Application of High Performance Computing for Calculation of Reserves for a Company

J Bhanu Teja , Pallav Kumar Baruah , Satya Sai Mudigonda , and Phani Krishna Kandala

Abstract— Calculating the reserves for an insurance company is very crucial. The exercise of reserving is done in periodic fashion. Depending upon the reserve estimates the company will plan how to invest their money in different avenues and gain profit. Time required to calculate the reserves will grow exponentially if the input data size increases. This computation of reserving might need to be done several times taking different factors into consideration. Hence the computation becomes even more costly in terms of time.

We applied HPC to calculate reserves using chain ladder method, inflation adjusted chain ladder method,Inflation adjusted Bornhuetter-Ferguson. Using GPUs we showed an improvement of 840X speed up compared to the serial execution for Inflation adjusted Chain Ladder Method and an improvement of 940X speed up for Inflation adjusted Bornhuetter-Ferguson compared to the serial execution maintaining the accuracy.

Index Terms— Accident year cohort, Chain ladder method (CLM), CUDA, Development year, Graphical Processing Unit (GPU) , IBNR, Inflation Adjusted CLM, Reserve,

1 INTRODUCTION

Reserving is a complex and intensive calculation process for estimating liabilities of an organization. Insurance and reinsurance companies invest lot of their resources in this activity on a continuous basis. To improve the performance of this frequent activity, we explored the possibility of using High Performance Computing (HPC).

J Bhanu Teja is currently pursuing masters degree in Computer Science in Department of Mathematics and Computer Science in Sri Sathya Sai Institute of Higher Learning, Puttaparthi, India.
PH: +91 7989387983, Email: bhanu257@gmail.com.

Pallav Kumar Baruah, Head of Department, Department of Mathematics and Computer Science in Sri Sathya Sai Institute of Higher Learning, Puttaparthi, India.
PH: +91 9440699887, Email: pkbaruah@sssihl.edu.in.

Satya Sai Mudigonda is a professionally qualified associate actuary and management consultant. He is currently teaching the postgraduate students in Department of Mathematics and Computer Science in Sri Sathya Sai Institute of Higher Learning, Puttaparthi, India. PH: +91 9603573032, Email:satyasaibabamudigonda@sssihl.edu.in

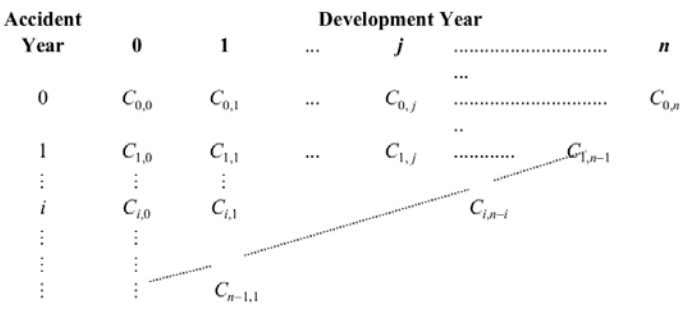
Phani Krishna Kandala, is currently Assistant Vice President in Swiss Re. He has done in Master's from Sri Sathya Sai Institute of Higher Learning, Puttaparthi, India.
PH:+91 91 82 472136, Email:kandala.phanikrishna@gmail.com

In recent years, HPC has been applied in diverse fields of finance. Focus has been towards security and derivative pricing. Joshi [2] priced Asian options and achieved a speed up of 150X. Nguyen [3] parallelized Cox-Ross-Rubinstein pricing model on GPUs and showed a speed up of 30X. Further, in 2012 Tucker and Bull [1] have explored HPC to insurance company solvency calculations and achieved a substantial improvement in performance over commercial software.

For many financial applications GPUs proved to be successful platforms for such intense calculations. In our work, we applied HPC to calculate reserves using the Chain Ladder Method (CLM) as well as inflation adjusted CLM with a more focus on the later as it involves increased set of calculations. Reserve estimation calculations provide a great opportunity to exploit technical superiority of HPC over traditional way of computing. Results of our work show that substantial improvement in performance (speed more than 940X) can be achieved using CUDA programming on GPUs**.**

There are many reasons why reserving is done the most common ones are:

- To equip managers make informed decisions based on estimated calculations.
- To assess the value of a company for purchase or sale.
- To compare the achieved versus expected results.
- To assess profitability business unit / product wise.
- To provide inputs for premium rating process
- To prepare accounts for insurer and regulators.



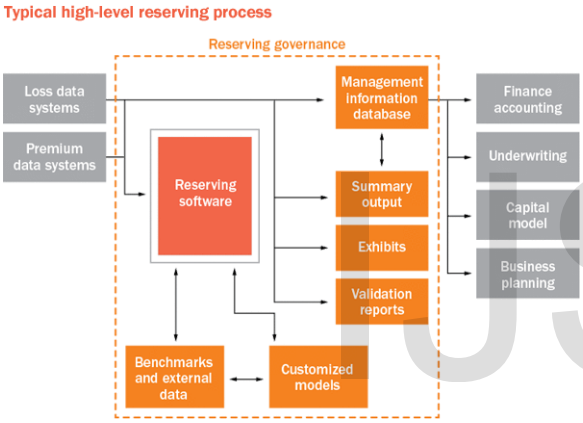


FIG.1 HIGH LEVEL VIEW OF RESERVING.[4]

1.1Motivation

Reserving is a costly process. High performance computing can reduce this price. The key reasons why there is need for HPC in reserving are:

- Scalability - be able to estimate liability for larger data sets
- Frequency - be able to perform calculations more frequently and also dynamically.
- Time - for the time-efficient use of the system resources
- Cost - for the cost-efficient use of the resources involved
- Quality - to maintain or improve the quality of output

The figure below shows how frequently each of the reserving methods are used. Chain ladder stands top of the chart with its wide use. Inflation adjusted is ever more powerful but rarely used in practice because it takes a lot of time. So we in this work parallelised this method to make it available in real time.

