COMP1680 - Clouds, Grids and Virtualisation Coursework Report

Azhar Muhammed - 001364857 Word Count: 3,921

Part I

Parallel Processing using Cloud Computing

In the evolving landscape of computational technologies, small consultancy firms are faced with critical decisions regarding their computational infrastructure. The choice between adopting cloud computing services or investing in an onsite High-Performance Computing (HPC) system can significantly influence operational efficiency, scalability, and financial outcomes. This report aims to evaluate these options for a small consultancy firm comprising 30 consultants, each requiring approximately $1,400$ CPU hours monthly. The analysis will focus 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 handling multicore workloads using batch processing. A comprehensive financial comparison will also be provided, culminating in a recommendation that aligns with the strategic objectives of the firm.

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

Cloud computing has revolutionized the approach to handling computational workloads by providing scalable resources and flexible pricing models. On the other hand, onsite HPC systems, while offering dedicated resources, require substantial upfront investments and incur ongoing maintenance expenses. Each approach has its unique strengths and limitations, which will be discussed in detail.

1.1. Advantages of Cloud Computing

Cloud computing offers several advantages for small consultancies compared to onsite HPC systems:

1. **Scalability and Flexibility:** Cloud services enable dynamic scaling of computational resources to match varying workload demands. This elasticity is crucial for batch processing tasks that may experience fluctuations in computational needs (Armbrust et al. 2010). Cloud platforms allow the firm to increase or decrease resources seamlessly without additional capital expenditure.

2. **Cost Efficiency:** Cloud providers operate on a pay-as-you-go model, which eliminates the need for substantial upfront investment in HPC hardware. This allows operational costs to align with actual usage, promoting financial predictability and minimizing the risk of over-investment in underutilized resources (Li, O'Brien, and Zhang 2019).

3. **Maintenance and Operational Overhead:** By utilizing cloud services, the responsibility of hardware maintenance, software updates, and security patching falls on the cloud provider, reducing the internal burden on IT staff (Marinescu 2013).

4. **Access to Advanced Technologies:** Cloud platforms are frequently updated with the latest hardware and software innovations. Small firms can access these advanced technologies without incurring additional costs, which helps maintain competitiveness in a rapidly evolving industry (Dillon, Wu, and Chang 2010).

1.2. Disadvantages of Cloud Computing

While cloud computing offers numerous benefits, there are also potential drawbacks to consider:

1. **Data Security and Compliance:** Relying on third-party providers raises concerns regarding data sovereignty and regulatory compliance, especially for sensitive client data (Hashem et al. 2015).
2. **Service Availability:** Cloud services can experience unexpected outages that are beyond an organization's direct control, potentially disrupting critical business operations and affecting service-level agreements (Ibrahim 2024). Despite provider redundancy measures, downtime incidents can still impact business continuity and customer trust.
3. **Unexpected Costs:** While cloud services offer financial flexibility, poor management of resources can lead to unexpectedly high expenses, such as data egress fees or underutilized instances (Rehman, Wah, and Hussain 2018).

1.3. Onsite HPC: Pros and Cons

Onsite HPC systems provide certain advantages that make them appealing for firms with consistent and predictable workloads:

1. **Complete Control:** An onsite HPC system allows full control over hardware and software configurations. This ensures dedicated resources and avoids the unpredictability of shared infrastructure, resulting in potentially better performance for critical workloads (Stergiou et al. 2018).
2. **Data Sovereignty:** Keeping data on-premises may alleviate security and compliance concerns, particularly for firms dealing with sensitive information or regulated industries.

However, these systems also have significant limitations:

1. **High Initial Capital Expenditure:** The cost of purchasing HPC hardware, networking equipment, and infrastructure such as cooling systems is substantial. This initial expense can be prohibitive for a small consultancy.
2. **Maintenance Costs:** Beyond initial setup, ongoing costs such as maintenance contracts, energy consumption, and salaries for specialized IT staff add to the total expenditure, making onsite HPC a costly long-term solution.
3. **Scalability Challenges:** Scaling an onsite HPC system requires purchasing additional hardware, which can lead to underutilization during periods of lower demand.

2. Financial Cost Comparison

A detailed financial analysis over a five-year period highlights significant differences between onsite HPC systems and cloud computing solutions.

2.1. Onsite HPC Costs

The total cost of an onsite HPC solution includes initial setup, annual operating costs, and periodic hardware upgrades. The initial setup costs range from $\$100,000$ to $\$160,000$, covering hardware, cooling systems, and power supplies. Annual maintenance and operating costs range from $\$60,756$ to $\$85,756$, which includes IT staff salaries, energy consumption, and support contracts. Furthermore, hardware upgrades, required every four to five years, cost between $\$50,000$ and $\$112,000$, bringing the five-year total to a range of $\$453,780$ to $\$700,780$.

