# 1 Results and Discussion

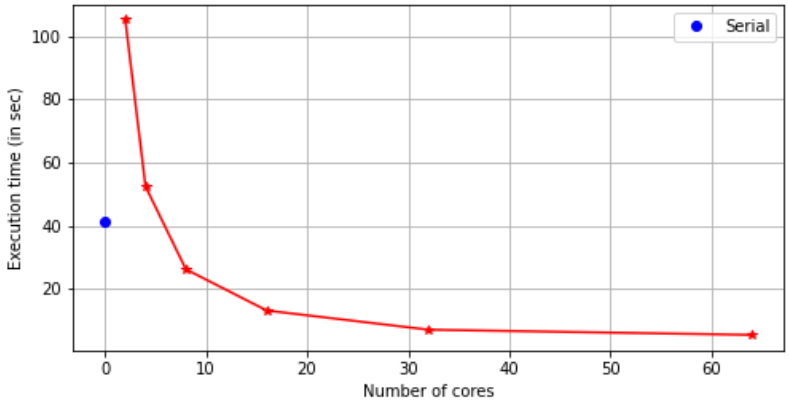


Figure 1: Execution time vs Number of MPI Ranks

| Number of Cores | Cell update per sec |
|---|---|
| 1 (serial) | 2.07e+08 |
| 2 | 8.17e+07 |
| 4 | 1.64e+08 |
| 8 | 3.27e+08 |
| 16 | 6.50e+08 |
| 32 | 1.20e+09 |
| 64 | 1.56e+09 |

Table 1: Cellupdate Rate

Figure 1 shows the plot of execution time for each MPI rank configuration. From the given figure, it can be seen that the execution time decreases with the increase in number of cores. However, the performance of serial code is better compared to the parallel execution with 2 and 4 MPI ranks. Table 1 shows the cell update rate for each core configuration. The fastest cell updates rate is shown by the blue color in the above table, which comes to be around $1.5 \times 10^9$. The performance studies can be better understood in terms of relative speedup. The speedup relative to a single CPU core can be calculated as,

**Speedup = Parallel cell update rate/ Serial cell update rate**

Table 2 and figure 2 show the speedup for different core configurations relative to using a single-core CPU. From the above table and figure, we can see that the maximum speedup is gained when we use 64 MPI ranks and the value

| Number of Cores | Speedup |
|---|---|
| 2 | 0.39 |
| 4 | 0.79 |
| 8 | 1.58 |
| 16 | 3.14 |
| 32 | 5.79 |
| 64 | 7.51 |

Table 2: Speedup of various MPI Ranks

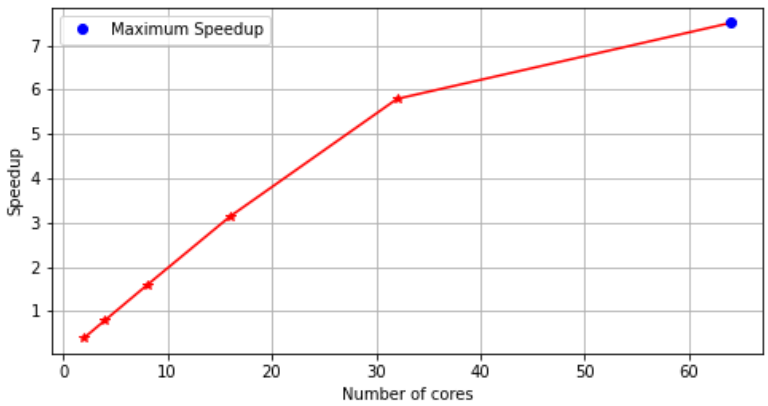


Figure 2: Speedup vs Number of MPI Ranks

of maximum speedup is turned out to be around 7. This shows that MPI parallel code (with 64 MPI ranks)is **7X** faster than serial code.

The following observations can be made from the above plots.

1. The MPI Highlife code scales efficiently with an increasing number of cores. In this case, the scalability of parallel algorithms is improved as the number of cores increases, leading to better performance on larger core counts.

2. The main reason behind the maximum speedup with 64 cores could be the reduced communication overhead between processes compared to smaller core counts. With a larger number of cores, the communication overhead can be mitigated due to increased parallelism and reduced contention on locks, leading to more efficient program execution.

3. The workload might be well-balanced across all cores on the 64-core configuration. Load balancing ensures that each core is effectively utilized which prevents any idle time or resource underutilization. Uneven load distribution can lead to performance degradation.

4. The system's specific architecture can influence performance, with some systems optimized for larger core counts, leading to improved performance as more cores are utilized. Factors such as memory bandwidth, cache coherence, and interconnect topology play a role in this optimization.

5. The parallelization strategy and algorithm design could be well-suited for the 64-core configuration. Some parallel algorithms exhibit optimal performance under specific core counts due to the nature of their parallelization approach or the problem characteristics.

6. Parallel execution adds overheads due to communication, synchronization, and parallelization. With only a small number of cores (particularly 2 and 4), this overhead might outweigh the benefits of parallelism, leading to slower execution compared to a serial implementation.

7. The other possible reason for the poor performance of 2 and 4-core configuration could be that the parallel algorithm couldnot not scale efficiently with a small number of cores. Some parallel algorithms exhibit poor scalability on small core counts due to suboptimal parallelization strategies or bottlenecks that become more pronounced with fewer resources.

```
#!/bin/bash
#SBATCH --job-name=highlife_mpi
#SBATCH --output=highlife_mpi.out
#SBATCH --error=highlife_mpi.err
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=32
#SBATCH --time=00:02:00
```

```
#SBATCH --partition=el8-rpi

module load xl_r spectrum-mpi
#cd /gpfs/u/home/PCPE/PCPEpdlb/scratch/
    parallel_assignments/assign3/

mpirun --bind-to core --report-bindings -np
    $SLURM_NPROCS ./highlife_mpi 5 16384 32 false >>
    output_64.txt
```

Listing 1: batch script for job submission

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt


files = [f"time_file{i}.txt" for i in range(1,8)]

exec_time = []
for file in files:
    with open(file,'r') as f:
        data_cont = f.read()
        time = data_cont.split('=')[1]
        time = float(time.strip())
        exec_time.append(time)

mpiranks = np.array([1,2,3,4,5,6,12])

cellupdate = 16384*16384*mpiranks*128
cellupdate_rate = cellupdate/exec_time
print(cellupdate_rate)
scaled_speedup = cellupdate_rate/cellupdate_rate[0]
print(scaled_speedup)

plt.figure(figsize=(8, 4))
plt.plot(mpiranks,exec_time,'*r-')
plt.xlabel("Number of GPUs/MPI ranks")
plt.ylabel("Execution time (in sec)")
plt.legend()
plt.grid()
plt.savefig("executiontime_vs_gpus.png")


plt.figure(figsize=(8, 4))
plt.plot(mpiranks,scaled_speedup,'*r-')
plt.plot(mpiranks[6],scaled_speedup[6],'ob',label='
    Maximum Speedup')
plt.xlabel("Number of cores")
plt.ylabel("Scaled Speedup")
plt.legend()
plt.grid()
plt.savefig("Speedup_vs_cores.png")
```

Listing 2: Python code for plots and table