



SE3082

Parallel Computing

3rd Year, 1st Semester

Assignment 3

**Parallel Implementation of 2D Jacobi Heat Diffusion
using OpenMP, MPI, and CUDA**

IT23226814 – Vakesh Ranganathan

Submitted to

Sri Lanka Institute of Information Technology

In partial fulfilment of the requirements for the

Bachelor of Computer Science degree

5th December 2025

Declaration

I hereby declare that this report is my own work and has not been submitted previously, in whole or in part, for any other academic requirement. All sources of information used in this report have been acknowledged through proper citation and referencing, in accordance with the required academic standards.

I confirm that this work complies with the guidelines on plagiarism and academic integrity set forth by the Sri Lanka Institute of Information Technology (SLIIT).

Name: Vakesh Ranganathan

Registration Number: IT23226814

Date: 5th December 2025

Table of Contents

Declaration.....	ii
1. Parallelization Strategies.....	1
1.1 OpenMP Implementation (Shared Memory)	1
1.2 MPI Implementation (Distributed Memory).....	1
2. Runtime Configurations.....	3
2.1 Hardware Specifications	3
2.2 Software Environment	3
2.3 Configuration Parameters	3
3. Performance Analysis	4
3.1 OpenMP Analysis	4
3.2 MPI Analysis.....	5
3.3 CUDA Analysis.....	6
3.4 Comparative Analysis	7
4. Critical Reflection.....	8
4.1 Challenges & Limitations	8
4.2 Lessons Learned.....	8
References.....	9

1. Parallelization Strategies

The selected algorithm is the **2D Jacobi Iteration** for solving steady state heat diffusion, which falls under the "Numerical Computation" domain. This algorithm updates a grid point (x,y) based on the average of its four neighbours (North, South, East, West).

1.1 OpenMP Implementation (Shared Memory)

The OpenMP implementation focuses on decomposing the nested loops of the stencil operation across available CPU threads.

- **Loop Collapse**

The directive `#pragma omp parallel for collapse(2)` was utilized. This flattens the nested y and x loops into a single iteration space, significantly improving load balancing by increasing the granularity of work chunks available to the 16 threads.

- **Race Condition Handling**

A `reduction(max:maxdiff)` clause was employed to safely compute the maximum error (residual) across all threads without manual locking mechanisms.

- **First Touch Policy**

To optimize for Non-Uniform Memory Access (NUMA) on the Ryzen architecture, the memory initialization loop was also parallelized. This ensures that memory pages are physically allocated on the RAM banks closest to the cores that will process them, reducing memory latency during the main computation loop.

1.2 MPI Implementation (Distributed Memory)

The MPI implementation uses Domain Decomposition to split the problem across multiple distributed processes.

- **Row wise Decomposition**

The grid is sliced horizontally. If there are N rows and P processes, each process is responsible for calculating a strip of N/P rows.

- **Ghost Cells (Halo Regions)**

Each process allocates two extra rows (top and bottom) to store boundary data received from neighbours. These "ghost rows" allow the stencil operation to proceed locally without checking remote memory for every pixel.

- **Communication**

At the start of every iteration, **MPI_Sendrecv** is used to exchange boundary rows with the ranks immediately above and below.

- **Convergence Check**

MPI_Allreduce is used to aggregate the local maximum error from every rank into a global maximum error to determine if the simulation should terminate.

1.3 CUDA Implementation (GPU Acceleration)

The CUDA implementation maps the fine-grained parallelism of the stencil operation to the massive thread count of a GPU.

- **Thread Mapping**

The strategy maps **one thread to one grid cell**. The global thread index is calculated as $x = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$ and y similarly.

- **Memory Coalescing**

The kernel indexing $\text{idx} = y * N_x + x$ ensures that threads within a "warp" (32 threads) access consecutive memory addresses in the Global Memory. This "coalesced" access pattern maximizes memory bandwidth, which is the primary bottleneck for stencil codes.

- **Host Device Transfer**

Data transfer overhead is minimized by copying the grid to the GPU (`cudaMemcpyHostToDevice`) once at the start and retrieving it only after convergence.

2. Runtime Configurations

2.1 Hardware Specifications

- **Host Processor** - AMD Ryzen 7 5800H (8 Physical Cores, 16 Logical Threads).
- **Virtualization** - VMware Workstation running Ubuntu Linux.
- **GPU Environment** - Google Colab with NVIDIA T4 Tensor Core GPU (16GB VRAM).

