Intelligent Automation & Soft Computing
DOI:10.32604/iasc.2021.019044
Article

AAP4All: An Adaptive Auto Parallelization of Serial Code for HPC Systems

M. Usman Ashraf[1,*], Fathy Alburaei Eassa[2], Leon J. Osterweil[3], Aiiad Ahmad Albeshri[2], Abdullah Algarni[2] and Iqra Ilyas[4]

[1]University of Management and Technology, Lahore (Sialkot), Pakistan
[2]Department of Computer Science, King Abdulaziz University, Saudi Arabia
[3]University of Massachusetts Amherst, USA
[4]Department of Computer Science, GC women University, Sialkot, Pakistan
[*]Corresponding Author: M. Usman Ashraf. Email: usman.ashraf@skt.umt.edu.pk
Received: 29 March 2021; Accepted: 30 April 2021

Abstract: High Performance Computing (HPC) technologies are emphasizing to increase the system performance across many disciplines. The primary challenge in HPC systems is how to achieve massive performance by minimum power consumption. However, the modern HPC systems are configured by adding the powerful and energy efficient multi-cores/many-cores parallel computing devices such as GPUs, MIC, and FPGA etc. Due to increasing the complexity of one chip many-cores/multi-cores systems, only well-balanced and optimized parallel programming technique is the solution to provide substantial increase in performance under power consumption limitations. Conventionally, the researchers face various barriers while parallelizing their serial code because they don't have enough experience to use parallel programming techniques in optimized way. However, to address these obstacles and achieve massive performance under power consumption limitations, we propose an Adaptive and Automatic Parallel programming tool (AAP4All) for both homogeneous and heterogeneous computing systems. A key advantage of proposed tool is an auto recognition of computer system architecture, then translate automatically the input serial C++ code into parallel programming code for that particular detected system. We also evaluate the performance and power consumption while computing the proposed AAP4All model on different computer architectures, and compare the results with existing state-of-the-art parallel programming techniques. Results show that the proposed model increases the system performance substantially by reducing power consumption as well as serial to parallel transformation effort.

Keywords: Automatic parallel computing; GPU; high performance computing; power consumption

1 Introduction

High performance computing (HPC) play a vital role in emerging computing technologies that process complex and large amount of data at very high speed. It has made groundbreaking scientific discoveries and improved the quality of life around the globe [1]. HPC is getting progressively significant across many disciplines such as computational fluid dynamics (CFD) simulations [2], data mining, climate and environmental modelling [3], molecular nanotechnology [4], big data applications [5], neural network algorithms for deep learning [6] and many more real life applications.

The frequent usage of HPC systems in these applications, demanding a tremendous increase in performance day by day. To increase system performance, different powerful and energy efficient parallel processing devices such as general purpose graphical processing unit (GPGPU), many integrated cores (MIC) [7] and field programmable gate array (FPGA) [8] are being configured in modern HPC systems. These parallel processing devices are programmed by using different parallel programming models including CUDA, OpenMP, OpenACC, OpenCL etc.

These technologies are being used very commonly in currently available HPC Tera-scale (10^{12}) and Peta-scale (10^{15}) supercomputing systems, but many applications are still taking a lot of time to complete single task. However, HPC development communities are moving toward new ambitions to introduce new level of supercomputing called “Exascale Systems” that will be capable to attain 1 ExaFlops (10^{18}) number of calculations in a second. To achieve this massive performance level, one straightforward solution is to increase the number of cores in homogeneous/heterogeneous systems that will be approximately up to 100 million number of cores in one supercomputer [9]. With this configuration, a projected performance is 10^{18} number of flops can be achieved to fulfil the required demand. Under these circumstances, current technologies are facing several challenges as discussed in [10] where massive power consumption is a major challenge when we intend to increase the said number of cores in the systems. Therefore, we should rethink and co-design the system architectures, new low power consuming hardware/software techniques, parallel programming methods, and applications [11].

In order to achieve the required performance under the power consumption limitations in emerging HPC system, massive parallelism is a vital solution. Although, various parallel programming models (CUDA, OpenACC, OpenMP, MPI, OpenCL etc.) are available to program parallel programming devices (GPU, FPGA, MIC etc.) but these models are implementable only on standalone or small cluster systems that cannot achieve the massive performance when we execute large applications. According to [12], hybrid (MPI+X) parallel programming strategies are the solutions that can achieve a massive performance under power consumption limitations by executing the applications on large cluster systems. Here, MPI is suggested to deal cluster systems by using message passing mechanism, whereas X might be any parallel programming model depending on system structure (Homogeneous/Heterogeneous). Leading to parallel programming for large cluster systems, translating serial code to parallel computing is a vital challenge for researchers. Traditionally, the researchers face various barriers while parallelizing their serial code as they don’t have enough experience to use parallel programming techniques in optimized way.

However, to address these obstacles and achieve massive performance under power consumption limitations, current study proposes an Automatic Parallel programming tool (AAP4All) for both homogeneous and heterogeneous large cluster systems. The key advantage of proposed tool is an auto recognition of computer system architecture, then translate automatically the input C++ serial code into parallel programming code. Further, the output parallel computing code is executed automatically on recognized computer system. Consequently, the proposed AAP4All model achieve massive performance by minimum power consumption through translate the input C++ code of any application into parallel computing code and run on targeted large cluster system. Further to contribution of our research is highlighted as follows:

- **Automatic translation of serial to parallel computable code:** We have proposed a new programming tool that take serial code written in C++ as input, and translate automatically into parallel code. Existing state-of-the-art methods translate only for single model either OpenMP (homogeneous) or CUDA (heterogeneous) parallel execution, whereas our proposed tool translate for multiple models such as: Single node models: MPI, OpenMP, CUDA, OpenACC, Cluster Systems models: MPI+CUDA, MPI+ OpenMP, MPI+ OpenACC, MPI+ OpenMP + CUDA. The details of each model is described in further sections.

- **Auto detection HPC computer system structure:** the proposed tool first detect the structure of targeted computer system automatically, then translate serial code to parallel code accordingly.
- **Parallel computation for multiple homogeneous/heterogeneous structures:** The proposed AAP4All tool auto translate serial to parallel code for both homogeneous and heterogeneous systems. It generates parallel computing code using MPI+ OpenMP hybrid model for homogeneous cluster system; CUDA, OpenACC for heterogeneous single node; MPI + CUDA, MPI + OpenACC for heterogeneous cluster system.
- **Massive parallelism for large HPC cluster system:** The proposed tool translates serial code to parallel code for large HPC heterogeneous cluster system, execute the output parallel code for tri-hybrid model (MPI + OpenMP + CUDA) and attain massive performance.
- **Reduce the power consumption:** Another key feature of our proposed model is reducing power consumption while executing large applications over large cluster systems.

The remainder of the paper is organized as follows: Section 2 briefly presents the background related to HPC technologies and related state-of-the-art methods used to translate sequential code to parallel code. Section 3 provides the detailed overview of proposed methodology of our research. In Section 4, we present the implementation and comparative analysis of our proposed model with existing approaches. Finally, Section 5 conclude the paper with future research directions.

2 Background Material and Related Work

We now give a brief background of different parallel programming techniques which are being used to program traditional CPU cores and accelerated GPGPU devices.

2.1 Technology Background

2.1.1 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a well-known library being used for data distribution and communication among the processes [13]. The primarily MPI was developed to use in distributed memory architectural systems which is considered as promising model for emerging HPC systems. Initially MPI first version 1.0 was introduced in 1994. Later on, MPI was improved with a number of fixes and new features for distributed parallel computing. A stable version of MPI 3.1 was released recently in 2015 where new features were introduced such as point to point message passing, collective communication and environmental management [14]. Any computing system should have following four fundamental reasons to use MPI model.

- **Standardization:** MPI is required to be implemented in any standard HPC system where message-passing is required among the processes.
- **Portability:** Where application is ported to another different platform that has MPI support.
- **Performance Opportunities:** An optimized communication should be implemented in any cluster/distributed system where performance is necessary objective.
- **Functionality:** The new MPI version includes around four hundred MPI routines that reduce the code writing effort and achieve the objectives within few line of code.

MPI has played a vital role throughout HPC development. Although originally MPI was not developed for future Exascale computing systems. However, in term of future super-computing systems, it has been anticipated that MPI will be the best option to be used for the communication/message passing among the process in large distributed systems. Under Exascale computing system requirements, still a number of improvements are required in MPI process synchronization and communication within the MPI processes.

2.1.2 OpenMP

Open Specification for Multi-Processing (OpenMP) is another parallel programming model which is very famous in SIMD based thread level parallelism over CPU cores. In short form OpenMP is called OMP. OMP is supported by FORTRAN and C++. Earlier OMP versions were used only for shared memory processing over CPU cores but according to OMP 4.5, it can be used to program accelerated GPU devices. Moreover, OMP 4.5 is considered a stable version that contains new features of error handling, atomics to control deadlock situations and task extensions as well [15]. It also contains the new strategies for threads synchronization where all parallel computing tasks are group using ‘taskgroup’ and then synchronized [16]. Now a day, OMP is used to attain fine grain parallelism by writing parallel computing statements in sections and corresponding sub-sections. Similarly, it refines the looping statements by using ‘taskloop’ directives and achieve fine grain parallelism [17].

2.1.3 CUDA

Compute Unified Device Architecture (CUDA) is another SIMD based parallel programming model to program NVIDIA GPU devices introduced by NVIDIA [18]. CUDA is supported by FORTRAN and C++ to write parallel program. The very first time CUDA 1.0 was introduced in 2006 when the idea of Exascale supercomputing was introduced. Continuing the enhancement in CUDA, currently most feature-packed and powerful 9.0 version is being used that has new profiling capabilities, support for pascal GPUs and lambda compilers [19]. According to CUDA architecture, it contains CUDA kernel that receive multiple necessary parameters (grid size, number of blocks, stream number etc.) along with data params for parallel execution on GPU cores. Before calling CUDA kernel, firstly data is transferred from host CPU cores to GPU cores by using same data types. After successful data transferring from CPU to GPU cores, CUDA kernels is called and executed. In same way, the processed data is transferred from device GPU cores to host CPU cores and complete execution process [20]. As GPU device contains thousands of cores, however each thread is executed on GPU core in parallel. In such way, CUDA complete the execution process very efficiently by parallelizing code.

2.1.4 OpenACC

With a rapid development toward heterogeneous programming models, OpenACC a new high level programming model was introduced recently by NVIDIA [21]. Several OpenACC compiler are available to perform in different platforms such as CAPS [22], Cray [23], and PGI [24]. The primary objective of OpenACC was to minimize writing effort by the programmers. Comparatively OpenACC can perform the same function in ten-folds less effort than CUDA and OpenCL. OpenACC contains new looping pragmas that can convert the data parallel version of kernel to OpenACC. In OpenACC, GPU is used effectively by minimizing data transferring between CPU processors and GPU devices. By using ‘copy’ clause in OpenACC program, cloverleaf is achieved on the device which results in a one-off initial data transfer to the device. The computational kernels exist at the lowest level within the application’s call-tree and therefore no data copies are required. As in any block-structured, distributed MPI application, there is a requirement for halo data to be exchanged between MPI tasks. In the accelerated versions, however, this data resides on the GPU local to the host CPU; hence, data to be exchanged is transferred from the accelerator to the host via the OpenACC “update host” clause. MPI send/receive pairs exchange data in the usual manner, and then the updated data is transferred from the host to its local accelerator using the

OpenACC "update device" clause. A key point to note is that the explicit data packing (for sending) and unpacking (for receiving) is carried out on the device for maximum performance.