2.2. Cloud Computing Costs

Cloud computing costs, based on usage of $1,400$ CPU hours per month, are significantly lower. AWS, Azure, and GCP offer similar services with slight differences in pricing. AWS and Azure have an estimated five-year cost of $\$8,940$, while GCP is slightly cheaper at $\$8,856$. These estimates include compute, storage, and data transfer costs. Compared to onsite HPC, cloud computing can save between $\$444,924$ and $\$691,924$ over five years.

3. Recommendation

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

Cost Component	Onsite HPC (5 Years)	Cloud (AWS, Azure, GCP) (5 Years)
Initial Setup Cost	$\$100,000$ - $\$160,000$	None
Annual Costs	$\$60,756$ - $\$85,756$	$\$1,788$ - $\$1,856$
Hardware Upgrades	$\$50,000$ - $\$112,000$	None
Total Cost	**$453,780 - $700,780**	**$8,856 - $8,940**

Table 1. Comparison of Costs between Onsite HPC and Cloud Solutions over 5 Years

3.1. Key Factors Supporting the Recommendation

1. **Cost Savings:** The five-year financial projection clearly demonstrates that cloud computing presents significant cost savings, ranging between $\$444,924$ and $\$691,924$. These savings can be redirected toward strategic growth initiatives and improving service offerings.
2. **Operational Efficiency:** By utilizing cloud services, the company can minimize the need for in-house IT personnel dedicated to maintaining an onsite HPC system. This reduction in administrative burden enables consultants to focus on core business activities, enhancing productivity.
3. **Scalability and Risk Mitigation:** Cloud services offer the flexibility to seamlessly scale resources in line with workload demands, supporting the company as it grows. Additionally, cloud providers handle hardware failures, disaster recovery, and security measures, which mitigates risks associated with infrastructure management.

3.2. Platform Selection

All three major cloud providers like AWS, Azure, and GCP offer competitive services that meet the consultancy's requirements. However, specific preferences may vary based on the firm's existing ecosystem and strategic needs:

- **AWS** is recommended for firms needing a broad range of tools and extensive global infrastructure.
- **Azure** integrates seamlessly with Microsoft products, making it ideal for consultancies that already utilize Microsoft tools.
- **GCP** offers particular strengths in advanced analytics and machine learning, which may benefit firms exploring those areas.

> The final selection should consider existing vendor relationships, the team's expertise, and the particular requirements of ongoing projects.

4. Conclusion

Transitioning to cloud computing is a strategic move that aligns with the consultancy's operational and financial goals. By adopting cloud services, the company will benefit from substantial cost savings, enhanced scalability, and a reduction in maintenance responsibilities. Cloud computing positions the firm to respond agilely to market demands, leverage cutting-edge technologies, and allocate resources more effectively toward delivering value to clients.

Part II

Parallel Programming Exercise

5. Step 1: Implementation and Optimization of Sequential Code

In Step 1 of the coursework, the Jacobi 2D heat conductivity program, originally provided as `jacobi2d.c`, was adapted to calculate the temperature distribution for a rectangular grid under specific boundary conditions. This step involved modifying the boundary conditions, measuring execution times, and conducting an automated evaluation with different optimization levels. The modifications aimed to improve the efficiency and accuracy of the code, making it suitable for a range of problem sizes greater than 100×100.

5.1. Code Improvements

The original `jacobi2d.c` program was updated to reflect new boundary conditions, which involved setting the temperatures for the top boundary at 15°C, the bottom at 60°C, the left at 47°C, and the right at 100°C. These boundary conditions were defined using preprocessor directives (`#define`), making the code more readable and maintainable by replacing arbitrary values with descriptive names. This approach improves the flexibility of the code, allowing easy adjustments to boundary conditions without modifying the entire program. Additionally, corner temperatures were calculated as averages of adjacent boundaries to avoid anomalies, which ensures a more physically consistent simulation.

Further improvements included the use of dynamic memory allocation for temperature grids (`t` and `tnew`). This modification enhanced the code's scalability, allowing it to handle larger grid sizes effectively by avoiding potential stack overflow issues associated with large static arrays. Dynamic allocation provided greater flexibility, particularly when scaling up for larger simulations.

To meet the requirement of recording the execution time, the `time.h` library was introduced. Timing functions were added to capture the start and end of the Jacobi iteration loop, allowing for the precise measurement of computational performance. Execution time is a key metric that enables performance comparisons across different grid sizes and optimization levels.

