International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 12– No. 36, March 2021 – www.ijais.org

A Translation Technique for Parallelizing Sequential Code using a Single Level Model

Hisham M. Alosaimi
Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia

Abdullah M. Algarni
Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia

Fathy E. Eassa
Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia

ABSTRACT
Running the code sequentially can be slower, and then the execution time will increase in case of the code has compute-intensive parts. Unfortunately, the sequential code does not employ the device's resources in ideal shape, because it executes one instruction at a time, which means it can perform only a single thread. To overcome the massive time taking issue while large executions, using a paralleling computing approach is a vital solution. A parallel computing code reduces the execution time by executing multiple tasks at the same time. Most researchers and programmers face some difficulties to run their sequential code as parallel due to a lack of knowledge about parallel programming models and the dependency analysis on their codes. Therefore, auto parallelization tools can be helpful to solve this issue. In this study, we have introduced a novel automatic serial to parallel code translation technique that takes serial code written in C++ as an input and generates its parallel code automatically. To validate the objectives of the current study, we compare the results of our proposed method with existing methods. Consequently, the proposed AP4OpenACC tool outperformed the other existing method mentioned in comparative analysis.

General Terms
Parallel Computing, High Processing Computing

Keywords
OpenACC, GPU, ANTLR, Automatic Translation

1. INTRODUCTION
In the recent past, scientific and business intensive tasks performed on traditional desktop computer or workstation generally consists of a single processing unit (Central Processing Unit) that perform a single task at a time [1]. This creates a bottleneck in fast processing of task as it doesn't clock faster, causes more waste of Central Processing Unit (CPU) cores, very little work gets done, and overall computation become very slow and systematic [2]. However, the performance of a single computer can be depending on the clock speed, memory latency, floating-point unit, the bandwidth of the memory, and Input/Output (IO) to computer storage. Scientific modeling and simulation drive the necessity for excessive computing power. Single-core processors cannot be made to have enough resources for the simulation needed. Producing processors with a faster clock speed is an arduous task due to cost and power limitations. Also, putting huge memory on a single core processor is expensive. This promotes a drastic need for parallel computing that break-down the work between numerous connected systems and have enough computation power to process the extensive task simultaneously.

Parallel Computing and High Processing Computing (HPC) are intimately related. In parallel computing, a huge problem is divided into smaller chunks that could interpret simultaneously to improve the overall performance of HPC systems. The essential idea behind the HPC can be explained in a brief example that is a single-core computer takes 100 hours to finish a task whereas it could finish in 1 hour by using 100 computers at the same time. In short, employing all the resources together at the same time as one device is more beneficial in terms of performance than a single computer.

High Performance Computing (HPC) leveraging distributed computing resources to solve complex problems large dataset usually terabyte to petabytes to zettabytes of data results in minutes to hours instead of days or week. Many scientists and researchers process tons of raw data to make a prediction or create simulations, giving a proficient advantage to enterprises by helping them to be more efficient and quickly discover new insights that drive revenue. HPC provides excellent simulation environments, and it helps applications under development to transact with marketing delivery challenges by providing the ability to accelerate or dispose of prototyping and testing phases. And also, for the decision-making, enhancing the quality, and predicting the overall performance and failure rate of the product [3]. For example, the everyday financial industry faces new regulations, security risks, and electronic payments. These organizations use HPC to complete financial transactions within a second, react quickly to market movements, and use the algorithm to detect credit card fraud.

High Processing Computing (HPC) systems utilize supercomputers and parallel computing methods to perform intensive tasks as the HPC system embodies a group of CPUs where each processor contains multicores besides its local memory to execute a variety of complex tasks and software applications. Supercomputers are a powerful machine that uses thousands of processors to tackle massive problems. For scientific and technical programs use Floating Points Operations per Second (Flops) [4],[5]. Recent supercomputers measured in Petaflops i.e., kilo 103, Mega 106, Giga 109, Tera 1012, Peta 1015. In parallel computing, higher performance requires more processing cores.

Compilation of the sequential code is one of the important topics for programmers and in the computer field in general. Besides computers are using CPU to do the computation job. Sequential code is the code that is executed using a single thread in the CPU with a particular procedure, so only execute

one instruction at a time. However, some of the sequential code contained lots of computations, and it needs huge computing power, so the code becomes more and more complex, for that reason CPUs were taking much time to compile the code in a single thread. Compiling sequential code with lots of computation will increase the execution time. As a result, companies, and researchers started to think about speeding up the compilation of the code and reduce the execution time. One of these techniques is called parallelization of the code. Although, the main concern to run code computationally is efficiency. Many different forms of parallel hardware are a multi-core processor, symmetric multiprocessor, graphic processing unit, field-programming gate array, and computer cluster.

Parallelization is the procedure of converting serial or legacy code to parallel [5]. Moreover, one way for doing that is using multi-core processors, and another way is using Heterogeneous System Architecture (HSA). HSA is a computer that combines CPU and Graphics Processing Unit (GPU) on the same device. Many parallel programming models were developed to deal with HAS such as Compute Unified Device Architecture (CUDA), and Open Accelerators (OpenACC). Essentially, the main benefit of them is targeting GPU if the possibility is found to accelerate the code [6]. GPU is a basic unit for parallel processing to accomplish high performance computing.

It may be a big burden for some scientists and programmers to learn in-depth about programming languages, and about parallel programming models. Also, they need a piece of prior knowledge about dependency between the regions in the code. Correspondingly, for bypassing this issue many types of research have proposed auto parallelizer tools and compilers that can automatically parallelize the code without any interference from the programmer.

C++ is an object-oriented programming language that is popular among scholars, and it is used in a variety of different applications [7], but few automatic tools convert serial C++ to parallel code [8]. Indeed, most of the existing tools are targeting multi-core CPU architecture. In this paper, we will create a translation technique for translating serial C++ code to parallel code to improve the performance and reduce the execution time by making some code analysis to locate which part can be parallelized through the directives of OpenACC.

Moving towards the goals of attaining massive performance, the main objective of the current study is to deal with the essential issues of running the code sequentially that can be slow the task execution time and processing speed. Hence, the execution time will escalate if the intensive part of the computer code is processed by multiple cores simultaneously supporting multithreading. Tragically, the sequential code has been unable to utilize the systems resources ultimately since line by line execution of the instruction at once. This means it can perform only a single thread. To overcome the ineptness in the execution of sequential code, the parallelization technique could be considered a promising solution. Accordingly, Code parallelization can minimize performance time by executing numerous instructions concurrently. Many computer programmers and researchers may deal with some complications to execute their sequential code due to the absence of an understanding regarding parallel programming models and the dependency analysis on their codes. Hence, parallelization tools can be helpful to solve this difficulty.

