**A0 Report**

**Overview:**

This analysis examines the scalability of a parallel code implementation. The investigation focuses on how the code performs when subjected to different input parameters and processor counts. The report addresses three key questions:

1. What is the function of the code under examination?
2. How does the code's complexity relate to its input parameters?
3. In what ways does the code's efficiency and speed up change as we increase the number of processors?

Through this study, the aim is to provide a comprehensive understanding of the code's behavior and performance across various scenarios.

**What is the Function**

The code implements Gaussian Kernal Density Estimation (KDE) in a 1-dimensional array. The code takes two input values, *n* and *k,* representing the problem size and number of neighbors that should be taken into account in gaussian kde.

**Code Complexity:**

The function “omp\_gaussian\_kde” has two loops which are nested.

The outer loop runs for ‘*n’* times, once for each element in vector *‘x’*. Whereas the inner loop number of iterations is dependent upon values *‘f’* and ‘l’ which in turn are dependent on values of *‘k’.* Also, we have ‘*p’* processors on which the code can be parallelized.

So, the Time complexity for Sequential implementation is O(*n\*k*).

Time complexity for Parallel implementation for p processors is O(*n\*k/p*).

**Experimental Setup:**

**Hardware and Software Requirements**

To replicate the readings please use the below mentioned specifications:

* **CPU:** Intel(R) Xeon(R) Gold 6330
* **RAM:** 503GB
* **Compiler:** g++
* **Compiler Command:** g++ -g -fopenmp -std=c++20 -O2 a0.cpp -o a0

**Input Parameters**

To complete the experiment, different values of n and k were tried to get the reading over the precision of 0.1s. Finally, the value of ‘k’ was fixed at 400 while ‘n’ was set to: 1000000, 2000000, 4000000, 6000000, 8000000, 16000000 and 32000000.

To assess the code's performance at various levels of parallelism, various number of processors (p) were used. Processors count selected are - 1 (representing a sequential run), 2, 4, 6, 8,16 and 32, while ensuring a consistent N/P ratio and staying within memory limits. This approach allowed effective evaluation of the code's scalability and efficiency across different processing scenarios.

**Scripting**

The data collected was done by running the cpp files directly on the compute nodes by getting access to them using ‘salloc’. The data can be regenerated by running the experiment with the following commands.

The following command helps us to get an exclusive node for CPU-Gold-6330 for 2 hours:

*salloc --partition=general-compute --qos=general-compute --constraint=CPU-Gold-6330 --exclusive --time=2:00:00 --nodes=1 --ntasks-per-node=1 --cpus-per-task=48 --no-shell*

Once the allotment is done a job id gets generated which is passed in the code below to access the compute node:

*srun --jobid=#JobID --export=HOME,TERM,SHELL --pty /bin/bash --login*

**Experiment Results**

The table below shows the result of the experiment conducted. These values can now be assessed to understand the scalability and performance of the code with different problem sizes(n) and number of processors (P/N). The execution times, reported in seconds, represent the average of five independent runs, ensuring statistical reliability.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Table showing Number of processors (P) and Input value N (N) when k=400 | | | | | | | |
| P/N | 1000000 | 2000000 | 4000000 | 6000000 | 8000000 | 16000000 | 32000000 |
| 1 | 5.5 | 11.14 | 22.28 | 33.43 | 44.58 | 89.1 | 178.38 |
| 2 | 2.79 | 5.58 | 11.15 | 16.74 | 22.31 | 44.61 | 89.2 |
| 4 | 1.39 | 2.79 | 5.58 | 8.35 | 11.18 | 22.35 | 44.7 |
| 6 | 0.93 | 1.86 | 3.73 | 5.59 | 7.45 | 14.91 | 29.82 |
| 8 | 0.7 | 1.4 | 2.78 | 4.19 | 5.59 | 11.18 | 22.36 |
| 16 | 0.35 | 0.7 | 1.39 | 2.09 | 2.79 | 5.58 | 11.18 |
| 32 | 0.18 | 0.35 | 0.7 | 1.05 | 1.4 | 2.79 | 5.59 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Speed Up when k=400 | | | | | | | |
| P/N | 1000000 | 2000000 | 4000000 | 6000000 | 8000000 | 16000000 | 32000000 |
| 1 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 2 | 1.97 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 |
| 4 | 3.96 | 3.99 | 3.99 | 4.00 | 3.99 | 3.99 | 3.99 |
| 6 | 5.91 | 5.99 | 5.97 | 5.98 | 5.98 | 5.98 | 5.98 |
| 8 | 7.86 | 7.96 | 8.01 | 7.98 | 7.97 | 7.97 | 7.98 |
| 16 | 15.71 | 15.91 | 16.03 | 16.00 | 15.98 | 15.97 | 15.96 |
| 32 | 30.56 | 31.83 | 31.83 | 31.84 | 31.84 | 31.94 | 31.91 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Efficiency when k=400 | | | | | | | |
| P/N | 1000000 | 2000000 | 4000000 | 6000000 | 8000000 | 16000000 | 32000000 |
| 1 | - | - | - |  | - | - | - |
| 2 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 4 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 6 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 8 | 0.98 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 16 | 0.98 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 32 | 0.95 | 0.99 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 |

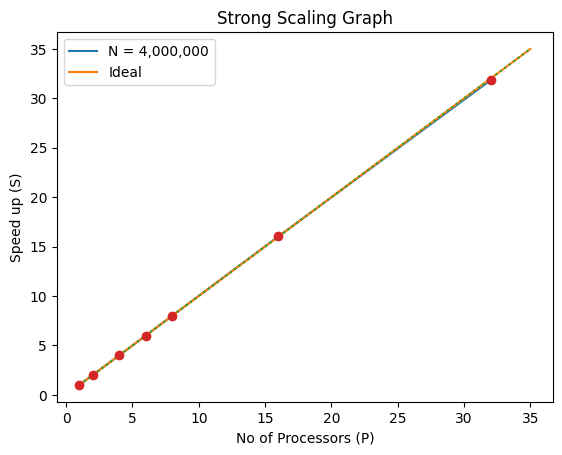
**Strong Scaling Analysis:**

While analyzing strong scaling, we must check how the code's execution time changes when input or problem size(n) remains the same, but number of processors(p) is increased.

For strong scaling I would be using the data for the column with n=4,000,000. The results for the table are summarized below:

|  |  |  |  |
| --- | --- | --- | --- |
| P | Runtime (s) | Speedup (S) | Efficiency (E) |
| 1 | 22.28 | 1 | 1 |
| 2 | 11.15 | 2 | 1 |
| 4 | 5.58 | 3.99 | 1 |
| 6 | 3.73 | 5.98 | 1 |
| 8 | 2.78 | 8.01 | 1 |
| 16 | 1.39 | 16.03 | 1 |
| 32 | 0.7 | 31.83 | 0.99 |

From the table we can see that we have recorded an efficiency of almost 100% in most cases, which is ideal for us. Hence it shows that there is strong scaling as with constant problem size, the runtime is decreasing as we keep increasing the number of processors.



The reason for strong scaling can be that the values are small enough to be stored in cache or memory.

**Conclusion**

The Gaussian Kernel Density Estimation (KDE) implementation analyzed in this report demonstrates impressive parallel performance and scalability. It exhibits strong scaling characteristics, maintaining near-ideal efficiency as the number of processors increases while keeping the problem size constant. This near-linear speedup indicates effective parallelization with minimal communication overhead. The performance remains consistent across various problem sizes, displaying the robustness of the implementation for handling large datasets. Efficient cache and memory utilization, particularly for smaller sizes, further enhances its performance. Overall, this implementation is well-suited for high-performance computing environments, making it a valuable tool for density estimation tasks.