2.1.5 OpenCL

OpenCL is a new parallel programming model for heterogeneous architectural systems. OpenCL is managed by Khronos group which has been implemented by more than ten vendors AMD [25], Intel [26], IBM [27], and NVIDIA [28]. The primary objective to design OpenCL was to run on heterogeneous architectures without re-compilation. In order to overcome c binding and FORTRAN codebase integration difficulties, Khronos group provides C++ header files that address this issue in object oriented manners [29]. These header files hold all the details about buffers and kernels used in the application. Based on these heterogeneous programmability features, OpenCL is becoming more famous and being.

Study and evaluated these programming models with respect to performance. Although many new programming models has been introduced recently such as pThread, C++11, TFluxSCC etc., but presented models in hybrid format are being considered the promising techniques for emerging massive parallel computing. Tab. 1 presents a comparative overview of these parallel programming.

Table 1: Single hierarchy models to program many-core GPU devices

PPM	Parallelism Patterns			Architecture abstraction		
	Data Paral-Lelism	Task Paral-lelism	Event Driven	Offloading	core Hierarchy	Data map-ping
OpenMP	Parallel for, SIMD	Task/taskwait	Depend (in —out)	Target device	OMP Places	Map (to—from)
OpenACC	Parallel/ kernel	Async/wait	Wait	Acc	Cache/gang/worker	Data copyin/ coupyout
CUDA	hhh…iii	Async kernel launching	Stream	hhh…iii	Blocks/threads, shared memory	Cudamecpy
OpenCL	Kernel	X	Pipe	clEnqueue	Work group	Buffer write fun

2.2 Related Work

A number of research articles have been published on automatic parallel computing, of which only those are directly related to our research work as depicted as follows. Existing techniques can be divided into two major categories as Homogeneous structured systems and Heterogeneous structured systems.

2.2.1 Homogeneous Structured Systems

In homogenous structured systems, a computer consists of traditional CPU cores only which are further parallelized by using different parallel programming models such as OpenMP for standalone systems [30] and hybrid of MPI+ OpenMP for large cluster systems [31]. However, Ishihara et al. [32] presented an interactive parallelizing tool called "iPat/OMP". iPat consists of some fundamental steps while translating serial to parallel code such as selection of specific section to be parallelized, dependency analysis, update the code for restructuring, and generate the final updated code. iPat translate the serial C code into parallel for OpenMP which is executable on multi-thread shared-memory structured system. The issue with iPat is that the generated output parallel code is not executable on any heterogeneous structure system. Moreover, after getting dependency analysis, user needs to inject the parallel OMP pragmas manually in the sections which are parallelizable.

Athavale et al. [33] proposed a new automatic translating serial to parallel (S2P) tool which transform the serial code written in C into multi-threaded parallel computing code. The output parallel code is computed by using OpenMP parallel pragmas. According to S2P model, the computer system should be supported by multi-threaded shared-memory architecture where the input data is parallelized using traditional CPU threads. However, in order to translate serial to parallel code, S2P analyze the input serial code statically where independent looping statements are recognized and profiled for parallel execution. One major loop whole with S2P is that it is applicable for only homogenous single node but not for any category of heterogeneous systems.

Manju et al. [34] introduced a mechanism of auto-translation serial to parallel code, but the proposed model was developed to achieve only coarse-grain parallelism where complete block of code was assumed to parallelize. In order to attain fine-grain parallelism where parallel pragmas are added at instruction level, the suggested model was not applicable. Raghesh [35] developed LLVM a low-level virtual machine based auto parallelizing tool with Polly integration. The proposed tool translates the C++ serial code into parallel code for OpenMP. It executes the output parallel code on simple CPU threads and perform operations. While translation process, it collaborates with polyhedral and z-polyhedra libraries to make dependency analysis and serial to parallel code translation. The proposed model also doesn't support for any accelerating GPU device to achieve massive parallelism over any heterogeneous structured systems.

Leading to automatic parallelization in homogenous systems, Reyes et al. [36] proposed a hybrid MPI +OMP technique using llc parallel programming language. The primary objective of the proposed model was to generate parallel computable code automatically for serial code written in high level programming languages. Although the proposed model generates hybrid MPI+OMP based parallel code but the authors highlighted some prominent challenges such as, by experimental perspectives, the proposed hybrid model was not implemented on variety of applications, and lack of implementation over heterogeneous GPU based structured systems.

Hamidouche et al. [37] also presented the similar hybrid MPI+OMP based auto parallel programming technique where the compiler analyzes the estimate execution time of sequential code. Once analysis completed, code generator translate serial to parallel code and generate a predictive execution time graph. Authors evaluated the proposed solution on different benchmarks and noticed that computations on symmetric multiprocessors (SMP) cluster based system outperformed the existing approaches and maintained the accuracy as well. The issue in this hybrid technique was the extensive usage of power while execution of large datasets. Authors mentioned in their future work that the proposed model should be implemented on GPU based cluster system to achieve massive performance under the low power consumption.

2.2.2 Heterogeneous Structured Systems

In heterogeneous structured systems, the modern low power consuming GPU devices are configured in traditional computer systems. These GPU devices contain thousands of parallel computing cores on which the data is computed in parallel. The host CPU cores transform the data from conventional CPU cores to accelerated GPU devices, once data processing is completed on parallel GPU cores, it return the processed data back to CPU cores for further processing. The advance GPU devices are very powerful and energy efficient, however, usage of GPU devices enhance the system performance as well as reduce the power consumption. Several studies have adopted these advance technologies and proposed different methods/tools as presented in this section. To utilize such powerful and energy efficient GPU devices, several automatic serial to parallel translating tools have been proposed in different studies. Marangoni et al. [38] proposed a tool TOGPU that automatic translate serial C++ code to CUDA based parallel code executable over NVIDIA GPU devices. During transformation, authors used Clang/LLVM to assist users in reining parallel computing over GPU devices. Although the proposed solution was useful and improved the system performance but it was applicable limited for image processing algorithms. In order to improve the Togpu tool, authors suggested to add some additional modules such as vectorization, code parsing and tokenization, and auto dependency analysis which are primary challenges for auto translation in homogenous/heterogeneous cluster systems.

Xie et al. [39] proposed a loop analysis framework named 'Proteus' that take the all looping statements along with set of dependent variables, summarize the loop effects via defining path dependency analysis. Authors also proposed a path dependency automaton (PDA) that determine the dependencies among the multiple paths in looping statements. They implemented the proposed Proteus framework on disjunctive loop based three different applications including loop bound analysis, program verification and test case generation. The primary focus of Proteus framework was to enhance the dependency analysis while automatic translation, however it was unable to translate code for large heterogeneous cluster systems. Moreover, authors proposed some future perspective research directions including (1) enhance PDA algorithm for extensive function summarization, (2) required a novel approach for loop summarization of different types, and (3) how to apply the proposed loop summarization on software engineering as well security related tasks.

Ventroux et al. [40] emphasized on dynamically code generation for new on chip multiprocessors based computer systems and proposed a combined SESAM/Par4All framework which is basically constitution of two different models SESAM [41] and Par4All [42] having different purposes. Authors used MPSoC architecture [43] in SESAM environment that was developed to design and explore asymmetric multiprocessor systems. Although this new joint framework provides a reasonable trade-off between performance and complexity for asymmetric homogeneous MPSoC applications. In contrast, the proposed joint featured model not supporting to advance NVIDIA GPU based heterogeneous cluster systems. Without cluster implementation, we cannot make sure the trade-off between complexity and performance of any framework. Augonnet et al. [44] worked on unified task scheduling for heterogeneous cluster system and proposed a new tool "StarPU". The main purpose of StarPU framework was to furnish numerical kernel designer that schedule the tasks to compute in parallel over heterogeneous system. From pre-define strategies, StarPU seamlessly select an appropriate method at run time and execute the task. The authors implemented StarPU and highly-optimized MAGMA library [45] in different benchmark applications and concluded that the proposed framework surprisingly outperformed the other strategies and executed the task efficiently. Authors themselves discussed the proposed StarPU framework was not able to execute the applications over versatile accelerated devices. Current proposed StarPU version supported only single GPU implementation by writing parallel computing code manually. However, it should be applicable over multiple GPU devices by automatically translating serial code into parallelizable code.

Planas et al. [46] proposed extended framework of OmpSs. The primary objective of this study was to attain the best level of performance through disclosing different versions of executing tasks. OmpSs is basically the combination of OpenMP and StarSs [47] that support to run the applications over symmetric multiprocessors and multiple accelerated GPU devices. Consequences showed the versioning scheduler feature in proposed framework outperformed the existing state-of-the-art scheduling techniques. Due to the complex architecture of proposed OmpSs version, programmers were inconvenient to translate the applications manually into parallelizable code which is the major drawback for it. The proposed OmpSs version can be improved by adding auto translation feature.

Hence, a novel framework is required that can translate serial to parallel executable over both homogeneous (MIC devices) and heterogeneous (advanced GPU based) large cluster systems. Under these considerations, we have proposed AAP4ALL a new automatic translating tool that convert the serial C++ code into parallel for multiple structured systems. The details of proposed model are given in following section.

3 AAP4ALL the Proposed Model

In this section we introduce our proposed AAP4All model, give its overview, and present its algorithmic refinement. AAP4All is an adaptive and automatic serial to parallel code translating tool designed to adaptively use multiple structured computer systems for parallel processing, facilitate researcher/developer by providing automatic serial to parallel code translation mechanism, and minimize power consumption while applications execution. Fig. 1 depicts the proposed AAP4ALL framework as follows. A prototype implementation of AAP4ALL system is described and evaluated in next Sections 4.