In this paper, we proposed a translation technique that reduces the execution time of the system whereas accomplishing enormous performance efficiently. The proposed translation technique helps to achieve auto parallelism by taking the input of serial code written in C++ and produces its parallel code automatically. It enhances the performance and reduces the execution time of the system by making a few code analysis to fix the portions which can be parallelized within the section of OpenACC directives as OpenACC implementation require minor effort and more importantly no modification of our existing CPU implementation. Hence, we get the measure of two fundamental HPC performance metrics including execution time and speed up to test the behavior of our proposed solution. Consequentially, it outperformed serial code and well-known auto parallel tool Cetus on larger dataset computations.

The rest of the paper is structured as follows. In section 2, we have discussed the background and the previous literature regarding our work. In section 3, the methodology has been described in detail with the architecture, and the algorithm of our proposed translation technique. In section 4, we have discussed the achieved experimental results in detail along with the experimental platform and the measuring factors considered for evaluating the proposed technique. We showed the discussion in section 5. Finally, the conclusion follows in section 6.

2. LITERATURE REVIEW

We divided this section into two parts. In the background part, we will be mentioning the history of a single processor and parallel computing. Further, we will see the distinction between CPU and GPU. As well as seeing the significant phases that automatic tools should put into consideration. Finally, parallel programming models will be discussed. In the related work part, we are looking at some auto parallel tools and their defects.

2.1 Background

In the 20th century, researchers from IBM build some of the first commercial parallel computing [9]. In 1967, Gene Amdahl says, for over a decade, the organization of a single computer has reached its limit and that truly significant advances can be made only by the interconnection of a multiplicity of the computer. At this time, parallel computing was confirmed to niche communities and used in high performance computing [10].

At the beginning of the 21st century, processor frequency hit the power wall. Processor vendors decided to provide multiple CPU cores on the same processor chips, each capable of executing separate instructions steams [11]. The common theme behind parallel computing was to provide computational power serial computing cannot do so. Parallel computing was present since the early days of computing. Actually, parallel computing is much harder than serial programming. Separating serial computation into parallel sub computations can be challenging or even impossible. Guaranteeing program correctness is more difficult, because of the new types of error. Speedup and fast computation are the only reason why we bother paying for this complexity. Furthermore, the parallel programs use parallel hardware to make the computation execution faster [12].

CPU is consisting of few numbers of cores, however, GPU has hundreds of cores [10]. Running a serial code is better to be on CPU [6] because the clock rate in each core is very high unlike the clock rate in GPU. Ideally, using GPU in the

compute-intensive part of the code make it appropriate than in CPU [11]. Combining CPU and GPU has remarkable benefits on the performance and cost-effective [12], [13]. Indeed, the GPU is used as a collaborative processor with the CPU when needed [14]. In [15] authors were testing the performance between CPU, GPU, and FPGA with different types of benchmarks. Consequently, they found that in execution time GPU has performed well and surpass other platforms. Although the GPU hardware performance growing faster than CPU, and that because of the semiconductor's ability and manufacturing technology [16].

To build an automatic tool, some phrases should be put into consideration. First, receive the source code file as an input, then identify the regions that can be parallelized followed by checking for the dependency in those regions. Finally, adding the parallel constructs to have the source code plus the parallel programming model [5], [8], [17].

Identifying the places in the code that can be parallelized is not an easy task. However, it is considered extremely essential for writing parallel code [18]. Parsing the input file is one of the techniques used to achieve the goal that identifying the potential regions that could be parallelized in the code [19]. The most important regions in any code that need to be run in parallel are loops because it takes much time in execution [5].

Dependency analysis is one of the important steps in any automatic parallelization tool. An independent segment of the code can be parallelized easily. In contrast, having a dependent section in the code is better to be executed in the CPU. Fortunately, this segment can be parallelized if there is a way to remove the dependency without affecting the program movement [5], [8]. Dependency between the statements inside loops can be categorized into two types, loop independent dependency which is divided into four types [5], [17], [20], and loop-carried dependency [5]. Understanding the type of dependency between the statements makes the parallelization process easier.

In 1992, OpenGL launched that was designed for fixed-function SGI hardware beyond consumer graphic hardware capabilities [13]. There are roughly five different generations of Graphic Library (GL) i.e., OpenGL 1.x (fixed function), OpenGL 2.x (early programmable), OpenGL 3.x/4.x (modern programmable, core profile and deprecation), OpenGL ES 1.x (mobile fixed function) and OpenGL ES 2.x (mobile programmable).

MPI is a de facto dating back in 1994, and it is introduced to write code that executes in parallel. It has language binding for FORTRAN, C, and C++ in the beginning [16]. Later on, C++ programming language binding was removed in MPI v3.0. MPI is one of many ways of coding program parallel, enable computer nodes to efficiently pass massage to one another [15]. MPI is a library of functions or subroutines calls. Memory in MPI is assumed to be distributed and not shared [21]. This means we cannot access data in a UE without the UE sending it as a message back. Here, UE is MPI processes.

OpenMP is an open-source API for writing multithreading applications, focusing on shared memory parallelism [17]. For the first time, OpenMP ARB releases OpenMP for the Fortran language in 1997. In the next year, it gave OpenMP for C/C++. OpenMP programming API contains a set of compiler directives e.g., #pragma omp parallel, runtime library routine i.e., omp_get_num_threads() and environment variable e.g., OMP_NUM_THREADS. OpenMP is also called a shared memory model as it is used to create multiple threads. Each process starts with one main thread. This thread is called a master thread on OpenMP [22]. Collectively known as the Fork-Join Model, we create multiple threads along with this master thread. These extra threads other than master threads are known as slave threads. Although, OpenMP [20] greatly simplifies writing multi-threading (MT) programs in FORTRAN, C, and C++ programming languages. Hence, it has been standardized for the last 20 years of SMP practice. The main advantage while parallel execution of the code results in standardization and portability.

