Parallel SMOTE with OpenMP

Jared Goldman - jaredmgoldman@gmail.com
Mahler Revsine - mrevsine@gmail.com

Why SMOTE?

Description

Synthetic Minority Oversampling Technique (SMOTE) is an algorithm that generates new values for classes that are underrepresented in a dataset. A common example of an application of this is in fraud classification where cases of non-fraudulent transactions vastly outweigh fraudulent ones. When training a ML model to identify instances of fraud, it is advantageous to generate artificial cases in order to balance the data.

The algorithm follows the process:

1) Determine the euclidean distance between all elements of the minority class.
2) Create new data points by interpolating between combinations of two datapoints using the formula:

$$x_{new} = x_{old} + rand(0, 1) * |x_{new} - x_{old}|$$

Motivation

We chose to focus on the SMOTE algorithm due to its wide applications within machine learning research as well as within our own fields of interest. Mr. Revsine is currently studying computational biology where he has developed dna sequencing applications and is beginning to develop ML classification models for immunological problems. Mr. Goldman is currently researching computational optics and works within cloud-native AI risk analysis.

All of these fields rely on unbalanced datasets. By applying SMOTE to the problems that we face, the results generated are noticeably improved. When we face problems that require massive datasets and fast reaction times, we need a data processing framework that is suited to our needs. Our parallel SMOTE implementation provides this.

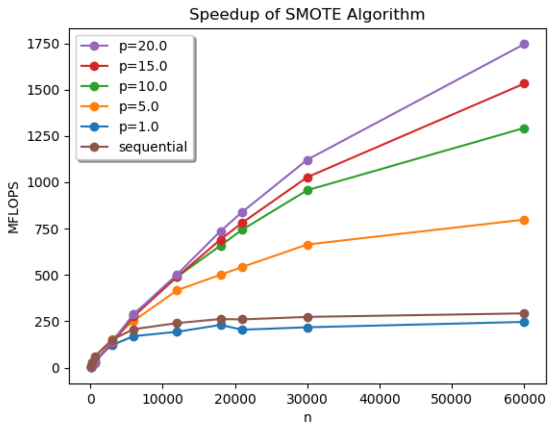
Methods

We chose to use OpenMP for our parallel algorithm due to its low-level control and ease of use. The body of our parallel SMOTE algorithm is dominated by a distance matrix calculation with time complexity N^2. We parallelized this loop using two strategies. For one, we surrounded it with a pragma parallel directive using a dynamic scheduler. Additionally, we split the nested loop into two loops as in the n-body problem. The converts the "upper triangle" execution pattern into a rectangle, where every iteration of the outer loop has even work. Without this step, the processor handling i=0 would have N work while the processor handling i=N-2 would have 1 work; this balances execution so that every loop has N/2 work.

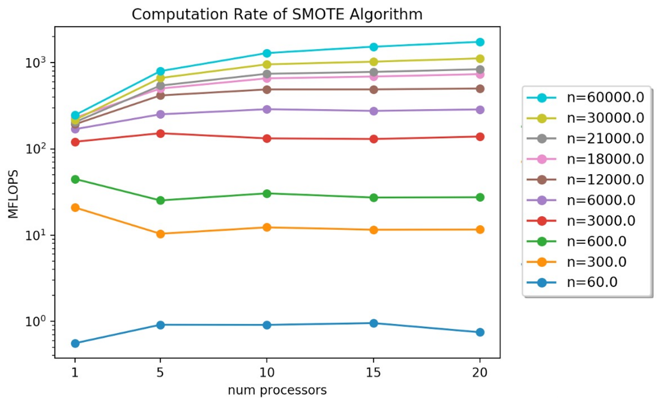
Another important loop in our parallel algorithm involves generating new data points for every original input data point. For this section we subdivide our input space into chunks of size N/p and assign each to a processor using a static pragma directive. On data generation, each processor only looks within its chunk of the distance matrix, improving cache locality. Finally, we set **KMP_AFFINITY="granularity=fine, compact, 1, 0"** environment variable to improve thread performance.

Conclusions

We noticed significant speedup between our parallel and sequential models, consistent with the desired optimal efficiency of a parallel system with p processors. The plot of the performance of our program using different numbers of processors relative to the sequential model is shown below. For this plot, we did not employ a logarithmic scale because the sequential model's relative performance remained unchanged as problem size increased and outperformed the parallel model for small problem size and p. This is a result of the increased overhead resulting from parallelization combined with relatively low floating point operation count. See the speedup plot for an illustration of this analysis.



Our analysis also demonstrates that after a certain minimum problem size, increasing the number of processors correlates to exponential improvement. Since the number of floating point operations grows at a rate of $O(N^2)$ where N is the number of minority elements, we observed significant improvements as more processors were incorporated at around 1 million floating point operations. This is a result of the relative cost of problem size versus parallelization. Meaning that for smaller problems, the overhead of the parallelization process is relatively large when compared to problems over a certain threshold. This trend is illustrated in the computational intensity plot below.


Computation Rate of SMOTE Algorithm
n=60000.0
n=30000.0
n=21000.0
n=18000.0
n=12000.0
n=6000.0
n=3000.0
n=600.0
n=300.0
n=60.0
MFLOPS
num processors
10^0
10^1
10^2
10^3