Parallelization Technique using Hybrid Programming Model

Abdullah Algarni[1], Abdulraheem Alofi[2], Fathy Eassa[3]
Department of Computer Science, King Abdulaziz University (KAU)
P.O. Box 80221, Jeddah 21589, Saudi Arabia

***Abstract*—A multi-core processor is an integrated circuit that contains multiple core processing unit. For more than two decades, the single-core processors dominated the computing environment. The continuous development of hardware and processors led to the emergence of high-performance computers that able to address complex scientific and engineering programs quickly. Besides, running the software codes sequentially increases the execution time in huge and complex programs. The serial code is converted to parallel code to improve the program performances and reduce the execution time. Therefore, parallelization helps programmers solve computing problems efficiently. This study introduced a novel automatic translation tool that converts serial C++ code into a hybrid parallel code. The study analyzed the performance of the proposed S2PMOACC tool using linear algebraic dense matrix multiplication benchmarking. Besides, we introduced Message Passing Interface (MPI) + Open Accelerator (OpenACC) as a hybrid programming model without preliminary knowledge of parallel programming models and dependency analysis of their source code. The research outcomes enhance the program performances and decrease the implementation time. Moreover, our proposed technique offers better performance than other tools.**

***Keywords*—Serial code translation; parallel code; C++; hybrid programming model; auto-translation; S2PMOACC**

I. Introduction

A single-core microprocessor dominated the computing environment for more than two decades as it offered better performance in execution of computer programs. With a rise in issues, such as power dissipation, design complexity, and high energy consumption [1] in the single-core, multicore architectures were proposed to address these problems. The multicore architectures [2] opened a new door for high-performance computing, dividing each task into different cores during execution. Also, the multi-core architecture plays a crucial role in developing parallel applications. Therefore, many industries build their programs using parallel computing architectures [3]. Besides, complex scientific programs require a huge computing power [4], which individual computer fails to provide. Therefore, programmers must write parallel programs to be running in multicore architectures. Also, parallel programming is computationally complex and requires different execution effort.

Nowadays, with the advent of computer technology, people rely heavily on computer systems to conduct all business-related tasks. A standard desktop computer or workstation can easily solve small computing problems, but it provides poor performance and runs into technical problems while performing a high number of operations per second. Besides, a standard computer faces challenges in completing a time-consuming operation in less time, completing operations under a tight deadline, and solving complex large problems. High-Performance Computing (HPC) eliminates these problems. Compared to a traditional desktop computer, HPC systems can perform complex calculations and process data at high speed. Fig. 1 shows HPC systems with CPUs. Each of the CPUs processors has local memory and multicores, which help to execute different applications and challenging tasks. Besides, HPC solves extensive problems via thousands of parallel processors.

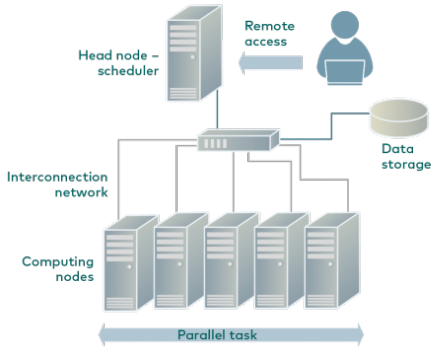

Fig. 1. An Overview of HPC System.

The HPC offers significant benefits to different sectors such as education, health, engineering, government, and business owing to its ability to solve complex and demanding problems. Additionally, HPC is an essential driver of innovation and fosters economic growth. Besides, HPC facilitates R&D in science and technology and enhances new products time-to-market. HPC also helps researchers to solve complex problems, such as developing new drugs through advanced computer simulation. Parallel computing is a computation that breaks larger computing problems into smaller tasks, in which many calculations are executed concurrently. Massive problems are divided into smaller units in order to enhance the overall performance of HPC systems. Besides, the development of future Exascale machines can become complex, which requires writing parallel programs [5], [6]. Parallel programming [3] is a multi-threaded or multi-processes mechanism used to write and run the parallel programs on the HPC. There are many existing parallel programming languages like OpenMP (Open Multi-Processing), OpenACC (Open accelerators) and MPI (Message Passing Interface).

Creating parallel programs manually is a hard job and leads to consuming time and so may not free from human mistakes [6]. Thus [7], programmers are increasingly using automatic parallelization tools owing to their ability to automatically translate serial code into parallel code, thereby saving programming time and costs. Many automatic parallelization tools we will discuss in the related works section, these tools are available to convert sequential codes to parallel codes in order to minimize programming errors, and offer accurate results [7]. Besides, these tools take different inputs and combine different programming models. Combining more than one model is crucial in achieving optimal performance through parallelizing programs [8]. Therefore, the hybrid MPI+OpenMP [5] model is perfect for parallel computing,because it is combine between shared and distributed memory. Clearly, each tool only suitable for specific parallel model [9], and no tool is good enough for all applications .

