COMP1680 - Clouds, Grids and Virtualisation Coursework Report

Azhar Muhammed - 001364857 Word Count:

Part I

Parallel Processing using Cloud Computing

In the contemporary landscape of computational technologies, small consultancy firms face critical decisions regarding their computational infrastructure. The choice between adopting cloud computing services or investing in an onsite High-Performance Computing (HPC) system significantly impacts operational efficiency, scalability, and financial expenditure. This report evaluates these options for a consultancy company comprising 30 consultants utilizing approximately 1,400 CPU hours each month. The analysis focuses on the advantages and disadvantages of leading cloud service providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) when compared to an onsite HPC system for multicore workloads using batch processing. A comprehensive financial cost comparison is provided, culminating in a recommendation that aligns with the company's strategic objectives.

1. Analysis of Cloud Providers vs. Onsite HPC for Multicore Workloads

Cloud computing has revolutionized the way businesses handle computational workloads, offering scalable resources and flexible pricing models. In contrast, onsite HPC systems provide dedicated resources but require substantial initial investments and ongoing maintenance.

Table 1. Onsite HPC Costs Breakdown

Cost Item	Minimum ($)	Maximum ($)
Initial Setup Costs		
HPC Hardware	80,000	120,000
Cooling Systems	10,000	20,000
Power Supplies and UPS	5,000	10,000
Networking Equipment	5,000	10,000
Total Initial Setup Cost	**100,000**	**160,000**
Annual Operating Costs		
Maintenance and Support Contracts	10,000	15,000
Energy Consumption	756	756
IT Personnel Salary	50,000	70,000
Total Annual Operating Costs	**60,756**	**85,756**
Hardware Upgrade Costs (Every 4–5 Years)	50,000	112,000
Total 5-Year Cost	**453,780**	**700,780**

1.1. Advantages of Cloud Computing Over Onsite HPC

Cloud computing offers several advantages over onsite HPC systems, particularly for small consultancies:

- **Scalability and Flexibility:** Cloud services allow dynamic scaling of computational resources to match workload

demands. This elasticity is essential for batch processing tasks that may have variable computational needs (Armbrust et al. 2010).

- **Cost Efficiency:** With pay-as-you-go models, companies avoid the substantial upfront capital expenditure associated with purchasing HPC hardware. Operational costs are aligned with actual usage, enhancing financial predictability (Li, O'Brien, and Zhang 2019).
- **Maintenance and Operational Overhead:** Cloud providers handle hardware maintenance, software updates, and security patches, reducing the burden on internal IT staff (Marinescu 2013).
- **Access to Advanced Technologies:** Cloud platforms frequently update their offerings, providing access to the latest hardware and software innovations without additional costs (Dillon, Wu, and Chang 2010).

1.2. Disadvantages of Cloud Computing

Despite these advantages, cloud computing presents certain challenges:

- **Data Security and Compliance:** Relying on third-party providers raises concerns about data sovereignty and compliance with regulations such as GDPR (Hashem et al. 2015).
- **Performance Variability:** Multi-tenant cloud environments can lead to inconsistent performance due to resource contention (Leitner and Cito 2016).
- **Potential for Unexpected Costs:** Without proper management, costs can escalate due to factors like data egress fees and underutilized resources (Rehman, Wah, and Hussain 2018).

2. Comparison of Leading Cloud Providers

Amazon Web Services (AWS): AWS offers a broad range of services suitable for batch processing, such as AWS Batch and EC2 compute-optimized instances. It provides robust scalability and a variety of pricing models, including on-demand, reserved, and spot instances (Amazon Web Services n.d.). AWS's global infrastructure ensures high availability and low latency.

Microsoft Azure: Azure provides services like Azure Batch and compute-optimized virtual machines suitable for parallel processing tasks. It integrates seamlessly with other Microsoft products, which can be advantageous for companies already utilizing Microsoft ecosystems (Microsoft Azure n.d.). Azure offers competitive pricing and various discount options, such as reserved instances.

Google Cloud Platform (GCP): GCP's Compute Engine and preemptible virtual machines offer cost-effective solutions for batch processing workloads. GCP emphasizes sustained use discounts, automatically reducing costs for consistent usage without long-term commitments (Google Cloud Platform n.d.). Google's expertise in containerization and Kubernetes can benefit firms looking to adopt modern deployment practices.

Table 2. Cloud Computing Costs for AWS, Azure, and GCP

Cost Item	AWS	Azure	GCP
Instance Type	c6a.large	F2s_v2	n2-standard-2
vCPUs	2	2	2
Price per Hour ($)	0.085	0.085	0.084
Monthly Compute Cost ($)	119.00	119.00	117.60
Storage Cost per Month ($)	20.00	20.00	20.00
Data Transfer Cost per Month ($)	10.00	10.00	10.00
Total Monthly Cost ($)	149.00	149.00	147.60
Annual Cost ($)	1,788.00	1,788.00	1,771.20
5-Year Cost ($)	8,940.00	8,940.00	8,856.00

2.1. Comparison Between Cloud Providers

While all three providers offer similar core services, differences exist in pricing models, discount structures, and additional services:

- **Pricing Models:** AWS and Azure offer reserved instances for long-term commitments, while GCP provides sustained use discounts without upfront commitments.
- **Ecosystem Integration:** Azure may be preferable for companies using Microsoft tools, whereas AWS has a more extensive range of services and a mature ecosystem.
- **Specialized Services:** GCP excels in machine learning and big data analytics services, which could be beneficial depending on the consultancy's focus.

2.2. Advantages of Onsite HPC

Onsite HPC systems offer complete control over hardware and software configurations. They may provide performance benefits due to dedicated resources and reduced latency (Stergiou et al. 2018). Additionally, data remains on-premises, which can alleviate certain security and compliance concerns.

2.3. Disadvantages of Onsite HPC

The primary drawbacks of onsite HPC include:

- **High Initial Capital Expenditure:** Significant upfront investment is required for hardware, infrastructure, and installation.
- **Maintenance Costs:** Ongoing costs for maintenance, energy consumption, and staffing can be substantial.
- **Lack of Scalability:** Scaling up requires additional hardware purchases, leading to potential underutilization during periods of low demand.

2.4. Financial Cost Comparison

A detailed financial analysis over a five-year period highlights the cost implications of both options.

Table 3. Five-Year Total Cost Comparison

Option	**Minimum Total Cost ($)**	**Maximum Total Cost ($)**
Onsite HPC	453,780	700,780
Cloud Computing (AWS)	8,940	8,940
Cloud Computing (Azure)	8,940	8,940
Cloud Computing (GCP)	8,856	8,856
Cost Savings with Cloud Computing	**444,924**	**691,924**

2.5. Onsite HPC Costs

Initial Setup Costs: The purchase of HPC hardware, including high-performance servers, networking equipment, and storage systems, is estimated between $80,000 and $120,000. Infrastructure costs, encompassing cooling systems, power supplies, and additional networking equipment, add another $20,000 to $40,000. Thus, the total initial setup cost ranges from $100,000 to $160,000.

Annual Operating Costs: Maintenance and support contracts cost between $10,000 and $15,000 annually. Energy consumption is calculated based on an estimated 0.3 kWh per CPU hour at $0.15 per kWh, resulting in an annual energy cost of $756. Staffing requires a part-time HPC specialist, with salary expenses ranging from $50,000 to $70,000 per year. The total annual operating costs amount to $60,756 to $85,756.

Hardware Upgrade Costs: Upgrades are anticipated every 4-5 years, costing 50-70% of the initial hardware investment, equating to $50,000 to $112,000.

Total Five-Year Cost: Combining the initial setup, operating costs over five years, and hardware upgrades, the total expenditure ranges from $453,780 to $700,780.

