Convert Component Based Serial Code to Parallel to an Automatic Framework

Dr Bhukya Krishna

Professor of CSE Dept, NGIT (Neil Gogte Institute of Technology) Kasavasingaram Uppal, Hyderabad,Telengana-500088

Abstract
With the increase of modular software development to reduce the delivery time, the increase of parallel processing is also increasing. The serial software codes, in the form of component or without the components, cannot leverage the performance of the modern hardware capabilities. Hence majority of the recent software developments are happening considering the parallel processing of the source code. The parallel processing of the source code is not only the responsibility of hardware or GPUs, rather the source codes are also to be written in specific form to take the benefits of parallel processors. Subsequently, not constrained to the improvement in the calculation structure, the utilization of parallel execution of the projects is likewise to be considered. GPUs are ordinarily utilized handling units to accelerate the application execution in diversion improvement. The GPUs can be used to parallelize the application execution to achieve the clock used to the most extreme. The real test is to plan or re-structure the application code from customary sequential programming dialects to the parallel codes, which can take the benefits of GPU centres. Nevertheless, developing the parallel software code components demand a higher development skill from the developer's community, which is difficult to obtain. Also, apart from the newer applications, there are many legacy software components which are also under the demand for conversion in the parallel form. A good number of parallel research attempts are carried out to convert the serial code into a parallel form code, nonetheless, the attempts were manual and needs a very high amount of programming and the application APIs knowledge. Hence, this research attempts to build a framework to automatically convert the serial source code components into parallel source code components to be plugged in to the parallel applications without misplacing the benefits of component-based and the benefits from parallel processing of software codes. The proposed novel framework demonstrates a nearly 97% reduction of code execution time.

Keywords: **Code Conversion, Parallel Execution, GPU, CPU, CUDA, NVIDIA, CUDA Stack**

I. INTRODUCTION

The availability of the multi-core or GPU enabled computing devices are influencing and motivating the software development community to make the applications capable for utilizing the benefits of parallel processing. The complexity of adopting the parallel processing is programming knowledge and to some extend the security flaws of the parallel computing. The survey work by Marry Hall et al. [1] defines the level of complexity for these two challenges. Nevertheless, the advantages of using GPUs is the reduction of time complexity of the software code components and reducing the response time for the application consumers. The work by Amit Barve et al. [2] defines the average response time reduction

by the GPUs or the multi core processors as significantly high and proposes a complete adaptation of parallel processing codes.
The conversion or building of a parallel software or a software code component mainly replies on the conversion of the tasks from serial to parallel without interrupting the dependencies of the software components. The task parallelism for the bigger software components are analysed and approached in the work of Nasser Giacaman et al [3] for modern complex applications. Nonetheless, this work is criticized by the research community for higher cost factors of the tasks. Further, the improvement of this algorithm can be observed in the work of Ying Liu et al. [4].

However, none of the parallel research outcomes have demonstrated the automatic component-based code conversion from serial component to parallel. Thus, this work proposes a novel algorithm for code conversion.

II. Software Code Component

In this section of the work, component-based software code development strategies are analysed [Fig – 1].

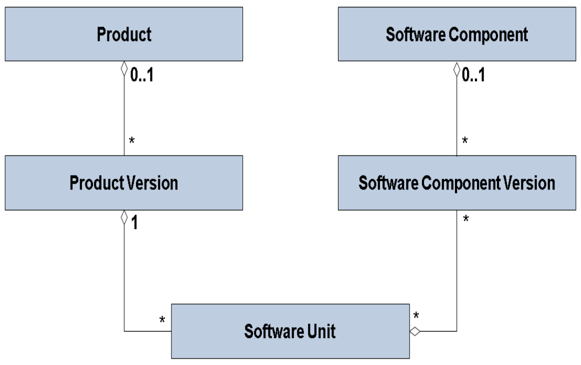

Fig. 1 Component-Based Software Development

Part based programming designing, likewise called as segment-based improvement, is a part of programming building that accentuates the detachment of worries concerning the wide-extending usefulness accessible all through a given programming framework. It is a reuse-based way to deal with characterizing, actualizing and forming inexactly coupled autonomous parts into frameworks. This training plans to achieve a similarly wide-going level of advantages in both the present moment and the long haul for the product itself and for associations that support such programming.

Firstly, we assume any given software application as S and the components in the software applications as s_i.

Thus, the complete software application can be denoted as,

$$S = \sum_{i=0}^{n} s_i \qquad \text{(Eq. 1)}$$

Considering the fact that, the total number of possible components in any software application is n as a positive number more than the cyclometric complexity of the application, Φ.

$$n \geq \Phi(S) \qquad \text{(Eq. 2)}$$

This can also be understood that,

$$n \geq \Phi(\sum_{i=0}^{n} s_i) \qquad \text{(Eq. 3)}$$

The identification of the software components in any software application is based on the independent characteristics based on prerequisite denoted as p, functional characteristics denoted as f and finally the data characteristics of the component denoted as v.