The current study highlighted there is no parallelization tool exists on the cluster/hybrid system that converts the serial code into parallel. In order to leading towards the objective, we develop a hybrid MPI-OpenACC tool able to translate sequence C++ codes to parallel codes, implemented by combining the MPI library with OpenACC directives. Fig. 2 demonstrates how the hybrid model works [10]. The hybridization increases performance [11], parallelism, and adapts to different environments.

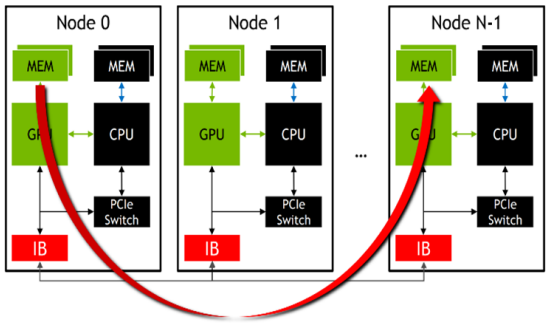

Fig. 2. Processing Mechanism of Hybrid MPI and OpenACC.

The study provides the following contributions: The study proposes a new automatic translation tool that converts serial C++ programming code into hybrid MPI+OpenACC parallel code. Besides, we propose an algorithm and theoretical architecture to enhance performance and decrease implementation run-time. We implement many applications using the proposed solution and compute the results on the different Graphics Processing Units (GPU) devices. Furthermore, the performance and features of the proposed model are compared with existing automatic tools. Based on experimental results, our proposed technique outperforms other models.

The rest of the paper is organized as follows. In Section 2, we discuss the detailed background of parallel computing models used in our proposed solution and then, Section 3 describes the related works of parallel programming models based on different hierarchical machines. Section 4, the system model has been described in detail with the architecture, and algorithm of the proposed parallelization technique. Section 5, discusses the experimental platform and the measuring factors for evaluating the proposed technique. Section 6, provides results and discussion. Finally, the conclusion in Section 7 followed by future work in Section 8.

II. Background

The rapid development of hardware and processors led to the emergence of parallel computing, which can address complex scientific and engineering programs quickly and efficiently. In parallel computing, the program is broken into several parts to solve computation problems concurrently. Each part is further broken into a set of instructions to be executed simultaneously on different processors. The primary benefit of parallel computing is suitability for modeling and simulation. Besides, parallel computing saves time and produces useful results for researchers. Single-core processors are unsuitable to solve many scientific problems. However, multi-core processors can solve these problems as they contain GPU. As shown in Fig. 3, the MPI parallel programming model standard was launched in 1994 for the application of distributed-memory communication.

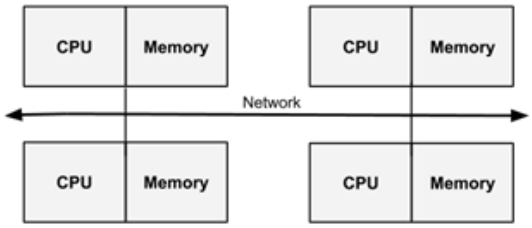

Fig. 3. MPI Distributed Memory.

MPI [12] is an efficient standard programming model applicable on distributed computing systems for many years, MPI is implemented on parallel machines and provides good performance and portability. MPI implementations are designed to work on different parallel environments and support classical communications [13]. MPI provides an explicit method for the message passing a programming technique on distributed memory clusters. Besides, distributed resources are spread over several computing nodes, and in MPI, synchronization is handled explicitly due to the distributed memory [14]. Many attributes in MPI such as portability, where it is has a ability to integrate with other programming models. Also, it is available for many implementations, such as open-source implementations like MPICH [15], and OpenMPI [16], and commercial implementations such as Intel MPI library [17], and IBM Spectrum MPI [18]. Further, functionality [11], where the MPI library has more than 400 routines. MPI programs have special structures and listings. First, demonstrating the basic commands, starting from the MPI header file. Then, initialize the MPI environment using MPI_Init() instruction. The next stage is by defining the rank and the size of processes using MPI_Comm_rank and MPI_Comm_size consecutively. In another stage, inserting MPI calling routine code and run parallelly. Finally using MPI_Finalize() to terminate the MPI execution [19]. With the advancement in GPU technologies, accelerators have been developed for GPU programming. Each accelerator follows a unique programming technique. For example, Compute Unified Device Architecture (CUDA) for

NVIDIA GPUs, Brook+ for AMD GPUs, and LEO for MICs, etc. [20]. NVIDIA released CUDA in 2007. In November 2011, OpenACC introduced as directive-based programming model designed for targeting heterogeneous CPU/GPU systems. OpenACC [11] has features to overcome the limitation of the CUDA model. CUDA works on NVIDIA GPU only whereas OpenACC works with many compilers. Besides, OpenACC offers excellent performance and accelerates scientific applications with little programming efforts [21]. The programmer only should insert directives in a suitable place to run the code on the GPU compiler [7]. Fig. 4 shows the OpenACC accelerator model, revealing how intensive computations are offloading from a host device to the GPU device to accelerate the execution [22].