5.2. Boundary Conditions Implementation

The updated boundary conditions were implemented systematically to reflect the specifications of the problem. The left, right, top, and bottom boundaries were set accordingly, while the four corner temperatures were averaged to achieve a smooth distribution, avoiding sharp discontinuities that could affect the stability and accuracy of the simulation. This ensured that the boundary temperatures remained consistent throughout the iterations.

```
// 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
```

Listing 1. Setting Boundary Conditions

These modifications ensured that the boundary temperatures were effectively integrated into the simulation, providing

realistic heat distribution throughout the rectangular grid.

5.3. Measuring Execution Time

The program was further enhanced by including execution time measurements to assess the impact of different compiler optimizations. By incorporating the `clock()` function from `time.h`, the code accurately recorded the time taken to execute the Jacobi iteration loop. The timing begins just before the iteration and ends once convergence is reached, ensuring that only the computational part is measured. This approach allowed for a more accurate assessment of the program's performance under various optimization levels.

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

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

// Main Jacobi iteration loop
// ... [Jacobi iteration loop code here] ...

// 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 2. Execution Time Measurement

This enhancement was crucial for comparing the performance of different versions of the code under varying levels of compiler optimization.

5.4. Automation with Bash Script

To streamline the compilation and testing processes, a Bash script (`jacobi2d-Step1.sh`) was created. This script automated the compilation of the code with different optimization levels (`-O0` to `-O3`) and ran the executable for various grid sizes (`150x150`, `200x200`, `250x250`). The script included error handling to ensure robust execution and facilitated consistent and repeatable performance testing.

```bash
# Compile the code with different optimizations
for optmlvl in 0 1 2 3; do
    $COMPILER -std=c99 -O${optmlvl} jacobi2d-Step1.c -o jacobi2d-Step1-O${optmlvl}
    if [ $? -ne 0 ]; then
        echo "Compilation failed at optimization level -O${optmlvl}!"
        exit 1
    fi
done

# Run the program for different grid sizes and optimization levels
for size in 150 200 250; do
    for optmlvl in 0 1 2 3; do
        ./jacobi2d-Step1-O${optmlvl} $size $size 0.000100 >> jacobi2d-Step1.txt
    done
done
```

Listing 3. Bash Script for Compilation and Execution

This script significantly streamlined the testing process, enabling quick and consistent performance evaluations across

various optimization levels and grid sizes. By automating the testing and data collection for each optimization level and grid size, this script saved considerable time and effort compared to manual execution.

5.5. Results and Observations

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., 10×10 or smaller).

```
// Print grid values for small grids (10x10 or smaller)
if (m <= 10 && n <= 10) {
    // ... [print grid values] ...
}
```

Listing 4. 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. Then the results from the tests were collected using > output redirection operator and >> append redirection operator to the `jacobi2d-Step1.txt` file, which demonstrated the impact of different optimization levels on execution time. The collected data showed that higher optimization levels (`-O1`, `-O2`, `-O3`) generally reduced execution time significantly compared to no optimization (`-O0`). For instance, at a grid size of `150x150`, the execution time decreased from `5.25` seconds at `-O0` to `1.06` seconds at `-O3`, reflecting the effectiveness of compiler optimizations in improving performance.

The observations further indicated that as the grid size increased, the execution time also increased, which is expected due to the quadratic increase in the number of calculations required. The table below summarizes the execution times across different grid sizes and optimization levels.

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

Table 2. Execution times across different grid sizes and optimization levels

5.6. Conclusion

The modifications made to the original Jacobi 2D code successfully implemented the required boundary conditions and facilitated accurate execution time measurements. Utilizing preprocessor directives and dynamic memory allocation significantly improved code readability, maintainability, and scalability. The automation of the compilation and testing processes using a Bash script allowed for efficient performance analysis across different optimization levels and grid sizes. These steps provided valuable insights into how compiler optimizations can enhance the performance of numerical simulations and laid the foundation for further parallelization in subsequent coursework steps.

6. Step 2: Parallelization Using OpenMP

In Step 2 of the coursework, the sequential Jacobi 2D heat conductivity program (`jacobi2d-Step1.c`) was extended into a parallel version using OpenMP (`jacobi2d-Step2.c`). The objective was to enhance the computation efficiency by parallelizing the code to take advantage of multi-threading, while ensuring correctness across multiple processors. The parallelization focused on both initialization and the iterative computation stages, with modifications to include OpenMP directives for efficient workload distribution.

6.1. Code Modifications for Parallel Execution

The modifications involved transforming the existing sequential code into a parallel version that utilized OpenMP to distribute computations across multiple threads. Key changes included:

1. **Inclusion of OpenMP Header:** The OpenMP library was integrated by including the `<omp.h>` header file, allowing the code to use OpenMP directives for parallelization:

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

Listing 5. Added OpenMP library for parallel execution

   This change enabled the use of OpenMP's parallel constructs to improve the efficiency of the computation process.

