the prime numbers in the range.

```
// Function to calculate prime numbers using the Sieve of Eratosthenes with OpenMP
        void sieveOfEratosthenes(int n) {
        vector<bool> isPrime(n + 1, true);
        isPrime[0] = isPrime[1] = false;
        int limit = sqrt(n);
        #pragma omp parallel for schedule(dynamic)
        for (int p = 2; p <= limit; ++p) {
        if (isPrime[p]) {
        for (int i = p * p; i <= n; i += p) {
        isPrime[i] = false; } } }
// Output the prime numbers
        cout << "Prime numbers up to " << n << " are:\n";
        for (int i = 2; i <= n; ++i) {
        if (isPrime[i]) {
        cout << i << " "; } }
        cout << endl; }

        int main() {
        int limit;
// Prompt the user to enter the limit
        cout << "Enter the limit to calculate prime numbers: ";
        cin >> limit;
// Calculate prime numbers
        SimpleTimer tm;
        tm.start();
        sieveOfEratosthenes(limit); // Call the function to calculate primes
        tm.end(); // Output the running time
        std::cout << "The parallel function running time is: " << tm.getTime() << "\n";
        return 0; }.
```

The #pragma omp parallel for directive parallelizes the outer loop where we iterate over potential prime numbers. Each thread computes different values of p simultaneously. The schedule(dynamic) clause dynamically assigns iterations to threads, which allows better load balancing, especially when working with a large n. In this case, there is no need for an explicit critical section because each thread works on a different part of the sieve array. OpenMP takes care of handling thread safety when updating shared resources (the isPrime array). The parallel version should show performance improvement for large n because multiple threads can mark non-prime numbers simultaneously. The overhead of managing multiple threads might slow it down for smaller input sizes.

4.8. Estimating Pi Using the Leibniz Formula.

Calculate the value of π\piπ using the Leibniz formula. The program prompts the user to input the number of iterations for the calculation and measures the execution time using a SimpleTimer class.

Function Definition:

Takes an integer iteration representing how many terms of the series to calculate.

• Logic:

- Initializes a variable pi to accumulate the value.
- Uses a loop to calculate each term of the Leibniz series for π\piπ:

$$\pi \approx 4 \sum_{i=0}^{n} \frac{(-1)^i}{2i+1}$$

Figure 4.2

- Each term is computed as frac{(-1)^i}/{2i + 1}, and then the result is multiplied by 4 at the end to estimate π\pi.

```
// Function to calculate pi using the Leibniz formula iteratively
        double calculatePi(int iterations) {
        double pi = 0.0;
        for (int i = 0; i < iterations; ++i) {
        double term = pow(-1, i) / (2 * i + 1); // Leibniz series term
        pi += term; }
        return pi * 4; // Multiply by 4 to get the value of pi }
```

```
            int main() {
            int numIterations;
// Prompt the user to enter the number of iterations
            cout << "Enter the number of iterations to calculate pi: ";
            cin >> numIterations;
// Calculate pi
            SimpleTimer tm;
            tm.start();
            double result = calculatePi(numIterations);
// Output the result with increased precision
            cout << std::setprecision(30);
            cout << "Increased precision: " << numIterations << " iterations is " << result << "." <<
            endl;
            tm.end();
            std::cout << "The non-parallel function running time is: " << tm.getTime() << "\n";
            return 0; }
```

Leibniz's formula is used to calculate π by iterating through the series and adding or subtracting terms. A SimpleTimer object is used to measure the running time of the calculation. The result is printed with high precision (30 decimal places).

```
// Function to calculate pi using the Leibniz formula iteratively with OpenMP
            double calculatePi(int iterations) {
            double pi = 0.0;
            #pragma omp parallel for reduction(+:pi)
            for (int i = 0; i < iterations; ++i) {
            double term = pow(-1, i) / (2 * i + 1); // Leibniz series term
            pi += term; }
            return pi * 4; // Multiply by 4 to get the value of pi
            }
            int main() {
            int numIterations;
// Prompt the user to enter the number of iterations
            cout << "Enter the number of iterations to calculate pi: ";
            cin >> numIterations;
// Calculate pi
            SimpleTimer tm;
            tm.start();
            double result = calculatePi(numIterations);
// Output the result with increased precision
            cout << std::setprecision(10);
            cout << "Increased precision: " << numIterations << " iterations is " << result << "." << endl;
            tm.end();
            std::cout << "The parallel function running time is: " << tm.getTime() << "\n";
            return 0; }.
```

The #pragma omp parallel for directive allows the loop to be executed in parallel, where multiple threads compute the sum of the series terms concurrently. The reduction(+:pi) clause ensures that each thread has its own private copy of pi, and at the end of the loop, the partial results are summed together.