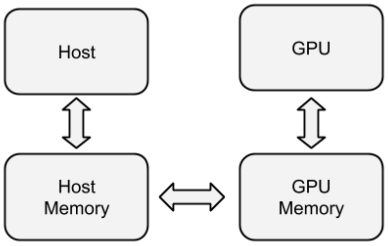

Fig. 4. OpenACC Accelerator.

Accelerating the program through OpenACC requires analyzing the source code to determine the part that requires more execution time. As illustrated in Listing 1, inserting the directives in a suitable place facilitates the code execution on the GPU compiler.

Listing 1: OpenACC directives

```
main()
{
  <serial code>
  #pragma acc kernels
  //automaticly runs on GPU
  {
        <parallel code>
  }
}
```

III. Related Works

In this section, we have discussed the detailed hybrid programming models and auto parallelization tools to convert serial code to parallel. The programming model combined with one or more models can increase the performance of parallelism and the capability to work with the heterogeneous systems. This combination [23] facilitates the application of large-scale powerful programming models. Fig. 5 depicts the hierarchy navigation programming model and is categorized as follows:

- Single model: MPI
- Dual model: MPI+X
- Tri model: MPI+X+Y

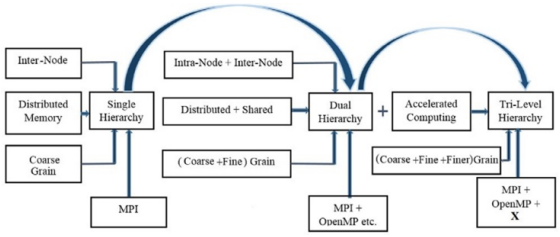

Fig. 5. Hierarchy Navigation in the Programming Model.

1) Single model: MPI
MPI is a standard message passing library for exchanging data between different nodes. MPI facilitates program execution among distributed memory and is an effective mechanism to parallelize the application [12]. Besides, it is a coarse-grained technique to execute and manage data on the level of the node [24]. The MPI version 3.1 introduces new features and capabilities to facilitate the parallelization process like creating group quires and processes [13], [12]. The HPC environments help to share data across different distributed nodes using the MPI library. Therefore, MPI programmers must understand future hardware development for effective compatibility as MPI provides an excellent model for future disparate systems.

2) Dual model: MPI+OpenMP
Previous studies [25], [26] introduced a common hybrid model MPI + OpenMP, using MPI for communication between nodes. Besides, OpenMP is used for the parallelization process inside the node. Fig. 6 shows the processing mechanism of hybrid MPI and OpenMP. Data are shared over several nodes that communicate with each other through the MPI message passing technique. The OpenMP region is designed for distributed data, assisting in deciding the available number of threads. This hybrid approach is one of the promising models for future HPC applications [27].

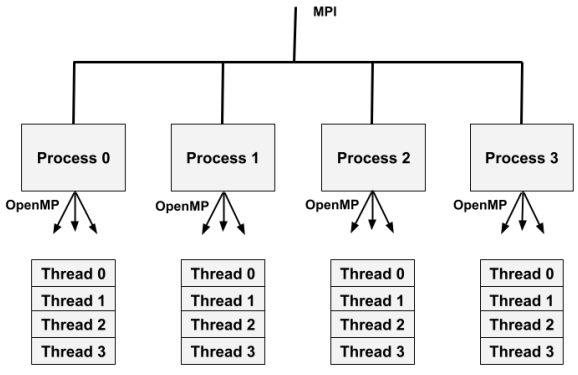

Fig. 6. Hybrid MPI and OpenMP Processing Mechanism.

3) Dual model: MPI+OpenACC

To use parallel programming, Hybrid MPI and OpenACC are an alternative model for hybrid MPI and CUDA mentioned in [28]. Similar to OpenMP, data parallelization in OpenACC is based on directives. The programmers must write their code, interchangeable to traditional serial code written in C/C++, and include the #pragma acc loop directive line before the loop statements. The program written in these directives is computed with accelerated GPU devices. In theory, the mixed mode architecture code should be more efficient more than a pure MPI model due to combination between shared and distributed memory. Listing 2 describes a briefing algorithm as a hybrid of MPI + OpenACC is the simplest way of parallelism [27].

Listing 2: MPI+OpenACC Processing

```
MPI_Init() //Initialize MPI
//Processes size
Size <-- Get MPI_Comm_size()
//Processes ranks
Rank <-- Get MPI_Comm_rank()
if (Rank==0) //Master Process
/* Make processing
 before Entering MPI comm world */
//Send data when rank > 0
MPI_Isend()
//Check processing periodically
MPI_Wait()
if(Rank>0)
/* receive data from
all processes rank > 0 */
MPI_Irecv()
 #pragma acc data copy(a) copyin(b)
 #pragma acc kernels
 {
   While(loop_statements 1 to N)
   #pragma acc loop
   loop statement 1
 }
MPI_Isend() //send data again
if(Rank==0) //collect data
MPI_Irecv() //receive final data
MPI_finalize() //finalize MPI
```

4) Tri model: MPI + OpenMP + CUDA (MOC)
In 2018, a group of developers introduced the Tri-Hierarchy hybrid MOC model [23], comprising of MPI + OpenMP + CUDA to achieve enormous parallelism objectives. MPI helps to broadcast data on the distributed node. OpenMP executes data on CPU threads; whereas, CUDA executes data on accelerated GPU cores [29]. Besides, the MOC model facilitates performance via inflexible parallelism. We develop a similar HPC application using a huge cluster system with multiple nodes and GPU. MOC model offers a coarse and efficient massive parallelism.