2.6. Cloud Computing Costs

Amazon Web Services (AWS): To fulfill the requirement of 1,400 CPU hours per month, AWS offers the c6a.large instance at $0.085 per hour. The monthly compute cost is $119, with additional storage and data transfer costs estimated at $30 per month. The total monthly cost is $149, leading to an annual cost of $1,788 and a five-year cost of $8,940. Potential discounts through reserved instances or spot instances can further reduce expenses.

Microsoft Azure: Azure's F2s_v2 instance, also priced at approximately $0.085 per hour, results in identical compute costs to AWS. Including storage and data transfer, the total monthly cost is $149, with a five-year cost of $8,940.

Google Cloud Platform (GCP): GCP's n2-standard-2 instance costs $0.084 per hour, leading to a monthly compute cost of $117.60. Including storage and data transfer, the total monthly cost is $147.60. The annual cost is $1,771.20, and the five-year cost is $8,856. Sustained use discounts may reduce costs by up to 30%.

2.7. Total Cost Comparison

Over five years, the total cost for cloud computing ranges from $8,856 to $8,940, significantly lower than the onsite HPC cost of $453,780 to $700,780. The cost savings with cloud computing amount to $444,924 to $691,924 over five years.

3. Recommendation

Based on the analysis of operational and financial factors, it is recommended that the consultancy company adopts a cloud computing solution for its parallel processing needs.

Significant Cost Savings: The financial comparison demonstrates substantial cost savings with cloud computing, reducing expenditures by over $440,000 over five years. This capital can be allocated to other strategic initiatives within the company.

Operational Efficiency: Cloud computing minimizes the need for in-house IT maintenance, allowing consultants to focus on core business activities. The responsibility for hardware maintenance and updates lies with the cloud provider, enhancing operational efficiency.

Flexibility and Scalability: Cloud services offer unparalleled flexibility, with the ability to adjust resources to match workload demands seamlessly. This scalability supports business growth without the need for additional infrastructure investment.

Risk Mitigation: Cloud computing eliminates risks associated with hardware failures and obsolescence. Providers offer high availability and disaster recovery solutions, ensuring business continuity.

3.1. Platform Selection

While all three major cloud providers offer services that meet the company's needs, slight distinctions may influence the decision:

- **AWS:** With its extensive range of services and mature ecosystem, AWS is suitable for companies seeking a broad array of tools and global infrastructure.
- **Azure:** For firms already utilizing Microsoft products, Azure offers seamless integration and may provide operational efficiencies.
- **GCP:** Companies interested in advanced analytics and machine learning capabilities may prefer GCP due to its strengths in these areas.

Given the consultancy's size and requirements, any of these providers could be appropriate. A detailed assessment of

existing partnerships, staff expertise, and specific service offerings should guide the final selection. The consultancy should also consider any existing vendor relationships or preferred ecosystems when making the final decision.

3.2. Implementation Considerations

To maximize the benefits of cloud computing, the company should take the following steps during the implementation phase:

- **Leverage Cost Savings Opportunities:** Utilize reserved instances or savings plans for predictable workloads and explore spot instances for non-critical tasks to reduce costs.
- **Implement Cost Management Practices:** Use cloud monitoring tools to track resource utilization, set alerts for unexpected cost spikes, and establish budgets to manage cloud spending effectively.
- **Ensure Security and Compliance:** Establish robust cloud security policies, implement encryption for sensitive data, and leverage the cloud provider's compliance certifications to meet regulatory requirements.
- **Provide Staff Training:** Equip consultants with the necessary skills to use cloud services effectively, ensuring a smooth transition and minimizing resistance to the new system.

Table 4. Comparison of Cloud Computing and Onsite HPC

Criteria	Cloud Computing	Onsite HPC
Scalability and Flexibility	High; resources can be scaled up or down easily	Limited; scaling requires hardware investment
Initial Capital Expenditure	Low; pay-as-you-go model	High; significant upfront costs
Maintenance Overhead	Low; provider handles maintenance	High; requires dedicated IT staff
Risk of Obsolescence	Low; access to latest technology	High; hardware can become outdated
Energy Consumption	Lower; optimized data centers	Higher; increased operational costs
Data Control	Less control; data stored off-premises	Full control; data stored on-site
Security and Compliance	High; robust security measures and certifications	Dependent on in-house capabilities
Total Cost over 5 Years	$8,856 – $8,940	$453,780 – $700,780

3.3. Conclusion

Transitioning to cloud computing is a strategic decision that aligns with the consultancy company's operational needs and financial objectives. The substantial cost savings, coupled with enhanced scalability and reduced maintenance burdens, make cloud services the optimal choice over an onsite HPC system. By adopting cloud computing, the company positions itself to leverage advanced technologies, respond agilely to market demands, and focus its resources on delivering value to clients.

Part II

Parallel Programming Exercise

4. Step 1: Implementation and Optimization of Sequential Code

The C program was adapted to simulate the temperature distribution in a rectangular 2D grid with specific boundary conditions: the top boundary at 15°C, the bottom at 60°C, the left at 47°C, and the right at 100°C. The updated code uses a global preprocessor directive, `#define` keyword that creates a constant macro. These macros allow the program to use meaningful names throughout the code instead of magic numbers, making the code more readable and maintainable. When the code is compiled, the preprocessor replaces every occurrence of these macro names with their corresponding numerical values. These conditions were implemented by setting the corresponding edges of the grid to the specified temperatures.

In the code, after initializing the entire grid to an initial temperature (e.g., 30°C), the boundary conditions are applied:

```
// Set boundary conditions
for (int i = 1; i <= m; i++) {
t[i][0] = LEFT_TEMP; // Left boundary set to 47C
t[i][n + 1] = RIGHT_TEMP; // Right boundary set to 100C
}
for (int j = 1; j <= n; j++) {
t[0][j] = TOP_TEMP; // Top boundary set to 15C
t[m + 1][j] = BOTTOM_TEMP; // Bottom boundary set to 60C
}


// Set corner temperatures as average of adjacent boundaries
t[0][0] = (TOP_TEMP + LEFT_TEMP) / 2.0; // Top-left corner
t[0][n + 1] = (TOP_TEMP + RIGHT_TEMP) / 2.0; // Top-right corner
t[m + 1][0] = (BOTTOM_TEMP + LEFT_TEMP) / 2.0; // Bottom-left corner
t[m + 1][n + 1] = (BOTTOM_TEMP + RIGHT_TEMP) / 2.0; // Bottom-right corner
```

These code snippets ensure that the boundary temperatures are correctly set according to the problem's specifications. The corners are handled separately by averaging the temperatures of the adjacent boundaries to avoid anomalies at those points.

4.1. Execution Time

To measure the execution time of the program, the `time.h` library is included, and timing functions are used:

```
#include <time.h> // For measuring execution time

// Start timing the execution
clock_t start = clock();

// ... [Jacobi iteration loop] ...

// Calculate execution time
clock_t end = clock();
double cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

// Print results
printf("Execution time: %f seconds\n", cpu_time_used);
```

Listing 1. Execution Time Measurement

The `clock()` function records the processor time consumed by the program. By capturing the time before and after the main computation loop and calculating the difference, the execution time is obtained. This measurement is crucial for comparing performance across different grid sizes and compiler optimization levels.

4.2. Adjustments for Realistic Execution Time Measurements

To obtain accurate execution time measurements, it is important to prevent the program from printing large amounts of data, which can significantly affect performance. In the code, the grid values are only printed if the grid size is small (e.g., 10x10 or smaller):

```
// Print grid values for small grids (10x10 or smaller)
if (m <= 10 && n <= 10) {
    for (int i = 0; i <= m + 1; i++) {
        for (int j = 0; j <= n + 1; j++) {
            printf("%3.5lf ", t[i][j]);
        }
        printf("\n");
    }
}
```

Listing 2. Printing Grid Values for Small Grids

By restricting the output for larger grids, the program avoids the overhead associated with I/O operations, leading to more accurate timing of the computational aspects.