The rest of the paper is organized as follows. Section 2 talks about the concept and reserving methods which are chain ladder,inflation adjusted chain ladder and inflation adjusted Bornhuetter methods. Section 3 provides implementation details and results we got. Section 4 concludes by giving the summary and provides the necessary information for further improvement in this study.

2 METHODS AND METHODOLOGY

2.1 Incurred Triangles

For every accident year cohort, let us think there are n number of claims which occurred during the period, but only x claims (x <n) were reported to the insurer. The unreported claims are known as IBNR claims which we are trying to estimate using chain ladder and Bornhuetter Ferguson methods. We basically use historical/past experience by obtain the future reported claims. Mathematically, the run off triangle general form will be expressed as follows:

Each entry cij, represents the incremental claims and can be expressed as

$$C_{ij} = l_j \; D_i * A_{i+j} + r_{ij}$$

Where r_j is the development factor for year j, representing the proportion of claim payments in development year j.

Each lj is independent of the origin year.

D_i is a parameter varying by origin year, i, representing of exposure.

A_{i+j}, is a parameter varying over the calendar year, e.g. inflation.

r_{ij} error term .

FIG 2. RUN OFF TRIANGLES

Reserving is estimating the lower triangle i.e. squaring the triangle as shown in the below figure.

2.2 CHAIN LADDER METHOD

Chain ladder[5] is a traditional method based on statistics, used for estimating the ultimate value of a set of development data. The main idea behind this method is that, an average of past development is projected onto the future. Based on calculations done by actuary, the projection for successive periods in future is done using the ratios of cumulative past development.

The basic chain ladder method takes the form:

Cij = lj * Di + rij using the statistical model which was described above.

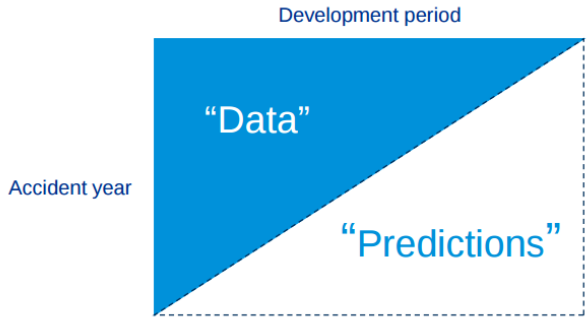


Fig. 3. ESTIMATING THE LOWER TRIANGLE

The assumptions for this method are:
– The assumed future pattern of claims development derived from past experience will remain stable in the future.
– The first accident year is fully run-off or its development to ultimate can be predicted with confidence.
– An explicit inflation assumption is made, for both past and future claims.
– The choice of inflation index is key to the accuracy of the method.
– Works well for stable, reliable and consistent data .

Algorithm 1: Chain Ladder Method

Input : Input triangle in incremental form
Output: Reserve estimates
1 **Step 1: (Finding cumulative):**
2 The incremental triangle is converted into cumulative triangle
3 **Step 2: (Development factors)**
4 Find the development factors $\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-1}$ from cumulative triangle.
5 **Step 3: (Estimate reserves)**
6 Using the development factors $\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-1}$ find the reserve estimates
7 **Step 4:(Incremental)**
8 The cumulative values are converted into incremental.

FIG.4. CHAIN LADDER METHOD

2.3 INFLATION ADJUSTED CHAIN LADDER

The difference between this method and the basic method is that an inflation index is applied to the past claims data to bring it into line with the latest year, and to inflate the projected claims to the expected year of payment.
This method requires an appropriate inflation index to be available for the business being considered.
The choice of index is key to the success of reserving using this method,including changes in inflation over time and choice of suitable benchmarks where data is sparse.

Dealing with past inflation:
In the case of run-off triangles which are grouped under calendar year, the claims inflation will affect the payments. The idea behind the working of inflation adjusted chain ladder method is that, the payments in the triangle are adjusted to the inflation by taking into the account the corresponding inflation factor Firstly the incremental payments, are to be calculated using the cumulative totals, because while adjusting to inflation, it is necessary to consider payments in each calendar year rather than cumulative totals. This is done by finding successive difference along each row.

Dealing with future inflation:
When adjusting to the effect of inflation, the cumulative payments don't consider the effect of future inflation. So assuming a rate of future inflation based on past information available on the inflation factors estimate.

Algorithm 2: Inflation adjusted Chain Ladder Method

Input : Input triangle in incremental form, inflation factor
Output: Reserve estimates
1 **Step 1: (Adjust to inflation):**
2 The input triangle is adjusted to inflation by applying the inflation factors to non cumulative data in order to get all the claims data into monetary terms of the recent accident year.
3 **Step 2: (finding cumulative):**
4 Accumulate the data and calculate development factors
5 **Step 3: (Development factors)**
6 Find the development factors $\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-1}$ from cumulative triangle.
7 **Step 4: (Estimate future claims)**
8 Using the development factors $\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-1}$ project the cumulative claims for the future.
9 **Step 5: (Incremental)**
10 Disaccumulate the data to make it incremental.
11 **Step 6: (Adjust to inflation)**
12 By applying inflation assumptions made for future, the outstanding claim payments are converted into amounts corresponding to future years

FIG. 5. INFLATION ADJUSTED CHAIN LADDER METHOD

2.4 BORNHUETTER-FERGUSON METHOD

The Bornhuetter-Ferguson method combines the estimated loss ratio with a projection method. The assumptions of this method are similar to the chain ladder method.
The concepts behind the method are:
– That whatever claims have already developed in relation to a given origin year, the future development pattern will follow that experienced for other origin years.
– The past development for a given origin year does not necessarily provide a better clue to future claims than the more general loss ratio.