As discussed previously, writing parallel code manually is tedious and time-consuming. The auto parallelization tools help to overcome these problems. Many tools can convert sequential codes to parallel codes and add the parallelization constructs or directives in a suitable place [9].

One study [30] provides a concise survey of existing parallel tools and classifies these tools based on different criteria like a history of tools, tools contributions, and support assisting for parallelization. A new tool called **EasyPar** proposed in the study and this tool capable to assisting and facilitating the program's development phase. The authors in [31] analyze the performance of two tool(Cetus and Par4All). The **Cetus** is a source-to-source transformation tool using OpenMP directives to convert 'C' sequential code to parallel codes. However, the **Par4All** is an open-source tool to convert sequential code to new OpenMP, CUDA, and OpenCL source codes. The study performed on two complex program [31] to find which the best performance between tools. The results showed the tools are effective for single loop programs but not for the nested loops. A study [32] proposes a new parallelization model called **PyParallelize**, which automates the parallel process by reading source code without modifications from the programmer. After implementing the model on different benchmark programs, the results showed a relatively high rate of accuracy. However, this model fails to work efficiently when nested loops are more than two in source code. Another study [4] presents a new converting tool called **S2P** (c serial to Parallel) to perform parallelization on different programs. Furthermore, they [4] compare the tool performance with other existing tools. The **S2P** tool offers better results in some cases and it assists in minimizing the overhead thread management during the execution of parallel code. Authors in [9] proposed an automatic code parallelization tool, which converted C serial code to the equivalent version of parallel using OpenMP parallel programming. However, this tool focuses only on parallelism tasks without considering loops parallelism.

A new model architecture suggested in the study [33], and that model can translate any serial application into parallel code, using individual parallel programming likes OpenMP, MPI, OpenCL, and OpenACC or hybrid OpenMP-MPI. This model is under development with a promise to solve automatic parallelization issues. Besides, a study [7] proposed a tool to speed up multicore processors. The proposed tool achieves 4.27 speedup after using 4 cores and 8 threads when increasing the length of the matrix.

We conclude from the previous literature, there is no tool provides full optimal translation. Thus, research is required to address the limitations of these tools. Tools such as SUIF, CAPO, and Polaris fail to support all operating systems. Also, these tools only used C and FORTRAN languages [30]. Besides, it is difficult to run the generated code on distributed memory because the tools such as Par4All and S2P use the OpenMP programming model, which works exclusively on shared memory. The parallelization in nested loops are an advantage of the tool. Therefore it is a limitation in the tools like SUIF and Intel compilers [32]. In the next section, we will discuss in detail the methodology, revealing the proposed S2PMOACC tool.

IV. Methodology

This study introduces a new automatic translation tool to reduce the software code's execution time sequentially by massively improving the performance of huge and complex programs. We use all the available resources to convert serial code to parallel code to enhance the program performances and reduce the execution time. We proposed a new automatic translation tool to convert serial C++ code into parallel code. Fig. 7 illustrates the proposed translation tool S2PMOACC (Serial To Parallel MPI and OpenACC) architecture, taking serial code as input and generating its parallel code automatically. The proposed solution enhances program performance and decreases implementation time.

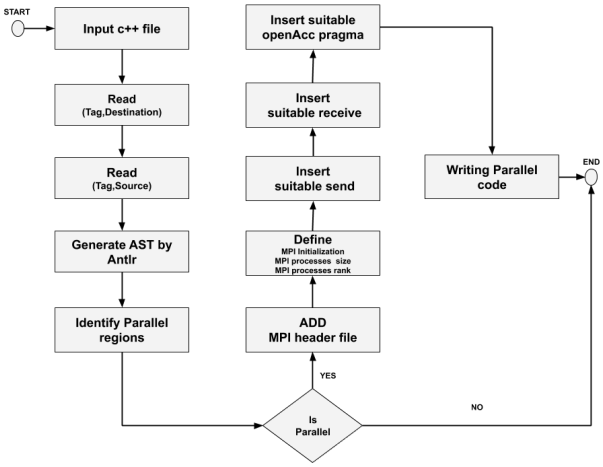

Fig. 7. Architecture of Proposed Automated Translation Tool S2PMOACC.

The steps of the translation tool are elaborated as follows:

1) Input C++ file

   This is the first phase where the source file enters the command line using the CLC library. The C++ source file must be written without any syntax error. Then, the file name is moved to the next steps for further processing.

2) Read (tag.Destination)

   This step reads input from the user and enters the information related to the MPI_Send call. Using tag in the MPI to distinguish between different processes and the destination is an INT number for process rank where every process has a unique rank in the MPI environment. This information should be provided by the user to write statements when generating the parallel code.