4.3. Automation of Testing with Bash Script

The provided Bash script automates the compilation and execution of the program across different optimization levels (`-O0` to `-O3`) and grid sizes (150x150, 200x200, 250x250). The script performs the following tasks:

1. **Compiler Selection:**

The script checks for the availability of `gcc-14`; if not found, it defaults to `gcc`:

```
if command -v gcc-14 &> /dev/null; then
    COMPILER=gcc-14
else
    COMPILER=gcc
fi
```

Listing 3. Compiler Selection

2. **Compilation with Different Optimization Levels:**

The program is compiled multiple times with different optimization flags:

```
for optmlvl in 0 1 2 3; do
    $COMPILER -std=c99 -O${optmlvl} jacobi2d-Step1.c -o jacobi2d-Step1-O${optmlvl}
    # Error checking omitted for brevity
done
```

Listing 4. Compilation with Different Optimization Levels

3. **Execution Across Grid Sizes and Optimization Levels:**

The script runs the compiled programs for each combination of grid size and optimization level, capturing the output:

```
for size in 150 200 250; do
    for optmlvl in 0 1 2 3; do
        echo "Testing grid size of ${size}x${size} with optimization level -O${optmlvl}" >> jacobi2d-Step1.txt
        ./jacobi2d-Step1-O${optmlvl} $size $size 0.000100 >> jacobi2d-Step1.txt
        echo "---------------------------------------" >> jacobi2d-Step1.txt
    done
done
```

Listing 5. Automated Testing Loop for Grid Sizes and Optimization Levels

This automated approach ensures consistency and efficiency in data collection, facilitating the analysis of how different optimization levels impact execution time for various grid sizes.

4.4. Results and Observations

The output collected demonstrates how compiler optimization levels affect the program's performance. Generally, higher optimization levels (`-O1`, `-O2`, `-O3`) result in reduced execution times compared to no optimization (`-O0`). The data also shows that as the grid size increases, the execution time increases, which is expected due to the larger computational workload.

4.5. Considerations Regarding Compiler Optimizations

It is noted that aggressive compiler optimizations can potentially alter the program's behavior. In this case, the program was tested across all optimization levels without any observed issues. However, caution is advised, and validation checks should be included to ensure that the optimizations do not introduce unintended side effects.

Table 5. Execution Times for Different Grid Sizes and Optimization Levels (`-O`N)

Grid Size	Iterations	Maximum Difference ($difmax$)	-O0 (s)	-O1 (s)	-O2 (s)	-O3 (s)
150x150	20,763	9.999041×10^{-5}	5.250	1.570	1.090	1.060
200x200	32,108	9.999248×10^{-5}	11.840	4.290	2.610	2.840
250x250	44,398	9.999424×10^{-5}	25.720	9.420	6.290	6.050

4.6. Conclusion

The modifications to the program successfully implemented the specified boundary conditions and enabled the measurement of execution time. The use of a Bash script to automate testing across different grid sizes and optimization levels provided valuable insights into the program's performance characteristics. These results form a solid foundation for further analysis and optimization in subsequent steps of the coursework.

5. Step 2: OpenMP Parallelization

The Jacobi 2D grid problem was extended from its sequential implementation to a parallel version using OpenMP. The primary objective was to modify the existing code to incorporate OpenMP directives for parallel execution, compile it with OpenMP support, and test its correctness using multiple threads. The performance was compared across different thread counts to observe the scalability and correctness of the parallel implementation.

5.1. Code Modifications for Parallel Execution

The original sequential code was adapted to leverage OpenMP for parallel processing. Key modifications were made to parallelize the computation-intensive sections while ensuring thread safety and correctness.

1. **Including OpenMP Header**

The OpenMP library was included by adding the header file:

```
#include <omp.h> // Added OpenMP library for parallel execution
```

Listing 6. OpenMP Header Inclusion

2. **Parallel Initialization**

The initialization of the temperature grids was parallelized using OpenMP directives (Chapman, Jost, and Pas 2007). The `#pragma omp parallel` directive was used to define a parallel region, and loops were parallelized using `#pragma omp for` with appropriate scheduling:

```
#pragma omp parallel
{
    // Initialize interior points to 30.0
    #pragma omp for collapse(2) schedule(static) nowait
    // ... [initializing interior points] ...

    // Set boundary temperatures
    #pragma omp for schedule(static) nowait
    // ... [setting left and right boundary temperatures] ...

    #pragma omp for schedule(static)
    // ... [setting top and bottom boundary temperatures] ...
}
```

Listing 7. Parallel Initialization

- **Use of `collapse(2)`:** This clause was used to collapse the two nested loops into a single loop for parallel execution (*OpenMP Application Program Interface* n.d.), enhancing load balancing among threads.
- **Scheduling and `nowait`:** Static scheduling was chosen for simplicity, and the `nowait` clause was used to eliminate unnecessary barriers, improving performance.

3. **Parallel Main Iteration Loop:**

The main computation loop, which performs the Jacobi iteration, was parallelized:

```
while (difmax > tol) {
    iter++;
    difmax = 0.0;
```

```
    #pragma omp parallel
    {
        // Compute new temperature values
        #pragma omp for private(j) schedule(static) nowait
        // ... [Compute new temperature values] ...

        // Update temperatures and compute maximum difference
        #pragma omp for private(j, diff) reduction(max:difmax)
        // ... [Update temperatures and compute maximum difference] ...
    }
}
```

Listing 8. Parallel Main Iteration Loop

- **Private Variables:** Variables `j` and `diff` were declared as private to ensure thread safety.
- **Reduction Clause:** The `reduction(max:difmax)` clause was used to correctly compute the maximum difference across all threads (Chapman, Jost, and Pas 2007).
- **No Wait Clauses:** The `nowait` clause was employed to eliminate unnecessary synchronization barriers between loops.

4. **Timing with OpenMP Functions:**

The execution time was measured using OpenMP's timing functions (*OpenMP Application Program Interface* n.d.):

```
double start_time = omp_get_wtime();
// ... [computation] ...
double exec_time = omp_get_wtime() - start_time;
printf("Execution time: %f seconds\n", exec_time);
```

Listing 9. Timing with OpenMP Functions

The use of `omp_get_wtime()` provides high-resolution wall-clock timing suitable for parallel regions.

5. **Memory Allocation Optimization:**

The arrays were allocated using contiguous memory blocks to improve cache performance:

```
double *t_data = (double *)malloc((m + 2) * (n + 2) * sizeof(double));
double *tnew_data = (double *)malloc((m + 2) * (n + 2) * sizeof(double));
double **t = (double **)malloc((m + 2) * sizeof(double *));
double **tnew = (double **)malloc((m + 2) * sizeof(double *));

// Setup 2D array pointers
for (i = 0; i < m + 2; i++) {
    t[i] = &t_data[i * (n + 2)];
    tnew[i] = &tnew_data[i * (n + 2)];
}
```

Listing 10. Memory Allocation Optimization

This approach reduces memory fragmentation and enhances data locality, which is beneficial for parallel execution.

5.2. Automation of Testing with Bash Script

To systematically test the parallel program across different thread counts, a Bash script was employed. The script automates the compilation and execution process, ensuring consistent and repeatable tests.