Algorithm 3: Inflation adjusted Bornhuetter Ferguson Method

Input : Input triangle in incremental form, inflation factor

Output: Reserve estimates

Step 1: (Determine Loss ratio):

Find the amount of Loss ratio

Step 2: (Adjust to inflation):

The input triangle is adjusted to inflation by applying the inflation factors to non cumulative data in order to get all the claims data into monetary terms of the recent accident year.

Step 3: (Find Development factors):

Find the development factors $\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-1}$ from cumulative triangle.

Step 3: (Cumulative development factors):

Calculate the cumulative development factors

Step 5: (future claims):

For each accident year that is not fully run-off:

Using premiums and loss ratios find the initial estimate of total ultimate loss.

The expected amount paid so far is calculated using the initial estimates and cumulative development factors.

Using the amounts calculated in the above step, estimate how much should be paid in future(emerging liability).

The emerging liability plus the reported liability is the revised estimate of ultimate loss

Step 6: (Find Total Estimates):

The total estimate of liability is determined by summing up all the revised estimates of ultimate losses for all accident years.

FIG. 6. BORNHUETTER-FERGUSON METHOD

3 RESULTS

Based on available industry data, relevant features required for calculating reserves have been extracted from the data. These features were used to generate 10 million records for this purpose. This data was validated by experts from the field of actuaries. All the implementations were performed on this data. For testing the accuracy of the model, we ran the model on the available real data. The implementation details, the analysis of the performance, and accuracy were discussed in below sections.

The input to all the methods, are matrices which are obtained from the previous chapter after cleaning the data. These are known as input triangles, because the lower triangle elements of the matrices excluding the diagonal elements are all zeros. Implementations of the reserve methods which were discussed in previous section, will estimate these lower triangle values using the values in the upper triangle of the matrix.

3.1 EXPERIMENTAL SETUP

For serial implementations we have used the intel i5 processor. The parallel Cuda[7] versions were run on NVIDIA GeForce TITAN X GPUs. OpenMP version 4.0 [6]was used. The technical details about both systems are given in tables below. For validation of the results the built in packages in R(version
3.4.3) were used.

Processor	Intel(R) Corei5-4670 CPU @ 3.40GHz
cores	4
CPU max MHz	3600MHz
CPU min MHz	800MHz
Memory	16GB DDR3
Max memory bandwidth	21 GB/s

FIG. 7. CPU SPECIFICATIONS

GPU Model	NVIDIA GeForce TITAN X
Cudacores	3072
Clock speed	1000MHz
TFLOPS	6.144
Effective memory speed	7012
Memory bus	384 bits
RAM	12GB GDDR5
Memory BW	336GB/s
Single Precision	7 TFLOPS
Double Precision	0.2 TFLOPS
CUDA toolkit	CUDA v7.5

Fig. 8. GPU SPECIFICATIONS

3.2 CHAIN LADDER METHOD

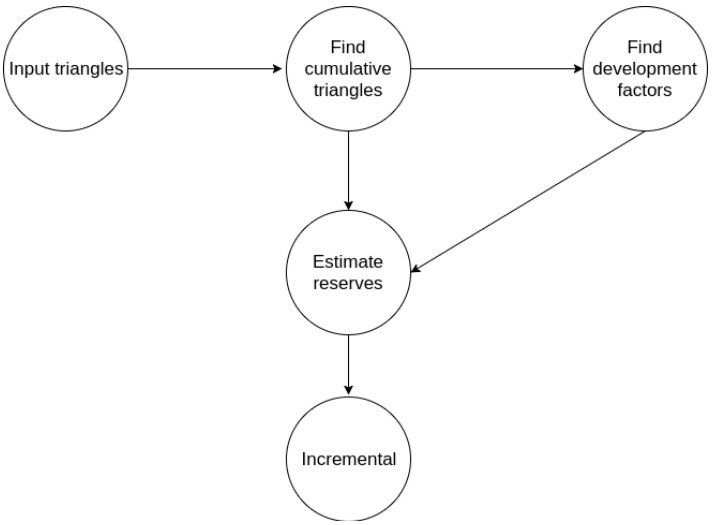


FIG.9 EXECUTION FLOW OF CHAIN LADDER METHOD

Firstly, the serial version of chain ladder method, which was discussed in the previous section, was implemented using C programming language. The method was run on different sizes of data. To the test the correctness of the implementation we have run it on the real data and results were compared.

Results were even compared to ones calculated using the built in packages in language R. Our implementation of the method give exact results, excluding few rounding off errors which are agreeable in this context. The below figures hows the time taken for the model to estimate the reserves for given sizes of the matrices.