Hence, any software component can be denoted as a combination of all the characteristics and can be represented as

$$S_i = \oiint (p \bullet f \bullet v) \qquad \text{(Eq. 4)}$$

Further, considering another software application S', must match with the same analogy and can be represented as,

$$S' = \sum_{i=0}^{n} s'_i \qquad \text{(Eq. 5)}$$

Also, the components can be analysed as,

$$S'_i = \oiint (p' \bullet f' \bullet v') \qquad \text{(Eq. 6)}$$

Furthermore, the software components must show similar behaviour to be considered for component replacements [Fig – 2].

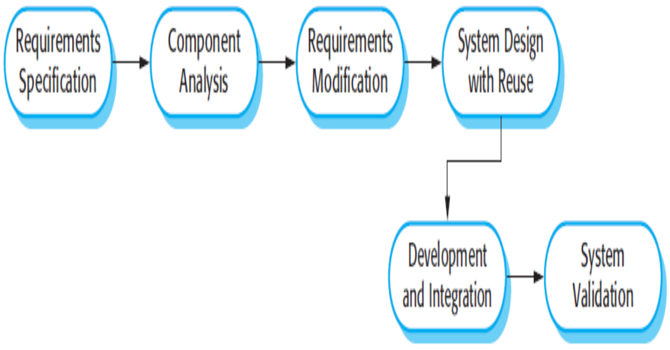
Requirements Specification
Component Analysis
Requirements Modification
System Design with Reuse
Development and Integration
System Validation

Fig. 2 Software Code Component Reusability

To analyse the equivalency of the software code components, a good number of equivalency tests are recommended by various research attempts. The most recommended equivalency test is the black box comparison denoted as ∂ for the components, under which both software components are tested with similar initial conditions and expected to generate similar final state of the application or the data.
This condition can be represented as

$$\partial(s_i) \leftrightarrow \partial(s'_i) \qquad \text{(Eq. 7)}$$

Nonetheless, the above equation can be elaborated as

$$\partial\left[\oiint (p \bullet f \bullet v)\right] \leftrightarrow \partial\left[\oiint (p' \bullet f' \bullet v')\right] \quad \text{(Eq. 8)}$$

This equivalency test can accurately identify the software code components, which are eligible for replacements or reusability.

Henceforth, with the detailed understanding of the component-based software application development, in the next section, this work proposes the functional foundation of code conversion.

III. SOFTWARE CODE PARALLELIZATION

In this section of the work, the execution pattern of the software codes is analysed under two different circumstances as serial code execution and parallel code execution.

The improvements of computing infrastructure have motivated the software development community to cherish the benefits from higher performing and parallel performing computing resources such as GPUs. The GPUs encourage the software code to be executed in parallel to achieve higher order of time complexity reduction.

In order to realize the computational complexity benefits, this work formulates the fundamental principle of code parallelism.

Assuming, any given software application as K and the components in the software applications as k_i, the relation can be formulated as,

$$K = \sum_{i=0}^{n} k_i \qquad \text{(Eq. 9)}$$

The software components must be independent in time frame for execution, which can be identified and denoted as,

$$\oint T(k_i) \neq \oint T(k_j) \qquad \text{(Eq. 10)}$$

Where, two software components can be executed without any time dependencies.

The benefits from the GPU can be obtained if the code components can also be executed in parallel. Assuming the parallel executing threats in the software component as t_i, the relation can be framed as

$$K_i = \sum_{j=1}^{n} t_j \qquad \text{(Eq. 11)}$$

Nonetheless, the computational time for each thread can be assumed as $\Delta\tau$ and the total time can be calculated as

$$T = \sum_{j=1}^{n} t_j \bullet \Delta\tau \qquad \text{(Eq. 12)}$$

Assuming, each thread in the software component can be executed in parallel and are independent from time frame of execution, the total time for execution will be reduced.

Henceforth, it is natural to realize that converting a serial code into parallel execution can leverage the advantages of GPU parallel processing with great reduction of time. Nonetheless, converting a serial code into a parallel code demands high efficiency and greater understanding of software programming. Hence, this work aims to build an automatic framework for code conversion.

Further, in the next section of the work, the parallel research outcomes are analysed.

IV. OUTCOMES FROM THE PARALLEL RESEARCHES

In order to formulate a better problem space and design a sophisticated algorithm for code conversion, in this section of the work, the parallel research outcomes are analyses.

The code conversion is expected to convert all possible types of software component codes ranging from batch processor codes to object-oriented codes. Considering the recent demand for OO software applications, majority of the code conversion frameworks focus on the OO based code conversion. The work of Wouter Joosen et al. [5] defines the fundamental strategies for converting the object-oriented codes in parallel version of the codes. This work is criticized by the other researchers are this algorithm exposes the object spaces into the global allocation spaces and reduces the security of the application.

Also, making the software component independent from the data members can bring a great parallelization in to the software codes. This concept is used by the work of W. J. Staats et al. [6]. Nonetheless, this work is also looked down by the researcher's community for the substantial assumption of available static data members in the serial code block.