With the advancement in GPU computing technologies [23], CUDA was introduced by NVIDIA in 2007 that enable new GPU-based technology for parallel execution of code. CUDA is the most popular GPU framework where programmers created graphical software and that code is broken down into a series of instructions that the central processing unit of the computer could carry out. CPU is the main chip of the system responsible for telling all the other components what to do by giving them a set of instructions processed sequentially. As the program gets more complex, GPU is capable of massive parallelism consisting of smaller more efficient cores designed for handling multiple tasks simultaneously. CUDA [24] provides an extension with standard C code with its programming models. CUDA provides a parallel computing platform supporting shared memory which enables the communication between threads and synchronization that decide which thread executes first and sequence of further threads so that the system performs efficiently. The CPU part consists of CUDA libraries, CUDA runtime, and CUDA driver running the sequential instructions from code written by the programmer. While GPU supports multi-threading by processing multiple core/transistors simultaneously. It executes the intensive part of code and kernel launched by the CPU.

OpenCL was proposed by Apple and its specification was maintained by the Khronos OpenCL Working Group in 2014. With OpenCL [25] we can leverage CPUs, GPUs, and other processors such as Cell/B.E. processors and DSPs to accelerate parallel computing. Write accelerated portable code across different devices and architecture. Moreover, OpenCL gets dramatic speedups for computationally intensive applications. With AMD OpenCL, we can leverage AMD's CPUs and AMD's GPUs to accelerate parallel computation.

To minimize the number of lines of code OpenACC [26] was introduced in 2013 that has an abstract accelerator model. OpenACC is a specification for high-level compiler directives for expressing parallelism for the accelerator [27]. OpenACC can be used with FORTRAN, C, and C++ to utilize the GPU in heterogeneous architecture systems as an accelerator of some regions in the code like compute-intensive regions [28]. From figure 1, accelerated computing includes two different processors in any device. First CPU that is designed to run serial tasks very well and GPU accelerator optimized for parallel tasks. By using OpenACC Library, we can transform serial code to parallel automatically, and it becomes easy to write the program with a minimum line of code that supports parallel computation. The main concern of this technology is performance and portability. Moreover, OpenACC supports multiple GPU, there are API calls to select the desired GPU for parallelism. One advantage of OpenACC is that it is unlike CUDA, OpenACC can be portable on a different type of co-processors not only Nvidia GPU [29]. [30] used OpenACC

with two real-world applications, and they compare OpenACC performance to PGI accelerator and OpenCL. Furthermore, the result of the comparison showed that OpenACC can perform well comparing with low-level programming APIs, and OpenACC shows that is it a promising API.

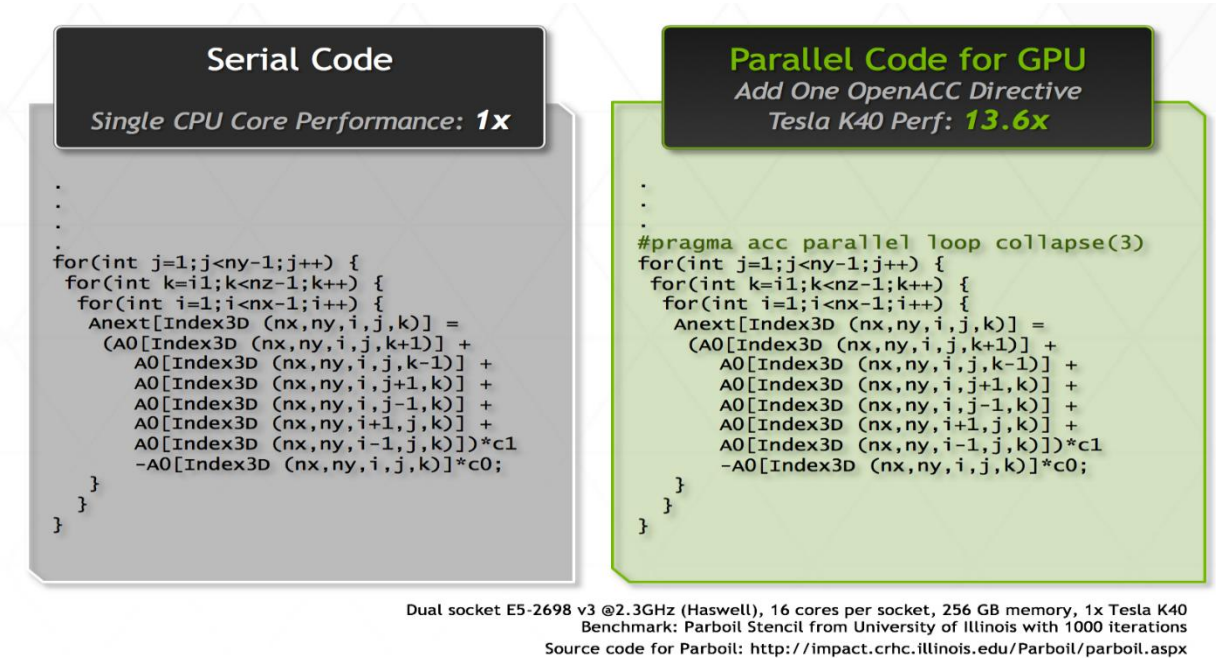

Fig 1: OpenACC with CPU + GPU [31]

2.2 Related work

Presently, parallelization of the code is becoming a hot topic in computer science, much of the scientific research has been written in this field. Supercomputers are also known as High Performance computers (HPC) and normal computer devices that are classified as heterogeneous or homogeneous can use parallelization techniques to run the code. Making the code parallel is vital on supercomputers, so there is no benefit in running your code sequentially on them. However, converting the code to parallel is one of the proper solutions to reduce the execution time by executing multiple series of instructions simultaneously either on heterogeneous or homogeneous system architecture.

Due to a lack of programming knowledge and no CS background, some researchers, and programmers unable to execute their sequential code as parallel [23]. To fill this gap, several works introduce automatic tools to convert sequential programs to parallel. [8], [17], [20], [32] are tools that targeting multi-core CPU by using OpenMP API. In [20], the tool takes a C serial code as an input and breaks it down to tasks known as the coarse-grained task to convert it to parallel by adding OpenMP directive. [32] they are targeting C code as input source code and parallelize it using OpenMP directive but they only targeting for loops. The author of [17] proposed an automatic tool that receives C code and inserts OpenMP directive to it under one primary condition that the code should not have any type of dependency in it. [8] it targets C++ code, and it is also targeting multi-core systems. They claimed that locating the parallel region was by using indicators for the beginning and the end of the potential areas then they took those areas to intermediate file for further analysis. The author of [33] purposed two automatic translation that receives serial java code and converts it to parallel to work on cloud one for Hadoop and the other for spark, it is targeting for loops by using indicators that inserted by the user as comments to locate the beginning and the end of the targeted region.