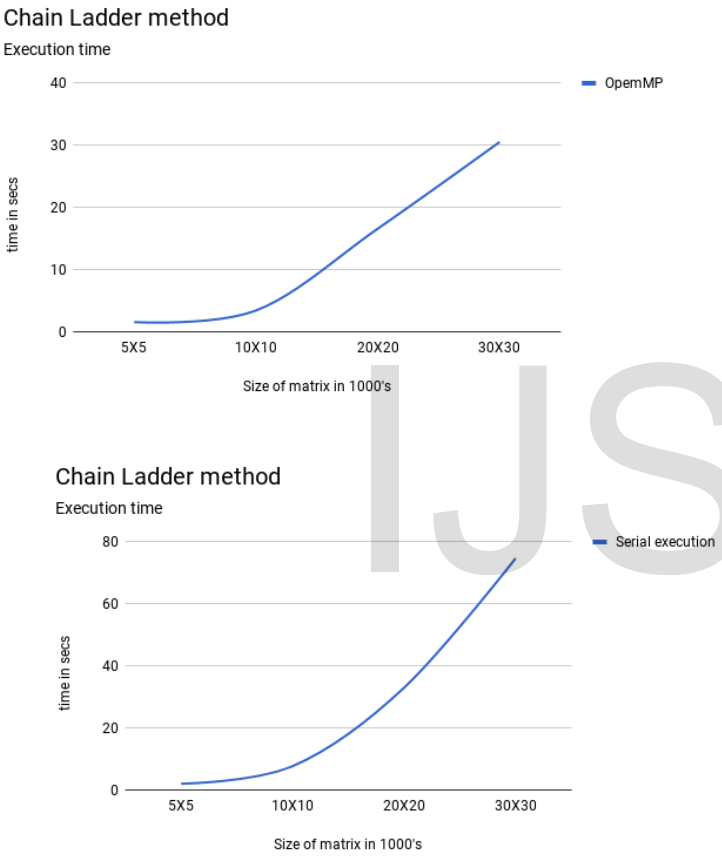


FIG .10. EXECUTION TIME OF CHAIN LADDER METHOD

As, we can clearly see from the above graph, time taken to run the method increased exponentially as the input size increased. So, using profiling tool gprof, the method was profiled for different sizes of matrices varying from 5000*5000 to 30000*30000. It was found that, for a $O(N^2)$size matrix there were $O(N^2)$ floating point operations being performed. The profiling results are as shown in the below pie chart.

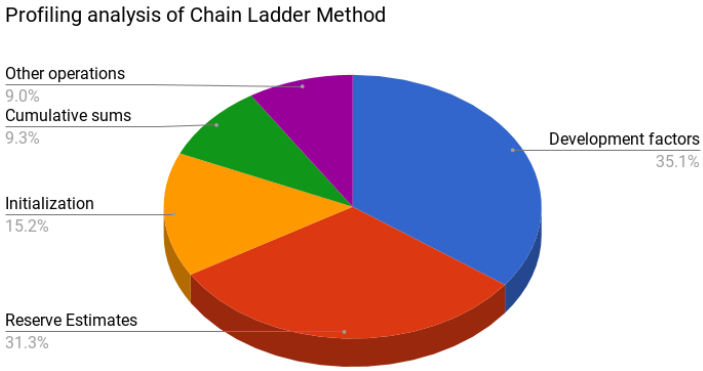


FIG.11. PROFILING ANALYSIS OF CHAIN LADDER METHOD

It was found out that 35% of execution time was spent in module which was calculating the development factors, and around 31% of the time was spent module which estimates the reserves using the development factors, i.e. the last step in the process. Other modules together contribute to 34% of the time. So using the OpenMP compiler directives the serial method was first parallelized. The two modules which were taking almost 66% of the execution time were parallelized by using OpenMP directives like #pragma omp parallel,#pragma omp parallel for, #pragma omp parallel sections etc.

This OpenMP parallel implementation was run on the intel i5 system. It was run for different data sizes as mentioned above for the serial implementation. Finding the best value of the num threads parameter was little tedious. Different values were tried to see which one gives the best performance, and finally it was inferred that 8 was the best possible value to get good performance.

The results were validated again using the real data and serial implementations. Except for few round of errors, the results were accurate. The below graph shows the time taken for execution of the OpenMP-parallel implementation. From the graph it can be inferred that almost 2X performance was gained using the parallel implementation.

FIG .12. OPENMP-EXECUTION TIME OF CHAIN LADDER METHOD.

Taking the cue from improvement we got using OpenMP, the method was coded using CUDA programming language to run on GPUs. The whole code was built from scratch using CUDA. All possible modules which don't have dependency among them were parallelized. For the computations, which are dependent on just single element of the matrix, each thread was given the work to compute that particular task. If the computations were row dependent or column dependent, then the division of work for threads was according to corresponding blocks. Not only every possible module was parallelized, memory management techniques like memory coalescing were implemented to achieve speed up.

So by allocating the work as described above, the CUDA implementation was run on GPUs for same data sizes as described above. Again the results were validated using the real data and the serial implementation results. The results were accurate.The graph below shows the time taken to run by CUDA implementation of chain ladder method.

Now if we compare the results of all three implementations. The GPUs performed better than both serial and OpenMP implementations. A maximum speed of up 4X was achieved using the GPUs. The figure below compares the time taken by all the three implementations and shows the improvement achieved using GPUs.