In the other hand, converting the dependencies of the software code modules into the inherited feature sets is also a very popular method as showcased in the work of Michael L. Nelson et al. [7]. This work is limited in capability as it is only applicable for the object-oriented applications. The work by Michael L. Nelson et al. [7] is further enhanced by Gregory V. Wilson et al. [8] by adopting the built-in library methods in C++ and also by E. Arjomandi et al. [9] using the default inheritance strategies of modern object-oriented programming languages.

Considering only the OO development strategies for making the code parallelization a good number of research attempts were made as A. Krishnamurthy et al. [10] using the split-C method, P. A. Buhr et al. [11] and Xining Li et al. [12] using the default concurrency control by the OO programming languages, S. Shelly et al. [13] using the inter-process communication between objects, F. Bodin et al. [14] by deploying customization to the runtime.

Also, many of the parallel research outcomes have demonstrated the use of a newer programming language to take the maximum advantage of the GPU as shown by Matthew Fluet et al. [15]. This work is criticised by the developer community as this enforces a newer learning complexity for them. A more realistic solution can be observed in the work by J. L. Sobral et al. [16] as the newly developed programming language shares the similar syntax of popular procedural programming language C.

Hence, with the detailed understanding of the parallel research outcomes, this work formulates the problem in the next section of the work.

V. PROBLEM FORMULATION

In this section of the work, the problem formulation is carried out. The major problem identified from the recent research outcome is to convert the serial code into parallel code, which will uncover the capabilities of the benefits from the GPU architecture.

In order to establish the thought of improvements of time based on automatic conversion of the serial code into parallel codes, this work establishes the following lemma.

Lemma: Conversion of the data members from the software code component into memory block reads will reduce the time complexity of the execution.

Assumptions:

n, Number of data members in code component

m, Maximum number of data members in the memory block
t1, Time to search the data element in the memory block
t2, Time to read the data element from the memory block
t3, Time to read the complete memory block

Proof: Firstly, the total time, T, for the complete code component data member read functionalities can be formulated as,

$$T = [n \bullet t1] + [n \bullet t2] \quad \text{(Eq. 13)}$$

Secondly, the total time, T', for the read functionalities for a single memory block can be formulated as,

$$T' = m \bullet t3 \quad \text{(Eq. 14)}$$

It is natural to realize that, in case of complete memory block read operation, the search operations are not expected to be included.
Further, total number of data members in a single code block is reasonably higher than the total number of data members can be allocated in a single memory block.
This phenomenon can be realized as,

$$n >> m \quad \text{(Eq. 15)}$$

Thus, it is obvious to realize that, the total time for a complete memory block read will be less than the read time complexity for a data member read time from various blocks.

$$T >> T' \quad \text{(Eq. 16)}$$

Hence, it can be concluded that allocating the entire data member set into complete memory blocks can reduce the read and further the complete execution time.
The proposed algorithm applies Lemma – 1 for full memory block allocation for the data items or data members in the code component.

VI. PROPOSED AUTOMATIC CODE CONVERSION ALGORITHM

In this section of the work, the automatic serial to parallel code conversion algorithm is elaborated.

Algorithm: Automatic Serial to Parallel Code Conversion Algorithm **(ASePaC)**

```
Step - 1.    Accept the Source Code Component List
Step - 2.    For each component C[i]
        a. Extract the Data Items as D[k]
        b. Extract the Kernel Data Items as K[l]
        c. Extract the Function flows as L[j]
Step - 3.    For each D[k]
        a. If D[k] is having Dependency on D[k+1]
              i. Convert D[k+1] to BlockDim variable
        b. Convert D[k] to BlockDim variable
Step - 4.    For each K[l]
        a. Convert the declaration using CUDAMalloc()
        b. Allocate memory space as CUDAMemcpy()
Step - 5.    For each L[j]
        a. If L[j] is pragma_kernel_regions
              i. Convert into kernel_call()
        b. Else If L[j] is static method
                i. Convert to __global__ method
        c. Else if L[j] is main[] function
                i. Maintain the syntax
```

In software engineering, the term programmed programming recognizes a kind of PC programming in which some component creates a PC program to enable human developers to compose the code at a higher reflection level.

Henceforth, the next section of this work, the obtained results from the auto-converted code is elaborated.

VII. Results and Discussion

In this section of the work, the results obtained from the proposed automatic code conversion algorithm is analysed. The results obtained from the algorithm is highly time efficient and demonstrates a nearly 80% of time complexity reduction.

The results are analysed in few sub-sections as initial code set analysis, conversion time and space complexity analysis, time complexity comparison during serial and parallel execution.

A. *Initial Code Set Analysis*

The proposed algorithm is tested on many legacy codes for conversion and the few of the outcomes are listed here [Table – 1].