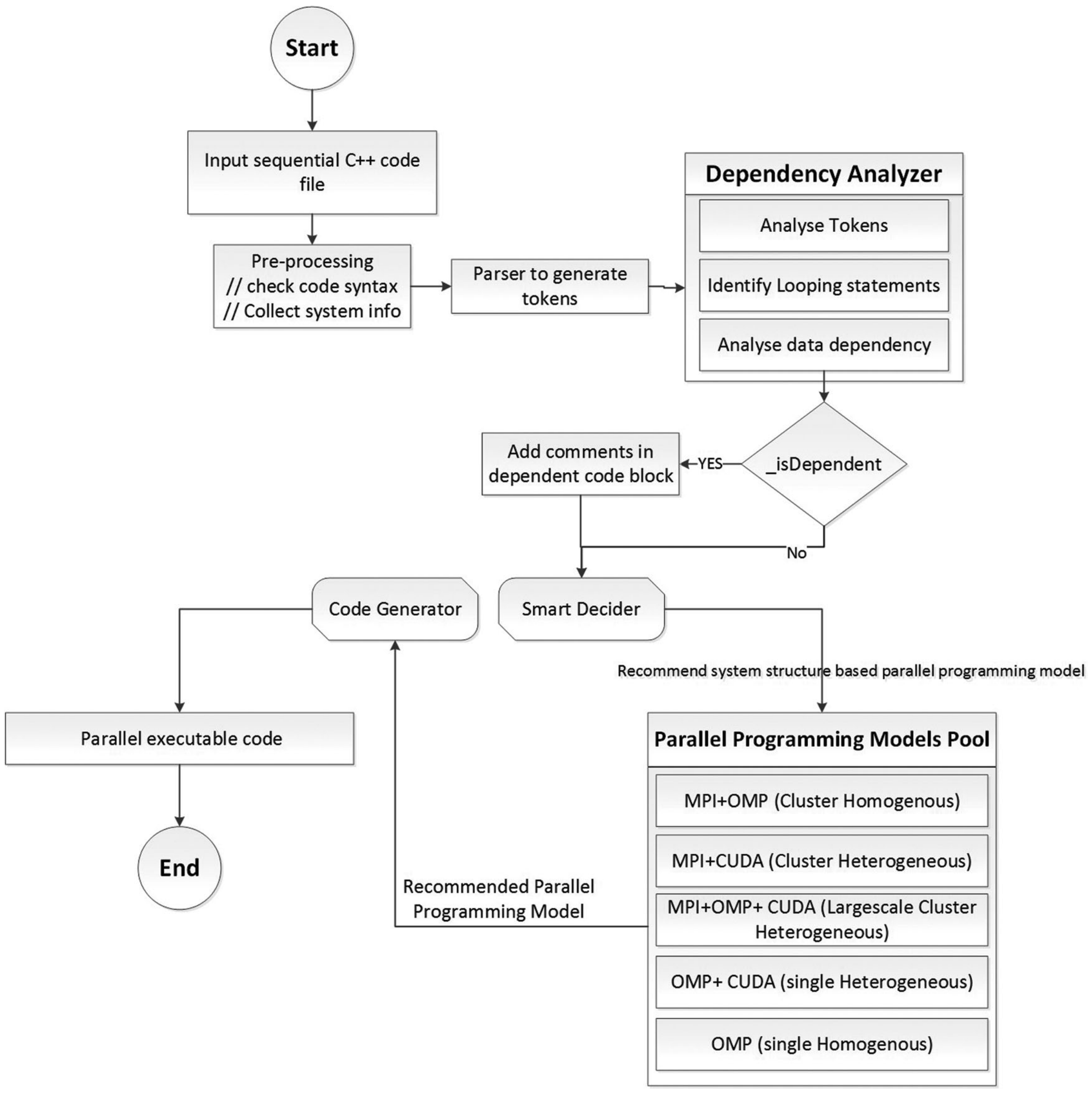
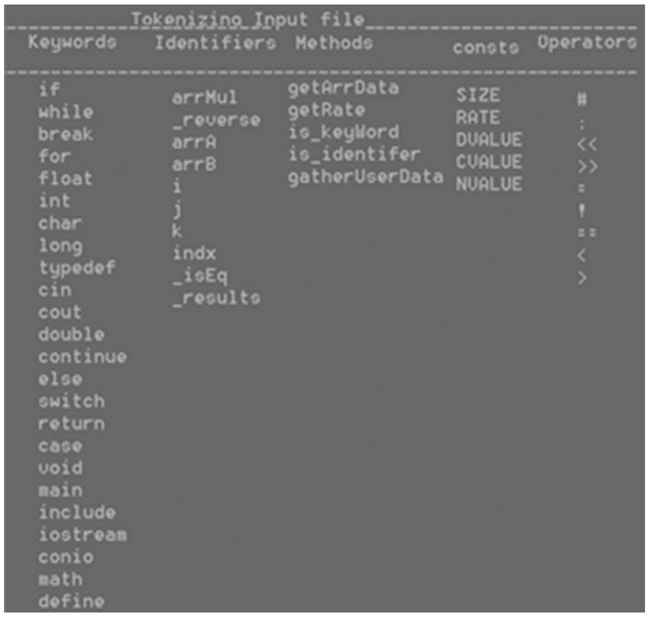

Start
Input sequential C++ code file
Pre-processing
// check code syntax
// Collect system info
Parser to generate tokens
Dependency Analyzer
Analyse Tokens
Identify Looping statements
Analyse data dependency
_isDependent
YES
No
Add comments in dependent code block
Code Generator
Smart Decider
Recommend system structure based parallel programming model
Parallel executable code
End
Recommended Parallel Programming Model
Parallel Programming Models Pool
MPI+OMP (Cluster Homogenous)
MPI+CUDA (Cluster Heterogeneous)
MPI+OMP+ CUDA (Largescale Cluster Heterogeneous)
OMP+ CUDA (single Heterogeneous)
OMP (single Homogenous)

Figure 1: Block diagram of proposed AAP4ALL framework

The AAP4ALL framework leverages on advanced and emerging technologies (High performance Computing, Deep learning, Machine learning, IoT) to ease in code writing for developers/researchers to enrich their knowledge for any computing system through adaptive and automatic parallel code generating tool. In order to attain the adaptive behavior and automatic generating parallel code, AAP4ALL framework contains multiple steps to be followed. From start, programmer inputs a file sequential code to generate parallel computing code with respect to target computer system. Before starting parallel code generate process, the system performs some fundamental operations such as call "Pre-processing" component that analyze the syntax of serial code, then collect some necessary information including system structure either it is homogeneous or heterogeneous, either the system is cluster or standalone machine, what type and number of accelerated GPU devices are configured in case of heterogeneous structure, bit system is 32-bits or 64-bits. Once the system information is gathered, the input sequential code is forwarded to "Parser" to parse code and generate number of tokens. Parser categorize the input data into five major lists such as list of keywords, identifiers, user defined methods, number of defined constant values, and operators used in the program. Fig. 2 depicts the generated tokens of input serial code.



```
___________Tokenizing Input file________________________
 Keywords    Identifiers  Methods           consts  Operators
---------------------------------------------------------------
   if                       getArrData
   while       arrMul       getRate          SIZE      #
   break       _reverse     is_keyWord       RATE      :
   for         arrA         is_identifer     DVALUE    <<
   float       arrB         gatherUserData   CVALUE    >>
   int         i                             NVALUE    =
   char        j                                       !
   long        k                                       ==
   typedef     indx                                    <
   cin         _isEq                                   >
   cout        _results
   double
   continue
   else
   switch
   return
   case
   void
   main
   include
   iostream
   conio
   math
   define
```

Figure 2: List of tokens generated by parser while parsing input serial C++ code

3.1 Dependency Analyzer

After parsing the serial code, the next step is to perform dependency analysis on the serial code that is supposed to be executed in parallel. Data dependency is an intractable challenge that is considered while transforming serial to parallel code, and parallelizing looping statements to achieve extensive performance through different parallel computing architectures. Unfortunately, such transformation mechanisms are not available in classical methods, and most of the tedious work of determining the dependency blocks for the serial-to-parallel code transformation is performed by the programmer manually.

To overcome this issue, the AAP4ALL framework provides a dependency analysis module called the "Dependency Analyzer" (DA), which is responsible for identifying the loop dependencies in the serial input code. In this study, the DA identifies different types of data dependencies, including the Flow, Input, Output, and Anti (FIOA) data dependency, as described in Tab. 2.

Table 2: Type of data dependencies identified in AAP4ALL framework.

Scenario	Sr.	Data dependencies	Description
For x = 1 to n For y = 2 to m a[x, y] = XYZ XYZ = a[x, y-1]	1	Flow Input	'x' precedes 'y', and 'x' access a value that 'y' uses
	2		'x' precedes 'y', and 'x' use a value that 'y' also use
	3	Output	'x' precedes 'y', and 'x' computes a value that 'y' also computes
	4	Anti	'x' precedes 'y', and 'x' uses a value that 'y' computes

Tab. 2 describes a scenario of nested Loops, where four different types of dependencies occur. To discern the data dependencies, the used terminologies are as follows. Data dependency exists in statement S1 to statement S2 if and only if the following:

Dependency, the used terminologies are as follows. Data dependency exists in statement S1 to statement S2 if and only if:

- Both S1 and S2 access the same memory location at the same time.
- S1 to S2, a flexible computation path exists.
- S2 is truly dependent on S1, if S2 access value of S1.
- S2 is anti-dependent on S1, if S2 writes a value read by S1.
- S2 is output dependent on S1, if S2 writes a value written by S1.
- Loop carried dependency exists and S1, S2 are dependent, if and only if x ¡ y or x= y, and a path occurs from S1 to S2.
- S1,S2 can be executed in parallel if there is no FIOA dependency between S1 and S2.

By following the above data dependencies terminologies, AAP4ALL dependency analyzer takes parser generated tokens as input and discern the all looping statements in the code. These looping statements along with perspective variables are further forwarded to another data dependency component that analyze each loop and inside statements. If DA finds the data dependency at any loop block, it returns true flag with index value of looping statement which is saved in a multi-dimensional array data dependency (arrDD). Based on arrDD values, DA generates a dependency graph for graphical representation with directed edges labeled with t, a, o and i. In next stage, DA traverse the whole arrDD and append comments on top of the loop body as "Loop carried dependency exists at this block. Remove dependency for parallel execution". Rest of parallelizable code is forwarded to 'Smart Decider' (SD) class to get parallel computing model recommendation. At this stage, SD analyze the received information during pre-processing and decide that which model is the most appropriate for parallel processing for targeted system structure. Once the decision is made, it SD choose the recommended model from parallel programming models pool. The details of each parallel programming model is given as in following section.

3.2 Parallel Programming Models

3.2.1 MPI + OpenMP

The hybrid of MPI and OMP is commonly being used for multi-cores distributed homogeneous systems. Data is transferred over distributed memory processors that communicate each other through MPI message passing interface. Once data is scattered, it further transferred to OMP region. OMP determines the available number of threads for that particular node and data written in OMP region is computed over these threads in shared memory access. In this implementation, we used MPI for coarse grain, whereas OMP for fine-grain parallelism over inter-node and intra-node respectively. Particularly fine-grain parallelism can be achieved by computing data through multiple loop pragmas within the outer OMP pragma. The latest OMP version is also capable to program GPGPUs for accelerated computing. OMP shared memory multi-processing programming model is available in C, C++ and FORTRAN for windows and UNIX platforms. The processing mechanism of hybrid MPI + OpenMP model has been presented in listing 1 as follows.

