Introduction to Parallel and Distributed Processing
Assignment 0

Author : Kavitha Elizebeth Varughese
UBID : 50560694

Overview

In this report, we will be benchmarking a parallel system by assessing the scalability and performance of the system based on varying inputs parameters and the number of processors. To do this, we will be conducting the following analysis.

1. System functionality.
2. Time complexity based on input parameters.
3. Performance scaling based on increasing the number of processors.

System Functionality

The system performs **a gaussian kernel density estimation (GKDE)**. It has the sequential and the parallel implementation for GKDE. The implementation takes in 2 inputs, n and k, where n represents the size of the datapoints in the input vector x and k helps to find the range of neighbors which should be included for the density estimation.

The system implementation consists of find the starting and ending values for the range of neighbors included in calculating the GDE of each of the values in the vector set x. The implementation, then loops through the range of neighbors for each value in the vector set x to compute the density estimation, and saves the value in the vector set y for the corresponding values in x. It is to be noted that, the i^{th} value in y doesn't depend on any other computed values in y. Its dependency is solely on multiple values in x.

While the sequential implementation computes the values for each x and saves it in y one after the other, the parallel operation computes a set of values of each x in separate threads simultaneously using *OPENMPI*. The set of values of x being computed simultaneously depends on the number of threads that can run in parallel, which is specified by the user.

We will be benchmarking the parallel implementation of the system against the sequential implementation and understand how parallelism is beneficial.

Time Complexity

<u>Sequential</u>

The implementation consists of two nested for-loops. The outer loop iterates through all the values of the vector set x and the inner loop iterates through the neighbors included for the density estimation of the corresponding value in x. Thus, the time complexity would be $(n \cdot (length\ of\ neighbors))$.

From the implementation, we can understand that the maximum number size of neighbors would be $2 \cdot k$ for $k < n/2$ and n for $k >= n/2$. As $2 < k < n$, we can say that the inner for loops takes
$O(n)$. Thus, the time complexity of the sequential implementation would be $O(n^2)$.

Parallel

From the implementation, we see that the system tries to assign a new thread for the kernel density estimation for each value of x. Thus, the time complexity for a single thread depends on the inner loop which loops through the range of neighbors for the corresponding value of x. We know that the inner for-loop takes $O(n)$. Let the number of processors/threads (assuming threads are processor cores for this analysis) be p.

Thus, for $p == n$, we can say that the time complexity would be $O(n)$.
For $p << n$, let's assume that n can be divided by p, the time complexity would be $O(n^2/p)$

As each processor saves the computed values to different memory location, and the computed values do not depend on any past computed values, the processors are running independently in the parallel implementation.

Performance Analysis

System Specification

Hostname	cpn-f13-04.core.ccr.buffalo.edu
Number of Cores	64
CPU	Intel(R) Xeon(R) Gold 6448Y
RAM	2000M
Operating System	Ubuntu 24.04.1 LTS
Compiler	g++ (Gentoo 10.4.0 p5) 10.4.0
L1d cache	3 MiB

Input Specification

We have already shown that the value of k matters little to the time complexity of the data as it's a constant. We also know that the processors are running independently in parallel. That is, communication between the processors probably doesn't exist. But we most probably cannot neglect the cost of the memory accesses made by the system. Thus, we will be performing the analysis by choosing our inputs such that the number of memory accesses are maximized. This would give our analysis a true upper bound.

From the implementation, we understand that the memory accesses are performed by the inner loop and the number of times it's performed depends on the length of range of neighbors for a value in the vector set x. The maximum length of the range of neighbors, that we can set is n. So, if we can find a value of k such that most of the processors will have to iterate through n neighbors, we will be able to test with the maximum memory accesses. From further analysis of the implementation, we find the value of k as $(n\text{-}1)$. That is, for $k = n\text{-}1$, most of the processor will be forced to perform memory reads and writes n times.