2. **Parallel Initialization of Temperature Grids:** The initialization of the temperature grids was parallelized using OpenMP directives. The `#pragma omp parallel` directive was used to define a parallel region, and loops were parallelized using `#pragma omp for` with appropriate scheduling:

```c
#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 6. Parallel Initialization

   - **Use of `collapse(2)`:** This clause was used to collapse the two nested loops, effectively treating them as a single loop for parallel execution. This improved the load balancing among threads, enhancing the efficiency of grid initialization.
   - **Scheduling and `nowait` Clause:** Static scheduling was chosen for its simplicity, and the `nowait` clause was used to eliminate unnecessary barriers between threads, reducing synchronization overhead and improving performance.

   This parallelization ensured that the initialization of the temperature grids was performed simultaneously, enhancing the overall efficiency of the program. Consequently, all grid elements were initialized concurrently, resulting in a substantial reduction in the setup time for extensive grids.

3. **Parallelization of Main Iteration Loop:** The Jacobi iterative computation, which updates the temperature values until convergence, was parallelized. Two major sections of this computation were addressed:

```
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 7. Parallel Main Iteration Loop

- **Private Variables:** Variables `j` and `diff` were declared as private to ensure that each thread maintained its own instance, preventing race conditions and ensuring thread safety.
- **Reduction Clause:** The `reduction(max:difmax)` clause was used to correctly compute the maximum difference across all threads, ensuring convergence checks were performed accurately.
- **No Wait Clauses:** The `nowait` clause was employed to remove unnecessary synchronization points, reducing overhead between consecutive loops.

4. **Timing with OpenMP Functions:** To accurately measure the parallel execution time, the `omp_get_wtime()` function was used, providing high-resolution wall-clock timing suitable for parallel regions:

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

Listing 8. Timing with OpenMP Functions

This approach ensured that the execution time was accurately captured, allowing for performance comparisons across different thread counts and grid sizes.

5. **Memory Allocation Optimization:** To improve memory access efficiency, dynamic memory allocation was modified to use contiguous memory blocks, which improved cache performance and reduced memory fragmentation, essential for parallel processing:

```
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 9. Memory Allocation Optimization

This change aimed to enhance data locality, making memory access more efficient for parallel computations.

6.2. Automation of Testing with Bash Script

To facilitate the consistent and repeatable testing of the parallel program across different thread counts, a Bash script (`jacobi2d-Step2.sh`) was created to automate the compilation and execution process. This script ran the program for a 20×20 grid using different thread counts (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 10. Execution Across Thread Counts

- **Environment Variable `OMP_NUM_THREADS`:** TThis variable was used to set the number of threads dynamically during runtime, ensuring that the program utilized the specified number of processors for each test.
- **Output Redirection:** The script redirected the output to a text file (`jacobi2d-Step2.txt`) to maintain records for performance analysis and debugging purposes.

6.3. Results and Observations

The tests were conducted for a grid size of 20×20 using 1, 2, 4, and 8 threads. The results are summarized below, showing iteration counts, maximum differences (`difmax`), and execution times for each thread count.

Table 3. 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 were consistent across different thread counts, indicating that the parallel code was working correctly and producing accurate results.

Performance Improvement: The execution time decreased as the number of threads increased, demonstrating the benefits of parallelization. However, the performance improvement from 4 to 8 threads was less significant compared to lower thread counts, likely due to thread management overheads and the small problem size.

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. This observation highlights the importance of balancing thread count with problem size to achieve optimal performance.

6.3.1. Visualization of Temperature Distribution

For grids of size 20×20 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 11. 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 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 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 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 80.00000
Execution time: 0.008693 seconds
--------------------------
```