2.2 Software Environment

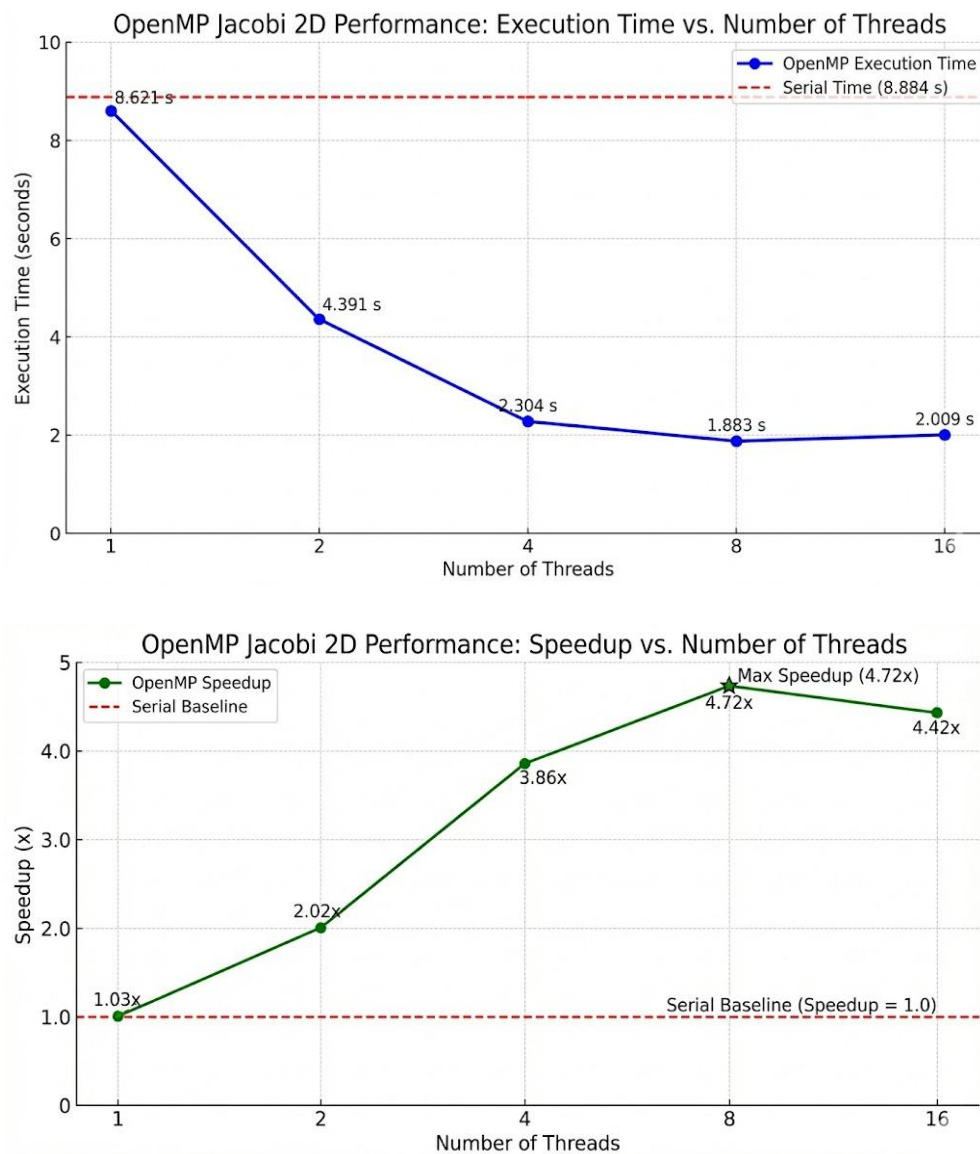
- **OS** - Ubuntu Linux (Virtual Machine).
- **Compiler** - GCC 9.4.0 (OpenMP 4.5 support).
- **MPI Library** - MPICC
- **CUDA Toolkit** - NVCC (NVIDIA CUDA Compiler) on Google Colab.