Our values of $n = (32000\ 64000\ 128000\ 256000\ 512000\ 1024000)$
Our values of $p = (1\ 2\ 4\ 8\ 16\ 32)$, where 1 signifies sequential execution.
We will keep $k = n - 1$

Analysis Implementation

A bash script is written to execute a batch job on CCR using slurm. *[APPENDIX 1.1]*
A bash script is written to read the output file and create our csv file. *[APPENDIX 1.2]*
Finally, a python script is written to read the csv file and plot our graphs. *[APPENDIX 1.3]*

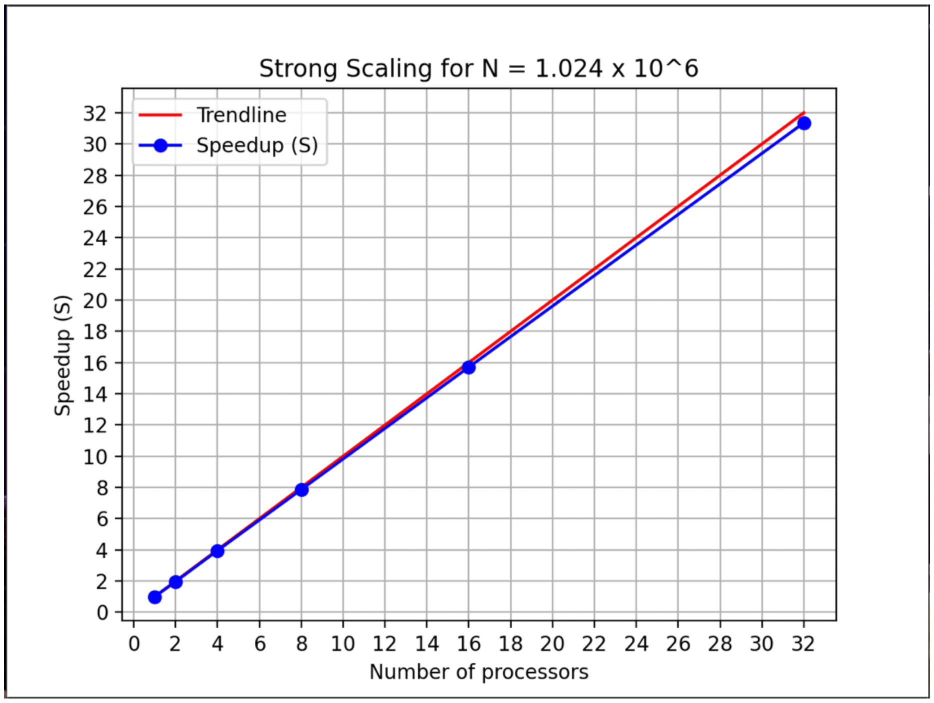
Results

N\P	1	2	4	8	16	32
32000	6.09158	3.08108	1.54175	0.772267	0.388739	0.198428
64000	24.0831	12.3029	6.15673	3.06992	1.54326	0.771596
128000	96.7398	48.9406	24.4729	12.283	6.15934	3.08285
256000	385.357	195.456	97.472	48.9763	24.5149	12.3052
512000	1537.12	778.643	388.079	195.423	98.1537	49.148
1024000	6157.36	3127.9	1556.53	782.859	392.098	196.389

Strong Scaling Analysis

As the processors are independently functioning, we will probably get a good scaling by keeping n constant.
Let $n = 1.024 \cdot 10^6$

P	Speedup	Efficiency %
1	1.0	100.0
2	1.9685284056395700	98.42642028197830
4	3.95582481545489	98.89562038637230
8	7.865222217538530	98.31527771923170
16	15.703625114129600	98.14765696331020
32	31.352876179419400	97.97773806068570