4.9 Measuring Performance of Modular Exponentiation

$$(a^b)\%p. \qquad \text{gamma}[i] = \left(\text{alpha}[i]^{\text{beta}[i]}\right)\%p.$$

Figure 4.3

The above-shown figure 4.3 is the modular exponentiation function, which we choose to measure the performance of the Raspberry Pi. To convert it to an iterative function assign gamma, alpha, and beta variables, respectively as shown in the figure. The function for N = 100,000 elements, computes the result for each element of gamma[i] = alpha[i]^beta[i] % p. The code allocates memory dynamically for large arrays (alpha, beta, and gamma).

The variables were assigned the values of p = 17 , N = 100000,alpha = new long[N], beta = new long[N], gamma = new long[N] and

```
alpha[0] = 8; beta[0] = 5183;
alpha[1] = 9; beta[1] = 4351;
alpha[2] = 7; beta[2] = 4711;
```

For N = 100000, the function is being iterated.

The function is defined as:

```
for (long i = 3; i < N; ++i)
 alpha[i] = alpha[i - 3]
beta[i] = beta[i - 3].
// Defining the function.
void nonParallelFunction(long p, long n, long* alpha, long* beta, long* gamma) {
```

```
for (long i = 0; i < n; ++i) {
gamma[i] = powerFunction(alpha[i], beta[i], p); } }
```

Each thread executes the power function for every pair of alpha[i] and beta[i], one by one. Here's how it works. Power Function calculates the value a^b % p iteratively by multiplying a for b times and taking modulo p at each step. Non-Parallel Function iterates through all the values from 0 to N-1 and applies the powerFunction to each alpha[i], beta[i] pair, storing the result in gamma[i]. A SimpleTimer instance is used to measure the time taken by the nonparallel Function.

Implemented two functions: parallelFun and parallelFunction. Both use OpenMP to parallelize the calculation.

Parallel Fun (auto threads):

```
void parallelFun(long p, long n, long* alpha, long* beta, long* gamma) {
int m = omp_get_max_threads(); // Get the number of available threads
#pragma omp parallel num_threads(m)
{
int myId = omp_get_thread_num(); // Get the thread's ID
std::cout << "No of threads auto =" << myId << "\n";
for (long i = myId; i < n; i += m) { // Distribute the workload across threads
gamma[i] = powerFunction(alpha[i], beta[i], p); } } }
```

The number of threads (m) is dynamically determined based on the system's available resources. Each thread is responsible for processing a portion of the array (gamma[i]), starting from an index determined by its thread ID.

Parallel Function (manual threads):

```
void parallelFunction(long p, long n, long* alpha, long* beta, long* gamma, int myId) {
int m = omp_get_max_threads(); // Get the number of available threads
std::cout << "No of threads =" << myId << "\n";
#pragma omp parallel num_threads(m) {
for (long i = myId; i < n; i += m) { // Distribute the workload across threads
gamma[i] = powerFunction(alpha[i], beta[i], p); } } }
```

In this case, you pass the thread ID (myId) to manually set which thread processes which part of the array. The main function initializes arrays for alpha, beta, and gamma, and runs the parallel functions multiple times, varying the number of threads from 1 to 20. It measures the execution time using SimpleTimer.

V. RESULTS AND DISCUSSION

Here, a comparison is done on the performance and analysis of parallel processing using OpenMP on Raspberry Pi by using runtime as the metric.We observe from the results that there is a relative speedup in the device performance while doing the same task using OpenMP. OpenMP substantially improves performance by utilizing multiple cores and parallelizing computational tasks. Performance analysis is done by comparing the various runtimes of different functions, as shown below.

Table 5.1: The table compares the performance of several computational functions when executed in non-parallel (sequential) and parallel modes on Raspberry Pi.

Function Name: Specifies the computational task being evaluated.

Non-Parallel Run Time (seconds): The time taken for the function to execute sequentially, without utilizing parallel processing.

Parallel Run Time (seconds): The time taken for the function to execute when parallelized, using OpenMP.

NAME OF THE FUNCTION	NON-PARALLEL RUN TIME (seconds)	PARALLEL RUN TIME(seconds)
MODULAR EXPONENTIATION	1.704	0.199
MATRIX EXPONENTIATION	10.54	3.59
FACTORIAL CALCULAION	8.033	2.474
ITERATIVE FIBONACCI SERIES	4.905	2.023
JACOBI ITERATIVE METHOD FOR SOLVING LINEAR SYSTEMS	0.513	0.417
MANDELBROT SET GENERATION	9.059	8.974
CALCULATION OF EULER's NUMBER USING SERIES EXPANSION	97.59	20.30
PRIME NUMBER CALCULATION USING SIEVE OF ERATOSTHENES	6.345	3.642
ESTIMATING PI VALUE USING THE LEIBNIZ FORMULA	62.908	8.23552

By using Time as a metric we compare the performance of the Raspberry Pi for Parallel and Non-parallel execution. Hence, observed that the performance of Raspberry Pi relatively increased with parallel processing.