3. METHODOLOGY

The proposed AP4OpenACC is aimed primarily at two objectives; the first one is to overcome the slowness in the sequential execution of code using the parallelization technique as the program may have compute-intensive components that require a lot of computation with increased execution time. Thus, parallelization of code can enhance performance by executing more than one instruction at a time. The second objective is to provide a roadmap for running their sequential code as a parallel for research and scientific purposes. Earlier, programmers and researchers facing serious problems due to a lack of knowledge about parallel programming models and the dependency analysis on their codes.

Here, we have introduced a novel translation technique for an environment that will be adopted for auto parallelization of sequential code to parallel code. The environment has four main components: parser, identify parallel regions, dependency analyzer, and a code generator [34]. This translation technique encompasses for translating sequential code into parallel code executed in single-level programming models such as OpenACC. It takes serial code as input and generates its parallel code automatically. The proposed translation technique enhances the performance and reduces the execution time by making some code analysis to locate the segments that can be parallelized throughout the OpenACC directives [35], [36].

Based on research objectives, we have provided a roadmap toward building an auto-translation from serial code to parallel code as shown in figure 2, which starts with serial code as an input written in C++ that must be reviewed by the developer to ensure the syntax. Followed by parsing the source code using Another Tool for Language Recognition (ANTLR) to generate the parse tree that will help in the next

step which is identifying the potential regions that could be parallelized. The following step is taking the statements inside each parallel region to check the dependency. After this, the code generator will be called to regenerate the parallel computing code after receiving the proper flags from the dependency analyzer. This will save the code to a directive file. If the checking of dependency in a certain region showed that there is no dependency, then OpenACC pragmas will be added. Otherwise, a comment will be inserted above the region to notify the developer there is a dependency. This translation technique for parallelizing sequential code using a single level programming model solves the problem statement of the current study and achieves the research objectives by introducing the auto parallelization technique taking the massive performance and reduced execution time into consideration.

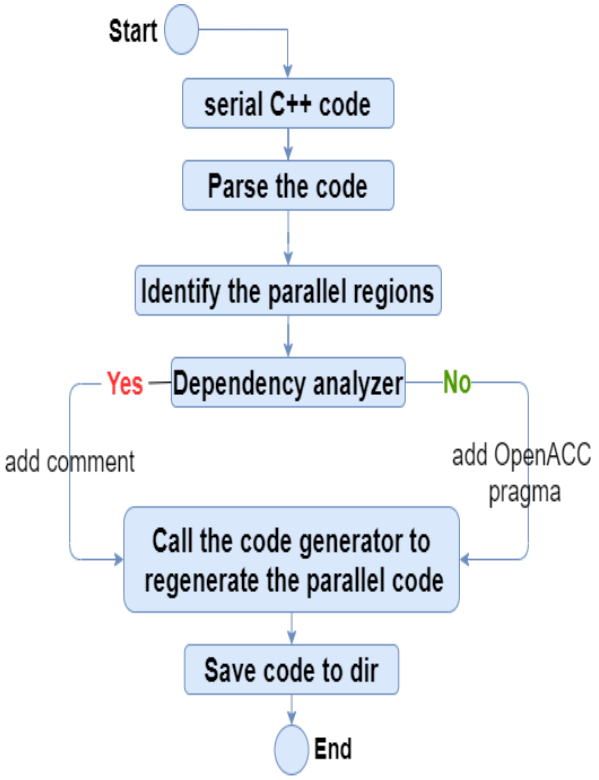

Fig 2: Architecture of Translation Technique

Leading to parallel computing, the most important step is to investigate either the input code is parallelizable or not, and scientifically this mechanism is called dependency analysis. In the current study, we performed the dependency analysis of input serial code by each statement inside loops before parallel code conversion. Figure 3 represents the flow of dependency analysis. According to figure 3, a layman user inputs a serial code written in C++ for conversion in parallelizable code. The whole parallel regions are scanned in sequence one by one each statement/scenario. According to the figure, S1 is considered as statement/scenario 1, whereas S2 is another statement. Each time, the dependency analyzer takes one statement/scenario and compares it with all existing statements from serial input code. It declares as TD (True dependent) once find any dependency in the code, otherwise forward to the next component "Code generator" to convert the no dependency approved code into parallel code.

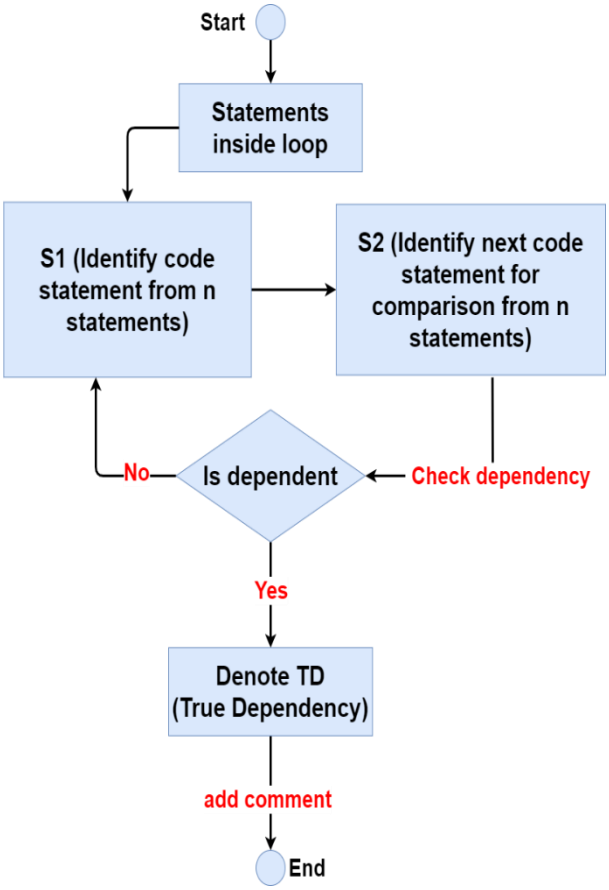

Fig 3:Flow Graph of Dependency Analysis

The fundamental steps for automatic conversion of serial code to parallel code flow have been described with each component and step very clearly in detail. The serial code written by the developer includes C++ programming language [37] [38] and OpenACC [39] as a single level programming model. The main components of the proposed technique include:

- **Parser:** This is the first step where the input will be scanned and read using ANTLR. Initially, the serial code written in C++ must be reviewed by the developer to guarantee that the syntax is correct. Antlr is a spectacular tool that receives the grammar file for the targeted language in our case C++ and the same grammar file can have both lexer and parser rules written together in the same file or it can be separated into different two files. Compiling the grammar file using ANTLR will generate different classes that will be used in our first step and the coming steps. After receiving the serial C++ code from the user, let us assume the name of the source code is code.cpp. We will read and scan the source file at the run time using CharStream provided by ANTLR runtime API [40]. During the scanning of the file, ANLTR does a lexical analysis to recognize the tokens that will be fed to the parser to build the parse tree of the received input file. This process is dynamic, and it depends on the received source file and the parse tree will be changing in every code that the tool received.
- **Identify regions:** Two of the well-known classes that ANTLR generates are BaseListener and BaseVisitor, both have the same functionality, however, in this study, we used BaseVisitor since it is few lines of code and it

returns values from methods when it is needed [41]. ANTLR will generate a method for each parser rule written in the grammar file. The developer can override these methods regarding the need and these methods will be recalled when a region matches the rule during the walking through the parse tree. The regions we are targeting loops i.e., for, while, and do while by overriding the methods of them. Also saving the local and global variables by using the methods. After identifying the parallel regions in the source code, we consider the whole loop block, and we took the statements inside each loop block to be processed in the next step.

- **Dependency analyzer:** In this step, we received the statements that we took after walking through the parse tree in the previous step. In this phase, we have two ways to check for dependency. First, we check the statements inside the loop one by one, once we find any statement has a dependency in it, we terminate, and our test will give a flag to the next phase. Second, if the first check does not return dependency then, in this check we check the dependency between the statements if there is a dependency, we return a flag to the next phase. Otherwise, we return a flag depending on the case of the dependency.
- **Code generator:** The dependency analyzer will produce a series of flags for each identified block in the source code. These flags will determine what will be written above and inside each region that identified whether is a comment when there is a dependency in the code, or the parallel pragma when there is no dependency. Finally, the header of OpenACC will be added at the beginning of the source code. After all the previous steps are finished then we will save the source file in the same directory and will be like this form Acc_filename.cpp, and the result file will be ready to run on a parallel environment.

A quick overview of how C++ and OpenACC in translation technique take part to auto paralyzed [42] the serial code using a single-level programming model. A detailed algorithm for serial code to auto parallelization has been proposed in the tool algorithm.

Algorithm: Serial C++ Source Code to auto parallel code.

1. Scan the source code.
2. Lexing and parsing the source code using ANTLR.
3. Using the parse tree generated by ANTLR to locate the parallel regions.
4. Take the statements inside parallel regions.
5. Implement dependency analysis on statements.
6. Determine whether the statements are parallelizable or not.
7. If the statements are parallelizable, insert the OpenACC directive.
8. Insert comment if statements are not parallelizable.
9. Call the code generator to regenerate the source code with the insertion.
10. Save the output file in the file directive.

As mentioned previously, the proposed technique can scan any serial C++ code and convert it to parallel code by using OpenACC directives. For supporting the parallelism [43] [44], the OpenACC parallel programming model has a parallel loop and kernels pragmas. The parallel loop goal is to inform the compiler which loop is parallelizable, however, kernels use the ability of the compiler to analyze the dependency in the targeted region [45]. The rest of the workflow of this proposed algorithm is elaborated as follows:

- (Line 1): Start with scanning of C++ source code using runtime ANTLR CharStream to read the file as a stream of character to be fed to the next step.
- (Line 2): ANTLR will start lexing and parsing the characters received from scanning the source file and it will build the parse tree according to the rules in the grammar file.
- (Line 3): Overriding loops methods that generated by ANTLR, so when we start reading any C++ code it will find a variable used in the code and it will find loop regions and prepare it for the next step.
- (Line 4): Using the same methods to take the statements inside the loop region to analyze them and check the dependency.
- (Line 5-8): Performing dependency analysis on statements one by one to see ff there is a dependency in the same statement we will terminate and send a flag to the next step to write a comment above the loop region. Otherwise, we move on to the next check which is checking the dependency between the statements and if there is a dependency flag will be sent to the next step and write a comment above the loop region. However, if there is no dependency, we will check if there is a non-deterministic case to send the flag to write the atomic to make the order of the result correct.
- (Line 9-10): The main functionality of the code generator is to take the flags for the dependency analyzer and write above each loop region and save the source code in the directive file. The output file will be under the name Acc_filename.cpp.

The following example demonstrates the C++ source code with different OpenACC directives to calculate and print the result of pi with precision up to 20 decimal points.

```
#include <openacc.h>
#include <iostream>
#include <iomanip>
#include <cstdlib>
int main(int argc, char *argv[])
{
    int nsteps = 100;
    double pi, step, sum = 0.0, x;
    step = (1.0) / nsteps;
#pragma acc parallel loop
    for (int i = 0; i < nsteps; ++i)
{
        x = (i + 0.5) * step;
#pragma acc atomic update
        sum = sum + 1.0 / (1.0 + x * x);
}
 pi = 4.0 * step * sum;
 std::cout<< std::fixed;
 std::cout<<"pi is:"<<std::setprecision(20)<<pi<<"\n";
}
```

4. IMPLEMENTATION AND RESULTS

This section first explains the selected experimental platform and a comprehensive description of the computation metrics measured for evaluating the proposed translation technique. The primary different measures taken for the computation test involve performance metrics that include execution time and speedup of the system. A detailed description of all selected measuring attributes is explained in the following section. Continuing investigation of the single-level programming model, we check the computation between the sequential form of the code, Cetus which uses OpenMP, and the code after the translation with AP4OpenACC. Observing the time taken by the computer to execute the code and print out the result.

4.1 Experimental Platform and Measuring Factors

To evaluate the proposed translation technique, we performed all the experiments on a personal computer. Comes with an Intel Core i7 4720HQ CPU that has 4 cores and 8 threads with a speed of 2.60GHz with turbo speed up to 3.60GHz. It is also shipped with GM107 GPU NVIDIA GeForce GTX 960M generation, and the architecture of the GPU is Maxwell, and it consists of 640 Cuda cores and 4GB GDDR5 memory. The GPU has a single-precision power up to 1388.8 GFLOPS which means it is competent of completing a billion floating-point operations per second. We compile the serial C++ code, OpenMP code that resulted from Cetus, and the code that resulted from AP4OpenACC by using the NVIDIA HPC SDK compiler [46].