Observation

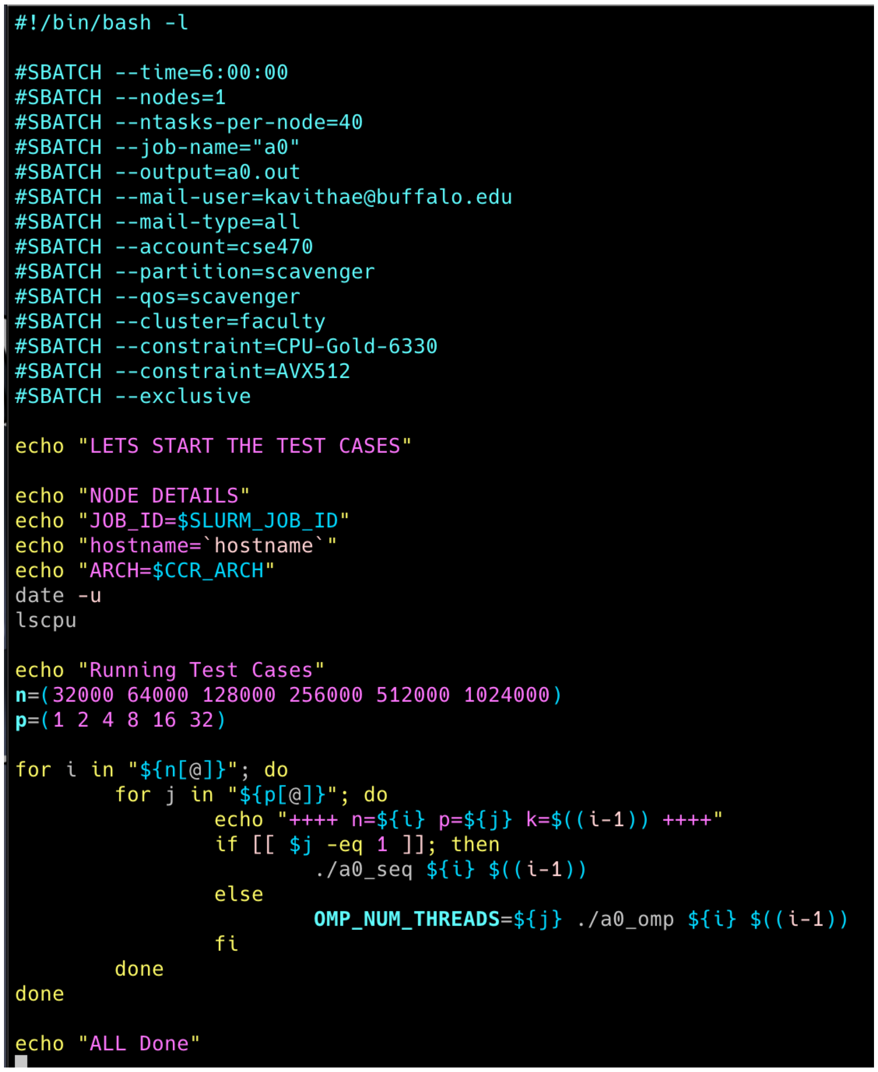
As the number of CPUs increase, the speedup (S) also improves very close the linear speedup. The small drop as we increase processors is probably due to the cost of memory accesses occurring simultaneously. If we further increase processors, the increased simultaneous memory accesses are going to be the bottleneck to this system.

References

1. https://en.wikipedia.org/wiki/Kernel_density_estimation
2. https://idl.cs.washington.edu/files/2021-FastKDE-VIS.pdf
3. https://docs.ccr.buffalo.edu/en/latest/
4. https://medium.com/@linuxadminhacks/understanding-the-bash-rematch-in-bash-183a5b065081