2.3 Configuration Parameters

- **OpenMP** - Tested with thread counts {1, 2, 4, 8, 16}. Schedule set to static.
- **MPI** - Tested with process counts {1, 2, 4, 8, 16}.
 - **Cluster Configuration**: Configured as a simulated cluster with 8 cores allocated for the Master node and 8 cores for the Worker node.
- **CUDA** - Tested with Grid Size 2048*2048. Block configurations tested – 8*8, 16*16, and 32*32.

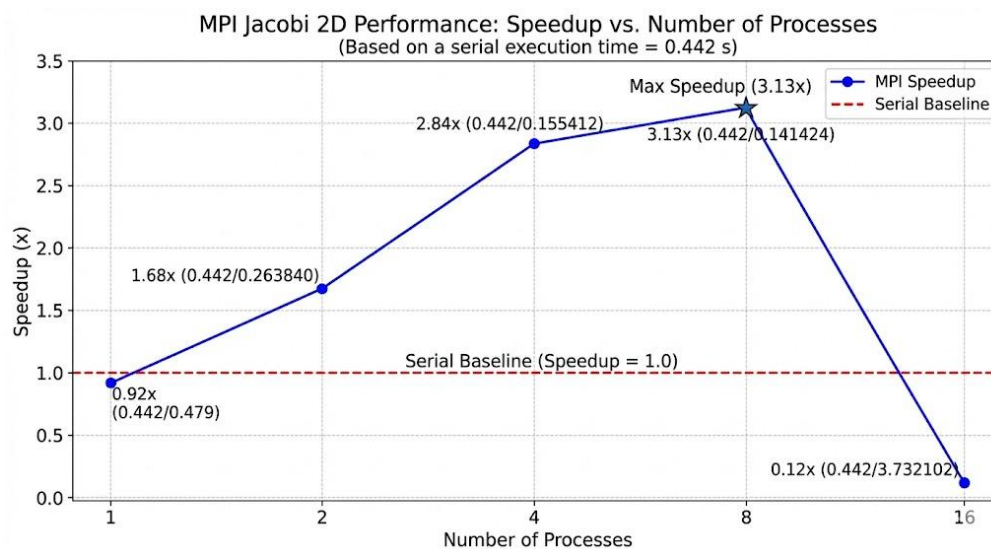
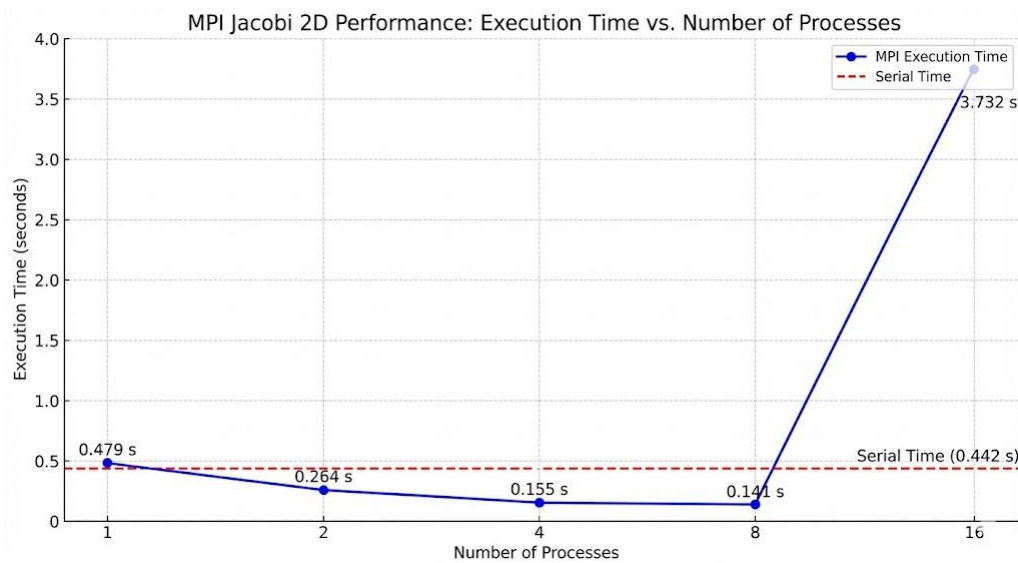
3. Performance Analysis