Table 5.2: The table provides a comparison of parallel run times, non-parallel run times, and the number of significant digits in the output using OpenMP.

Function Name: Estimating Pi Using the Leibniz Formula.

Number of Significant Digits in the Output: Reflects the precision or accuracy of the result produced by the computation. Tasks with higher precision often require more iterations or computations, which directly impacts runtime.

If the comparison is extended to include the number of significant digits in the output along with parallel and non-parallel run times, here's how the relationship could be explained:

Number **of significant digits in the output**	**Parallel run time(seconds)**	**Non-parallel run time(seconds)**
4	8.3307	67.26
5	8.4589	81.008
9	8.5336	85.370
15	8.24	85.13
20	8.26	86.51
25	8.32	86.86
30	8.31	92

The results highlight the substantial benefits of parallel processing in improving computational efficiency. Most functions showed a significant decrease in run time when executed in parallel, with tasks like modular exponentiation and Euler's number calculation showing speedups of over 85%. Particularly, modular exponentiation achieved a reduction from 1.704 seconds to just 0.199 seconds, demonstrating the value of parallelization for repetitive and complex mathematical operations. Factorial calculation and matrix exponentiation also saw marked improvements. On the other hand, certain functions, like Mandelbrot set generation, showed relatively smaller performance gains, indicating that the efficiency of parallelization may vary depending on the specific nature of the task. The performance differences between two versions of programs: one using OpenMP for parallelization and one without OpenMP (sequential execution). The key aspects considered include run time, efficiency, and scalability.

Speedup Observed:

• The largest speedup is in the Calculation of Euler's Number (97.59s → 20.30s). Parallel execution drastically improves performance.
• Functions like Modular Exponentiation and Estimating Pi using Leibniz Formula also show significant gains.
• Functions like Mandelbrot Set Generation (9.059s → 8.974s) and Jacobi Iterative Method (0.513s → 0.417s) show minimal improvement. This could be due to inherent limitations in parallelization or overhead costs.
• Tasks like Estimating Pi and Euler's Number Calculation are computationally expensive but benefit greatly from parallel execution.
• Jacobi Iterative Method and Mandelbrot Set Generation see only slight improvements. These algorithms may involve data dependencies or require synchronization between threads, reducing the benefits of parallelization.
• The slight improvement in Mandelbrot Set Generation suggests that the overhead of managing parallel threads may outweigh the benefits for this specific task, especially if the workload per thread is small.

For comparison of **parallel run times, non-parallel run times,** and **the number of significant digits** in the output. Here's a detailed analysis:

• The parallel run time remains relatively stable (around 8.2–8.5 seconds) across all levels of precision, even as the number of significant digits increases.
• This suggests that parallelization handles the additional computational workload efficiently, with minimal overhead added for higher precision.
• Non-parallel run time increases significantly with the number of significant digits, growing from 67.26 seconds for 4 digits to 92 seconds for 30 digits.
• This reflects the increased computational complexity as higher precision requires more iterations or finer calculations, which accumulate in sequential execution.

REFERENCES

[1]. Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). Introduction to Parallel Computing. Addison-Wesley.
[2]. Chapman, B., Jost, G., & van der Pas, R. (2008). Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press.
[3]. Upton, E., & Halfacree, G. (2014). Raspberry Pi User Guide. Wiley.
[4]. Cox, S., Cox, J., Boardman, R., Johnston, S. J., & Scott, M. (2014). "Iridis-pi: a low-cost, compact demonstration cluster." Cluster Computing, 17(2), 349–358.
[5]. Doerfert, J., & Thoman, P. (2015). "Supporting multi-core parallelism on embedded systems using OpenMP." Proceedings of the Workshop on OpenMP Applications and Tools (pp. 1-10). ACM.
[6]. Pacheco, P. S. (2011). An Introduction to Parallel Programming. Morgan Kaufmann.
[7]. Borrmann, D., & Grimm, G. (2020). "Optimizing performance on low-cost hardware: Raspberry Pi and OpenMP." Journal of Embedded Computing and Applications, 12(3), 209–218.
[8]. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., & Menon, R. (2001). Parallel Programming in OpenMP. Morgan Kaufmann.
[9]. Hager, G., & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers. CRC Press.
[10]. OpenMP Architecture Review Board. (2020). The OpenMP API Specification for Parallel Programming. Version 5.1.