TABLE I Initial Code Set Analysis

Code Set Name	Number of Line in Serial Code format
Binary Search Based Implementation	69
KnapSack Based Implementation	72
Vector Sum Based Implementation	47

The initial code sets are also visualized graphically [Fig – 3].

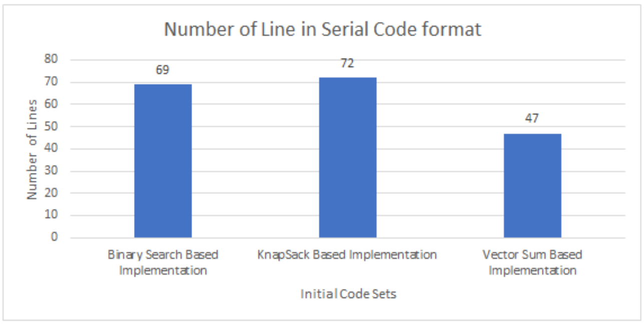

Fig. 3 Initial Code Set Analysis

B. *Automatic Code Conversion Analysis*

Secondly, the analysis results during the code conversion is furnished here [Table – 2].

TABLE II Code Automatic Conversion Analysis

Code Set Name	Number of Line in Serial Code format	Number of Line in Parallel Code format	Time to Convert (Sec)

Binary Search Based Implementation	69	91	5
KnapSack Based Implementation	72	102	7
Vector Sum Based Implementation	47	79	5

The results are visualized here [Fig – 4] [Fig – 5].

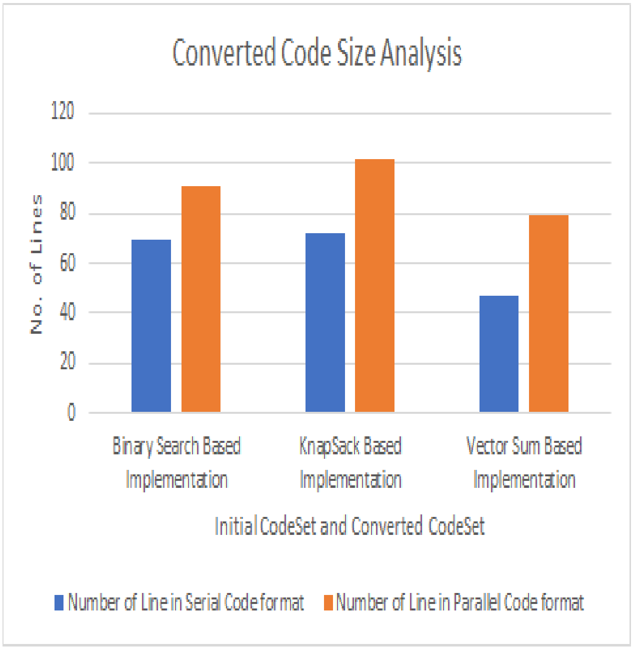

Fig. 4 Code Conversion Analysis – Space Complexity

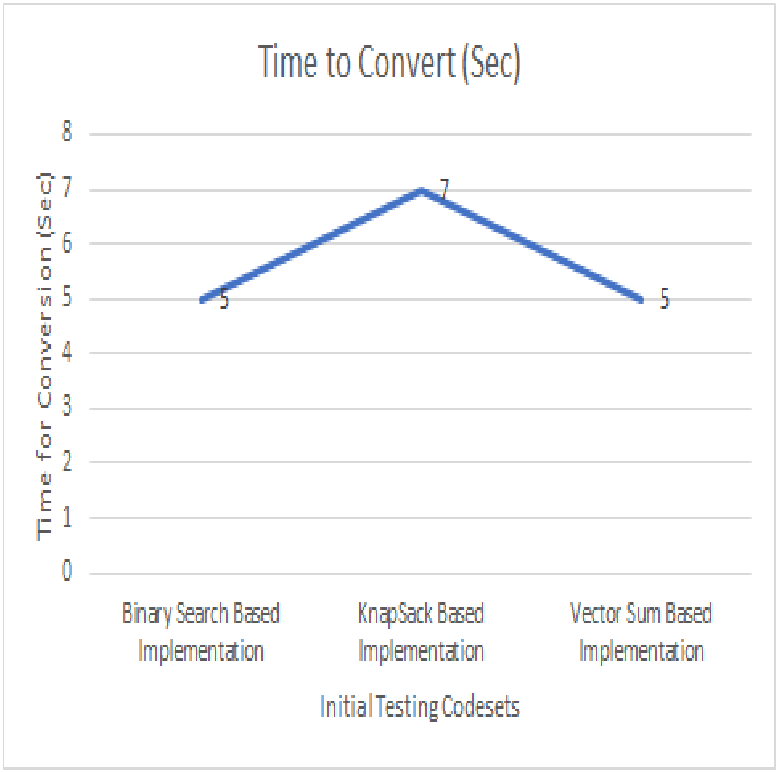

Fig. 5 Code Conversion Analysis – Time Complexity

C. *Binary Search Based Implementation Time Reduction*

Further, the time complexity reduction comparison for binary search-based implementations are analysed here [Table -3]. The analysis is carried out on five different time frames as 0 – 15 sec, 16 to 25 sec, 26 to 35 sec, 36 to 45 sec and finally 46 to 60 sec intervals and the times to execute on CPU or serial code execution and GPU or parallel code execution are noted.