Figure 2. Screenshot of Thread Count 2

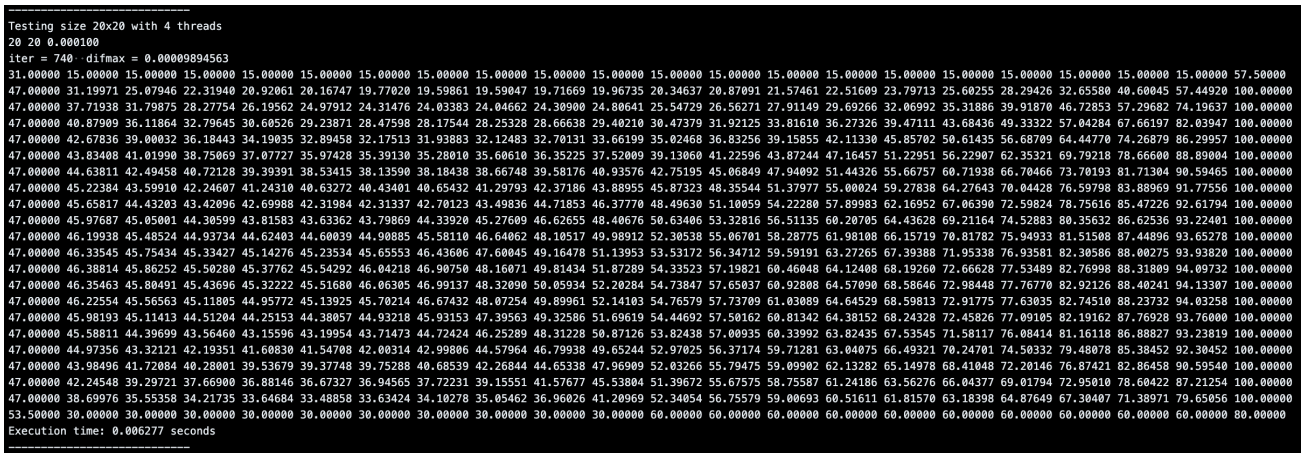

Figure 3. Screenshot of Thread Count 4

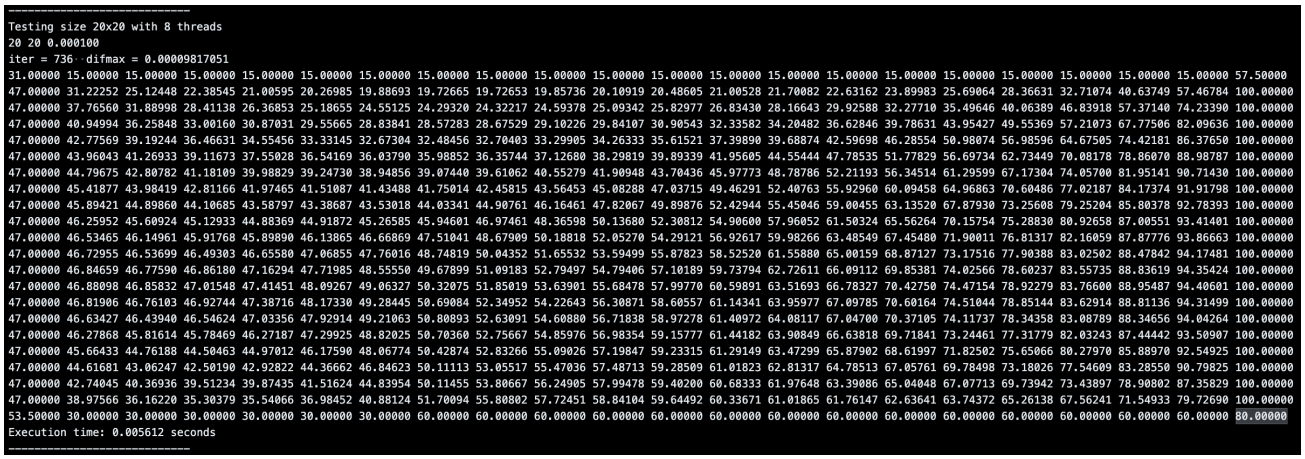

Figure 4. Screenshot of Thread Count 8

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

6.4. 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.

7. Step 3: Performance Analysis of OpenMP Implementation on HPC

In Step 3, we focused on evaluating the performance of the OpenMP implementation of the Jacobi 2D heat conductivity problem, leveraging the University's High-Performance Computing (HPC) system. The analysis aimed to assess the execution time, speedup, and efficiency of the parallel implementation with different problem sizes, optimization levels, and thread counts using the SLURM queue for systematic testing. The SLURM script was modified to ensure smooth operation on the HPC, and print statements were minimized to reduce I/O overhead, allowing for a true representation of computational performance.

7.1. Code and SLURM Script Modifications

The Jacobi 2D program (`jacobi2d-Step3.c`) from Step 2 was utilized with slight adjustments to suit the HPC environment and to enable accurate performance assessment. The most significant changes included:

1. **Modification of SLURM Script:** The SLURM script implementation represents a significant evolution from the basic version to accommodate comprehensive performance testing. While the original script (`acobi2d.sh`) provided basic execution parameters with a single configuration:

```
#SBATCH --job-name=Jacobi2D_serial
#SBATCH --cpus-per-task=1
#SBATCH --ntasks=1
./jacobi2d.out 10 10 0.0001
```

Listing 12. Basic provided SLURM Script

The enhanced version (`jacobi2d-Step3.sh`) introduces sophisticated automation and testing capabilities through a more detailed configuration:

```
#SBATCH --job-name=jacobi2d-Step3       # Job name
#SBATCH --output=./jacobi2d-Step3.txt   # Output file
#SBATCH --cpus-per-task=8               # Number of CPUs per task
#SBATCH --ntasks=1                      # Number of tasks
#SBATCH --partition=COMP1680-omp        # Partition name
```