Listing 1: Parallel processing hybrid MPI + OMP model

```
1. MPI_Init() \\ initilize MPI region.
2. Size ← Get MPI_Comm_size() \\ get communication size
3. Rank ← MPI_Comm_rank () \\Get MPI ranks
4. If (Rank == 0) \\ for master process to send data
5. Make processing before Entering MPI Communication World
6. MPI_Send() \\ Send data to all processes rank > 0
7. MPI_Wait()\\ To check the processing periodically.
8. If (Rank > 0)
9. MPI_Recv() \\ Reieve data from all processes rank > 0
10. #pragma omp parallel
11. { \\ OMP parallel Region starting
12. #pragma omp parallel for default() shared (prm 1←N)
13. Processing Statements;
14. } \\ OMP parallel region ending
15. MPI_Send() \\ Send data again to all processes rank > 0
16. If(Rank == 0) \\ now master process collect data.
17. MPI_Recv() \\ Reieve finalized data from all rank > 0
18. MPI_finalize() \\ Finalize MPI region.
```

3.2.2 MPI + CUDA

Hybrid of MPI+CUDA is another promising model being used for node level and thread level optimization. During implementation in our experience, we have noticed that sometime the usage of more resources decrease the system efficiency. However, equal number of CPU processors were used as the number of GPUs devices connected in the system. The master MPI processor scatter data to all slave processes. These slave processes further used to transfer data from CPU cores to GPU devices. Once data is transferred on GPU devices, kernel is invoked where grid and thread block configurations were mentioned to optimize the resources. After GPU processing completion, data is copied back on Host CPU cores and utilized. According to latest API of CUDA 8.0, it has been improved by supporting PASCAL [48] and many debugging and profiling tools as well. A general data distributing and processing mechanism through hybrid of MPI+CUDA has been presented in listing 2 as follows.

Listing 2: Parallel processing hybrid MPI + CUDA model

```
1. MPI_Init() \\ initilize MPI region.
2. Size ← Get MPI_Comm_size() \\ get communication size
3. Rank ← MPI_Comm_rank () \\Get MPI ranks
4. If (Rank == 0) \\ for master process to send data
5. Make processing before Entering MPI Communication World
6. MPI_Send() \\ Send data to all processes rank > 0
```

(Continued)

Listing 2 (continued).

```
7. MPI_Wait()\\ To check the processing periodically.
8. If (Rank > 0)
9. MPI_Recv() \\ Reieve data from all processes rank > 0
10. #pragma omp parallel
11. { \\ OMP parallel Region starting
12. #pragma omp parallel for default() shared (prm 1←N)
13. Processing Statements;
14. } \\ OMP parallel region ending
15. MPI_Send() \\ Send data again to all processes rank > 0
16. If(Rank == 0) \\ now master process collect data.
17. MPI_Recv() \\ Reieve finalized data from all rank > 0
18. MPI_finalize() \\ Finalize MPI region.
```

3.2.3 OMP + CUDA

Another way of thread level parallelism at both CPU and GPU cores is hybrid of OMP+CUDA. This type of hybrid strategy is implementable for the application, which requires computing on shared memory access and accelerated devices. Although hybrid of OMP+CUDA is very efficient but not as much applicable for cluster-distributed systems because there is not message passing mechanism to communicate among distributed processors. Especially for intra-node coarse grain and fine grain parallelism, OpenMP along with CUDA is considered as the best approach. OpenMP directives are used to achieve coarse grain parallelism by adding "pragma omp parallel" directive before writing the code. Using resource optimizing strategy, the number of GPU devices are determined and equally number of OMP threads are allocated for heterogeneous cores processing. Listing 3 describes the processing way using hybrid OMP+CUDA.

Listing 3: Parallel processing hybrid OMP + CUDA model

```
1. MPI_Init() \\ initilize MPI region.
2. Size ← Get MPI_Comm_size() \\ get communication size
3. Rank ← MPI_Comm_rank () \\Get MPI ranks
4. If (Rank == 0) \\ for master process to send data
5. Make processing before Entering MPI Communication World
6. MPI_Send() \\ Send data to all processes rank > 0
7. MPI_Wait()\\ To check the processing periodically.
8. If (Rank > 0)
9. MPI_Recv() \\ Reieve data from all processes rank > 0
10. #pragma omp parallel
11. { \\ OMP parallel Region starting
12. #pragma omp parallel for default() shared (prm 1←N)
13. Processing Statements;
14. } \\ OMP parallel region ending
15. MPI_Send() \\ Send data again to all processes rank > 0
16. If(Rank == 0) \\ now master process collect data.
17. MPI_Recv() \\ Reieve finalized data from all rank > 0
18. MPI_finalize() \\ Finalize MPI region.
```

3.2.4 MPI + OMP + CUDA

In order to deal a large cluster system for massive parallelism, we proposed another tri-hierarchy level hybrid parallel programming model for AAP4ALL model. The purpose of this model was to achieve massive performance through monolithic parallelism when we compute any HPC application over a large-scale cluster system having multiple nodes and number of GPUs ¿ 2. As discussed in earlier section, the proposed tri-hierarchy model was incorporated of MPI, OpenMP and CUDA where MPI is responsible for broadcasting data over distributed nodes, OpenMP to run received data in parallel on CPU threads, and CUDA is responsible to run the code on GPU, which is third level of parallelism. Tri-hybrid model is capable to achieve massive parallelism by computing data through inter-node, intra-node and GPUs. This model provides three level of granularity including coarse grain by MPI, fine grain using OMP threads and finer granularity over GPU devices. Data computation start CPU from CPU processors, which is distributed by MPI process, further this data, is processed on CPU threads by OMP. Within the OMP pragmas, CUDA kernels are invoked to compute data over GPU cores. In such way, this tri-hybrid compute data and return to host MPI master process. Listing 4 describes the processing of all MPI, OMP and CUDA in combine.

Listing 4: Parallel processing hybrid MPI+ OMP + CUDA model

```
1. MPI_Init() \\ initilize MPI region.
2. Size ← Get MPI_Comm_size() \\ get communication size
3. Rank ← MPI_Comm_rank () \\Get MPI ranks
4. If (Rank == 0) \\ for master process to send data
5. Make processing before Entering MPI Communication World
6. MPI_Send() \\ Send data to all processes rank > 0
7. MPI_Wait()\\ To check the processing periodically.
8. If (Rank > 0)
9. MPI_Recv() \\ Reieve data from all processes rank > 0
10. #pragma omp parallel
11. { \\ OMP parallel Region starting
12. #pragma omp parallel for default() shared (prm 1←N)
13. Processing Statements;
14. } \\ OMP parallel region ending
15. MPI_Send() \\ Send data again to all processes rank > 0
16. If(Rank == 0) \\ now master process collect data.
17. MPI_Recv() \\ Reieve finalized data from all rank > 0
18. MPI_finalize() \\ Finalize MPI region.
```

3.3 Code Generator

After getting the recommended parallel computing model, the final step is to transform input serial C++ code into parallelizable code by adding the perspective listing mechanism described above. Code generator (CG) component is responsible to implement the listing of recommended parallel programming model. CG recall the generated tokens during serial code parsing process, identify the code blocks using looping keywords which are supposed to be executed in parallel then add that block inside the parallel pragmas and directives.

4 Implementation and Results

This section presents the experimental setup that was used to implement the proposed model. Moreover, we quantified different HPC-related metrics, including performance (number of Gflops/s) and energy efficiency (Gflops/Watt) in the system. This section contains a detailed description of these HPC metrics.

4.1 Experimental Setup

All the experiments presented in this section are performed on Aziz Supercomputer Aziz-Fujistu Primergy CX400 Intel Xeon Truescale QDR supercomputer is manufactured by Fujistu [49]. According to top-500 supercomputing list, Aziz was ranked at 360th position [50]. Originally Aziz was develop to deal HPC applications in Saudi Arabia and collaborated projects. It contained 492 number of nodes which are interlinked within the racks through InfiniBand where 380 are regular and 112 are large nodes with additional specifications. Previously Aziz was capable to run the applications only on homogeneous node but due to requirements of massive parallel computing, it was upgraded by adding two SIMD-architecture-based accelerated NVIDA Tesla k-20 GPU devices where each device has 2496 CUDA cores. Moreover, two MIC devices with an Intel Xeon Phi Coprocessor with 60 cores were also installed to upgrade homogeneous computational architecture. In such way, Aziz contains total 11904 number of core in it. Regarding memory, each regular node contained by 96 GB and larger (FAT) nodes with 256 GB. Each node has Intel E5-2695v2 processor that contains twelve physical cores with 2.4 GHz processing power. All the nodes and accelerated devices are interlinked through three different networks including user, management and InfiniBand network. For Aziz, user network and management networks are specifically used for the login and job submission handling whereas InfiniBand to parallelize the file system. According to LINKPACK benchmarks, Aziz's peak performance was measured with 211.3 Tflops/s and 228.5 Tflops/s as theoretical performance [51]. Regarding software specifications, it run using Cent OS with release 6.4. For accelerated programming CUDA recent toolkit 11 and many HPC libraries are installed.

4.2 Performance Evaluation

In HPC system, a set of metrics are being used to quantify the performance of any particular algorithm. Traditionally, the metrics including time and space are enough to quantify the performance in sequential algorithms but complication of parallel computing requires some additional factors including the number of FLOPs and speedup [52]. To measure the number flops, time execution and number of floating point operations are required to be determined in a particular algorithm. Therefore, number of flops can be determined by dividing the total number of floating point operations (FPOs) of an algorithm by measured execution time through parallel $T_{p.exe}$ algorithm as expressed in Eq. (1).

$$\mathrm{Flops} = \mathrm{TN}_{\mathrm{FPOs}}/\mathrm{T}_{\mathrm{p.exe}} \tag{1}$$

Regarding speedup, we measured it by dividing sequential execution time with parallel execution time. Theoretically, speedup can be expressed as follows in Eq. (2).

$$SpeedUp\ (S_p) = \frac{T_{serial}\ (n, 1)}{T_{parallel}\ (n, p)} \tag{2}$$

In Eq. (2), $T_{serial}\ (n, 1)$ is the optimal time in sequential processing and $T_{parallel}\ (n, p)$ for parallel computing algorithm for any n size problem. The possibilities of speedup [53] in an algorithm could be expressed as follows in Fig. 3 that belongs to a linear path. These possibilities can be higher or less than the linear path where upper paths called as perfect linear or super linear and lower paths are called sub-linear path.

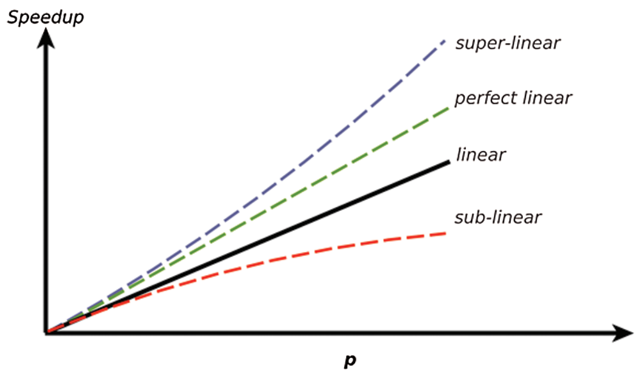


Figure 3: Speedup possible curves

We performed all the experiments on Aziz supercomputer by using different number nodes (4, 8, 16, 32, 64, 128). We implemented the proposed AAP4ALL model in two HPC linear algebraic systems such as dense matrix multiplication algorithm (DMMA) and Jacobi solver for 2-D Laplace equation. As we talk about the HPC large system systems, however we selected the large computation datasets as 7000 × 7000 and 8000 × 8000 matrix size and mesh size for DMMA and Jacobi iterative method respectively. First, we discuss DMMA and selected existing state-of-the-art approaches that we used to compare our results with proposed AAP4ALL model. These approaches include TOGPU used for heterogeneous systems and S2P for homogenous cluster systems.