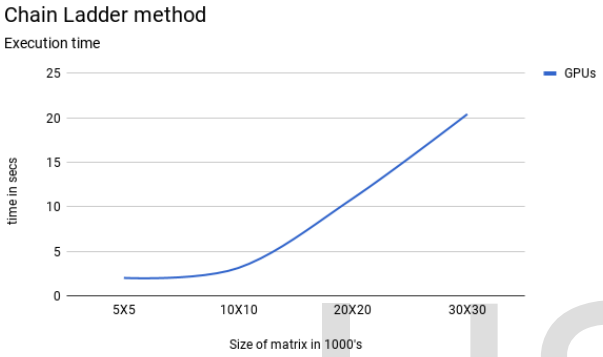


FIG .13. CUDA-EXECUTION TIME OF CHAIN LADDER METHOD

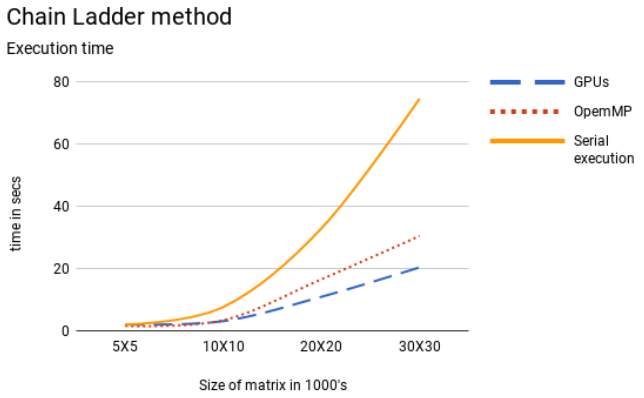


FIG .14. COMPARISON OF EXECUTION TIMES OF CHAIN LADDER METHOD

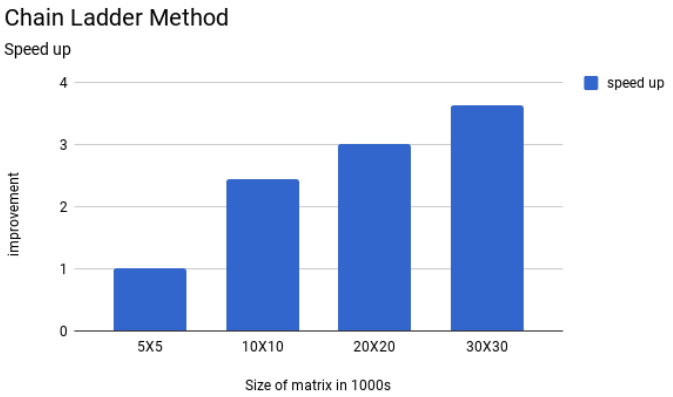


FIG .15. SPEED UP ACHIEVED FOR CHAIN LADDER METHOD

3.3 INFLATION ADJUSTED CHAIN LADDER

The difference between the Chain Ladder and Inflation adjusted Chain Ladder method was described in the previous section. Accordingly the appropriate changes were made in the serial code of Chain Ladder to make it Inflation Adjusted CLM. Then the serial code was run for different data sizes similar? .

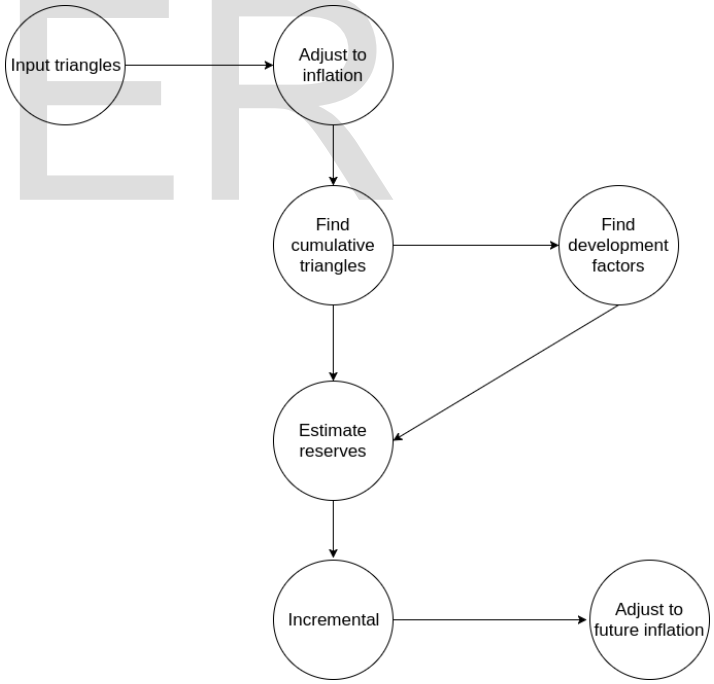
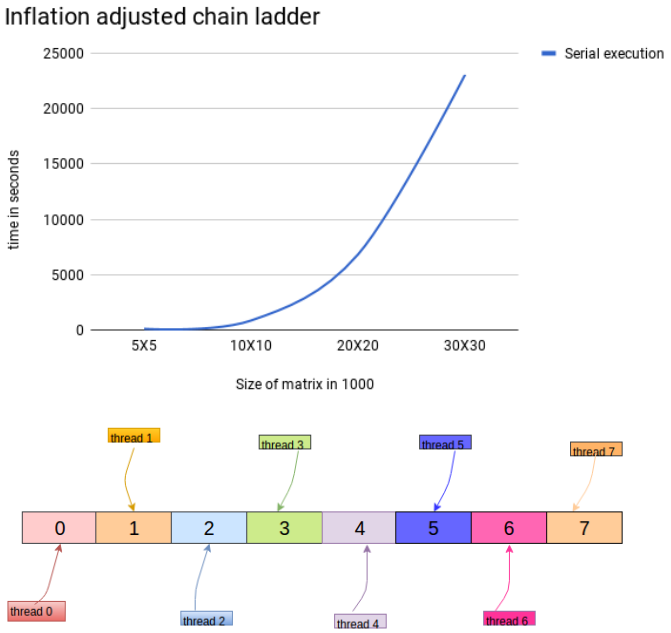


FIG.16. EXECUTION FLOW OF INFLATION ADJUSTED CHAIN LADDER METHOD

The figure below shows the time taken for the serial implementation of inflation adjusted CLM. From graph we can observe that as the data size was increased the time taken to calculated the reserves grew exponentially. To formulate in terms of complexity for a O(N2) size matrix there are O(N3) floating point operations. Profiling through gprof tool for different sizes of data has shown that almost 99% percent of

the execution time is spent in calculating the effect of inflation



on the input triangle.