TABLE III: ANALYSIS ON BINARY SEARCH-BASED IMPLEMENTATIONS

Diagnosis Session Duration	**15 Sec**		**25 Sec**		**35 Sec**		**45 Sec**		**60 Sec**	
Serial	39	39	78	78	117	117	156	156	195	195
Parallel	29	29	58	58	87	87	116	116	145	145
Improvement (%)	74.36	74.36	74.36	74.36	74.36	74.36	74.36	74.36	74.36	74.36

Twofold pursuit thinks about the objective incentive to the centre component of the cluster. In the event that they are not equivalent, the half in which the objective can't lie is dispensed with and the pursuit proceeds on the staying half, again taking the centre component to contrast with the objective esteem, and rehashing this until the objective esteem is found.

If the hunt closes with the staying half being unfilled, the objective isn't in the cluster. Despite the fact that the thought is straightforward, actualizing parallel pursuit effectively expects consideration regarding a few nuances about its leave conditions and midpoint count, especially if the qualities in the exhibit are not the majority of the entire numbers in the range.

The results are visualized here [Fig – 6].

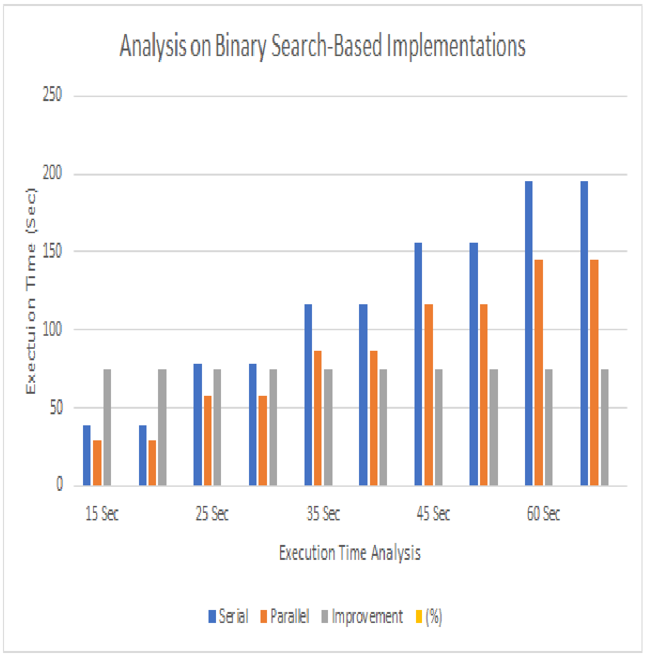

Fig. 6 Analysis on Binary Search-Based Implementations

Also, the simulation real time analytics are furnished here [Fig – 7][Fig – 8].

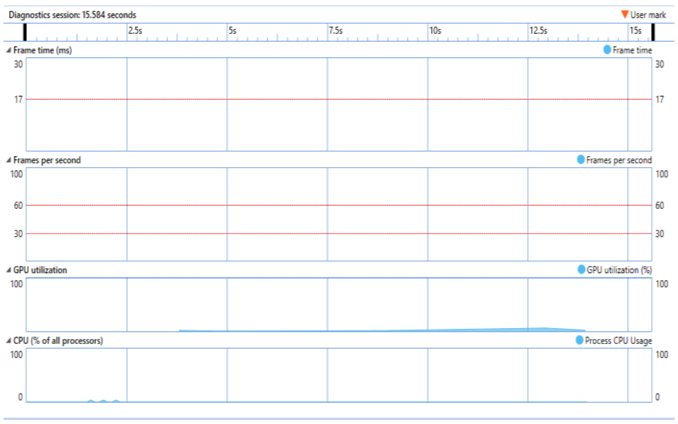

Fig. 7 Serial Simulation Results

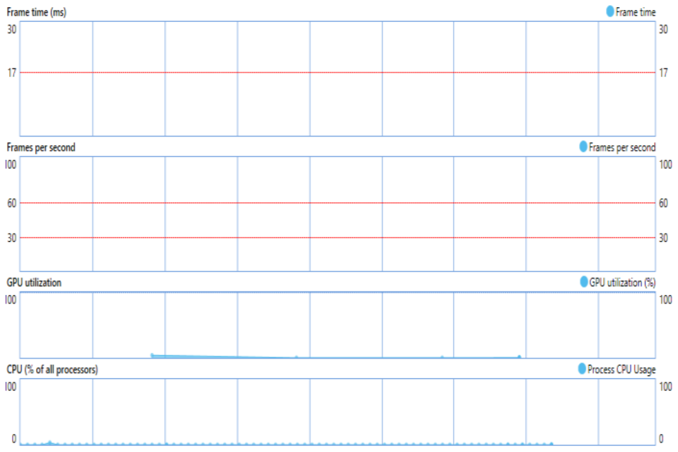

Fig. 8 Parallel Simulation Results

D. *KnapKack Search Based Implementation Time Reduction*

Further, the time complexity reduction comparison for KnapKack-based implementations are analysed here [Table -4]. The analysis is carried out on five different time frames as 0 – 15 sec, 16 to 25 sec, 26 to 35 sec, 36 to 45 sec and finally 46 to 60 sec intervals and the times to execute on CPU or serial code execution and GPU or parallel code execution are noted.

