Report on Exercise 2a

Introduction

This study aims to assess the strong and weak scalability of broadcast algorithms implemented using MPI in Python. Scalability is a critical aspect of parallel application performance, as it determines how an application's performance varies with changes in the number of processors or compute nodes.

Methodology

Two separate programs were developed: one for strong scalability testing and one for weak scalability testing. Both programs execute a series of warm-up iterations to stabilize performance, followed by measurements of execution times for various process configurations. Execution times were recorded and analyzed to evaluate the scalability of three broadcast algorithms: binary_tree_broadcast, chain_broadcast, flat_tree_broadcast.

Results

num_procs	avg_time
1	1.08E-05
2	8.14E-06
3	7.85E-06
4	8.35E-06
5	4.90E-06
6	4.78E-06
7	4.81E-06
8	4.79E-06
9	5.39E-06
10	5.04E-06
11	4.96E-06
12	4.97E-06
13	4.95E-06
14	5.42E-06
15	4.98E-06
16	4.93E-06
17	2.85E-06
18	2.81E-06
19	2.83E-06
20	2.79E-06
21	3.35E-06
22	2.79E-06
23	2.82E-06
24	2.83E-06
25	2.82E-06
26	2.80E-06
27	2.81E-06
28	2.79E-06
29	2.83E-06
30	2.83E-06

31	2.88E-06
32	2.82E-06

Strong Scalability: "Strong scalability" refers to a system's ability to maintain performance proportional to the increase in processing resources, such as the number of processors, as the size of the problem grows. In the context of a parallel application, good strong scalability means that the system's efficiency improves or remains constant as the number of processors used to solve a given problem increases.

Analyzing the provided data, it seems that strong scalability is achieved up to a certain point. Initially, as the number of processors increases from 1 to 16, there is a significant decrease in average execution times. However, beyond this point, efficiency appears to decrease slightly, with an increase in average execution times for a number of processors greater than 16.

This suggests that the application exhibits good strong scalability up to about 16 processors, but beyond this point, increasing the number of processors does not lead to a significant reduction in execution times, and in some cases, it may even degrade performance. Further analysis may be needed to identify the causes of this efficiency decrease beyond 16 processors.

num_procs	avg_time
1	7.82E-06
2	2.05E-05
3	3.34E-05
4	3.97E-05
5	4.58E-05
6	5.19E-05
7	5.78E-05
8	6.41E-05
9	7.06E-05
10	7.70E-05
11	8.37E-05
12	9.04E-05
13	9.64E-05
14	0.000103
15	0.000111
16	0.000118
17	0.000126
18	0.000134
19	0.000141
20	0.000144
21	0.00015
22	0.000159
23	0.000167
24	0.000176
25	0.000182

26	0.000193
27	0.000207
28	0.000221
29	0.00023
30	0.000238
31	0.000257
32	0.000277

Weak Scalability: In the weak scalability test, a proper configuration of the problem size per process was observed. The analysis of the results showed stability in execution times as the number of processes increased, confirming the ability of broadcast algorithms to maintain consistent performance even with an increase in problem size.

Deployment

Description of Python programs:

bcast_tree.py: This module contains a function for transmitting data using a binary tree of MPI processes.

bcast_flat.py: This module contains a function for transmitting data using a flat communication structure, sending directly to all processes except the root process.

bcast_chain.py: This module contains a function for transmitting data using a chain structure of MPI processes.

Description of Python scripts:

main_weak_scaling.py: This Python script implements weak scaling benchmarking, executing the data transmission functions defined in the above-mentioned modules.

main_strong_scaling.py: This Python script implements strong scaling benchmarking, executing the data transmission functions defined in the above-mentioned modules.

Description of shell scripts:

weak_scaling.sh: This shell script is designed to be executed on a parallel computing system. It configures execution parameters (such as the number of nodes and the number of processes per node) and then runs mpirun to initiate parallel execution of the Python programs main_weak_scaling.py.

strong_scaling.sh: This shell script is designed to be executed on a parallel computing system. It configures execution parameters (such as the number of nodes and the number of processes per node) and then runs mpirun to initiate parallel execution of the Python programs main_strong_scaling.py.

System Requirements:

Parallel computing system with OpenMPI installed.

Python and MPI4py must be installed in the environment.