In the experiment, we have measured different performance attributes including execution time (Secs) and the speedup (Serial/Parallel) of the system. To evaluate these performance attributes, we selected a benchmark application that calculates the sum of a double number entered by the user and then calculates the result. During the experiments, we entered multiple different random numbers to see the behavior and the result of the performance attribute by increasing the number every time. In our experiment, we compare between serial, Cetus and OpenACC codes. Since Cetus does not support translating C++ code, we write the equivalent C code to be translated by Cetus. Depending on the sum implementation in our suggested technique, we measure the performance of the application using two main metrics including time execution, and speedup.

Concerning time execution, we determine different random numbers starting from 6 numbers and ending to 12 numbers. We used diverse numbers to investigate the behavior of the proposed model and calculate execution time when running the code on a multi-core CPU and running it on GPU. Actually, we can compute the execution time of parallel computation by time execution performance metric which is considered to be a very straightforward evaluation mechanism. Besides, we evaluated the performance metric as speedup where the sum of the number computes using a single number of CPU core to determine how much time does the code takes sequentially. Theoretically, one way of measuring the speedup of the program is by employing the following equation [47].

$$Speed\ up = \frac{Serial\ exe\ time}{Parallel\ exe\ time}$$

4.2 Results

In experiment 1, we choose three random numbers for our measurements by generating 6,7, and 8 random numbers. First, we use a single CPU core and write down the result of the serial computation. Then, we convert C code to parallel

using Cetus. Serial and parallel code provided by Cutes was giving better results than AP4OpenACC because they do not have to do any type of data transfer unlike OpenACC [30]. As a result, most of the time taken by AP4OpenACC was only because of the data transfer between CPU and GPU. The actual values of experiments with execution time showed in table 1.

Table 1. Random numbers with execution time for experiments

Number	Serial	Cetus	AP4OpenACC
778974	0.0091	0.0030	0.26958
1918035	0.0219	0.0073	0.2638
43487668	0.4927	0.1639	0.2637
959733842	10.8817	3.6220	0.3484
6797675395	79.7817	25.6301	0.85993
6797675395	982.5109	315.5568	7.4159
208425664461	2446.2123	786.5757	18.1008

We also have presented the graphical representation of this experiential data for both experiments. Figure 4 demonstrated the execution time in experiment 1. It is noticeable that AP4OpenACC not increasing the time that much during the experiment, and it is almost executed at the same time. However, serial and OpenMP codes showed a clear increase throughout the experiment.

Fig 4: Performance (Execution time) for three random numbers

Using the speedup equation, our first experiment gives no touchable increase in the speedup. Figure 5 demonstrated the calculated speedup in all three numbers. The first two numbers do not show any tangible speedup. However, in the third number, it can be very clear that AP4OpenACC showed a tremendous increase in the speedup, unlike Cetus which showed almost the same speedup without a noticeable increase.

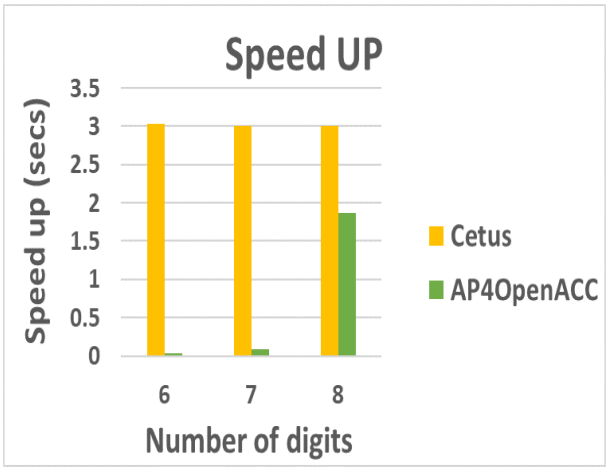

Fig 5: Performance (Speed up) between Cetus and AP4OpenACC

Further, to investigate the behavior of the proposed translation technique, we increase the number in the second experiment i.e., we increase from 6 digits to 9 digits ending up to 12 digits. Eventually, as shown in figure 6, we observed that with the increase in the number of digits, our AP4OpenACC translation tool outperformed other executions with a humongous difference in time execution.

Fig 6: Performance (Execution time) for four big random numbers

By using the speedup equation, we have seen a tremendous increase in the speedup of our proposed AP4OpenACC technique in contrast to Cetus as depicted in figure 7. Eventually, when implementing our proposed technique, it is declared that it can give outstanding performance, especially when increasing the number of digits. As a result, it is observed that our proposed model speedily computes its computation and outperformed all the implementation.

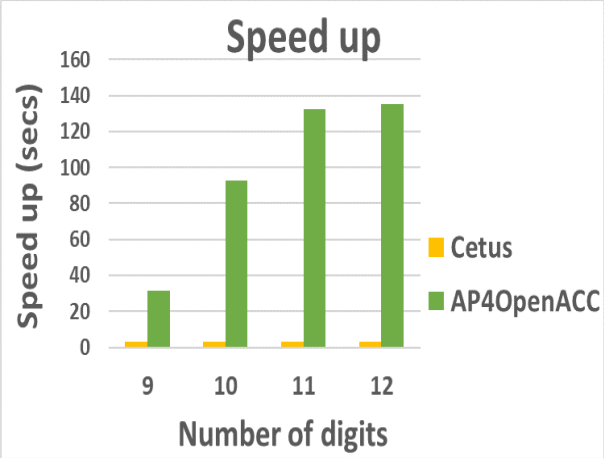

Fig 7: Performance (Speed up) between Cetus and AP4OpenACC

5. DISCUSSION

Currently, parallel computing becomes a dominant means for the interpretation of huge and intensive computation problems. Consequentially, auto-detect of parallelism in applications is an important powerful tool for the development of parallel software by aiding programmers to use FORTRAN or C programming languages that are used in numerical applications. Parallelizing data across multiple nodes at the same time will improve the performance of the HPC systems. Utilizing fewer resources while communicating among different processes will consume less power and enhance performance while a parallel programming model enables a larger task to run on multiple processors at the same time. The first objective of the current study was to enhance system performance by increasing the execution time of the sequential code translated to parallel. Achieving such a level of performance by the end of the decade will require applications to exploit the billion-way parallelism provided by future HPC systems. The second objective of this research was to enhance the auto-translation of the serial code to parallel by using a parallelization technique once the dependency in code is achieved and detected successfully.