TOGPU is CUDA based fast GPU-accelerated implementation of the standard basic linear algebra subroutines. TOGPU speed up the applications by deploying compute-intensive operations to a single GPU or scale up and distribute work across multi-GPU configurations efficiently. Researchers use TOGPU for automatic code translation in different domains including high performance computing, image analysis and machine learning. In contrast, AAP4ALL framework translate the serial code to parallelizable hybrid of MPI+CUDA code for perspective system. During the experiments, we noticed that before starting the execution on the system, a critical analysis of execution code and then efficient scheduling play a vital role in performance improvement. Fig. 4 presents the achieved PFlops of AAP4ALL framework at different number of nodes and compared with existing highly optimized models as TOGPU, S2P, OmpSs, and StarPU. According to Figs. 4a and 4b, throughout at all node levels, the proposed AAP4ALL model outperformed to all existing strategies TOGPU (heterogeneous GPU computation), S2P (homogeneous CPUs computation), StarPU and OmpSs. During the execution at maximum number of nodes 128 on Aziz supercomputer, we observed that AAP4ALL scheduler analysed the input code seamlessly and preferred Tri-hybrid (MPI+OMP+CUDA) model at runtime that achieved maximum number of Petaflops 48.69 and 53.69 against the matrix size 7000 × 7000 and 8000 × 8000 respectively. In contrast, the TOGPU heterogeneous computing library could achieve maximum 16.74 and 28.14 number of Pflops at same executions. Although TOGPU results were also not much efficient but comparatively better than S2P homogenous execution. On other hand, we noticed due to efficient scheduling strategy, OmpSs highly optimized scheduling algorithm achieved 36.4 number of Pflops that was the second maximum level of performance among all the methods. We also observed that although StarPU contain efficient tasks scheduling mechanism but because of limitations in using multiple GPUs, it couldn't achieve the extensive performance and achieved only 15 Pflops at maximum matrix multiplication executions.

Similarly, we assessed the speedup another essential performance metric. Using measured execution time, we determined the speedup in all selected state-of-the-art methods and compared with our proposed approach as shown in Figs. 5a and 5b. According to the determinations, Speedup gradually increased in

AAP4ALL model while increasing resources and matrix size. The results show that our approach achieved maximum speedup 36.13, whereas OMPSs from existing state-of-the-art strategies could achieve 22.7 at second maximum speed-up. We found this difference due to optimized and automatic translation of serial to parallel computing code, and the right selection of parallel programming paradigm for the targeted structure. Mostly people try to compute their code in parallel using single CUDA library over any GPU based machine which is less efficient approach by all aspects (time, resources, power consumption etc.).

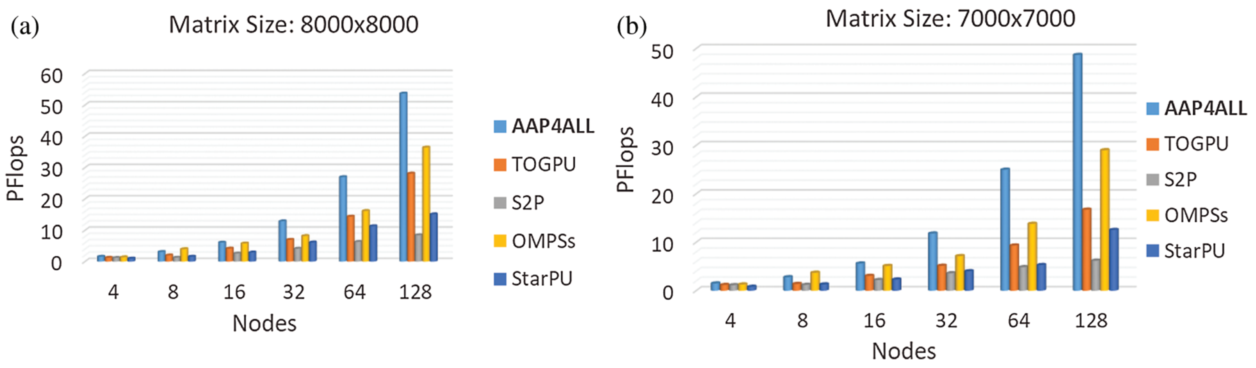


Figure 4: Performance (Petaflops) evaluation: AAP4ALL, S2P, TOGPU, OPMSs, and StarPU comparative analysis while executing matrix multiplication on Aziz supercomputer with matrix size (a) 8000 × 8000, (b) 7000 × 7000

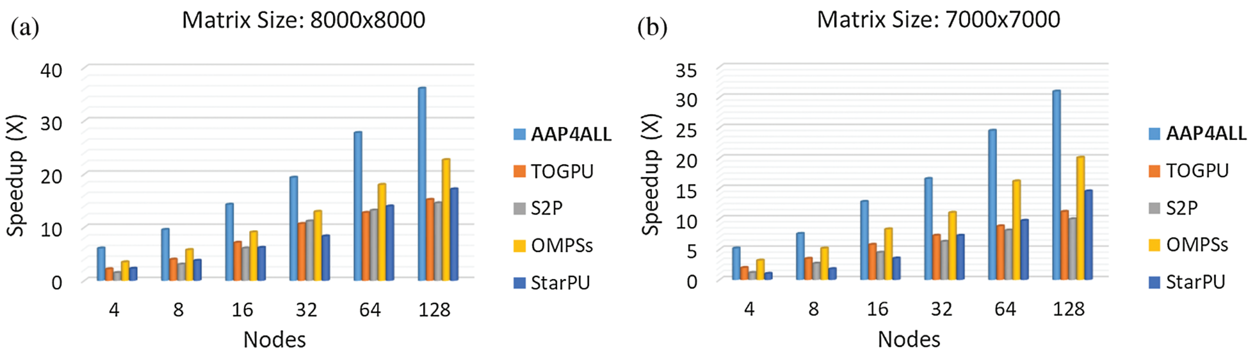


Figure 5: Performance (speed-up) evaluation: AAP4ALL, S2P, TOGPU, OPMSs, and StarPU comparative analysis while executing matrix multiplication on Aziz supercomputer with matrix size (a) 8000 × 8000, (b) 7000 × 7000

As discussed in earlier sections, the fundamental property of a classical dense matrix multiplication (DMM) algorithm is asymptotic complexity AC3 that includes multiplication, addition and casting operations. Based on the application complexities AC3, we presented the results by implementing in different models. In contrast, what if the number of floating point operations are increased when we use different HPC applications? For instance, asymptotic complexity in a Jacobi iterative solver for 2-D Laplace equation method is considered as AC36, which is nine time greater than DMM AC. However, below the list of question are open challenges for HPC research communities that what else when we have massive dataset applications for executions:

- What will be the behavior of implementing model?
- What will be the effect on system performance and response time?

- Can we optimize the hardware utilization?
- Can we use the multiple kernels during executions?
- What will be effect on system temperature when we try to utilize the optimized hardware resources?
- What will be the effect on power consumption?

This is a very critical to answer these questions without an empirical analysis by implementing any HPC application having higher AC. Therefore, in order to meet these queries, we selected Jacobi iterative solver for 2-D Laplace equation another benchmarking application. We implemented Jacobi iterative solver for 2-D Laplace equation in AAP4ALL and same existing techniques. In the experiments, number of Jacobi iterations were fixed as 1000 and change the mesh size along x-axis and y-axis. Increase in mesh size means the increase in accuracy. However, we selected two large values 7000 × 7000 and 8000 × 8000 which are respectively 3.528E+12 and 4.608E+12 a very large number of floating point operations. In the light of higher order AC in Jacobi iterative solver for 2-D Laplace equation, we quantified the same performance parameters as number of flops and speedup. Fig. 6 and 7 depict the achieved PFlops and Speedup respectively.

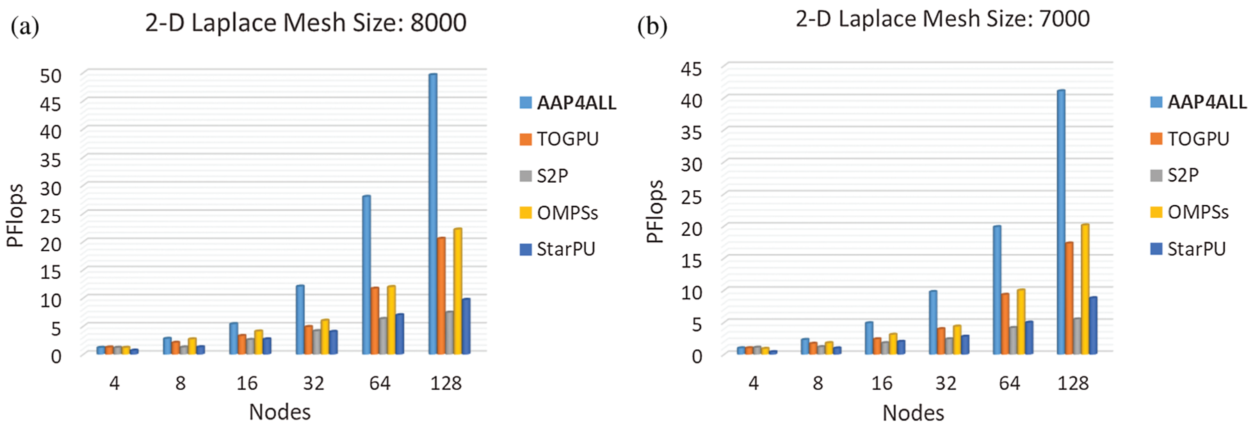


Figure 6: Performance (Petaflops) evaluation: AAP4ALL, S2P, TOGPU, OPMSs, and StarPU comparison while executing Jacobi solver 2-D Laplace equation on Aziz supercomputer with mesh size (a) 8000, (b) 7000

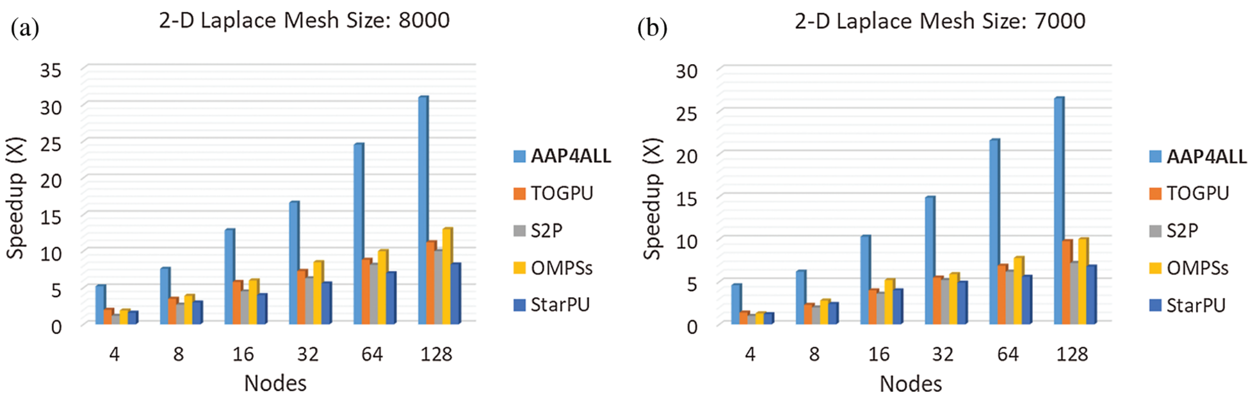


Figure 7: Performance (Speedup) evaluation: AAP4ALL, S2P, TOGPU, OPMSs, and StarPU comparison while executing Jacobi solver 2-D Laplace equation on Aziz supercomputer with mesh size (a) 8000, (b) 7000

According to Fig. 6, despite of a very large complex problem, AAP4ALL executed the large mesh size 7000 × 7000 efficiently and attained 41.06 Pflops by using 128 nodes in Aziz supercomputer. In contrast, we observed that OmpSs is not as such efficient when the application code is complex. However, OmpSs and TOGPU relatively achieved 19 number of Petaflops. By increasing the mesh size at 8000 × 8000, AAP4ALL achieved a tremendous performance and maintained the gradual increase in Petaflops. Fig. 6a shows that our proposed approach outperformed to all other models and achieved 43.62 maximum Petaflops. The primary reason behind performance improvement is the correct allocation and resources utilization. During executions once a job is submitted, it reserves some resources to complete this task, meanwhile what happen with rest of the resources? That should be utilized meanwhile to perform other tasks instead of waiting unlinked executions.