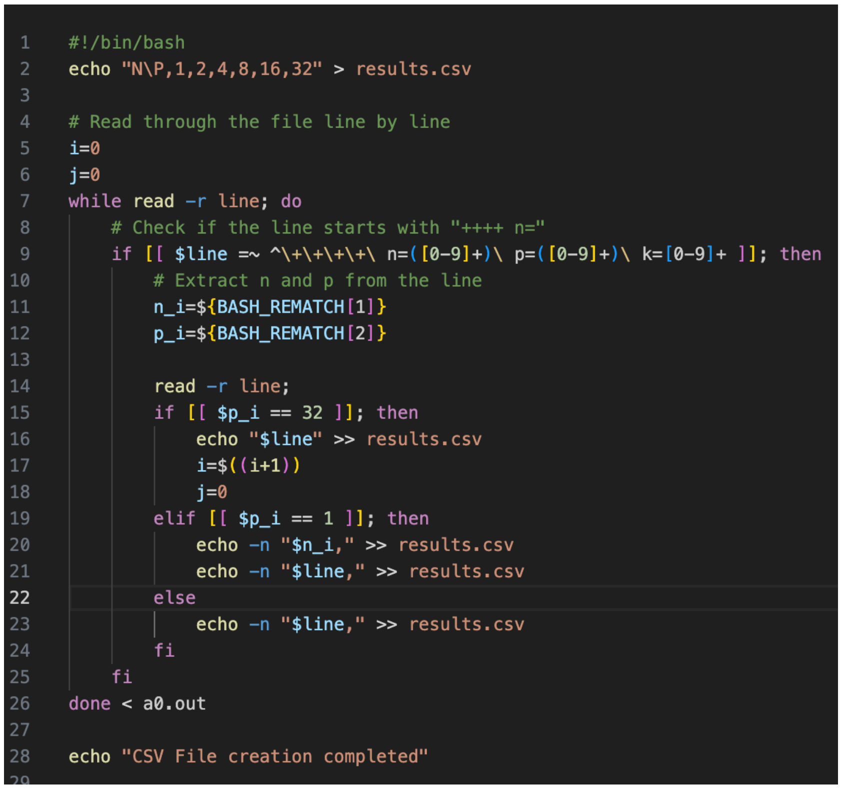
Appendix

1.1. CCR shell code to start Batch process



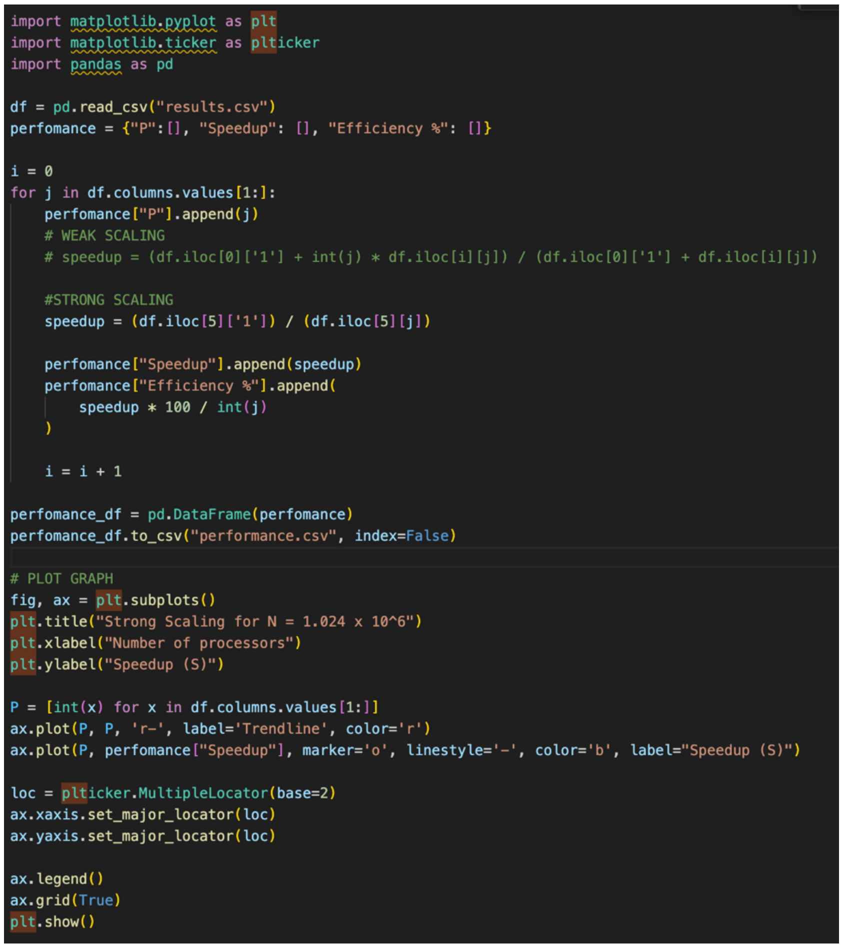
```
#!/bin/bash -l

#SBATCH --time=6:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --job-name="a0"
#SBATCH --output=a0.out
#SBATCH --mail-user=kavithae@buffalo.edu
#SBATCH --mail-type=all
#SBATCH --account=cse470
#SBATCH --partition=scavenger
#SBATCH --qos=scavenger
#SBATCH --cluster=faculty
#SBATCH --constraint=CPU-Gold-6330
#SBATCH --constraint=AVX512
#SBATCH --exclusive

echo "LETS START THE TEST CASES"

echo "NODE DETAILS"
echo "JOB_ID=$SLURM_JOB_ID"
echo "hostname=`hostname`"
echo "ARCH=$CCR_ARCH"
date -u
lscpu

echo "Running Test Cases"
n=(32000 64000 128000 256000 512000 1024000)
p=(1 2 4 8 16 32)

for i in "${n[@]}"; do
        for j in "${p[@]}"; do
                echo "++++ n=${i} p=${j} k=$((i-1)) ++++"
                if [[ $j -eq 1 ]]; then
                        ./a0_seq ${i} $((i-1))
                else
                        OMP_NUM_THREADS=${j} ./a0_omp ${i} $((i-1))
                fi
        done
done

echo "ALL Done"
```