The proposed translation tool outperformed as compared with CETUS. According to the main objectives of the current study, first, we have to detect the parallel computing regions in the sequential code. Once it is done, dependency analysis is performed on each region. Then we auto-translate the serial code to parallel. This challenging task has been successfully achieved by our proposed solution. For implementations, we quantified different performance metrics which is execution time and speedup. Underperformance, we use the sum application and run it on different random numbers. For small numbers, AP4OpenACC showed slow computation compared to serial and OpenMP, since the overhead of data transfer was taking lots of the execution time. However, after increasing the number from 6 to 9, AP4OpenACC outperformed other implementations with unmatched results.

6. CONCLUSION AND FUTURE WORK

The emergence of high performance computing requires significant usage of supercomputers to address complex scientific programs and solving complicated computational tasks quickly. Although the sequential execution of code is slower and requires more time for program execution compilation. Most of the researchers are facing issues in working parallel computing systems. Leading to objective, parallelization tools could be a vital solution to address the said issue. In order to attain the said objectives, we proposed a new translation technique that translates any serial C++ code into parallel using OpenACC programming models. The code translation procedure is performed after checking the dependency in the input serial code. Thus, no dependency leads towards the parallel zone where the proposed translation tool converts the sequential program to compute for the parallel computing system. AP4OpenACC supports a single parallelization model which is OpenACC for homogeneous systems that utilize GPU devices for providing massive parallelism. In order to evaluate the proposed translation technique, we implement sum application and results have been compared with the famous auto-translation tool Cetus. Based on experimental consequences, it has been observed that the AP4OpenACC outperformed the existing studies, particularly with huge numbers.

From a future perspective, the researcher can consider our tool for translating sequential code into parallel for any given parallel programming mode such as OpenMP, CUDA, OpenCL, OpenGL, or large cluster systems to achieve massive parallelism.

7. REFERENCES

[1] S. Perarnau, R. Gupta, and P. Beckman, “Argo: An Exascale Operating System and Runtime,” p. 2.

[2] J. Shalf, S. Dosanjh, and J. Morrison, “Exascale Computing Technology Challenges,” in High Performance Computing for Computational Science – VECPAR 2010, Jun. 2010, pp. 1–25, doi: 10.1007/978-3-642-19328-6_1.

[3] “Petascale adaptive computational fluid dynamics - ProQuest.” http://kau.proxy.deepknowledge.io/MuseSessionID=0o10p48z4/MuseProtocol=https/MuseHost=search.proquest.com/MusePath/docview/304985752/20FC96D8B18547A7PQ/1?accountid=43793 (accessed Nov. 20, 2020).

[4] J. J. Dongarra and D. W. Walker, “The quest for petascale computing,” Comput. Sci. Eng., vol. 3, no. 3, pp. 32–39, May 2001, doi: 10.1109/5992.919263.

[5] S. Prema, R. Jehadeesan, B. K. Panigrahi, and S. A. V. Satya Murty, “Dependency analysis and loop transformation characteristics of auto-parallelizers,” in 2015 National Conference on Parallel Computing Technologies (PARCOMPTECH), Bengaluru, India, Feb. 2015, pp. 1–6, doi: 10.1109/PARCOMPTECH.2015.7084524.

[6] A. Tabuchi, M. Nakao, and M. Sato, “A Source-to-Source OpenACC Compiler for CUDA,” in Euro-Par 2013: Parallel Processing Workshops, Aug. 2013, pp. 178–187, doi: 10.1007/978-3-642-54420-0_18.

[7] A. Barve, S. Khomane, B. Kulkarni, S. Ghadage, and S. Katare, “Parallelism in C++ programs targeting objects,” in 2017 International Conference on Advances in Computing, Communication and Control (ICAC3), Dec. 2017, pp. 1–6, doi: 10.1109/ICAC3.2017.8318759.

[8] A. Barve, S. Khomane, B. Kulkarni, S. Katare, and S. Ghadage, “A serial to parallel C++ code converter for multi-core machines,” in 2016 International Conference on ICT in Business Industry Government (ICTBIG),

Nov. 2016, pp. 1–5, doi: 10.1109/ICTBIG.2016.7892700.

[9] E. Strohmaier, J. J. Dongarra, H. W. Meuer, and H. D. Simon, "Recent trends in the marketplace of high performance computing," Parallel Comput., vol. 31, no. 3, pp. 261–273, Mar. 2005, doi: 10.1016/j.parco.2005.02.001.

[10] K. Krewell, "What's the Difference Between a CPU and a GPU?," The Official NVIDIA Blog, Dec. 16, 2009. https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (accessed Mar. 20, 2019).

[11] "Parallel Computing on a Personal Computer | Biomedical Computation Review." http://www.bcr.org/content/parallel-computing-personal-computer (accessed Nov. 25, 2020).

[12] M. Arora, "The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing," undefined, 2012. /paper/The-Architecture-and-Evolution-of-CPU-GPU-Systems-Arora/8a77e1722b37fe3d8f5ac56bb50e548b218c4427 (accessed Nov. 21, 2020).

[13] "Combining GPU data-parallel computing with OpenGL | ACM SIGGRAPH 2013 Courses." http://0o10e49gv.y.https.dl.acm.org.kau.proxy.deepknowledge.io/doi/10.1145/2504435.2504449 (accessed Nov. 21, 2020).

[14] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," Sci. Program., vol. 18, no. 1, pp. 1–33, Jan. 2010, doi: 10.3233/SPR-2009-0296.

[15] C. Cullinan, T. R. Frattesi, and C. Wyant, "Computing Performance Benchmarks among CPU, GPU, and FPGA," undefined, 2012. /paper/Computing-Performance-Benchmarks-among-CPU%2C-GPU%2C-Cullinan-Frattesi/cbecd8cfb5264f8b36dee412c5980e3305c996e6 (accessed Nov. 21, 2020).

[16] S. Mittal and J. S. Vetter, "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency," ACM Comput. Surv., vol. 47, no. 2, p. 19:1-19:23, Aug. 2014, doi: 10.1145/2636342.

[17] N. Singh, "Automatic parallelization using OpenMP API," in 2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES), Oct. 2016, pp. 291–294, doi: 10.1109/SCOPES.2016.7955837.

[18] Y. Qian, "Automatic Parallelization Tools," p. 5, 2012.

[19] A. Athavale et al., "Automatic Sequential to Parallel Code Conversion," p. 7.