FIG .17. EXECUTION TIME OF INFLATION ADJUSTED CHAIN LADDER METHOD

Module	Percentage of time taken
Adjust to Inflation	99.29
Development factors	00.45
Estimate future claims	00.18
Others	00.10
Dis-accumulating	00.05
finding cumulative sum	00.03

FIG.11. PROFILING ANALYSIS OF INFLATION ADJUSTED CHAIN LADDER METHOD

As before first, the serial code was converted into parallel using OpenMP compiler directives similar to ones discussed in section 3.2 and an improvement of around 1.5X was seen. This can be visualized from the figure below.

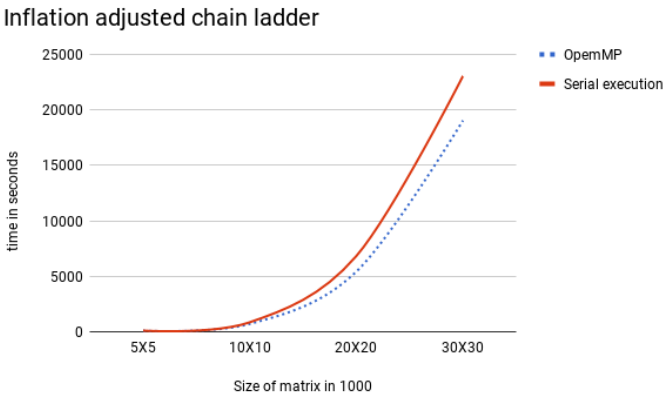


FIG .18.. COMPARISON OF EXECUTION TIME OF INFLATION ADJUSTED CHAIN LADDER METHOD

The method was then implemented on GPUs using the CUDA programming language. Since the method uses matrices for storing the data and most of the operations are on matrices, it is quite evident that GPU architecture suits best for this method and our results prove this fact.

For finding the effect of inflation on the input triangle, each element of incremental triangle is multiplied with the inflation factor recursively. This part of code is taking most of execution time. Since the effect of inflation on each of the element can be calculated independently of others, each thread is given the work to compute the inflation effect on one element each. The figure below summarizes the work allocation of threads. Each thread element works on just one element of the input data.

FIG .17. WORK ALLOCATION OF THREADS

Since GPUs work well with vectors, the input was taken in

form of single dimensional vector. By dividing work among threads we can achieve speed up, in this method each thread was given the work to compute the effect of inflation on that input element. As the data size increased the number of threads were also increased accordingly. The below figures show the time taken by GPUs compared to serial and OpenMP and the improvement achieved by parallelizing the effect of inflation module.

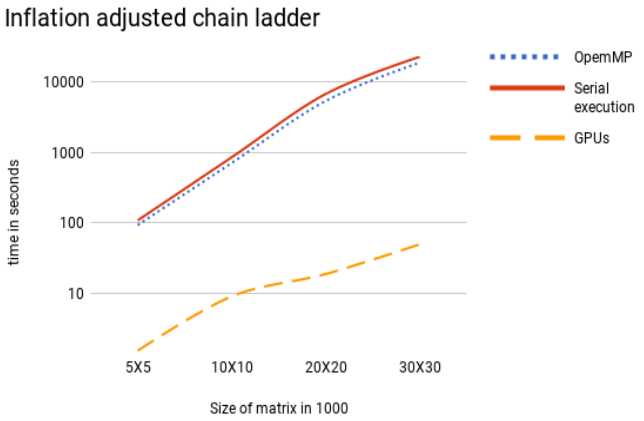


FIG .18. COMPARISON OF EXECUTION TIMES OF INFLATION ADJUSTED CHAIN LADDER METHOD

GPUs proved to be very successful for implementation of Inflation Adjusted CLM, as the data size increased the speed up also increased exponentially.

We achieved a maximum speed up of around 470X.

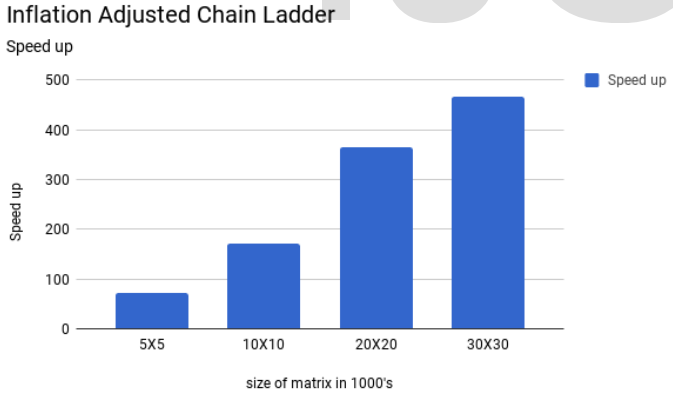


FIG .19. SPEED UP ACHIEVED USING INFLATION ADJUSTED CHAIN LADDER METHOD

Similarly the other modules were parallelized. The dis-accumulation module is reverse of the accumulation module. The work allocation of threads was similar to the ones described in section 3.2. We finally achieved max speed up of 840X.