1. **Execution Across Thread Counts:**

The script runs the program for a grid size of 20x20 using 1, 2, 4, and 8 threads:

```
for size in 20; do
    for threads in 1 2 4 8; do
        echo "Testing grid size of ${size}x${size} with ${threads} threads" >> jacobi2d-Step2.txt
        OMP_NUM_THREADS=$threads ./jacobi2d-Step2 $size $size 0.000100 >> jacobi2d-Step2.txt
        echo "----------------------------------------" >> jacobi2d-Step2.txt
    done
done
```

Listing 11. Execution Across Thread Counts

- **Environment Variable `OMP_NUM_THREADS`:** This variable sets the number of threads for OpenMP at runtime.
- **Output Redirection:** The results are appended to `jacobi2d-Step2.txt` for later analysis.

5.3. Results and Observations

The output obtained from running the tests demonstrates the correctness and performance of the parallel implementation. For a grid size of 20x20, the program was executed using 1, 2, 4, and 8 threads. The iteration counts, maximum differences, and execution times were recorded.

Table 6. Performance Comparison for Different Thread Counts

Thread Count	Iterations	Maximum Difference ($difmax$)	Execution Time (s)
1	752	9.929954×10^{-5}	0.009949
2	754	9.832363×10^{-5}	0.008693
4	740	9.894563×10^{-5}	0.006277
8	736	9.817051×10^{-5}	0.005612

Correctness Across Threads: The iteration counts and maximum differences are consistent across different thread counts, indicating that the parallel code operates correctly.

Performance Improvement: There is a noticeable reduction in execution time as the number of threads increases, demonstrating the benefit of parallelization.

Diminishing Returns: The speedup from 4 to 8 threads is less significant than from 2 to 4 threads, which may be due to overheads associated with thread management and the small problem size.

For grids of size 20x20 or smaller, the program prints the final temperature distribution. This way, the program can be used to visualize the temperature distribution for small grids while maintaining performance for larger grids.

```
if (m <= 20 && n <= 20) {
    // .. [print grid] ...
}
```

Listing 12. Conditional Printing of Grid Values

Here are the of the output grids produced by the Jacobi program is as follows:

```
------------------------------
Testing size 20x20 with 1 threads
20 20 0.000100
iter = 752  difmax = 0.00009929954
31.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 57.50000
47.00000 31.24365 25.16588 22.44545 21.08222 20.35954 19.98688 19.83356 19.83714 19.96860 20.21830 20.59062 21.10331 21.79070 22.71210 23.96998 25.74980 28.41402 32.74671 40.66155 57.47990 100.00000
47.00000 37.80873 31.97444 28.53372 26.52393 25.36911 24.75445 24.51027 24.54645 24.81902 25.31404 26.04093 27.03199 28.34745 30.08778 32.41807 35.61523 40.15961 46.91131 57.41962 74.25805 100.00000
47.00000 41.01685 36.38945 33.19113 31.11072 29.83861 29.15164 28.90672 29.01947 29.44711 30.17803 31.22718 32.63637 34.47944 36.87359 39.99937 44.13351 49.69796 57.31933 67.84762 82.13268 100.00000
47.00000 42.86926 39.37541 36.73072 34.88929 33.72310 33.10692 32.94563 33.17777 33.77210 34.72396 36.05356 37.80704 40.06052 42.92791 46.57246 51.22159 57.17947 64.82052 74.51890 86.42507 100.00000
47.00000 44.08482 41.51229 39.46714 37.99273 37.05773 36.60747 36.59131 36.97408 37.73977 38.89235 40.45630 42.47791 45.02792 48.20525 52.14116 57.00107 62.97795 70.26446 78.98246 89.04872 100.00000
47.00000 44.95777 43.12187 41.63294 40.55692 39.90781 39.67414 39.83829 40.38773 41.32082 42.64966 44.40163 46.62065 49.36823 52.72425 56.78606 61.66375 67.46696 74.27703 82.09783 90.78741 100.00000
47.00000 45.62444 44.38456 43.38595 42.69439 42.34265 42.34323 42.70024 43.41801 44.50642 45.98413 47.88024 50.23512 53.10040 56.53771 60.61534 65.40112 70.94929 77.27901 84.34450 92.00314 100.00000
47.00000 46.15549 45.40607 44.83205 44.49225 44.42540 44.65614 45.20175 46.07796 47.30307 48.90054 50.90041 53.33954 56.26086 59.71114 63.73672 68.37634 73.65026 79.54536 85.99812 92.88071 100.00000
47.00000 46.59152 46.25228 46.04410 46.01735 46.21083 46.65447 47.37297 48.38935 49.72770 51.41493 53.48168 55.96212 58.89270 62.30960 66.24434 70.71752 75.73026 81.25422 87.22200 93.52165 100.00000
47.00000 46.95837 46.96753 47.07489 47.32243 47.74637 48.37823 49.24663 50.37914 51.80381 53.55019 55.64966 58.13493 61.03860 64.39053 68.21383 72.51941 77.29925 82.51941 88.11414 93.98395 100.00000
47.00000 47.27449 47.58470 47.96568 48.45134 49.07424 49.86576 50.85654 52.07714 53.55860 55.33276 57.43222 59.88972 62.73660 66.00043 69.70136 73.84728 78.42817 83.41020 88.73131 94.30007 100.00000
47.00000 47.55496 48.13121 48.75196 49.44324 50.23378 51.15432 52.23696 53.51466 55.02107 56.79041 58.85716 61.25550 64.01801 67.17359 70.74420 74.74047 79.15615 83.96207 89.10093 94.48509 100.00000
47.00000 47.81418 48.63333 49.46788 50.33610 51.26356 52.28109 53.42265 54.72382 56.22101 57.95102 59.95089 62.25748 64.90672 67.93204 71.36167 75.21450 79.49411 84.18119 89.22538 94.53940 100.00000
47.00000 48.06849 49.12017 50.15027 51.16991 52.20354 53.28413 54.44903 55.73730 57.18847 58.84216 60.73826 62.91717 65.41970 68.28650 71.55624 75.26201 79.42481 84.04333 89.08012 94.44719 100.00000
47.00000 48.33965 49.62871 50.84327 51.98994 53.09679 54.20313 55.35234 56.58822 57.95373 59.49121 61.24318 63.25356 65.56872 68.23830 71.31505 74.85273 78.89997 83.48737 88.60467 94.16929 100.00000
47.00000 48.66145 50.21182 51.60431 52.84994 53.99075 55.07950 56.16926 57.30979 58.54734 59.92607 61.49001 63.28548 65.36362 67.78320 70.61318 73.93409 77.83516 82.40163 87.68199 93.62535 100.00000
47.00000 49.09438 50.95288 52.51234 53.81492 54.93697 55.95506 56.93564 57.93457 59.00004 60.17597 61.50558 63.03498 64.81733 66.91791 69.42061 72.43546 76.10510 80.60214 86.09637 92.65018 100.00000
47.00000 49.76321 51.99306 53.67734 54.96054 55.98728 56.86833 57.68384 58.49304 59.34250 60.27243 61.32158 62.53175 63.95301 65.65072 67.71606 70.28218 73.54779 77.80554 83.45123 90.87903 100.00000
47.00000 50.96544 53.57884 55.24349 56.36274 57.18341 57.84725 58.43852 59.01140 59.60464 60.24984 60.97673 61.81760 62.81240 64.01603 65.51086 67.42952 69.99842 73.62108 79.02403 87.41475 100.00000
47.00000 53.51971 56.11341 57.35511 58.06356 58.53644 58.89884 59.21170 59.50950 59.81494 60.14568 60.51800 60.94965 61.46306 62.09024 62.88194 63.92670 65.39536 67.65638 71.60909 79.75596 100.00000
53.50000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 80.00000
Execution time: 0.009949 seconds
------------------------------
```