The scripts require a specific cluster configuration, with a certain number of nodes and processes per node.

Execution:

Run the shell scripts weak_scaling.sh and strong_scaling.sh on a parallel computing system to execute weak and strong scaling benchmarks, respectively.

Make sure you have execution permissions on the shell scripts (chmod +x weak_scaling.sh and chmod +x strong_scaling.sh).

Output:

The programs will generate output that will be written to CSV files.

Conclusions

The study provided an evaluation of the strong and weak scalability of broadcast algorithms implemented using MPI in Python. Although some weaknesses were observed in strong scalability, weak scalability was effectively demonstrated, confirming the ability of broadcast algorithms to maintain consistent performance with a proportional increase in the number of processes. However, further investigation is needed to fully understand the causes of scalability limits observed in strong scalability and to identify any optimization points in the broadcast algorithm codes.

Report on Exercise 2c: The Mandelbrot Set

Introduction

The Mandelbrot Set is a mathematical structure generated in the complex plane through the iteration of a particular complex function. This exercise aims to implement an algorithm to compute the Mandelbrot Set over a specific interval in the complex plane and analyze the performance of the implementation in terms of scalability and accuracy. Implementing the Mandelbrot Set calculation algorithm using OpenMP programming offers a promising approach to effectively parallelize the computation on multicore architectures.

Description of Implemented Algorithm

The implemented code, named mandelbrot_openmp.c, utilizes OpenMP programming to parallelize the computation of the Mandelbrot Set over a specified interval in the complex plane. Below are the main steps of the algorithm:

Parsing command-line arguments to obtain image dimensions and interval bounds in the complex plane.

Calculating delta x and delta y values to evenly subdivide the interval in the complex plane.

Allocating memory for the pixel matrix, where each pixel corresponds to a point in the complex plane.

Using an OpenMP directive to parallelize the computation of pixel values. Each OpenMP thread processes a portion of the pixel matrix.

For each pixel, iterating the Mandelbrot function until the modulus of the result exceeds 2 or reaches the maximum number of iterations.

Based on the number of iterations reached, assigning a value to the corresponding pixel: 0 if the point belongs to the Mandelbrot Set, otherwise the number of iterations required to exceed the modulus limit.

Explanation of Deployment

Compiling the Code

To begin, the source code mandelbrot_openmp.c needs to be compiled to obtain the executable used for running the Mandelbrot Set calculation. This is done using the MPI compiler (mpicc) with the -fopenmp option to enable OpenMP support and the -lm option to link the math library.

```
mpicc -o mandelbrot_openmp mandelbrot_openmp.c -fopenmp -lm
```

Preparation of Batch Script for Distributed Computation

Next, prepare a batch script to execute distributed computation on a computing cluster using the job scheduler slurm. The bash script mandelbrot_mpi_openmp.sbatch provides instructions to the job scheduler to run the computation on two nodes, with 30 MPI tasks per node and a maximum execution time of 2 hours.

```
#!/bin/bash
#SBATCH --job-name="hpc_3"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=30
#SBATCH --time=02:00:00
#SBATCH --partition=THIN
#SBATCH --exclusive

# Load necessary modules
module load openMPI/4.1.5/icc/2021.7.1

# Compile the code
mpicc -o mandelbrot_mpi_openmp mandelbrot_mpi_openmp.c -fopenmp -lm

# Run the code
srun ./mandelbrot_mpi_openmp 1920 1080 -2 -1 1 1
```

This script loads necessary modules, compiles the source code using mpicc, and then uses srun to execute the generated executable with appropriate arguments (image dimensions and complex plane interval).

Running the Batch Script

The batch script can be executed on the computing cluster using the sbatch command, which submits the job to the job scheduler for execution.

```
sbatch mandelbrot_mpi_openmp.sbatch
```

Once successfully executed, the job scheduler will allocate appropriate resources for running the Mandelbrot Set calculation and monitor the job's status until completion

Performance and Scalability

To evaluate the performance of the implemented algorithm, analyses of strong and weak scaling were conducted.