TABLE IV: ANALYSIS ON KNAPKACK-BASED IMPLEMENTATIONS

Diagnosis Session Duration	15 Sec		25 Sec		35 Sec		45 Sec		60 Sec	
Serial	45	45	90	90	135	135	180	180	225	225
Parallel	31	31	62	62	93	93	124	124	155	155
Improvement (%)	68.89	68.89	68.89	68.89	68.89	68.89	68.89	68.89	68.89	68.89

The backpack issue or backpack issue is an issue in combinatorial improvement. Given a lot of things, each with a weight and esteem, decide the quantity of everything to incorporate into a gathering so the all-out weight is not exactly or equivalent to a given cut-off and the absolute esteem is as huge as could reasonably be expected.

It gets its name from the issue looked by somebody who is compelled by a fixed-estimate backpack and should fill it with the most profitable things.

The results are visualized here [Fig – 9].

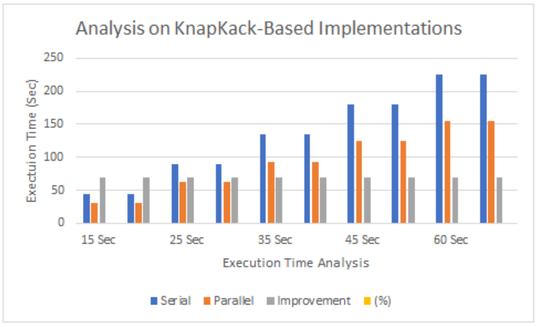

Fig. 9 Analysis on KnapKack-Based Implementations

Also, the simulation real time analytics are furnished here [Fig – 10][Fig – 11].

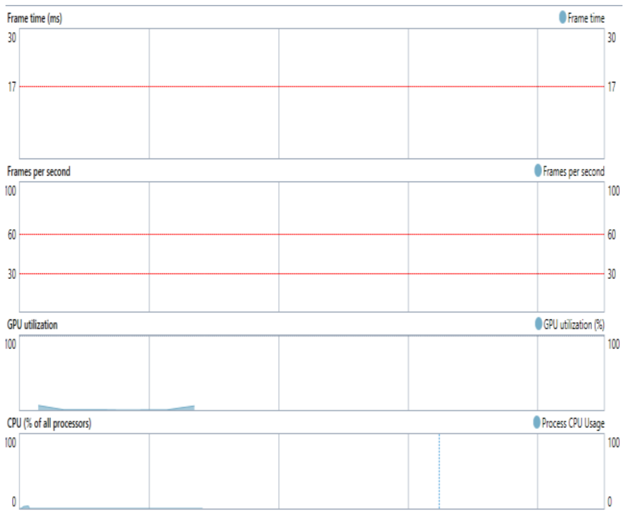

Fig. 10 Serial Simulation Results

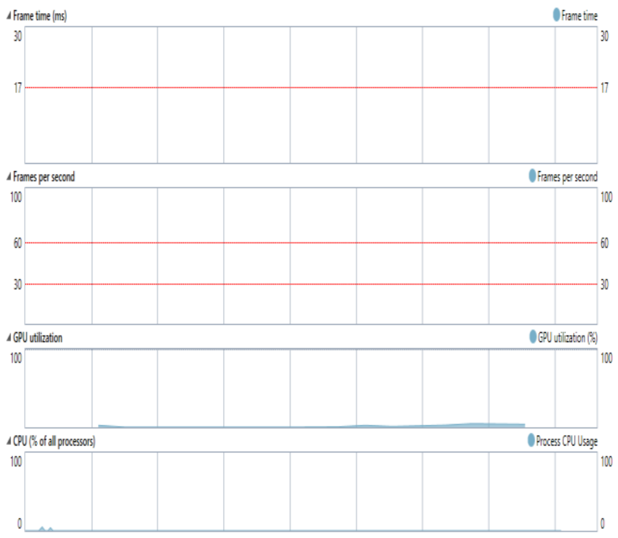

Fig. 11 Parallel Simulation Results

E. *Sum of Vectors Based Implementation Time Reduction*

Further, the time complexity reduction comparison for Sum of Vectors based implementations are analysed here [Table -4]. The analysis is carried out on five different time frames as 0 – 15 sec, 16 to 25 sec, 26 to 35 sec, 36 to 45 sec and finally 46 to 60 sec intervals and the times to execute on CPU or serial code execution and GPU or parallel code execution are noted.