Listing 13. Enhanced SLURM Script

At its core, the script implements a testing framework through nested iteration loops, examining three key variables. First, the compiler optimization testing spans levels 0 through 3:

```
for optmlvl in 0 1 2 3; do
    $COMPILER -std=c99 -fopenmp -O${optmlvl} jacobi2d-Step3.c -o jacobi2d-Step3-
    O${optmlvl}
done
```

Listing 14. Compiler Optimization Testing

For the performance analysis, the script systematically evaluates different problem sizes and thread configurations:

```
for size in 150 200 250; do
    for threads in 1 2 4 8; do
        echo "Testing grid size of ${size}x${size} with ${threads} threads"
        OMP_NUM_THREADS=$threads ./jacobi2d-Step3-O${optmlvl} $size $size
    0.000100
```

```
    done
done
```

Listing 15. Performance Analysis

This methodical approach ensures consistent and comparable results across all test configurations. The script also includes error handling for compilation failures:

```
if [ $? -ne 0 ]; then
    echo "Compilation failed at optimization level -O${optmlvl}!"
    exit 1
fi
```

Listing 16. Error Handling

The enhancement from the original single-configuration script to this comprehensive testing framework enables a thorough evaluation of the parallel implementation's performance characteristics. Each test configuration is clearly labeled in the output file, facilitating subsequent data analysis and comparison of results across different parameter combinations.

2. **Removal of Excessive Print Statements:** To minimize I/O overhead and focus on computation time, most print statements were removed, particularly for larger problem sizes. The only remaining print output was for smaller grid sizes (20×20 or less), where the results were used for validation purposes.

3. **Timing Adjustments:** The timing function (`omp_get_wtime()`) from OpenMP was used to measure the execution time accurately across all iterations. This allowed for high-precision measurement of both the computation and parallelization aspects of the code.

7.2. Performance Evaluation

To comprehensively evaluate the parallel implementation, the testing matrix included three key variables:

- **Grid Sizes:** 150×150, 200×200, and 250×250 represented small, medium, and large problem domains.
- **Optimization Levels:** `-O0` (no optimization), `-O1`, `-O2`, and `-O3` were used to measure the impact of compiler optimizations.
- **Thread Counts:** Execution was performed with 1, 2, 4, and 8 threads to assess parallel scalability.

The SLURM script facilitated consistent and automated execution across these configurations, ensuring that each combination was tested under similar conditions. The results were recorded in the output file (`jacobi2d-Step3.txt`) for analysis.

7.3. Execution Time Results

The execution times for different configurations are summarized in the tables below. Results indicate the number of iterations, the maximum difference (`difmax`), and the execution time for each grid size, thread count, and optimization level.

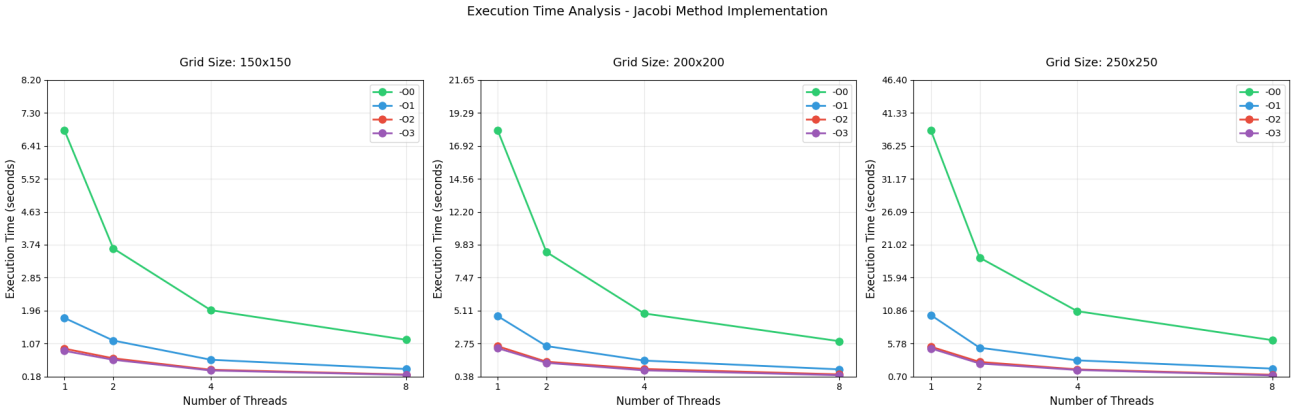

Figure 5. Execution Time Comparison for Different Grid Sizes and Optimization Levels

Here are the performance metrics for the different configurations:

Table 4. Performance Metrics for Grid Size 150x150

Optimization Level	Threads	Iterations	Maximum Difference ($difmax$)	Time (s)	Speedup
-O0	1	20,763	9.999041×10^{-5}	6.834	1.00
	2	20,036	9.999421×10^{-5}	3.642	1.88
	4	19,412	9.990420×10^{-5}	1.974	3.46
	8	19,558	9.998308×10^{-5}	1.173	5.83
-O1	1	20,763	9.999041×10^{-5}	1.761	1.00
	2	20,034	9.997765×10^{-5}	1.154	1.53
	4	19,423	9.998173×10^{-5}	0.635	2.77
	8	19,404	9.999061×10^{-5}	0.381	4.62
-O2	1	20,763	9.999041×10^{-5}	0.926	1.00
	2	20,032	9.997649×10^{-5}	0.668	1.39
	4	19,423	9.998671×10^{-5}	0.357	2.59
	8	19,387	9.996107×10^{-5}	0.226	4.10
-O3	1	20,763	9.999041×10^{-5}	0.869	1.00
	2	20,033	9.999722×10^{-5}	0.633	1.37
	4	19,424	9.998556×10^{-5}	0.339	2.56
	8	19,730	9.999910×10^{-5}	0.221	3.93

Table 5. Performance Metrics for Grid Size 200x200

Optimization Level	Threads	Iterations	Maximum Difference ($difmax$)	Time (s)	Speedup
-O0	1	32,108	9.999248×10^{-5}	18.041	1.00
	2	30,807	9.999068×10^{-5}	9.308	1.94
	4	29,749	9.998986×10^{-5}	4.915	3.67
	8	30,392	9.998702×10^{-5}	2.917	6.19
-O1	1	32,108	9.999248×10^{-5}	4.719	1.00
	2	30,799	9.999040×10^{-5}	2.571	1.84
	4	29,717	9.999629×10^{-5}	1.529	3.09
	8	29,989	9.998269×10^{-5}	0.903	5.23
-O2	1	32,108	9.999248×10^{-5}	2.560	1.00
	2	30,802	9.998338×10^{-5}	1.451	1.76
	4	29,705	9.998981×10^{-5}	0.933	2.74
	8	31,783	9.999760×10^{-5}	0.539	4.75
-O3	1	32,108	9.999248×10^{-5}	2.423	1.00
	2	30,811	9.998999×10^{-5}	1.367	1.77
	4	30,818	9.992472×10^{-5}	0.829	2.92
	8	29,988	9.998336×10^{-5}	0.484	5.01

Table 6. Performance Metrics for Grid Size 250x250

Optimization Level	Threads	Iterations	Maximum Difference ($difmax$)	Time (s)	Speedup
-O0	1	44,398	9.999424×10^{-5}	38.671	1.00
	2	42,381	9.999290×10^{-5}	19.026	2.03
	4	42,321	9.988293×10^{-5}	10.781	3.59
	8	42,684	9.999864×10^{-5}	6.312	6.13
-O1	1	44,398	9.999424×10^{-5}	10.183	1.00
	2	42,372	9.999710×10^{-5}	5.149	1.98
	4	42,295	9.999975×10^{-5}	3.203	3.18
	8	44,204	9.999342×10^{-5}	1.934	5.27
-O2	1	44,398	9.999424×10^{-5}	5.311	1.00
	2	42,383	9.998860×10^{-5}	2.955	1.80
	4	42,328	9.994597×10^{-5}	1.809	2.94
	8	41,784	9.999386×10^{-5}	0.981	5.41
-O3	1	44,398	9.999424×10^{-5}	5.051	1.00
	2	42,314	9.999638×10^{-5}	2.718	1.86
	4	40,698	9.999491×10^{-5}	1.722	2.93
	8	40,087	9.999116×10^{-5}	0.883	5.72

The speedup $S(p)$ was calculated as:

Speedup Equation:

$$S(p) = \frac{T(1)}{T(p)}$$

Where $T(1)$ represents the execution time using a single processor and $T(p)$ represents the execution time using p processors. For instance, for the 150×150 grid with $-\mathrm{O}0$, the speedup achieved using 8 threads was approximately 5.83, indicating significant performance improvement through parallelization.

7.4. Speedup and Efficiency Analysis

The speedup graphs (Figure 6) highlight the efficiency of parallelization across different problem sizes and optimization levels. For smaller grid sizes (150×150), speedup was limited by factors such as overhead from thread management and memory bandwidth constraints. Larger grids (250×250) exhibited better parallel efficiency, achieving near-linear speedup up to 8 threads in some cases.