We also evaluated the speedup by executing Jacobi iterative solver for 2-D Laplace equation through proposed AAP4ALL model and compared the results with existing tools. Fig. 7 depicts that AAP4ALL attained 31X speedup when we compute the mesh size 8000 × 8000 using 128 nodes over Aziz supercomputer. On other hand, with same benchmark, S2P, TOGPU, StarPU, and OmpSs could reach only (11, 11.2, 8.2, and 13) X speedup.

4.3 Power Consumption Evaluation

Power consumption is one of the vital challenge for current and emerging HPC systems. The primary objective of future research for Exascale computing system is the correct selection of hardware/software to achieve massive performance under power consumption limitations [54]. Many HPC pioneers have taken initiatives and developed energy efficient devices such as NVIDIA GPGPU, AMD GPU, and Intel MIC. As discussed in earlier sections, the future Exascale supercomputing system will heterogeneous architectural system, however it is necessary to emphasize primarily on power consumption for both homogenous and heterogeneous architectural based systems.

The power consumption in any heterogeneous system is considered into three major parts including power consumed by host CPU processors, power consumed by memory operations (inter-memory and intra-memory), and power consumed by accelerated GPU devices. However, the total power consumption of a system can be quantified by aggregating the power consumption by all these key elements as given in Eq. (3).

$$E_{total} = E_{CPU} + E_{GPU} + E_{data} \tag{3}$$

From above equation, the total consumed energy was calculated by integrating the energy consumption by CPU, GPU and data. Traditionally, a computer system is quantified in power consumption that can be derived as follows in Eq. (4).

$$\text{Power (P)} = \text{E/t}, \text{ (where power P is in watts)} \tag{4}$$

The measurement of power consumption can be classified [55] majorly into following two categories.

- System Specification
- Application Specification

A computer system having configured GPU devices in it, the power consumption is calculated by using following Eq. (5):

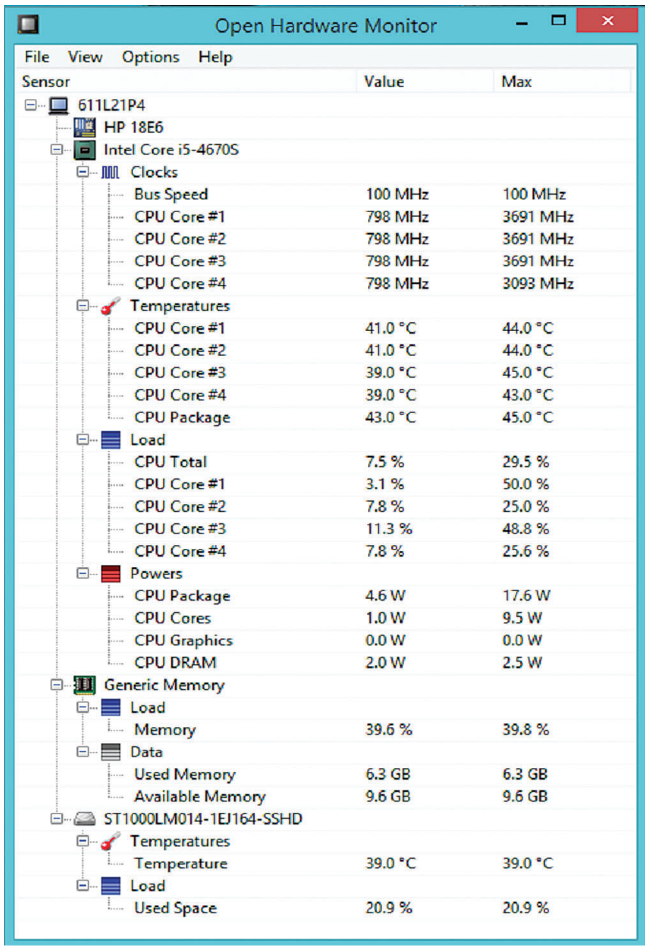
$$P_{system}(w) = \sum_{i=1}^{N} P^{i}{}_{GPU}\left(w^{i}\right) + P_{\ CPU} \sum_{j}^{M} \left(w^{j}\right) + P_{mainboard}(w) \tag{5}$$

Where Psystem, Pcpu, Pgpu, Pmainboard represent the power of the system, GPU, CPU and mainboard, respectively. N is the number of GPUs; M is number of CPU threads; w is the workload running on the system, W_i and W_j represent the workload assigned to GPU i and CPU threads j respectively. The power consumption varies with workload, however on application side, it can be quantified using following Eq. (6):

$$P_{app}(w) = \sum\nolimits_{i=1}^{Napp} P^{i}{}_{GPU}\left(w^{i}\right) + P_{CPU} \sum\nolimits_{j}^{M} \left(w^{j}\right) + P_{mainboard}\left(w_{app}\right) \tag{6}$$

4.3.1 Open Hardware Monitor

The Open Hardware Monitor is a free open source software that monitors temperature sensors, fan speeds, voltages, load, and clock speeds of a system. The Open Hardware Monitor supports most hardware monitoring chips found on today's mainboards and run on 32/64-bit Microsoft Windows all versions and any x-86 based Linux operating systems without installation. Fig. 8 depicts the running state of OpenHardwareMonitor as follows.

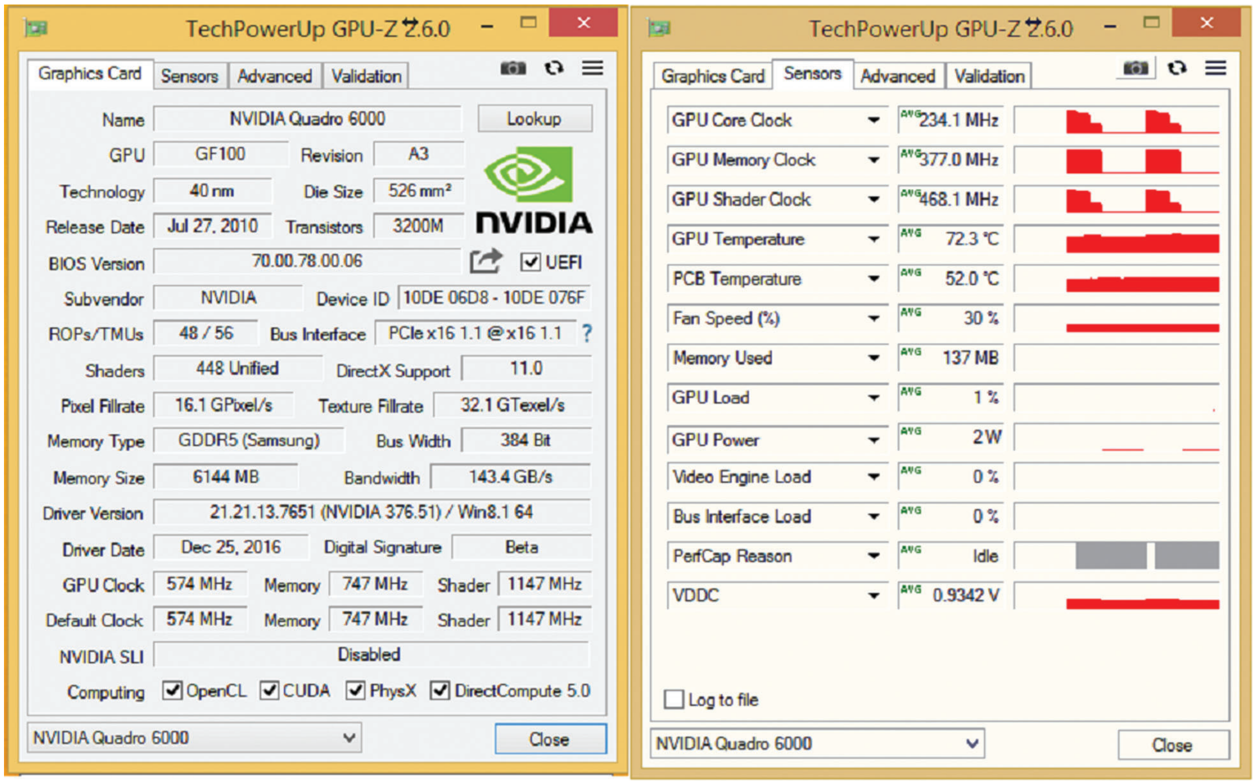


Sensor	Value	Max
611L21P4		
HP 18E6		
Intel Core i5-4670S		
Clocks		
Bus Speed	100 MHz	100 MHz
CPU Core #1	798 MHz	3691 MHz
CPU Core #2	798 MHz	3691 MHz
CPU Core #3	798 MHz	3691 MHz
CPU Core #4	798 MHz	3093 MHz
Temperatures		
CPU Core #1	41.0 °C	44.0 °C
CPU Core #2	41.0 °C	44.0 °C
CPU Core #3	39.0 °C	45.0 °C
CPU Core #4	39.0 °C	43.0 °C
CPU Package	43.0 °C	45.0 °C
Load		
CPU Total	7.5 %	29.5 %
CPU Core #1	3.1 %	50.0 %
CPU Core #2	7.8 %	25.0 %
CPU Core #3	11.3 %	48.8 %
CPU Core #4	7.8 %	25.6 %
Powers		
CPU Package	4.6 W	17.6 W
CPU Cores	1.0 W	9.5 W
CPU Graphics	0.0 W	0.0 W
CPU DRAM	2.0 W	2.5 W
Generic Memory		
Load		
Memory	39.6 %	39.8 %
Data		
Used Memory	6.3 GB	6.3 GB
Available Memory	9.6 GB	9.6 GB
ST1000LM014-1EJ164-SSHD		
Temperatures		
Temperature	39.0 °C	39.0 °C
Load		
Used Space	20.9 %	20.9 %

Figure 8: Running state of OpenHardwareMonitor to measure CPU parameters

4.3.2 GPU-Z

On other hand, GPU-Z is also open source application for personal and commercial usage. However, we may not redistribute GPU-Z as part of a commercial package. GPU-Z development community also offers a GPU-Z SDK, which is provided as simple-to-use DLL with full feature set. Fig. 9 shows the running state during programming execution.