[20] M. Mathews and J. P. Abraham, "Implementing Coarse Grained Task Parallelism Using OpenMP," vol. 6, p. 4, 2015.

[21] "Message Passing Interface (MPI)." https://computing.llnl.gov/tutorials/mpi/ (accessed Nov. 27, 2020).

[22] A. Podobas and S. Karlsson, "Towards Unifying OpenMP Under the Task-Parallel Paradigm," in OpenMP: Memory, Devices, and Tasks, Oct. 2016, pp. 116–129, doi: 10.1007/978-3-319-45550-1_9.

[23] J. D. Owens et al., "A Survey of General-Purpose Computation on Graphics Hardware," Comput. Graph. Forum, vol. 26, no. 1, pp. 80–113, Mar. 2007, doi: 10.1111/j.1467-8659.2007.01012.x.

[24] Department of Computer Science, King abdulaziz University Jeddah, Saudi Arabia, M. U. Ashraf, F. Fouz, and F. Alboraei Eassa, "Empirical Analysis of HPC Using Different Programming Models," Int. J. Mod. Educ. Comput. Sci., vol. 8, no. 6, pp. 27–34, Jun. 2016, doi: 10.5815/ijmecs.2016.06.04.

[25] M. Scarpino, "OpenCL in Action: How to Accelerate Graphics and Computations," Dec. 2011, Accessed: Nov. 21, 2020. [Online]. Available: https://hgpu.org/?p=6708.

[26] N. Newsroom, "NVIDIA, Cray, PGI, CAPS Unveil 'OpenACC' Programming Standard for Parallel Computing," NVIDIA Newsroom Newsroom. http://nvidianews.nvidia.com/news/nvidia-cray-pgi-caps-unveil-openacc-programming-standard-for-parallel-computing (accessed Nov. 25, 2020).

[27] "Getting Started with OpenACC," NVIDIA Developer Blog, Jul. 14, 2015. https://developer.nvidia.com/blog/getting-started-openacc/ (accessed Nov. 25, 2020).

[28] "OpenACC Tutorial - Adding directives - CC Doc." https://docs.computecanada.ca/wiki/OpenACC_Tutorial_-_Adding_directives (accessed Nov. 21, 2020).

[29] "OpenACC: More Science Less Programming," NVIDIA Developer, Jan. 13, 2012. https://developer.nvidia.com/openacc (accessed Nov. 21, 2020).

[30] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC — First Experiences with Real-World Applications," in Euro-Par 2012 Parallel Processing, Aug. 2012, pp. 859–870, doi: 10.1007/978-3-642-32820-6_85.

[31] "OpenACC Directives," NVIDIA Developer, Mar. 02, 2016. https://developer.nvidia.com/openacc/overview (accessed Mar. 26, 2021).

[32] A. G. Bhat, M. N. Babu, and A. M. R, "Towards automatic parallelization of 'for' loops," in 2015 IEEE International Advance Computing Conference (IACC), Jun. 2015, pp. 136–142, doi: 10.1109/IADCC.2015.7154686.

[33] B. Li, "Manual and Automatic Translation From Sequential to Parallel Programming On Cloud Systems," Comput. Sci. Diss., Apr. 2018, [Online]. Available: https://scholarworks.gsu.edu/cs_diss/135.

[34] A. Alghamdi and F. Eassa, "Parallel Hybrid Testing Tool for Applications Developed by Using MPI + OpenACC Dual-Programming Model," vol. 4, pp. 203–210, Mar. 2019, doi: 10.25046/aj040227.

[35] E. Calore, A. Gabbana, J. Kraus, S. F. Schifano, and R. Tripiccione, "Performance and portability of accelerated lattice Boltzmann applications with OpenACC," Concurr. Comput. Pract. Exp., vol. 28, no. 12, pp. 3485–3502, Aug. 2016, doi: 10.1002/cpe.3862.

[36] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Achieving Portability and Performance through OpenACC," in 2014 First Workshop on Accelerator Programming using Directives, Nov. 2014, pp. 19–26, doi: 10.1109/WACCPD.2014.10.

[37] J. Kraus, M. Schlottke, A. Adinetz, and D. Pleiter, "Accelerating a C++ CFD Code with OpenACC," in 2014 First Workshop on Accelerator Programming using Directives, Nov. 2014, pp. 47–54, doi: 10.1109/WACCPD.2014.11.

[38] "[PDF] DawnCC: a Source-to-Source Automatic Parallelizer of C and C++ Programs | Semantic Scholar." https://www.semanticscholar.org/paper/DawnCC%3A-a-Source-to-Source-Automatic-Parallelizer-C-Guimar%C3%A3es-Mendonca/ac4ee9490909aa0161e8278596e85ecd6ece4148 (accessed Dec. 01, 2020).

[39] M. U. Ashraf, F. A. Eassa, and A. A. Albeshri, "Massive Parallel Computational Model for Heterogeneous Exascale Computing System," in 2017 9th IEEE-GCC Conference and Exhibition (GCCCE), May 2017, pp. 1–6, doi: 10.1109/IEEEGCC.2017.8448062.

[40] "org.antlr.v4.runtime Class Hierarchy (ANTLR 4 Runtime 4.9 API)." https://www.antlr.org/api/Java/org/antlr/v4/runtime/package-tree.html (accessed Dec. 01, 2020).

[41] "Antlr 4 - Listener vs Visitor." https://jakubdziworski.github.io/java/2016/04/01/antlr_visitor_vs_listener.html (accessed Dec. 01, 2020).

[42] M. Viñas, B. B. Fraguela, D. Andrade, and R. Doallo, "Towards a High Level Approach for the Programming of Heterogeneous Clusters," in 2016 45th International Conference on Parallel Processing Workshops (ICPPW), Aug. 2016, pp. 106–114, doi: 10.1109/ICPPW.2016.30.

[43] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman, OpenACC Parallelization and Optimization of NAS Parallel Benchmarks. 2014.

[44] A. Paudel and S. Puri, "OpenACC Based GPU Parallelization of Plane Sweep Algorithm for Geometric Intersection," in Accelerator Programming Using Directives, Nov. 2018, pp. 114–135, doi: 10.1007/978-3-030-12274-4_6.

[45] "OpenACC Programming and Best Practices Guide," p. 64.

[46] "HPC SDK | NVIDIA," NVIDIA Developer, Mar. 11, 2020. https://developer.nvidia.com/hpc-sdk (accessed Dec. 13, 2020).

[47] J. Zapletal, "Amdahl's and Gustafson's laws," p. 28.