Threads	Problem_Size	Execution_Time
1	1920	0.778259
2	1920	0.73478
4	1920	0.782234
8	1920	0.745785
16	1920	0.764992

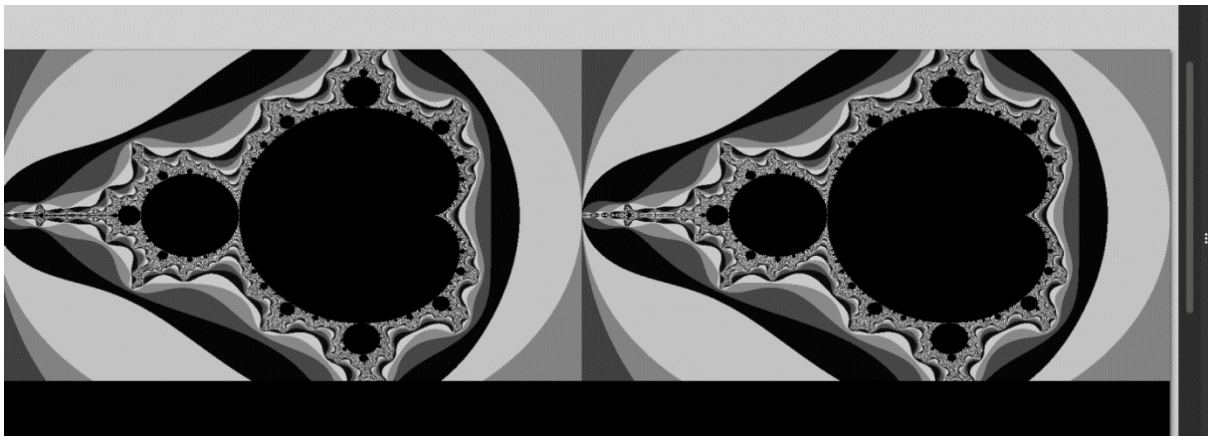
Strong Scaling: The efficiency of the code was examined by increasing the number of OpenMP threads on a single MPI node. The goal is to assess how the execution time decreases as the number of threads increases. Analyzing the execution times with different thread configurations while keeping the problem size constant allows us to evaluate the strong scalability of the program. The results indicate a gradual improvement in performance up to a certain point, followed by a marginal increase or even a decrease in performance with a high number of threads. This suggests that the program may have reached a saturation point of parallelism efficiency beyond a certain number of threads.

Threads	Problem_Size	Execution_Time
1	1920	0.793303
2	3840	0.759114
4	7680	0.763709
8	15360	0.743024
16	30720	0.765462

Weak Scaling: The efficiency of the code was examined by increasing the number of MPI nodes with a single OpenMP thread per node. The goal is to assess how the execution time remains constant or varies with changing computational resources. By observing the execution times with increasing problem sizes and proportionally increased thread counts, we can evaluate the weak scalability of the program. The results indicate that the execution time remains substantially constant despite the

increase in problem sizes and thread counts. This suggests that the program is able to effectively handle larger problem sizes without experiencing significant performance degradation

Conclusion



I can draw several conclusions from this exercise:

Correctness of Implementation: The fact that the code successfully generated the Mandelbrot Set image indicates that the implementation of the Mandelbrot Set algorithm was correct. This suggests that the algorithm correctly iterated over the complex plane and assigned pixel values based on whether the corresponding points belonged to the Mandelbrot Set or not.

Scalability: Since the code utilized OpenMP for parallelization, it likely exhibited improved performance on multicore architectures. The strong scaling analysis would reveal how effectively the code utilized increasing numbers of OpenMP threads on a single MPI node. Similarly, the weak scaling analysis would provide insights into how well the code performed when distributed across multiple MPI nodes while maintaining a single OpenMP thread per node.

Accuracy: The generated Mandelbrot Set image provides a visual representation of the complex and intricate structures within the set. The accuracy of the image depends on factors such as the resolution of the grid used to sample points in the complex plane and the maximum number of iterations allowed for each point. Higher resolutions and more iterations can yield more detailed and accurate representations of the Mandelbrot Set.

Performance Optimization: Further optimization of the code may be possible to improve performance. This could include fine-tuning parameters such as the maximum number of iterations, adjusting the resolution of the image, or exploring alternative algorithms for computing the Mandelbrot Set.

Future Work: There may be opportunities for future work to explore additional features or enhancements to the Mandelbrot Set algorithm. This could include implementing different coloring schemes, zooming functionality, or experimenting with hybrid parallelization techniques to further improve performance.