Figure 1. Screenshot of Thread Count 1

```
------------------------------
Testing size 20x20 with 2 threads
20 20 0.000100
iter = 754  difmax = 0.00009832363
31.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 57.50000
47.00000 31.24366 25.16589 22.44547 21.08224 20.35957 19.98691 19.83359 19.83717 19.96864 20.21834 20.59066 21.10335 21.79074 22.71213 23.97001 25.74982 28.41404 32.74673 40.66157 57.47990 100.00000
47.00000 37.80874 31.97446 28.53376 26.52397 25.36916 24.75451 24.51033 24.54652 24.81909 25.31412 26.04100 27.03206 28.34752 30.08784 32.41812 35.61528 40.15965 46.91134 57.41965 74.25806 100.00000
47.00000 41.01687 36.38948 33.19118 31.11078 29.83869 29.15173 28.90681 29.01957 29.44722 30.17814 31.22729 32.63648 34.47954 36.87368 39.99946 44.13358 49.69802 57.31938 67.84765 82.13269 100.00000
47.00000 42.86928 39.37545 36.73078 34.88936 33.72319 33.10703 32.94575 33.17790 33.77224 34.72410 36.05370 37.80718 40.06066 42.92803 46.57257 51.22169 57.17955 64.82058 74.51894 86.42509 100.00000
47.00000 44.08485 41.51234 39.46721 37.99283 37.05785 36.60760 36.59146 36.97424 37.73994 38.89252 40.45647 42.47808 45.02807 48.20540 52.14129 57.00118 62.97805 70.26453 78.98251 89.04875 100.00000
47.00000 44.95780 43.12192 41.63302 40.55703 39.90794 39.67429 39.83846 40.38791 41.32101 42.64985 44.40183 46.62084 49.36841 52.72442 56.78622 61.66388 67.46707 74.27712 82.09789 90.78744 100.00000
47.00000 45.62447 44.38462 43.38604 42.69452 42.34280 42.34340 42.70043 43.41821 44.50664 45.98435 47.88046 50.23534 53.10060 56.53790 60.61551 65.40127 70.94941 77.27910 84.34456 92.00318 100.00000
47.00000 46.15553 45.40614 44.83215 44.49238 44.42556 44.65632 45.20195 46.07818 47.30330 48.90078 50.90064 53.33977 56.26108 59.71135 63.73690 68.37651 73.65040 79.54546 85.99819 92.88075 100.00000
47.00000 46.59156 46.25235 46.04421 46.01749 46.21100 46.65466 47.37318 48.38958 49.72794 51.41518 53.48193 55.96236 58.89294 62.30981 66.24454 70.71769 75.73040 81.25433 87.22208 93.52169 100.00000
47.00000 46.95841 46.96761 47.07501 47.32258 47.74654 48.37843 49.24686 50.37938 51.80406 53.55045 55.64991 58.13518 61.03883 64.39075 68.21403 72.51958 77.29940 82.51952 88.11421 93.98399 100.00000
47.00000 47.27453 47.58477 47.96579 48.45149 49.07442 49.86596 50.85676 52.07738 53.55885 55.33301 57.43248 59.88996 62.73683 66.00065 69.70156 73.84746 78.42831 83.41031 88.73138 94.30011 100.00000
47.00000 47.55499 48.13128 48.75207 49.44338 50.23394 51.15451 52.23717 53.51489 55.02132 56.79065 58.85740 61.25574 64.01824 67.17380 70.74439 74.74064 79.15629 83.96218 89.10101 94.48512 100.00000
47.00000 47.81422 48.63340 49.46798 50.33623 51.26372 52.28128 53.42285 54.72404 56.22123 57.95126 59.95112 62.25771 64.90694 67.93224 71.36186 75.21466 79.49424 84.18129 89.22545 94.53944 100.00000
47.00000 48.06852 49.12024 50.15037 51.17003 52.20369 53.28430 54.44922 55.73751 57.18868 58.84237 60.73848 62.91738 65.41990 68.28668 71.55641 75.26216 79.42493 84.04343 89.08018 94.44722 100.00000
47.00000 48.33968 49.62876 50.84336 51.99005 53.09692 54.20328 55.35251 56.58840 57.95392 59.49140 61.24338 63.25375 65.56890 68.23847 71.31521 74.85286 78.90008 83.48745 88.60473 94.16932 100.00000
47.00000 48.66148 50.21187 51.60439 52.85004 53.99087 55.07963 56.16940 57.30994 58.54751 59.92624 61.49018 63.28564 65.36378 67.78334 70.61332 73.93420 77.83526 82.40171 87.68204 93.62538 100.00000
47.00000 49.09440 50.95292 52.51240 53.81499 54.93706 55.95517 56.93576 57.93470 59.00018 60.17611 61.50572 63.03512 64.81746 66.91804 69.42072 72.43555 76.10518 80.60220 86.09641 92.65020 100.00000
47.00000 49.76323 51.99309 53.67739 54.96060 55.98736 56.86841 57.68394 58.49314 59.34260 60.27254 61.32169 62.53186 63.95311 65.65081 67.71614 70.28225 73.54785 77.80559 83.45126 90.87905 100.00000
47.00000 50.96545 53.57886 55.24353 56.36278 57.18346 57.84731 58.43859 59.01146 59.60471 60.24991 60.97680 61.81767 62.81247 64.01609 65.51092 67.42957 69.99846 73.62111 79.02405 87.41476 100.00000
47.00000 53.51971 56.11342 57.35513 58.06358 58.53647 58.89887 59.21173 59.50953 59.81498 60.14572 60.51804 60.94969 61.46309 62.09028 62.88197 63.92672 65.39539 67.65639 71.60910 79.75596 100.00000
53.50000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 80.00000
Execution time: 0.008693 seconds
------------------------------
```