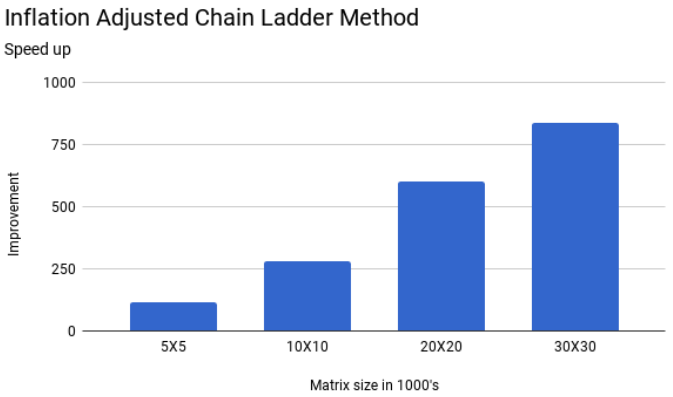


FIG .20. FULL SPEED UP ACHIEVED USING INFLATION ADJUSTED CHAIN LADDER METHOD

2.4 BORNHUETTER-FERGUSON METHOD:

Similar to other methods first the serial method was implemented using C language. As before, as the size increased the time taken to run the method also increased exponentially. The input to this method is also a matrix. First the initial loss ratio is estimated. Then it is adjusted to inflation and then after finding factors, use them to project to ultimate. The below figure shows time taken by the method to run on different sizes of the data. For validation of results, real data was used, except for few rounding off errors the results were accurate.

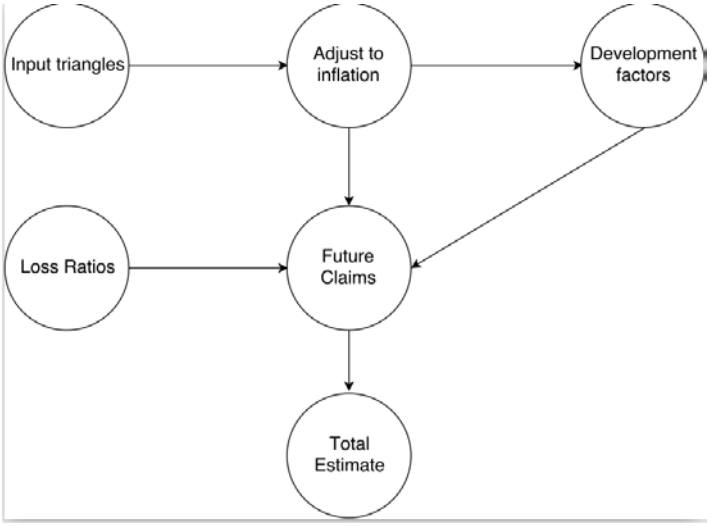


FIG. 21.EXECUTION FLOW OF BORNHUETTER-FERGUSON METHOD

After observing the graph, it was quite evident that this method is taking lot of time to run, in order to parallelize, we need to know the modules which are taking most of the time

first the code was profile using gprof. The figure below shows the percentage of amount of time each module takes to run.
For a $O(N^2)$ size matrix there are $O(N^3)$ floating point operations being
performed.

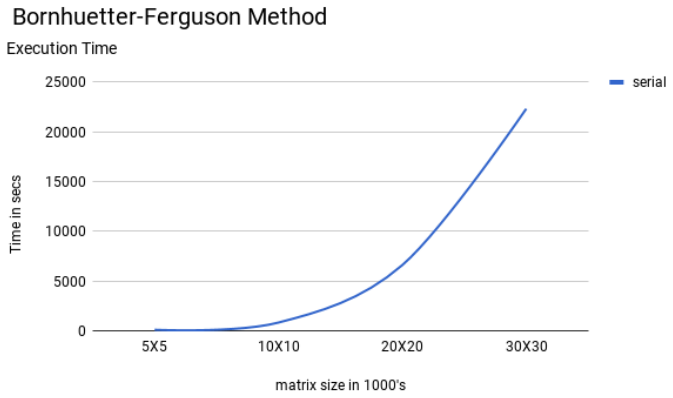
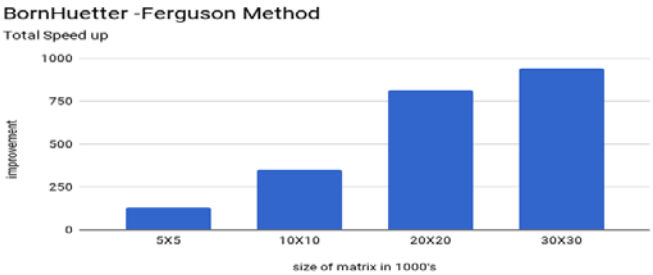


FIG.22. SERIAL EXECUTION TIME OF BORNHUETTER-FERGUSON METHOD.