1.2. Script to create the csv results file



```bash
#!/bin/bash
echo "N\P,1,2,4,8,16,32" > results.csv

# Read through the file line by line
i=0
j=0
while read -r line; do
    # Check if the line starts with "++++ n="
    if [[ $line =~ ^\+\+\+\+\ n=([0-9]+)\ p=([0-9]+)\ k=[0-9]+ ]]; then
        # Extract n and p from the line
        n_i=${BASH_REMATCH[1]}
        p_i=${BASH_REMATCH[2]}

        read -r line;
        if [[ $p_i == 32 ]]; then
            echo "$line" >> results.csv
            i=$((i+1))
            j=0
        elif [[ $p_i == 1 ]]; then
            echo -n "$n_i," >> results.csv
            echo -n "$line," >> results.csv
        else
            echo -n "$line," >> results.csv
        fi
    fi
done < a0.out

echo "CSV File creation completed"
```

1.3. Script to read the results and analyze and plot performance



```python
import matplotlib.pyplot as plt
import matplotlib.ticker as plticker
import pandas as pd

df = pd.read_csv("results.csv")
perfomance = {"P":[], "Speedup": [], "Efficiency %": []}

i = 0
for j in df.columns.values[1:]:
    perfomance["P"].append(j)
    # WEAK SCALING
    # speedup = (df.iloc[0]['1'] + int(j) * df.iloc[i][j]) / (df.iloc[0]['1'] + df.iloc[i][j])

    #STRONG SCALING
    speedup = (df.iloc[5]['1']) / (df.iloc[5][j])

    perfomance["Speedup"].append(speedup)
    perfomance["Efficiency %"].append(
        speedup * 100 / int(j)
    )

    i = i + 1

perfomance_df = pd.DataFrame(perfomance)
perfomance_df.to_csv("performance.csv", index=False)

# PLOT GRAPH
fig, ax = plt.subplots()
plt.title("Strong Scaling for N = 1.024 x 10^6")
plt.xlabel("Number of processors")
plt.ylabel("Speedup (S)")

P = [int(x) for x in df.columns.values[1:]]
ax.plot(P, P, 'r-', label='Trendline', color='r')
ax.plot(P, perfomance["Speedup"], marker='o', linestyle='-', color='b', label="Speedup (S)")

loc = plticker.MultipleLocator(base=2)
ax.xaxis.set_major_locator(loc)
ax.yaxis.set_major_locator(loc)

ax.legend()
ax.grid(True)
plt.show()
```