3) Read (tag.Source)

   As the previous step, the user enters information of tag and source related to MPI_Recv call. Source is an INT number to determines the rank of the source process. To write a suitable receive statement, the information must be entered correctly.

4) Generate AST by ANTLR

   ANTLR tool will read the file contents and generate the AST to be used in the following steps. As shown in Fig. 8, many procedures in ANTLR are revealed. First, reading a file's contents by passing the name to a specific routine to handle the process. Then, the contents streaming fed to lexer that contains lexer grammars to identify and produce different token streams. Finally, the token streams are entered into the parser to generate AST based on pre-defined parser rules.

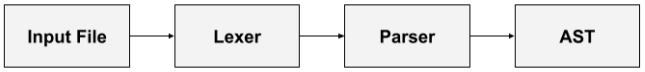

Fig. 8. ANTLR Tool Phases.

5) Identify parallel regions

   In this step, the generated AST is used to identify the parallel regions of the code using inherited class to access and identify different loops and variables.

6) Determine the dependency

   In this step, the dependency test occur to determine the possibility of parallelization. Special java class used to find out parallel regions like loops. The dependency test used to: accessing loop, fetch statements in the loop body, and decide if dependency inside the body block existing or not. For example if **S1** and **S2** are statements inside the loop, if **S2** depend on **S1** result, then the dependency is detected. Therefore, the target file is generated without parallel directives. Otherwise, the dependency not found and parallel file generated.

7) Add MPI header file

   Here, include MPI header file library 'mpi.h' in order to use MPI routines when generating parallel code.

8) Define MPI instructions

   Initialize MPI environment through using MPI_Init() instruction. Then define Ranks for MPI to check cluster system specification for defining the ranks via MPI_Comm_rank that provide a logical way of numbering the MPI process. Further, define MPI communication world statements via MPI_Comm_world in order to run all MPI jobs across several processes. Ranks in this step used to find out master and slave processes in MPI environment.

9) Insert suitable MPI_ Send statement

   In order to write a suitable MPI send statement in the parallel file the information get it from a user in the previous steps will be used. The statement will be insert based on a pre-defined indicator from the user. Also, the information should be correct to avoid mistakes in writing process.

10) Insert suitable MPI_ recv statement

The information entered from the user is used in this step. Besides, the pre-defined indicator is used to identify the place to insert receive MPI operation.

11) Insert suitable OpenACC directive

Insert OpenACC directives based on the parallelization annotations. This step involves inserting an OpenACC directive to the source code to notify the OpenACC-compiler to parallelize the objects of the classes, run in parallel during run-time.

12) Writing Parallel code

Write parallel code to generate the code, which includes the MPI routines and OpenACC parallel computing pragmas via #pragma acc atomic and #pragma acc parallel loop.

Regarding the proposed automated translation, we discuss a comprehensive overview of how C++, MPI, and OpenACC are translated from serial code to dual auto parallelization using a hybrid programming model. Algorithm 1 presents serial code to dual auto parallelization.

Algorithm 1 : Serial Code to Dual Auto Parallelization

1: Input C++ file.
2: Read (Tag, Destination) information.
3: Read (Tag, Source) information.
4: Generate AST by ANTLR.
5: Identify parallel regions(for-while-do while)
6: Perform dependency analysis for each region.
7: Determine the possibility of parallelization.
8: Add MPI header file 'mpi.h'.
9: Initialize MPI environments 'MPI_Init()'
10: Define MPI communication world statements
11: Define MPI communications ranks
12: Insert suitable MPI_Send().
13: Insert suitable MPI_Recv().
14: Insert OpenACC directives based on the parallelization annotations.
15: Writing parallel code.
16: Save the output file in the directory.

The next sections will shows the experimental platform and measuring factors followed by results and discussion for testing the proposed tool and illustrates the capabilities and limitations of our tool compared with other existing tools.

V. Experimental Platform and Measuring Factors

This section demonstrates the experimental platform to analyze the performance of the proposed solution. We quantified different measures, taking HPC benchmarks as performance metrics of the execution time (Secs) and system speedup (Serial/Parallel) to compare the proposed tool and other tools. We perform all the experiments on Intel i7 with four cores and eight threads. Table I shows the testing environment specifications. The HPC system and applications are evaluated based on fundamental performance metrics. We measured different performance attributes including execution time (Secs) and speedup (Serial/Parallel) of the system.

TABLE I. Environment Specifications

Feature	value
Processor Type	Intel(R) Core(TM) i7-1065G7
Number of cores	4
Number of threads	8
Operating Systems	Windows 10
Clock speed	1.30 GHz
Graphic card	NVIDIA GeForce MX250
RAM	16 GB

To evaluate the performance attributes, we select a dense matrix multiplication (DMM) with different sizes and computed on the different numbers of MPI processes and GPUs devices. We evaluate the speed up performance metric where matrix multiplication is computed without using a single number of GPU core to determine the speed trend. Without matrix multiplication, we run speed on the experimental setup and calculate the time taken in sequential processing. Ideally, the speedup is calculated by following the fundamental Amdahl's law [34] and Gustafson [35].

$$SpeedUp(S_P) = \frac{T_{serial}}{T_{parallel}} \quad (1)$$

Where T_{serial} is the optimal time in sequential processing and $T_{parallel}$ for parallel computing algorithms. According to (1), we can calculated speedup based on execution time by implementing proposed solution on DMM benchmarking application against varying dataset. The possible curve of speedup [36] in an algorithm could be super-linear, perfect linear, linear and sub-linear as demonstrate in Fig. 9.