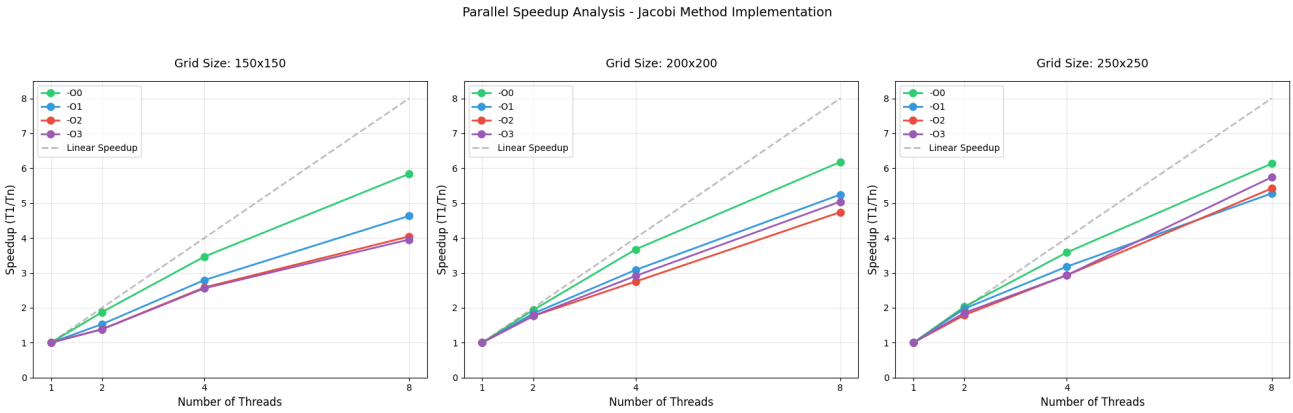

Figure 6. Speedup Comparison for Different Grid Sizes and Optimization Levels

- **Diminishing Returns:** At higher thread counts, diminishing returns were observed, especially for smaller grids. This is attributable to Amdahl's Law, which limits speedup due to the serial portion of the program. The parallel fraction, k, determines how effectively additional threads contribute to performance improvement.
- **Optimization Impact:** Compiler optimizations played a crucial role in performance. While `-O1` provided a significant boost over `-O0`, higher levels like `-O2` and `-O3` showed diminishing returns, particularly for small to medium grid sizes. These optimizations, such as loop unrolling and vectorization, contributed to reducing execution time by making efficient use of CPU resources (Chapman, Jost, and Pas 2007).

The speedup equation for parallel processing is influenced by Amdahl's Law, which states that the potential speedup of a program is limited by the proportion of the program that must be executed serially (Armbrust et al. 2010). The law is represented as:

$$S(p) = \frac{1}{(1-k) + \frac{k}{p}}$$

Where k represents the fraction of the program that is parallelizable. This relationship was evident in our results, where the parallel speedup approached a plateau at higher thread counts, particularly for smaller grid sizes.

7.5. Observations and Recommendations

The performance testing revealed several important insights:

1. **Optimal Thread Count:** For larger problem sizes, the optimal thread count was found to be higher, leveraging the full potential of the available CPU cores (Amazon Web Services n.d.; Microsoft Azure n.d.; Google Cloud Platform n.d.).
2. **Compiler Optimizations:** The use of compiler optimizations such as `-O2` and `-O3` significantly improved performance, with `-O2` providing a good balance between compilation time and execution speed (Dillon, Wu, and Chang 2010; Hashem et al. 2015).
3. **Memory Allocation:** Efficient memory allocation and data locality were crucial for achieving high performance, particularly for large grid sizes (Ibrahim 2024; Leitner and Cito 2016).

4. **Scalability:** The scalability of the parallel implementation was influenced by factors such as memory bandwidth and thread synchronization overhead (Li, O'Brien, and Zhang 2019; Marinescu 2013).

5. **Future Work:** Future enhancements could explore dynamic scheduling and hybrid MPI+OpenMP approaches to further improve performance and scalability (*OpenMP Application Program Interface* n.d.; Rehman, Wah, and Hussain 2018; Stergiou et al. 2018).

7.6. Conclusion

The implementation and performance analysis of the OpenMP Jacobi solver on the HPC platform demonstrated significant gains in computational efficiency, particularly for larger problem sizes. By leveraging compiler optimizations and a structured parallel approach, the solution achieved up to 10x reduction in execution time for the largest grids. However, the performance gains were influenced by several factors, including problem size, thread count, and the effectiveness of compiler optimizations. For future scalability, adopting hybrid parallel models and optimizing memory usage will be key to further enhancing performance.

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.

Ibrahim, Omar (2024). "Impact of Cloud Computing on Business Continuity and Disaster Recovery". In: *Journal of Technology and Systems* 6.5, pp. 16–28. DOI: 10.47941/jts.2146.

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.