Figure 2. Screenshot of Thread Count 2

```
------------------------------
Testing size 20x20 with 4 threads
20 20 0.000100
iter = 740  difmax = 0.00009894563
31.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 57.50000
47.00000 31.19971 25.07946 22.31940 20.92061 20.16747 19.77020 19.59861 19.59047 19.71669 19.96735 20.34637 20.87091 21.57461 22.51609 23.79713 25.60255 28.29426 32.65580 40.60045 57.44920 100.00000
47.00000 37.71938 31.79875 28.27754 26.19562 24.97912 24.31476 24.03383 24.04662 24.30900 24.80641 25.54729 26.56271 27.91149 29.69266 32.06992 35.31886 39.91870 46.72853 57.29682 74.19637 100.00000
47.00000 40.87909 36.11864 32.79645 30.60526 29.23871 28.47598 28.17544 28.25328 28.66638 29.40210 30.47379 31.92125 33.81610 36.27326 39.47111 43.68436 49.33322 57.04284 67.66197 82.03947 100.00000
47.00000 42.67836 39.00032 36.18443 34.19035 32.89458 32.17513 31.93883 32.12483 32.70131 33.66199 35.02468 36.83256 39.15855 42.11330 45.85702 50.61435 56.68709 64.44770 74.26879 86.29957 100.00000
47.00000 43.83408 41.01990 38.75069 37.07727 35.97428 35.39130 35.28010 35.60610 36.35225 37.52009 39.13060 41.22596 43.87244 47.16457 51.22951 56.22907 62.35321 69.79218 78.66600 88.89004 100.00000
47.00000 44.63811 42.49458 40.72128 39.39391 38.53415 38.13590 38.18438 38.66748 39.58176 40.93576 42.75195 45.06849 47.94092 51.44326 55.66757 60.71938 66.70466 73.70193 81.71304 90.59465 100.00000
47.00000 45.22384 43.59910 42.24607 41.24310 40.63272 40.43401 40.65432 41.29793 42.37186 43.88955 45.87323 48.35544 51.37977 55.00024 59.27838 64.27643 70.04428 76.59798 83.88969 91.77556 100.00000
47.00000 45.65817 44.43203 43.42096 42.69988 42.31984 42.31337 42.70123 43.49836 44.71853 46.37770 48.49630 51.10059 54.22280 57.89983 62.16952 67.06390 72.59824 78.75616 85.47226 92.61794 100.00000
47.00000 45.97687 45.05001 44.30599 43.81583 43.63362 43.79869 44.33920 45.27609 46.62655 48.40676 50.63406 53.32816 56.51135 60.20705 64.43628 69.21164 74.52883 80.35632 86.62536 93.22401 100.00000
47.00000 46.19938 45.48524 44.93734 44.62403 44.60039 44.90885 45.58110 46.64062 48.10517 49.98912 52.30538 55.06701 58.28775 61.98108 66.15719 70.81782 75.94933 81.51508 87.44896 93.65278 100.00000
47.00000 46.33545 45.75434 45.33427 45.14276 45.23534 45.65553 46.43606 47.60045 49.16478 51.13953 53.53172 56.34712 59.59191 63.27265 67.39388 71.95338 76.93581 82.30586 88.00275 93.93820 100.00000
47.00000 46.38814 45.86252 45.50280 45.37762 45.54292 46.04218 46.90750 48.16071 49.81434 51.87289 54.33523 57.19821 60.46048 64.12408 68.19260 72.66628 77.53489 82.76998 88.31809 94.09732 100.00000
47.00000 46.35463 45.80491 45.43696 45.32222 45.51680 46.06305 46.99137 48.32090 50.05934 52.20284 54.73847 57.65037 60.92808 64.57090 68.58646 72.98448 77.76770 82.92126 88.40241 94.13307 100.00000
47.00000 46.22554 45.56563 45.11805 44.95772 45.13925 45.70214 46.67432 48.07254 49.89961 52.14103 54.76579 57.73709 61.03089 64.64529 68.59813 72.91775 77.63035 82.74510 88.23732 94.03258 100.00000
47.00000 45.98193 45.11413 44.51204 44.25153 44.38057 44.93218 45.93153 47.39563 49.32586 51.69619 54.44692 57.50162 60.81342 64.38152 68.24328 72.45826 77.09105 82.19162 87.76928 93.76000 100.00000
47.00000 45.58811 44.39699 43.56460 43.15596 43.19954 43.71473 44.72424 46.25289 48.31228 50.87126 53.82438 57.00935 60.33992 63.82435 67.53545 71.58117 76.08414 81.16118 86.88827 93.23819 100.00000
47.00000 44.97356 43.32121 42.19351 41.60830 41.54708 42.00314 42.99806 44.57964 46.79938 49.65244 52.97025 56.37174 59.71281 63.04075 66.49321 70.24701 74.50332 79.48078 85.38452 92.30452 100.00000
47.00000 43.98496 41.72084 40.28001 39.53679 39.37748 39.75288 40.68539 42.26844 44.65338 47.96909 52.03266 55.79475 59.09902 62.13282 65.14978 68.41048 72.20146 76.87421 82.86458 90.59540 100.00000
47.00000 42.24548 39.29721 37.66900 36.88146 36.67327 36.94565 37.72231 39.15551 41.57677 45.53804 51.39672 55.67575 58.75587 61.24186 63.56276 66.04377 69.01794 72.95010 78.60422 87.21254 100.00000
47.00000 38.69976 35.55358 34.21735 33.64684 33.48858 33.63424 34.10278 35.05462 36.96026 41.20969 52.34054 56.75579 59.00693 60.51611 61.81570 63.18398 64.87649 67.30407 71.38971 79.65056 100.00000
53.50000 30.00000 30.00000 30.00000 30.00000 30.00000 30.00000 30.00000 30.00000 30.00000 30.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 80.00000
Execution time: 0.006277 seconds
------------------------------
```

Figure 3. Screenshot of Thread Count 4

```
============================
Testing size 20x20 with 8 threads
20 20 0.000100
iter = 736  difmax = 0.00009817051
31.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 15.00000 57.50000
47.00000 31.22252 25.12448 22.38545 21.00595 20.26985 19.88693 19.72665 19.72653 19.85736 20.10919 20.48605 21.00528 21.70082 22.63162 23.89983 25.69064 28.36631 32.71074 40.63749 57.46784 100.00000
47.00000 37.76560 31.88998 28.41138 26.36853 25.18655 24.55125 24.29320 24.32217 24.59378 25.09342 25.82977 26.83430 28.16643 29.92588 32.27710 35.49646 40.06389 46.83918 57.37140 74.23390 100.00000
47.00000 40.94994 36.25848 33.00160 30.87031 29.55665 28.83841 28.57283 28.67529 29.10226 29.84107 30.90543 32.33582 34.20482 36.62846 39.78631 43.95427 49.55369 57.21073 67.77506 82.09636 100.00000
47.00000 42.77569 39.19244 36.46631 34.55456 33.33145 32.67304 32.48456 32.70403 33.29905 34.26333 35.61521 37.39890 39.68874 42.59698 46.28554 50.98074 56.98596 64.67505 74.42181 86.37650 100.00000
47.00000 43.96043 41.26933 39.11673 37.55028 36.54169 36.03790 35.98852 36.35744 37.12680 38.29819 39.89339 41.95605 44.55444 47.78535 51.77829 56.69734 62.73449 70.08178 78.86070 88.98787 100.00000
47.00000 44.79675 42.80782 41.18109 39.98829 39.24730 38.94856 39.07440 39.61062 40.55279 41.90948 43.70436 45.97773 48.78786 52.21193 56.34514 61.29599 67.17304 74.05700 81.95141 90.71430 100.00000
47.00000 45.41877 43.98419 42.81166 41.97465 41.51087 41.43488 41.75014 42.45815 43.56453 45.08288 47.03715 49.46291 52.40763 55.92960 60.09458 64.96863 70.60486 77.02187 84.17374 91.91798 100.00000
47.00000 45.89421 44.89860 44.10685 43.58797 43.38687 43.53018 44.03341 44.90761 46.16461 47.82067 49.89876 52.42944 55.45046 59.00455 63.13520 67.87930 73.25608 79.25204 85.80378 92.78393 100.00000
47.00000 46.25952 45.60924 45.12933 44.88369 44.91872 45.26585 45.94601 46.97461 48.36598 50.13680 52.30812 54.90600 57.96052 61.50324 65.56264 70.15754 75.28830 80.92658 87.00551 93.41401 100.00000
47.00000 46.53465 46.14961 45.91768 45.89890 46.13865 46.66869 47.51041 48.67909 50.18818 52.05270 54.29121 56.92617 59.98266 63.48549 67.45480 71.90011 76.81317 82.16059 87.87776 93.86663 100.00000
47.00000 46.72955 46.53699 46.49303 46.65580 47.06855 47.76016 48.74819 50.04352 51.65532 53.59499 55.87823 58.52520 61.55880 65.00159 68.87127 73.17516 77.90388 83.02502 88.47842 94.17481 100.00000
47.00000 46.84659 46.77590 46.86180 47.16294 47.71985 48.55550 49.67899 51.09183 52.79497 54.79406 57.10189 59.73794 62.72611 66.09112 69.85381 74.02566 78.60237 83.55735 88.83619 94.35424 100.00000
47.00000 46.88098 46.85832 47.01548 47.41451 48.09267 49.06327 50.32075 51.85019 53.63901 55.68478 57.99770 60.59891 63.51693 66.78327 70.42750 74.47154 78.92279 83.76600 88.95487 94.40601 100.00000
47.00000 46.81906 46.76103 46.92744 47.38716 48.17330 49.28445 50.69084 52.34952 54.22643 56.30871 58.60557 61.14341 63.95977 67.09785 70.60164 74.51044 78.85144 83.62914 88.81136 94.31499 100.00000
47.00000 46.63427 46.43940 46.54624 47.03356 47.92914 49.21063 50.80893 52.63091 54.60880 56.71838 58.97278 61.40972 64.08117 67.04700 70.37105 74.11737 78.34358 83.08789 88.34656 94.04264 100.00000
47.00000 46.27868 45.81614 45.78469 46.27187 47.29925 48.82025 50.70360 52.75667 54.85976 56.98354 59.15777 61.44182 63.90849 66.63818 69.71841 73.24461 77.31779 82.03243 87.44442 93.50907 100.00000
47.00000 45.66433 44.76188 44.50463 44.97012 46.17590 48.06774 50.42874 52.83266 55.09026 57.19847 59.23315 61.29149 63.47299 65.87902 68.61997 71.82502 75.65066 80.27970 85.88970 92.54925 100.00000
47.00000 44.61681 43.06247 42.50190 42.92822 44.36662 46.84623 50.11113 53.05517 55.47036 57.48713 59.28509 61.01823 62.81317 64.78513 67.05761 69.78498 73.18026 77.54609 83.28550 90.79825 100.00000
47.00000 42.74045 40.36936 39.51234 39.87435 41.51624 44.83954 50.11455 53.80667 56.24905 57.99478 59.40200 60.68333 61.97648 63.39086 65.04048 67.07713 69.73942 73.43897 78.90802 87.35829 100.00000
47.00000 38.97566 36.16220 35.30379 35.54066 36.98452 40.88124 51.70094 55.80802 57.72451 58.84104 59.64492 60.33671 61.01865 61.76147 62.63641 63.74372 65.26138 67.56241 71.54933 79.72690 100.00000
53.50000 30.00000 30.00000 30.00000 30.00000 30.00000 30.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 60.00000 80.00000
Execution time: 0.005612 seconds
============================
```