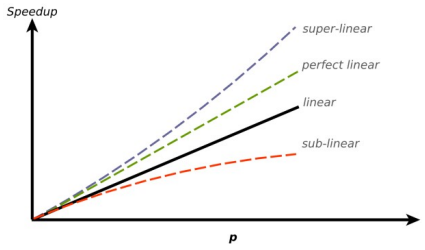

Fig. 9. Speedup Possible Curves.

VI. Results and Discussion

We evaluate the proposed parallelization technique using a dual hierarchical hybrid parallel programming model via the implementation of linear algebraic dense matrix multiplication. All results have been executed using the quad-core processor. Besides, we quantified various matrix sizes. We also examine linear DMM application performance in different datasets, assisting in determining the execution times in secs. Besides, four well-known automatic parallelization tools S2PMOACC, Cetus, Par4all, and S2P were included in the study. The speedup of studied tools was calculated to determine the efficiency of each tool.

In the 1st experiment, we run a serial matrix multiplication by different sizes. Then, we run same matrix using hybrid parallel code executing in different number of MPI processes to evaluate the execution time between serial and hybrid version. We observed the increasing in execution time in serial version unlike parallel codes. Fig. 10, Fig. 11, and Fig. 12 shows the obtained results.

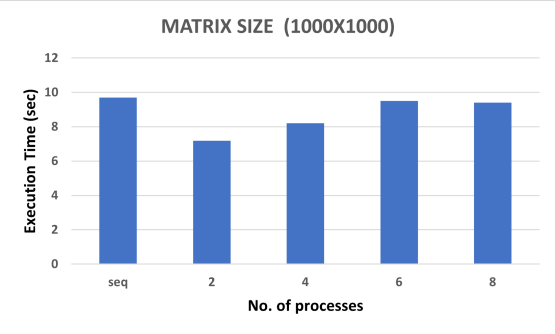

Fig. 10. Execution Time of Matrix 1000x1000.

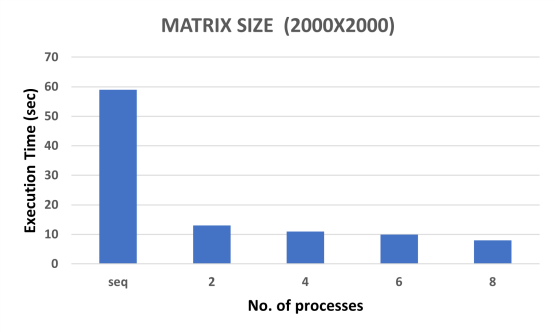

Fig. 11. Execution Time of Matrix 2000x2000.

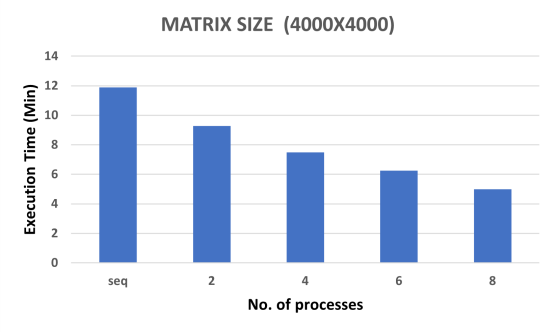

Fig. 12. Execution Time of Matrix 4000x4000.

In the 2nd experiment with a minute amount of matrix multiplication by 1000 x 1000 matrix size, we experiment this dataset with single GPU by running the existing tools along with our proposed solution. S2PMOACC take 8 secs to complete the execution while Cetus computes in 13 secs, Par4All end its execution in 22 secs, and S2P calculated executed time is 24 secs. From Fig. 13, the results show that our proposed model execution time is negligible in minimum resources. Besides, we increase the GPU cores and analyze the same translation tools with an equal number of given devices.

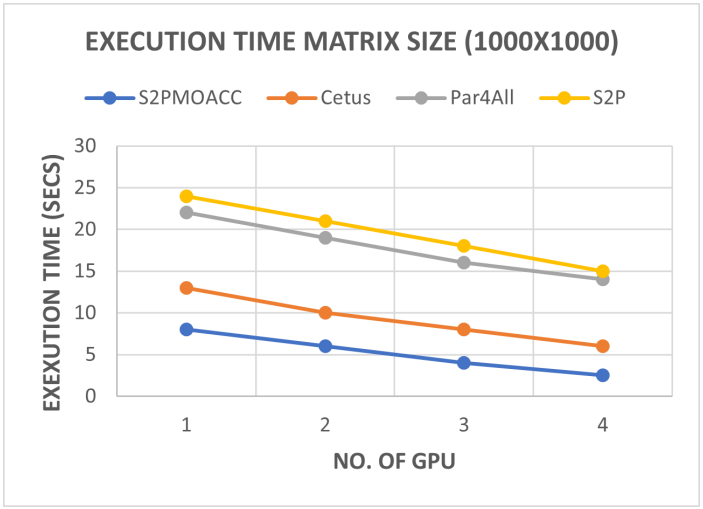

Fig. 13. Performance (Execution Time) in DMM.