3.1 OpenMP Analysis



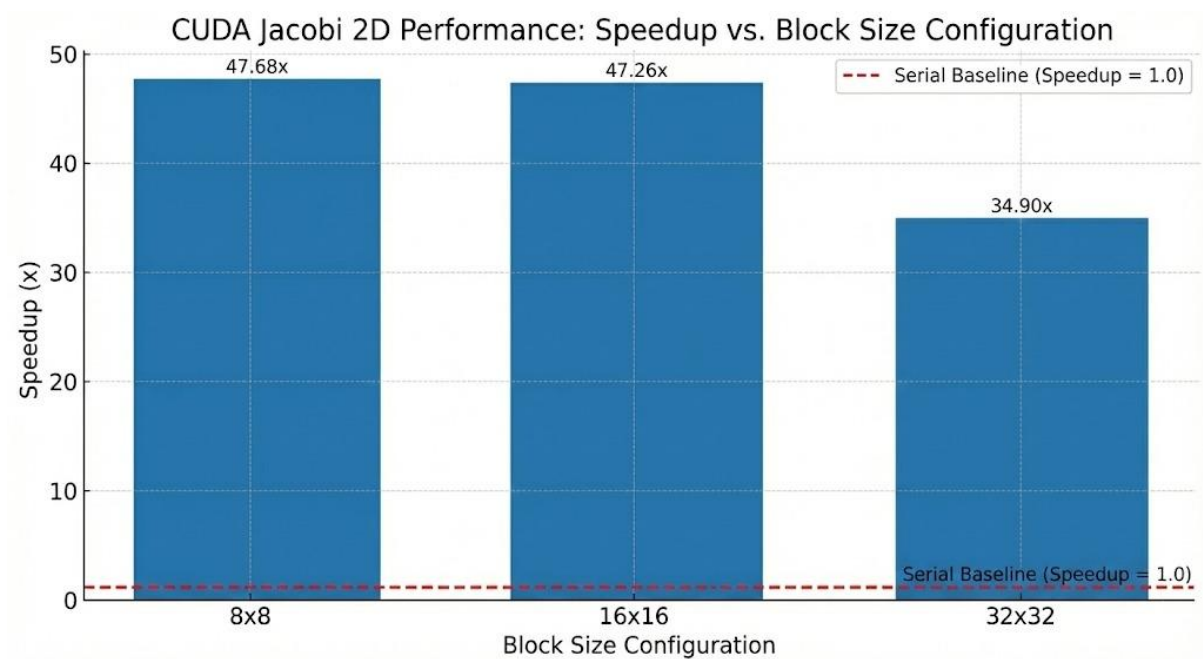
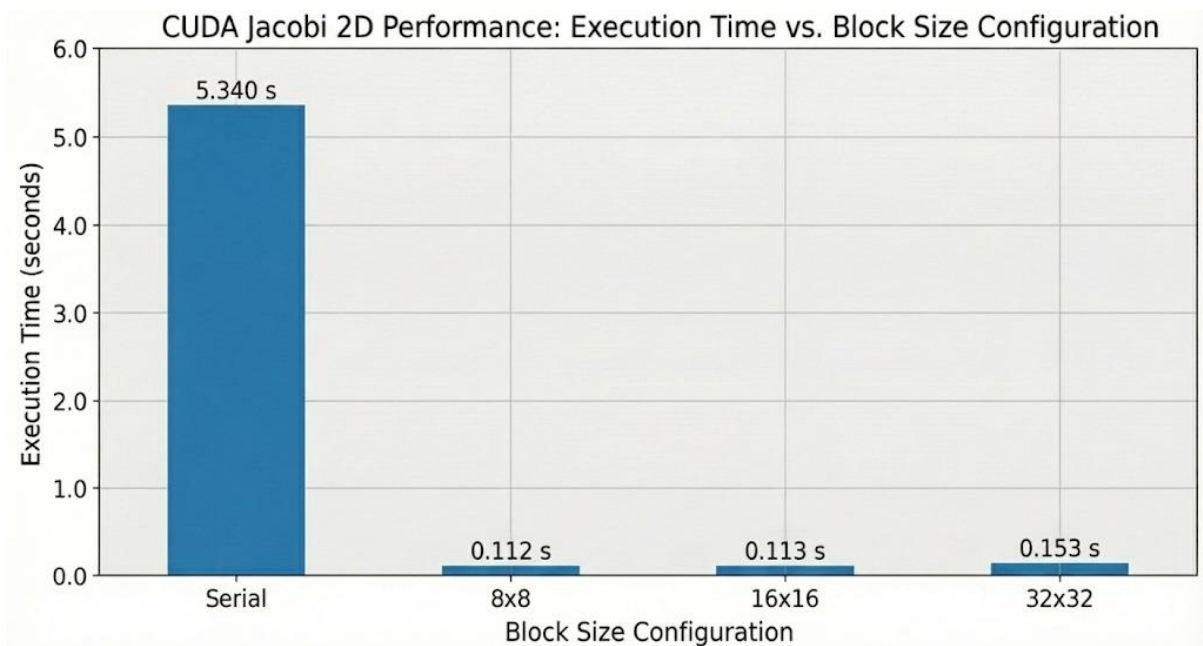
- The OpenMP implementation demonstrates near-linear speedup up to **8 threads** (Speedup 4.72x). However, at 16 threads, the performance plateaus and slightly degrades (4.42x). This is a direct result of the hardware architecture; the Ryzen 7 5800H has 8 physical cores. While it supports 16 threads via SMT (Simultaneous Multithreading), floating-point intensive tasks like the Jacobi iteration often saturate the physical FPU resources, yielding diminishing returns for logical threads.