Figure 4. Screenshot of Thread Count 8

This outputs confirm that the temperature distribution conforms to the specified boundary conditions, and the program functions correctly for different thread counts.

5.4. Recommendations for Further Work

- **Larger Problem Sizes:** Testing with larger grids would provide more insight into the scalability and efficiency of the parallel implementation.
- **Optimization Strategies:** Exploring different scheduling strategies and optimizing memory access patterns could further improve performance.
- **Hardware Considerations:** Running the program on hardware with more cores or different architectures may yield different performance characteristics.

5.5. Conclusion

The OpenMP parallelization of the Jacobi 2D grid problem successfully leveraged multiple threads to reduce execution time while maintaining correct results. The modifications to the code focused on parallelizing the initialization and main computation loops, utilizing OpenMP directives and clauses to manage data dependencies and reductions. The automated testing script facilitated consistent performance measurements across different thread counts, and the results demonstrate both the correctness and scalability of the parallel implementation.

6. Step 3: Performance Analysis of OpenMP Implementation

The objective of this step is to evaluate the performance of the OpenMP implementation of the Jacobi method for solving partial differential equations. The analysis focuses on measuring the execution times and speedup achieved by varying the number of threads, optimization levels, and problem sizes. The experiments were conducted on the University's High-Performance Computing (HPC) system using the SLURM workload manager. This report presents the testing methodology, performance analysis, key findings, and recommendations based on the results obtained.

6.1. Testing Methodology

To comprehensively assess the performance, a systematic testing approach was adopted. A Bash script was developed to automate the compilation and execution of the program across different configurations. The script varied the following parameters:

Optimization Level: The program was compiled with different optimization levels (-O0, -O1, -O2, -O3) to assess the impact of compiler optimizations on performance.

Thread Counts: The number of threads was varied from 1 to 8 to evaluate the scalability of the parallel implementation.

Grid Sizes: The program was executed on grids of varying sizes (10x10, 50x50, 100x100, 150x150) to evaluate the impact of problem size on performance.

Execution Time Measurement: The execution times were recorded for each configuration to analyze the performance metrics.

The OpenMP directives were used to parallelize the computational loops, and the texttttomp_get_wtime() function was utilized to measure execution times accurately. Print statements were minimized to reduce I/O overhead, especially for larger grid sizes. Each test case was executed, and the iteration counts, maximum difference (textttdifmax), and execution times were recorded.

6.2. Performance Analysis

Here are the performance metrics for the different configurations:

Table 7. Performance Metrics for Grid Size 150x150

Optimization Level	Threads	Iterations	Maximum Difference ($difmax$)	Time (s)
-O0	1	20,763	9.999041×10^{-5}	6.834
	2	20,036	9.999421×10^{-5}	3.642
	4	19,412	9.990420×10^{-5}	1.974
	8	19,558	9.998308×10^{-5}	1.173
-O1	1	20,763	9.999041×10^{-5}	1.761
	2	20,034	9.999776×10^{-5}	1.154
	4	19,423	9.999817×10^{-5}	0.635
	8	19,404	9.999061×10^{-5}	0.381
-O2	1	20,763	9.999041×10^{-5}	0.926
	2	20,032	9.999765×10^{-5}	0.668
	4	19,423	9.999867×10^{-5}	0.357
	8	19,387	9.996107×10^{-5}	0.226
-O3	1	20,763	9.999041×10^{-5}	0.869
	2	20,033	9.999972×10^{-5}	0.633
	4	19,424	9.999856×10^{-5}	0.339
	8	19,730	9.999910×10^{-5}	0.221

Table 8. Performance Metrics for Grid Size 200x200

Optimization Level	Threads	Iterations	Maximum Difference ($difmax$)	Time (s)
-O0	1	32,108	9.999248×10^{-5}	18.041
	2	30,807	9.999068×10^{-5}	9.308
	4	29,749	9.998986×10^{-5}	4.915
	8	30,392	9.998702×10^{-5}	2.917
-O1	1	32,108	9.999248×10^{-5}	4.719
	2	30,799	9.999040×10^{-5}	2.571
	4	29,717	9.999629×10^{-5}	1.529
	8	29,989	9.998269×10^{-5}	0.903
-O2	1	32,108	9.999248×10^{-5}	2.560
	2	30,802	9.999834×10^{-5}	1.451
	4	29,705	9.998981×10^{-5}	0.933
	8	31,783	9.999760×10^{-5}	0.539
-O3	1	32,108	9.999248×10^{-5}	2.423
	2	30,811	9.998999×10^{-5}	1.367
	4	30,818	9.992472×10^{-5}	0.829
	8	29,988	9.998336×10^{-5}	0.484

Table 9. Performance Metrics for Grid Size 250x250

Optimization Level	Threads	Iterations	Maximum Difference ($difmax$)	Time (s)
-O0	1	44,398	9.999424×10^{-5}	38.671
	2	42,381	9.999290×10^{-5}	19.026
	4	42,321	9.988293×10^{-5}	10.781
	8	42,684	9.999864×10^{-5}	6.312
-O1	1	44,398	9.999424×10^{-5}	10.183
	2	42,372	9.999710×10^{-5}	5.149
	4	42,295	9.999975×10^{-5}	3.203
	8	44,204	9.999342×10^{-5}	1.934
-O2	1	44,398	9.999424×10^{-5}	5.311
	2	42,383	9.999860×10^{-5}	2.955
	4	42,328	9.999459×10^{-5}	1.809
	8	41,784	9.999386×10^{-5}	0.981
-O3	1	44,398	9.999424×10^{-5}	5.051
	2	42,314	9.999638×10^{-5}	2.718
	4	40,698	9.999491×10^{-5}	1.722
	8	40,087	9.999116×10^{-5}	0.883

6.2.1. Impact of Problem Size

The execution times increased with the grid size due to the quadratic complexity of the Jacobi method. For single-threaded execution at optimization level -O0, the following base execution times were observed:

- 150×150 Grid: Approximately 6.83 seconds.
- 200×200 Grid: Approximately 18.04 seconds.
- 250×250 Grid: Approximately 38.67 seconds.