TABLE V: ANALYSIS ON SUM OF VECTORS-BASED IMPLEMENTATIONS

Diagnosis Session Duration	**15 Sec**		**25 Sec**		**35 Sec**		**45 Sec**		**60 Sec**	
Serial	281	281	562	562	843	843	1124	1124	1405	1405
Parallel	274	274	548	548	822	822	1096	1096	1370	1370
Improvement (%)	97.51	97.51	97.51	97.51	97.51	97.51	97.51	97.51	97.51	97.51

A vector is what is expected to "convey" the indicate A point B; the Latin word vector signifies "bearer". It was first utilized by eighteenth-century cosmologists examining planetary unrest around the Sun.
The size of the vector is the separation between the two and the heading alludes to the course of removal from A to B. Numerous arithmetical activities on genuine numbers, for example, expansion, subtraction, augmentation, and nullification have close analogy for vectors, tasks which comply with the natural logarithmic laws.

The results are visualized here [Fig – 12].

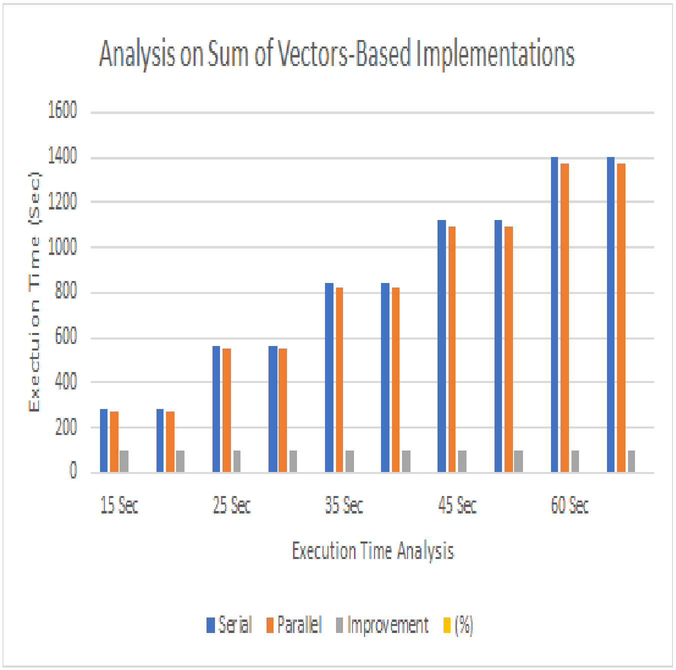

Fig. 12 Analysis on Sum of Vector-Based Implementations

Also, the simulation real time analytics are furnished here [Fig – 13][Fig – 14].

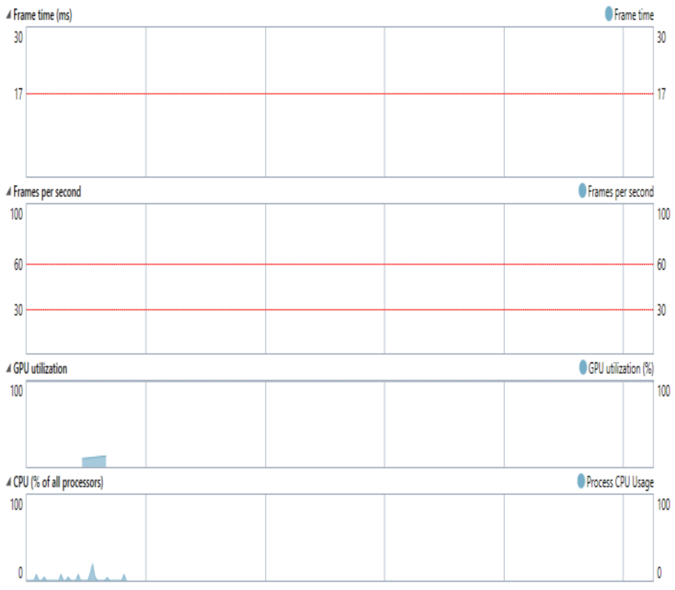

Fig. 13 Serial Simulation Results

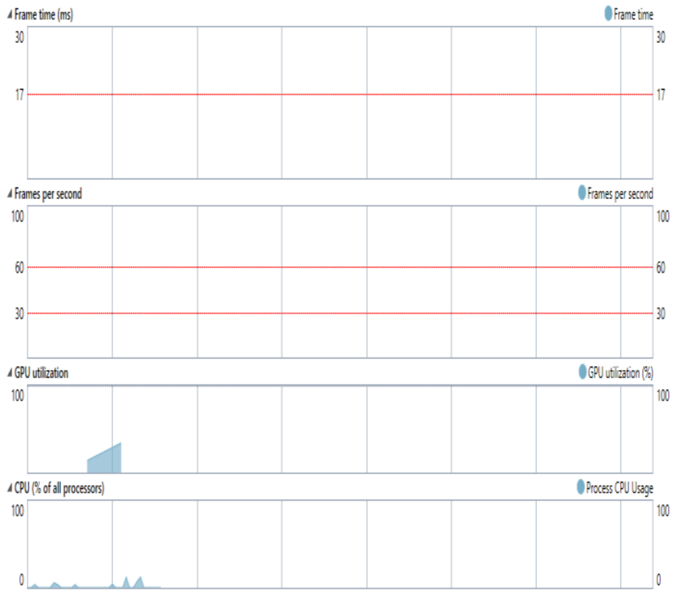

Fig. 14 Parallel Simulation Results

Henceforth, it is natural to realize that, the automatic code conversion results are significantly improving the execution time.