3.2 MPI Analysis



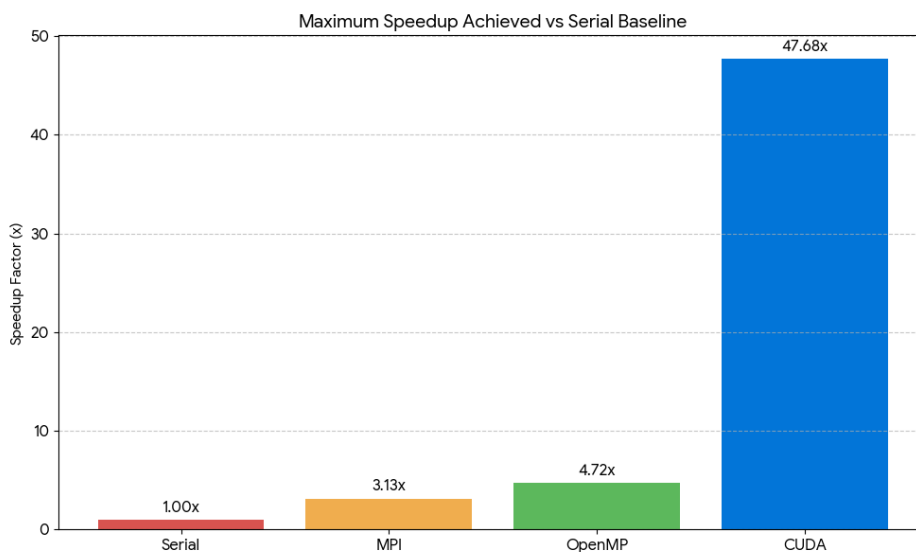
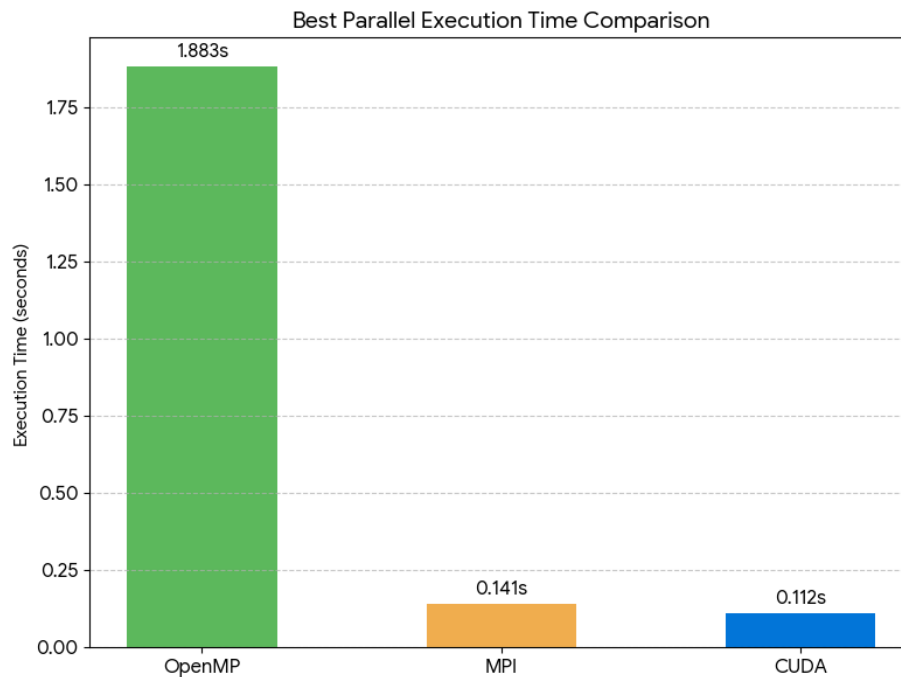
- The MPI implementation scales well up to **8 processes** (3.13x speedup). However, a significant performance cliff is observed at **16 processes**, where the speedup drops to 0.12x (slower than serial).
- **Bottleneck** - This degradation is caused by the **computation-to-communication ratio**. As the number of processes increases, the number of rows assigned to each process decreases. At 16 processes, the time spent transmitting halo rows over the network (communication) exceeds the time taken to compute the stencil for the small chunk of data (computation). This highlights the scalability limit for small grid sizes.