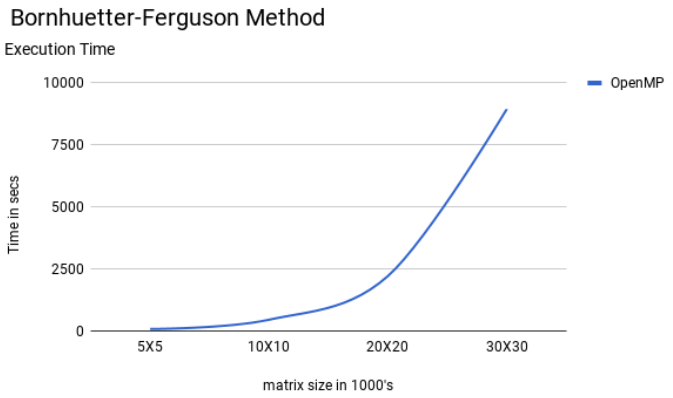
Profiling through gprof tool for different sizes of data has shown that almost 75% percent of the execution time is spent in calculating the effect of inflation on the input triangle. In this module each element is adjusted according to the inflation factor recursively. Finding future claims is taking almost 8% of the execution time.

Module	percentage of time
Adjust to inflation	75.35
Remaining	8.45
Future claims	7.48
Development and cumulative development factors	5.16
Find loss ration	3.56

FIG.23. PROFILING ANALYSIS OF BORNHUETTER-FERGUSON METHOD



Using OpenMP, the inflation adjusted module and finding initial estimates modules were parallelized, using the compiler directives like #pragma omp parallel, #pragma omp parallel for, #pragma omp parallel sections etc.



.

FIG.24. OPENMP EXECUTION TIME OF BORNHUETTER-FERGUSON METHOD

Then using C-CUDA programming language, the method was implemented on GPUs. The inflation adjusted module was parallelized by giving each thread the work to compute the effect of inflation on that element recursively. The speed up achieved was quite significant. Almost 480X speed up was observed which can be seen from the figures below.

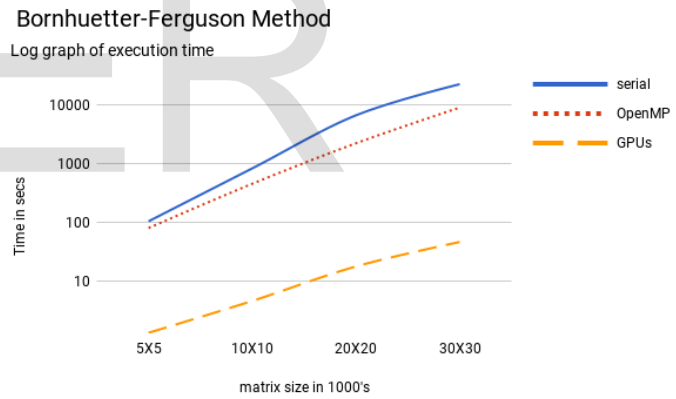


FIG.25. CUDA EXECUTION TIME OF BORNHUETTER-FERGUSON METHOD

After parallelizing the remaining modules, where there was no dependency and using memory management techniques for faster access of the data, like memory coalescing, a maximum speed up 940X was achieved.

FIG .26. SPEED UP ACHIEVED FOR BORNHUETTER-FERGUSON METHOD

2.4 INFERENCE

There are maximum 2^{10} threads available per block and there are 2^{10} blocks in X direction and 2^{10} blocks in Y direction. So total number of threads available for our use is 2^{30} . We have

varied the matrix sizes from 5000*5000 to 30000*30000. So the total number of threads which the implementations require were available. For 30000*30000 matrix almost $5 * 10^9$ threads were used.

As the data size increased the speed up also increased exponentially, the bottleneck for the speed up will be the number of thread available for execution of the method.

For OpenMP implementation we have used 8 threads to decrease the execution time to almost half of the serial time. For this implementation 8 threads seemed to be suiting most, more than 8 because of communication between threads the speed up was degrading.

Maximum efficiency can be gained only when the hardware resources are used utilized to their full extent. In our implementation too we gained peak performance by utilizing all the hardware resources available.

4 CONCLUSIONS AND FUTURE WORK

Using our implementation of the chain ladder and Bornhuetter-Ferguson methods on GPUs we are getting speed up that is increasing as data size is increasing maintaining the accuracy. We have got down the execution time of these methods from 6 hrs to just under one minute.

We have parallelized three of the most widely used methods for calculation of reserves, but there are few more like CAPE COD method, which can be parallelized to be used in real time. We have explored only one domain in field of actuaries, namely reserving other concepts like pricing a policy, catastrophic modeling which deals with natural disasters can be parallelized..

5 REFERENCES

1.) Application of High Performance Computing to Solvency and Profitability Calculations for Life Assurance Contracts Mark Tucker and J. Mark Bull.

2.) Joshi, M.S., Graphical Asian Options, The University Of Melbourne.

3.) Jauvion, G. and Nguyen, T., Parallelized Trinomial Option Pricing Model On GPU With CUDA. www.arbitragis-research.com/cuda-in-computational-finance/ coxross-gpu.pdf

4.) Claims Reserving Working Party Paper Graham Lyons,Will Forster,Paul Kedney,Ryan Warren,Helen Wilkinson

5.) Peter D England and Richard J Verrall, “Stochastic claims reserving in general insurance,” British Actuarial Journal, vol. 8, no. 3, pp. 443–518 2002.

6.) OpenMP Architecture Review Board, OpenMP Application Program Interface, Version 3.1, July 2011, www.openmp.org/mp-documents/OpenMP3.1.pdf.

7.) NVIDIA, NVIDIA CUDA C Programming Guide Version 4.2, 2012.

8.)Jones, A. R., P. J. Copeman, E. R. Gibson, N. J. S. Line, J. A. Lowe, P. Martin, P. N. Matthews, and D. S. Powell. "A change agenda for reserving. Report of the general insurance reserving issues task force (GRIT)." British Actuarial Journal 12, no. 3 (2006): 435-599.