TechPowerUp GPU-Z 2.6.0

Graphics Card | Sensors | Advanced | Validation

Name	NVIDIA Quadro 6000		
GPU	GF100	Revision	A3
Technology	40 nm	Die Size	526 mm²
Release Date	Jul 27, 2010	Transistors	3200M
BIOS Version	70.00.78.00.06		✓ UEFI
Subvendor	NVIDIA	Device ID	10DE 06D8 - 10DE 076F
ROPs/TMUs	48 / 56	Bus Interface	PCIe x16 1.1 @x16 1.1
Shaders	448 Unified	DirectX Support	11.0
Pixel Fillrate	16.1 GPixel/s	Texture Fillrate	32.1 GTexel/s
Memory Type	GDDR5 (Samsung)	Bus Width	384 Bit
Memory Size	6144 MB	Bandwidth	143.4 GB/s
Driver Version	21.21.13.7651 (NVIDIA 376.51) / Win8.1 64		
Driver Date	Dec 25, 2016	Digital Signature	Beta
GPU Clock	574 MHz	Memory 747 MHz	Shader 1147 MHz
Default Clock	574 MHz	Memory 747 MHz	Shader 1147 MHz
NVIDIA SLI	Disabled		
Computing	✓ OpenCL ✓ CUDA	✓ PhysX	✓ DirectCompute 5.0

NVIDIA Quadro 6000 — Close

TechPowerUp GPU-Z 2.6.0 — Sensors

GPU Core Clock	AVG 234.1 MHz
GPU Memory Clock	AVG 377.0 MHz
GPU Shader Clock	AVG 468.1 MHz
GPU Temperature	AVG 72.3 °C
PCB Temperature	AVG 52.0 °C
Fan Speed (%)	AVG 30 %
Memory Used	AVG 137 MB
GPU Load	AVG 1 %
GPU Power	AVG 2 W
Video Engine Load	AVG 0 %
Bus Interface Load	AVG 0 %
PerfCap Reason	AVG Idle
VDDC	AVG 0.9342 V

☐ Log to file

NVIDIA Quadro 6000 — Close

Figure 9: Running state of Tech PowerUp GPU to measure GPU parameters

According to Eqs. (5) and (6), a single node in Aziz supercomputer consume 12-watt power at idle state. Further, in order to quantify the power consumption during application execution, we used well-known software applications 'Open Hardware Monitor' [56] and GPU-Z.2.6.0 [57] for measurement of CPU and GPU temperature and power consumption respectively.

By following the same specification described in performance measurements, we quantified the aggregated power consumption in proposed AAP4ALL and existing state-of-the-art approaches. Fig. 10 demonstrates the power consumption while executing dense matrix multiplication application over Aziz supercomputer using various nodes. The results showed that our proposed model consumed the minimum power 31 KW for largest computation matrix size 8000 × 8000 using 128 nodes. In contrast, S2P and TOGPU consumed a massive amount of power and completed the same executions in 76.54 and 118.27 KW respectively. We noticed that although OmpSs is optimized in scheduling but not energy efficient which consumed 83 KW. The similar ratio was found during execution of matrix size 7000 × 7000 as presented in Fig. 10b. Statistics show that only AAP4ALL is considerable to for parallel computing in HPC system by all aspects (Performance and power consumption).

We also measured the power consumption for Jacobi iterations for solving 2-D Laplace equation as presented in Fig. 11. It was observed that despite of massive asymptotic complexity in 2-D Laplace equation, the proposed AAP4ALL completed the large execution of mesh size 8000 within ~36 KW power whereas the other existing strategies (TOGPU, S2P, OmpSs, StarPU) consumed massive power in the range of 82~145 KWs which is not affordable for computers in third world countries.

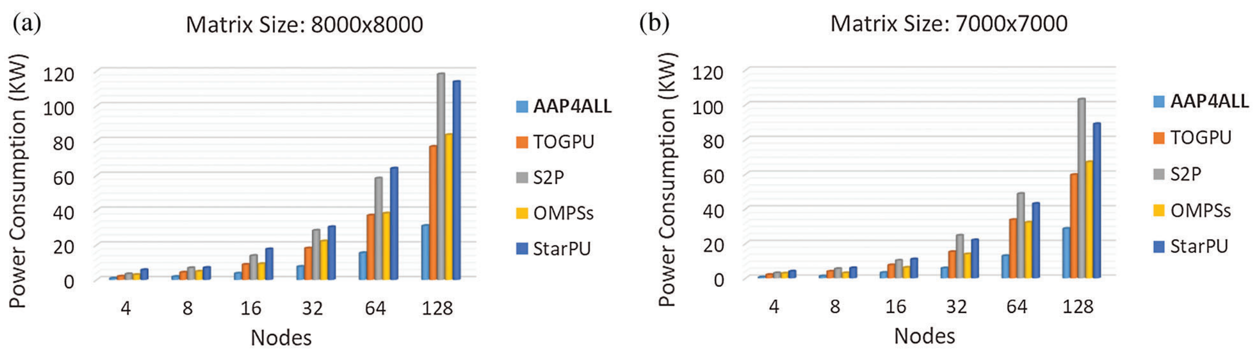


Figure 10: Power consumption (kilo-watt) evaluation: AAP4ALL, S2P, TOGPU, OPMSs, and StarPU comparative analysis while executing matrix multiplication on Aziz supercomputer with matrix size (a) 8000 × 8000, (b) 7000 × 7000

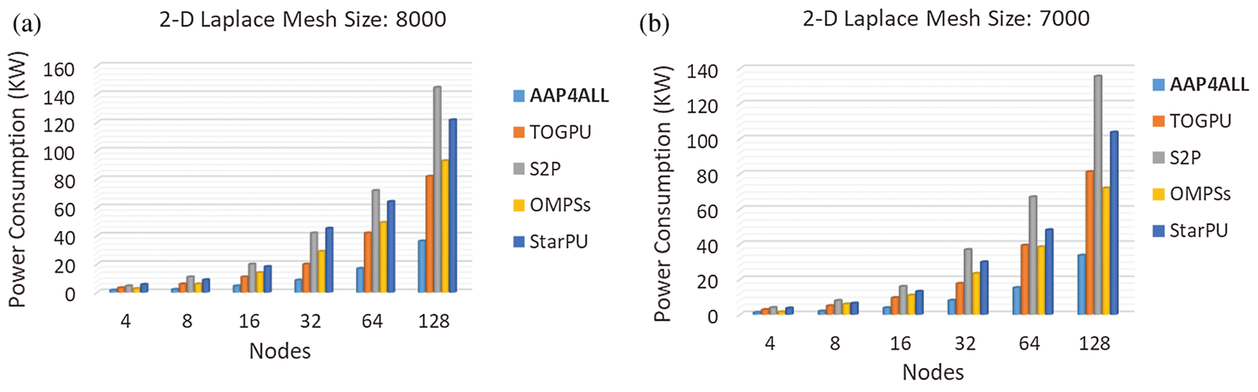


Figure 11: Power consumption (kilo-watt) evaluation: AAP4ALL, S2P, TOGPU, OPMSs, and StarPU comparison while executing Jacobi solver 2-D Laplace equation on Aziz supercomputer with mesh size (a) 8000, (b) 7000

5 Discussion, Conclusion and Outlook

In this paper, we propose an Adaptive & Automatic serial to parallel translation tool (AAP4ALL) for multiple computer structure systems. The proposed framework provides three primary features including (1) analyse code dependency then automatic translate sequential C++ code to parallelizable code, (2) enhance system performance by generating parallelizable code for large Exascale level HPC system, and (3) decrease the power consumption while executing larger applications. To the best of our knowledge, this is the first optimized strategical framework that analyse the input serial code, recognize the dependent/independent statements, and translate the parallelizable code to a specific parallel programming model which is most suitable for the targeted computer system. In order to validate the said features, we implemented our proposed AAP4ALL model and some existing state-of-the-art methods (TOGPU, S2P, StarPU, and OmpSs) in two benchmarking applications including (1) dense matrix multiplication and (2) Jacobi iterative solver for 2-D Laplace equation. All experiments were performed on Aziz supercomputer that contains 492 nodes along with two K-20 NVIDIA GPU devices. Theoretically the peak performance of Aziz supercomputer is 211.3 Tflops/s. The first primary difference in our proposed and existing models is that our technique supports to all kind of computer structures (Single/cluster both homogenous and heterogeneous structured systems) while existing techniques were developed for any specific structured system. Further in experiments, we evaluated different metrics such as performance attributes (PFlops, Speedup) and power consumption (KW) against each execution. Regarding performance in term of PFlops, our proposed framework achieved 54.3 and 43.6 PFlops while executing 8000 x 8000 matrix size in matrix multiplication, and 8000 mesh size in Jacobi iteration for 2-D Laplace equation respectively using 128 nodes of AZIZ supercomputer. On other hand, from existing mechanisms, because of a huge difference in evaluated results, we can't compare the all techniques with our model. However, we noticed OmpSs the highly optimized mechanism could attain maximum ~36, 29 PFlops against matrix size 8000 and 7000 respectively. Further, we calculated the speedup in all model and observed that AAP4ALL reached up to 31~37 speedup, whereas other methods achieved speed up 14-22x. During the experiments, we measured the power consumption while all executions and observed that our AAP4ALL framework consumed maximum 31 KW power while executing larger matrix size 8000 × 8000 on 128 nodes of AZIZ supercomputer. Alternatively, S2P homogeneous model consumed 118 KW but heterogeneous models (TOGPU, StarPU, OmpSs) comparatively depleted less power (76, 83, 113) to complete the similar tasks. We also found a massive power consumption by all existing parallel computing strategies in the range of 82–144 KW while executing Jacobi solver for 2-D Laplace equation, whereas AAP4ALL completed the same executions within 36KW power consumption. It was noticed that, the power consumption in existing methods increase gradually when we increase the dataset size. Notwithstanding, AAP4ALL balanced the load efficiently and completed all the executions under the power consumption range 30~36KW. The consequences showed that our proposed AAP4ALL framework outperformed to all existing models throughout the executions by consuming minimum power.

By future perspectives, our plan is to improve the proposed framework by adding some essential automated features such as:

- Removing dependency in the code. In current proposed version, instead of removing dependency, our model highlights the piece of code which is actually dependent and not parallelizable. But this is still challenge for the researcher/developer that how to remove the dependent statements to execute in parallel?
- Add OpenACC module. As OpenACC is also considered as an easy to GPU coding parallel programming paradigm, however we 'll add OpenACC module in parallel programming pool in our future AAP4ALL version.
- AAP4ALL support for FORTRAN. As the used libraries are supportive for C++ and FORTRAN programming language as well but our proposed framework support only the code written in C++. In future our plan is to add the support for FORTRAN language as well that will translate the FORTRAN serial code to parallel programming code.

Acknowledgement: The authors, acknowledge with thanks DSR King Abdulaziz University, Jeddah, Saudi Arabia for technical and financial support.