3.3 CUDA Analysis



- The CUDA implementation achieves massive acceleration, with a peak speedup of **47.68x** compared to the serial baseline.
- **Block Size Impact** - Configurations of 8*8 and 16*16 yielded the best performance (~0.11s). The 32*32\$ block size (1024 threads) performed slightly worse (34.9x speedup). This is likely because 1024 threads is the maximum limit per block, which can reduce occupancy due to register pressure, leaving no room for the scheduler to hide memory latency.

3.4 Comparative Analysis



- **Best Performer** - CUDA is the clear winner for this specific problem size, completing the task in milliseconds where CPU methods took seconds. The massive parallelism of the GPU fits the independent nature of the stencil update perfectly.
- **CPU Methods** - OpenMP provided the most efficient CPU-based solution for a single node. MPI introduced overhead that was not justifiable for the problem size on a single machine but would be necessary for grid sizes too large to fit in a single node's RAM.

4. Critical Reflection

4.1 Challenges & Limitations

- **MPI Scalability on Small Data**

A major challenge encountered was the performance degradation in MPI at 16 cores. With a fixed grid size, increasing processes diluted the workload per core to the point where the overhead of `MPI_Sendrecv` dominated the execution time. This taught the importance of "Weak Scaling" (increasing problem size with processors) vs "Strong Scaling" (fixed problem size).

- **Resource Availability**

Configuring a true 16-core cluster was challenging. AWS instances were unavailable due to region/cost limits, necessitating the use of a local VMware Ubuntu instance. This simulation (cores on a single chip) introduces memory bandwidth contention that might not exist in a true distributed physical cluster.

- **Hardware Restrictions**

Running high-performance CUDA code requires specific GPU hardware. I utilized Google Colab's T4 instances as a workaround, which introduced latency in session management but provided the necessary compute power.

4.2 Lessons Learned

- **Hybrid Approaches**

I learned that strictly distinct paradigms have limits. A hybrid approach (MPI for internode communication + OpenMP for intranode computation) would likely solve the MPI scaling issue observed at 16 cores.

- **Memory Bound Nature**

The Jacobi algorithm is memory-bandwidth bound, not compute-bound. The superior performance of CUDA was largely due to the high-bandwidth memory (HBM/GDDR6) on the GPU compared to system RAM.

References

[Jacobi method — overview & convergence \(Wikipedia\). Wikipedia](#)

<https://membres-ljk.imag.fr/Christophe.Picard/teaching/gp-gpu/References/zhang-IJCM-2011.pdf>

[CS267 \(Berkeley\): Solving the Discrete Poisson Equation using Jacobi, SOR, CG, FFT. People @ EECS](#)

LLM Usage

<https://chatgpt.com/share/6931a83d-ac10-8001-893c-c9afe7b73983>

<https://aistudio.google.com/app/prompts?state=%7B%22ids%22:%5B%221Fgc5XHSV31WFHpGY9CssjYQnljXSrUNf%22%5D,%22action%22:%22open%22,%22userId%22:%22113350596718129030996%22,%22resourceKeys%22:%7B%7D%7D&usp=sharing>