VIII. CONCLUSION

The development strategies for modern software application moved to component-based application development. The component-based application extends greater benefits to the application developers to reuse the components. Also, the software application, due to the high expectations from the code consumers, are expected to satisfy the parallel execution. In the generic situation, the software code components are expected to be available in the code repository. Nevertheless, many of the situations, specifically for the legacy applications, the parallel code component may not be available for reuse, rather the serial code version will be available. Hence, this work, attempts to automatically convert the serial code into parallel codes. The proposed automated algorithm targets the generic data members, kernel-based data members and the functional flow separately and converts in to parallel code by converting the memory management of the data members, applying CUDA architectural benefits to the kernel data members and making the functional flow to a global program register respectively. During the testing phase of this research, the automatically converted codes demonstrates a nearly 97% improvements compared to the serial versions of the same software component codes. Based on the strategies applied and improvements furnished, this work can be considered as one of the prominent milestones for making the code parallelization a better domain to address.

References

1. Marry Hall, David Padua, Keshav Pingali, "Compiler Research: The next 50 years", Communication of the ACM, vol. 52, no. 2, pp. 60-67, February 2009.
2. Amit Barve, Sneha Khomane, Bhagyashree Kulkarni, Shubhangi Katare, Sonali Ghadage, "A Serial to Parallel C++ Code Converter for Multi-Core Machines", Internation Conference on ICT in Business Industry and Government 2016 held on, 18-19 Nov. 2016.
3. Nasser Giacaman, Oliver Sinnen, "Task Parallelism for Object Oriented Programs", The International Symposium on Parallel Architectures Algorithms and Networks held on, 7-9 May 2008.
4. Ying Liu, Fuxiang Gao, "OpenMP Application in TM-based Parallel Program", 2012 International Conference on Computer Science and Electronic Engineering, March 23-25, 2012.
5. Wouter Joosen, Stijn Bijnens, Pierre Verbaeten, "Massively parallel programming using object parallelism", Proceedings of the Conference on Massively Parallel Programming Paradigms, pp. 144-151, September 1993.
6. W. J. Staats, J. Mauney, "Wood-a C++ parallelizer", Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications, pp. 378-384, 1996.
7. Michael L. Nelson, "Concurrency & object-oriented programming", SIGPLAN Not., vol. 26, no. 10, pp. 63-72, October 1991.
8. Gregory V. Wilson, Paul Lu, Forward by Bjarne Stroustrup, MIT Press, July 1996, ISBN 9780262731188.
9. E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Ch. Eigler, G. R. Gao, "ABC++: concurrency by inheritance in C++", IBM Syst. J, vol. 34, pp. 120-137, January 1995.
10. A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, K. Ye1ick, "Parallel programming in Split-C", Proceedings of the 1993 ACM/IEEE conference on Supercomputing (Supercomputing '93). ACM, pp. 262-273, 1993, July 1997.
11. P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger, C. R. Zarnke, "µ C++: Concurrency in the object-oriented language C++", Softw: Pract. Exper., vol. 22, pp. 137-172, 1992.
12. Xining Li, He Huang, Osman Abou-Rabia, Carl K. Chang, Waldemar W. Koczkodaj, "On the Concurrency of C++", Proceedings of the Fifth International Conference on Computing and Information (ICCI '93), pp. 215-219, 1993.
13. S. Shelly, D. L. Stubbs, "ZPCC++: a C++ extension for interprocess communication with objects", Proceedings Nineteenth Annual International Computer Software and Applications Conference (COMPSAC'95) IEEE Comput. Soc. Press Nineteenth Annual International Computer Software and Applications Conference (COMPSAC'95), 9-11 Aug. 1995.

14. F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, B. Mohr, "Implementing a Parallel C++ Runtime System for ScalableParallel Systems", Proc. Supercomputing IEEE Computer Society, pp. 588-597, Nov. 1993.
15. Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, Yingqi Xiao, "Manticore: a heterogeneous parallel language", Proceedings of the workshop on Declarative aspects of multicore programming (DAMP '07), pp. 37-44, 2007, 2007.
16. J. L. Sobral, A. J. Proenca, "ParC++: a simple extension of C++ to parallel systems", Parallel and Distributed Processing 1998. PDP '98. Proceedings of the Sixth Euromicro Workshop on, pp. 453-459, 1998.