According to Fig. 13, we measure second performance matric as speedup in experiment 2 that involves the optimal time taken in serial to parallel processing for all the implementation run with an equal number of resources. Fig. 14 shows the speedup of our tool along with other tools when Using 1000 x 1000 matrix multiplication with different number of GPU.

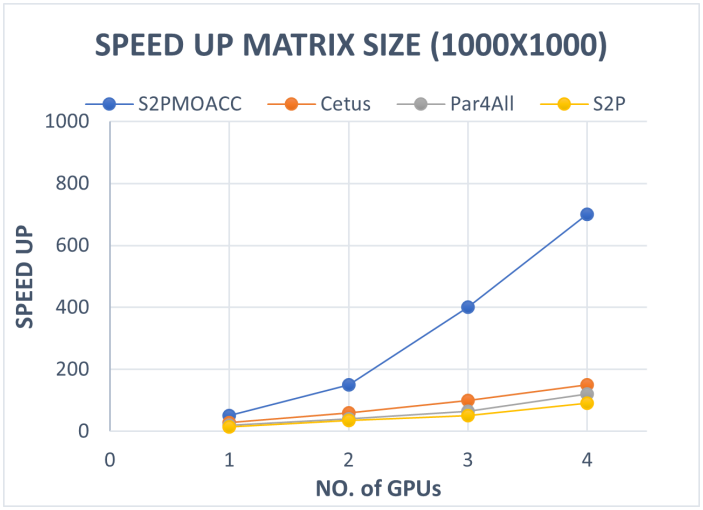

Fig. 14. Performance (Speedup) in DMM.

Finally, we study the capabilities and features of the included tools. We observed that our tool has more features than other tools. Table II provide the summary of the comparison.

TABLE II. TOOLS SPECIFICATIONS COMPARISONS.

Feature	Tools			
	S2PMOACC	**Cetus**	**Par4all**	**S2P**
Operating Systems	Windows Linux	Linux	Linux	Linux
Input Language	C,C++	C	Fortran,C	C
Technique used	MPI OpenACC	OpenMP	OpenMP CUDA	OpenMP Pthread
Hybrid Output	Yes	NO	NO	NO
Support For loop	Yes	Yes	Yes	Yes
Support While	Yes	NO	NO	NO
Support Do_while	Yes	NO	NO	NO

VII. CONCLUSION

The application of HPC has increased significantly in all scientific fields and HPC systems has been used to solve complex computational programs. Despite running software programs sequentially, researchers and programmers face difficulties in dealing with huge and complex programs, which increase the execution time. The serial code must be converted to parallel code to improve the program performance and reduce the execution time. Therefore, parallelization tools must assist programmers in the converting process. In this work, we proposed a novel automatic translation tool that converts serial C++ code into parallel code using a hybrid parallel programming model. This auto-translation tool supports a dual hierarchical MPI and OpenACC parallel computing model for heterogeneous systems that use GPU devices for providing parallelism. We implement the proposed solution in the DMM application by using different number of MPI processes in the first experiment. The second experiment compare the proposed tool execution time and speedup with well-known auto-translation tools such as Cetus, Par4all, and S2P tools. The Third experiment compare the features and limitations between tools. Based on the experimental results, the S2PMOACC outperformed the existing tools and provides complete auto parallelism in all performance metrics.

VIII. FUTURE WORK

In the near future ,the auto-parallel computing systems are in high demand as they support ECS applications. Therefore, we must have an adaptive auto-translation technique for parallelizing sequential code to support the future Exascale-computing system, large-scale cluster system, multi-core distributed system, and heterogeneous cluster system. For that, we aim to implement more enhancement to our proposed tool to keep pace with the continuous development of future systems.

REFERENCES

[1] J. Diaz, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[2] A. Roy, J. Xu, and M. H. Chowdhury, "Multi-core processors: A new way forward and challenges," in *2008 International Conference on Microelectronics*, 2008, pp. 454–457.

[3] 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)*, 2017, pp. 1–6.

[4] A. Athavale, P. Ranadive, M. Babu, P. Pawar, S. Sah, V. Vaidya, and C. Rajguru, "Automatic sequential to parallel code conversion," *GSTF Journal on Computing (JoC)*, vol. 1, no. 4, 2014.

[5] D. Akhmetova, R. Iakymchuk, O. Ekeberg, and E. Laure, "Performance study of multithreaded mpi and openmp tasking in a large scientific code," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 756–765.