This increase aligns with the theoretical expectation that doubling the grid dimension results in a fourfold increase in computational work. The larger problem sizes provided more computational workload, which is beneficial for observing the effects of parallelization and optimization.

6.2.2. Impact of Compiler Optimization Levels

Compiler optimizations significantly affected performance. The optimization levels provided the following improvements over the baseline (-O0):

- -O1: Achieved a speedup of approximately 3 to 4 times compared to -O0.
- -O2: Further improved performance, yielding a speedup of approximately 5 to 7 times over -O0.
- -O3: Offered marginal improvements over -O2, typically around 5% to 10% faster.

The diminishing returns at higher optimization levels suggest that most beneficial optimizations occur at the -O2 level. The optimizations included loop unrolling, vectorization, and efficient memory access patterns, which are crucial for numerical computations.

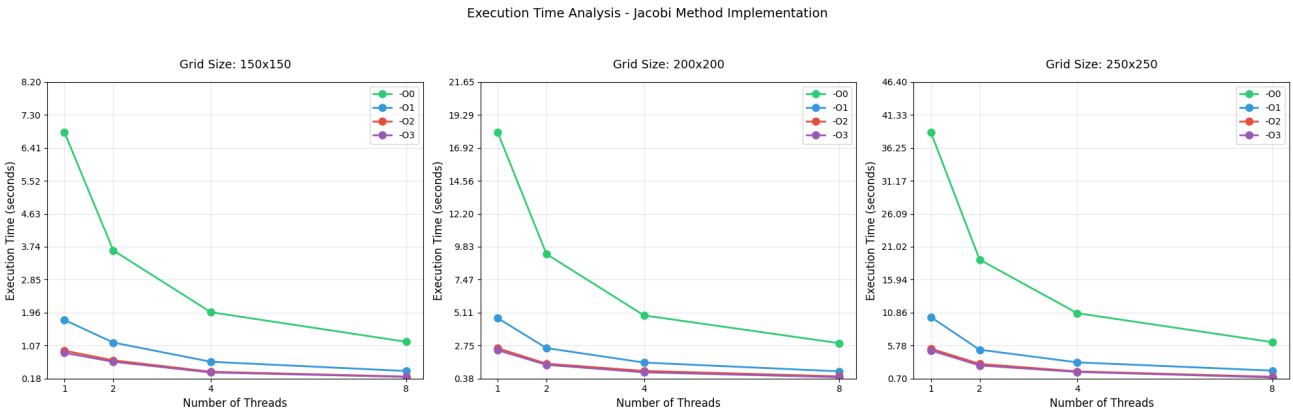

Figure 5. Execution Time Comparison for Different Grid Sizes and Optimization Levels

6.2.3. Parallel Scaling Analysis

The parallel performance was evaluated by measuring the speedup achieved with increasing thread counts. Speedup is defined as the ratio of the execution time with one thread to the execution time with multiple threads.

- **150×150 Grid:**
  - At -O0, an 8-thread execution achieved a speedup of approximately 5.8 times.
  - At -O3, the speedup reduced to approximately 3.9 times with 8 threads.
  - Diminishing returns were observed beyond 4 threads, likely due to insufficient workload per thread.
- **200×200 Grid:**
  - At -O0, an 8-thread execution achieved a speedup of approximately 6.2 times.
  - At -O3, the speedup was approximately 5.0 times with 8 threads.
  - Better scaling was observed compared to the smaller grid due to increased computational work.
- **250×250 Grid:**
  - At -O0, an 8-thread execution achieved a speedup of approximately 6.1 times.
  - At -O3, the speedup increased to approximately 5.7 times with 8 threads.

– The highest parallel efficiency was noted for this grid size, attributed to a higher computation-to-communication ratio.

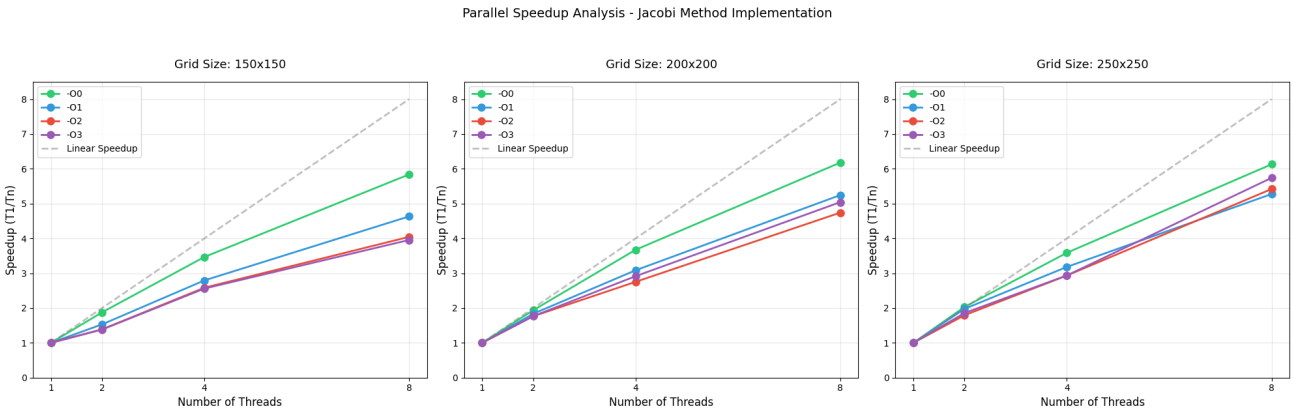

Figure 6. Speedup Comparison for Different Grid Sizes and Optimization Levels

References

Amazon Web Services (n.d.). *AWS Pricing*. Retrieved from https://aws.amazon.com/pricing/.

Armbrust, Michael et al. (2010). "A View of Cloud Computing". In: *Communications of the ACM* 53.4, pp. 50–58. DOI: 10.1145/1721654.1721672.

Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press. ISBN: 978-0-262-03313-7.

Dillon, Tharam, Chen Wu, and Elizabeth Chang (2010). "Cloud Computing: Issues and Challenges". In: *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, pp. 27–33. DOI: 10.1109/AINA.2010.187.

Google Cloud Platform (n.d.). *Google Cloud Platform Pricing*. Retrieved from https://cloud.google.com/pricing/.

Hashem, IAT et al. (2015). "The Rise of "Big Data" on Cloud Computing: Review and Open Research Issues". In: *Information Systems* 47, pp. 98–115. DOI: 10.1016/j.is.2014.07.006.

Leitner, Philipp and Juergen Cito (2016). "Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds". In: *ACM Transactions on Internet Technology* 16.3, 15:1–15:23. DOI: 10.1145/2885497.

Li, Zhenkun, Liam O'Brien, and He Zhang (2019). *Decision Support for Cloud Computing*. Springer. ISBN: 978-3-030-05645-2. DOI: 10.1007/978-3-030-05646-9.

Marinescu, Dan C. (2013). *Cloud Computing: Theory and Practice*. Morgan Kaufmann. ISBN: 978-0-12-404627-6.

Microsoft Azure (n.d.). *Azure Pricing*. Retrieved from https://azure.microsoft.com/en-us/pricing/.

OpenMP Application Program Interface (n.d.). https://www.openmp.org/. Accessed: 2023-03-24.

Rehman, Muhammad HU, Teh Ying Wah, and Farookh Khadeer Hussain (2018). "Towards Multi-Criteria Cloud Service Selection: A Systematic Literature Review". In: *Journal of Network and Computer Applications* 104, pp. 1–13. DOI: 10.1016/j.jnca.2017.12.018.

Stergiou, Charalampos et al. (2018). "Secure Integration of IoT and Cloud Computing". In: *Future Generation Computer Systems* 78, pp. 964–975. DOI: 10.1016/j.future.2016.11.031.

Due to space constraints, only relevant portions of the code have been included in the main report. The full code is provided in the attached files.