Funding Statement: This project was funded by the Deanship of Scientific Research (DSR) at King Abdulaziz University, Jeddah, under grant no. (RG-3-611-38).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] P. Czarnul, J. Proficz and K. Drypczewski, "Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems," *Scientific Programming*, vol. 2020, pp. 1058–9244, 2020.

[2] D. B. Changdao, I. Firmansyah and Y. Yamaguchi, "FPGA-based computational fluid dynamics simulation architecture via high-level synthesis design method," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th Int. Symp., ARC 2020*, Toledo, Spain, Springer Nature. vol. 12083, pp. 232, 2020.

[3] D. Wang and F. Yuan, "High-performance computing for earth system modeling," *High Performance Computing for Geospatial Applications*, vol. 23, pp. 175–184, 2020.

[4] K. Jongmin and E. Franco, "RNA nanotechnology in synthetic biology," *Current Opinion in Biotechnology*, vol. 63, pp. 135–141, 2020.

[5] L. Zhenlong, "Geospatial big data handling with high performance computing: current approaches and future directions," *High Performance Computing for Geospatial Applications*, vol. 23, pp. 53–76, 2020.

[6] M. S. Ahmed, M. A. Belarbi, S. Mahmoudi, G. Belalem and P. Manneback, "Multimedia processing using deep learning technologies, high-performance computing cloud resources, and big data volumes," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 17, pp. 56–99, 2020.

[7] N. Melab, J. Gmys, M. Mezmaz and D. Tuyttens, "Many-core branch-and-bound for GPU accelerators and MIC coprocessors," *High-Performance Simulation-Based Optimization*, vol. 833, pp. 275–291, 2020.

[8] R. Kobayashi, N. Fujita, Y. Yamaguchi, A. Nakamichi and T. Boku, "OpenCL-enabled gpu-fpga accelerated computing with inter-fpga communication," in *Proc. of the Int. Conf. on High Performance Computing in Asia-Pacific Region Workshops*, pp. 17–20, 2020.

[9] M. U. Ashraf, F. A. Eassa, A. Ahmad and A. Algarni, "Empirical investigation: Performance and power-consumption based dual-level model for exascale computing systems," *IET Software*, vol. 14, no. 4, pp. 319–327, 2020.

[10] M. U. Ashraf, F. A. Eassa, A. Ahmad and A. Algarni, "Performance and power efficient massive parallel computational model for HPC heterogeneous exascale systems," *IEEE Access*, vol. 6, pp. 23095–23107, 2018.

[11] P. Messina and S. Lee, "Exascale computing project-software," *Los Alamos National Lab.(LANL)*, vol. 366, pp. 17–31, 2017.

[12] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson *et al.,* "DOE advanced scientific computing advisory subcommittee (ASCAC)," in *Report: Top Ten Exascale Research Challenges. USDOE Office of Science (SC) (United States)*, 2014.

[13] B. Brandon, "Message passing interface (mpi)," in *Workshop: High Performance Computing on Stampede*, Cornell University Center for Advanced Computing (CAC), vol. 262, 2015.

[14] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp *et al.,* "An implementation and evaluation of the MPI 3.0 one sided communication interface," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 17, pp. 4385–4404, 2016.

[15] S. Royuela, X. Martorell, E. Quinones and L. M. Pinho, "OpenMP tasking model for ada: Safety and correctness," *Ada-Europe International Conference on Reliable Software Technologies*, vol. 10300, pp. 184–200, 2017.

[16] C. Terboven, J. Hahnfeld, X. Teruel, S. Mateo, A. Duran *et al.,* "Approaches for task affinity in openmp," in *Int. Workshop on OpenMP Springer International Publishing*, Nara, Japan, pp. 102–115, 2020.

[17] A. Podobas and S. Karlsson, "Towards unifying openmp under the task-parallel paradigm," in *Int. Workshop on OpenMP. Springer International Publishing*, Springer International Publishing, Nara, Japan, vol. 9903, pp. 116–129, 2016.

[18] C. U. D. A. Nvidia, "Compute unified device architecture programming guide," 2007. [Online]. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[19] N. Developers, *CUDA Toolkit 11.3 Update*, Jul 2020. [Online]. https://developer.nvidia.com/cuda-downloads.

[20] M. U. Ashraf, F. Fouz and F. A. Eassa, "Empirical analysis of hpc using different programming models," *International Journal of Modern Education & Computer Science*, vol. 8, no. 6, pp. 27–34, 2016.

[21] J. A. Herdman, W. P. Gaudin, S. Smith, M. Boulton, D. A. Beckingsale *et al.,* "Accelerating hydrocodes with openacc, opencl and cuda," *High Performance Computing, Networking, Storage and Analysis (SCC)*, vol. 66, pp. 465–471, 2012.

[22] A. Smith, "CAPS OpenACC compiler the fastest way to manycore programming," 2012. [Online]. Available at: http://www.caps-entreprise.com.

[23] O. Hernandez and R. Graham, *OpenACC accelerator directives*, 2020. [Online]. Available: https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/IntroOpenACC.pdf.

[24] B. Lebacki, M. Wolfe and D. Miles, "The PGI fortran and c99 openacc compilers," in *Cray User Group*, USA, 2012.

[25] D. Bouvier, B. Cohen, W. Fry, S. Godey and M. Mantor, "Kabini: An AMD accelerated processing unit system on a chip," *IEEE Micro*, vol. 34, no. 2, pp. 22–33, 2014.

[26] S. D. K. Intel, "For openCL applications," 2013.

[27] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell *et al.,* "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 2, pp. 173–193, 2011.

[28] A. Lashgar, A. Majidi and A. Baniasadi, "IPMACC: Open source openacc to cuda/opencl translator," *ArXiv Preprint ArXiv*, vol. 14, no. 12, pp. 11–27, 2014.

[29] "OpenCL 1.1 C++ bindings header file," 2012. [Online]. Available at: http://www. khronos.org/registry/cl/api/1.2/cl.hpp.

[30] P. Kulkarni and S. Pathare, "Performance analysis of parallel algorithm over sequential using openmp," *IOSR Journal of Computer Engineering (IOSR-JCE)*, vol. 16, no. 2, pp. 58–62, 2014.

[31] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang *et al.,* "High performance computing using mpi and openmp on multi-core parallel systems," *Parallel Computing*, vol. 37, no. 9, pp. 562–575, 2011.

[32] M. A. K. O. T. O. Ishihara, H. Honda, T. Yuba and M. I. T. S. U. H. I. S. A. Sato, "Interactive parallelizing assistance tool for openmp: Ipat/omp," in *Proc. Fifth European Workshop on OpenMP (EWOMP '03)*, 2003.

[33] A. Athavale, R. Priti and A. Kambale, "Automatic parallelization of sequential codes using s2p tool and benchmarking of the generated parallel code," 2011. [Online]. Available at: http://www. kpit. com/downloads/research-papers/automatic-parallelization-sequential-codes.pdf.

[34] M. Manju and J. P. Abraham, "Automatic code parallelization with openmp task constructs," in *Int. Conf. on Information Science (ICIS). IEEE*, Department of Computer Science & Engineering, College of Engineering Cherthala, pp. 233–238, 2016.

[35] A. Raghesh, "A framework for automatic OpenMP code generation," M. Tech thesis, Indian Institute of Technology, Madras, India, 2011.

[36] R. Reyes, A. J. Dorta, F. Almeida and F. Sande, "Automatic hybrid MPI+ OpenMP code generation with llc," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, Berlin, Heidelberg: Springer, vol. 5759, pp. 185–195, 2009.

[37] K. Hamidouche, F. Joel and E. Daniel, "A framework for an automatic hybrid MPI+ OpenMP code generation," *SpringSim (hpc)*, pp. 48–55, 2011.

[38] M. Marangoni and T. Wischgoll, "Togpu: Automatic source transformation from c++ to cuda using clang/llvm," *Electronic Imaging*, vol. 1, pp. 1–9, 2016.

[39] X. Xie, B. Chen, L. Zou, Y. Liu, W. Le *et al.,* "Automatic loop summarization via path dependency analysis," *IEEE Transactions on Software Engineering*, vol. 45, no. 6, pp. 537–557, 2017.

[40] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, R. Keryell *et al.,* "SESAM/Par4All: A tool for joint exploration of MPSoC architectures and dynamic dataflow code generation," *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, vol. 12, pp. 9–16, 2012.

[41] N. Ventroux, T. Sassolas, R. David, A. Guerre and C. Bechara, "SESAM extension for fast mpsoc architectural exploration and dynamic streaming application," in *IEEE/IFIP International Conference on VLSI and System-on-Chip*, Madrid, Spain, pp. 341–346, 2010.

[42] HPC Project, "Par4All, automatic parallelization," 2020. [Online]. Available at: http://www.par4all.org.

[43] H. Shen, P. Gerin and F. Petrot, "Configurable heterogeneous mpsoc architecture exploration using abstraction levels," in *IEEE/IFIP Int. Symp. on Rapid System Prototyping*, Paris, France, pp. 51–57, 2009.

[44] C. Augonnet, S. Thibault, R. Namyst and P. A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[45] S. Tomov, J. Dongarra and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5, pp. 232–240, 2009.

[46] J. Planas, R. M. Badia, E. Ayguadé and J. Labarta, "Self-adaptive OmpSs tasks in heterogeneous environments," in *IEEE 27th Int. Symp. on Parallel and Distributed Processing*, pp. 138–149, 2013.

[47] P. M. Josep, R. M. Badia and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *IEEE Int. Conf. on Cluster Computing*, Tsukuba, pp. 142–151, 2008.

[48] A. Elena, D. Rossetti and S. Potluri, "Offloading communication control logic in GPU accelerated applications," in *Proc. of the 17th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, IEEE Press, Madrid, Spain, pp. 248–257, 2017.

[49] A. Mashat, *Fujitsu High-Performance Computing Case Study King Abdulaziz University*, 2020. [Online]. Available: https://www.fujitsu.com/global/Images/CS_2015Jul_King%20Abdulaziz_University.pdf.

[50] Aziz supercomputer King Abdulaziz University, *Top 500 the list*, 2020. [Online]. Available at: https://www.top500.org/site/50585.

[51] Aziz - Fujitsu PRIMERGY CX400 Intel Xeon, *Top 500 the list*, 2020. [Online]. Available at: https://www.top500.org/system/178571.

[52] L. J. David, *Measuring computer performance: A practitioner's guide*. Cambridge: Cambridge University Press, 2005.

[53] N. A. Cristobal, N. Hitschfeld-Kahler and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using GPU architectures," *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014.

[54] L. A. Barroso, "The price of performance," *Queue*, vol. 3, no. 7, pp. 48–53, 2005.

[55] R. DaQi and R. Suda, "Power efficient large matrices multiplication by load scheduling on multi-core and gpu platform with cuda," in *Int. Conf. on Computational Science and Engineering*, Vancouver Canada, vol. 1, pp. 424–429, 2009.

[56] M. Moeller, *Open Hardware Monitor*, 2020. [Online]. Available at: http://openhardwaremonitor.org.

[57] Tech PowerUp, *TechPowerUp GPU-Z*, 2020. [Online]. Available at: https://www.techpowerup.com/download/techpowerup-gpu-z/.