[6] K. R. Varsha, "Automatic parallelization tools : A review," *International Journal of Engineering Science and Computing IJESC*, vol. 7, no. 3, pp. 5780–5784, 2017.

[7] A. Barve, S. Khandelwal, N. Khan, S. Keshatiwar, and S. Botre, "Serial to parallel code converter tools: A review," in *International Journal of Research in Advent Technology (E-ISSN: 2321-9637) Special Issue National Conference" NCPCI-2016*, vol. 19, 2016.

[8] A. M. Alghamdi, F. E. Eassa, M. A. Khamakhem, A. S. A. AL-Ghamdi, A. S. Alfakeeh, A. S. Alshahrani, and A. A. Alarood, "Parallel hybrid testing techniques for the dual-programming models-based programs," *Symmetry*, vol. 12, no. 9, p. 1555, 2020.

[9] M. Mathews and J. P. Abraham, "Automatic code parallelization with openmp task constructs," in *2016 International Conference on Information Science (ICIS)*, 2016, pp. 233–238.

[10] J. Etancelin and J. Kraus, "Multi-GPU programming with OpenACC and MPI," in *GPU Technology Conference*, 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01471165

[11] A. M. Alghamdi and F. E. Eassa, "Parallel hybrid testing tool for applications developed by using mpi+ openacc dual-programming model," *Adv. Sci., Technol. Eng. Syst. J.*, vol. 4, no. 2, pp. 203–210, 2019.

[12] "Message passing interface forum." [Online]. Available: https://www.mpi-forum.org/

[13] P. Balaji and W. Gropp, "Advanced mpi programming," 2016. [Online]. Available: https://www.mcs.anl.gov/~thakur/sc16-mpi-tutorial/slides.pdf

[14] C.-T. Yang, C.-L. Huang, and C.-F. Lin, "Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters," *Computer Physics Communications*, vol. 182, no. 1, pp. 266–269, 2011.

[15] "Mpich implementation." [Online]. Available: https://www.mpich.org/

[16] "Open mpi: Open source high performance computing." [Online]. Available: https://www.open-mpi.org/

[17] "Intel mpi library." [Online]. Available: https://software.intel.com/

[18] "Ibm spectrum mpi - overview — ibm." [Online]. Available: https://www.ibm.com/products/spectrum-mpi

[19] A. S. A. Alghamdi, A. M. Alghamdi, F. E. Eassa, and M. A. Khemakhem, "Acc_test: Hybrid testing techniques for mpi-based programs," *IEEE Access*, vol. 8, pp. 91 488–91 500, 2020.

[20] J. Kim, S. Lee, and J. S. Vetter, "Impacc: a tightly integrated mpi+ openacc framework exploiting shared memory parallelism," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 189–201.

[21] R. Farber, *Parallel programming with OpenACC*. Newnes, 2016.

[22] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 2017.

[23] M. U. Ashraf, F. A. Eassa, and A. A. Albeshri, "Efficient execution of smart city's assets through a massive parallel computational model," in *International Conference on Smart Cities, Infrastructure, Technologies and Applications*. Springer, 2017, pp. 44–51.

[24] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "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.

[25] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using mpi and openmp on multi-

core parallel systems," *Parallel Computing*, vol. 37, no. 9, pp. 562–575, 2011.

[26] D. Akhmetova, R. Iakymchuk, O. Ekeberg, and E. Laure, "Performance study of multithreaded mpi and openmp tasking in a large scientific code," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 756–765.

[27] T. Katagiri, "Basics of mpi programming," in *The Art of High Performance Computing for Computational Science, Vol. 1*. Springer, 2019, pp. 27–44.

[28] D. Jacobsen, J. Thibault, and I. Senocak, "An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters," in *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2010, p. 522.

[29] F. Bonelli, M. Tuttafesta, G. Colonna, L. Cutrone, and G. Pascazio, "An mpi-cuda approach for hypersonic flows with detailed state-to-state air kinetics using a gpu cluster," *Computer Physics Communications*, vol. 219, pp. 178–195, 2017.

[30] S. Sah and V. G. Vaidya, "A review of parallelization tools and introduction to easypar," *International Journal of Computer Applications*, vol. 56, no. 12, 2012.

[31] S. Prema and R. Jehadeesan, "Analysis of parallelization techniques and tools," *International Journal of Information and Computation Technology*, vol. 3, no. 5, pp. 471–478, 2013.

[32] A. J. Almghawish, A. M. Abdalla, and A. B. Marzouq, "An automatic parallelizing model for sequential code using python," *International Journal*, vol. 7, no. 3, 2017.

[33] K. Alsubhi, F. Alsolami, A. Algarni, K. Jambi, F. Eassa, and M. Khemakhem, "An architecture for translating sequential code to parallel," in *Proceedings of the 2nd International Conference on Information System and Data Mining*, 2018, pp. 88–92.

[34] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[35] S. Ristov, R. Prodan, M. Gusev, and K. Skala, "Superlinear speedup in hpc systems: Why and when?" in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2016, pp. 889–898.

[36] C. A. Navarro, 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.